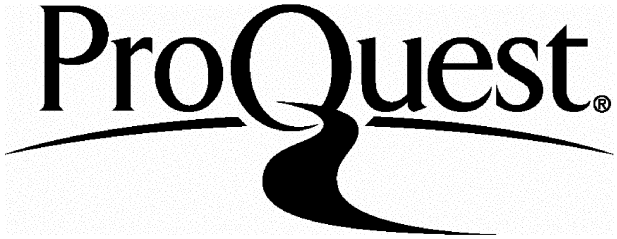
UNIVERSITY COLLEGE LONDON
DEPARTMENT OF COMPUTER SCIENCE

Extending the Kohonen Self-Organising Map by Use of Adaptive Parameters and Temporal Neurons

David A. Critchley

A thesis submitted for the degree of
Doctor of Philosophy
in the University of London

February 1994



Acknowledgements

No PhD thesis gets written without a lot of backing. This one is no exception:

A huge vote of thanks is due to Denise Gorse, my supervisor, for continued interest and constructive support throughout the whole duration of my PhD study. I have been lucky in being able to follow through an area of personal interest whilst being guided to a completed thesis. Her expertise in proof reading has been invaluable throughout my writing up period and her knowledge of the entire field of neural-networks has been invaluable.

Thanks are due to Geoff Chappell and John Taylor for several useful discussions on the Temporal Kohonen Map.

Thanks are due to David Lee and Simon Courtenage for help with proof reading and a special mention for Dave Parrott for being a source of wisdom in all areas during my time at UCL.

I would like to acknowledge the Science and Engineering Research Council of Great Britain for funding of my study.

Finally, I want to thank my wife Gillian Romano for the huge support and belief that she has given me throughout. Over the months of writing up she has encouraged, cajoled and bribed me to finish. I cannot overstate her contribution.

David A. Romano-Critchley

February 1994.

Abstract

This work extends the Kohonen self-organising map in two primary ways:

- A dynamic extension to the model which allows the neighbourhood size and learning rate timecourse to be deduced during learning.
- Inclusion of temporal features, both in single layer and hierarchical networks.

The dynamic learning parameter model is developed as a consequence of how the self-organising map forms 'stable states' under fixed values of the learning parameters whilst exposed to a driving probability distribution. Such stable states can be used to deduce an appropriate stage to make a transition to a new set of learning parameters. This leads to a sequence of states that ultimately result in convergence.

Temporal features are developed in the light of the Temporal Kohonen Map model of Chappell and Taylor. It is shown that application of the standard Kohonen learning law to such a network can lead to instability in the weightspace. This problem is shown to be soluble by moving the integrating characteristics from the cell body (where it is a scalar quantity) to the synapses (where it is a vector quantity).

Multilayer temporal topographic mappings are discussed in terms of coding strategies between layers. The codings examined include complete feed-forward, feed-forward with enforced output spectrum and 'triangular coding', a binary coding of topology.

Contents

Chapter 1

Introduction

1.1 Why Neural Computing?

1.1.1 What is neural computing?

The first useful electronic digital computer appeared in 1946. From that point until the late 1980s, practically all information processing has been tackled by a single stratagem, that of *programs*. A computer programmer has always sat down and compiled a set of *rules* and *algorithms* which are then embodied in software. The program then follows the specifications set down to the letter, its behaviour only changing from incremental improvements and revisions *added by the programmer* (ignoring *bugs*, which are just unforeseen errors in the implementation of the rules). This situation is ideal for where the rules that apply to a particular application are clearly defined. Because computers are entirely logical in operation, code must be perfect for it to work as anticipated. Exhaustive design and testing, followed by cycles of refinement is a time consuming and expensive process. The conventional approach, then, falls down in two areas:

- Formulation of rules for solving a problem is often highly complex, if not impossible.
- Translation of the rules into a computer program will *always* result in errors at some level, and these errors will typically result in a program 'crash' or entirely unanticipated outcome.

Neural Computing has emerged as the antidote to this situation for many areas of difficult problems such as pattern recognition and data analysis. It has long been known that the ***brain*** solves the tasks of vision, speech recognition, co-ordination of muscle movements and other complex data processing operations that are so difficult to even attempt on a digital computer. It does so in a *fault tolerant manner*: many thousands of cells die in the brain each day and are not replaced, but our faculties do not disappear in a drastic fashion (in contrast with the effect of a programming

error as described above). Furthermore, in a case like speech or vision, the problem is an intrinsically *parallel* one, with a multitude of different and conflicting inputs that trigger memories and ideas. It is the combination of all these different processes that allow us to perform these tasks. The brain, with its massive parallelism, is able to store and represent this knowledge in an accessible way and combine it with other stimuli. It is not speed that dominates the operation, the basic computing speed of a modern computer being a million times faster than the firing rate of a neuron. It is the *parallelism* which is why the brain is good at its job. For example, humans when asked to carry out a task like comprehending a phrase in English, can do so in about half a second. The basic computing speed of a neuron is around a few milliseconds and so such a task is only needing about a hundred steps [16].

The brain does have a large number of neurons, estimated at around 10^{10} and each one is connected to about 10^4 other neurons. However, this seemingly huge number is quite a limitation. An illuminating example is that of vision, which has about a million parallel inputs. Clearly, the brain isn't running an $O(n^2)$ algorithm as it wouldn't fit! [16]

But perhaps the most striking and important aspect to note, is that biological brains *learn* from experience and learn *without* rules. Children learn to speak from example, to write, in fact everything we associate with intelligent behaviour. People make mistakes, but the difference is that they learn from them and are less likely to make them in the future.

Thus, neural computing is an attempt to extract from living neural networks algorithms and architectures that can be used for information processing in artificial neural networks. The bottom up approach to the problem includes detailed analysis of biological neurons, synapses etc. In contrast, the top down approach includes study of architectures of functionally specific areas of the brain such as the hippocampus (believed to relate to short term memory) and higher cognitive processes. The trend is towards "a reverse engineering" perspective in which biological mechanisms are used to solve difficult information processing problems [20].

1.1.2 The History Of Neural Networks Research.

The field is currently enjoying a renaissance, following in the wake of Rumelhart et al and the multi-layer perceptron (circa 1986) [45]. Its origins, however, are much earlier; Aristotle had analogies for mental processes, or thoughts, based on hydrodynamics [34]. Turing was aware of the possibility of modelling intelligent functions of the brain by neuronal computation [25], as was von Neumann who showed that neural networks could be equivalent to Turing machines [23]. Modern approaches to neural modelling can be traced back to the work of McCulloch and Pitts in 1943 on the collective properties of thresholding, neuron-like processing elements [46]. These networks were put forward as general computing devices.

In 1949 Donald Hebb suggested that in biological networks, it is the synapses i.e. connections between separate neurons that are modified when the system *learns* [22]. Hebb went further to say that frequently active synapses should be modified so as to achieve greater chance of being active in the future. This is an example of unsupervised learning. Frank Rosenblatt used a different error-correcting *supervised* learning paradigm in 1957, by building an artificial network of what he called 'perceptrons' [64]. Such a system could solve simple linearly separable pattern recognition problems.

Work continued through the sixties but was dealt a crippling blow in 1969 due to the work of Minsky and Papert that highlighted the failure of single layer perceptrons to solve simple non-linearly separable problems such as exclusive-or and also highlighted how important an issue *scaling* was [47]. They went further to speculate that even the addition of extra layers would not improve computational power significantly, saying that "the extension is sterile" in the lack of a general learning rule. Dreyfus and Dreyfus describe Minsky and Papert's 'attack' on 'gestalt thinking in AI' as having succeeded beyond their wildest dreams [12]; interest and more importantly funding was diverted almost exclusively to their own domain of rule-based artificial intelligence.

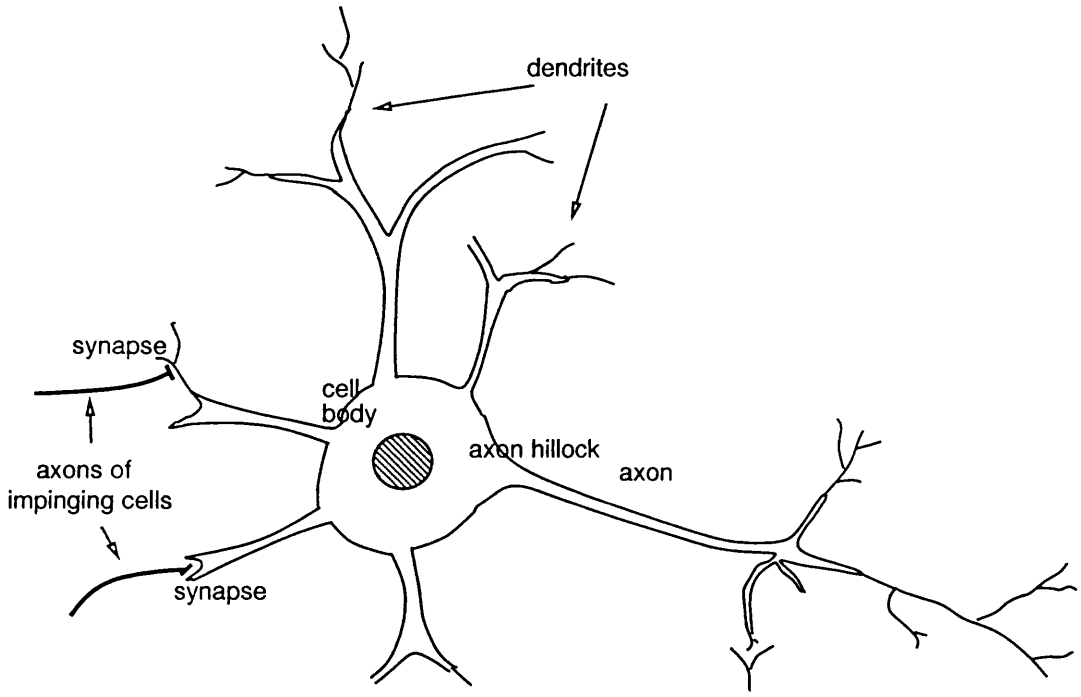


Figure 1.1: Generic biological neuron.

Some researchers, notably Grossberg, Kohonen, Aleksander, Hopfield and Taylor were active in the 'quiet period' before the recent upsurge of interest in the subject, but it was certainly the multilayer perceptron and 'back-propagation', its learning rule, that opened the floodgates.

1.2 Fundamentals Of Neural Networks

1.2.1 Biological Neurons

All artificial neural networks consist of simple processing units, *nodes* or *neurons* which have some similarities with neurons in biological brains. Essentially, a neuron is a *decision unit.*

The basic features of a biological neuron are shown in Figure 1.1. Inputs are

'collected' by the cell's *dendrites* through synaptic connections from other neurons. These inputs are then summed at the axon hillock, some being excitatory and some being inhibitory.

The neuron fires (sends out an electrical signal along its axon) if the summed potential V is greater than the critical potential V_{crit} or *threshold.*

1.2.2 Artificial Neurons: Connections, Weights and Synapses

In biological neurons, the physical gap between impinging neurons' axons and the collecting neuron's dendrites are bridged by synapses. Vesicles containing neurotransmitter are released on stimulation from the nerve impulse and they migrate across the synaptic cleft to acceptor sites in the dendrites. Synaptic efficiency determines what effect the incoming signal has on the receiving neuron.

The generic artificial neuron consists of a summation and thresholding device (analogous to the cell body) which receives input from other units, weighted appropriately (See Figure 1.2). The thresholding function maybe a step-function or sigmoid, or some other function which usually saturates.

Real neurons are much more complex than this model suggests. The cells in the brain have complex chemical reactions within them and leaving out such details may be important. Real synapses are subject to random release of neurotransmitter quanta which can lead to spontaneous firing of neurons, even when there is no input [68]. The ability for neurons to have temporal properties i.e leaky integrators, is also an important factor that this simple picture neglects.

1.2.3 Fault Tolerance and Generalisation

In artificial neural networks, the synaptic efficacy is modelled by connections having different strengths or *weights.* Information is represented by these weights, in a *distributed representation,* meaning that the whole network is involved in computation. If a few nodes or weights fail, then the system does not fail catastrophically, but

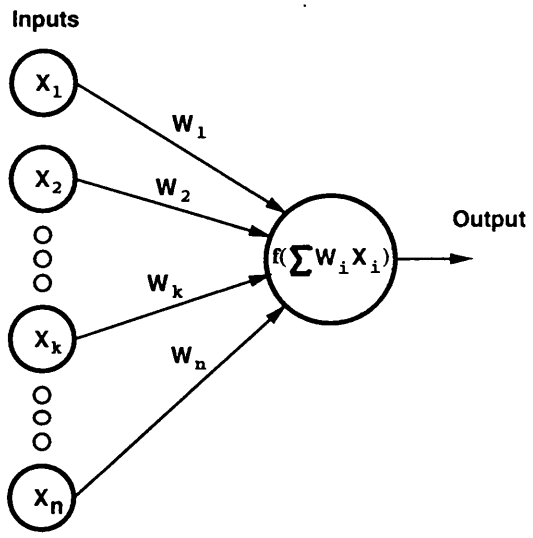


Figure 1.2: Generic artificial neuron.

its performance degrades slightly. Neural networks are thus both robust and *fault tolerant* devices (although recent work by Bolt has shown that fault tolerance should be a specific issue in the *design* of neural network architectures for this to be relied on [7][6]). Damage to a neural network must very extensive before there is serious degradation of performance.

Recovery from damage will also be much quicker than re-learning an entire problem - the weightspace will only have to perform a little reorganisation to account for the disruption.

The distributed nature of neural networks is also responsible for their ability to *generalise*. Particular 'feature detectors' may develop in groups of neurons in the evolving network, which are sensitive to some certain aspect of an input space. A network that is exposed to a pattern that is similar to one it is has already learnt before will be classified correctly. An input pattern that may be a composite of several recognised patterns will be classified by the 'strongest feature'. Generalisation, means therefore, abstraction and removal of redundancy - why store a separate

exemplar for a pattern that can be described by others? This makes good sense in brains of finite size.

On the whole, neural networks are good at *interpolation*, i.e. they can allow for intermediate states between patterns seen, enabling them to classify unfamiliar ones. They are bad at *extrapolation*, i.e. classifying patterns which are outside the range of ones already seen, that is there is little to compare them with.

1.2.4 The Binary Decision Node (BDN)

Caianiello investigated networks using the simple model of a neuron described above with a hard-limiter thresholding function [8]. The state space of a network of n of these nodes is thus $\{0,1\}^n$.

He found that the net activity formed cycles in the state space, but that these cycles had lengths much greater than the age of the universe... and so had difficulty in making an analogy between these cycles and 'thoughts'.

An important theorem is that a network of BDNs can perform *any* logical function - AND gates and OR gates can both be formed from them. Any logical function can be decomposed into a combination of ANDs, ORs and NOTs and be performed by a 2-layer feedforward BDN network.

However, for such a universal architecture, it has been shown that the average number of BDNs required to perform a boolean function in n variables is given by

$$\begin{aligned} < N > &=< A > +1 \\ &= 2^{n-1} + 1 \end{aligned}$$

where N is the number of BDNs required and A is the number of AND gates used [1]

An illustrative example is one of a classifier trying to classify a 10 by 10 binary image. The average number of BDNs required is $\sim 10^{30}$, about 10^{10}. $[WORLDMEM]$. $WORLDMEM$ is the total amount of computer memory in the world...

Thus, this architecture is not of much practical use and direct setting of weights and thresholds is not something that the brain does. Neural networks need to *learn*

the parameters to perform a particular mapping and this is what sets them apart from other computing devices.

1.3 Categorisation of Neural Networks

A network will ideally develop an internal representation of information to which it is exposed. The law which governs how the weights should be changed to achieve this, and any concomitant architecture needed, is the principle distinction between different kinds of network.

The above process is in contrast to standard algorithmic programs on digital computers which are programmed to behave in a certain way. Neural networks can be employed in situations where the rules or the program is non-obvious, for example classifying very complicated patterns.

1.3.1 Functions Performed By A Neural Network

Neural networks can perform three basics types of operation: Auto-Association, Hetero-Association and Classification (See Figure 1.3). All three functions are *pattern recognition* problems. In essence, neural networks is all about pattern recognition of some form, where in general a pattern means any kind of arbitarily complex spatio-temporal data that is being processed.

1.3.2 Architectures

The architecture of a neural network is the description of the type of nodes employed together with their connectivity. This definition then leads on to further description in terms of computation *style* performed by the network. The two styles are *feed-forward* and *feed-back* or *recurrent* computation:

- In feed-forward networks, there is a well defined,single direction of information flow, from *inputs* to *outputs* (See Figure 1.4). Typically, the flow is from layer

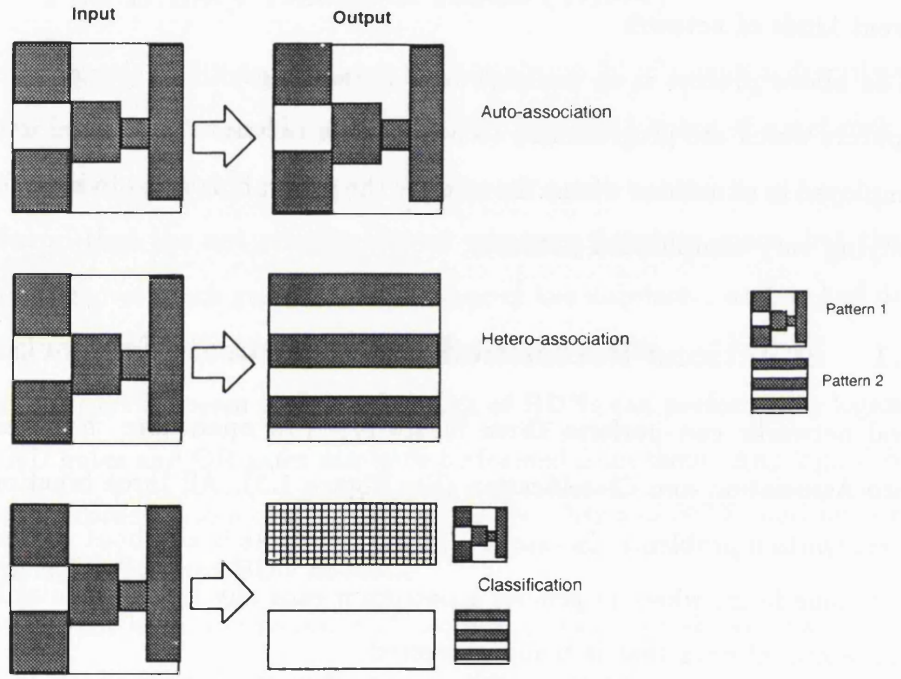


Figure 1.3: Operations performed by neural networks.

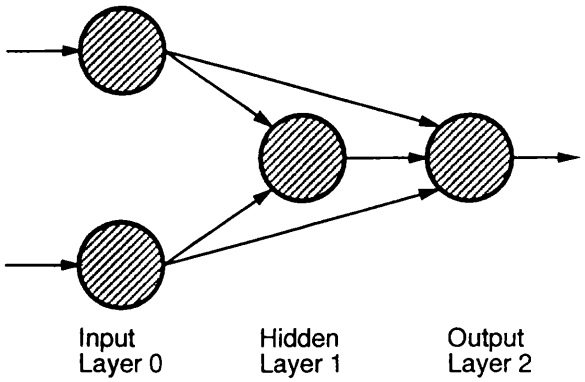


Figure 1.4: A feed-forward neural network.

n to layer $n+1$.

The feed-forward net, by use of adaptive connections or *weights* can then perform a *mapping* operation, in keeping with its input/output computation style, i.e. for an input vector $\underline{x}_i$ there is a corresponding output vector $\underline{y}_i$. A prime example of a feedforward network is the multilayer perceptron (this is strictly true only once the network is trained: during training *error signals* propagate *backwards* towards the earlier nodes)

- In recurrent networks, outputs of units are fed back into other units, or themselves, as input, so that the input to a particular unit at any one time may be a combination of information from the environment and input from any other node in the network. The system performs iterative convergence towards some fixed state, an *absorbing state.*

$$\begin{aligned} \underline{x}(t+1) &= F(\underline{x}(t)) \quad \text{for iteration} \\ \underline{x}(t) &= F(\underline{x}(t)) \quad \text{for the absorbing state } \underline{x}(t) \end{aligned}$$

The most simple instances of this kind of network exhibit complete connectivity. They do not distinguish between input layers and output layers and input

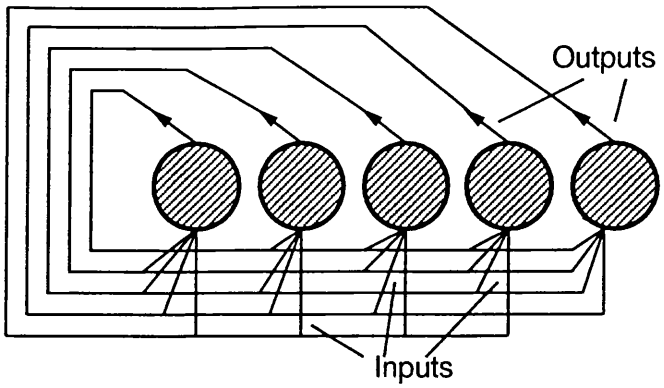


Figure 1.5: A recurrent neural network.

to the system must consist of 'clamping' the network in some initial state. The Hopfield network is a prime example of a recurrent network (See Figure 1.5).

1.4 Learning in Neural Networks

As has been mentioned, knowledge is stored in the weights of connections of a neural network. This knowledge is stored in a distributed way, so that each node is involved in representing many separate mappings. The crucial feature of neural networks is the ability to *learn* these weights from experience, implementing an *update rule* for the modification of the weights.

There are broadly three classes of learning:

1. Supervised Learning.

   In this scheme, an external teaching input is introduced from the environment which prescribes the desired output, and the network compares this with its own output. The weights of the system are then updated so as to minimise the discrepancy, or error, between the teacher and itself.

2. Reinforcement Training

In this scheme, a global reward/punishment signal is received from the environment. The update rule serves so as to minimise the probability of receiving a penalty signal and maximise the chance of a reward signal when the network produces an output for a given input. This is "learning with a critic", whereas supervised learning is "learning with a teacher".

3. Unsupervised Learning

   The aim of this learning strategy is for the network to discover statistical regularity and structure within an input space it is exposed to and be able to form *classes* of input vectors. This inevitably requires some external supervised learning at some stage, to 'label' the different classes. But this is a practical constraint, rather than a necessity.

1.5 Summary of Chapter 1

We have briefly reviewed the field of neural computing and its history, noting that it has emerged as the antidote to algorithmic computing.

Chapter 2

The Kohonen Self-Organising Map

2.1 The Kohonen Self-Organising Map

2.1.1 Introduction

The self-organising map or **SOM** was devised by Teuvo Kohonen in the period 1979-1982 [31]. Self-organisation as a field of study, however is older, with work carried out by von der Malsburg in 1973 and Willshaw and von der Malsburg in 1976 [76].

Kohonen's model has some similarities to classical k-means clustering analysis in statistics [41][23] (although this method requires an assumption of the number of classes in the data, and *does not* preserve topological information). It also draws from biological evidence that biological brains form topological representations of input stimuli, most notably in the visual cortex of humans. The central region of the visual field is mapped topographically onto the external surface of the cortex. Furthermore, the central 10% of the visual field occupies about 60% of the total brain map. Thus there is *variable magnification* in the mapping.

There are many other such variable magnification topographic maps in the brain such as the 'motor homunculus' which is responsible for control of muscles in the body, or more generally, *sequences* of muscle contractions that are in keeping with body's perceived environment (the supplementary motor area) [10].

2.1.2 Aim Of The Self-Organising Map

The aim in developing the SOM was to produce a mapping system which retained as much information about the *structure* of the input space as possible, whilst of course abstracting the most important features. In most other artificial neural networks there is some distributed storage of patterns, but the actual structure i.e. topology and probability distribution of the feature space is effectively lost. The topographic map *compresses* data: high-dimensional feature spaces are typically mapped onto a 2-d sheet of cells. The resulting map thus allows complex statistical relationships, i.e. class clustering information, to be rendered in a graphical format. In this way,

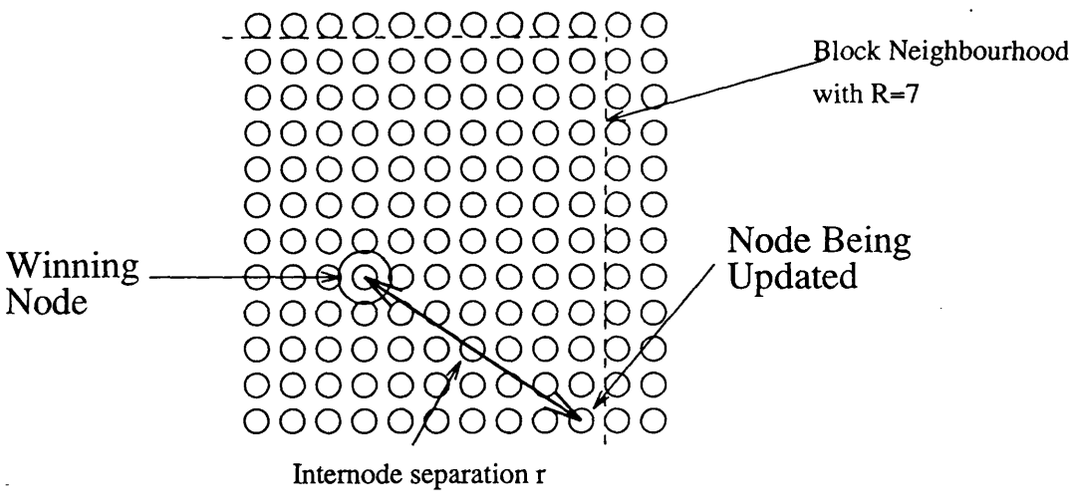


Figure 2.1: Cartesian node arrangement in the SOM.

the SOM acts as a *vector quantiser* i.e. it classifies N dimensional input vectors onto M clusters [41]. Clearly, the number of nodes in the SOM layer gives the upper bound on the number of classes that can be distinguished.

Perhaps one of the most important aspects of the self-organising map is what its name suggests, i.e. the formation of a mapping occurs by *unsupervised learning*. Thus classes in the input data are 'discovered' and not imposed. Clearly, real biological systems (brains) have to learn to abstract and classify without rules.

The SOM is a competitive network, meaning that a node's response depends on it winning some form of competition over the other nodes.

2.1.3 Basic Architecture

In keeping with the biological analogy, the SOM comprises a single layer *surface* of neurons, typically two-dimensional although 1-D and higher dimensional structures are possible, onto which it is intended to form a topological mapping of a space of presented input vectors. The actual geometry of the network is unimportant [37] eg. hexagonal lattice versus cartesian. The latter is often employed for its simplicity.

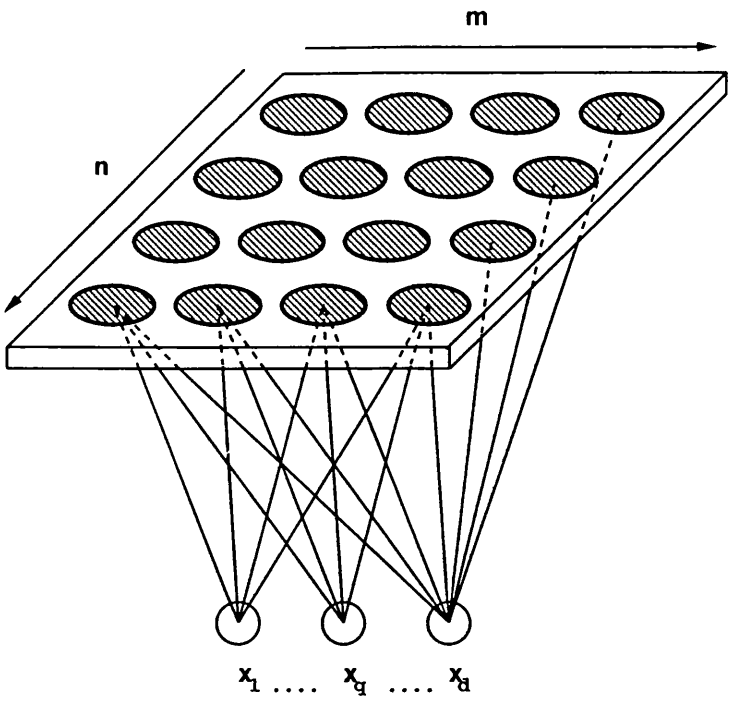


Figure 2.2: Weight arrangement in the SOM.

Say then that the network comprises $m.n$ nodes arranged in a rectangular grid. An input vector arrives from the environment and is fed in parallel to each node in the network by a group of distributor nodes, one for each dimension of the input (See Figure 2.2). Typically, the network is fully connected i.e. there is a weight from each distributor line to each node. For example, with a 2-D input, there are two weight components per node. The number of weights in the system is thus $m.n.d$, d being the dimensionality of the input patterns.

Each node then processes the input in parallel.

2.1.4 Node Function

The operation that the nodes perform in the basic SOM is one of measuring *similarity* to a presented pattern. It is not the dot-product of weight and input that the

nodes calculate, as without normalization this can ultimately only measure the angular separation of the vectors, not magnitudes as well. Hence each node calculates the *Euclidean distance* D between its weight vector and the current input (or more simply the square of this quantity, as finding the square root is redundant in terms of comparing the nodes' relative success ($a > b \rightarrow a^2 > b^2$, if a and b both positive))

$$D_i^2 = (\underline{w}_i - \underline{I})^2 \tag{2.1}$$

The input pattern is then *claimed* by the node n which *minimises* Equation 2.1 i.e. there is competition between all nodes to produce a *winner* that most closely matches the input vector.

How then is this winner generated? In biological topographic maps, there is evidence for *lateral-inhibition* mechanisms whereby nodes can turn each other on/off depending on their proximity. This interaction can be modeled by the 'Mexican Hat' function (See figure 2.3) or difference of Gaussians, which has a central region of excitatory interaction close to the central node, which then reverses to become inhibitory and then drop to zero at larger distances. Kohonen has demonstrated that such lateral inhibition can indeed produce a winning region on the network which develops dynamically in response to input [37].

In practice, the above lateral interaction computations can be side-stepped by *choosing* the winner of the distance competition. This necessitates an $O(m.n)$ search of the all nodes. The output of this winner is then taken as 1 and 0 for all other nodes.

The above exhaustive search step required by the algorithm does not lend itself to parallelisation. Indeed, one can more easily conceive of a real hardware parallel machine that *does* perform lateral inhibition and would be appreciably faster. But in the realm of sequential simulation the overhead of *simulating* the lateral inhibition process is prohibitive in the average application and complicates the implementation unnecessarily.

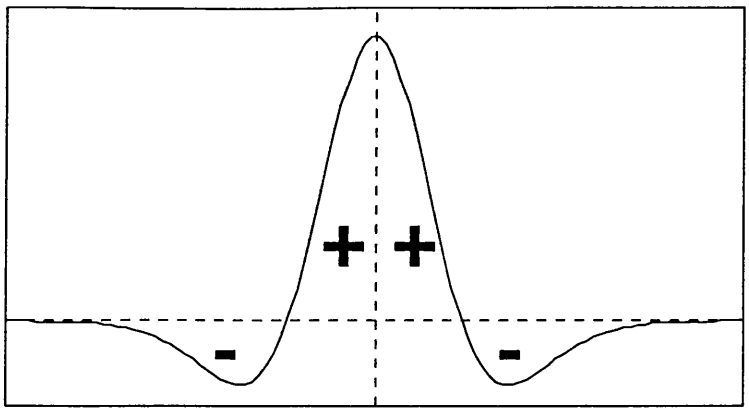


Figure 2.3: "Mexican Hat" lateral interaction function.

2.1.5 Learning Law

The learning (weight-change) law is given by

$$\underline{w}_i(t+1) = \underline{w}_i(t) + \eta(t)H_{win}(t)(\underline{I}(t) - \underline{w}_i(t)) \tag{2.2}$$

The objective of this law is to increase the likelihood that the winning node, and ones around it, will win again, when presented with the vector $\underline{I}$ and ones similar to it. The law serves so as to *rotate* each node's weight vector towards the current input vector. η is the learning rate and H_{win} is a function which describes a *neighbourhood of interaction* surrounding the physical site of the winner on the grid of nodes (See figure 2.1). This neighbourhood function describes to what extent neighbouring nodes are rotated towards the input vector. Initially the range of interaction is large. The functional form of H_{win} can most simply be a discontinous block i.e. all nodes within a square region a distance R around the winner rotate towards the current input by the same relative amount, those outside the block remaining unchanged. This is the original form used by Kohonen (see Figure 2.4(a)).

The adjustment of weights of nodes that are *neighbours* to the winning unit is the key factor in the system forming a topographic map of the input space. Ideally, pattern vectors which are close together in their embedding feature space, should

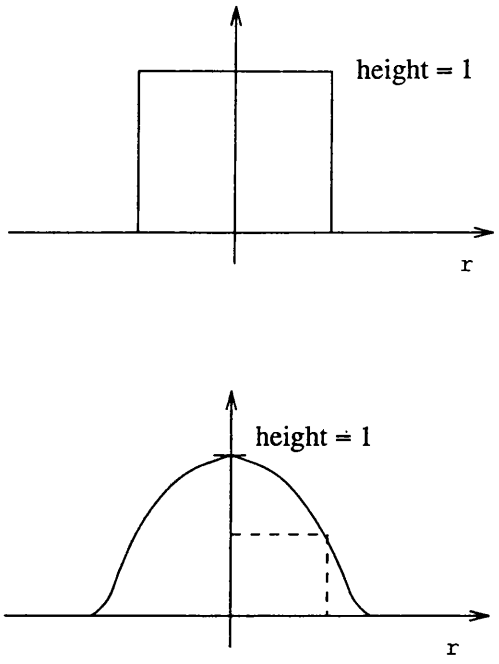


Figure 2.4: Block and gaussian shaped neighbourhood functions.

be mapped to nodes which are physically close on the output layer of the network.

2.1.6 Neighbourhood Functional Form

H_{win} can also be some smoothly decaying function, such as a gaussian; this has been demonstrated to give greatly improved convergence rates [42]. The reason for this is fairly obvious in that the strength of attraction of nodes towards the current winner should be strongest for those topologically close to the winner and progressively weaker for more distant nodes which will be attempting to map distant regions of the input space. It doesn't make sense to rotate all nodes' weight vectors equally, even ones within a restricted region, as this serves to undo previous learning for some nodes. Ultimately, a block neighbourhood will provide convergence on account of the reduction in block size but it is considerably slower.

The actual process of learning involves first initialising the weights to some (po-

tentially random) values and having the neighbourhood size and learning rate large. Then these parameters are arranged to decay with time as learning progresses.

The formation of the mapping takes place in roughly two stages:

1. Arrangement of the weights into topological order.

2. Tuning period where the weights increasingly accurately map the input space.

The network is exposed to randomly presented training patterns and initially forms a coarse mapping of the input space, being allowed to make modifications of global scope to the initial weight space. This is the requirement of having a large initial neighbourhood and learning rate. As the learning parameters decay, the scope of modifications becomes smaller and so the detail and accuracy of the mapping improve until finally, the neighbourhood size becomes such that only the winning unit is afffected and individual weights are modified by a small residual learning rate - so called 'Kohonen learning' [23]

Kohonen gives a proof that such a learning law leads to a stable topographic map, in the one dimensional case [37] and Ritter and Schulten have analysed the convergence and probability distribution of a two dimensional map [57].

It is important to note that the probability density of patterns is recorded in the mapping, as in biological topographic maps. Thus higher frequency of presentation of a particular pattern results in greater overall rotation of all vectors in the current neighbourhood to that pattern and ultimately more nodes will be devoted to the representation of that pattern. It is therefore important during training to consider the order in which patterns are presented.

2.1.7 The Initial Weightspace Configuration

The initial state of the weight space also needs to be considered, as a purely random starting set cannot guarantee correct convergence. Incorrect convergence is typified by 'twisted maps' which are locally topographically correct but globally in-

correct and can never untwist themselves and have been demonstrated in Kohonen's 'movies' [37].

Kohonen has pointed out that if the SOM is relevant to biology, then the initial weightspace must somehow be formed with reasonable default values i.e. genetics must provide a layout that is basically *topologically* correct which can then self-organise into precise topological mappings [23].

2.1.8 Time Dependence Of Learning Parameters

As has been discussed, the training process takes place in two stages: The forming of a coarse, but topographically correct mapping, and then the fine tuning of the weight vectors. It is however not usually clear as to what timescale this should occur on, with respect to decreasing the neighbourhood size and learning rate. Kohonen talks of 'rules of thumb' in determining such matters. A linear decay to one and zero respectively in a time T_{end} is an obvious choice, but suffers from the problem that T_{end} is arbitrary and the rate of convergence is quite poor.

It is clear that the coarse mapping phase requires much more severe and wide reaching changes in the weightspace, ie. the learning rate and neighbourhood size both need initially to be very large. For example, the latter might cover 75% of the map and the former be as high as 0.8. However, the learning parameters do not need to stay at this magnitude for long, perhaps only a few tens of iterations. A linear decay of parameters can thus not provide both a large, initial 'burst of activity' and a protracted convergence phase. The obvious next step is to divide up the learning into two linear phases, of steep and shallow negative gradients respectively. But again, one always returns to the problem of actually deciding these time periods. The best that can be done is running trials.

Exponential decay of the learning parameters better models the requirements of the timecourse required. It also has the virtue of being characterised by just one parameter, the half-life $T_{1/2}$. This too, of course is arbitrary.

2.2 Visualising Network Behaviour

2.2.1 A 2-Dimensional Training Example

Kohonen and others have used a graphical format for displaying the state of an SOM. These are 'weightspace' plots, in which the weight vectors of nodes that are physically adjacent on the output layer are connected together. Figure 2.5 shows a network of 16 nodes, arranged in a 4 by 4 grid, together with two weightspace plots. The weights are two dimensional, and can thus be plotted as points (x, y) The points are marked with labels Wn, where n refers to the corresponding node in the grid. The upper plot is almost topologically complete, but has two weight vectors W11 and W15 interchanged. The lower plot is complete. In this way, the topology of a mapping is clearly visible i.e adjacent nodes on the grid have vectors which point to adjacent points in weightspace and thus do not cross. The probability distribution is also visible, with higher density regions (corresponding to more frequently presented patterns) having more weight vectors allocated to them and hence a tighter 'mesh' in the weightspace plot.

We now consider a typical example of the SOM in operation, performing the near bench-mark task of mapping the unit square. In this example, an eight by eight network is exposed to training vectors drawn uniformly and randomly from the above region. There is no dimensional compression here, but a direct correspondence desired between points in the weightspace and nodes' spatial positions.

Figure 2.6 shows six snap shots of the evolving network. The parameters used for the simulation were

1. An initial learning rate η of 0.1
2. An initial gaussian neighbourhood of standard deviation (radius) $\sigma = 3$
3. A parameter half life $T_{1/2} = 2000$ iterations.

The six frames are for epochs 0, 35, 170, 500, 1500 and 5000. Initially, the weights

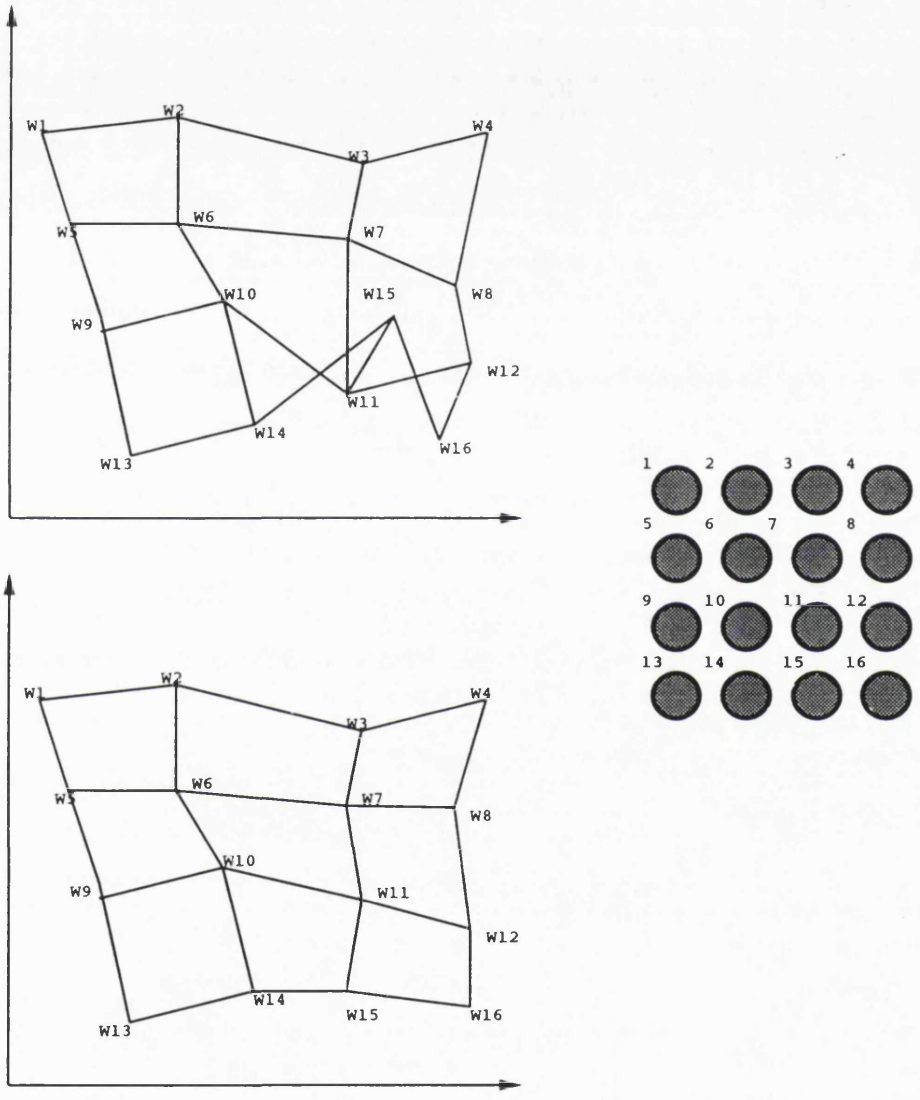


Figure 2.5: Example weightspace plots - top plot is not fully ordered, bottom plot is fully ordered.

are randomly distributed around the centre of the input space ie. $w_{ij} \in 0.5 \pm \epsilon$ where ϵ is a random variable in the range $0 \rightarrow 0.5$ and $i = 1, 2, .., 64$ and $j = 1, 2$. The frame around each plot marks the boundary of the unit square. It can be clearly seen how the network first unravels the initial tangled set of weight vectors and by 170 epochs has almost completed the ordering of the weight vectors. At 5000 epochs, the network is well on the way to asymptotic convergence. Note that the edges of the plot will never quite reach the edges of the space it is trying to map. This is because there are no training vectors outside the square, and so on average these boundary nodes have their weights pulled back into the central region, due to attraction from interior nodes. Kohonen has shown that this boundary effect scales as 1/m, where m is the number of nodes in the side of the square grid.

The boundary effect can, however, be reduced by means other than introducing more nodes. If the parameters decay such that when the neighbourhood has shrunk to only affecting the winning node, then further useful fine tuning can be done to the individual node's weight vector if there is a residual learning capacity a, so that the learning rate $\eta(t)$ is given by

$$\eta(t) = \eta(0) \exp(-\lambda t) + a \tag{2.3}$$

By the stage that a is the dominant term, then all weight vectors will be in almost their optimal positions, with the exception of the boundary nodes. At this point, the system is essentially disconnected i.e. each winning node behaves as an autonomous agent, and can move according to local fluctuations in the probability density, according to the size of a. Now freed, the boundary nodes can move their weights out further into the periphery of the input space. Of course, if a is too large, then the map may well become corrupted, since that topology is no longer being enforced by the co-rotation of neighbouring weight vectors.

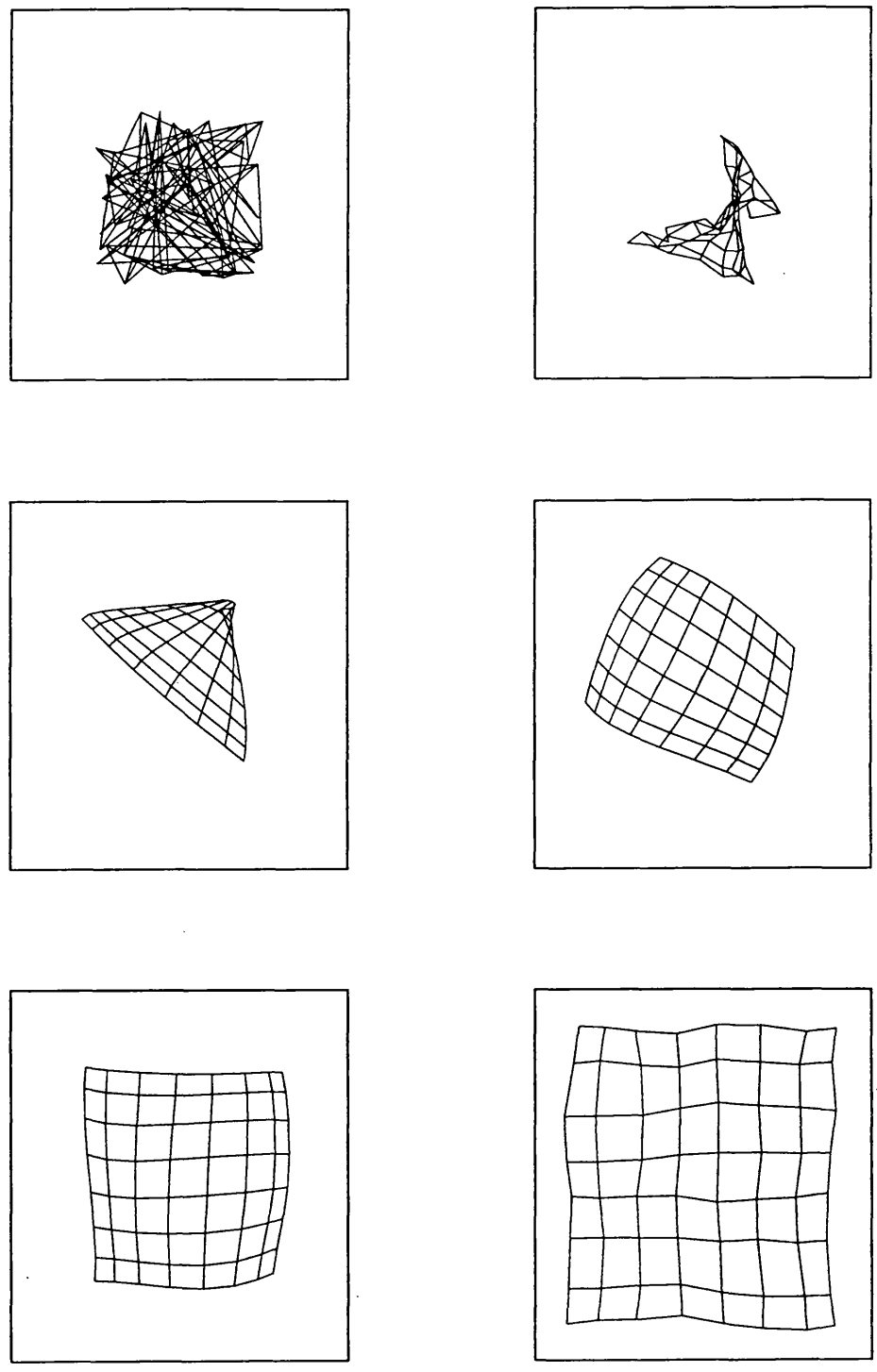


Figure 2.6: Snapshots of a network learning to map the unit square.

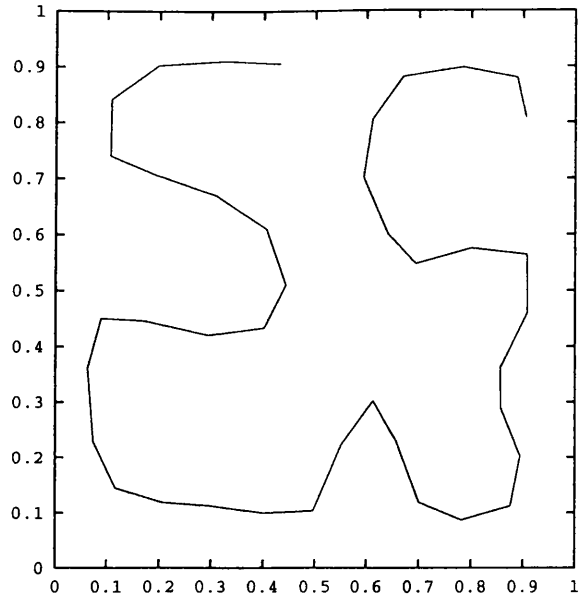


Figure 2.7: The unit square approximated by a linear array SOM.

2.2.2 Dimensional Reduction

One of the great properties of the self-organising map is its ability to perform data-compression or dimensional reduction. Here we consider the mapping of the unit square again, but this time onto a 1-dimensional *chain* of nodes. Here, the network must compress the 2-D training vectors into a meaningful abstraction in 1-D. The result after 10,000 epochs can be seen in Figure 2.7.

This is an example of a Peano curve and serves so as to 'fill out' as much of the input space, the unit square, as possible (it is reminiscent of some of Mandelbrot's space filling fractals). Linear arrays often tend to approximate higher-dimensional distributions by such curves [37]

2.3 Visualisation: Clustering, Classification and Labelling

Weightspace plots are ideal for visualisation of low dimensional input spaces. Obviously, however, any space of dimension 4 or higher does not have a direct way of displaying the weightspace and a 2-D projection of a 3-D space (necessary in order to view the 'interior' of a 3-D plot) is likely to be too complicated. In general, the SOM is applied to finding *classes* within a feature space and so a visual description of this classification appropiate to input vectors of arbitrary dimension is required.

Ideally, if an n-dimensional space has m classes embedded within it, we hope that the map will develop m regions that respond preferentially to presentation of members of that class and are arranged in a meaningful topographic map i.e. if class x is in some way similar to class y and class z, then it should occupy a portion of the map spatially close to x and y. So a *labelling process* is required. In essence, this just means tagging a node as belonging to a particular class, if it won on presentation of a training vector that belonged to that class. Unfortunately, this is somewhat of a circular argument, as it is the network itself that forms the clusters and so for the outside environment to then say that a vector $\underline{I}$ belongs to class X is perverse. However, if the SOM is to be of any practical value, then class labelling must be performed. This requires a set of prototype patterns for which the class label is undisputed.

An excellent demonstration of the abstraction and classification properties of the map and the above visualisation procedure, is in the work of Ritter and Kohonen on semantic hierachies [60][61]. Here, the SOM is used to display semantic relations between symbolic data.

This example is one where a collection of objects are to be mapped as a discrete symbol set. The objects chosen in the example are sixteen different animals, the symbol for each one being a 16-D unary coded binary vector. The relationships

		dove	hen	duck	goose	owl	hawk	eagle	fox	dog	wolf	cat	tiger	lion	horse	zebra	cow
is	small	1	1	1	1	1	1	0	0	0	0	1	0	0	0	0	0
	medium	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0
	big	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
has	2 legs	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
	4 legs	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
	hair	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
	hooves	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
	mane	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0
	feathers	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
likes to	hunt	0	0	0	0	1	1	1	1	0	1	1	1	1	0	0	0
	run	0	0	0	0	0	0	0	0	1	1	0	1	1	1	1	0
	fly	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0
	swim	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0

Figure 2.8: Data used in semnatic map simulation.

between the animals are introduced implicitly by an *attribute vector*, a 13 component binary vector (See Figure 2.8 which shows the attribute vectors for each of the sixteen creatures). The full description of each animal is then given by concatenating these two vectors.

$$\underline{x} = \begin{pmatrix} \underline{x}_S \\ \underline{x}_A \end{pmatrix} \tag{2.4}$$

During training, the two parts of the vector are given different weightings. The $\underline{x}_S$ are clearly just arbitrary labels and as such serve only to make each class instance unique. The feature vector $\underline{x}_A$ is thus made to dominate formation of the mapping. Also, the $\underline{x}_A$ are normalised, reducing the number of degrees of freedom by one and hence improving convergence.

After training, each node in the grid is tested to see which animal name produces the greatest response in it, by presenting the name part of the input vector only to the network i.e.

$$\underline{x} = \begin{pmatrix} \underline{x}_S \\ \underline{0} \end{pmatrix}$$

The results of the visualisation procedure Ritter and Kohonen call "Simulated electrode penetration mapping" are shown in Figure 2.9. It can be clearly seen that

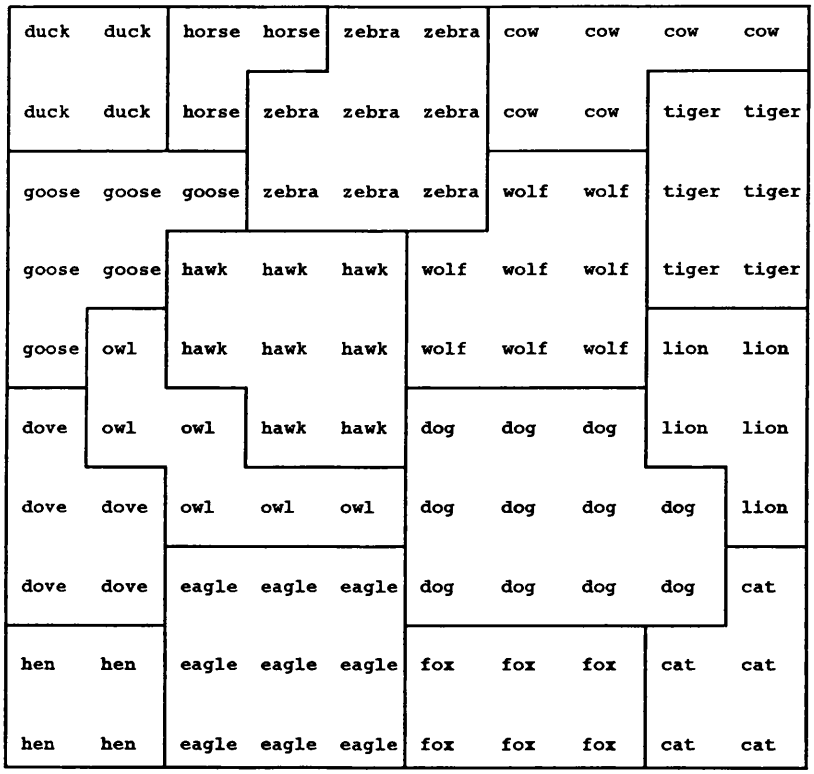


Figure 2.9: The labelled semantic map after training.

common hierachies are represented, e.g. "birds" on the left hand side, "herbivores" along the top and "carnivores" towards the right. Similarly, animals falling into multiple categories have spatial mappings which reflect this e.g. hawks are both carnivores and birds, so thus appear on the right hand edge of the bird groups.

2.4 Learning Vector Quantisation (LVQ)

When the self-organising map is applied to practical classification problems, the resulting weightspace will require some fine tuning to properly adjust the class boundaries. This process can be performed by a supervised technique called *learning vector quantisation* [37] [39].

In its simplest variant, LVQ has the form

$$\begin{aligned} \underline{m_c}(t+1) &= \underline{m_c}(t) + \alpha(t)\,[\underline{I}(t) - \underline{m_c}(t)] \quad \text{classes of } \underline{m_c} \text{ and } \underline{I} \text{ agree} \\ \underline{m_c}(t+1) &= \underline{m_c}(t) - \alpha(t)\,[\underline{I}(t) - \underline{m_c}(t)] \quad \text{classes of } \underline{m_c} \text{ and } \underline{I} \text{ disagree} \\ \underline{m_i}(t+1) &= \underline{m_i}(t) \quad \text{for } i \neq c \end{aligned} \tag{2.5}$$

where $\underline{m_c}$ is the weight of the maximally active unit c after presenting the training vector $\underline{I}$ and $\alpha(t)$ is a learning rate as before. Each member of the set of training inputs $\underline{I}$ *has a known classification.* It should be noted that LVQ is a clustering method in its own right (Kohonen describes it as a special case of the self-organising map [37]). It is applicable to the self-organising map because self-organisation of the training vectors allocates weights according to the input distribution of the patterns. LVQ needs to have *initial* weights reflecting the probability distribution *of each class* before it is applied. Each node is then labeled according to its *class* [35].

Equation 2.5 is applied iteratively as per the self-organising map algorithm and α is reduced slowly, from an initially low value, typically 0.1. The decision surface of the classes produced by LVQ complies very closely with that of the equivalent Bayes classifier. It should be noted that the interior of the class density functions

are of much less importance than the decision surfaces themselves (typically they will not remain faithful). [37]

Experiments have shown that LVQ gives considerable improvement to speech classification problems (See next section) and can enable a trained self-organising map to quickly adapt to new speakers [5].

Kohonen et al [39] have introduced variants of the LVQ method which better comply with Bayesian classifiers.

2.5 Applications of The Self-Organising Map

We will now review two applications of the SOM. The first, the phonetic typewriter, is perhaps the most famous use of the map algorithm. The second, an optimisation problem, shows the flexibility of the SOM.

2.5.1 The Phonetic Typewriter

Kohonen has applied the SOM in a hybrid system for the difficult problem of speech recognition. This "Neural" Phonetic Typewriter, as it is called, can translate Kohonen's native tongue of Finnish into text [36]

The problem of speech recognition is difficult because

- Speech phonemes have varying amplitudes and waveforms from person to person.
- Pronounciation of phonemes is context-dependant: the larynx and soft palate do not always have time to return to their initial rest positions and thus co-articulation effects come into play.
- Humans can infer the meaning of an unclear word by analysing sentence structure and semantics of what is being said

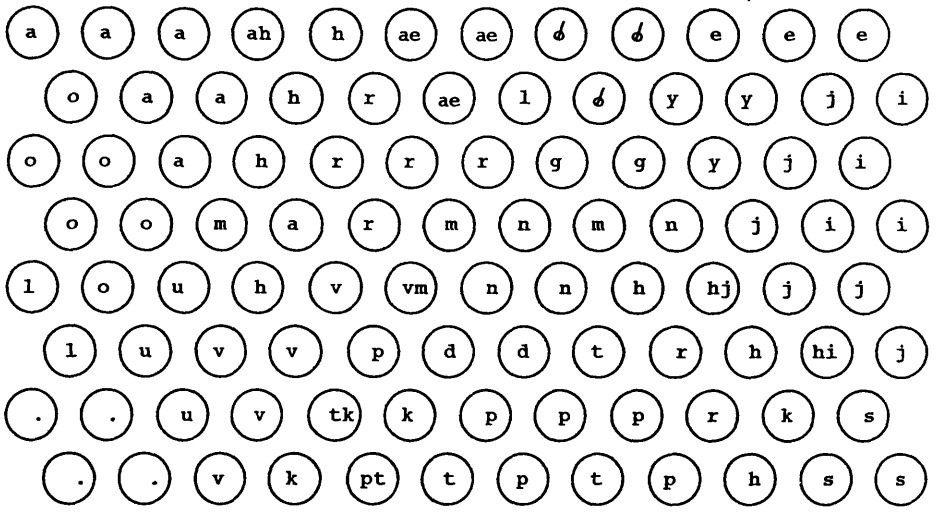


Figure 2.10: Phonotopic map for Finnish speech.

The phonetic typewriter is a combination of conventional digital signal processing, neural and rule-base system technology. It consists of three sections:

1. **Preprocessing**

   The speech-waveform is first Fourier transformed in time-slices of 9.83ms. This data is then used to form a 15-dimensional continuous pattern vector, the components representing the instantaneous power in one of 15 frequency bands from 200Hz to 5kHz. A sixteenth component is used to represent the rms value of the speech signal.

2. **The Topographic Map**

   The above pattern vectors are then mapped on to an SOM, using fifty samples of each test phoneme. The map is labelled to form a "phonotopic map" , symbols being attached according to the phoneme each node was,on average,the most responsive to. Figure 2.10 shows the labelled output layer.

   Figure 2.11 shows the trajectory of phonemes as recognised by the network whilst articulating the Finnish word "humppila". Kohonen has suggested that

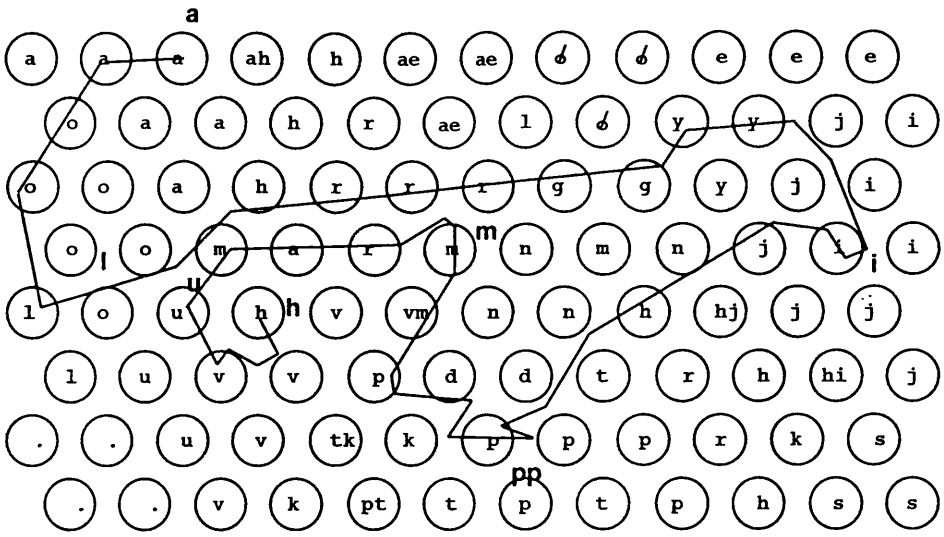


Figure 2.11: Trajectory of Finnish word "humppila".

this kind of visual representation of a phonetic string may be of use for speech training and therapy. Profoundly deaf people may find it advantageous to be able to associate a visual sequence with the speech they have just formed.

3. **Postprocessing**

Finally, the recognised phonemes must be translated into typescript. Correction of errors from the previous stage, which has been mentioned are primarily due to co-articulation effects, are carried out. To this end, Kohonen has added a rule base of 15,000 to 20,000 rules to cope with re-constructing correct grammar from trajectory data obtained from the phonotopic map.

The complete system, hosted by an IBM PC can operate in near real time. Only a slight pause is required between words. Phoneme classification is between 80-90% correct at the phonotopic map stage and is improved to 92-97% correct after processing by the grammar rule base. The system adapts well to new users, needing about 100 words for each new user and a training time of about ten minutes.

The phonetic typewriter is commerically viable, but has not been adapted to cope with English or other languages. Finnish is a phonetic language, which makes the speech to text problem more straight forward. However, the system does show the advantages of a hybrid approach, and the preprocessing stage is an excellent example of providing a neural network with the 'right kind' of data i.e. the data which is most suitable for training in this case is FFT time slices and not the raw data itself.

2.5.2 The Travelling Salesman Problem (TSP)

This application, presented by Ritter and Schulten [59], was inspired by the 'elastic net' method of solving the TSP [13]

Definition of the TSP

For completeness, we state the notion of the travelling salesman problem: A travelling salesman must visit all of a number of cities within a given area. He must visit each city only once and his goal is to minimise the distance that he travels on the tour of all cities ie find the shortest round trip. Finding the *best* solution to this problem is costly, as the search time is $O(n!)$ for n cities.

For the simulation, two dimensional inputs and weights were used with a closed linear chain of 100 neurons. The probability distribution was concentrated to a set of 30 randomly chosen locations (cities), located within the unit square. A regular hundred-side polygon was used as the intial value of the weights.

During the training sequence, the polygon gradually deforms into a path connecting all 30 cities in a closed loop. The tendency of the map algorithm to preserve neighbourhood structure results in the system producing very short tours as the final weightspace. It is pointed out, however, that slightly longer tours than minimal may be obtained with this method of solving the TSP.

In summary, this work demonstrates how the self-organising map can be successfully applied to a difficult optimisation problem.

2.6 Summary of Chapter 2

We have reviewed the basic formalism of the Kohonen self-organising map and its successful application in areas such as speech recognition, semantic maps and optimisation, showing that the map algorithm is extremely versatile.

Chapter 3

Learning with Adaptive Parameters In the Kohonen Network

3.1 A Review of Convergence Properties of SOMs

There has been much mathematical analysis of Kohonen's self-organising map model. We report here some theoretical results that are relevant to any simulation work using the model.

3.1.1 Constraints on Learning Rate Functional Form

In [58] Ritter and Schulten derived conditions for the learning rate $\epsilon(t)$ namely

$$\lim_{t\to\infty} \int_0^t \epsilon(t')\,dt' = \infty \tag{3.1}$$

$$\lim_{t\to\infty} \epsilon(t) = 0 \tag{3.2}$$

These conditions are necessary and sufficient for the convergence to an asymptotic equilibrium weightspace $\underline{w} = (\underline{w}_1, \underline{w}_2, \cdots, \underline{w}_N)$ (N is the number of weights in the system) from any initial state that lies sufficiently close to $\underline{w}$ i.e. meaning that the weights have completed the initial ordering phase of the map algorithm. Equations 3.1 and 3.2 are satisfied by the set of all functions $\epsilon(t) \propto t^{-\alpha}$ with $0 < \alpha \leq 1$. Note that this is *not* exponential decay, as used in much of the literature! However, in *practical* applications, apart from a small residual $\epsilon(t)$, the condition $\int \epsilon(t)\,dt \gg 1$ is sufficient. Moreover, the *precise* time course of $\epsilon(t)$ is not significant so long as it decreases monotonically. Hence, the benchmarking examples that follow in this chapter use exponential decay to be consistent with other work.

3.1.2 Selecting Optimal Network Parameters

Much work has been carried out on the optimal parameters that a Kohonen network should be initialised to, in order to provide maximum convergence rate. A lot of this work has looked at the effect of the form of the neighbourhood function.

Two recent articles have looked at the effect of different neighbourhood functions. The first investigated the one dimensional case of a chain of neurons, the second to what extent the results in [15] apply in the general case and how the end probability distribution is also affected by the choice of neighbourhood function [65].

In [15], it was concluded that the *rate* at which the map algorithm converges depends on the shape of the neighbourhood function. Furthermore, for a fixed learning rate, there is an optimal width of this function, for which convergence time is shortest. The "best" neighbourhood function should be one that is "convex" over a large range around the winning unit and yet also has large differences in values at neighbouring nodes.

For the gaussian neighbourhood

$$H(r,s) = \exp(-(r-s)^2/2\sigma^2)$$

the full width at half height (2σ) should be of the order of the number of neurons in the chain.

3.1.3 Metastable States

Metastable states are discussed in [15], which are fixed points of the mapping algorithm other than the optimal *ordered* representation. The algorithm may become "trapped" in these metastable states for a finite number of iterations, before the optimal representation is discovered. It can be proved that no metastable states exist for broad, convex neighbourhood functions. However, for gaussians with σ below a certain width, there are metastable states. Ideally, then, σ needs to be as large as needed to avoid metastable states, but not so large that convergence time is increased by spending too long in an already ordered state. Those neighbourhood functions which are not convex anywhere have metastable states for all parameters, and the ordering time is much longer than for a gaussian.

3.1.4 Neighbourhood Functions In The General Case

The generality of the above is examined in [65]. The critical parameters for the algorithm are the number of units in the chain N, the initial learning rate $\alpha(0)$ and the functional form of the neighbourhood. Obviously, the wider the neighbourhood function, the greater the effect that the winning node has on other neurons' weights.

If time dependence is removed by using a fixed α and σ, we can ask the question what value of σ gives the fastest convergence to *an ordered state*, i.e. when the weights are ordered in increasing or decreasing order. In [65], the point is made that the value of α should also be allowed to take on other values apart from 0.01, as used in [15]. For a chain of 10 units, the case $\sigma = 10$ (approximately) gives the fastest convergence to an ordered state.

Two main conclusions were drawn from the results:

- $\alpha(0)$ could be increased to very large values and still get good convergence properties. In fact, values of greater than 1.0 are still acceptable.
- The basin of "good convergence" is very large, ranging to the non-optimal values of σ. This illustrates that self-organising maps are very robust over a wide range of parameter values.

The value of σ of course depends on N, the size of the net, i.e. for a bigger net bigger values of σ are required. In [65], a rule of thumb is put forward relating σ and N:

$$\sigma = N + N/10$$

Of course, there are considerations other than those of 'What parameters get the quickest ordered state?'. Perhaps more important is the statistical aspect of the weight positions, i.e. that the weights approximate the point density function of the input space.In [65], it is shown by simulation that the optimal σ in terms of convergence rate is not at all optimal in terms of the system faithfully capturing the

probability distribution. Better results are obtained from using slightly non-optimal values of σ.

Other neighbourhood functions such as triangular or step impulse do have worse convergence to ordered state properties, but perform *better* in producing a faithful probability map. Thus the *total run-time* may well be shorter and the distribution much better, if "non-optimal" (in the sense of the ordering phase) values of σ are used.

In different problem domains such as function approximation, a gaussian neighbourhood function actually gives *worse* results than the step function.

In conclusion, great care must be taken when making claims about the performance of a system using a particular neighbourhood function. The intended problem domain has to be taken into account.

The real problem is that a general theory of map formation is still out of sight [15], with many basic problems unsolved.

3.2 Problems With Kohonen's Learning Algorithm.

Self-organisation, in all its modified forms, has involved the neighbourhood (whether smoothly decaying or block) and the learning rate shrinking on some arbitrary time scale. Kohonen's original work [37] used a simple linear fall off of both these parameters. Thus, the system starts with an initial neighbourhood size R_0 and learning rate η_0 and these fall linearly to zero after a time T. The network is assumed to have reached convergence at this time, but T is arbitrary.

The parameter timecourse function can be any monotonically decreasing one. Piecewise linear with negative slope and decaying exponential have both been employed in the literature with improved convergence, but no suggestion as to the underlying reason for this improved learning speed has been put forward.

3.3 Adaptive Learning Parameter Change

It would also be possible to decrement the learning rate and neighbourhood size in a 'stepwise' manner, provided a suitable criterion for the transition to new (smaller) values of these parameters could be established. In the following sections, it is examined how a network whose learning parameters are decremented in this way goes through a series of stable states whereby no further learning is possible until the learning parameters are altered. The detection of a stable state provides the desired criterion for transition to new parameter values.

3.4 Stable States In Kohonen Maps.

The learning mechanism in SOMs can be viewed as one where ultimately the weight vectors map out the extent of an n-dimensional structure i.e. the input probability distribution. Thus from an initial, arbitrary state, the weight vectors become self-organised to reflect this structure, so long as they are picked according the input probability distribution. It is also important to remember that the density of this structure at any particular point is represented by the number of weight vectors that cluster around that point. This density is analogous to mass-volume density in a three dimensional structure.

When learning begins, the network is quite plastic in the sense that the neighbourhood size is large, as is the learning rate. A particular weight that should optimally point to a particular point in the input structure can only move so far towards that point before its evolution is checked by attraction from other nodes. Thus the total 'volume' that can be mapped for a given set of learning parameters can not exceed a certain extent. An equilibrium comes into being, defined by the driving effect of the input probability distribution and the retarding effect of other interior nodes pulling back those on the boundary. This represents a stable state of the network where the input cannot be better mapped without an alteration of the

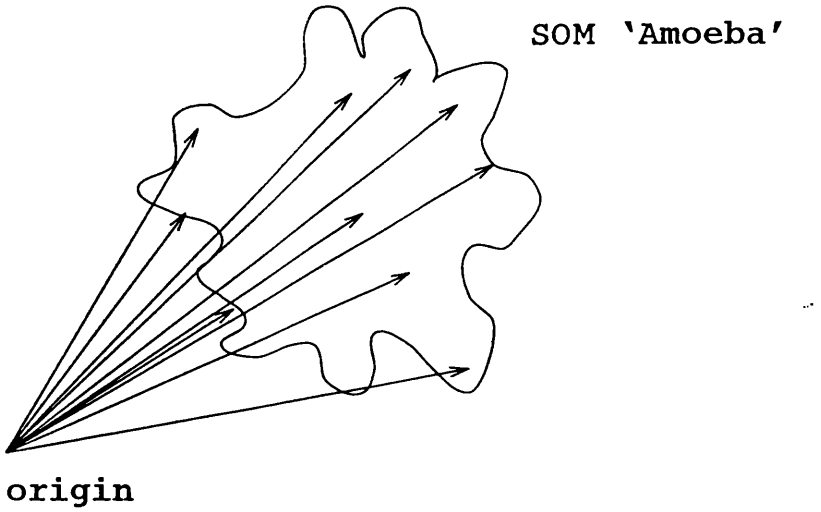


Figure 3.1: Weight vector envelope.

learning parameters.

This state can be visualised as an 'amoeba' structure in the space of the input. In the example of Figure 3.1, the input to be mapped is a 2-dimensional distribution that surrounds the amoeba. The amoeba itself is the network's *current* representation in weightspace of the input space. Note that the weights w_{ij} are not normalised. The creature can flex and writhe but is unable to *expand* out further to improve the mapping if it has not yet reached the ideal mapping. It is thus trapped at a certain 'size' and can only oscillate. Something must happen in order to break the symmetry of the state.

3.5 Triggering Adaptive Parameter Change

The inability of the network to perform further learning would suggest that the stable state could be used as a signal for a transition to a different set of learning parameters; the neighbourhood size and learning rate must both be reduced. Once this transition has occurred, the symmetry is broken and the driving attraction of

the extremes of the input distribution are now stronger than the retarding effects of interior nodes.

The system is now no longer in an equilibrium state and the 'amoeba' can expand further, until the attraction from other nodes balances the boundary of the input probability density function. This is a new stable state, one of a sequence which we hope will converge on the optimal solution. The problem of deciding when to make a transition is then one of monitoring some easily accessible parameter of the network. Each state results in a more detailed mapping of the weight space, following the tuning process discussed before. Explicitly calculating this weight space 'volume' is a computationally expensive process; however, another parameter can be more easily used as a gauge of the rate of progress of the algorithm.

It has been observed over the course of much experimentation that for a given learning rate and neighbourhood size

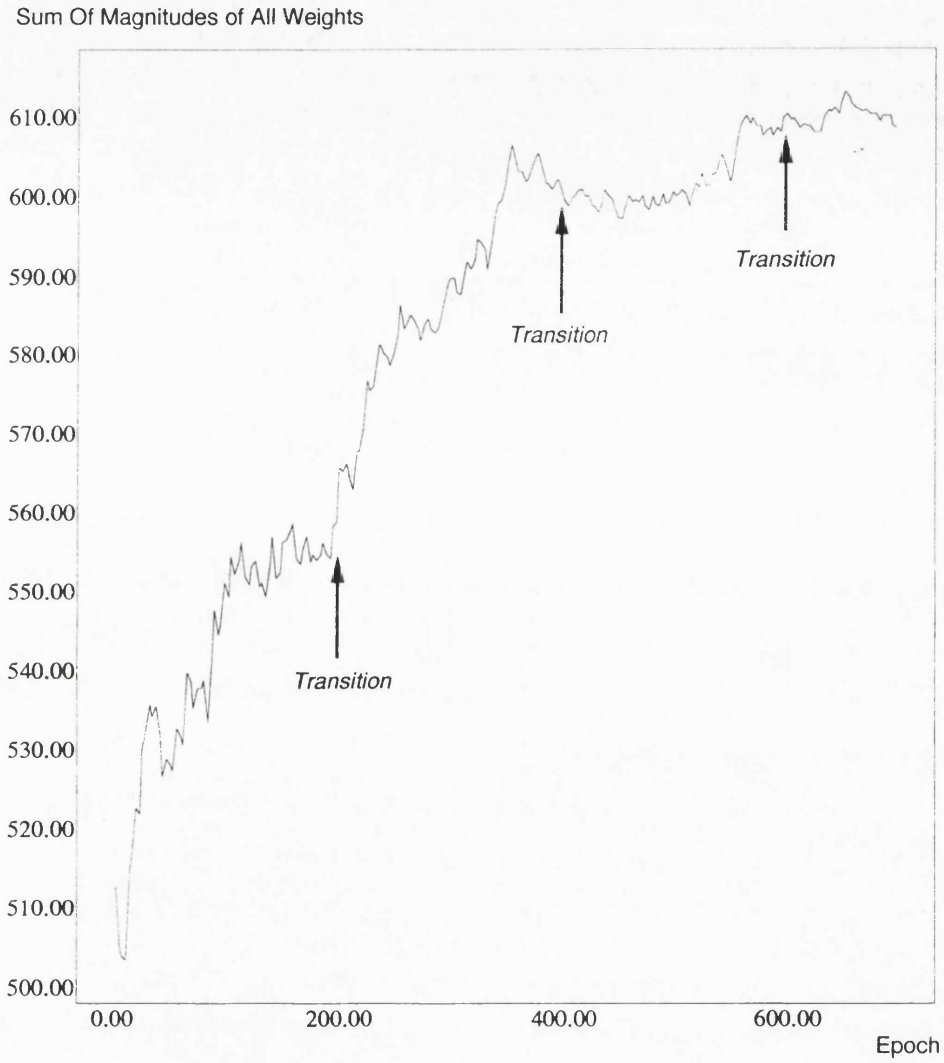
$$\bar{M} = \langle \sum_{i=1}^{n} \underline{w}_i^2 \rangle \tag{3.3}$$

has stable values, which are correlated with stable, sub-optimal configurations of the weight vector envelope ('amoeba').

The form of such a measure can only be a *guess*, but is attractive for its simplicity. It permits getting a handle on the problem.

Figure 3.2 shows the evolution of the above measure for a network which has artificially induced parameter reductions. The changes occur at 200, 400 and 600 epochs, when both neighbourhood size and learning rate are reduced. It is apparent that the system has approached some kind of stable state in between the steps, as the measure approaches an average equilibrium value. These plateaus correspond to the amoeba of Figure 3.1 being trapped at some size. Clearly, after a succession of many states, the change in size can become arbitrarily small and the system could be said to have converged.

In a higher dimensional view point, the system has a state $\underline{\Psi}(t)$ which is a vector of all the weights in the network $\underline{\Psi} = (w_{11}, w_{21}, ...w_{nm})$ where m is the dimension of



Graph showing the effect of externally introduced transitions at Epochs 200, 400 and 600.

Figure 3.2:

the input vectors and n is the number of nodes in the network. This state vector represents a single point in phase space. It is the trajectory of this point that charts the evolution of the network, and the trapping of this point in the neighbourhood of a sub-optimal solution. This *evaluative stable mapping* signals the necessity of a transition to new learning parameters.

3.6 Identifying Stable States.

In the analysis of Ritter and Schulten [57], the final stationary mapping condition is a re-arrangement of Equation 2.2

$$\langle \underline{w}(t+1) - \underline{w}(t) \rangle = \langle H(R,r)\eta(t)\underline{I}(t) - \underline{w}(t) \rangle = 0 \tag{3.4}$$

where $\langle \cdots \rangle$ means the average over presentation of training patterns. An evaluative stable mapping should satisfy a similar kind of constraint, due to the fact that on average the weights are 'trapped' in a region of weight space i.e. the average position of the weight should be constant. The *variance* of the position should decrease to a local minimum at a stable state.

The evolution of the system can productively be discussed in terms of the global state vector $\underline{\Psi}(t)$. Then more aptly, equation 3.4 can be written as

$$\langle \underline{\Psi}(t+1) - \underline{\Psi}(t) \rangle = \underline{0} \tag{3.5}$$

and the diminishing variance of the state vector, $\sigma^2_{\underline{\Psi}}$, reflects an evaluative mapping, as the state vector becomes increasingly localised in the mn dimensional space of the weights (m is the number of neurons, n is the dimension of the weights). Using the sum of weights in the network is a coarser, but perhaps more straight-forwardly calculable parameter of the network than calculating explicit variances in the state vector.

The averaging process required to label a stable state starts at the beginning of *each* transition. The average value of the parameter used to gauge the network's

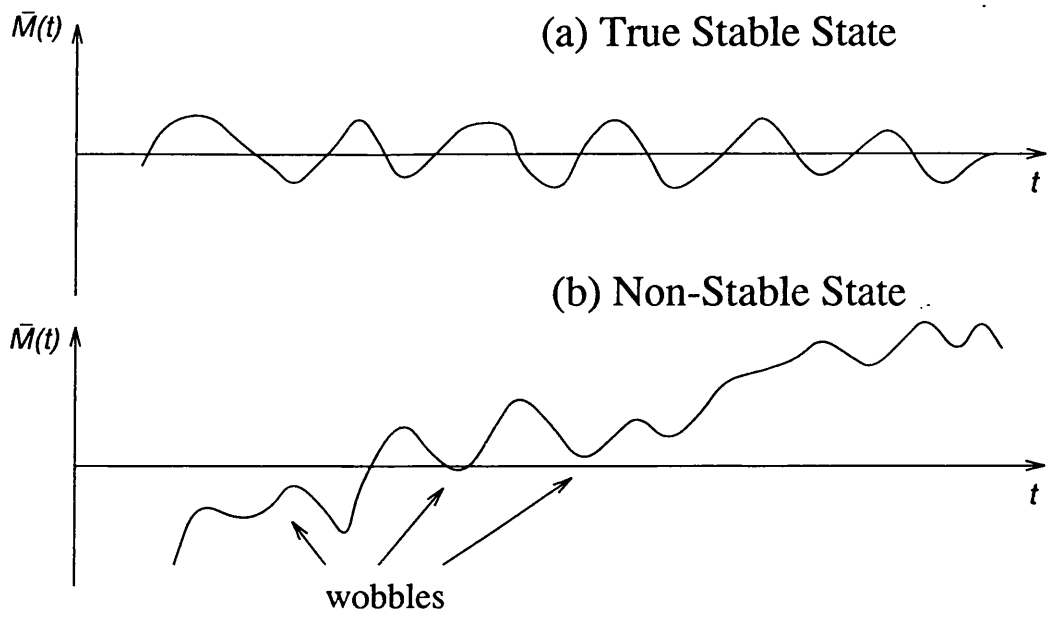


Figure 3.3: Schematic difference between stable and non-stable states.

progress needs to be recorded and then changes in the derivative of this smoothed function need to be monitored.

Thus at the start of forming a stable state, begin to calculate Equation 3.3

$$\bar{M}(t+1) = \frac{t\bar{M}(t) + M(t)}{t+1}$$

Now consider the derivative of the above average. The value of $\bar{M}$ still has many small local minima that are superimposed on the overall trend. Figure 3.3 shows schematically the difference between a stable state (a) and a non-stable state (b), which is still evolving. The former has clearly reached a stable value, despite the short time-scale fluctuations in its structure.

Consider then using a *trace*, X, of $\bar{M}$

$$X(t+1) = \delta_s X(t) + \bar{M}$$

where δ_s is a suitable smoothing decay constant. Stable states should then correspond to zeroes of $\partial X/\partial t$.

In practice, the trace function $X(t)$ still has a number of local minima in it, particularly within the first few epochs after a transition - interior nodes get more of the activity (are more frequently winners) as the map *unfolds* and can thus generate transient contractions of the 'amoeba'. It is thus a practical decision to *forbid* a new transition for say the first l epochs after the last one, just to allow the system to get a reasonable gauge on the average value of the measure $\bar{M}$.

A typical value of l is 30. It is a scale length that is usually much shorter than the time spent in a stable state. It can sometimes be seen that the system performs a transition about l steps after the last one. All this may mean is that the system spent longer in that stable state than ideally it would have and hence a slightly longer convergence time overall.

3.7 Making Transitions.

Once the condition for a stable state has been fulfilled, a transition must be made to permit further learning of the input space. The transition consists of reducing the neighbourhood R and the learning rate η

$$\begin{aligned} R &\rightarrow \lambda R \\ \eta &\rightarrow \lambda \eta \end{aligned} \tag{3.6}$$

where $0 < \lambda < 1$. The parameter λ is thus a 'decay constant' but the decay occurs only at each transition of the network and *not* at each epoch. The network then generates a new evaluative mapping under the new learning parameters.

The value of λ is not crucially fixed and would typically be around 0.95. If the value is too small, then after a few transitions, the learning rate and neighbourhood size become very small and further formation of improved evaluative mappings is thwarted i.e. the mapping becomes *choked.*

3.8 Effect Of Network Size and Neighbourhood Function On Stable States

The number of neurons in the output layer clearly has an effect on trying to establish a stable state. This is because the measure $\bar{M}$ takes into account the weights of *all* the nodes. In the extreme case of there being just a chain of two nodes, each node can effectively control the weight of the other, meaning that $\bar{M}$ will have a large variance. Increasing the number of nodes means that the input structure is being approximated much more accurately and hence $\bar{M}$ will show greater stability.

Gaussian shaped neighbourhoods mean that the stable state measure $M(t)$ is automatically more stable due to the decaying strength that surrounding nodes are influenced by the winner. This is in stark contrast to an impulse neighbourhood where all nodes in the neighbourhood are affected equally. This difference is clear from the results in section 3.9.1.

3.9 Results Of Simulations.

3.9.1 Using a Fit Function

In this first example, comparisons of the standard algorithm, use of Gaussian neighbourhoods (with exponential decrease of neighbourhood size) and the new dynamic approach were made using the sum of weight magnitudes measure (See Figure 3.4 for comparison of models). The standard test of the unit square mapping onto itself was employed for a network of 32x32 nodes which is the size used in [42]. These results, together with a brief description of the adaptive model, were first presented in [11].

The comparison is made by calculating a fit function $f(t)$ which compares the current weight vector $\underline{w_i}$ of all the nodes i to that of an 'optimal' vector $\underline{\hat{w}_i}$ which we would hope the network to reach. This approach has also been employed in [42]

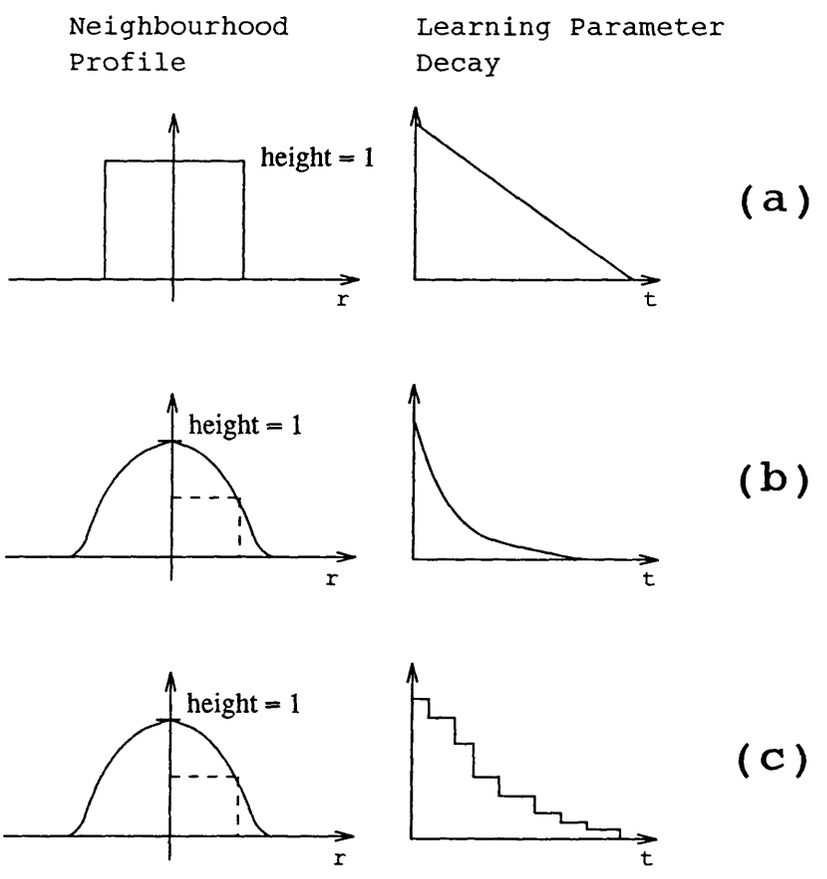


Figure 3.4: Details of the models compared in simulation.

In this example, the $\underline{\hat{w}}_i$ are given by the cartesian coordinates of the positions of each node i on the grid of the network.

$$f(t) = \sum_{i=1}^{n} (\underline{w}_i(t) - \underline{\hat{w}}_i(t))^2 \tag{3.7}$$

Clearly, for a network which converges correctly, i.e. it is not twisted, $f(t) \rightarrow 0$ as $t \rightarrow \infty$. The rate of fall off of this parameter gives a measure of the convergence rate.

The downside of prescribing these ideal weights is that it is only practical to calculate them for $n \rightarrow n$ dimensional mappings. It is not at all clear, in general, what the projection of a fifteen dimensional space onto a 2-d one should look like! That is usually what the network is trying to discover anyway.

Secondly, the starting weights must be seeded so the orientation and handedness of the mapping is the same as the idealised weights that we prescribe.

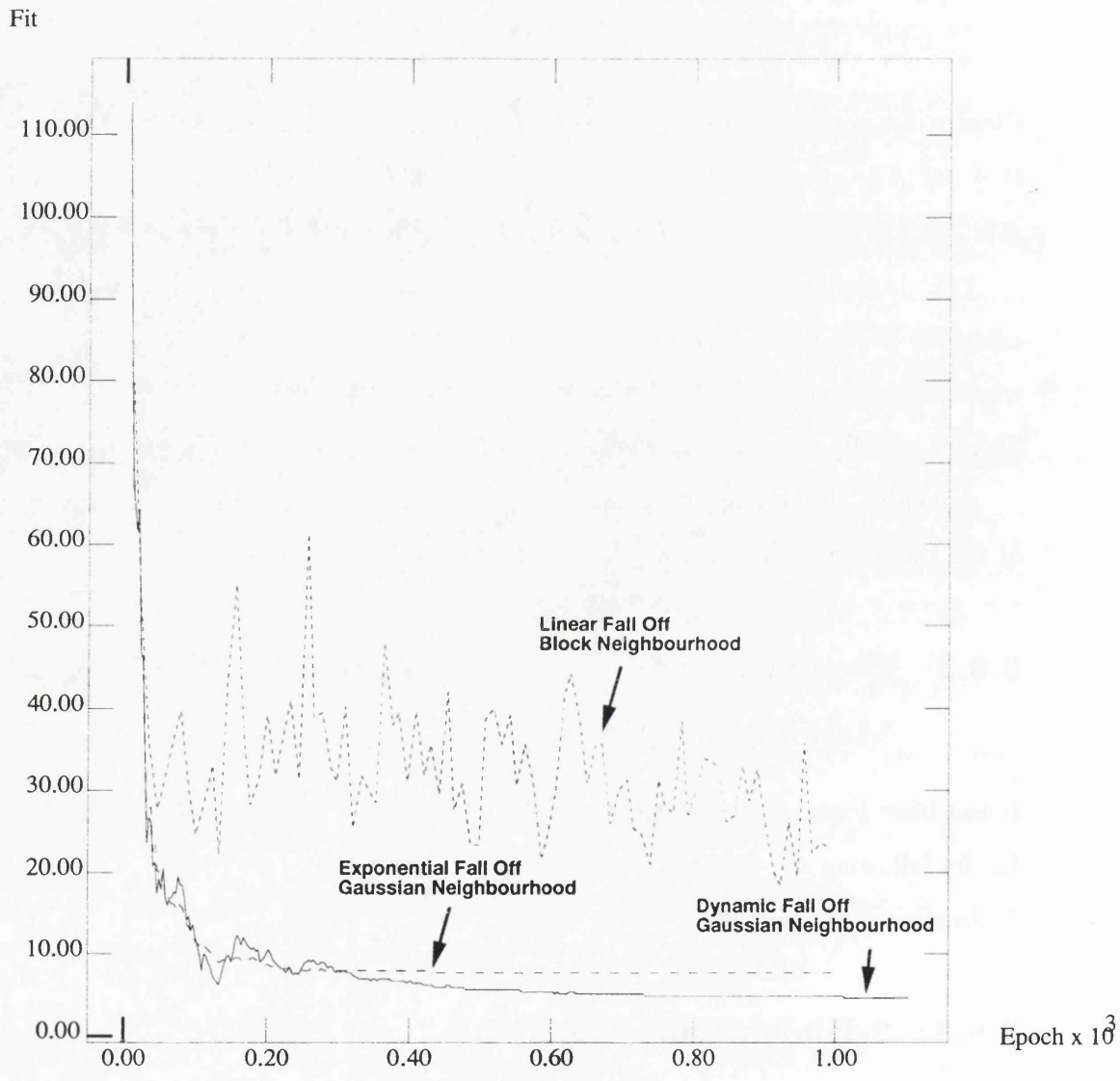
3.9.2 Comparison of the Dynamic and Standard Models: Mapping the Unit Square

It has been found that the new model performs well for a wide range of parameters. In the following two sets of examples, the unit square is mapped onto both a 1 and 2 dimensional output layer.

3.9.3 2-Dimensional Case

The parameters used for both models were: 100 neurons 10x10 output grid, initial learning rate $\alpha(0) = 0.35$, initial neighbourhood standard deviation $\sigma(0) = 5$. The standard model had a decay half life $T_{1/2}$ of 400 epochs.

Additionally, the dynamic model had a transition decay value $\lambda = 0.6$, trace smoothing value of $\delta = 0.5$ and transitions were forbidden for $l = 30$ epochs from the start of a transition. The simulations were allowed to run for 5000 epochs.



Graph Showing Comparitive Convergence Rates For The Three Models. All three networks were 32x32 nodes.

Figure 3.5:

- Measure $\bar{M}$ vs Epoch Figure 3.6(a) shows the value of $\bar{M}$ for both the dynamic model and the exponential decay model. Note that we have not plotted the trace $X(t)$ as this is only defined for the dynamic model. It can be seen that the dynamic model produces a stable value of $\bar{M}$ much more rapidly than does the standard ('fixed') model. This is because the neighbourhood and learning rate fall off much faster than exponential decay, the weight vector envelope becoming much more rapidly defined.

- Radius (neighbourhood σ) vs Epoch

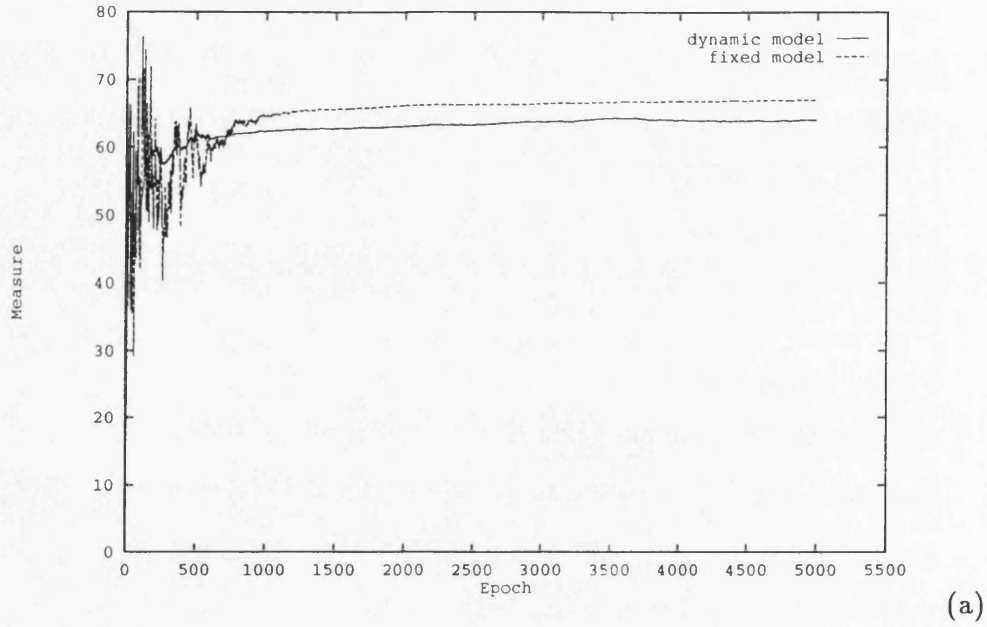
  Figure 3.6(b) shows the comparative time course for the two models. We reiterate that the fall off of parameters is much steeper than exponential decay. Note that the fall off of the radius bottoms out at around 0.5. This is fairly typical behaviour and represents the point when the neighbourhood function is essentially only affecting the winning node.

- Final Weightspace plots

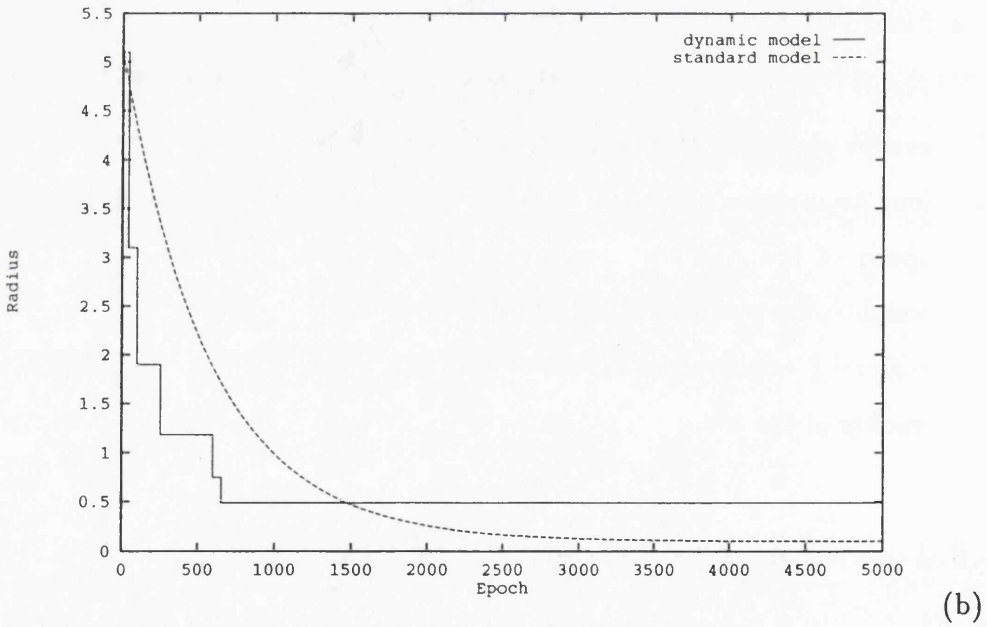
  Figures 3.7(a) and (b) show the final weightspace plots for the two models. It can be seen that they are very similar, as would be suggested by the similar long term values of $\bar{M}$ in Figure 3.6(a). Note however that the convergence speed of the standard model has been pushed to the limits, resulting in a weightspace plot that is noticeably crumpled. Thus despite occupying a larger region of weightspace, as indicated by the *different* long term values of $\bar{M}$, the *quality* of the mapping has suffered.

3.9.4 1-Dimensional Case

For the one dimensional case, a linear array of 100 neurons was employed, with $\sigma(0) = 50.0$. The learning rate $\alpha(0)$ was 0.35. The simulation time was 3000 epochs. For the standard model, $T_{1/2}$ was 200, again arranged to produce maximal performance.

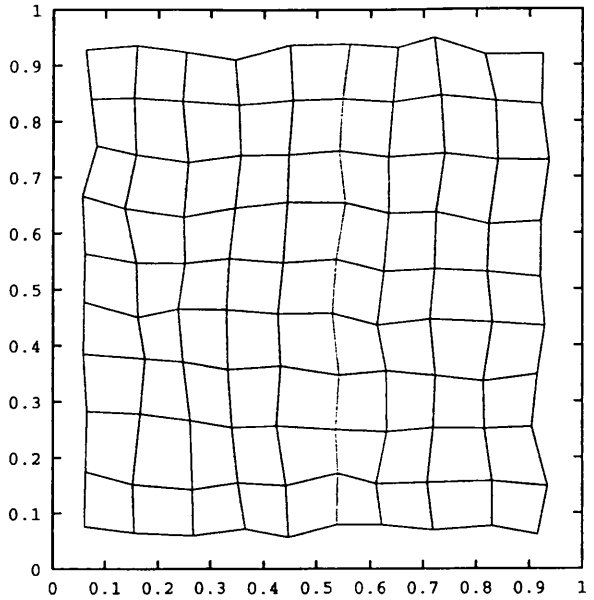


(a)

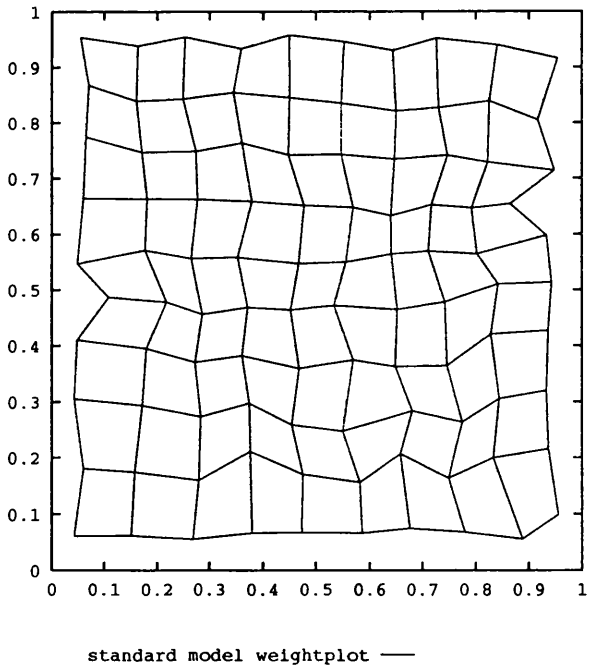


(b)

Figure 3.6: Evolution of $\bar{M}$ and σ for 2-d SOM unit square simulation for both dynamic and standard models.



(a)



(b)

Figure 3.7: Final weightspace plots for 2-d SOM unit square simulation for both dynamic and standard models.

In the case of dimensional reduction, it is highly unlikely that there will be a unique optimal state that the network will tend to, but in the end, the *approximations* that the network produces will be similar in their space-filling form. In this case, the linear chain should try to 'fill-out' the unit-square as best as it can.

Figure 3.8(a) shows the value of the measure against epoch. Note how rapidly the dynamic system collapses to a steady value. At around epoch 1700, the two systems have almost exactly the same value of measure.

Figure 3.8(b) shows the radius reduction points ie transitions which resulted in the above timecourse of the measure. The initial transitions occur rapidly, accounting for the observed stabilisation of $\bar{M}$

Figures 3.9(c) and (d) show the state of the weightspace after 1000 epochs of training for the dynamic and standard model respectively. The dynamic model is more evolved (convoluted, in this case) at this point as would be suggested by Figure 3.8.

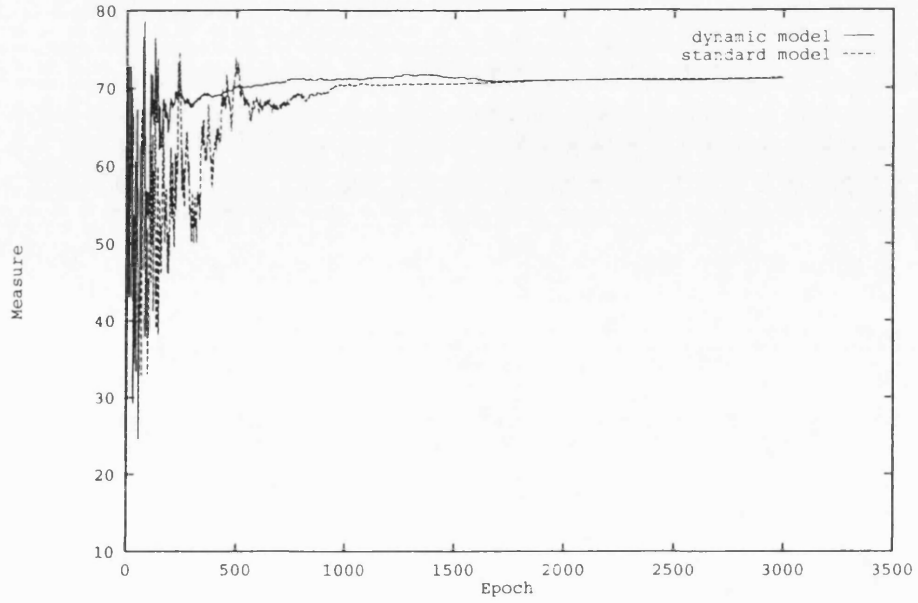
Figures 3.9(a) and (b) show the final weightspace plots for the linear array. It can be seen that both curves fill-out the unit-square in a very similar fashion, but that the fine detail in the standard model is better.

3.9.5 Effect of Varying λ and δ

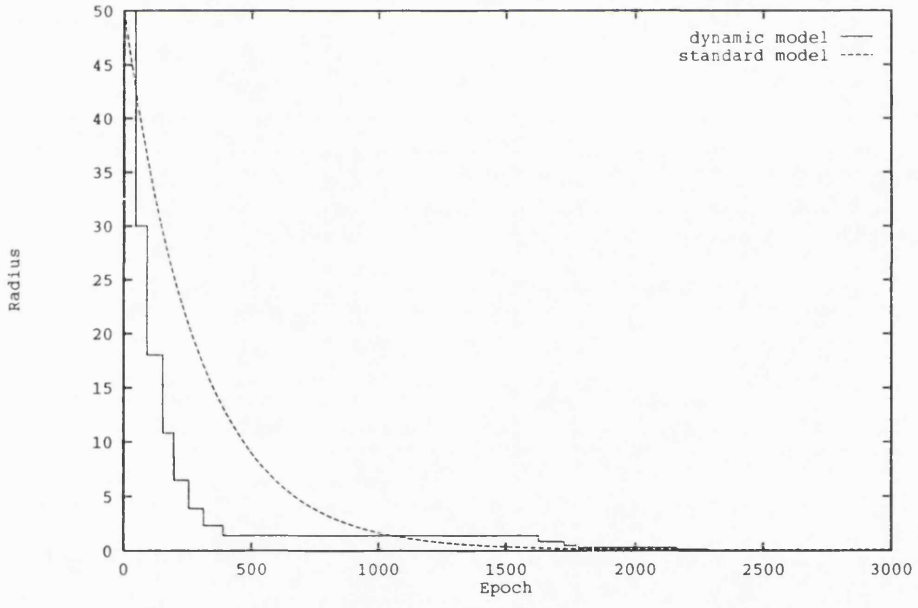
The effect on convergence of varying the parameters λ and δ, for mapping the unit square, was examined. Recall that λ is the parameter reduction factor at each transition and δ is the smoothing constant in the trace X of $\bar{M}$.

Varying λ

The plots in Figure 3.10 show weightspace plots after 1000 epochs for values of λ, the parameter reduction factor at each transition, of 0.1, 0.6 and 0.95. It can be seen that the very low value of λ has caused evolution of the map to 'choke'; the network is dealt a crippling reduction in learning parameters which it cannot recover from



(a)



(b)

Figure 3.8: Evolution of $\bar{M}$ and σ for 1-d unit square simulation for both dynamic and standard models.

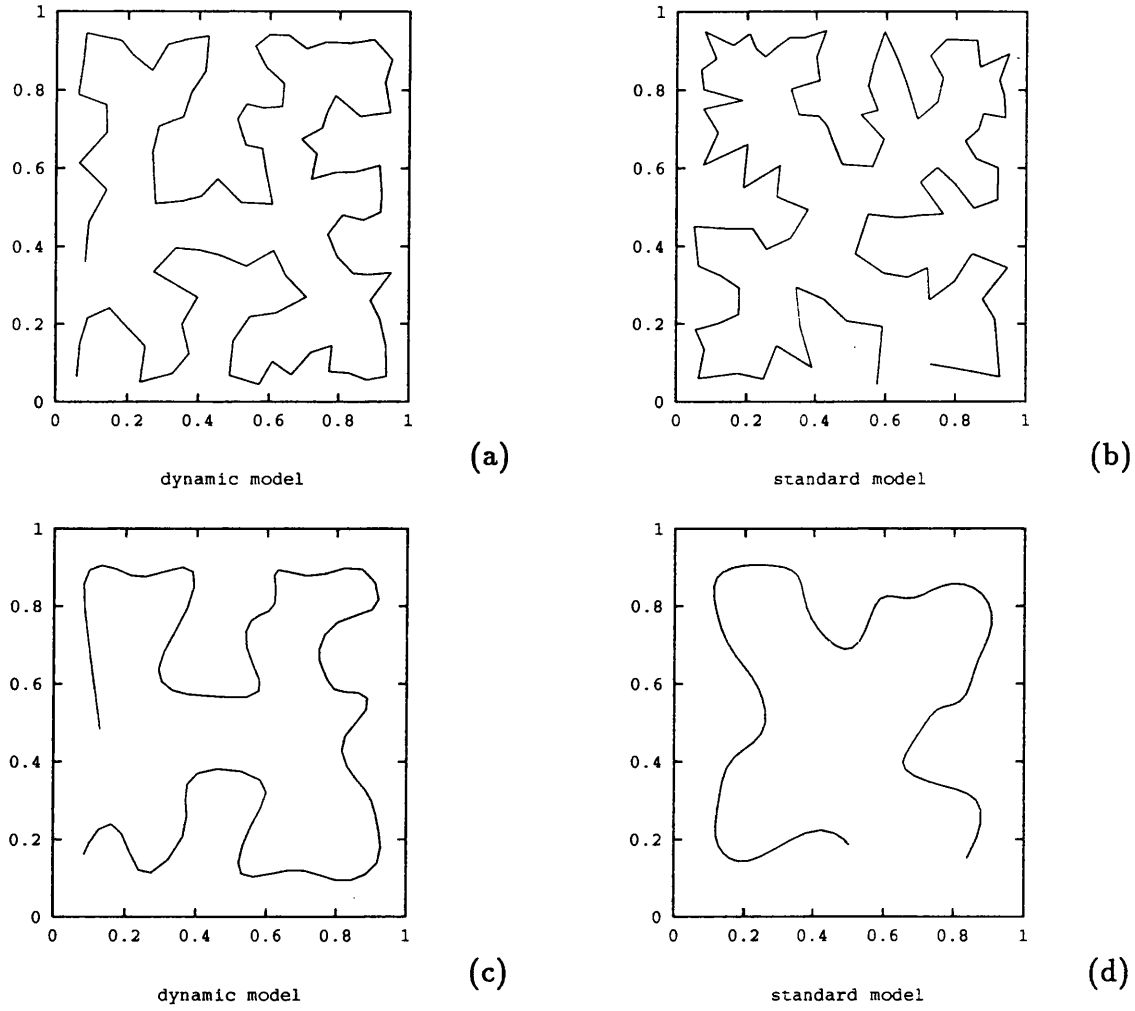
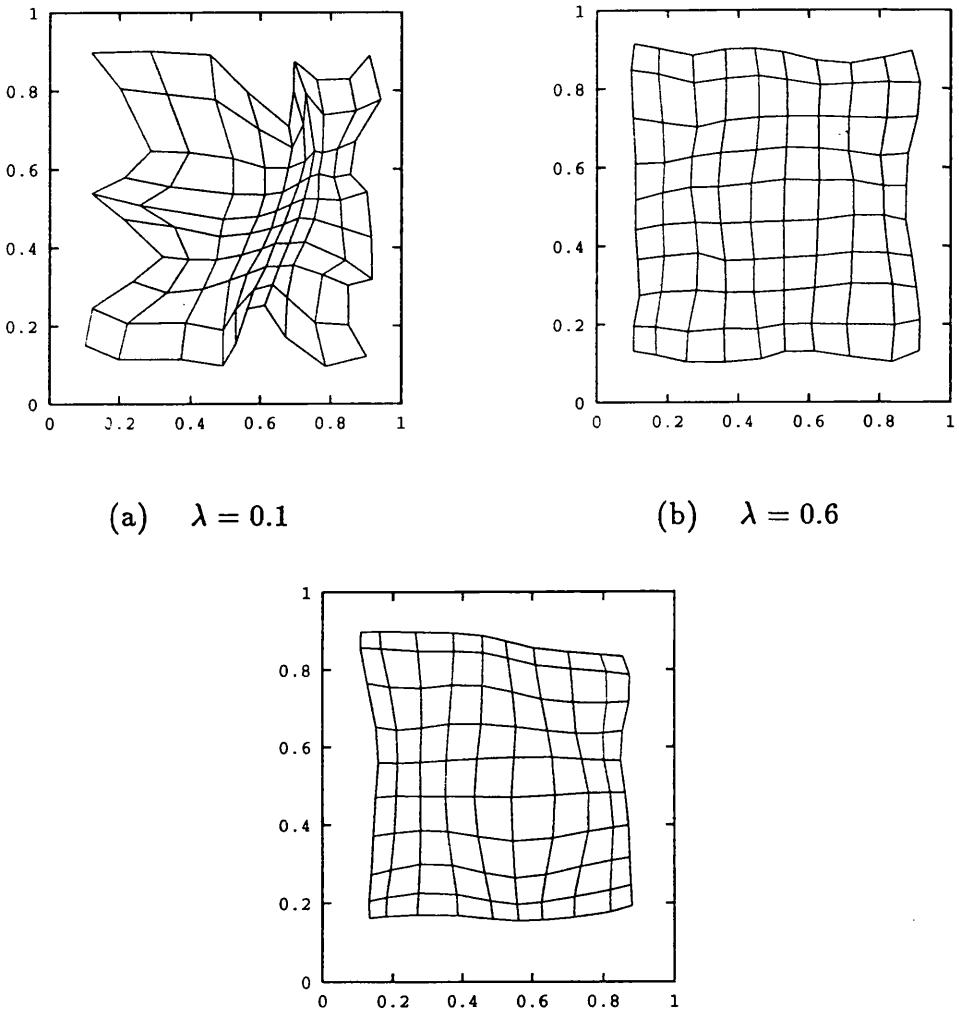


Figure 3.9: Plots (a) and (b) show final weightspace configuration, plots (c) and (d) show snapshots of weightspace at $t = 1000$. (1-d SOM unit square simulation)



(a) $\lambda = 0.1$ (b) $\lambda = 0.6$

(c) $\lambda = 0.95$

Figure 3.10: Effect on final weightspace of different values of λ, the transition decay constant.

and hence the map can not unfold correctly. The other two plots are very similar at this final stage. Earlier in the simulations, the middle value λ simulation was more evolved, the higher value causing more frequent transitions. After 6000 epochs, the two plots were almost identical. In summary, higher extreme values are more robust than low extreme ones, the only effect of a higher λ being a longer convergence time.

Figure 3.11(a) shows the neighbourhood radius timecourse and 3.11(b) shows the evolution of $\bar{M}$ for each of the three values of λ. The medium and high values of λ have radii and values of $\bar{M}$ which are almost identical in the long term, which agrees with their very similar final weightspaces.

Varying δ

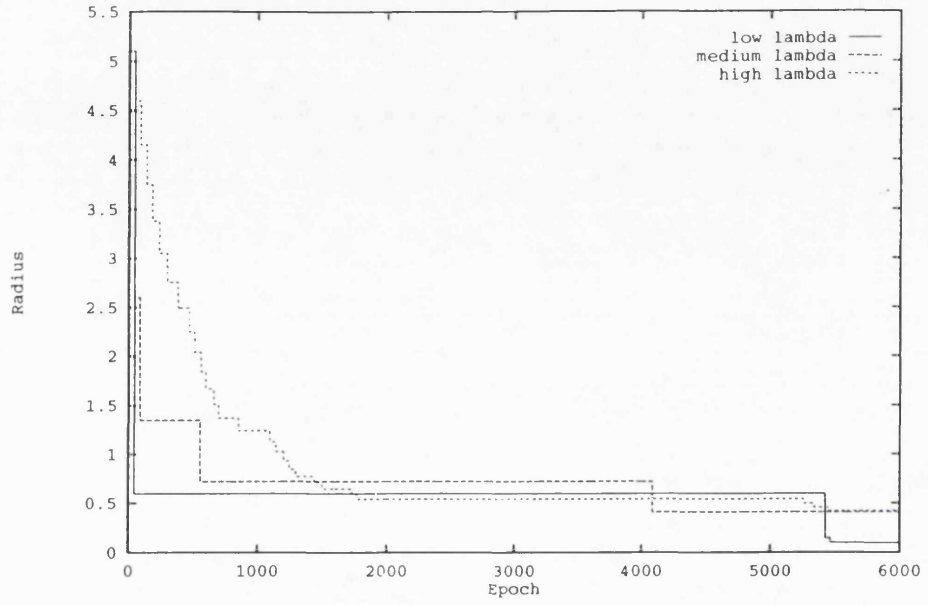
Figure 3.13(a) shows the timecourse for the neighbourhood radius for values of δ, the smoothing parameter, of 0.2,0.8 and 0.95. Figure 3.13(b) shows the value of $\bar{M}$ for each of the three values. It can be seen from (b) that the high value δ spends too long in stable states, and takes much longer to reach $\bar{M}$ values comparable to the other two. The medium δ plot spends just a few extra time-steps in the first few transitions, but it is enough to separate it from the low δ simulation. Figure 3.12 shows the state of the weightspace in each case after 1000 epochs.

The final states after 6000 epochs were almost identical.

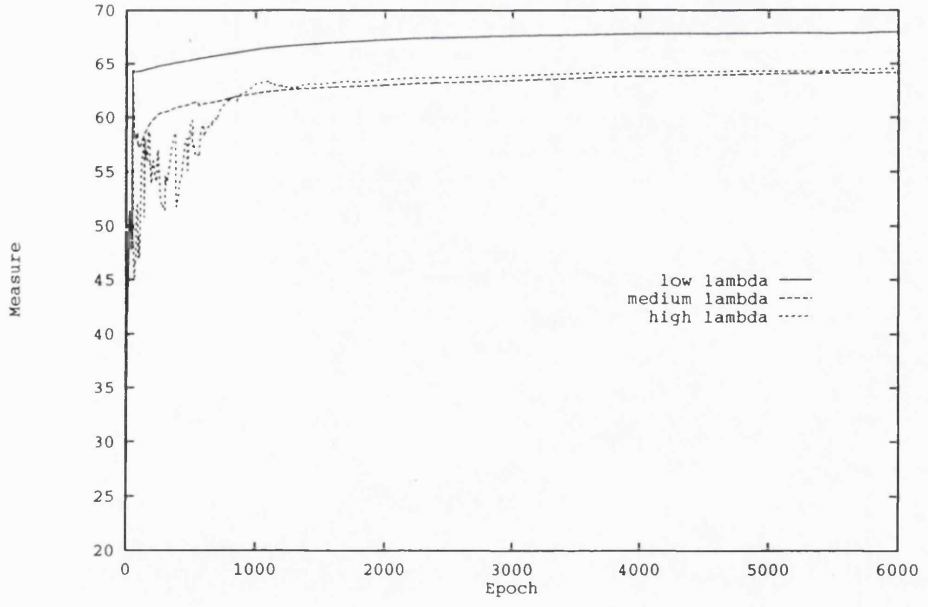
3.9.6 Effect of Different Initial Learning Rate Values ($\alpha(0)$)

For completeness, we now present the effect of different initial learning rates on the adaptive model. We have so far tacitly assumed Kohonen's rule of thumb of 'as big as you like it', meaning values of $\alpha(0)$ that could be up to 0.9 or so.

We now repeat the unit square bench mark using values of $\alpha(0)$ of 0.1,0.35 and 0.9 and again compare the evolution of the smoothed measure X(t), the radius of the gaussian neighbourhood and the final weightspace plots. The map was a 10×10 grid, with $\sigma(0) = 5.0$, $\lambda = 0.6$ and $\delta = 0.5$. The simulations ran for 5000 epochs.

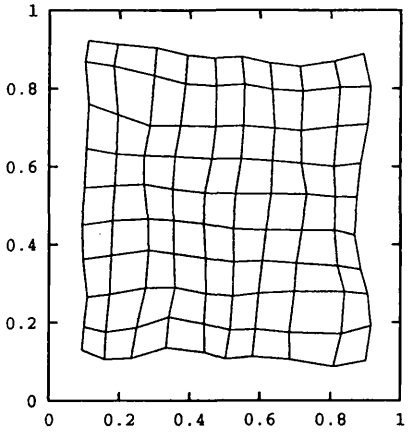


(a)

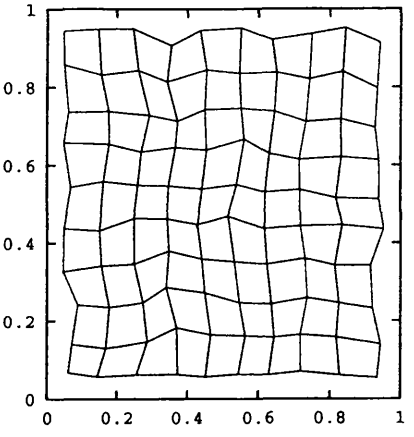


(b)

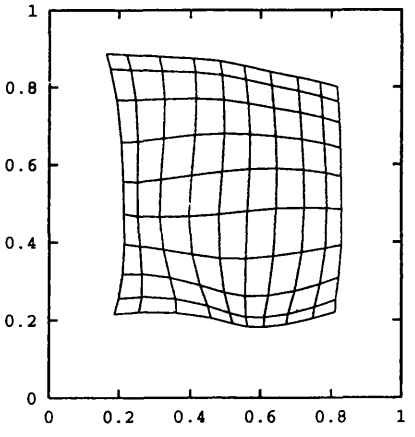
Figure 3.11: Evolution of $\bar{M}$ and σ for different values of λ.



(a) $\delta = 0.2$

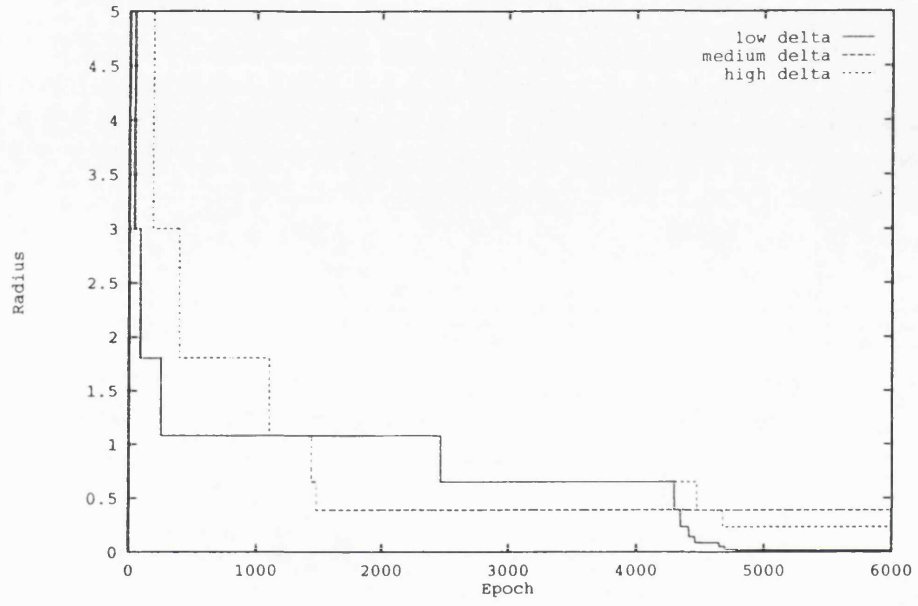


(b) $\delta = 0.8$

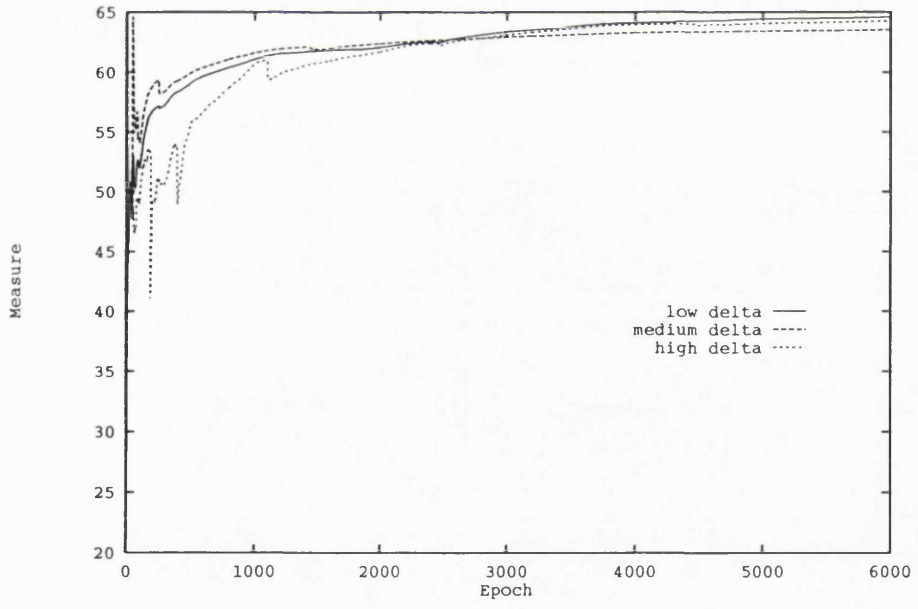


(c) $\delta = 0.95$

Figure 3.12: Weightspace snapshots at $t = 1000$ for different values of δ, the smoothing parameter.

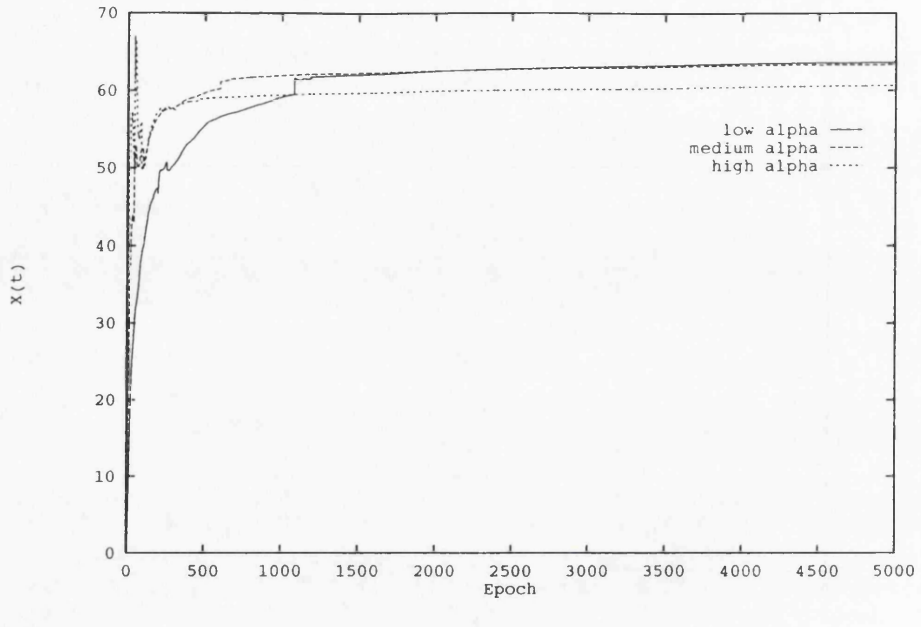


(a)

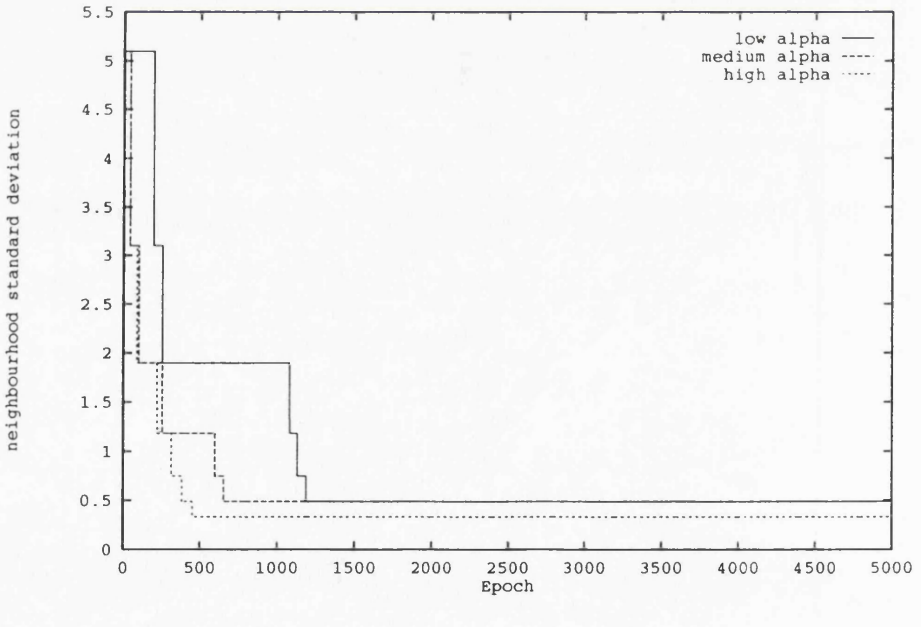


(b)

Figure 3.13: Evolution of σ and $\bar{M}$ for different values of δ.

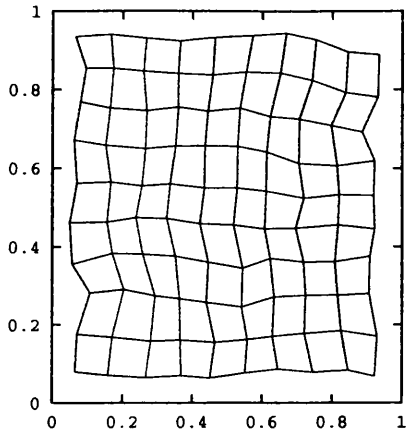


(a)

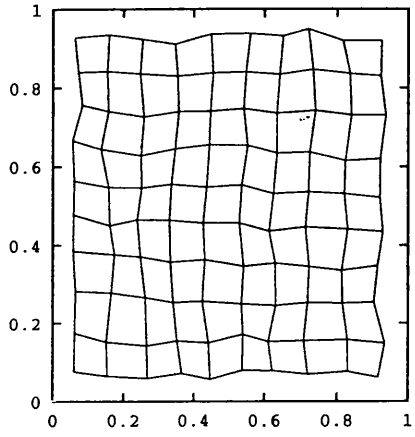


(b)

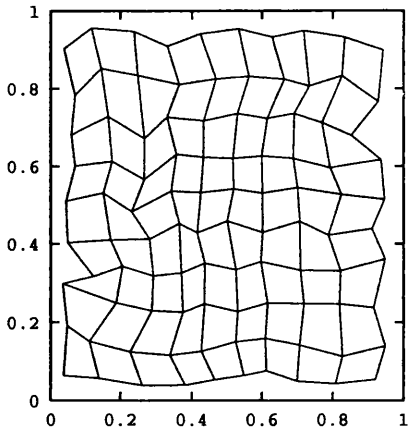
Figure 3.14: Evolution of X and σ for different values of $\alpha(0)$ (2-d SOM unit square simulation).



(a) $\alpha(0) = 0.1$



(b) $\alpha(0) = 0.35$



(c) $\alpha(0) = 0.9$

Figure 3.15: Final weightspace plots for different values of $\alpha(0)$ (2-d SOM unit square simulation).

Figures 3.14(a) and (b) show respectively the evolution of $X(t)$ and $\sigma(t)$ for the three values of alpha and Figures 3.15(a), (b) and (c)show the corresponding final weightspace plot.

From Figure 3.14(a) we can see that the smaller values of $\alpha(0)$ reach a very similar long term value of X although initially the medium value run evolves quicker. This is reflected in the σ plots - the lower value run spends a longer period of time with σ about 1.9. At around epoch 1200, both low and medium valued $\alpha(0)$ value runs have the same σ and their values of X are very similar. The final weightspace plots for these two runs are also very similar, with perhaps the medium valued run having the edge.

In contrast, the run using the high $\alpha(0)$ value has a final weightspace plot that is quite crumpled, although topologically intact. The $X(t)$ plot shows that the system rapidly develops a maximum sized weight vector envelope and makes many transitions to smaller learning parameters to overcome the large α value. However, in reducing α, σ will have shrunk more rapidly than required for a good mapping.

Since we have varied $\alpha(0)$, it is instructive to plot $\alpha(t)$ for the three cases. This is shown in Figure 3.16. It can be seen that all three runs have reached the same value by around epoch 600. Note that the low $\alpha(0)$ run underwent a further series of transitions from about epoch 1100 to epoch 1200. Due to the already small learning rate, the amount of useful learning that could be achieved was apparently very little and so the system made transitions to attempt further improvements.

Summary

In summary, the value $\alpha(0)$ does not appear in practice to drastically affect the mapping ability of the model. However, as might be expected, lower end values do give better results in terms of the quality of the mapping produced.

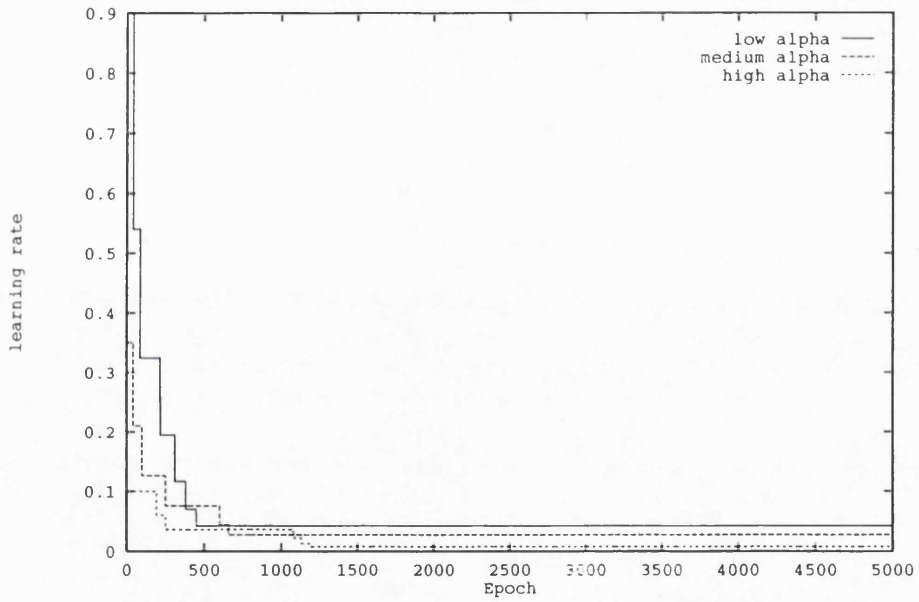


Figure 3.16: Evolution of α for different values of $\alpha(0)$

3.10 An Improved Measure For Identifying Stable States

The average value of the squared weight lengths (call this $\bar{M}_{W^2}$) is a coarse, although still effective, measure of a stable state. It can be prone to very large 'false minima' and hence premature labelling of stable states.

A greatly superior measure is one which would compare the *relative* separation of weight vectors between topologically close nodes on the output layer.

$$\bar{M}_L = \langle \sum_{i=1}^{r-1} \sum_{j=1}^{c-1} \left[(\underline{w}_{ij} - \underline{w}_{ij+1})^2 + (\underline{w}_{ij} - \underline{w}_{i+1j})^2 \right] \rangle$$

where r is the number of rows in the output layer and c is the number of columns i.e. rc nodes in total. Figure 3.17 shows the geometric meaning of the measure: it is the sum of the squares of all adjacent weight vector separations running both along rows and columns (i.e. all contributions like $a^2 + b^2 + c^2$ and $x^2 + y^2 + z^2$). This measure, as was the case for $\bar{M}_{W^2}$, has nothing to say about avoiding *twisted mappings* since such locally (but not globally) optimal maps will still form stable states.

The reason for the greatly improved performance of this measure is straightforward: it is calculating relative distances between weights, and is thus *translation invariant* and *rotation invariant* with respect to the geometry of the output layer. This is of great importance in the early stages of network evolution, when the whole weightspace can be shifted owing to very large initial neighbourhood sizes. Furthermore, it provides an easier way to compare behaviours under different starting conditions for a network, as these will affect the orientation of the topographic mapping (i.e. a left to right mapping can be compared directly with a right to left etc).

Figure 3.18 shows the evolution of the measure for a unit-square learning example. Plot (a) shows the averaged value of the raw measure value in plot (b),

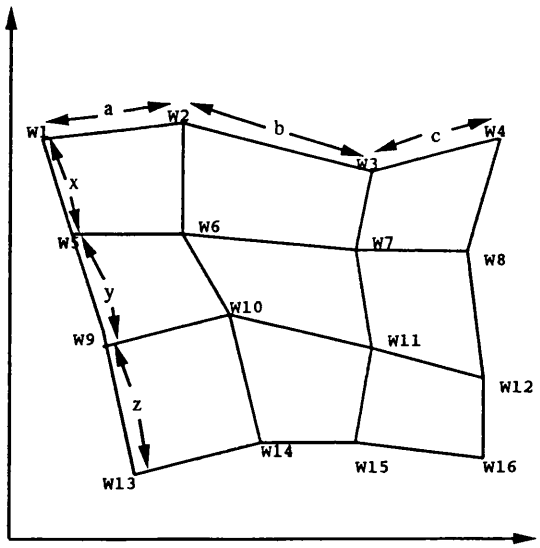


Figure 3.17: Geometric meaning of the new measure

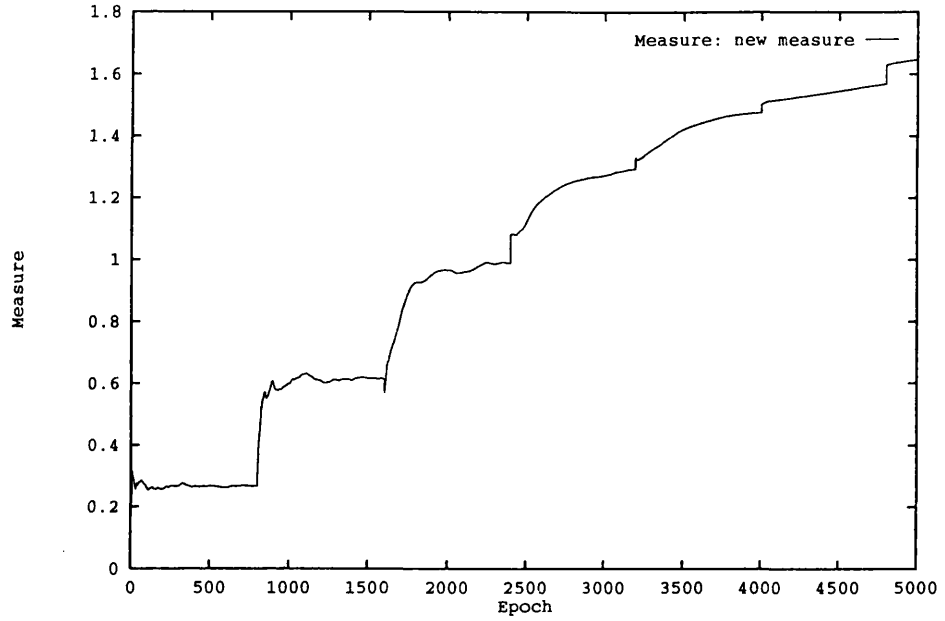
where the average is over the time window since the the start of the last transition. Transitions are externally imposed every 800 epochs.

Note how visually *apparent* the stable states are. This should be compared with the same run, using the old measure (See Figure 3.19(a) and (b)). Here the stable states are much less apparent.

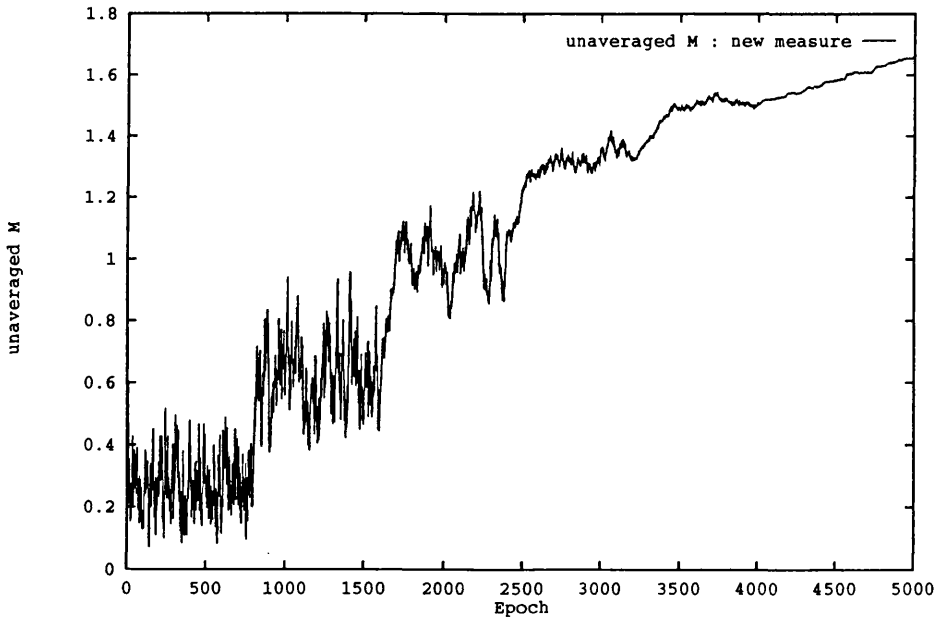
There is a point worth noting about this measure; it is sensitive to widely separated initial weight vectors. The distance between weights in such a case may be very large and will thus affect the initial *average* value of the measure ie. during the ordering phase of the map. This shortfall can easily be overcome by ensuring that the initial weights are closely clustered, even though still random; this is in any case the usually recommended practice [37].

3.10.1 A More Practical Example

The above simulations show the dynamic method working on very simple benchmark training data. However, if it is to be of general use, then it should also be



(a)



(b)

Figure 3.18: Plots of $\bar{M}$ and X for the new measure $\bar{M}_L$ (forced transitions).

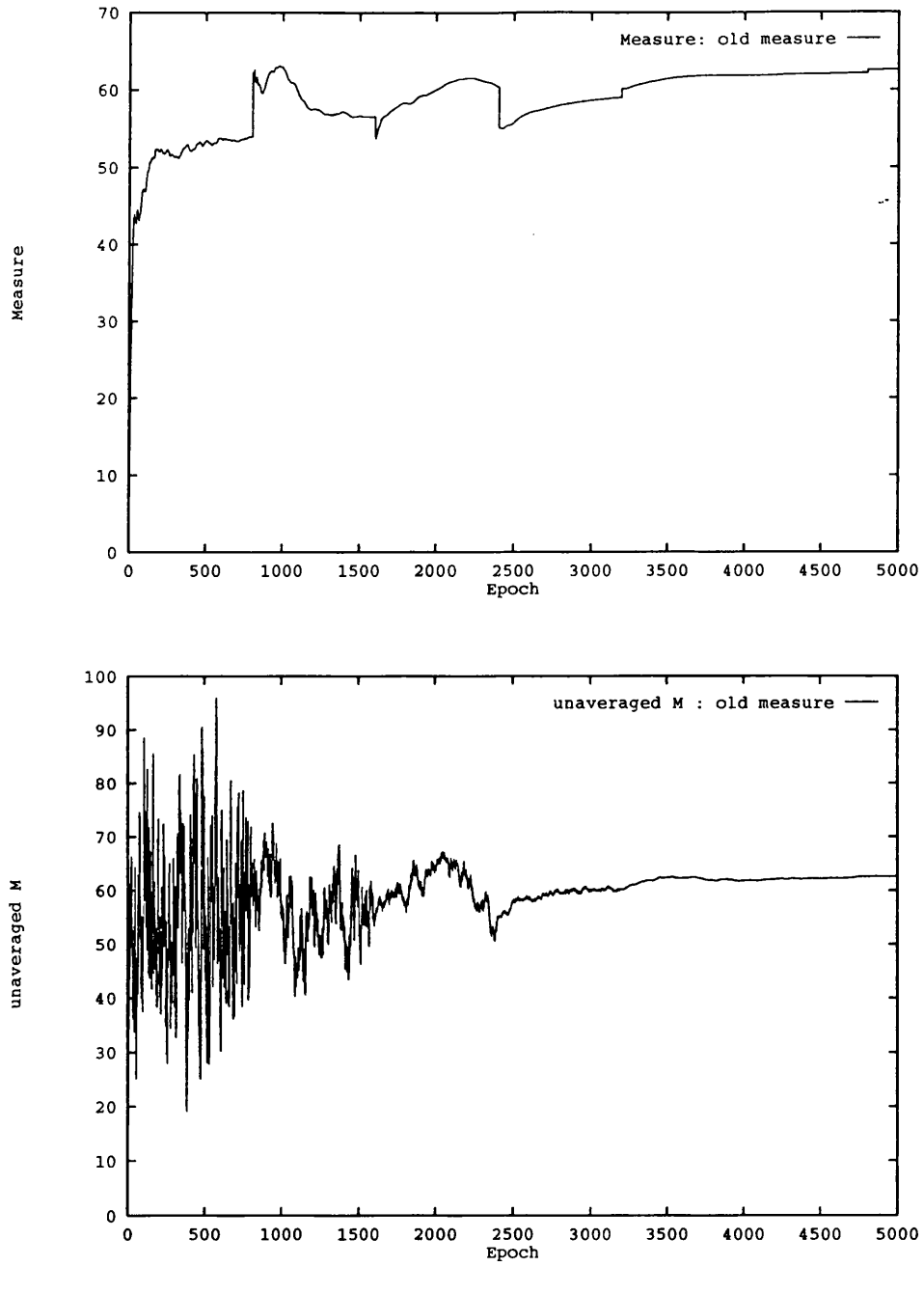


Figure 3.19: Plots of $\bar{M}$ and X for the old measure $\bar{M}_{W^2}$ (forced transitions).

applicable to mapping high-dimensional data onto a 2-d output grid.

In this final simulation, an example is taken from Kohonen's book [37], where the map generates a topographical representation of a minimal spanning tree. The data used is the same artificial set used by Kohonen but is a clear example of the map in operation.

The training data consists of a set of 32 5-d vectors. They are given the labels A,B,C,...,Y,Z,1,2,...,6:

(1,0,0,0,0)	A	(3,4,0,0,0)	I	(3,3,7,0,0)	Q	(3,3,6,3,0)	Y
(2,0,0,0,0)	B	(3,5,0,0,0)	J	(3,3,8,0,0)	R	(3,3,6,4,0)	Z
(3,0,0,0,0)	C	(3,3,1,0,0)	K	(3,3,3,1,0)	S	(3,3,6,2,1)	1
(4,0,0,0,0)	D	(3,3,2,0,0)	L	(3,3,3,2,0)	T	(3,3,6,2,2)	2
(5,0,0,0,0)	E	(3,3,3,0,0)	M	(3,3,3,3,0)	U	(3,3,6,2,3)	3
(3,1,0,0,0)	F	(3,3,4,0,0)	N	(3,3,3,4,0)	V	(3,3,6,2,4)	4
(3,2,0,0,0)	G	(3,3,5,0,0)	O	(3,3,6,1,0)	W	(3,3,6,2,5)	5
(3,3,0,0,0)	H	(3,3,6,0,0)	P	(3,3,6,2,0)	X	(3,3,6,2,6)	6

As described in [37], if a hierarchical cluster analysis of the data above was performed, the *minimal spanning tree* of Figure 3.20 would result.

Simulations were carried out on the above data for exponential decay of parameters and for the dynamic scheme. In all cases, the output grid was 10 by 7 neurons. The learning rate and neighbourhood size were $\alpha(0) = 0.35$, $\sigma(0) = 5.0$. The simulations were allowed to run for 4000 epochs.

For the dynamic model, the other parameters were $\delta = 0.8$ and $\lambda = 0.7$ and for the standard model, $T_{1/2} = 1200$

Figures 3.22 and 3.23 show the labelled state of the output layer for the standard model and the dynamic model respectively. At the end of the training session, each label was assigned to the unit that generated maximum response to it. It can be seen that the basic topographical relations of the tree in Figure 3.20 have been preserved. The branches of the tree, of course, are not straight.

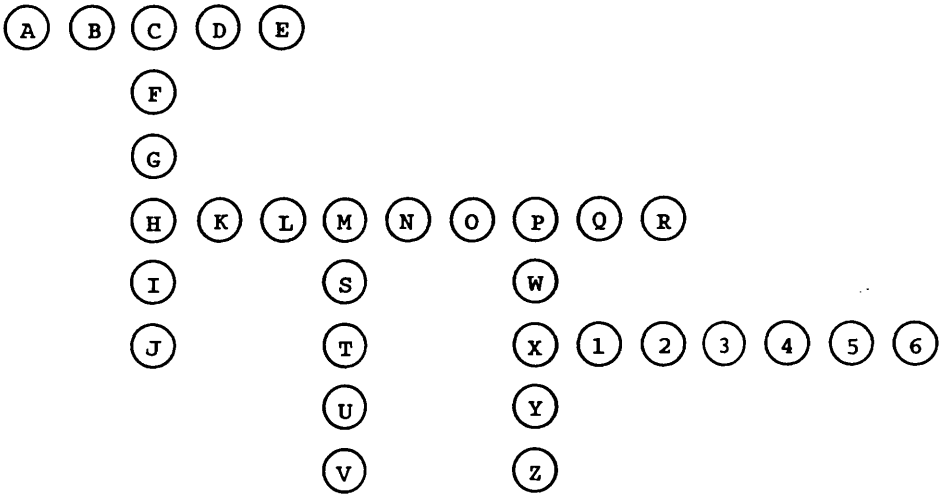


Figure 3.20: Minimal spanning tree of simulation data.

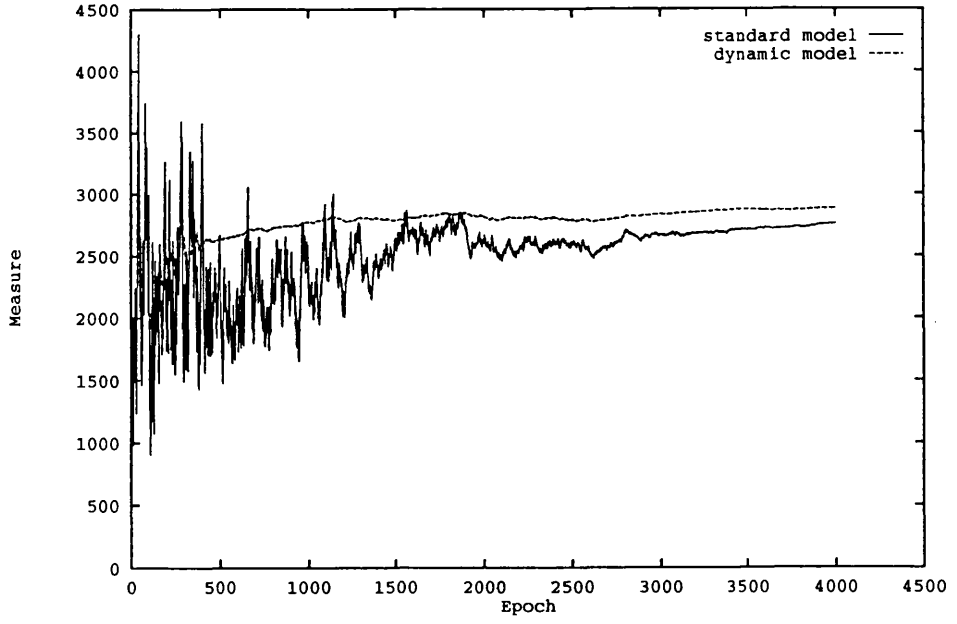


Figure 3.21: Evolution of $\bar{M}$ for standard and dynamic models (hierarchical data simulation).

Z	Y	#	R	Q	#	#	A	B	#
#	#	X	#	P	O	#	#	C	D
#	1	#	W	#	N	#	#	#	E
#	2	#	#	#	#	M	#	F	#
#	3	#	#	#	S	#	L	#	G
#	4	#	#	#	#	#	K	#	H
6	5	#	V	U	T	#	#	I	J

Figure 3.22: Labelled output layer for standard model.

6	5	#	R	#	O	N	M	#	J
#	4	#	Q	#	#	#	L	#	I
3	#	#	#	P	#	#	#	K	H
#	2	#	W	#	#	S	#	G	#
#	1	#	#	#	T	#	#	F	#
#	#	X	#	U	#	#	#	C	B
Z	Y	#	#	V	#	E	D	#	A

Figure 3.23: Labelled output layer for dynamic model.

X	Z	#	#	V	#	#	#	#	A
#	#	#	#	#	#	#	#	#	#
1	#	#	#	S	#	#	#	#	#
2	#	#	#	#	#	#	#	#	G
#	#	#	#	#	#	#	#	#	H
3	#	#	#	#	#	#	#	#	K
6	#	#	#	#	N	M	#	L	J

Figure 3.24: Labelled snapshot at $t = 1000$ for standard model.

6	#	#	R	#	O	N	#	#	J
5	4	#	#	P	#	#	#	#	I
#	3	#	W	#	#	M	L	K	H
#	2	#	#	#	S	#	#	#	G
#	1	#	#	T	#	#	#	F	#
#	#	X	#	U	#	#	#	#	B
Z	Y	#	#	V	#	#	E	C	A

Figure 3.25: Labelled snapshot at $t = 1000$ for dynamic model

Figure 3.21 shows $\bar{M}$ vs Epoch for the simple and dynamic case. The upper curve is the dynamic model. This would suggest that the dynamic model converges quicker in this example.

We can give a qualitative comparison of convergence speed by looking at the output clustering early on in training and see how this compares with the measure $\bar{M}$ being employed to monitor performance.

Figures 3.24 and 3.25 show the labelling results for the standard and dynamic model respectively. The snapshot was taken after 1000 epochs of training. The dynamic model clearly shows more clustering structure at this point in the simulation

and also the value of M is *larger* and *more stable.* This suggests that $\bar{M}$ is indeed a good measure of convergence.

3.10.2 Extensions and Improvements : Further Work On The Dynamic Model

It has been seen that the new dynamic model copes well in a variety of scenarios. The formalism could be improved by looking for a dynamic control of the *magnitude* of a parameter change at a transition i.e a control of the parameter α in Equation 3.7. A likely candidate for this is the *variance* of the measure $\bar{M}$ during a stable state. Clearly this variance diminishes as the system converges.

3.11 Other Dynamic Variants On The SOM

A number of other schemes have been introduced which aim to improve performance of the SOM in some respect, whether it be convergence rate or the way in which the map represents the input probability distribution.

3.11.1 A Novel Approach To Improving Learning Speed

An interesting attack on the convergence speed problem is presented by Rodrigues and Almeida [63] The approach is based on the idea of starting the map with just a few nodes and then progressively increasing that number until the map reaches its final state. New units are added by interpolation of the weights of the old units (See Figure 3.26).

The method works by the initial ordering and unfolding stage being handled by just a few neurons. Only a rough gauge of the input space is required at this point. There is clearly a major saving in time both from calculating similarity between input vector and weight for each node and from the searching for the winning node. In the early stages, as there are only a few nodes to check.

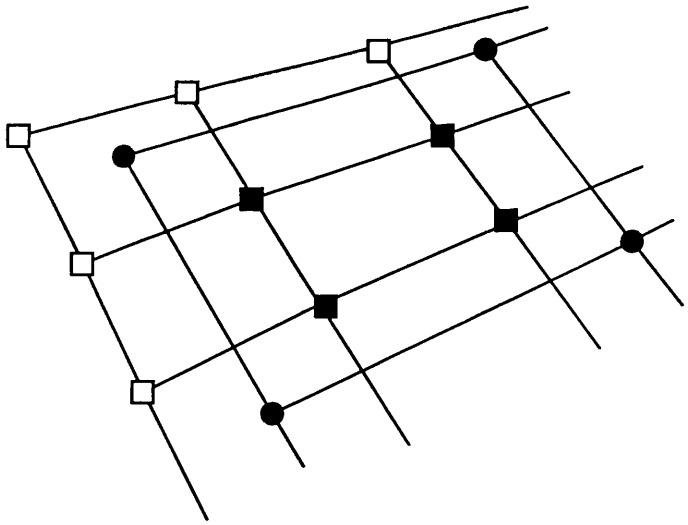


Figure 3.26: Adding new units by interpolating the weights of the old units.

The paradigm also uses a *constant neighbourhood size.* The reason for this is that when the network has only a few nodes, the initial chosen neighbourhood is large in comparison with its dimension. The neighbourhood can thus unfold the map. As the number of nodes is increased, the relative scale of neighbourhood size to net size reduces, thus effectively shrinking the neighbourhood, but from a different perspective.

Rodrigues and Almeida report an order of magnitude improvement in CPU time required to run simulations, with particularly large improvements for big networks (e.g. 384x384).

3.11.2 Dynamic MST Neighbourhoods.

Dynamically defined neighbourhoods based on Minimal Spanning Tree (MST) topologies represent a new approach to the mapping of probability density functions in the input space that have a prominent, peaked regions [30]. In this scheme, nodes are assigned to a particular neighbourhood by defining MST arcs between nodes, so that all nodes in the neighbourhood are connected by single links and the sum of

the lengths of these arcs is minimized. These 'lengths' are the Euclidean distance between the weight vectors of the nodes in the tree.

Neighbourhoods are initially large, which corresponds to following through more arcs off the winning node. These are then shrunk as per the original algorithm, by traversing less arcs. It is not necessary to recalculate the MST neighbourhood after each epoch, only after between 10 and 100 epochs. This is because the map has temporally smooth adaptation.

MST neighbourhoods have the advantage of allocating nodes to non-zero regions of probability density in the input space. This is however not so good for more symmetric distributions. They can also adapt to dynamic changes in the input distribution with remarkable flexibility [30].

This scheme does not however address the issue of the rate of decrease of learning parameters. The dynamicism is employed to allocate the nodes to better follow the input distribution.

3.11.3 Adaptive, Tensorial Weighting

This method, also presented in [30], dynamically modifies the distance function used to determine the winner. The impetus for this extended model is the fact that when the variances of the components of the input vector $\underline{x} = \underline{x}(t)$ are not of the same order of magnitude, then the resultant mapping has an oblique orientation (See Figure 3.27); there needs to be scaling of the different components.

A better orientation can be achieved by introducing the following weighted Euclidean distance measure into the winner competition:

$$d^2{}_w\,[\underline{x}(t), \underline{m}_i(t)] = \sum_{j=1}^{N} \omega_{ij}^2\,[\xi_j(t) - \mu_{ij}(t)]^2 \tag{3.8}$$

Here, the ξ_i are the components of $\underline{x}$, the μ_{ij} are the components of the $\underline{m}_i$ (the weight vectors) and the ω_{ij} are the component weightings from input line j to node i. Note that *each synapse* has an associated weighting factor.

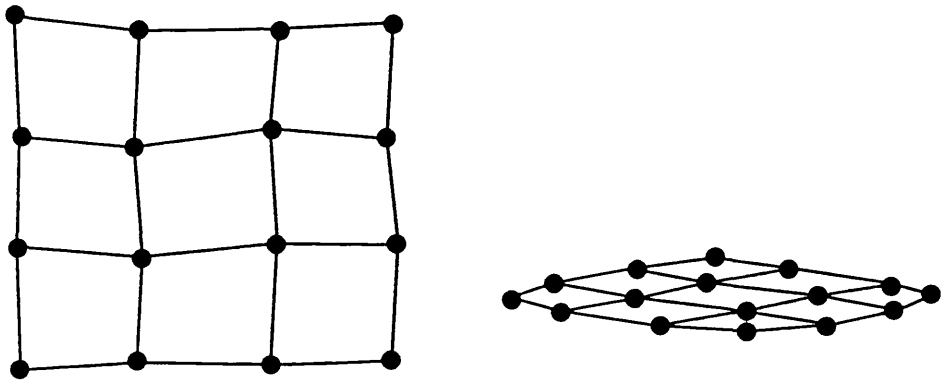


Figure 3.27: Final weightspace plots for two 4×4 networks with different variances in input dimensions

The main thrust of the method is to estimate the values of ω_{ij} recursively i.e. on the fly during the learning process such that the effects of variance disparity are balanced. Each node is thus made to store δ_{ij}, a backwards weighted exponential measure of the *error* defined as $|\xi_j(t) - \mu_{ij}(t)|$:

$$\delta_{ij}(t+1) = (1 - \kappa_1)\delta_{ij}(t) + \kappa_1 \omega_{ij} \, |\xi_j(t) - \mu_{ij}(t)| \tag{3.9}$$

where κ_1 is a small scalar.

The value of δ_{ij} is then averaged over all the inputs j to a node and the system is made to maintain the same average level of weighted errors over all the inputs.

A geometric interpretation of this weighting procedure is given in [30]. The equidistant surface around a particular node then becomes elliptic in N dimensions.

The scheme has been shown to work well in practice.

3.11.4 Growing Cell-Structures

Fritzke has introduced the notion of self-organising maps that have problem dependent cell structure [17][18]. In the extended model, the system can dynamically follow an input signal distribution more accurately by inserting or deleting nodes

from its structure. The system starts with a small number of cells, new cells are added successively, using dynamically gathered information about the underlying probability distribution to determine *where* additions/deletions should take place. The modifications are arranged such that the topology-preserving and distribution-preserving properties of the self-organising map are retained.

3.12 Summary of Chapter 3

We have briefly reviewed the "rules of thumb" that define the parameters of the Kohonen self-organising map, including some theoretical bounds on the learning rate and also experimental results on the effect of the neighbourhood function profile.

We recapitulated the standard form of learning in the SOM, whereby the time-course of learning parameter decay is set arbitrarily in advance. We then presented evidence from simulations that the map forms stable weightspace configurations when the learning parameters are held constant and the network is driven by input vectors presented randomly and according to the input vector probability density function. An experimental gauge of these stable states, the sum of the norms of all the weights was discussed, together with how this measure could be used to trigger a reduction in learning parameters. This would produce a sequence of states, experimentally observed to be stable, each state correlated with a set of learning parameters. This sequence of states was seen to converge towards an optimal solution. Three sets of simulations were then presented to show the new model in operation. These showed that the new model performed as well as or better than the original fixed timecourse model with regard to convergence rate. An experimental investigation of the dependence of model behaviour on model parameters was then presented.

An improved measure for identifying stable states, based on weight vector separations of topologically adjacent nodes, was presented and compared by simulation to the original. This new measure was shown to be a much better indicator of the

presence of stable states.

We concluded the chapter by reviewing some other dynamic models based on the basic Kohonen model.

Chapter 4

Temporal Kohonen Maps

4.1 Representing Time In Neural Networks

All neural networks are to some extent *temporal* in operation, ie there are identifiable steps in the computation procedure that lead on to the next. *Temporal Neural Networks* deal with inputs that are explicit functions of time and so thus process *temporal sequences* of patterns.

The 'traditional' approach to processing temporal patterns in neural networks has followed one of two routes:

- Combining a sequence of spatial patterns by *concatenating* them, so called *time delay* networks.
- Using recurrent networks i.e. feedback

4.1.1 Time-Delay Networks

A time-delay neural network (TDNN) works on the principle of representing a temporal sequence by concatenating the individual patterns of a sequence at times $t = 1, 2, ..T$ into a larger one [71][70]. Figure 4.1 shows a generalized architecture for the input layer of a TDNN.

The time-delay aspect comes into effect as patterns are arranged to have different transmission velocities along the connections to the separate parts of the input layer's output vector. Hence at $t = 0$ the section of this concatenated vector with zero transmission time has the current pattern vector as its output. At the next time step, the old pattern has propagated to the section of the vector which has transmission time of 1 time unit and the zero transmission section has the new pattern vector etc etc. In this way, an input layer's output vector comprised of N blocks of n units can hold a complete record of an n-dimensional temporal input sequence over the past N time steps.

The idea is then to classify this concatenated vector in some standard way, i.e. as a *static* pattern.

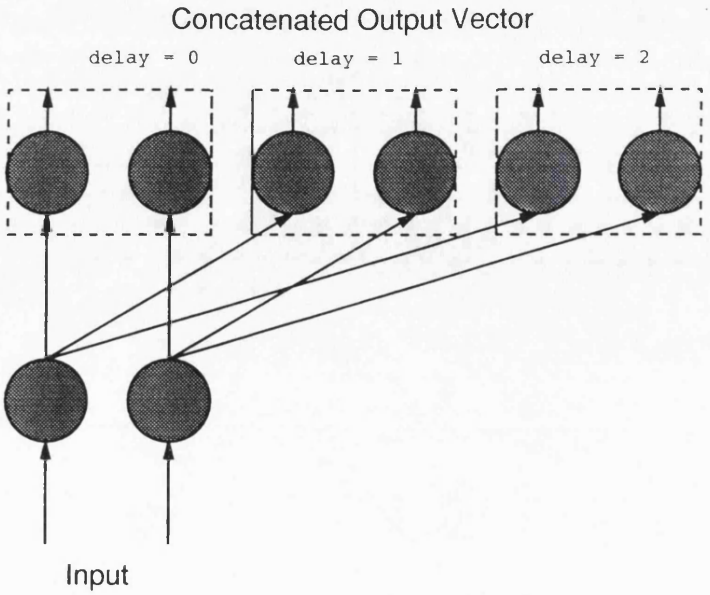


Figure 4.1: Input layer of a time-delay network.

The major drawback of this scheme is that the *architecture* has to change for different lengths of sequence. Here, the time-window manifests itself as the number of delay lines needed in the preprocessing stage. It is thus inflexible and biologically unrealistic in its operation. This form of network will also suffer from *temporal translation problems* i.e. recognition should be independent of *when* the pattern occurs in time. Scaling is also a problem in that concatenating a large number of pattern vectors will make the introduction of temporal translations *during* a pattern sequence very significant to performance.

In [71] the temporal translation problem was addressed. A network was constructed with TDNN-units that scan an input token over time in order to find *local* clues. This is as opposed to one large-network being presented with the whole input pattern. In [71] this is achieved by multiple hidden layers. The first layer concatenates three out of fifteen *frames*. A five frame window then combines the outputs of the first layer. This method forces the hidden units to develop short term ab-

stractions, but is heavily dependent on the architecture being *clocked* i.e. the first hidden layer only classifies after filling a three time step shift-register, the second after filling a five time-step shift-register etc.

4.2 Recurrent Networks

4.2.1 Why Recurrent Networks?

Networks which have recurrent connections, together with suitable learning algorithms, can implement dynamical systems of arbitrary complexity (given a sufficient number of hidden units). Such networks have important capabilities that are not found in feedforward networks. These include:

- Attractor dynamics.
- The ability to store information for later use i.e. there is *state preservation* in some form.

The latter property is vital in difficult temporal tasks that may require state preservation over potentially unbounded periods of time.

Figure 4.2 shows a general architecture for a recurrent backpropagation network. The network has a set of N nodes, n of which comprise the output vector $\underline{y}'$. The remaining $m = N - n$ nodes are *hidden* that is they are not part of the external output, but are used in forming internal representations. Call the full set of node states $\underline{y}$ and call the external input vector $\underline{x}$. Then the concatenated training input that the network sees is $\underline{z} = (\underline{y}, \underline{x})$

4.2.2 Fixed Point Networks

A number of studies of networks which settle to stable states or *fixed points* have been studied e.g. Hopfield's model [26], but the majority have been extensions of the *recurrent backpropagation model* developed independently by Pineda and

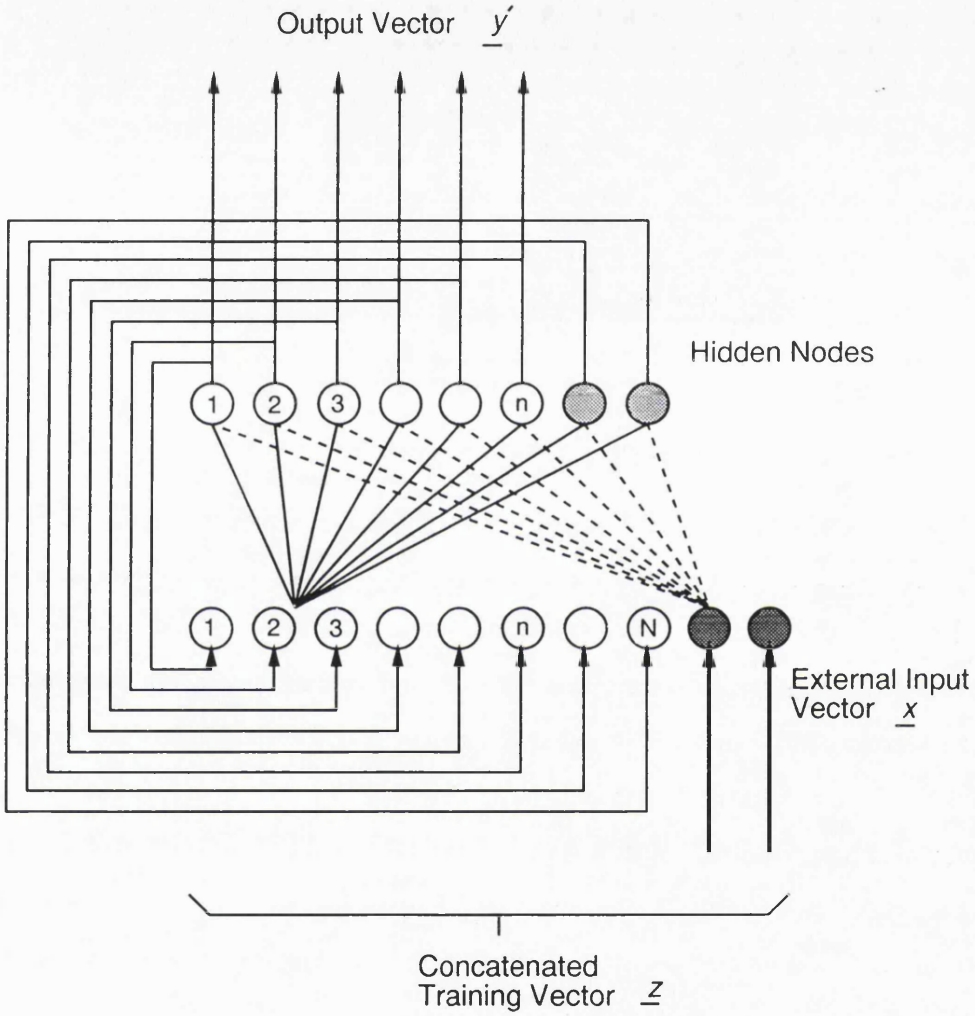


Figure 4.2: Recurrent back-propagation architecture.

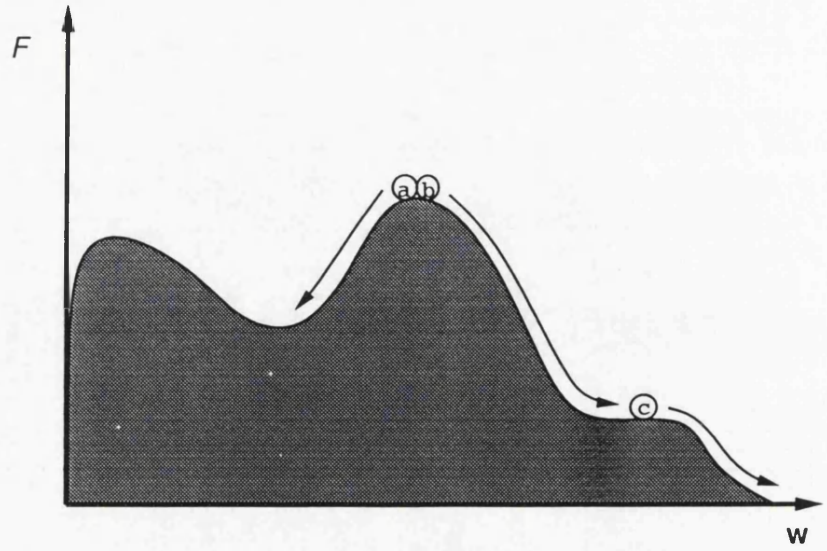


Figure 4.3: Energy landscape in a network with fixed points.

Almeida[54][2]. For such 'fixed point' networks, a particular problem is given to the network in the form of initial conditions or a constant external input and the the result is defined by the state of the network once a fixed point has been reached [52]. The learning algorithm in such cases is a rule or dynamical equation which changes the fixed points to encode information [54] i.e. perform gradient descent on some suitable energy function and have patterns represented by minima of this function. These networks are interesting in that they can solve classes of problems like constraint satisfaction and associative memory tasks. However, the requirement that both the actual and desired network dynamics have only *point attractors* and that the form of the input is as described above, puts severe limitations on the practical uses of such networks.

The problem with fixed points is discussed lucidly by Pearlmutter [52]. Figure 4.3 shows a schematic energy function $\mathcal{F} = \mathcal{F}(\mathbf{w})$, where $\mathbf{w}$ is the weight matrix of the system. It is possible for initial weight conditions $\mathbf{a}$ and $\mathbf{b}$ to be infinitesimally close but still map to different fixed points. Similarly, boundaries between two different

attractors may be changed by an infinitesimal change to the weights during learning. The point **c** (See diagram) may under similar circumstances change from being a fixed to a non-fixed point.

Such networks can thus run into serious problems when faced with generalisation.

Pearlmutter has developed extensions of recurrent back-propagation that can learn state space trajectories [53]. The network learns to minimize an energy function $E(\underline{y}) = \int_{t_0}^{t_1}(\underline{y}(t) - \underline{f}(t))^2\,dt$ and thus $\underline{y}$ imitates the function $\underline{f}$. Such networks can learn to replicate circular and figure-of-eight 2-d orbits as well as the more mundane exclusive-or type benchmark examples.

4.2.3 Real-Time Recurrent Learning

In most work with recurrent networks, the weights are assumed to be fixed over presentation of the temporal pattern $\underline{x}$ and generation of the corresponding output sequence $\underline{y}$ and contributions to $\nabla\mathcal{F}(\underline{w})$ are integrated over the duration of the sequence $\underline{x}$. This condition can be relaxed, in a similar fashion to non-recurrent backpropagation, so that weight changes are made as the network is running. This removes the constraint of running 'batches' over the duration of the sequence $\underline{x}$. This scheme is called *real-time recurrent learning* [75]. There is the usual requirement that the learning rate must be sufficiently small so that errors introduced by not following the true gradient are kept small.

4.2.4 Teacher-Forced Real-Time Recurrent Learning

Williams and Zipser [75] describe the process of replacing the feedback of the actual output $\underline{y}_k(t)$ by the teacher signal $\underline{d}_k(t)$, whenever it is known, as *teacher forcing.* The correct value of $\underline{y}_k(t)$ is then used in subsequent computation of network behaviour. This technique can only be applied in a discrete time formalism. Changing the state of an output unit at (potentially) each time step only makes sense under this restriction [52]. However, Pearlmutter has reported that teacher forcing with

large numbers of hidden units has caused difficulties [52].

Williams and Zipser have performed interesting simulations using real-time recurrent learning [74]. Theses include complex tasks such as:

- **Learning to Be a Turing Machine** The network observed the actions (but not the internal state) of a finite state controller of a Turing Machine that had to decide whether an arbitrary length tape of left and right parentheses was balanced. This demonstrates very strikingly the preservation of state information over very long periods.

- **Learning to Oscillate** This included training a 2-unit net so that one of the units produced approximately sinusoidal oscillations of a period on the order of 25 time steps.

Clearly, recurrent networks are very powerful, drawing on their ability to preserve state information and to deal with time-varying input or output through their own natural temporal operation.

4.3 Adapting The Self-Organising Map To Temporal Problems

4.3.1 The Phonetic Typewriter

The phonetic typewriter was discussed in Chapter 2. It is an application of the self-organising map, which in its original form classifies *static* patterns, to a temporal domain where context is crucial i.e. speech recognition. The input uses sampling over fixed temporal windows, so that this a form of time-delay neural network.

Kohonen got around the context problem by using a rule-based system to correct errors of statically classified speech data [32][33]. However, this approach requires in excess of 15000 rules to work correctly.

4.3.2 Allographotopic Maps

Morasso has applied the self-organising map to the problem of cursive script handwriting recognition [48]. The problem domain is represented by a set of three continuous variables: $x = x(t), y = y(t), z = z(t)$ with (x, y) being the coordinates of the path followed by the pen and z is the pen pressure (a binary up/down signal). The signals are segmented into 'strokes' i.e. pen traces delimited by points of minimum velocity, with each stroke being represented by a 5-point polygonal curve. A sample of cursive script writing is then coded into a set of ten-dimensional 'stroke descriptors' which are self-organised to form a *graphotopic map.*

As per speech recognition, this problem domain has context dependent features, arising from the physical movements that the pen has to make to link up with previous and following strokes. Morasso approached this problem by using an array of separate maps, one for each of a possible set of sequence lengths e.g. 'receptive fields' between 2 and 7 strokes [49]. These are *Allographotopic maps.* The original graphotopic map used only single strokes. A segmentation module separates allographic patterns and sends them to the appropriate network for that sequence length. During recognition, there is competition between the arrays of maps in order to build a tree of feasible segmentations/interpretations from which a list of the best matches is matches is extracted. At this point, linguistic post-processing takes over to select the overall winner eg by ruling out nonsense words.

Using arrays of maps in this way does introduce a limited notion of context, but simply on the grounds of the number of patterns in a sequence and the segmentation is performed in supervised (even user-interactive) way. Furthermore, the system still needs substantial post-processing to determine the correct result, as per the phonetic typewriter.

4.3.3 Kohonen's Hypermap Architecture

Another approach to solving temporal problems has been introduced by Koho-

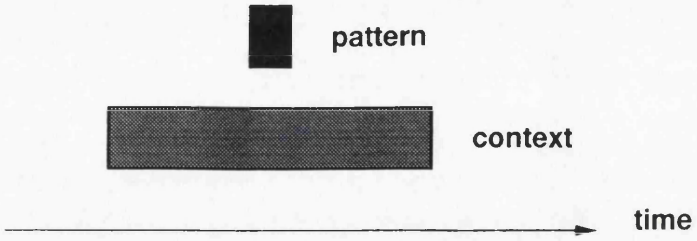


Figure 4.4: Context timescale in the hypermap.

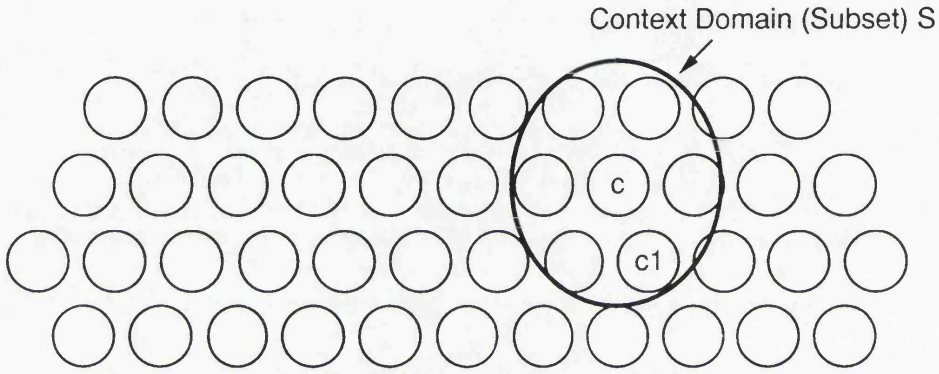


Figure 4.5: Activated subset of nodes in the hypermap.

nen, called the hypermap architecture [38]. The idea revolves around a two phase classification procedure and two sets of weights, a *context vector* $\underline{x}_{cont}$ and *pattern vector* $\underline{x}_{patt}$:

- The system is presented with $\underline{x}_{cont}$. This is formed from the concatenation of pattern vectors occurring within the same timescale as the pattern vector (See Figure 4.4). The network produces an *activated subset* of nodes which responded maximally within some constraint i.e. nodes corresponding to some certain *class* of context remain activated (See Figure 4.5). All other nodes are *de-activated.*

- $\underline{x}_{patt}$ is then presented and only the activated subset from the first phase can

take place in the competition. This subset thus represents the set of those nodes whose pattern vectors occur within a context similar to the one just seen. The formation of subsets can proceed down to any desired level, hence the hypermap name, maps formed within the previous level map.

The learning algorithm has multiple stages:

- Period 1 Unsupervised training of context weights
- Period 2 Adaptation of pattern weights. This follows the two-stage sequential presentation of context then pattern vectors: The winning context subset is chosen. Then the subset nodes's pattern vectors are updated, according to the winner of the now presented pattern vector.

  A further sub-period of supervised learning is then required to label the nodes.

The hypermap was designed to be applied to the co-articulation problems of speech. Unfortunately, it still suffers from the same problems of time-windowing i.e. the selection of the context window around the pattern vector which forms $\underline{x}_{cont}$. It also requires, in exactly the same way as time-delay schemes, a different set of context weights for each level of context. Furthermore, the learning algorithm is very complicated, being divided into many stages.

Kangas has extended the notion of subset formation to occur in a *single* map [29]. The same set of inputs are thus used for the subset (pre-active) region as for the final selection of the winner. The pre-active area is defined to be an environment of the previous best-matching units, for example. The next best match can only be selected from the close neighbourhood region of the previous best match. The next best match then defines the centre of the next pre-active region and so on. In this way, the centres of the pre-active regions define a curve that moves around on the surface of the map.

This model has several attractive properties:

- It utilises a single weightspace
- It reduces to the standard map by having the entire map as the pre-active region
- The map can be divided into separate regions with separate inputs, but the active area can pass over the boundary regions ie the subset forming mechanism is continuous and the input connections are not. 'Attention Shifting' can thus occur between different signal sources and is driven by the signal itself.
- The map can monitor multiple signals in parallel with automatic combination of the representation of these signals by the lateral connections that form the subsets.

Although there is no explicit discussion in [29], it appears as though the weight space must have a globally non-topographic structure; there can be multiple locations, A-areas, on the map that are sensitive to a signal J, but only the one enclosed within the pre-active area P can respond to it. This surely means that weight adaptation only occurs within the pre-activated region P.

It is not clear how the map returns to a 'ground-state' i.e. the case where the pre-active area is the entire map. It states in [29] that "...the pre-active area is (usually) changed after every new sample..."

4.3.4 Response Integration, Data Averaging and Pattern Concatenation Models

Kangas has discussed three models for time-dependent self-organising maps [27] [28]:

- **Response Integration Model.** A *response* is defined as the vector of output layer activities $\underline{y}(t)$. This vector is then *integrated* according to

$$\underline{x}_w(t) = \beta \underline{y}(t) + (1-\beta)\underline{x}_w(t-1) \tag{4.1}$$

where β is a constant controlling the retention of the response. This equation is the same form as for a leaky integrator neuron (see next section), but in this case the whole net's activity is held in the 'memory vector' $\underline{x}_w$. This vector is then classified by a *second* net.

- **Pattern Concatenation** This is time-delay used in conjunction with the basic map formalism.

In the speech data experiments discussed in [28], it is reported that the response integration model is between 4 and 7 percentage points better in recognition accuracy than the old model and that concatenation is between 6 and 10 percentage points better.

4.4 Leaky Integrator Neurons

It is a known property of biological neurons that they *retain* activity on the cell-membrane, i.e. charge *leaks away* from the cell over a period of time. The neuron thus acts as a *leaky integrator* and combines previous (decayed) activity with new activity. The cell thus has a *history* of its recent interaction with input stimuli.

This opens the way to the construction of artificial neural networks that have an intrinsic temporal character; in such networks, the need for time-delay architectures is removed.

In discrete time, a general equation describing the time-evolution of the *activity* for a single neuron is given by

$$A_i(t+1) = dA_i(t) + f(\underline{w}(t), \underline{I}(t)) \tag{4.2}$$

where d is a time constant of the neuron, $\underline{w}$ is a weight vector associated with the neuron i, $\underline{I}$ is an input vector arriving at i and f is a function describing how the neuron processes an input for a given weight vector (e.g. weighted summation). This model omits other biological phenomena such as cell geometry, various ionic

channels, an active membrane which produces non-linear terms etc [69] which affect the temporal behaviour of the neuron.

Temporal Sequence Storage (TSS) using leaky integrators has been investigated by Taylor and Reiss [56], together with the effect of introducing ionic channels. They employed a network model having neurons with a range of time constants which learn to correlate a pattern with the one next in a sequence. It also preserves temporality i.e. correctly reproduces the lengths of the times that each pattern in the sequence was presented. It was shown that leaky integrator neurons could store a temporal sequence directly, holding the incoming activity long enough to learn the transitions between the different patterns in a sequence.

4.5 The Temporal Kohonen Map

Chappell and Taylor have included leaky integrator neurons in the basic Kohonen network [9]. The activity of each neuron is defined by

$$A_i(t+1) = dA_i(t) - 1/2\,|\underline{I}(t) - \underline{w}(t)|^2 \tag{4.3}$$

$$A_{win}(t) = \max_i\{A_i(t)\} \tag{4.4}$$

This activity law is such that the node which wins with *greatest* (i.e. most positive) activity $A_{win}(t)$ will be the *time aggregate* winner of the minimum distance competition. Note that the model reduces to the basic map by setting $d = 0$

In short, if a node's weight vector is similar to the pattern presented at time t, then the contribution to the integrated activity will be small. The converse will be true for a node with weight very different from the presented pattern and hence such a node will be *less likely* to win at the next presentation of a pattern.

4.5.1 Virtual Training Vectors

For the moment, consider a "ready prepared" example weightspace. Suppose we wish the network to be able to discriminate between all the binary pair sequences of length two i.e. all sixteen permutations of the set $\{(0,0),(0,1),(1,0),(1,1)\}$. Call this set S_w.

Equation 4.3 is effectively selecting winners which have best matching *virtual weight vectors.* They are positions in the weightspace that lie *between* the actual training vectors that the network sees. For our example, the real vectors lie at the corners of the unit square. We can then *choose* to partition the unit square so that there will be a unique winner for all sequences $s \in S_w$ i.e.

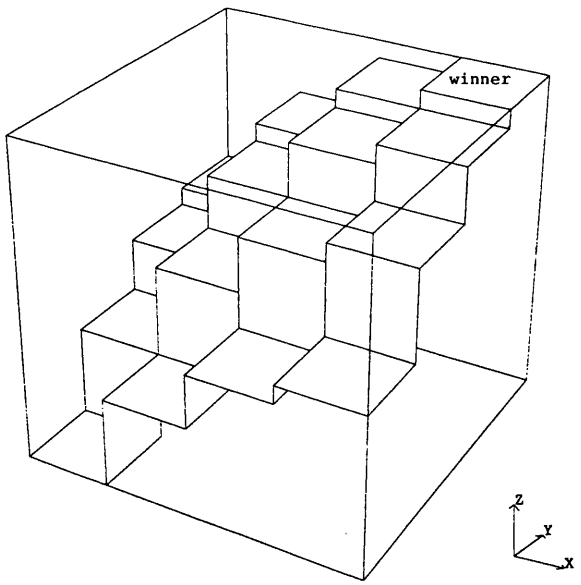
$$\underline{w}_{ij} = ((i-1)/3, (j-1)/3) \tag{4.5}$$

where i and j are the row and column positions of the nodes on the output layer and both run from 1 to 4 for a total of sixteen nodes.

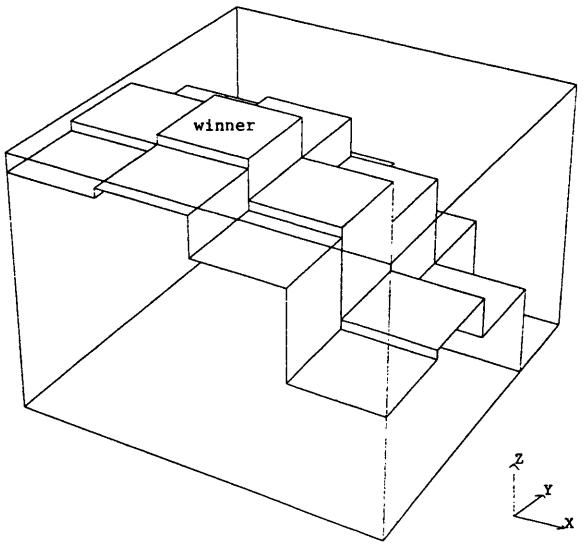
We can now consider the time evolution of A_i for each node i. We can plot this as a histogram at each row and column position on the output layer of the network. Call this activity matrix C_{ij} and assume that at $t = 0$, $\mathbf{C} = \mathbf{0}$.

Figure 4.6(a) shows the activity profile of the network after presentation of the pattern (1,1). It can be seen that corner node corresponding to C_{44} has the most positive activity and is thus the winner of the distance competition. The node diagonally opposite i.e. C_{11} has weight vector (0,0) and hence has the most negative activity.

Figure 4.6(b) shows the activity profile at the next time-step, after having presented the pattern (0,0). If there were no history then the state would be the opposite of the first time-step ie the node at C_{11} would have zero activity. The retention of activity means that instead, the node at C_{22} wins as it is close to (0,0) but was closer than node C_{11} to the pattern (1,1) at the previous time-step.



(a)



(b)

Figure 4.6: Activity levels in the TKM leading to the selection of a context dependent winner.

22	32	23	33
02	12	03	13
20	30	21	31
00	10	01	11

Figure 4.7: Labelled output layer of the TKM.

4.5.2 Nature of the Clustering

In our example, we can now label the output layer with *sequences* comprised of the S_w. If we call the individual patterns $(0,0) \equiv 0$, $(0,1) \equiv 1$, $(1,0) \equiv 2$ and $(1,1) \equiv 3$ then the sequence (0,0)(1,1) can be written succinctly as '03'. The result of giving all sixteen nodes such labels is shown in Figure 4.7.

We can see that clustering occurs according to the most recently presented pattern i.e. there are four groups of four nodes, one per pattern. This is exactly what we would expect from Equation 4.3 as the most recent pattern vector has the largest contributing term in the activity equation.

4.5.3 Rolling Property: Classification Without a Context Window

One of the most attractive features of the TKM formalism is the fact that no context window has to be decided in advance. The sensitivity to particular sequence lengths is determined completely by the decay parameter d in Equation 4.3. This could be described as the 'rolling property' of the model. It means that if the network is trained to be sensitive to sequences of length two, say, and we present the patterns A and B, if we then present a third pattern C, the network will classify the pattern sequence BC. There will of course be noise in the activity from the pattern A and any other patterns that have been presented in the past history, but these are all

weighted by increasingly large powers of d. The value of d is chosen such that sequences of length two are responded to in preference to longer sequences and thus noise from previous patterns should not affect classification for a suitably trained weightspace.

Thus, there is no requirement for sequences to have 'beginnings' and 'ends' and hence no external context window that has to scan through the input pattern stream. Context sensitivity is built into the network dynamics.

4.5.4 Learning Law: Instability and Weight Bunching

The above discussion of the classification was based on the a-priori weightspace defined by 4.5. We must now ask 'How should this weightspace be learned?'

The learning law used in [9] is the same as the standard algorithm, i.e. rotate the neighbourhood of the winner towards the current input vector. Unfortunately, using this law leads to an unstable weightspace because the only data the network is able to train against are the actual training patterns; virtual vectors that exist between the real patterns are *not* represented in the update law.

A simulation was performed to see whether an 8x8 TKM could correctly learn to distinguish sequences of length three. The parameters used were $\sigma(0) = 5, T_{1/2} = 1800, \alpha(0) = 0.2, d = 0.4$. The simulation ran for a total of 8000 epochs. The labelled output layer at epochs 5000 and 8000 are shown in Figure 4.8 (a) and (b) respectively. The corresponding weightspace plots are shown in Figure 4.9(a) and (b).

It can be seen from Figure 4.9(a) that after 5000 epochs, the weightspace is very distorted. This 'distortion' is the typical effect observed when a Kohonen network is used to map a discrete weightspace and the number of training vectors is less than the number of available nodes. The map algorithm *correctly* reproduces the probability distribution i.e. δ-functions located at each pattern vector value, the strengths of which are determined by the *frequency* of presentation of that pattern.

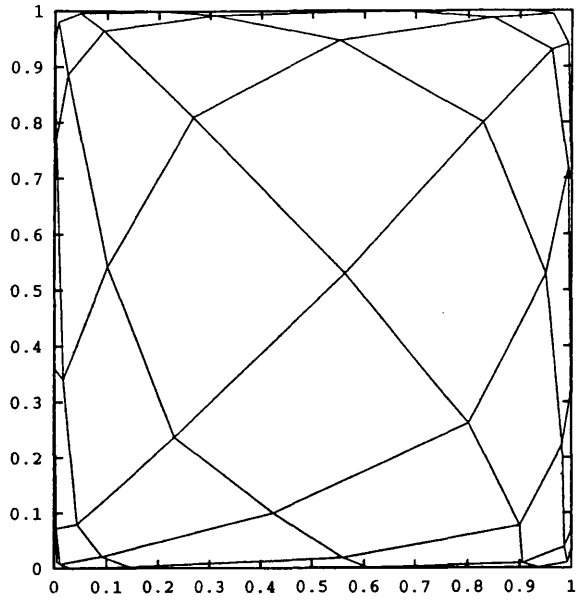
#	#	#	000	100	#	#	#
200	#	#	#	#	110	101	011
220	120	#	#	300	310	301	#
302	102	#	320	330	201	#	111
#	022	122	312	003	321	311	#
222	#	032	#	303	013	231	131
#	#	132	323	033	#	313	#
#	322	232	223	233	333	#	133

(a)

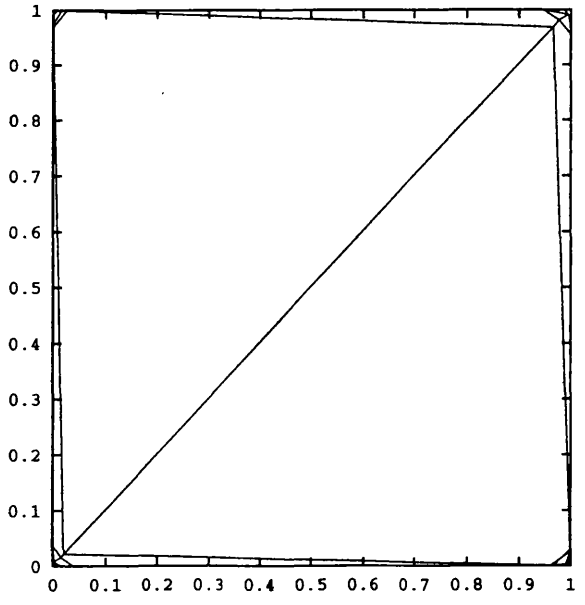
#	#	#	000	#	310	#	#
#	#	#	#	#	#	301	#
320	#	#	#	#	#	#	#
#	#	302	#	330	#	#	111
#	#	#	#	303	321	#	#
222	#	#	#	#	#	#	#
#	#	#	#	333	#	#	313
322	#	#	323	#	#	#	#

(b)

Figure 4.8: Labelled output layer snapshots: (a) $t = 5000$ (b) $t = 8000$ for binary pair sequences of length three.



(a)



(b)

Figure 4.9: Weight bunching snapshots in the TKM: (a) $t = 5000$ (b) $t = 8000$.

This is observed as 'bunching' of nodes' weight vectors around the isolated patterns. For our simulation, we see from Figure 4.9(b) that by 8000 epochs, the network has drawn in all available weight vectors to map the corners of the unit square, because the neighbourhood has shrunk to where only the winning node is affected.

From Figure 4.8(a), we can see that at 5000 epochs, 39/64 sequences were classified. By 8000 epochs, this has dropped to 14/64.

With regards to the TKM, the neighbourhood cannot be allowed to tend to a small size i.e. one only affecting a small fraction of the total number of nodes. There is only a shallow basin of attraction where the learning law adequately interpolates the weightspace to form virtual vectors *and* there is sufficient accuracy of these virtual vectors for correct classification. For the sequences of length two example, the balance between these two opposing requirements can produce a solution. For sequences of length three, only 'tweaking' of learning parameters and training time can produce any degree of classification. In either case, there is no ultimate stable state where the weights can be arbitrarily, accurately self-organised and bunching effects to have not started to appear. A more generally applicable learning law is thus required.

4.5.5 Using The TKM for Syntactic Analysis

Chappell and Taylor also looked at using the TKM for classifying the context of words in simple sentences i.e. occurrences of the same word but in different contexts should be mapped to different/nodes or regions but within some enclosing set which represents all uses of that word. A simulation was presented in which the effect of *position* of a word in a particular sentence could not give artificial assistance to disambiguating different contextual meanings to that word i.e. verbs and nouns frequently occur at fixed locations within a sentence). The following set of sentences was thus used as training data:

1. My hair is dry now.

START	0000
my	0001
hair	0010
is	0011
dry	0100
now	0101
I	0110
comb	0111
must	1000
off	1001

Figure 4.10: Training data for semantic map simulation

2. I comb my dry hair.

3. My hair must dry off

Clearly, all three sentences have the word 'dry' as the fourth word, but acting in different contexts e.g. as a verb in the last example. Each word was encoded on four binary inputs and the sequences were presented randomly to an 8×8 TKM, each one separated by the START symbol (all lines off). The training vectors are shown in Figure 4.10 Figure 4.11 shows the resulting labelled map, the number(s) attached to each word giving the number of the sentence to which the word was a part of. It can be seen that the three instances of the word 'dry' form a clear cluster and that the mapping for dry from sentences 1 and 3 occupy adjacent nodes. This reflects that these two sentences are the same bar one word at the point 'dry' is seen by the network. The TKM is thus able to perform disambiguation in longer sentences.

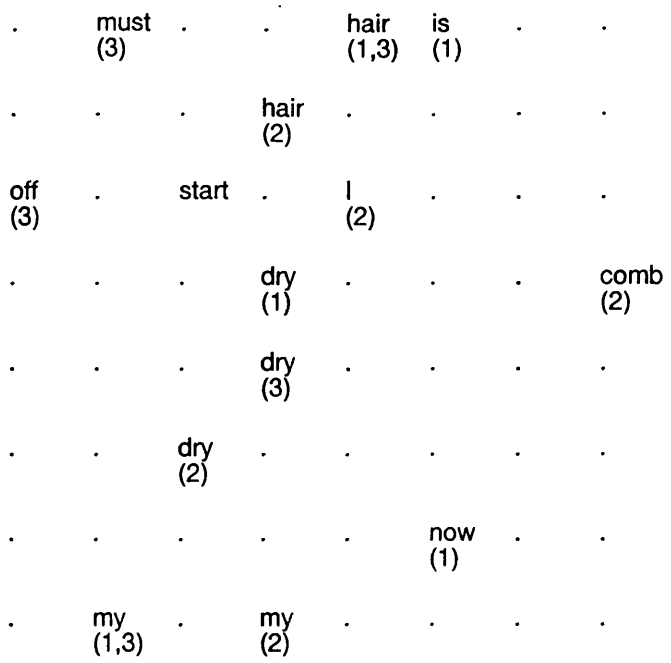


Figure 4.11: Semantic map of words presented in training

4.6 Using Virtual Vectors In Training

Virtual vectors have already been defined as the optimal positions in weightspace of sequence representations. We can identify two types of virtual vector for the case of sequences of length two.

1. Representations of sequences that are composed of *different* pattern vectors e.g. 23 or 10

2. Representations of sequences that are repeated copies of *one* pattern e.g. 11 or 33

The latter type have virtual vectors which coincide with the single pattern vector.

How might we go about constructing virtual vectors which can be used to perform training for all times? We might try and construct them so that they lie on the line

between two patterns A and B i.e. fulfill the criteria of existing *between* the real vectors. We could try

$$\underline{V}_{AB}(t+1) = \underline{B}(t) + d(\underline{A}(t) - \underline{B}(t)) \tag{4.6}$$

for a sequence of two patterns A and B, i.e. use the current pattern shifted towards the old pattern. This recipe will provide suitable $\underline{V}_{AB}$ values for training, but defeats the main purposes of the TKM of being a biologically plausible extension of the SOM and of being context-window free. Externally producing the $\underline{V}_{AB}$ is no better than a time-delay network, as the A and B have to be held somewhere.

Instead, we might consider somehow directly integrating the *training vectors* and not some activity measure.

$$\underline{V}(t+1) = d\underline{V}(t) + \underline{I}(t) \tag{4.7}$$

However, this cannot be done directly within our artificial neurons as they can only retain a scalar history.

4.7 Formation of Virtual Vectors Using Traces

4.7.1 What are Traces?

Traces are models of synapses that can store a history of input signals that they are exposed to. This is in contrast to such a history summing mechanism only being present on the cell body where individual weighted inputs are combined, i.e. leaky integrators. Traces have been used in difficult, learning control problems, such as pole balancing [4]. In this example, traces implement so called 'eligibility'. Basically, there is a pairing between input on a particular pathway and output of a neuronlike device at some later time. A weight on this pathway is modified in relation to whether its effect *in the future* was good or bad i.e. there is reinforcement learning,

but the eligibility for modification should die away rapidly to preserve the *context* of the pairing.

The synaptic trace model presented here allows for the mapping of sequences in a way which incorporates context information automatically, as in the Temporal Kohonen Map (TKM), but is rigidly stable when neighbourhood and learning rate become arbitrarily small. We will see that such models can map longer sequences.

4.7.2 Learning Law and Training Data

The most desirable property of the TKM is that sequences of input signals are classified without the need to take explicit account of context. This is in contrast to time-delay schemes and any model requiring a time-window. The problem with the TKM is that no history of the *actual patterns* is retained, only the nodes' relatively similar response over time. So although with a ready prepared suitable weightspace, the discrimination function works perfectly, it is by no means clear as to how this weightspace can be learnt during training.

This trace model uses activities on each line, i.e. at each synapse, to provide a history of the actual components of each training vector as it is presented. The 'rolling property' of the TKM is preserved i.e. if three patterns constitute a sequence and then a fourth is presented, a new sequence is seen by the network, comprising of the new pattern, the two most recent and noise from any previous patterns that have been seen (if decay parameters are chosen appropriately).

In the TKM work, binary pairs were randomly presented to the networks. In this work, *bipolar* pairs are adopted for reasons which will be discussed below.

4.7.3 Trace Architecture

Consider an input vector $\underline{I}$ of dimension n. Each line i of the n lines is summed leakily:

$$A_i(t+1) = d_T A_i(t) + I_i(t) \tag{4.8}$$

where A_i is a trace activity on line i, d_T is the *decay rate* on the line and I_i is the corresponding input component.

4.8 Analysis of Trace Activity

Let us consider the activity A of a single synapse, i.e. drop the index i in Equation 4.8, when a sequence of 1-dimensional patterns is presented to it. Let us furthermore assume that the initial value of A i.e. $A(0)$ is set to be the *reset value*. What we mean by the reset value is the *average* of our possible pattern space (see the end of this section for full description). The reset value is the value of $A(0)$ that corresponds to an infinite stream of random patterns having being presented at that synapse.

Let the pattern sequence presented to our synapse be called $P_l(t)$ and let it be m time-steps in duration. Here, l is a label which refers to a specific temporal sequence. At time $t = m$, A will have m terms, i.e. the backwards exponentially weighted values of P_l at times $t = 1, 2, \ldots, m$. Call this value of $A(m)$ the *synaptic history* S_l of the temporal sequence l. We can thus write

$$S_l = \sum_{t=1}^{m} P_l(t) \cdot {d_T}^{m-t} \tag{4.9}$$

Noise and Error Bandwidth

In general, the value of $A(0)$ will not be the reset value. To preserve the rolling property of the TKM, we must ensure that a particular synaptic history S_l can still be classified uniquely, no matter what the *noise* at a particular synapse is. It should be independent of the value $A(0)$ that resulted from an arbitrary presentation of patterns in the past.

Let $S_l{}'$ be the value of S_l containing arbitrary interference from $A(0)$. We can write for a sequence l of length m

$$S_l{}' = S_l + O({d_T}^m)$$

$$= S_l + E_m \tag{4.10}$$

where E_m is an error term. We can calculate the worst possible effect that this term can have. This occurs when all inputs $I(t)$ for $t = -\infty, \ldots, 0$ have the same direction, e.g. for the bipolar case, all +1 or all -1. Equation 4.10 is a geometric series. If we take E_m as the sum of all terms excluding the first n, then the first term is $a = {d_T}^n$ and the ratio of terms is $r = d_T$.

$$|E_m| = S_\infty = a\left(\frac{1 - r^\infty}{1 - r}\right) = \frac{{d_T}^n}{1 - d_T} \tag{4.11}$$

This gives a measure of the effect of a non-zero A(0) in the worst case, which we call the *error bandwidth.* For any sequence of length m, $|E_m|$ must be less than the smallest separation between any two adjacent synaptic histories, or else interference from activity at $t = 0$ will be too great to allow the necessary discrimination.

Trace Reset Value and Binary versus Bipolar Data

If we are using a trained network to classify a sequence, the reset value of the traces has to be considered. Call this reset value A_R. This value is *not* simply the average of the basis vectors i.e. for binary data it is not 0.5. We need to find the centre of the *state-space* i.e. the range of the integrated synapse values. For binary data, the range of the state space is

$$\left[0, \frac{1}{(1 - d_T)}\right]$$

and hence

$$A_R = \frac{1}{2 \cdot (1 - d_T)}$$

For bipolar values, the reset value is always *zero.* Bipolar data thus has a slight practical advantage in that A_R need not be calculated.

4.9 Experimental Results

4.9.1 An Example - Pair Sequences of Length Three

We first present a practical example to elucidate the analysis of the previous section. A simulation was performed to record pattern-density statistics, final convergence state of the weight-space and sequence labelling for a network learning the complete set of 64, 2-D, length *three* sequences of bipolar training vectors. The pattern statistics are shown for one input line only, but the weight-space plot shows no coupling effects between lines (i.e. due to poor random number generation) in this simulation.

The trace decay value used was $d_T = 0.45$. The neighbourhood was Gaussian with initially $\sigma = 5$, initial learning rate $\alpha = 0.35$ and an exponential decay of learning parameters, $T_{1/2} = 1800$. The total training time was 8500 epochs.

According to Equation 4.9, the set of 8 *noiseless* ($A(0) = 0$) synaptic histories for the set of 1-dimensional bipolar sequences of length three is

$$S_l \in \{-1.6525, -1.2475, -0.7525, -0.3475, 0.3475, 0.7525, 1.2475, 1.6525\} \quad (4.12)$$

calculated by taking all possible permutations of inputs thus:

$$\{\{I(1) = -1, I(2) = -1, I(3) = -1\}, \cdots, \{I(1) = +1, I(2) = +1, I(3) = +1\}\}$$

Figure 4.12 shows the probability distribution, in one dimension, of formed training vectors for the simulation. The black arrows mark the values of S_l from 4.12. The clusters are clearly evident and are labelled 1 through 8. Note that the centres of the clusters are in agreement with the values of S_l.

Of note is the *bandwidth* of the clusters, which is a key factor in the network being able to form a correct mapping. This is the problem that any Kohonen network faces when trying to map an input space formed of discrete vectors. If this accumulated error giving rise to the bandwidth is not present during training, then the mapping in general will not converge to a correct solution and is instead

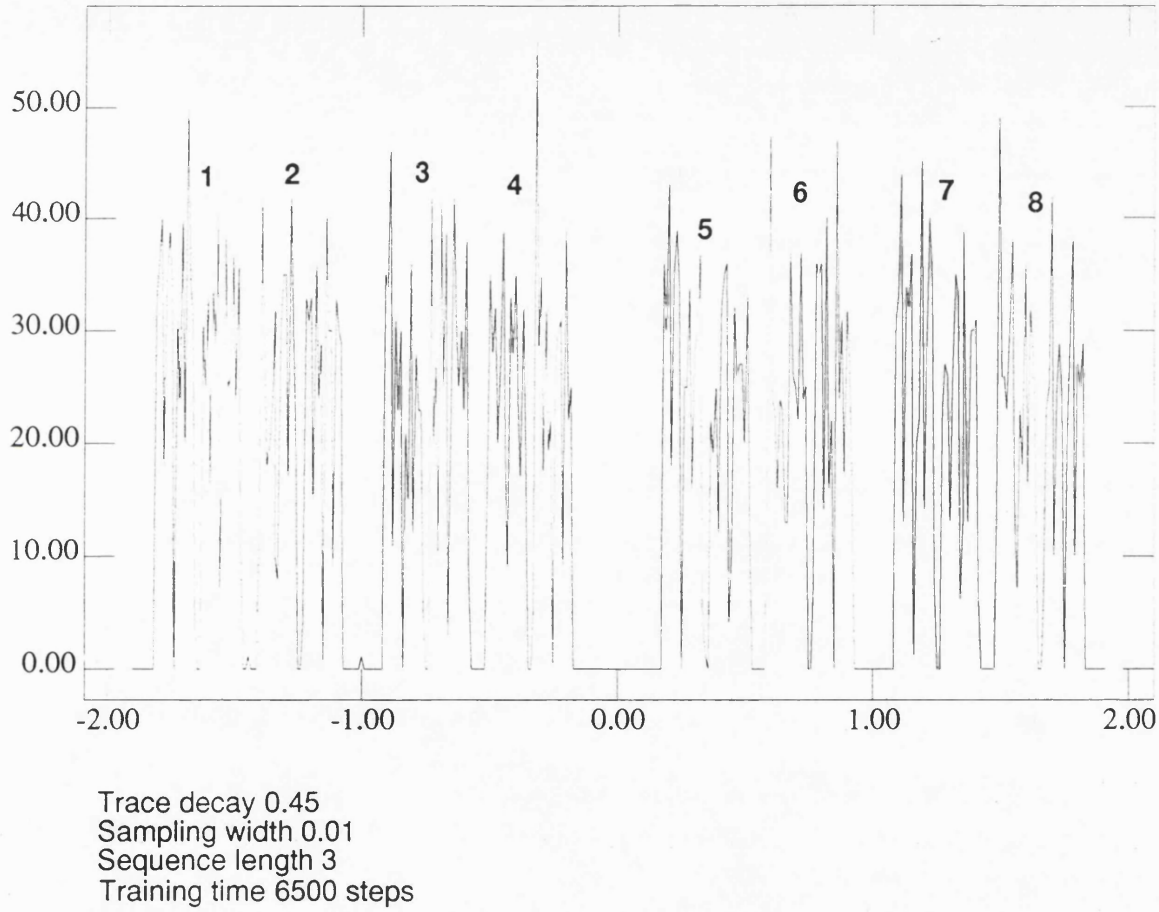


Figure 4.12: Pattern probability distribution for a single synapse.

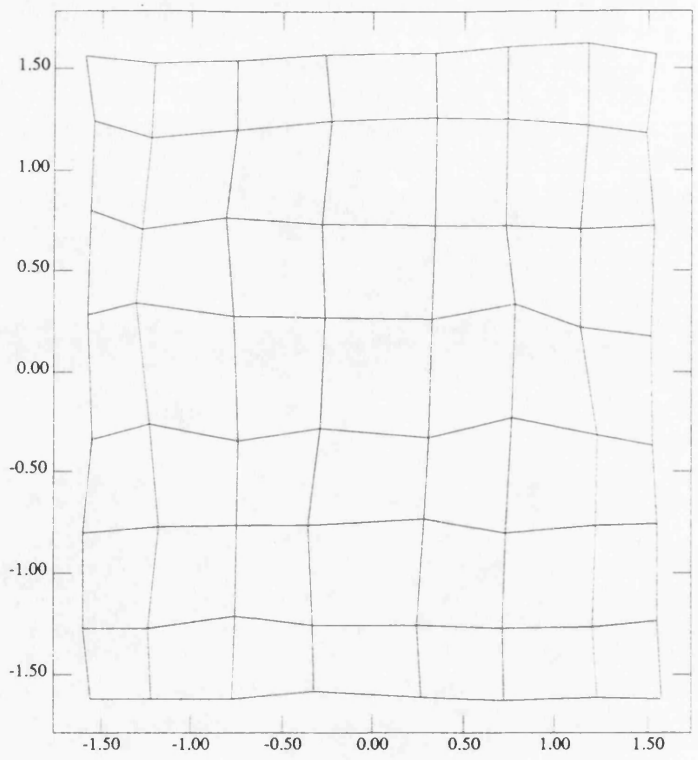


Figure 4.13: Final weightspace plot for bipolar sequences of length three.

distorted, particularly in the much lower density regions of the map which occur in the centre.

The bandwidth measured from the results is ± 0.16. This agrees very well with the calculated value of $|E_m| = 0.1657$

Figure 4.13 shows the final state of the weight-space for the simulation. There is good correspondence between node points and the values for S_l.

The labelling for the nodes is shown in Figure 4.14. Each three digits in parenthesis is the label for sequences made up of the four patterns [$0 = \{-1,-1\}, 1 = \{-1,+1\}, 2 = \{+1,-1\}, 3 = \{+1,+1\}$]

It can be seen that the clustering is of the same form for the TKM i.e by most recently presented pattern e.g. bottom-right 16 nodes are the sequences terminating

(111)(311)(131)(331)(113)(313)(133)(333)
(011)(211)(031)(231)(013)(213)(033)(233)
(101)(301)(121)(321)(103)(303)(123)(323)
(001)(201)(021)(221)(003)(203)(023)(223)
(110)(310)(130)(330)(112)(312)(132)(332)
(010)(210)(030)(230)(012)(212)(032)(232)
(100)(300)(120)(320)(102)(302)(122)(322)
(000)(200)(020)(220)(002)(202)(022)(222)

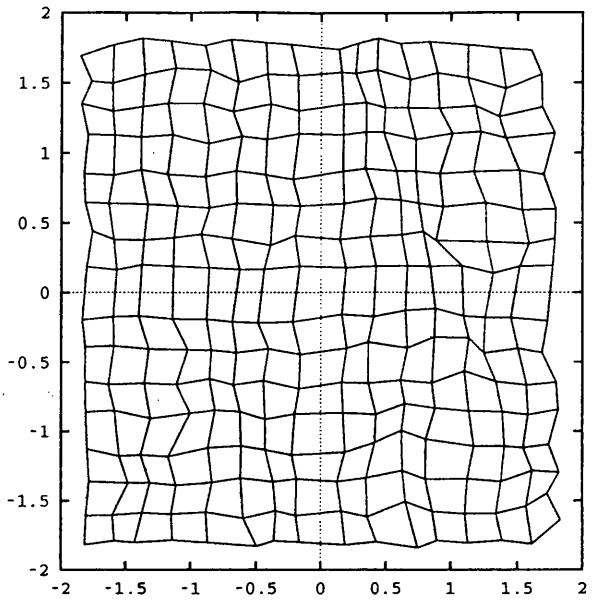
Figure 4.14: Labelled output layer for bipolar sequences of length three.

in pattern 2.

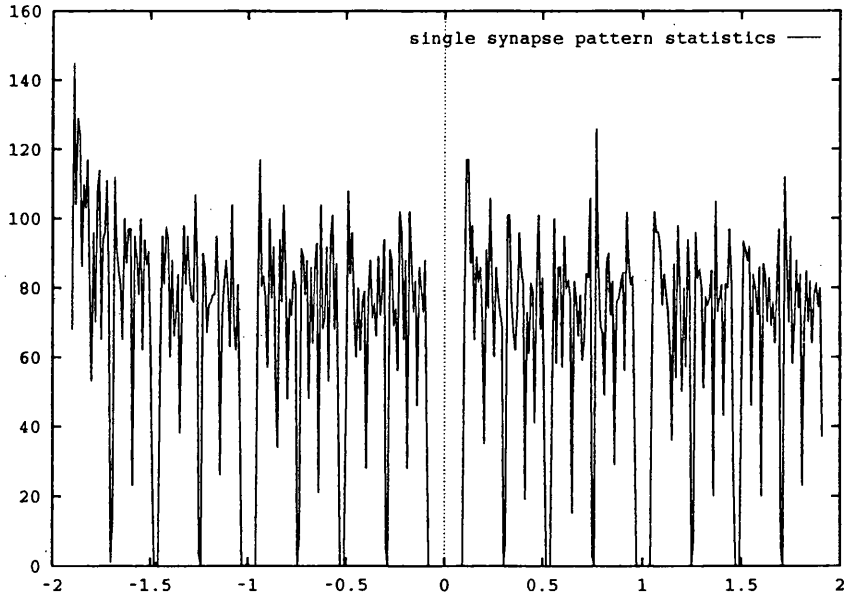
4.9.2 Learning Sequences of Length Four

The above simulation was repeated for sequences of length four, using a square network of 256 nodes with parameters $\sigma(0) = 8.5, \alpha(0) = 0.35, d_T = 0.475$ and a training time $T = 25000$ epochs. It was found, over the course of numerous simulations (around 50), that the best result was one where the network classified all but 11 of the 256 sequences (a success rate of 95.7 %). Figure 4.15(a) shows the final weight space plot, which shows a few 'crinkles' in the positive quadrant responsible for the misclassifcations. It seems that with suitable parameter 'tweaking' that *all* sequences could be captured, but it is likely, that the *limit* for what can be easily achieved in a single layer is sequences of length *four*. This is born out by the 'crowded' pattern spectrum for a single synapse shown in Figure 4.15(b).

Clearly, classification of sequences longer than four will have to be tackled by *multilayer networks*. This will be discussed in Chapter 5.



(a)



(b)

Figure 4.15: (a) Final weightspace plot and (b) single synapse pattern statistics for sequences of length four.

4.10 The Importance of Noise In Map Formation

We have seen that the random accumulated noise at each synapse performs a crucial role in providing a suitable weightspace to be mapped. In the simulations above, a trace decay value of around 0.45 was used which is biologically unrealistic.

Another simulation was carried out, using a trace value of 0.35. It can be seen that the weightspace is somewhat distorted with resultant misclassification.

The main point is that noise improves the exploration of the state space [68]. We could thus alter the model to have noisy synapses, which would remove the need for such slow decay rates for the trace histories.

4.11 Comparison Of Trace Model With Kangas' Data Averaging Model

Kangas has discussed a model which is mathematically equivalent to the trace architecture discussed in the previous section [27]. A *memory vector* $\underline{x}_w(t)$ is defined as forming the average of the input patterns over time.

$$\underline{x}_w(t) = (1-w)\underline{x}_w(t-1) + w\underline{x}(t) \tag{4.13}$$

Kangas described this model as being expected to have better tolerance in a noisy environment as the averaging afforded by Equation 4.13 will compensate for additive noise. But more importantly, he says that the model was *"not expected to produce very good representations for sequential data, because the following patterns will effectively shadow the preceding ones"*.

His test system was a simulated object moving in a two dimensional space on a figure of eight shaped track with the facility for adding various degrees of noise. The problem was to determine which direction the object was travelling, particularly in

the cross over region. He reported that in experiments, the classification accuracy was improved by data averaging, but that there was poor tolerance on the parameter w.

Kangas is quite correct to say that shadowing of past patterns will occur in *a continuous input pattern space.* This is however *not* the case for the discrete (bipolar) patterns that have been discussed in this chapter. They are in fact *guaranteed* not to overshadow previous patterns.

4.12 Summary of Chapter 4

We began by reviewing methods of representing time in neural networks i.e. pattern vector concatenation and recurrent architecture models. The advantages and disadvantages of such approaches were discussed. We then reviewed work which had adapted the Kohonen model to temporal applications.

The leaky integrator model of the neuron was then reviewed and discussed in the context of the Temporal Kohonen Map of Taylor and Chappell. The problem of weightspace instability when using the standard Kohonen update law with the TKM was discussed and a simulation presented showing the model attempting to map the complete set of binary pair sequences of length three. A new model which moved the site of leaky integration from the cell body to the synapses was then introduced. This new model was shown to retain a history of the actual training patterns which allowed the map to more easily form a stable weightspace and thus map longer sequences. An analysis of the properties of a single synapse was presented which lead to the idea of 'error bandwidth' of the representation of a single sequence. Bounds on the synapse trace decay were given so that the non-overlapping of representations was assured. The importance of having a large noise term was demonstrated by simulation; this is because the SOM performs much better when mapping quasi-continuous input distributions than quasi-discrete ones.

Chapter 5

Hierarchical Maps

5.1 Hierarchical Classification of Temporal Sequences

We have investigated the classification ability of a single layer network. The amount of training required and the accuracy of the weightspace discriminations needed inevitably limit the useful abstraction that can be performed in a single layer. The next logical step is to consider how multilayer systems can cope with longer sequences.

We will use both binary and bipolar variables interchangeably in this section.

5.1.1 Connections Between Layers

In the standard formalism, the output of the Kohonen layer is a unary vector, with the one active line corresponding to the winning node. Such an output vector is not of much use when trying to relate *topographic* relationships between layers. For example, we would expect a 4x4 output layer to have a *similar* output for say the node at row 3, column 3 and the node at row 2, column 3. The corresponding unary output vectors might be $(0,0,\cdots,0,0,0,0,1)$ and $(0,0,\cdots,1,0,0,0,0)$ respectively. These can hardly be considered similar! There is thus little that a *static* Hierarchical map can do unless it has *graded response* outputs. How might the graded response y_i for each neuron i be defined? Possibilities are

- Some function of similarity i.e. a variant of $y = 1/(\epsilon+\beta)$ where $\beta_i = D^2(\underline{I},\underline{w})$ and D is the Euclidean distance between input $\underline{I}$ and weight vector $\underline{w}$. This then produces a 'spectrum' of activities, with the winner having the output value 1. This is what Kangas calls a 'response' [28][27]. Kangas used a multilayer strategy to make a 'more orthogonal' representation of the raw input vectors which could then be integrated according to Equation 4.13 and then self-organised by another map. In all cases, he reports that using pattern concatenation gives superior results.

- Some output layer topology variant of the above. This might be a gaussian set of outputs on the output layer, centred at the winning node. The winner thus has output 1 and other nodes are given outputs according to their distance from the winner.

5.1.2 Simulating Topographic Location by Co-ordinate Passing

We can circumnavigate having to feed forward a very large output vector and the associated problems of giving the vector a topographic meaning, by arranging the output layer to pass forward the *location* on the grid of the winning node directly. Such a mechanism was employed by Whittington et al [73] in their Hierarchical Adaptive Kohonen Feature Map Model (HAKFM). In this case, the coordinates were concatenated by a shift register i.e. a time delay model and then this vector is classified by a higher layer.

Co-ordinate passing can be achieved by an arrangement of two weights per node, each weight having a value linearly related to the particular coordinate (See Figure 5.1). The input to the next layer then consists of $\sum_{i=1}^{n} z_i \underline{W}_i$ for all the n nodes in the feeding layer which is just $\underline{W}_{win}$ for a unary output vector $\underline{z}$ and winning node *win*.

This obviously preserves topological information and means that the need for a large interconnecting weightspace is sidestepped. Of course, this does not make any biological sense as real layers of neurons are unlikely to be doing this. Having said this, the model has simplicity on its side and the dimension of the interconnecting weights, namely two, is the same for all subsequent layers.

5.1.3 Co-ordinate Passing With Trace Architecture

Co-ordinate passing works well with the trace architecture, with the caveat that the trace decay value d_T must be less than the smallest inter-pattern separation e.g. if

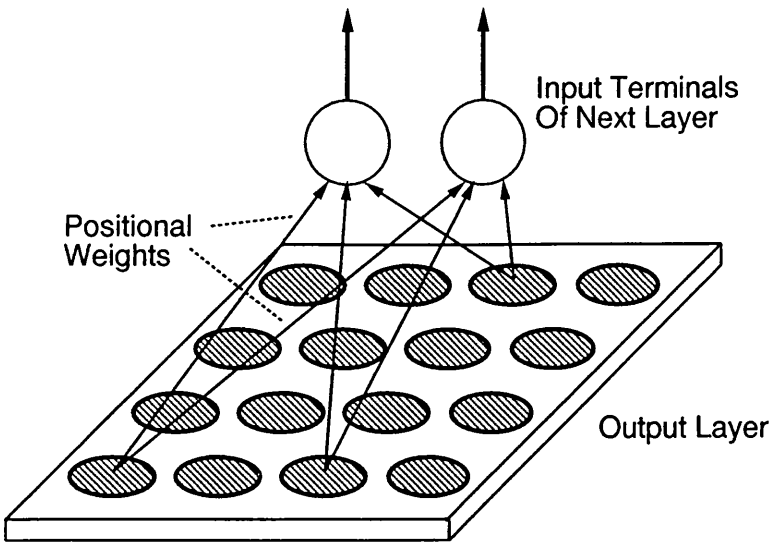


Figure 5.1: Weight arrangement for coordinate passing.

a 1 dimensional array of four points is represented by the set { 0.0, 0.3, 0.7 and 1.0 }, then the boundary condition is

$$\left(\frac{1}{1-d_T}-1\right) < 0.3$$

that is, the worst case scenario would be an infinite string of value 1.0, followed by 0.0. The above condition must be met if the trace value is not to overlap with the first pattern value 0.3.

5.1.4 Clocking of Connected Layers

Suppose we are trying to train a hierarchical layer B from the output of a previously trained layer A. If layer A classifies sequences of length two, say, then we might naturally want layer B to classify a sequence of length *four* i.e. detect two winners from layer A. Of course, layer A generates winners at *every* time-step. If we want there two be just two isolated events on layer A, we have to choose to count that there have been two groups of two. The layers thus have to have some form

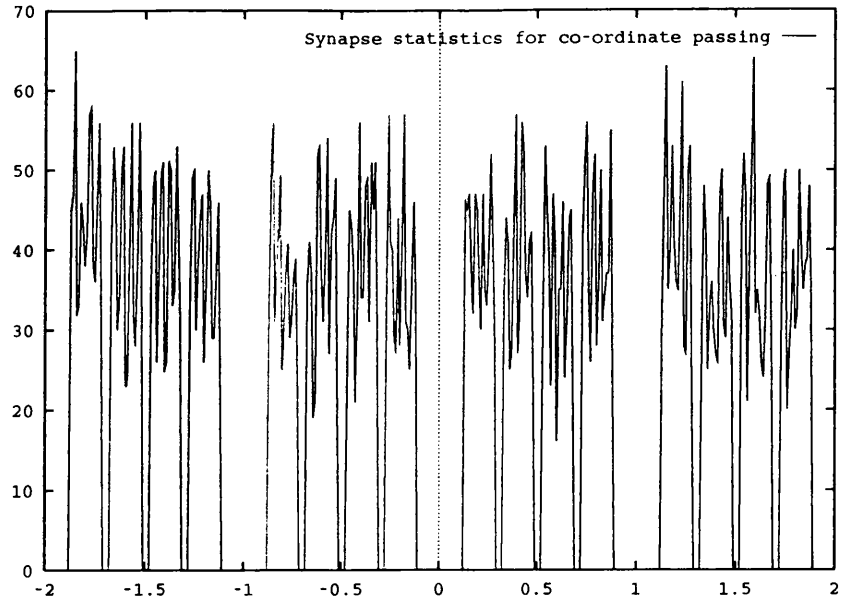


Figure 5.2: Single synapse statistics for coordinate passing example (16 × 16 grid).

of synchronisation or *clocking* to guarantee that learning only occurs at the right times [72].

A simulation was carried out using clocking on a two layer system, using the coordinate set $\{-1.5, -0.5, +0.5, +1.5\}$. The single synapse statistics for this set is shown in Figure 5.2. The first layer was trained to respond to sequences of length two and then the second layer was trained on the coordinate data of the first layer. This correctly classified all 256 sequences, as predicted.

5.1.5 Learning at Every Time-Step

There is no reason why a two-layer feed-forward system, where the first layer is classifying sequences of length two, has to have a clocking mechanism as described. There is no requirement for it to segment the sequences into chunks of length two. In the hierarchical stack of Kohonen maps used by Tatersall et al, there is learning

at every time-step [67][66].

Suppose in our two-layer system, we have feed-forward at *every* time-step. What will the second layer actually be classifying?

Consider a 1-dimensional first layer (a linear chain) which has been trained on one dimensional bipolar data and the decay rate of the traces is such that sequences of length two are classified. Now consider that the state of this layer is such that it has just been presented with the sequence AA. This will have produced a winner at node 1, say. Then assume that a third pattern is now presented and that it is a B. Node 3 will then respond as layer 1 will have correctly classified the sequence AB (preceded by AA at the previous time-step). The output vector therefore at times t and $t+1$ will be (+1,-1,-1,-1) and (-1,-1,+1,-1) respectively. Hence, if layer 2 is also classifying sequences of length two, it will actually classify sequences of length *three* from the two patterns presented from layer 1.

With learning at every-time step, a co-ordinate passing two layer system easily maps the set of 64, length three sequences onto a second layer of 8×8 nodes.

5.2 Fully Connected Two-Layer Systems

We want to investigate other forms of communicating co-ordinate information between layers that does not require multivalued neuronal outputs, but equally importantly has an intrinsic topographic structure. The most obvious choice to investigate is the full pass forward of all outputs of the feeding layer i.e. for a 4×4 feeding layer, this output is a 16-d unary coded vector. We would expect some limited clustering ability due to the integrated trace values i.e. the output vector will no longer appear to be unary coded to the second layer. We will look at learning at every time-step for the second layer.

A simulation was performed where layer 1 was first trained on two-dimensional bipolar sequences (sensitive to length 2, i.e. there were 16 neurons arranged in a square grid). The sixteen dimensional output of this layer was then used to train a

113	213	#	212	012	112	#	111	011	#	301	201
013	313	#	312	#	#	311	211	#	#	001	101
#	#	#	#	#	#	#	#	#	#	#	#
131	331	#	133	333	#	#	130	230	#	110	010
031	231	#	233	033	#	330	#	#	#	210	310
#	#	#	#	#	#	#	#	030	#	#	#
021	#	#	#	332	032	#	303	#	#	322	122
121	321	#	132	#	#	#	#	003	#	#	222
221	#	#	#	#	#	203	103	#	#	022	#
#	#	#	232	#	#	#	#	#	#	#	#
020	320	#	#	323	#	#	102	002	#	200	300
120	220	#	223	123	023	#	302	202	#	100	000

Figure 5.3: Labelled output layer for fully connected two-layer system.

12x12 layer 2. The output of the first layer was arranged to be bipolar also i.e. +1 for the winning node and -1 otherwise. A 12x12 grid was used for the second layer due to the extreme dimensional reduction the layer was expected to form. It was found to be impossible to compress the patterns on to anything smaller.

Figure 5.3 shows the labelled output layer at the end of the run. All sixty-four patterns have been captured and it can be seen that each group of four patterns terminating in a particular pair (eg X00 at the bottom right) are locally correct in their clustering. However, global clustering is not good.

5.2.1 An Enforced Output Spectrum For Preserving Topological Information

We have seen that there is no connection between particular elements of the output vector and that this vector is very sparse. For a network of n nodes there is only a

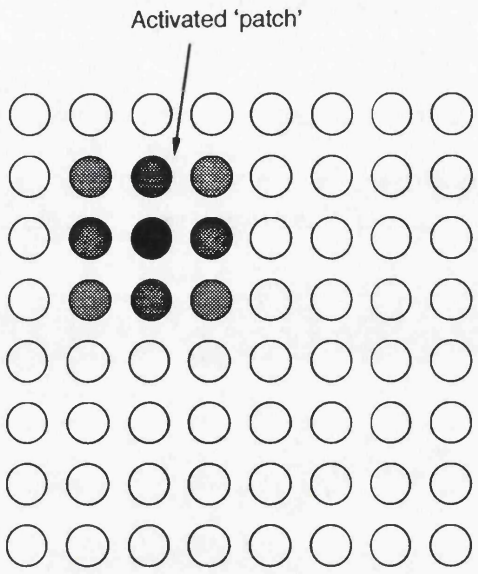


Figure 5.4: Use of activated patch to represent topology.

$1/n$ chance of that node being a winner (assuming that the nodes have mapped the input distribution correctly). Consider that we now choose to *enforce* some structure on the nature of the output (as described above). Say that a winner has been found in response to some pattern/pattern sequence. Instead of the output being +1 for this node and -1 for all others, that there is an 'activated patch' that surrounds the position of the winning node (See Figure 5.4)

Each output y_i might be defined along the lines of

$$\begin{aligned} y_i &= \exp(-R_i^2/2\sigma^2) \text{ for } R_i < R_{cut} \\ y_i &= -1 \text{ otherwise} \end{aligned} \tag{5.1}$$

where $\sigma = n/4$, n is the number of nodes in the (square) output grid, R_i is the physical distance between the winning node and the node i and $R_{cut} = 2\sigma$ is a cut-off distance beyond which nodes are 'off' i.e. have output -1.

We repeat the simulation of the previous section, utilising the above form for y_i. For a 4x4 network, this results in just a cross-shape of activity around the winning

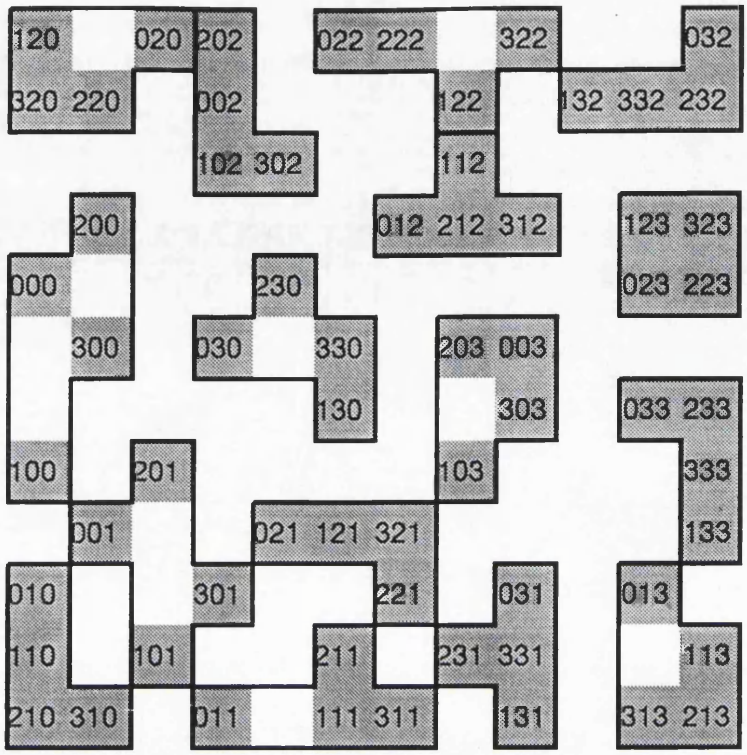


Figure 5.5: Clustering using gaussian profile activated patch.

node. But this simple correlation of topologically close outputs produces a dramatic improvement in the global clustering properties exhibited by layer 2. Figure 5.5 shows the output state of layer2 which has been divided up to show the clusters. All sixteen clusters are present as before, but now they are all topologically correctly positioned as per the topology of layer 1. The outer ring of twelve clusters are easy to follow round and the positions of the inner four are also correct (of course, the layer does not look perfect because of dimensional reduction).

Although each cluster is formed and is in a globally correct position, we can see that there is no structure *internal* to each class. If we extend the cutoff range to $R_{cut} = 3\sigma$, we note significant improvement of local performance in clustering i.e. 0 in the top left, 1 in the top right, 2 in the bottom left and 3 in the bottom right of

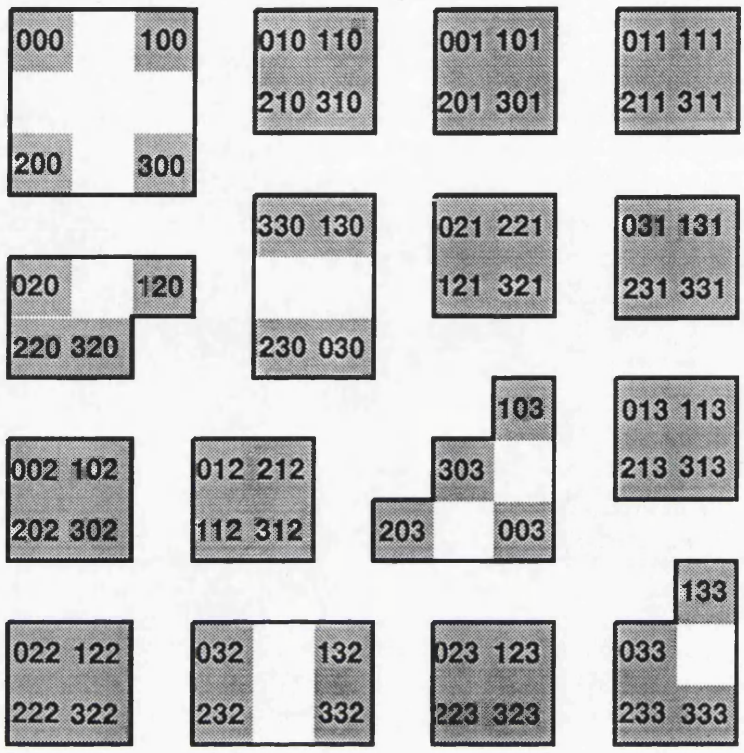


Figure 5.6: Clustering using an activated patch with extended cut-off range.

most clusters (See Figure 5.6). The subdivision is not perfect, however, within the central 4 clusters. This can be accounted for by recognising that interior nodes on the feeding layer have a higher probability of having a positive output i.e. making an active contribution to the output vector. They have physically more nodes around them to force them to be part of the output spectrum.

5.3 Grey-Code Representation of Topology

An alternative approach to communicating topological information that does not require continuous values (and hence should be much more applicable to the synaptic trace model) is one where the topographic position of a node on the grid is repre-

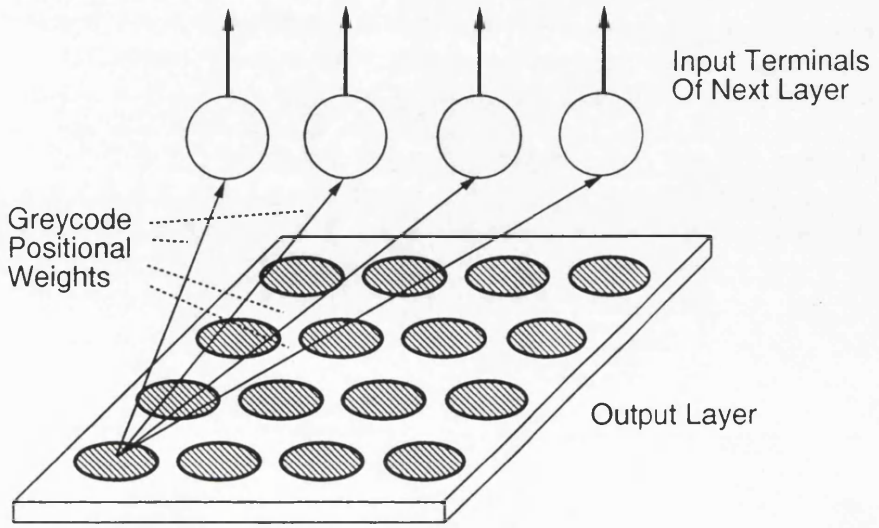


Figure 5.7: weight organisation for greycoding of a 4×4 layer

sented as a 'grey-code' [3]. This means that there is a fixed number of bits available to represent each of the x and y coordinates. For example, a 4×4 square layer requires 2 bits for both x and y dimensions, so that each node requires a total of *four* bits or weights. The arrangement of weights seen by the next layer is shown in Figure 5.7

In general, an $M \times M$ grid will require $2\log_2(M)$ weights per node. The training vector for the next layer is the concatenation of the greycodings for the x and y components i.e. $(\underline{x}_g, \underline{y}_g)$

Figure 5.8 shows the codings for each node. Moving along either the x or the y direction, we can see that each node's weight vector differs by 1 bit and that this arrangement is topological e.g. the two highlighted weights in the diagram. This code is topological with respect to the *Hamming distance* metric [24].

Simulations were carried out on a 4×4 initial layer and an 12×12 final layer as before, with the greycode weights shown in Figure 5.8.

We can see from Figure 5.9(b) that coarse grouping by last pattern presented

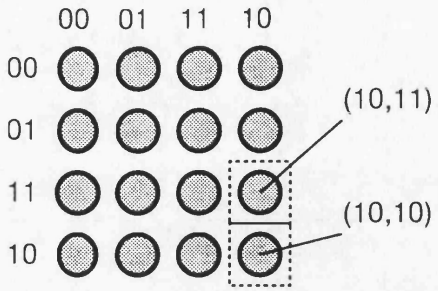


Figure 5.8: x and y greycode components for a 4×4 grid

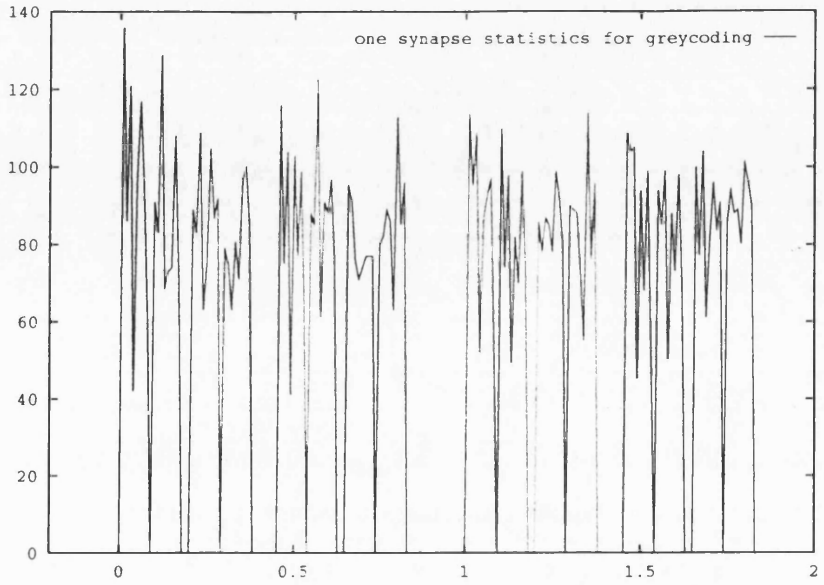
has occurred, but that the fine structure of the map is fairly poor. Why should this be the case?

5.3.1 Mapping Four Dimensional Grey-Code Vectors On A Single Layer

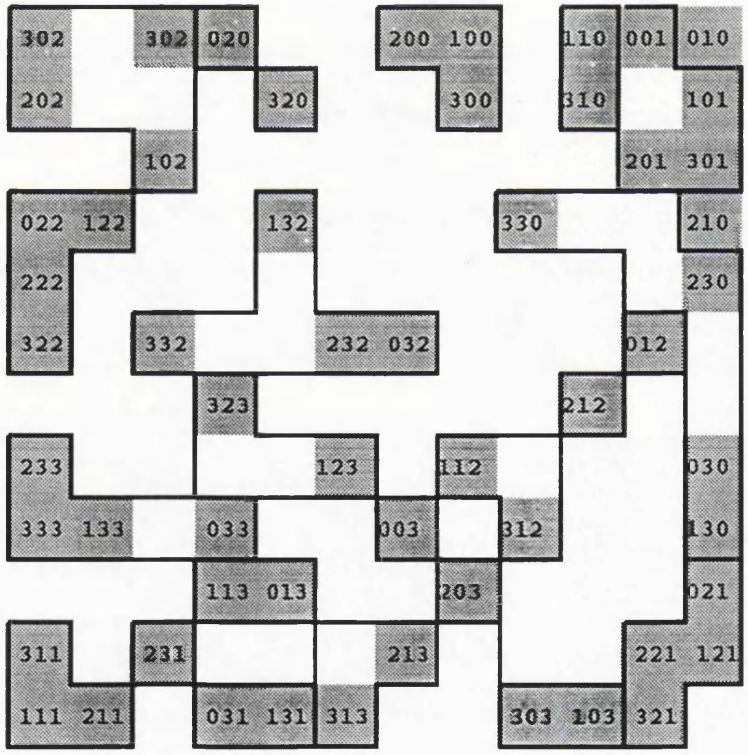
We can simplify the question of why the grey code model does not live up to expectation by choosing to map an explicit set of 4-dimensional training vectors on to a single layer. There is now no confusion as to what the layer is attempting to map i.e. we can distinguish between any errors in *implementing* the grey-code model and the coding scheme itself. The following set of training vectors was used in the simulation:

(0,0,0,0)	00	(0,1,0,0)	10	(1,1,0,0)	20	(1,0,0,0)	30
(0,0,0,1)	01	(0,1,0,1)	11	(1,1,0,1)	21	(1,0,0,1)	31
(0,0,1,1)	02	(0,1,1,1)	12	(1,1,1,1)	22	(1,0,1,1)	32
(0,0,1,0)	03	(0,1,1,0)	13	(1,1,1,0)	23	(1,0,1,0)	33

Here, each training pattern is followed by a label formed from the (x, y) coordinate of the node we might ideally like the pattern to represent (subject of course to the usual rotation and reflection transformations of the self-organising map). We can use these vectors just to test the topographic properties of the greycode scheme used



(a)



(b)

Figure 5.9: Single synapse statistics and clustering diagram for grey coding.

ie we use single time step sequences consisting of each of the above pattern set. Two typical results are:

30	00	10	20	33	32	02	03
33	03	13	23	23	22	12	13
32	02	12	22	20	21	11	10
31	01	11	21	30	31	01	00

It can be seen that there is some locally correct (but this is not consistent from simulation to simulation), but not globally correct topographic structure. The reason for this is that vectorially, the separation between the vector with label 00 and that with label 01 is the same as that between 00 and 30. In such a binary coded case, the Hamming distance and the Euclidean distance are the *same*. Hence Euclidean distance relationships between vectors with real components (our co-ordinates in this case) are not preserved in their grey-code representation. The meaning of the position of a particular vector component is not taken into account by the distance matching algorithm as the Hamming distance can only measure the number of places in which two vectors differ [24]. Topological information is thus not preserved and there are multiple correlations between vectors which should ideally *not* have such correlations.

5.4 'Triangular' Coding of Topology

We want a coding scheme that is similar to the grey-coding, but which has an unambiguous global form. This coding must have a restricted set of correlations between vectors that we want to be topologically close. Consider now the following coding for (x, y) positions of a 4×4 grid, based on a higher dimensional weightspace of six bits, three per dimension:

(0,0,0,0,0,0)	(0,0)	(0,0,1,0,0,0)	(1,0)
(0,0,0,0,0,1)	(0,1)	(0,0,1,0,0,1)	(1,1)
(0,0,0,0,1,1)	(0,2)	(0,0,1,0,1,1)	(1,2)
(0,0,0,1,1,1)	(0,3)	(0,0,1,1,1,1)	(1,3)
(0,1,1,0,0,0)	(2,0)	(1,1,1,0,0,0)	(3,0)
(0,1,1,0,0,1)	(2,1)	(1,1,1,0,0,1)	(3,1)
(0,1,1,0,1,1)	(2,2)	(1,1,1,0,1,1)	(3,2)
(0,1,1,1,1,1)	(2,3)	(1,1,1,1,1,1)	(3,3)

This is a 2-d concatenated form of so called *thermometer coding* [3] [21]. Now, each position representation has a greatly restricted number of nearest neighbours e.g. the corner position $(0,0)$ has a representation that is (equally) topologically close to $(0,1)$ and (1,0). These are exactly the two vectors that it should be adjacent to. Similarly, an interior point such as $(2,2)$ has only *four* nearest neighbours: (2,3),(1,2),(2,1) and (3,2). This coding thus preserves the Euclidean to Hamming transformation. The results of self-organising these vectors on to a 4×4 network is

(3,3)	(2,3)	(1,3)	(0,3)
(3,2)	(2,2)	(1,2)	(0,2)
(3,1)	(2,1)	(1,1)	(0,1)
(3,0)	(2,0)	(1,0)	(0,0)

i.e. a perfect mapping. The number of weights required is $O(M) = 2(M-1)$ per node which is much more preferable to the $O(M^2)$ weights used in the fully connected model.

5.4.1 Isomorphic Codings

The 3-bit coding scheme is one of a set of equivalent coding schemes for representing 4 topological scalars. They are generated from the following rules:

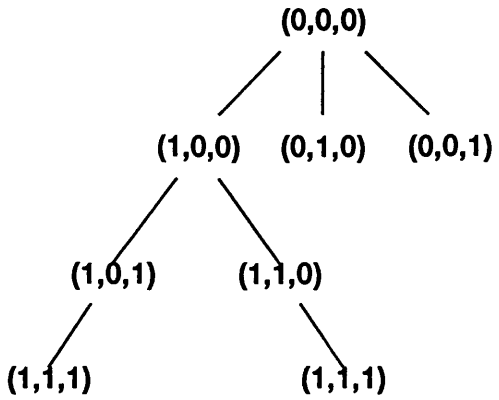


Figure 5.10: Triangular coding isomorphisms.

1. Start with the vector (0,0,0)
2. Generate all vectors that differ from the parent by 1 bit.
3. Form a further generation by changing another bit. Do not repeat any previously generated vector from any part of the tree above.

The partially completed tree of vectors generated by this algorithm is shown in Figure 5.10. In this instance, the path ending in the second terminal represents the familiar triangular coding. All six codings can be generated by permuting the three column vectors $(0,0,0,1)^T$, $(0,0,1,1)^T$ and $(0,1,1,1)^T$.

5.4.2 Hierarchical Classification Using Triangular Coding

We now consider sequences of length two, comprised of the above set of 6-d training vectors. Some initial simulations carried out on a 16×16 grid for all 256 possible sequences showed only a mapping success rate of 58%. A simpler training set was used (sixteen, length two sequences comprised of (0,0) and each of the above set) to investigate this poor result and it was found that a *rectangular* grid of 4×6 neurons produced a near perfect topological mapping of this reduced set of sequences (See Figure 5.11).

(00,33)	(00,32)	(00,31)	(00,30)
(00,23)	(00,22)	(00,21)	(00,20)
(00,13)	(00,12)	(00,11)	(00,10)
(00,03)	(00,02)	(00,01)	#
#	#	#	#
#	#	#	(00,00)

Figure 5.11: Labelled output layer for reduced set of sequences.

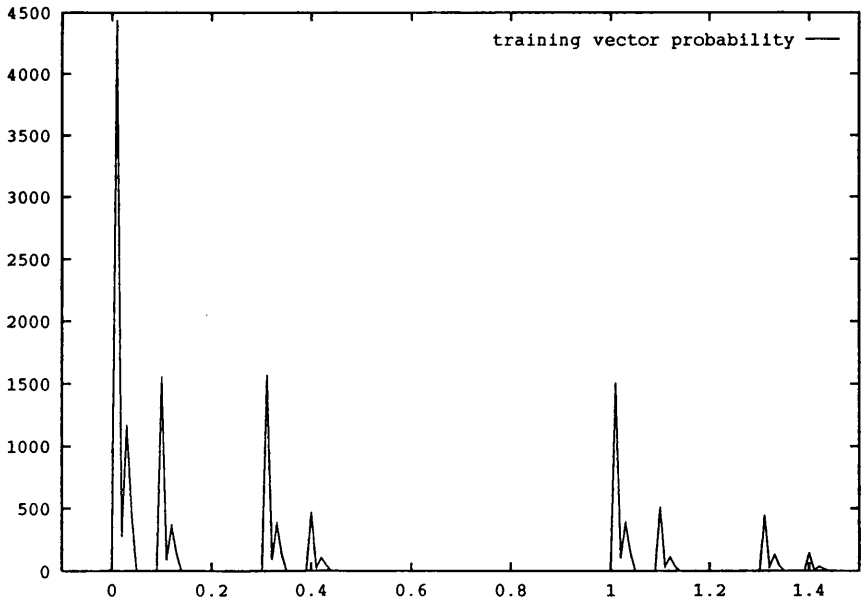


Figure 5.12: Single synapse statistics for triangular coding.

The reason for the increased difficulty in mapping sequences of this coding is that the different synapses are *not equiprobable* to receive an on signal. Thermometer coding is not an *equal weight code* [24] meaning that representations of increasing magnitude scalars have increasing number of set bits, and these bits remain on. A particular synapse will have the elements of one of the coding's three column vectors (recall these are $(0,0,0,1)^T$, $(0,0,1,1)^T$ and $(0,1,1,1)^T$) as its training set. For example, lets say the training set is {0,0,0,1}. Thus $P(1) = 1/4$ and $P(0) = 3/4$. The probabilities for possible sequences of length two are thus $P(\{0,0\}) = 9/16, P(\{0,1\}) = 3/16, P(\{1,0\}) = 3/16$ and $P(\{1,1\}) = 1/16$ i.e. in the ratio $9:3:3:1$. Figure 5.12 shows the pattern density statistics of a single synapse exposed to the training set of our example. The measured ratios in this simulation were 9.07 : 3.11 : 3.11 : 1.00 (trace decay value was 0.3). The symmetry used in the trace analysis of the previous chapter (which assumed a 50% probability of an active input) is thus broken with a resultant 'strain' on the weightspace. Clearly, if the four patterns were presented with the above frequency, then there would need to be *sixteen* nodes to produce the correct mapping.

When the geometry of the 16×16 grid was changed to 30×20 (i.e. an increase in the number of nodes by a factor 2.34) the capture rate of sequences of length two rose to 74%. This rose to 88% for a grid of 40×30 nodes (4.69 times the desired number of nodes for the job).

5.5 Super-Lattice Networks

It is clear that complete feedforward of large dimensionality vectors to increasingly large networks is a poor strategy for abstraction. It is unreasonable to expect one single self-organising layer to be able to accurately process a geometric increase in the number of training patterns at each level of abstraction. Each layer is essentially *duplicating* the information from the previous layer.

A very interesting approach to the use of hierarchical Kohonen maps is the

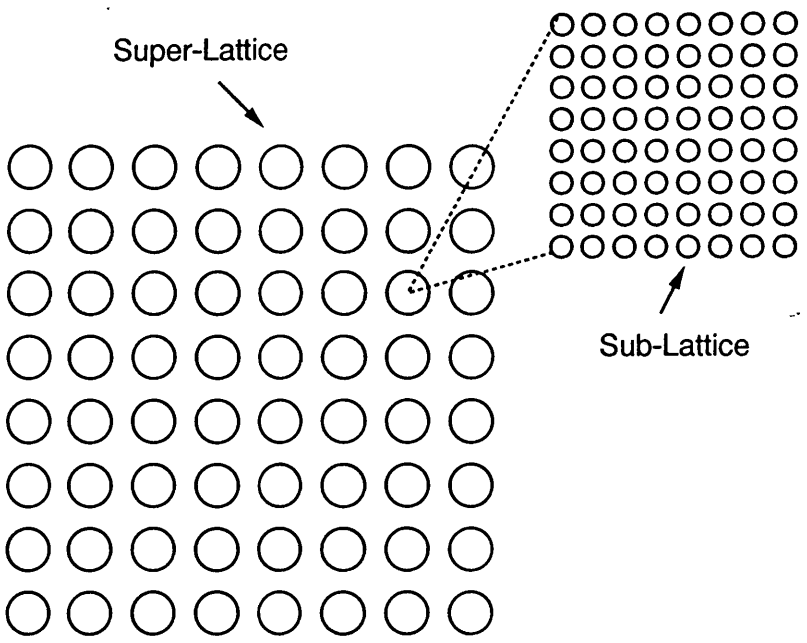


Figure 5.13: The super-lattice architecture.

'super-lattice' of Martinetz and Schulten that they applied to controlling a robotic arm and its gripper [43][62].

The form of the network consists of a normal lattice (in the robot learning example a three-dimensional one) to which each node is assigned a *sub-lattice* (See Figure 5.13). The main or *super-lattice* self-organises its weights in the normal way, but then the training vector is also transferred to the winning node's sub-lattice. The sub-lattice then provides a local expansion of the weight space in the vicinity of the super-lattice weight vector value.

The big advantage of this scheme is the searching procedure scales in a much less time-expensive way. For example, if N_{super} is the number of nodes along one dimension of the super-lattice and N_{sub} is the number of nodes along one dimension of each sub-lattice, then the search time will be

$$t_{search} \sim N_{super}^{p} + N_{sub}^{q} \tag{5.2}$$

where p is the dimensionality of the super-lattice and q is the dimensionality of each sub-lattice. If all nodes were in one network, then the search time would scale as $N^p_{super} \cdot N^q_{sub}$. Of course, each sub-lattice could potentially have a sub-lattice of its own.

The learning algorithm is altered to reflect the new hierarchical structure; there is a neighbourhood defined for the super-lattice and a second for all neurons p in each sub-lattice s. Furthermore, the set p is allowed to contain *neighbouring* subnets' neurons so that there is topographic continuity in the subnets. The effect of the winner on these other neurons is scaled by distance accordingly, determined by the super-lattice neighbourhood. The super-lattice and and sub-lattices are thus learnt concurrently.

This kind of structure would be eminently applicable to the synaptic trace model. The sequence length limit of four for the single layer model would not apply in such a tree-structured network as the theoretical separation of sequences in pattern space could be expanded by progressively more localised sub-lattices.

Interestingly, similar hierarchical structures exist in the visual cortex of higher animals [62][51]

5.6 Summary of Chapter 5

Different strategies for coding the information passed between hierarchical layers that would retain topographic information were discussed and reviewed. These included co-ordinate passing and Kangas' 'response'. The role of clocking was discussed, along with the meaning of learning at every time step. The results of a naive fully connected system were then shown by simulation to be poor.

An enforced output spectrum model was then presented, whereby the output activity around the winning node forms a 'patch' which is used to train a higher layer. This was shown by simulation to give much improved clustering performance. A 'grey-code' encoding was then presented and its failure discussed in terms of

the Hamming-distance vs Euclidean-distance preservation conflict. A concatenated form of 'thermometer code' was then presented which has the desired Euclidean-distance preserving properties, but is not an equal-weight code which would ideally be required for the trace architecture.

A discussion of the super-lattice model of Ritter and Schulten was then presented, together with the suggestion that this would be a good basis for an hierarchical trace architecture model due its local expansion of the weightspace.

Chapter 6

Discussion and Conclusions

The self-organising map is a very successful neural network algorithm; it has been applied to many problem domains, including speech recognition, optimisation and robot control. Its primary strengths are its simplicity of definition and its robustness under a wide variety of learning parameters. These are made even more attractive by the observation that it displays many of the properties of biological topographic mappings.

Kohonen himself has in the past described using 'rules of thumb' for the initial learning parameters of the self-organising map, and for their rates of change during learning. These are usually gleaned from experience of using the SOM. We have presented here an extended model of the SOM which addresses the problem of learning parameter time course and how this may be determined dynamically. We have developed this in terms of a sequence of stable states. These states are reached during learning periods of constant parameter values (learning rate and neighbourhood size), these parameters restricting the 'volume' of pattern space that can currently be effectively mapped. A simple metric was then used to identify the stable states and allow a transition to a new, smaller learning rate and neighbourhood size. We then showed how this model performs under standard tests of the self-organising map through simulations. Furthermore, we have shown that the new parameters, the transition decay constant λ and the smoothing constant δ, introduced by the model are more general than the parameter (an externally imposed time scale $T_{1/2}$) that they replaced. They can be taken 'off the shelf' and applied unchanged to a variety of scenarios, both performing benchmark examples like mapping the unit-square and more complicated inputs requiring dimensional reduction.

Attempts in the past to introduce temporality into the Kohonen model have primarily consisted of pattern concatenation. The Temporal Kohonen Map of Taylor and Chappell overcame many of the problems of such a time-delay model, primarily through simplicity of architecture and removal of an explicit time-window. The leaky-integrator neurons employed in this scheme utilised a known property of real neurons, that of integration of cell body activity, decaying over a short period of time.

We have shown here that the application of the standard Kohonen law (i.e. rotation of the neighbourhood to the most recently seen pattern) to such temporal nets is not general enough to cope with producing a suitably accurate enough weightspace configuration for binary pair sequences longer than two time steps and that even then the weightspace cannot tolerate the neighbourhood size and learning rate tending to zero at long times.

We have highlighted the need for the system to maintain a history of the actual pattern vectors that comprise a pattern sequence and not just the nodes' scalar activity in response to those presented patterns in the past (as in the model of Taylor and Chappell). We thus provided the motivation to include the integration of incoming signals at the synapses of every node in the network. This forms the 'trace architecture' temporal Kohonen map. We have shown by simulation that the binary pair problem has a stable weightspace even as the learning parameters tend to zero with this model. We have also shown that in principle such traces can distinguish binary sequences to any desired length, although in practice the SOM algorithm itself severely restricts what can be done (in terms of the information overload of dimensional reduction). We presented the case of a 16×16 SOM that learnt almost the complete set of 256 sequences of length four bipolar pairs.

We have investigated ways of implementing multilayer temporal topographic mappings, as a means of further extending the model so that longer sequences can be more easily classified. Various coding schemes have been used to represent the topographic nature of the feeding layer. Coordinate passing was found to be the easiest and most obvious route, but is biologically implausible. We found that imposing a structure on the otherwise unary output vector of the feeding layer met with some success, but was too close to being real valued to provide effective discrimination of temporal sequences.

The triangular coding scheme has certain desirable properties

- It is binary valued
- It is unique (subject to a simple set of transformations) for a given number of bits
- It has an unambiguous topographic meaning

We saw however, that necessarily the coding scheme has different probability spectrums on the different synapses (for equiprobable occurrence of a particular topographic position). This means that the weightspace is distorted to reflect this new probability spectrum and hence more nodes are required to produce a more complete mapping.

Feedforward to ever larger layers seems to be a poor policy for abstraction. Instead, some architecture along the lines of the Super-Lattice of Martinetz and Schulten seems appropriate. Such an architecture frees any particular layer from having to cope with too much information to accurately self-organise and scales well with regard to searching. Work is under way to implement such a model.

In conclusion, the Kohonen map remains a focus of interest because of its links to observed biological organisation and information processing. It is the most biologically plausible of the popular artifical neural network algorithms. This work has presented two ways in which the algorithm could be extended in the direction of increasing flexibility and biological plausibility: the replacement of an externally imposed time scale for learning parameters and the inclusion of temporal features. The temporal Kohonen map has aspects still to be explored, but seems likely to play a role in biological information processing, since some mappings of this kind must be used whenever it is necessary to transform temporal information into a spatial representation. In any case it is a worthwhile topic for future research into artificial neural systems, being a significant conceptual extension of Kohonen's original model.

Appendix A

An Object-Oriented Environment For Research In Neural Networks

A.1 Object-Oriented Programming for Neural Networks

A.2 Introduction

The Spectrum of 'Neurosoftware'

The huge interest in neural networks over the past few years has spawned much so called *neurosoftware*, that is programs or programming environments aimed at simulating neural networks.

Much of this software has been aimed at the business community. A package might consist of a graphical front end through which the user can select one of a fixed number of neural network models to analyse his/her data. Financial institutions are interested in such topics as credit risk or share prices. Hence, such packages are restricted to being data-processing tools, and do not offer the researcher the flexibilty of constructing and exploring new neural network models.

Programming environments exist to provide a 'toolkit' approach to research into neural networks. They may offer the researcher libraries of algorithms and a way of specifying network topology, for example. AXON, the proprietary language of HNC Ltd, provides such an environment [23]. It is essentially the C language augmented by data structures and programming constructs tailored for modelling neural networks. It however suffers from the following problems:

- It is proprietary - another researcher will need to buy AXON before software can be exchanged. Therefore relatively few people will use the language, unlike C++, for example which is much more widespread.

- Each model has to specified *from scratch.* There is no incremental modification in AXON.

The PYGMALION environment is an ambitious project aimed at producing a

full neural network environment, or platform. It is aimed at both the application user and also the researcher/developer. It suffers in trying to be too general and as a result is extremely complicated - it has for example two separate languages, one for high level neural network programming called N and a separate neural network specification langauge called **nC**[44][55].

A.2.1 Why Object-Oriented?

Object-oriented programming offers many advantages to the designer of a software system. These include

- A vehicle for *abstraction*
- Inheritance and the re-use of code
- Polymorphism
- Information hiding

We will look briefly at what the above offer a programmer trying to write *simulations* and more specifically neural network simulations.

Abstraction

Most programming languages provide the facility for some kind of abstraction. Even modern BASIC allows for writing functions and procedures. This could be called *functional abstraction.* However, these functions are limited to processing *built-in* types i.e. integers, strings, arrays of these simple types etc. Functions in languages such as BASIC and FORTRAN must therefore process data explicitly at the level of the primitive data types.

Data-abstraction allows grouping of primitive types to form a new data-structure e.g. a complex number which might consist of two real numbers. This facility is typified by the C language's 'structure' which permits such aggregates. The

programmer will then develop a suite of functions which can operate on this structure e.g. functions to return real and imaginary parts from our complex number.

This is all well and good for such simple examples. However, let us return to our basic question: *How should we model neural networks?* We could go along the path of our complex number, developing structures and associated functions for different parts of a neural network. But we would then find that these elements would not *fit together.* They would all be stand-alone. If we want them to communicate, it's back to functions passing built-in types again. We want to be able describe the building blocks of our simulation, be they neurons or layers etc, in a *generic way.*

Inheritance and Code Re-use

Inheritance is the lynchpin of object-oriented design. It is a framework which allows us to model some entity in a *generic* way and then to go on and describe specific instances of that entity. In our problem domain of neural networks, it allows us to define a *neuron* and what data and functions will in general be associated with neurons. These functions are called *virtual functions*, meaning that they only need to be *declared* as belonging fundamentally to our generic definition. Each specialized form of neuron, *derived* from our generic neuron, can then provide a *definition* of what these functions do, or it can *inherit* the definition from the generic neuron if it is provided.

Virtual functions thus provide an *invariant interface* to a set of related (derived) data-types. If you know this interface, you can communicate with any subsequent version of the type. For example, we can form lists of different types of neuron and then treat them in a homogeneous way. In C++, this is achieved by implicit conversion to base-type of pointers, objects and references of derived types i.e. a derived class can be assigned to any of its public base-classes without requiring an explicit cast [40, pp298]. What this means is that it is *safe* to say that an integrating neuron, for example, is *just* a neuron. This is because the integrating neuron will

automatically have an ordinary neuron as part of its internal structure. If then our generic neuron has a function `foo` associated with it, then so will the integrating neuron. The C++ language automatically matches up the correct version of `foo` to use on the type of neuron being considered.

Inheritance means that anything that is unchanged in a derived type is automatically carried through to that new type. Thus, if we are deriving from a class that has ten functions and we want to change the operation of one of them, then we only have to provide the code for that one function. All the rest are automatically included in their original form. This means lots less typing! New types can easily be created from 'off the shelf' ones and tailored for individual use.

Polymorphism

Polymorphism is the facility for a language to have consistent syntax and semantics for different types. For example, we can say `a+b` where `a` and `b` are two reals, or integers or strings or two *anything.* All we need to remember is that we can add these types together. This extends to the whole spectrum of operators and user-defined functions e.g. `sqrt` can be `defined` to have a meaning for real numbers, complex numbers etc. The top level source code is thus vastly more comprehensible by humans and allows the programmer to concentrate on the *logic* of the program rather than the implementation details.

Information Hiding

The C++ language allows the programmer to control the visibility of data members and member functions within a class. This means that the end user can be 'shielded' from the internal representation of a data structure and be denied access to state that might be dangerous to access directly. Furthermore, provided the *interface* to a class remains invariant, then the internal structure and internal functions can be changed without having to rewrite further code that depends on that interface.

Summary

The features of an object-oriented language discussed above provide a powerful and descriptive framework for modelling neural networks. There can be a close semantic mapping between say the model of a neuron and its C++ class description [14]. Furthermore, the incremental specification change that inheritance can provide together with invariant interfaces makes for more readable and intuitive code and code that is easier to write and maintain. In the following section we will explore in more detail an object-oriented model for describing neural networks.

A.3 Basic Abstraction Model

A.3.1 Layers and Neurons As A Basic Building Block

A lot of neural networks can be discussed productively in terms of *layers* i.e. a collection of *neurons* that have a clear role in the computation process (e.g. input layer, hidden layer, output layer) and sets of inputs and outputs. *Networks* can then be defined in terms of a suitably *connected* set of such layers. Hecht-Nielsen calls such building-blocks *slabs* [23]. Figure A.1 shows a basic model for describing neural networks, in terms of *generic* layers that contain *generic* neurons and a connection protocol for linking these layers. A collection of layers will thus constitute a neural network.

We will now examine how to model neurons, layers, networks and other objects associated with performing a neural network simulation. All implementation is in C++.

A.3.2 Defining An Abstract Super-Class for 'Neurons'

What are the defining properties of our generic neuron?

Data connected with the neuron might be:

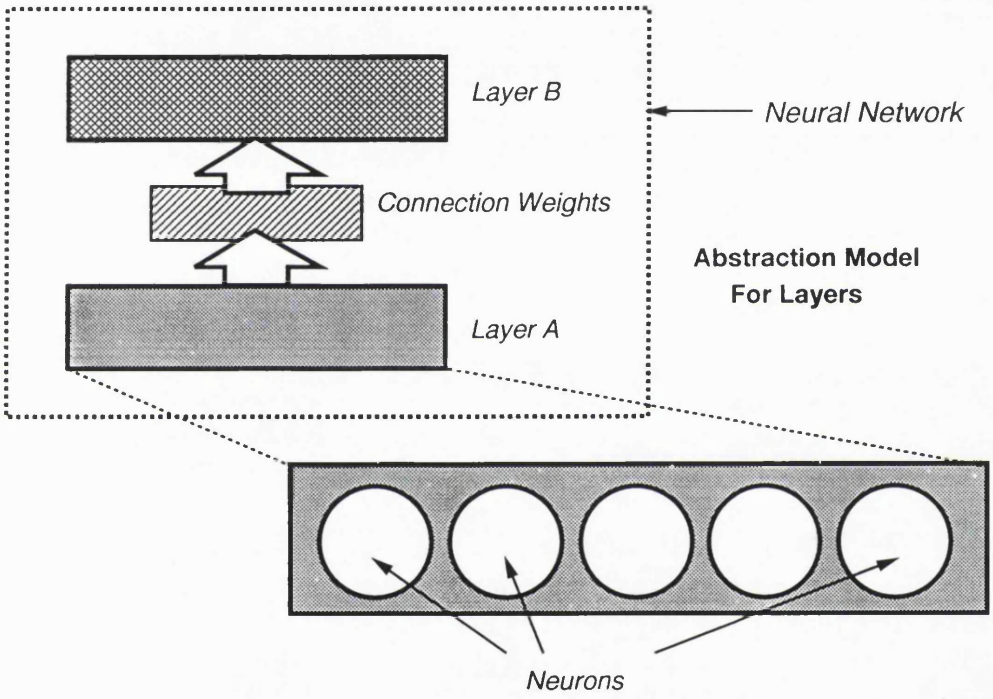


Figure A.1: Abstraction Model For Neural Networks

- An activity
- An output
- A threshold value
- A description of its inputs

Functions connected with the neuron might be:

- A calculation function e.g. weighted summation of its inputs
- A transfer function
- Functions to access the neuron's state
- A reset function i.e. clear activity etc

So, in C++ we have

```
class Neuron
{
protected:
        double activity;
        double output;
        double threshold;
        Layer* parent;
        int NeuronID;
        // etc
public:
        virtual void Update() = 0;
        virtual double Transfer(Vector& I, Vector& W) = 0;
        double GetOutput();
```

```
        // etc

};
```

where Update and Transfer are *pure* virtual functions i.e. the abstract class provides *declarations* of these functions, but provides no default implementation. parent is a pointer to the Layer where the node is situated. NeuronID is just a tag-number that is used in locating the neuron's weight vector.

Update and Transfer Function Definitions

The current implementation is such that each neuron has a pointer to its *parent* layer. The neuron then runs through each set of connections of the parent and applies the transfer function to the relevant weight and input vector.

Here is an example of the Update and Transfer functions for a weighted summation binary decision node, here called BDN_Neuron:

```
void BDN_Neuron::Update()
{
        SiteIterator list(parent); // Declare iterator
        Site* site;                // Site object from iterator
        activity = 0;              // clear current activity

        while(site = list())       // weighted sum from connected layers
                activity +=
                        Transfer(site->InputVector() -
                                site->weight[NeuronID]
                                );

        output = Threshold(activity);     // perform threshold
}
```

```
double BDN_Neuron::Transfer(Vector& I, Vector& W)
{
        return I*W;         // returns dot product
}
```

A.3.3 Layer class definition

A Layer class is the *container* for a collection of neurons. It is comprised either of a link-list or array of Neuron pointers (and the number of nodes), but more importantly defines the following virtual functions:

```
void Train();
void Run();
Site* ConnectFrom(Layer* from_layer);
```

These functions implement the *learning algorithm*, the *operating algorithm* and the *connection protocol*. Train and Run call the updating and transfer rules on all neurons in the layer.

Layer Connection Protocol and Site Objects

The ConnectFrom function provides a connection *from* the supplied layer to the calling layer and returns a pointer to an object of type Site. A Site contains the *from_layer* pointer and also a connection weight Matrix object. It has access functions for these relevant objects. Each layer maintains a list of Site objects so that on a training or run cycle, there is complete access to all relevant weights and inputs. A weight matrix may be read in from a file if required.

Currently, the default connection produces *complete connectivity*, but there is no requirement for this. The implementation is being altered to read in a *connectivity matrix* from an istream.

The protocol allows *self-connection* i.e. recurrent networks to be specified in this way.

Epilogue and Prologue

Each `Layer` can specify an `Epilogue` and `Prologue` function. These functions perform housekeeping functions at the beginning and end of a simulation e.g. opening data files, checking for and initialising connection weights etc. The default operation for both of these functions is to do nothing.

A.3.4 Network class definition

A `Network` is the container for collections of layers. The principle functions that the class declares are

```
void Train();
void Run();
void RunSimulation();
```

The `Train` and `Run` methods call the corresponding layer methods for each layer in the network, in the order that layers were added to the network i.e. the *order* of the components in the network determines information propagation between layers.

`RunSimulation` is the top level function of the simulation environment. It manages structured training sessions (see Section A.5.2 below) and calls the epilogue and prologue functions on all constituent layers.

A `Network` maintains a list of its component layers. Each layer is added along with a tag-name to allow associative manipulations to be performed e.g. `Network` implements a `Site* Connect(String& toLayer, String& fromLayer)` function, where the arguments are the tag-names of the layers to be connected together. Duplication of tag-names when adding a new layer causes the simulation to exit in error state.

A.4 The LEDA Package

The implementation described in the previous section is somewhat inefficient, as each neuron has to go through its parent layer to find its weight. It also assumes complete connectivity between layers which makes the model much less general. The implementation is currently being updated along the same lines as Näher's LEDA (Library of Efficient Data types and Algorithms) package [50]. This is a general purpose set of classes, implementing basic structures such link-lists, sorting algorithms etc, but more importantly, a set of classes for modelling *graphs*. The package has been used by Fritzke to implement the 'growing cell structure' Kohonen maps described in [17][18].

Fritzke [19] makes the obvious mapping between graphs and neural networks:

edge	$\leftrightarrow$	weight
vertex	$\leftrightarrow$	neuron
graph	$\leftrightarrow$	neural-network

Following this framework, each neuron should maintain a link-list of weights which in turn contain a pointer to the neuron feeding it. The weightspace is thus stored directly as edges in a graph and the there is no need to consider their storage in the layer structure itself. Arbitrary connectivity is thus natural to this model.

A.5 Classes for Simulation Support

A.5.1 Data Sources and Input Layers

The simulator is of little use without specifying a source of input patterns. The `DataSource` class specifies the form to which data is supplied to a network, typically via an `InputLayer` (See Figure A.2).

A `DataSource` supplies a stream of patterns, one for each cycle of the network. These may originate from 'hard-wired' generators or present sets of patterns read

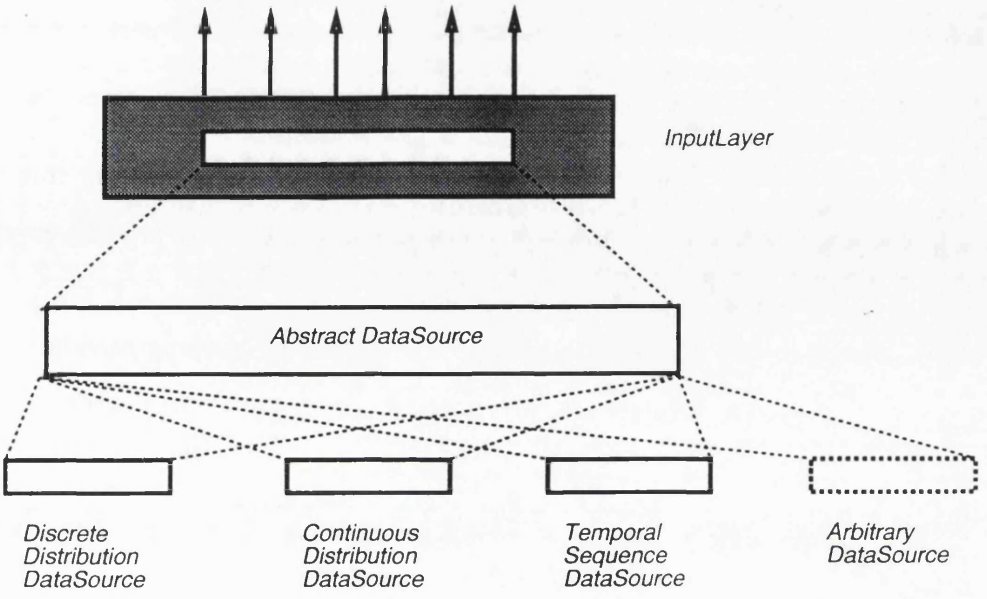


Figure A.2: Data Presentation Model

from an `istream`. The `DataSource` is designed to deal with *temporal data* i.e. it presents *sequences* of patterns. Hard-wired sources usually present just single patterns according to some probability distribution function and the `DataSource` can be interrogated to see whether the data actually has any temporal structure associated with it.

An `InputLayer` is a derived class of `Layer` and serves as the interface between a `DataSource` and the rest of the network i.e. it makes sure that the data stream is updated, but otherwise provides the same functionality of a normal layer.

A.5.2 Training Sessions

A network has a list of `TrainBlocks`. These specify what action should be performed on a particular layer i.e. one of

1. `train`

2. run

3. dormant

where dormant means that the particular layer should be ignored in this training session. This is useful for training hierarchical networks where later layers may not be used in training the earlier ones.

The TrainBlock also specifies the duration of the training session in epochs. The simulation will abort if there is not at least one valid TrainBlock.

A.6 An Example: Defining A Layer To Model Kohonen Networks

In this section, we examine a programming model of a Kohonen layer. The extensions to the basic Layer type required are: A description of the geometry i.e. rows and columns and a function to index the linear list of the basic layer type, a function that implements the competitive search for a winner and suitable training algorithm function.

The basic Neuron also needs to be modified. Its Transfer function will be one that sets the neuron's activity to some function of the Euclidean distance between input vector and weight vector.

```
void KohonenNeuron::Update()
{
        SiteIterator list(parent); // Declare iterator
        Site* site = list(); // only one connection  allowed

        activity =
                Transfer(site->InputVector() -
                        site->weight[NeuronID]
                        );
```

```
}

double KohonenNeuron::Transfer(Vector& I, Vector& W)
{
        return mod_sqrd(I-W); // mod_sqrd calculates
                              // modulus squared
}
```

The connection protocol is changed so that a Kohonen layer can only have one set of connections in keeping with the specific meaning of weights in a self-organising map. Thus, each simple Kohonen layer must have one and only one `ConnectFrom` call made on it. The `Prologue` function for the Kohonen layer then checks that a weightspace exists and initialises it to small, random values.

Processing Order

The processing of a Kohonen layer thus takes the following course:

1. Each node calculates its separation from the input vector. This is then defined as the activity.
2. Select the winning node for the competition. Set the output of the winner to 1, all others to 0.
3. If training, adjust all weights towards the winner.

A.6.1 Labelling Function

The `KohonenLayer` class provides a function `AttachLabels` which is invoked by `KohonenLayer::Epilogue` i.e. at the end of a simulation run. It communicates with a specified `InputLayer` and interrogates that layer's `DataSource` to check for discrete patterns that have a label associated with them. If the data is of this labelled form, then `AttachLabels` assumes control of the *entire network* and replays

the entire sequence set of the DataSource, attaching the label of each sequence to the maximally activated node.

A.6.2 Deriving a Temporal Kohonen Layer

It is now a trivial matter to form a TKM from a standard Kohonen layer. All that needs to be changed is: addition of a decay rate to Kohonen neurons and change their Update functions to integrate their activity. We will thus derive the classes

- `class TempKohonenNeuron:public KohonenNeuron`
- `class TempKohonenLayer:public KohonenLayer`

and the new Update function becomes

```
void TempKohonenNeuron::Update()
{
        SiteIterator list(parent); // Declare iterator
        Site* site;                // Site object from iterator
        double new_activity = 0;   // a variable for new input activity

        while(site = list())  // weighted sum from connected layers
                new_activity +=
                        Transfer(site->InputVector() -
                                site->weight[NeuronID]
                                );

        activity = decay_rate*activity
                + new_activity; // integrate
}
```

The constructor for the temporal Kohonen class then just needs to supply the layer with temporal Kohonen neurons. The basic FindWinner,Train and Run functions are all identical and can simply be inherited. The modifications require only

about twenty lines of code (the temporal classes must supply access functions for the new decay rate parameter).

A.7 Building a Front-End for Simulation

A.7.1 The Two Levels of Simulation Support

The construction of derived Layer classes as described above provides a consistent and reusable method of programming simulations. However, having to hand code a complete simulation in C++ can still be a lengthy and error prone task. For example, if we wrote a simulation for running a TKM connected to an input layer, we would need not only to construct these layers, but then to set all the relevant parameters such as network geometry, learning rate, neighbourhood size etc. These parameter values need to change in the course of performing many experiments. We can thus identify two levels at which a researcher will interact with the development of a simulation:

- *Macroscopic* This is the development of new C++ types.
- *Microscopic* This is the setting of run time values for particular parameters.

It is the job of the *front-end* to provide the microscopic interaction level. It allows the rapid exploration of a model through easy access to its parameters.

A.7.2 Parsing Member Function Approach

Suppose we want to be parse the parameter values for a Kohonen layer. All we will specify are values for the learning rate and the neighbourhood size. We will ignore all other parameters for the purpose of clarity. The form of the scriptfile which contains this information might be:

```
KohonenLayer example
```

```
{
alpha = 0.2;
radius = 4.0;
}
```

Suppose that this scriptfile is lexically split up into the following token stream: 'KohonenLayer', 'example', '{', 'alpha', '=', '0.2', ';', 'radius','=', '4.0', ';' and '}'.

We can imagine defining two virtual functions, one for the header 'KohonenLayer example' and one for the body part, i.e. the parameter settings inside the curly brackets (the opening and terminating brackets are matched by the non-virtual `lbracket` and `rbracket` functions). A function `parse` to parse the parameters would then consist of calling these virtual functions on the `this` pointer:

```
Layer::parse(TokenList TL)
{
        header(TL);

        lbracket(TL);
        body(TL);
        rbracket(TL);
}
```

Thus, the header function would check that the type of the layer was correct and to ascertain the tag-name of the layer for connection purposes. It would then call any relevant functions to set the tag-name etc. The body part would then continue to match input tokens for the parameters until a terminating brace was encountered.

A derived class would reimplement both header and body functions. The former would alter the type check and the latter may typically call its namesake in the base class eg.

```
Derived::body(TokenList TL,int offset = 0)
{
        int pos = this->Base::body(TL);
        int offset;
        .
        . // class specific code goes here
        .
        return offset;
}
```

where the argument pos serves to pass on the position in the token list reached.

Thus, in this methodology, inheritance of parsing behaviour is achieved by the derived class explicitly calling a namesake function in the base class. Having the parse function as a member of class Layer is fine for a fixed network structure i.e. one layer of type A, one layer of type B etc which the parsing function is explicitly invoked upon. What we want, however, is to be able to specify the order and type of simulated network components as part of the microscopic level, that is *at run time.* Rules for adding layers must thus be capable of being applied in variable order and the scheme discussed does not offer a route to achieving this. Furthermore, we want to be able to perform functions other than just construction of a layer, such as specification of layer inter-connections, in a homogeneous fashion. We thus look to abstracting the microscopic level in terms of very general *network rules.*

A.7.3 Network Rules

We introduce a `NetRule` abstract class which matches a set of internal patterns and then performs an operation on a `Network`. Our top-level simulation program then consists of creating a bare network, reading in a scriptfile from the outside world and then applying all possible rules in a rulebase to the network, according to that scriptfile. In this way, different combinations of layers and simulation parameters

can all be specified at run time. Because all `NetRules` operate on a `Network` object, they can perform any function that a `Network` understands. If further rules are written, all that needs to happen is an update of the rulebase. We shall see that inheritance of behaviour follows very naturally in this approach.

A.7.4 Rules for Run-Time Building of Network Layers

The key idea behind a software-engineering approach to rule parsing is that each `Layer` has a `LayerRule` associated with it. A `LayerRule` is an abstract class derived from `NetRule`, specifically tailored for setting a collection of parameters associated with a particular layer.

Subsequent derivations of a `Layer` use analogous derivations of the corresponding `LayerRule` for that layer. For instance, in our TKM model, we would expect the parsing procedure to be identical to the basic Kohonen layer, except for the addition of a rule to determine the activity decay rate. We note that this rule structure exists *outside* the `Layer` classes themselves. It does not make sense that each instance of a particular layer carries around a complete copy of the parsing information - once constructed, the layer needs no further use for it.

Rules are built up from a few simple primitives. These are `FixedPatterns` which attempt to match an exact copy of themselves, and `Var<class T>`, a template class parameterised by `T` which must have a global function `Convert` defined for it. This conversion function parses a string and decides whether it has a valid interpretation for that type. Instances for the three usual types of `String`,`int` and `double` are provided by default. A variable of type `T` must be passed to the constructor of each `Var<T>` which will then hold the matched and converted value associated with that rule.

`CompoundRules` contain other primitive rules including other compound rules, allowing for recursive pattern matching. `UnBlocks` are derived from compound rules and allow for matching of blocks of rules in an unstructured manner i.e. sub rules

can appear in any order.

The input to rules is in the form of segmented strings i.e. input is read from an `istream` and simple lexical analysis is performed by segmenting at the symbols ' ' (space character), '=', ':',';' and at brackets and newlines. This crude process is all that is required.

Each rule implements an `ApplyRule` function which attempts to match from a given location in the input list. If it matches, then it returns the position in the list where the matching process concluded i.e. the next unused string, and returns -1 otherwise. For valid values, the rules are called recursively on all sub rules. If a valid value reaches the top-level, then the rule parse was successful and all the variables passed into `Var` objects will have values ready to be used in the construction of a new layer.

A.7.5 Layer Rules

A `LayerRule` is the basic building block for parsing instructions about the run time setup of a particular neural network layer. Its structure is simple, consisting of a header and a body enclosed by braces. The body is a rule of type `UnBlock` so that parameters can be specified in any order.

```
<Type>Layer <TagName>
{
        .
        . // the 'body'
        .
}
```

Each new rule derived from this basic structure can insert a sub-rule into the body of the main rule using a function `AppendToBody`. In this way *inheritance* of parsing information is achieved. The deriving rule can also elect to call a `RemoveSubRule` function for any inherited access to state that is redundant in the derived layer.

This function takes a string argument and removes a sub-rule which has that string as its first component (each type of rule has a virtual string comparison operator `int operator==(String& a)` which returns false for non-fixed rules).

Example Sub-Rule

Say we want to match the construct '`alpha = 0.2;`', where `alpha` is a learning rate for example. The constructor for our particular `LayerRule` would then have an entry

```
AppendToBody(new CompoundRule(new FixedPattern("alpha"),
                                        equals,
                                        new Var<double>(alpha),
                                        semicolon
                                        )
                    );
```

which assumes that the rule has a variable called `alpha`. After a successful `ApplyRule` call, `alpha` will hold a valid value for this parameter and can be used in the constructor/access function of the target layer. `equals` and `semicolon` are macros to produce rules that must match '=' and ';' respectively.

A.7.6 Other Kinds of Rule

Other kinds of rule, derived from the `NetRule` class, perform more specific operations. These include a rule to connect two named layers together and a rule to specify training sessions.

A.7.7 Example Script File

In this subsection we give an example file used to run a simulation of an input layer connected to a temporal Kohonen map.

```
#Comments may begin a line or terminate them

#declare an InputLayer
InputLayer layer1
{
neurons = 2;              #number of neurons
seq_length = 3;           #provide sequences of length 3
bipolar = 0;              #use binary data, not bipolar
seed = 29;                #seed for random number
pattern_file_name=none; #use hard wired DataSource
}

#declare a TempKohonenLayer i.e. TKM
TempKohonenLayer layer2
{
cols = 8;                 #8 columns
rows = 8;                 #8 rows
radius = 5;               #neighbourhood size of 5
half_life = 1800;         #parameter half life 1800 epochs
alpha = 0.2;              #learning rate of 0.2
alpha_offset = 0.08;      #learning rate residual of 0.08
patt_len = 3;             #sequences of length three
label_layer = layer1;     #labels from layer1's DataSource
weight_seed = 22;         #seed for weight initialisation
decay_rate=0.4;           #cell body activity decay rate
}

Connect{layer1,layer2}  #join input layer and TKM layer

#declare a training session
```

```
TrainBlock T1
{
epochs = 5000;          #duration of 5000 epochs
layer1:run;             #layer1 will use Run()
layer2:train;           #layer2 will use Train()
}
```

A.7.8 Automatic Rule Compilation

A compiler is being designed that generates LayerRule classes directly from Layer header files. The basic idea is that the programmer will put comments after particular access functions in a header file to indicate these functions will need to be accessed by the simulation driver. These will all have to conform to the naming scheme Set_<VarName>. The compiler will then generate a sub-rule for a variable called 'alpha' of type 'double'. In the processing function, the compiler will generate the call to Set_alpha(alpha) on an object of the correct type.

```
class FooLayer:public <Bar>Layer
{
protected:
        double alpha;
        // etc
public:
        //
        void Set_alpha(double a);        // <ACCESS>
};


class FooRule:public <Bar>Rule
{
protected:
```

```
        double alpha;
        ProcessVariables();

        // etc
};

void FooRule::ProcessVariables()
{
        FooLayer* layer = new FooLayer();

        // etc
        layer->Set_alpha(alpha);
}
```

The compiler currently is no further than the drawing board stage. Clearly, much care will need to be taken over such issues as inheritance and data hiding.

Bibliography

[1] Igor Aleksander. *An Introduction to Neural Computing.* Chapman and Hall, 1990.

[2] Luis B Almeida. A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. In *IEEE First International Conference on Neural Networks*, volume II, pages 609–618. IEEE, 1987.

[3] A Badii, M J Binstead, Antonia J Jones, T J Stonham, and Christine L Valenzuela. Applications of n-tuple sampling and genetic algorithms to speech recognition. In Igor Aleksander, editor, *Neural Computing Architectures: The Design of Brain-like Machines.* North Oxford Academic, 1989.

[4] Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on systems, man and cybernetics.*, (5):834–846, 1983.

[5] Russell Beale and Tom Jackson. *Neural Computing: An Introduction.* Adam Hilger, 1990.

[6] George Bolt. Fault tolerance of lateral interaction networks. In *IJCNN Singapore 91*, volume III, pages 1918–23. IEEE, 1991.

[7] George Bolt. Operational fault tolerance of the adam neural network system. In *IJCNN Singapore 91*, volume I, pages 1–6. IEEE, 1991.

[8] Eduardo R Caianiello. Outline of a theory of thought - processes and thinking machines. *Journal of Theoretical Biology*, (204), 1961.

[9] Geoffrey J Chappell and John G Taylor. The temporal kohonen map. *Neural Networks*, 6:441–445, 1993.

[10] Paul M Churchland. *Matter and Consciousness.* Bradford MIT Press, 1988.

[11] David A Critchley. Stable states, transitions and convergence in kohonen self-organising maps. In *Artificial Neural Networks, 2*, volume I, pages 281–284. North-Holland, 1992.

[12] Hubert L Dreyfus and Stuart E Dreyfus. Making a mind versus modelling the brain: artificial intelligence back at a branch point. In Margaret A Boden, editor, *The Philosophy of Artificial Intelligence.* Oxford, 1990.

[13] R Durbin and D Willshaw. An analogue approach to the travelling salesman problem using an elastic net method. *Nature*, (326):689–691, 1987.

[14] Gary Entsminger. An object-oriented neural network. *AI Expert*, pages 19–23, February 1991.

[15] E Erwin, K Obermayer, and K Schulten. Convergence properties of self-organising maps. In *Artificial Neural Networks*, pages 409–414. Elsevier Science, 1991.

[16] Jerome A Feldman. Connections - massive parallelism in natural and artificial intelligence. *BYTE*, pages 277–284, April 1985.

[17] Bernd Fritzke. Let it grow - self-organizing feature maps with problem dependent cell structure. In *Artificial Neural Networks*, pages 403–408. North Holland, 1991.

[18] Bernd Fritzke. Growing cell structures - a self-organizing network in k dimensions. In *Artificial Neural Networks*, volume II, pages 1051–1056. North Holland, 1992.

[19] Bernd Fritzke. Using a library of efficient data structures and algorithms as a neural network research tool. In *Artificial Neural Networks*, volume II, pages 1273–1276. North Holland, 1992.

[20] Denise Gorse, Trevor G Clarkson, and John G Taylor. From wetware to hardware - reverse engineering using probabilistic rams. *Journal of Intelligent Systems*, 2:1–4, 1992.

[21] Peter J B Hancock. Data representation in neural nets: an empirical study. In *Proceedings of the 1988 Connectionist Models Summer School.* Morgan Kaufmann Publishers Inc., 1988. ISBN: 0-55860-015-9.

[22] Donald O Hebb. *The Organisation of Behaviour.* John Wiley and Sons, 1949.

[23] Robert Hecht-Nielsen. *Neurocomputing.* Addison-Wesley, 1990.

[24] Raymond Hill. *A First Course in Coding Theory.* Oxford University Press, 1986.

[25] Andrew Hodges. *Alan Turing : the enigma.* Vintage, 1983.

[26] John J Hopfield. Neural networks as physical systems with emergent collective computational abilities. *Proc. National Academy of Sciences*, 79:2554–2558, 1982.

[27] Jari Kangas. Time-delayed self-organising maps. In *International Joint Conference on Neural Networks*, volume II, pages 331–336. IEEE, 1990.

[28] Jari Kangas. Time-dependent self-organising maps for speech recognition. In *Artificial Neural Networks*, pages 1591–1594. Elsevier Science (North Holland), 1991.

[29] Jari Kangas. Temporal knowledge in locations of activations in a self-organising map. In *Artificial Neural Networks*, volume II, pages 117–120. Elsevier Science, 1992.

[30] Jari A Kangas, Teuvo K Kohonen, and Jorma T Laaksonen. Variants of self-organizing maps. *IEEE Transactions On Neural Networks*, 1(1), March 1990.

[31] Teuvo Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, (43):59–69, 1982.

[32] Teuvo Kohonen. Dynamically expanding context, with application to the correction of symbol strings in the recognition of continuous speech. In *Proc. Eighth International Conference on Pattern Recognition*, pages 1148–1151. IEEE Computer Society, 1986.

[33] Teuvo Kohonen. Self-learning inference rules by dynamically expanding context. In *IEEE First International Conference on Neural Networks*, volume II, pages 3–9. IEEE, 1987.

[34] Teuvo Kohonen. State of the art in neural computing. In *International Conference on Neural Networks*, pages 1–12. IEEE, 1987.

[35] Teuvo Kohonen. An introduction to neural computing. *Neural Networks*, 1:3–16, 1988.

[36] Teuvo Kohonen. The neural phonetic typewriter. *IEEE Computer*, March 1988.

[37] Teuvo Kohonen. *Self Organisation and Associative Memory.* Springer-Verlag, 1989.

[38] Teuvo Kohonen. The hypermap architecture. In *Artificial Neural Networks*, pages 1357–1360. Elsevier Science, 1991.

[39] Teuvo Kohonen, György Barna, and Ronald Chrisley. Statistical pattern recognition with neural networks: Benchmarking studies. In *Proc. International Conference on Neural Networks*, volume I, pages 61–68. IEEE Computer Society, 1988.

[40] Stanley B Lippman. *C++ Primer*. Addison Wesley, 1989.

[41] Richard P Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, pages 4–22, April 1987.

[42] Zhen-Ping Lo and Benham Bavarian. Improved rate of convergence in kohonen neural network. In *IJCNN*, volume II, pages 201–206. IEEE, 1991.

[43] Thomas Martinetz and Klaus Schulten. Hierarchical neural net for learning control of a robot's arm and gripper. In *IJCNN-90*, volume III, pages 93–100. IEEE, 1990.

[44] M.Azema-Barac, M.Hewetson, M.Recce, J.Taylor, P.Treleaven, and M.Vellasco. PYGMALION neural network programming environment. In *International Neural Network Conference (INNC)*, pages 709–712, 1990.

[45] James L McClelland, David E Rumelhart, and The PDP Research Group. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* MIT Press, 1986.

[46] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[47] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry.* MIT Press, 1969.

[48] Pietro Morasso. Neural models of cursive script handwriting. In *International Joint Conference on Neural Networks*, volume II, pages 539–542. IEEE, 1989.

[49] Pietro Morasso. Self-organising feature maps for cursive script recognition. In *Artificial Neural Networks*, volume II, pages 1323–1326. North-Holland, 1991.

[50] Stefan Näher. *L.E.D.A. User Manual Version 3.0.* Max-Planck-Institut für Informatik, Im Stadtwald D-6600 Saarbrücken.

[51] K Obermayer, G G Blasdel, and Klaus Schulten. A neural network model for the formation and for the spatial structure of retinotopic maps, orientation- and occular dominance columns. In *Artificial Neural Networks*, volume I, pages 505–511. North Holland, 1991.

[52] Barak A Pearlmutter. Dynamic recurrent neural networks. Technical Report CMU-CS-90-196, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213, 1989.

[53] Barak A Pearlmutter. Learning state space trajectories in recurrent neural networks. *Neural Computation*, (1):263–269, 1989.

[54] Fernando J Pineda. Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, 59(19):2229–2232, 1987.

[55] M. Recce, P. V. Rocha, and P. C. Treleaven. Neural network programming environments. In *Artificial Neural Networks, 2*, volume II, pages 1237–1244. North-Holland, 1992.

[56] Mark Reiss and John G Taylor. Storing temporal sequences. *Neural Networks*, 4:773–787, 1991.

[57] H Ritter and K Schulten. On the stationary state of kohonen's self-organizing sensory mapping. *Biological Cybernetics*, (54), 1986.

[58] H. Ritter and K. Schulten. Convergence properties of Kohonen's topology preserving maps: fluctuations, stability, and dimension selection. *Biological Cybernetics*, 60(1):59–71, 1988.

[59] H. Ritter and K. Schulten. Kohonen self-organizing maps: exploring their computational capabilities. In *Proceedings of IEEE International Conference on Neural Networks*, volume I, pages 109–116, 1988.

[60] Helge Ritter and Teuvo Kohonen. Self-organizing semantic maps. *Biological Cybernetics*, (61):241–254, 1989.

[61] Helge Ritter and Teuvo Kohonen. Learning 'semantotopic maps' from context. In *IJCNN-90-WASH-DC*, pages 23–26, 1990.

[62] Helge Ritter, Thomas Martinetz, and Klaus Schulten. *Neural Computation and Self-Organizing Maps.* Addison-Wesley, 1992.

[63] Joaquim S Rodrigues and Luis B Almeida. Improving the learning speed in topological maps of patterns. In *International Neural Network Conference*, pages 813–816. IEEE, 1990.

[64] Frank Rosenblatt. *The Principles of neurodynamics.* Spartan, 1962.

[65] H P Siemon. Selection of optimal parameters for kohonen self-organising feature maps. In *Artificial Neural Networks, 2.* Elsevier Science, 1992.

[66] Graham Tatersall. Neural map applications. In Igor Aleksander, editor, *Neural Computing Architectures: The Design of Brain-like Machines.* North Oxford Academic, 1989.

[67] Graham Tatersall, Paul Linford, and Bob Linggard. Neural arrays for speech recognition. *British Telecom Technology Journal*, pages 140–163, April 1988.

[68] John G Taylor. Spontaneous behaviour in neural networks. *Journal of Theoretical Biology*, 36:513–528, 1972.

[69] John G Taylor. Temporal patterns and leaky integrator neurons. In *INNC 90*, pages 952–955. Kluwe Academic, 1990.

[70] Alex Waibel. Modular construction of time-delay neural networks for speech recognition. *Neural Computation*, 1:39–46, 1989.

[71] Alex Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin Lang. Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37(3):328–339, March 1989.

[72] Gary Whittington and Tim Spracklen. The application of a neural network model to sensor data fusion. In *Applications of Artificial Neural Networks*, volume 1294, pages 276–283. SPIE, 1990.

[73] Gary Whittington, Tim Spracklen, Jon Haugh, and Helen Faulkner. Automated radar behaviour analysis using neural network architectures. In *Applications of Artificial Neural Networks*, volume 1965, pages 44–59. SPIE, 1993.

[74] Ronald J Williams and David Zipser. Experimental analysis of the real-time recurrent learning algorithm. *Connection Science*, 1(1):87–111, 1989.

[75] Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, (1):270–280, 1989.

[76] David Willshaw and Cristoph von der Malsburg. How patterned neural connections can be set up by self-organisation. *Proc. R. Soc. London B*, 194:431–445, 1976.