Investigations into the capabilities of the SDM
and
combining CMAC with PURR-PUSS

A Thesis
presented for the degree of
Doctor of Philosophy
in Electrical and Electronic Engineering
in the
University of Canterbury,
Christchurch, New Zealand
by

Shaun W. Ryan BE (Hons)

University of Canterbury
1996

Abstract

This thesis consists of two sections analysing aspects of associative memories. The first section compares the usefulness, limitations, and similarities of the sparse distributed memory (SDM), the cerebellar model articulation controller (CMAC) and the Hopfield network. This analysis leads in the second section to a proposal for combining CMAC with an a form of robot learning through exploration, the PURR-PUSS system. It is then demonstrated the combination of the PURR-PUSS and CMAC systems produce a system capable of robot control.

There are a number of critical factors in the performance of a neural network as a memory. These include the capacity and the efficiency of the training. Of the three networks considered, the Hopfield network is by far the most common in the literature. In spite of this, this thesis shows that the SDM and CMAC are almost identical and, in fact, have significant advantages over the Hopfield network in terms of capacity. This is particularly evident in the storage of sequences, where the SDM shows a significant improvement over the Hopfield network.

The major contribution of this thesis is the analysis and development of the full potential of the SDM for data storage. The first contribution is a correction of an error in the existing analysis of the capacity of the SDM. The corrected figure is verified both theoretically and experimentally. The second contibution is an improvement in capacity resulting from an alternative method of generating the outputs. Finally, the capacity is further improved, by using an iterative approach to information storage previously employed on the Hopfield network. The latter aproach helps produce a significant advantage in capacity for SDM.

Another contribution of this thesis is the combination of associative memory with the a means of learning through experimentaion. The PURR-PUSS system was originally developed as a means to enable a robot to learn through interacting with its environment. It is shown that its strengths and weaknesses complement those of the CMAC and SDM systems. PURR-PUSS and CMAC are combined and the result is a system which is capable of superior control than either system by itself This is demonstrated through an example, in which the combined system learns to control a ball rolling in a tilting maze of unknown dynamics.

The system begins by learning through random exploration controlled by the PURR-PUSS system. As the knowledge of the environment increases, the PURR-PUSS system is able to successfully achieve goals, although the quality of the control is poor. However the addition of CMAC which in turn learns from PURR-PUSS's movements produces an improvement in the quality of the control.

Table of contents

Acknowledgements

I would like to thank my supervisor, John Andreae, for his guidance, support and patience over the last six years. John always returned my work quickly with constructive criticism, showing that he took time to read it carefully.

I am grateful to the University Grants committee for the scholarship during the first three years of my postgraduate study.

Finally, I am grateful to my family and friends for their support and encouragement throughout my decade at University.

CHAPTER 1

Introduction

Artificial neural networks originated from models of biological neurons (McCulloch & Pitts, 1943). Since then the study of artificial neural networks has grown and it is now an established field of research. There are many different types of artificial neural network and they have many practical applications. This thesis demonstrates the usefulness, and examines the limitations, of two types of artificial neural network which are not currently widely used. These artificial neural networks are called the sparse distributed memory (SDM) and the cerebellar model articulation controller (CMAC).

Some of the practical applications of artificial neural networks include language processing (Sejnowski & Rosenberg 1987; Rumelhart, McClelland et al, 1986), image compression (Cottrell et al 1987), character recognition, (Fukishima 1987, 1988, 1993; Fukishima & Imagawa, 1993), pattern recognition in images (Gorman & Sejnowski 1988; Glover 1988), signal processing (Lapedes & Farber 1987), and servo control (Millar et al, 1988). New applications are being discovered regularly. Some examples of applications of the SDM and CMAC networks are given in this thesis. Chapter 5 describes how the SDM can be applied to a simple speech recognition problem. Chapter 7 describes how the CMAC can be used in a robot control application.

An artificial neural network consists of an interconnected group of units, some of which receive input signals from outside the network, and some which transmit output signals. The units which receive input signals are called *input units* and they receive the input signals along input connections. The units which transmit output signals are called *output units* and they transmit signals along output connections. Units which neither receive input signals or transmit output signals are called hidden units because they do not have any direct connection with the outside. Often the units in an artificial neural network are arranged in layers, as shown in Figure 1.1.

All connections between units are associated with weights. The weight of a connection

determines the extent to which the signal is modified in passing through the connection. Weights may be fixed or variable. Variable weights allow an artificial neural network to store information. The process of storing information is referred to as training. It will be seen later that when an artificial neural network is trained the weights are adjusted according to a training algorithm using example data.

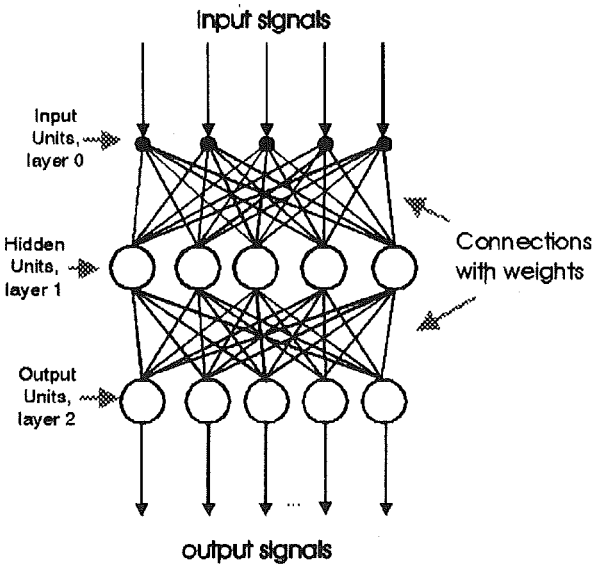


Figure 1.1 An artificial neural network with three layers of units, input units, hidden units and output units. The connections between units have weights associated with them.

The most widely used type of artificial neural network is the multi-layer network using the back-propagation algorithm (Rumelhart, McClelland et al, 1986; Parker 1985), or variations of it, to adjust the weights. The main advantage that the CMAC and SDM have over networks using back-propagation is that the CMAC and SDM can be trained much faster. In a speech recognition example given in chapter 5 an SDM and a back-propagation-based multi-layer network are trained with the same data. The performance of the two networks is the same after training in the speech recognition task. However, training takes only 25 iterations for the SDM and 65000 iterations for the back propagation network. An *iteration* involves adjusting the weights once for every set of data in the training set using the training algorithm. In this example the SDM trains much faster than the back-propagation network. The training speed will not be important in all applications but when it is the SDM or the equivalent CMAC are clearly superior. The main disadvantage of the SDM and the CMAC is that they require more units than a back-propagation network. In the example in chapter 5 the SDM had 500 units and the back-propagation network had only 32.

Before the SDM and CMAC are introduced in chapters 3 and 4 respectively, research on another type of artificial neural network, called the Hopfield network, is summarised in chapter 2. The summary of the work on the Hopfield network is included because some of

this research can be applied to the SDM and CMAC as will now be briefly described.

Chapter 2 describes how the number of patterns that the Hopfield memory can store is limited by the number, N, of binary elements in a pattern. Hopfield (1982) estimated the number of patterns that can be stored to be 0.15N. A mathematical analysis performed by Gardner (1988) shows that by using a different training algorithm it is possible to store up to 2N uncorrelated (random) patterns. This limitation on the number of patterns is significant for large N because with N binary elements in a pattern there are 2^N possible patterns and it is quite likely that an application would need to store more than 2N patterns. For example, if N is 100 then there are $2^{100} \approx 10^{30}$ possible patterns and the Hopfield network could only store 200 of these which is a tiny fraction of the possible patterns.

The capacity limitation of the Hopfield network can be overcome by using the SDM. Kanerva (1988) analyzed the capacity of the SDM and showed that the number of patterns that can be stored depends mainly on the number of location units, s, which can be chosen arbitrarily in the SDM. Chapter 3 introduces the SDM and analyses its capacity to show that it does overcome the limitation of the Hopfield network. An error in Kanerva's analysis of capacity is highlighted and corrected. It is also shown that a small improvement in the capacity can be achieved by using an alternative method of generating the outputs. Chapter 3 then goes on to describes how Gardner's analysis of the Hopfield network can be applied to the SDM with the 2N limit of the Hopfield network becoming the much larger 2s for the SDM. The analysis of the SDM has been published in Ryan & Andreae (1995).

As is demonstrated in chapter 2, the Hopfield memory stores static patterns. For many applications temporal information is critical, as for example in speech recognition. A significant amount of research has been performed on modifications to the Hopfield network which allow it to store sequences of patterns rather than just single patterns. A sequence of patterns can represent a time varying pattern sampled at regular intervals. When the Hopfield network is modified to store sequences of patterns it still has the capacity limitation described above. However, as before, this limitation on the number of patterns which can be stored can be overcome by using the SDM. Chapter 5 describes how the modifications that allow the Hopfield network to store patterns can be applied to the SDM and the CMAC. Using an

SDM, the length of the sequence that can be stored is limited by the number of location units, s, which can be chosen arbitrarily.

Chapter 6 introduces a system called PURR-PUSS (PP for short) which is designed to enable a robot to learn. This system was developed by Andreae and others (Andreae, 1977; Andreae & MacDonald, 1991; Andreae et al, 1993). The PURR-PUSS system is introduced because it is used in Chapter 7 with CMAC (this is discussed in more detail below). Two features new to the PP system are introduced in Chapter 6: learning from random actions and the goal-setting GO command. Chapter 6 shows how, with these new features, PP can learn to control a ball rolling around in a tilting maze.

The control of the ball that PP learns in Chapter 6 is adequate but it is not perfect because the ball often overshoots. In Chapter 7, this same problem is used to demonstrate how PP and CMAC can operate together, with PP providing coarse discrete actions, and CMAC learning from those actions and interpolating to produce smooth, fine tuned movements. The resulting system controls the ball much better than the PP system alone.

CHAPTER 2

The Hopfield Network

2.1 Introduction

The Hopfield network was first introduced by J. Hopfield in 1982. Since then much work has been done on the Hopfield network and its variations (e.g. Hopfield, 1984; Amit & Gutfreund, 1985; Gardner, 1988; Barro et al, 1991; Bauer & Krey, 1991; Bressloff & Taylor, 1991). This thesis compares the Hopfield network to the CMAC and SDM networks (covered in chapters 3 and 4 respectively). All these networks have similar structures and some of the work that has been done on the Hopfield network is also applicable to the SDM and CMAC networks.

The format of this chapter is as follows:

- Because the Hopfield network was first described as a content addressable memory, section 2.1.1 describes what a content addressable memory is.
- Section 2.2 describes the Hopfield network as a layer of N fully connected units. This includes an example of a small Hopfield network.
- Section 2.3 summarises work which has analyzed the capacity of the Hopfield network, and it introduces an iterative learning algorithm which allows the maximum capacity to be realized. This work is applicable to the SDM and CMAC networks.
- Section 2.4 shows how the Hopfield network can be viewed as a two layer network, with connections between the layers. This alternative view of the Hopfield network makes it easier to compare it to the CMAC and SDM, which can also be viewed as having a layered structure.

2.1.1 Content addressable memories

A content addressable memory (CAM) is a memory from which a stored item can be retrieved by presenting an incomplete or noisy version of the stored item. The stored item is the content of the memory. The noisy or incomplete version of the stored item is the input to the memory (also referred to as the address). So the memory is addressed by a version of the content of

the memory; hence the name content addressable memory. Figure 2.1 shows a simple example of what a CAM may do. The CAM in Figure 2.1 has stored in it an image of a grey box on a white background. The stored image should be retrievable by presenting the memory with a noisy version of it, such as the one on the left in Figure 2.1.

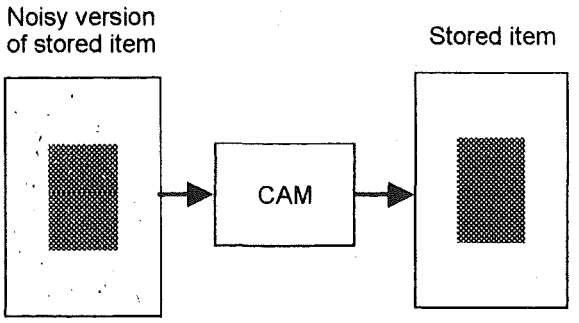


Figure 2.1 In a CAM stored items can be retrieved from noisy versions of the stored item.

The Hopfield network (also referred to as the Hopfield memory) is a content addressable memory. The items which are stored in the memory are patterns of N binary elements. The Hopfield network is described in detail in section 2.2. A brief description of the operation of the Hopfield network follows here.

An input pattern puts the Hopfield network into a state corresponding to that input. From that input state, the Hopfield network retrieves an item stored in the memory by changing the state according to a set of equations (this is explained fully in section 2.2). Eventually the Hopfield network settles into a stable state which corresponds to a stored pattern. The dynamics of the Hopfield network guarantee that it will settle into a stable state.

In an example network (detailed in 2.2.2) two patterns P^1 and P^2, each consisting of ten elements, are stored. The patterns are:

P^1: -1 1 -1 -1 -1 -1 1 -1 1 -1

P^2: -1 1 -1 1 -1 -1 -1 -1 -1 -1

The following pattern is used as an input to the network:

I^1: -1 1 -1 -1 1 -1 -1 -1 -1 1

After the pattern, I^1, has been presented to the network, the network begins to change state and eventually settles into a state corresponding to pattern P^2. The input pattern, I^1 can be viewed as a noisy version of pattern P^1 (four elements are different) or pattern P^2 (three elements are different). In this case pattern P^2, which is closer to the input, is retrieved. This is an example of a content addressable memory; stored patterns are retrieved using inputs

(addresses) which are noisy versions of the patterns.

The Hopfield network will now be described in detail.

2.2 The Hopfield network

The Hopfield network consists of a single layer of fully connected units (see Figure 2.2), i.e. each unit is connected to every other unit and not to itself. Unlike other neural networks discussed in this thesis, the connections between the units in the Hopfield network are bidirectional, i.e. signals can flow in both directions. This means that between every pair of units there is only one connection. Each connection has a weight, which indicates the strength of the connection. There are N units in total so the input and output patterns each consist of N elements, (I_1, ..., I_N) and (O_1, ..., O_N) respectively. Each unit receives a signal from the input pattern and from the other units, and combines these to produce an output signal. The range of values of the input and output signals depends on how the network is configured. Only networks with binary inputs and outputs (elements of {1,-1}) will be considered here. The output of a unit is also referred to as the *state* of the unit, and the outputs from the N units together define the *state* of the network.

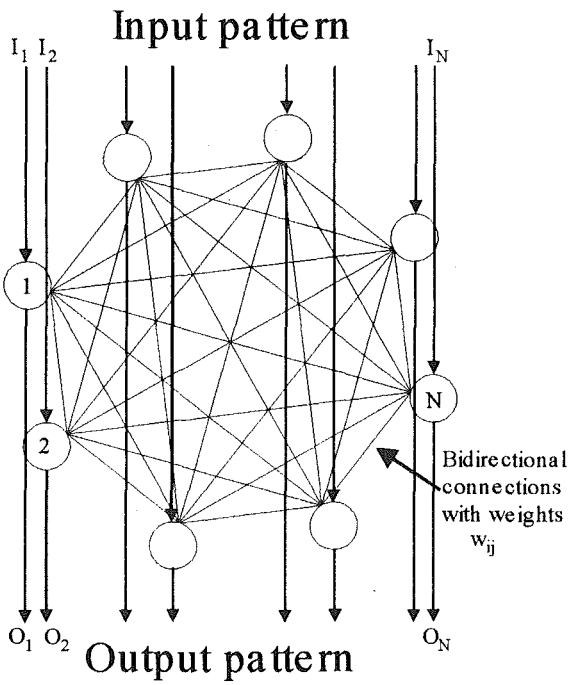


Figure 2.2 The Hopfield network

The Hopfield network can be viewed as having two modes:

- the input mode, where the input determines the state of the units and the connections between the units have no effect:

$$O_i(t+1)=I_i(t) \qquad (1)$$

and,

- the settling mode where the inputs have no effect and in each time step a randomly chosen unit, unit i ($1 \le i \le N$), is updated according to the following expression:

$$O_i(t) = \begin{cases} 1 & \text{if } h_i(t) > T \\ -1 & \text{if } h_i(t) \le T \end{cases} \quad (2)$$

where T is a threshold (typically T=0), t is the time in steps, t= 0, 1, 2, ..., and $h_i(t)$, termed the *local field* of unit i, is given by the following expression:

$$h_i(t+1) = \sum_{j=1}^{N} w_{ij} O_j(t) \quad (3)$$

where w_{ij} is the strength of the connection (referred to as the weight) from unit j to unit i, for i, j=1..N.

A particular pattern, P^l is a stable pattern on the Hopfield network if, after it has been asserted (in the input mode), none of the outputs change when updated according to (2). If the threshold, T, is 0 then the pattern will be stable if, for each unit, the local field has the same sign as its output:

$$P_i^l h_i^l > 0 \quad \text{for } i=1..N \quad (4)$$

where P_i^l is the ith element of pattern P^l, $P_i^l = \pm 1$, and h_i^l is the local field of unit i when the network is in the state P^l. Replacing O_j in (3) with P_j^l, using this to expand (4), and bringing the factor P_i^l inside the sum gives:

$$\sum_{j=1}^{N} w_{ij} P_i^l P_j^l > 0 \quad (5)$$

One way of choosing w_{ij} to ensure that (5) is positive and P^l is stable is to set

$$w_{ij}^l = P_i^l P_j^l \tag{6}$$

This will ensure that each term of the sum in (5) is a square, $\left(P_i^l P_j^l\right)^2$, and so the sum will be positive. When there are N_p patterns to be stored as stable states, P^1, ..., P^{Np} (N_p is the number of patterns to be stored), Hopfield proposed that the weights be given by summing (6) over the N_p patterns, i.e.

$$w_{ij} = \begin{cases} \sum_{l=1}^{N_p} P_i^l P_j^l & \text{for } i \neq j \\ 0 & \text{for } i = j \end{cases} \tag{7}$$

This method of assigning weights, (7), does not guarantee that all of the N_p patterns will be stored as stable states. However if the number of patterns stored, N_p, is low compared to N (N_p<0.15N) then there is a high probability that the patterns will be stored as stable states. This is demonstrated in section 2.3.

The weights specified by (7) are symmetrical, that is the weight from unit j to unit i equals the weight from unit i to unit j, or $w_{ij} = w_{ji}$. Equation (7) shows that there are no connections from a unit to itself by setting the diagonal weights, w_{ii}, to zero. Having these weights zero is one of the requirements that ensures the network will converge towards stable patterns, as described below.

2.2.1 Convergence properties of the Hopfield Network

Hopfield formulated the dynamics of the network in terms of spin glass physics (Amit & Gutfreund, 1985). This allows the state of the network to be viewed as an energy surface. The energy surface has a number of minima which correspond to stable states. Hopfield defined the energy of the system to be:

$$E(t)=-\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}w_{ij}O_i(t)O_j(t) \tag{8}$$

Now it will be shown that when the Hopfield network changes state its "energy" always decreases and the state will keep changing until the energy reaches a local minimum. When the output unit k changes by $\Delta O_k(t)=O_k(t+1)-O_k(t)$ then, using (8), the energy is:

$$\begin{aligned}
E(t+1)&=-\frac{1}{2}\sum_{i\neq k}\sum_{j\neq k}w_{ij}O_i(t)O_j(t)-\frac{1}{2}\sum_{i=1}^{N}w_{ik}O_i(t)\left(O_k(t)+\Delta O_k(t)\right)\\
&\quad-\frac{1}{2}\sum_{j=1}^{N}w_{kj}\left(O_k(t)+\Delta O_k(t)\right)O_j(t)-\frac{1}{2}w_{kk}\left(\Delta O_k(t)^2-O_k(t)^2\right)\\
&=-\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}w_{ij}O_i(t)O_j(t)\\
&\quad-2\left(-\frac{1}{2}\sum_{j=1}^{N}w_{kj}\Delta O_k(t)O_j(t)\right)\qquad \text{since } w_{ij}=w_{ji} \text{ and } w_{ii}=0\\
&=E(t)-\sum_{j=1}^{N}w_{kj}\Delta O_k(t)O_j(t)
\end{aligned} \tag{9}$$

The change in energy is:

$$\begin{aligned}
\Delta E(t)&=E(t+1)-E(t)\\
&=-\Delta O_k(t)\sum_{j=1}^{N}w_{kj}O_j(t)\\
&=-\Delta O_k(t)\,h_k(t)
\end{aligned} \tag{10}$$

When there is a threshold of T=0, $\Delta O_k(t)$ and $h_k(t)$ will have the same sign (see (2)) and so ΔE will always be negative, i.e. the energy function is a monotonically decreasing function. This means the energy of the network will decrease until it reaches a local minimum. The Hopfield network is therefore guaranteed to settle into a stable state.

An example of the Hopfield network is given in the following section.

2.2.2 An example of the Hopfield network

Figure 2.3 shows a simple example of the Hopfield network. In this example there are only N=10 units, and two patterns, P^1 and P^2, are stored on the network. Half of the weight matrix, as calculated by (7) is shown (only half needs to be shown because of the symmetry).

The results of a test are shown in Figure 2.3. In this test, a pattern, T^1 is presented at the input and then randomly chosen units are updated according to (2). A threshold of T=0 is used. As the units are updated the energy of the network, as given by (8), either decreases or stays the same, and the network converges towards one of the stored patterns. In this example the network begins in the state T^1 and then moves through two intermediate states, X^1 and X^2, before settling on P^2.

The units which are updated are listed on the right, in the results section of Figure 2.3. Also listed is the energy of the Hopfield network, E, as given by (8). If the output of the unit changes, then on the following line the new states of all the units are shown, along with the new value of the energy. For example after the test pattern (T^1) is presented the energy is 0. Units 3 and 6 are updated, but there is no change. Then on step 3 unit 5 is randomly selected to be updated. The calculation for the local field, $h_5(3)$, using (3), is:

$$\begin{aligned} h_5(3) &= (-1).(2)+(1).(-2)+(-1).(2)+(-1).(0)+(1).(0) \\ &\quad +(-1).(2)+(-1).(0)+(-1).(2)+(-1).(0)+(1).(2) \\ &= -8 \end{aligned} \tag{11}$$

Using (2), output $O_5(3)$ will be -1 because $h_5(3)$ in (11) is negative. This is shown on the second line of the results, steps 4-6. Also shown is the new value for the energy. The calculation for the change of energy, using (10), is:

Stored Patterns

Pattern P^1: -1 1 -1 -1 -1 -1 1 -1 1 -1
Pattern P^2: -1 1 -1 1 -1 -1 -1 -1 -1 -1

Weights w_{ij} (given by (7), where $w_{ij}=w_{ji}$)

Unit	j=1	2	3	4	5	6	7	8	9	10
i=1	0									
2	-2	0								
3	2	-2	0							
4	0	0	0	0						
5	2	-2	2	0	0					
6	2	-2	2	0	2	0				
7	0	0	0	-2	0	0	0			
8	2	-2	2	0	2	2	0	0		
9	0	0	0	-2	0	0	2	0	0	
10	2	-2	2	0	2	2	0	2	0	0

Results

Step (t)	*Pattern Name*	*Output of Units 1-10* 1	2	3	4	5	6	7	8	9	10	E	*Units updated*
1-3	T^1	-1	1	-1	-1	1	-1	-1	-1	-1	1	0	3,6,5
4-6	X^1	-1	1	-1	-1	-1	-1	-1	-1	-1	1	-16	9,1,4
7-17	X^2	-1	1	-1	1	-1	-1	-1	-1	-1	1	-24	6,6,6,2,2,8,8,1,6,5,10
18	P^2	-1	1	-1	1	-1	-1	-1	-1	-1	-1	-48	

Figure 2.3 A Hopfield network with N=10 units. Patterns P^1 and P^2 are stored. Pattern T^1 is presented and the network converges to P^2. The energy decreases when a unit changes state.

$$\begin{aligned}\Delta E(3) &= -\Delta O_5(3) h_5(3) \\ &= -(-2).(-8) \\ &= -16\end{aligned} \qquad (12)$$

This is the difference in the energy between steps 3 & 4 in Figure 2.3. After 17 steps the network settles into one of the stored patterns, in this case pattern, P^2. P^2 is a stable pattern (i.e. no units will change state when updated).

This example shows how patterns can be stored in the Hopfield network. It also shows how each time a unit's output changes the energy decreases, until a stable pattern is reached.

2.3 The capacity of the Hopfield network

The capacity of the Hopfield network in its original content addressable form is the number of patterns that can be stored as stable patterns at the same time. Hopfield (1982) estimated the capacity of the network to be approximately 0.15N, where N is the number of units in the network. This estimation was obtained experimentally.

If a pattern is stored correctly on a Hopfield network then it should be stable, i.e. if the network is in the state corresponding to the pattern then none of the outputs should change. If N_p random patterns are stored on a network then when the network is in a state corresponding to one of those patterns the probability that an output will change is:[1]

$$P_{Error}=\phi\left(-\alpha^{-\frac{1}{2}}\right) \tag{13}$$

where ϕ is the standard cumulative normal distribution and α is the number of patterns stored as a proportion of N, $\alpha=N_p/N$.

An experiment was performed to confirm the validity of (13). In this experiment a content addressable Hopfield memory was set up with N=100 inputs. Fifty random[2] patterns were chosen. The patterns were stored in the memory in groups of five. After each group of five patterns had been stored, the memory was tested using all the stored patterns. Testing with a pattern involved setting the memory into the state

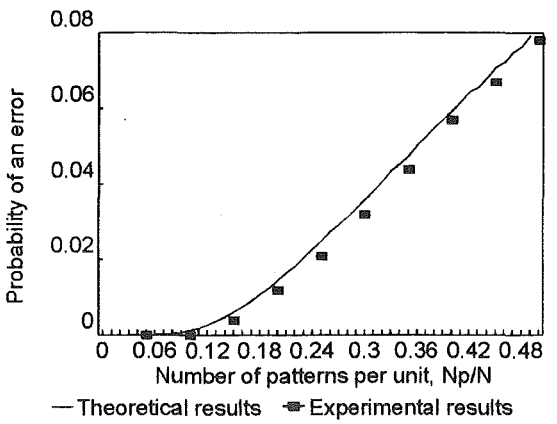


Figure 2.4 The theoretical and experimental error rates for the Hopfield memory.

[1] the derivation of (13) is given in appendix A.1.

[2] the random number generator used to generate the random patterns is a standard random number generator known as the additive congruential generator [Rubinstein, 1981; Knuth, 1969].

corresponding to the pattern and then checking to see if any output units changed state. Each output unit that did change state was recorded as one error and the total number of errors was recorded. The experiment was performed 100 times with different sets of random patterns. The results from the 100 experiments were averaged and are shown in Figure 2.4. The experimental error rates are very close to those predicted by (13)[3].

Gardner (1988) has performed a thorough analysis of the capacity of the Hopfield memory. For a pattern to be stable it must satisfy (4) for all i. Gardner used the stronger condition that for a pattern to be stored it must satisfy the following:

$$P_i^l h_i^l > \kappa \sqrt{\sum_{j=1}^{N} w_{ij}^2} \qquad (14)$$

for all i, where κ is a non-negative constant. If $\kappa=0$ then (14) is the same as (4). Larger values of κ should imply larger *basins of attraction*. The basin of attraction for a particular pattern, P^l, is the set of starting patterns for which the network will converge towards P^l. A pattern, P^k, which is similar to P^l, will have a small number of elements which are different. This means that, for any particular unit (say unit i), the local field from P^k, h_i^k, will be slightly different than it would be for P^l, h_i^l. If this difference between the size of the local fields of P^l and P^k is less than $\kappa \sqrt{\sum_{j=1}^{N} w_{ij}^2}$ then the unit's output O_i will be the same as P_i^l. This is demonstrated below.

For unit i to have an output of P_i^l when the state of the network corresponds to P_i^k (15) must be true:

$$P_i^l h_i^k > 0 \qquad (15)$$

By adding and subtracting the term $P_i^l h_i^l$ this can be rewritten as:

---

[3]The small difference can be put down to the assumptions made in the derivation given in appendix A.1.

$$P_i^l h_i^l + P_i^l (h_i^k - h_i^l) > 0 \tag{16}$$

If (14) is true then (16) becomes:

$$P_i^l (h_i^l - h_i^k) < \kappa \sqrt{\sum_{j=1}^{N} w_{ij}^2} \tag{17}$$

If all of the units satisfy (15) then the network will converge to P^k. Larger values of κ will mean more patterns satisfy (17) and hence (15) and so by definition this means larger values of κ will result in a larger basin of attraction.

Gardner demonstrated that for uncorrelated (random) patterns and $\kappa=0$ the capacity is 2N. The capacity decreases as κ increases and the capacity increases if the patterns are correlated (the capacity tends to infinity as the patterns tend towards being completely correlated). Only sets of uncorrelated patterns are considered in this thesis. Gardner demonstrated this capacity by showing that solutions exist for the inequality given in (14). Furthermore Gardner gave an iterative learning algorithm that would converge to the solution. The algorithm is derived from the generalization of the perceptron convergence theorem (Rosenblatt 1962, Minsky and Papert, 1969). The algorithm defines an array of errors for each pattern P^l:

$$E_i^l = \kappa \sqrt{\sum_{j=1}^{N} w_{ij}^2} - P_i^l \sum_{j=1}^{N} w_{ij} P_j^l \tag{18}$$

If this error is positive then (14) will not be true. The algorithm attempts to make all these errors non-positive. Note that if $\kappa=0$ then making the error non-positive is the same as making (4) true. For the iterative algorithm, the weights to a unit are updated only if the error for that unit is positive. Unlike (6), which sets the weight to the product of P_i^l and P_j^l, this algorithm uses this product to give the direction of the weight change, scaled by the error, E_i^l, and the learning rate, α. Hence the weights for unit i are updated according to the following expression:

$$\Delta w_{ij}=\begin{cases} \alpha E_i^l P_i^l P_j^l & \text{if } E_i^l>0 \\ 0 & \text{if } E_i^l \le 0 \end{cases} \tag{19}$$

The weights must be updated over all the patterns, and iterated until $E_i^l \le 0$ for all the units, i and all the patterns, *l*. This algorithm can also be used in the CMAC and SDM memories described in chapters 3 and 4. Experimental validation of the ability of the iterative algorithm to realise the full capacity of 2N has not been performed here, but an equivalent experiment was performed using the SDM. That experiment is described in section 3.4 of the next chapter.

2.4 Alternative views of the Hopfield network

An alternative way of viewing the Hopfield network is as a two-layer network rather than a fully interconnected group of units. The Hopfield network is described here as a two-layer network because in this format it has a similar architecture and uses similar terminology to the SDM and CMAC networks described in the next chapters. The two layers in the Hopfield network consist of a layer of input units (layer 0) and a layer of output units (layer 1), with the outputs connected to the inputs (as shown in Figure 2.5).

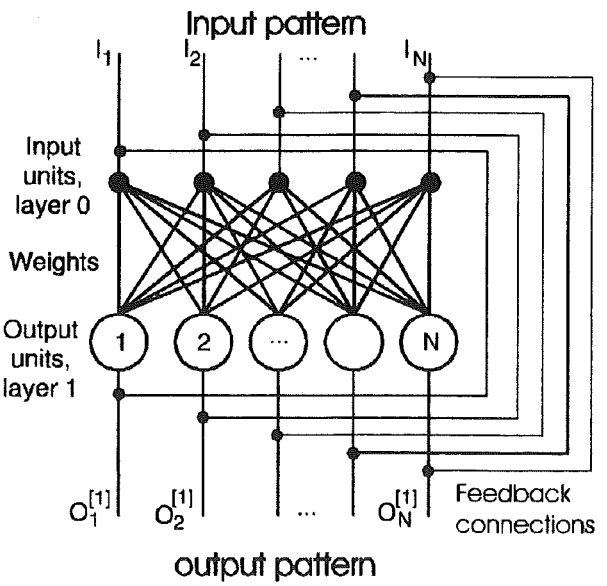


Figure 2.5 The Hopfield network can be viewed as a two-layered network.

When the network is in input mode the state of input units is determined by the inputs:

$$O_i^{[1]}(t)=I_i(t) \tag{20}$$

where $O_i^{[k]}(t)$ is the output of unit i in layer k at time step t. The input layer is layer k=0. When the network is in settling mode the state of the input units is determined by the state of the output units:

$$O_i^{[0]}(t)=O_i^{[1]}(t) \tag{21}$$

When the network is in settling mode a randomly chosen output unit is updated in each time step according to the following:

$$O_i^{[1]}(t)=\begin{cases} 1 & \text{if } h_i^{[1]}(t)>T \\ -1 & \text{if } h_i^{[1]}(t)\leq T \end{cases} \tag{22}$$

The local field, $h_i^{[1]}$, of output unit i is:

$$h_i^{[1]}(t+1)=\sum_{j=1}^{N} O_j^{[0]}(t)w_{ij} \tag{23}$$

where w_{ij} is the weight from unit j on layer 0 to unit i on layer 1.

When viewing the Hopfield network as a two-layer network, the connections with weights are in one direction only; from the input units to the output units. However, the weights of the connections can be assigned in exactly the same way as they are in the original Hopfield memory, (7).

The reason for presenting the Hopfield network in this way is because then the terminology and the architecture are similar to that of the SDM and CMAC networks described in the following chapters. The SDM and CMAC have units arranged in layers like those in the alternative view of the Hopfield network given above. The major difference is that the CMAC and SDM have an extra layer of units between the input layer and the output layer, and they have no feedback connections.

2.4.1 The Hopfield network as an associative memory

Up to this point the Hopfield network has been described as a content addressable memory; however it can be configured as an associative memory. An associative memory associates input patterns, X^l, with output patterns, Y^l. The Hopfield network can be configured as an associative memory by removing the feedback connections from the output to the input in the two-layer model of the Hopfield network (described above). Some of the equations which describe how the network operates also change.

The state of the input units is always determined by the input patterns:

$$O_i^{[0]}(t)=I_i(t) \tag{24}$$

The state of the output units is given by the same equations as before, (22) and (23). However, all the output units are updated in a single time step rather than just one randomly chosen unit in each time step.

The weights can be assigned in a similar way to those in the original Hopfield network, (7):

$$w_{ij}^{[1]}=\sum_{l=1}^{N_p} X_j^l Y_i^l \tag{25}$$

where Y_i^l is the ith element of output pattern Y^l. Note that the stipulation that diagonal elements be 0 has been removed and the weights are no longer symmetrical.

Gardner's algorithm, which was described in the previous section, can be applied to the Hopfield memory when it is configured as an associative memory. The errors are defined as:

$$E_i^l=\kappa\sqrt{\sum_{j=1}^{N} w_{ij}^2}-Y_i^l\sum_{j=1}^{N} w_{ij}X_j^l \tag{26}$$

and the weights are updated using the following:

$$\Delta w_{ij}=\begin{cases} \alpha E_i^l Y_i^l X_j^l & \text{if } E_i^l>0 \\ 0 & \text{if } E_i^l\le 0 \end{cases} \tag{27}$$

The Hopfield network as an associative memory can be used to store sequences of patterns, P^1, P^2, P^3, ... To do this each pattern in the sequence is associated with the next pattern in the sequence, i.e the Hopfield network associates $X^l=P^l$ with $Y^l=P^{l+1}$. This is explained further in chapter 5.

2.5 Summary

The Hopfield network was originally described (Hopfield, 1982) as a content addressable. memory consisting of N fully connected units. This memory is guaranteed to settle into a stable state when the weights are assigned using Hopfield's algorithm (7).

The number of patterns that can be successfully stored in the content addressable Hopfield network is approximately 0.15N when using the original algorithm for assigning weights, (7). Gardner (1988) has demonstrated that using an iterative algorithm for assigning weights the number of patterns that can be successfully stored is 2N.

The Hopfield network can be viewed as a two-layer network with connections between the two layers, and feedback connections from the output to the input. Without the feedback connections the Hopfield network becomes an associative memory.

CHAPTER 3

Sparse Distributed Memory (SDM)

3.1 Introduction

The sparse distributed memory (SDM) is an associative memory which was proposed by Pentti Kanerva (1988)1. The SDM is very similar to the CMAC described in the following chapter. This chapter introduces the SDM in section 3.2, which shows how the SDM can associate binary patterns. Section 3.5 explains how patterns of real valued numbers can be associated. An example of a small memory is given in section 3.2.2.

Section 3.3 compares the actual performance of the SDM for storing associations against the theoretical performance outlined by Kanerva (1988) and finds a discrepancy. A corrected theory is proposed which fits the actual performance better. An alternative method of reading from the SDM is also proposed which improves the performance. The results outlined in section 3.3 have been published in Ryan & Andreae (1995).

Section 3.4 outlines how Gardner's analysis of the capacity of the Hopfield network can be applied to the SDM. Using an iterative method of storing data the capacity is found to be twice the number of location units. This is confirmed experimentally. Section 3.6 lists the major differences between the SDM and Hopfield memory.

Chapter 5 describes how the SDM can be used to store sequences.

3.2 The Sparse Distributed Memory (SDM)

The SDM is an associative memory. When an input pattern, I, is presented to the memory, it produces an output pattern, O, as described below. The input pattern consists of N elements and the output pattern consists of M elements. The elements of the input and output patterns can be binary or real. Only binary elements (I_i, O_j = ±1, i=1..N, j=1..M) will be considered

in this section. Section 3.5 explains how real valued elements are handled.

As an associative memory the SDM is able to associate input patterns, X^l, with output patterns Y^l. If the associations are stored correctly then, when the input is $I=X^l$, the output will be $O=Y^l$. This is analogous to a computer memory, which is able to store data (groups of numbers) at addresses. Given a particular address the computer memory produces the stored data. The input pattern of the SDM is analogous to the address of the computer memory and the output patterns are analogous to the data.

One significant difference between the SDM and a conventional computer memory is that the SDM has a built-in ability to generalize. This means that when an association, (X^l, Y^l), is stored then for inputs which are similar enough to X^l (i.e. inputs in which not more than a small percentage of the elements are different) the SDM will produce outputs which are the same as Y^l. This ability to generalize would be very undesirable in a conventional computer memory, but there are a number of applications where this ability would be useful.

For example if an SDM is used to recognise spoken words then the input patterns for a particular word said by two different people will not be exactly the same, but they should be similar. The SDM's ability to generalize means that it should be able to recognise these two words as the same without having been trained with their exact inputs. An example of the SDM being used to recognise speech is given in Chapter 5 (section 5.4.1).

The SDM can also be configured as a content addressable memory (CAM). CAMs are described in the previous chapter in section 2.1.1. To configure the SDM as a CAM the number of elements in the input and output patterns should be the same, and the outputs should be connected to the inputs. Storing a pattern, P^l, on the SDM when it is configured as a CAM involves storing association (X^l, Y^l) with $X^l=Y^l=P^l$. The SDM as a CAM is a special case of the SDM as an associative memory and so the rest of this chapter will consider the SDM as an associative memory only.

3.2.1 The structure of the SDM

The SDM consists of three layers of units (see Figure 3.1):

- a layer of N input units, one for each element of the input pattern, layer 0,
- a layer of s location units (which Kanerva termed hard locations), layer 1, and
- a layer of M output units, one for each element of the output pattern, layer 2.

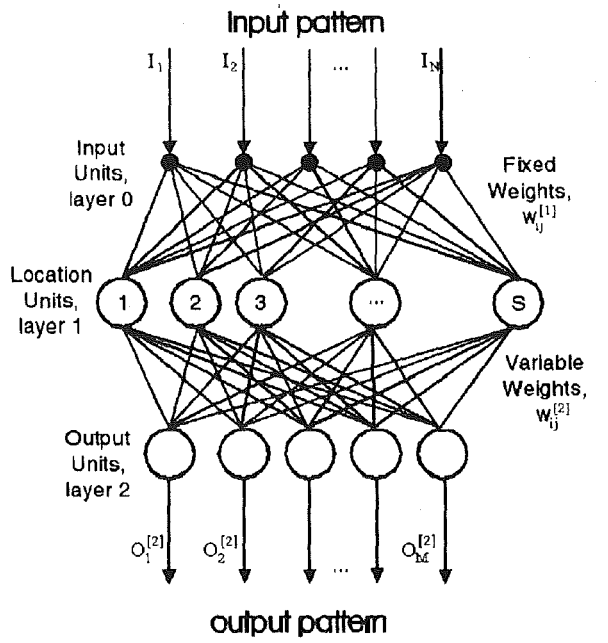


Figure 3.1 A diagram of Kanerva's sparse distributed memory (SDM).

There are two sets of connections in the SDM:

- one set from the input units to the location units, which have fixed weights, and
- one set from the location units to the output units, which have weights which change according to the associations stored in the memory.

The input units each receive one element from the input pattern, and they output the same value:

$$O_i^{[0]}(t) = I_i(t) \tag{28}$$

where $O_i^{[k]}(t)$ is the output of unit i on layer k at time t. The input units are layer k=0.

The fixed weights of the connections from the input units to the location units, are chosen when the SDM is set up. The weights have a value of either -1 or 1 and are chosen *randomly* at set-up. Each location unit has N weighted connections going to it (one from each input unit), the values of which form a point in N-bit input space. These N binary values of the fixed weights of the connections going to a location unit are also referred to as the *address* of the location unit.

The location units in the SDM are analogous to the physical locations in a conventional computer memory and the input and output patterns are analogous to an address and data respectively. A conventional computer memory has a physical location, where the data is stored, for every possible address, i.e. there are 2^N addresses and so 2^N locations. An SDM however, has only s location units, $s<2^N$, and the addresses of these (the fixed weights) are chosen randomly from the set of possible addresses.

In the example given in section 3.2.2 the SDM has N=4 inputs and so there are $2^4 = 16$ possible input patterns. There are 8 location units and so the fixed weights of the connections going to the location units represent 8 of these possible patterns. For example the fixed weights of the connections going to location unit 1 (an arbitrary label) are: {-1 1 -1 1}.

In a conventional memory each address (or input) points to a unique location. This is not possible in the SDM because there is not a unique location unit for every possible input. Therefore in the SDM each input effectively points to the set of location units having addresses "close" to the input.

In the SDM an input pattern is "close" to an address of a location unit if the Hamming distance between the two patterns is less than or equal to a number, r, where r is a parameter of the memory, termed the *access radius*. The Hamming distance between two patterns is the number of elements which are different. For example the Hamming distance between the patterns {-1 1 1 1} and {-1 1 -1 1} is one because there is one element which is different (the third element).

The output of the location units indicates which location units have addresses which are close to the current input. Location units have two possible outputs: 1 (active, address is close to the input) and 0 (not active). The output of a location unit is called its activation. The Hamming distance between two patterns can be calculated by finding half the difference between the number of elements and the sum of the product of the elements of the two patterns. Thus the activation of location unit i at time (t+1) is:

$$O_i^{[1]}(t+1)=\begin{cases}1 \text{ if } \frac{1}{2}\left(N-\sum_{j=1}^{N} O_j^{[0]}(t)w_{ij}^{[1]}\right)\le r \\ 0 \text{ if } \frac{1}{2}\left(N-\sum_{j=1}^{N} O_j^{[0]}(t)w_{ij}^{[1]}\right)>r\end{cases} \quad (29)$$

where $O_j^{[0]}(t)$ is the signal from input unit j (layer 0) at time t and $w_{ij}^{[1]}$ is the weight of the connection from input unit j to location unit i; $O_j^{[0]}$, $w_{ij}^{[1]}=\pm 1$.

Therefore, rather than pointing to one location, like a computer memory, an input to the SDM points to a set of location units which have addresses close to the input. The location units which have addresses close to the current input, have an activation of one (the others have an activation of zero). In a computer memory data is stored and retrieved from the single location pointed to by the address. In the SDM the data is stored in, and retrieved from, the weights of the connections from the active location units to the outputs. The weights have signed integer values.

Data is retrieved from the variable weights by the output units. An output unit will output one if the sum of the connection weights from the active location units is positive, otherwise it will output negative one. The output of unit i in the output layer at time t is:

$$O_i^{[2]}(t)=\begin{cases}1 & \text{if } h_i^{[2]}(t)>T \\ -1 & \text{if } h_i^{[2]}(t)\le T\end{cases} \quad (30)$$

where the local field, $h_i^{[2]}(t)$, is the sum of the weights of the connections from the active units at time t:

$$h_i^{[2]}(t+1)=\sum_j w_{ij}^{[2]}O_j^{[1]}(t) \quad (31)$$

where $w_{ij}^{[2]}$ is the weight of the connection from location unit j to output unit i, and $O_j^{[1]}(t)$ is the activation (1 or 0) of the location unit j at time t as given by (29).

When an SDM is set up all the variable weights are set to zero. Data is stored in the SDM by adjusting the variable weights of the connections from the active location units to the output units. If a one is to be stored at an output unit then the weights from the active location units to that output are incremented (by one). If a negative one is to be stored then the weights are decremented (by one). In the example in section 3.2.2 the input pattern X^1={-1 1 1 1} activated location units 1, 3, and 8. The output pattern (data) stored at X^1 is Y^1= {-1 1 1}. When this pattern is stored the weights of the connections from location units 1, 3 and 8 to output unit 1 ($w_{11}^{[2]}$, $w_{13}^{[2]}$ and $w_{18}^{[2]}$) are decremented and the weights of the connections to output units 2 and 3 ($w_{21}^{[2]}$, $w_{23}^{[2]}$, $w_{28}^{[2]}$, $w_{31}^{[2]}$, $w_{33}^{[2]}$ and $w_{38}^{[2]}$) are incremented.

An (input pattern, output pattern) pair or (address, data) pair is referred to as an association. Using the algorithm described above to store an association, (X^l, Y^l), the weight of the connection to each output unit will change by

$$\Delta w_{ij}^{l} = O_j^{[1]l} Y_i^{l} \tag{32}$$

where $O_j^{[1]l}$ is the activation of location unit j when the input is X^l, and Y_i^l is the ith element of output pattern Y^l.

If there are N_p associations to be stored in an SDM, {(X^1, Y^1) ... (X^{Np}, Y^{Np})}, then the weights will be given by summing (32) over N_p:

$$w_{ij}^{[2]} = \sum_{l=1}^{N_P} O_j^{[1]l} Y_i^{l} \tag{33}$$

The expression, (33), giving the weights between the location units and the output units is very similar to (25) that gives the weights for the Hopfield network as an associative memory. The only difference is that rather than summing the products of the input and the output elements, the SDM sums the products of the activations of the location units, $O_j^{[1]l}$, and the output elements, Y_i^l. Section 3.6 compares the SDM and the Hopfield network.

The name "sparse distributed memory" was given by Kanerva. Kanerva compared the SDM

to a computer memory which has a physical location for every address and for which data stored at an address is stored in only one location. The sparse distributed memory is termed "sparse" because the number of location units (or addresses), s, is very small (sparse) compared to the number of possible location units, 2^N. The memory is termed "distributed" because the storage of any association is distributed over the weights from the set of active location units.

The following section outlines a simple example of an SDM.

3.2.2 A simple example of the SDM.

This example shows how associations are stored. For each input this involves calculating which location units will become active, and adjusting the weights from the active location units to the output units. This example also shows how associations are recalled and how the SDM has the ability to generalize.

The example is presented in four tables which show:

- the fixed weights giving the location units' addresses and the Hamming distance between the location units' addresses and the inputs used (Table 1);
- the location units activated by each of the five inputs used (Table 2);
- the value of the variable weights after the associations have been stored (Table 3), and
- the output of the memory over a range of inputs (Table 4).

For this example a very small network is used with just N=4 binary inputs M=3 binary outputs and s=8 location units. The memory is to associate the input-output patterns:

(X^1, Y^1) = ({-1 1 1 1}, {-1 1 1})

(X^2, Y^2) = ({-1 1 -1 -1}, {-1 -1 -1})

As well as the input patterns above, three other input patterns will be used to test the memory,

$X^3 = \{-1\ 1\ 1\ -1\}$, $X^4 = \{1\ 1\ 1\ 1\}$, and $X^5 = \{-1\ -1\ -1\ -1\}$.

The randomly chosen fixed weights of the connections from the input units to the location units are shown in Table 1 along with the Hamming distance between the input patterns used, $X^1..X^5$, and the addresses of each location unit. For example the Hamming distance between the weights of the connections going to the first location unit, {-1 1 -1 1} and X^1, {-1 1 1 1}, is one. There is one element that is different between these patterns, $w_{13}^{[1]} \neq X^1_3$. These Hamming distances are used to determine the active location units for each input.

Table 1 The fixed weights and the Hamming distances between the input patterns and the location unit addresses.

Location unit number	Fixed weights (location unit addresses)	Hamming distance between location unit address and the 5 inputs				
	j 1 2 3 4	X^1 -1 1 1 1	X^2 -1 1 -1 -1	X^3 -1 1 1 -1	X^4 1 1 1 1	X^5 -1 -1 -1 -1
1	-1 1 -1 1	1	1	2	2	2
2	1 1 -1 -1	3	1	2	2	2
3	-1 1 1 1	0	2	1	1	3
4	-1 -1 -1 1	2	2	3	3	1
5	-1 -1 1 -1	2	2	1	3	1
6	1 1 1 -1	2	2	1	1	3
7	-1 -1 -1 -1	3	1	2	4	0
8	-1 1 1 -1	1	1	0	2	2

With an access radius of r=1, location units whose addresses are less than or equal to a Hamming distance of 1 away from the input pattern will have an activation of 1. Table 2 shows the location units activated by the input patterns, $X^1..X^5$. For example X^1 activates location units 1, 3 and 8. Table 1 shows that the Hamming distance between X^1 and the addresses of those location units is less than 2. The other five units (2, 4, 5, 6, 7) will have an activation of 0.

One point to note about the sets of active location units in Table 2 is that inputs which are close to each other (in terms of Hamming distance) have overlapping sets of active location units. For example inputs X^1 and X^4 are close (there is a Hamming distance of 1 between them) and they both activate location unit 3. However X^5 is not very close to X^1 (a Hamming distance of 3 away) and none of its active location units are the same as those for X^1.

Table 2 The location units activated by each input.

Input pattern	Location units activated
X^1	1 3 8
X^2	1 2 7 8
X^3	3 5 6 8
X^4	3 6
X^5	4 5 7

Table 3 shows the value of the weights from the location units to the output units after the associations have been stored, using (33). For example, the weight from location unit 1 to output 1, $w_{11}^{[2]} = -1$. This value is calculated as follows: when the input is X^1, the activation of location unit 1 is $O_1^{[1]} = 1$, and $O_1^{[2]}=0$ when the input is X^2. The outputs are $Y^1_1 = -1$ and $Y^2_1 = 1$. (33) becomes:

$$\begin{aligned} w_{11}^{[2]} &= O_1^{[1]1} Y_1^1 + O_1^{[1]2} Y_1^2 \\ &= (1).(-1)+(0).(1) \\ &= -1 \end{aligned} \qquad (34)$$

Table 3 The variable weights

Location unit no. j	Variable weights, $w_{ij}^{[2]}$ *Output unit number, i*		
	1	2	3
1	-1	1	1
2	-1	-1	-1
3	-1	1	1
4	0	0	0
5	0	0	0
6	0	0	0
7	-1	-1	-1
8	-2	0	0

Finally, Table 4 shows the local fields and the outputs of the SDM for each of the five inputs. The local fields are calculated using (3). For example:

$$\begin{aligned} h_1^{[2]1} &= \sum_{j=1}^{N=8} O_j^{[1]1} w_{1j}^{[2]} \\ &= (1).(-1)+(0)(-1)+(1).(-1)+(0).(0) \\ &\quad +(0).(0)+(0).(0)+(0).(-1)+(1).(-2) \\ &= -4 \end{aligned} \qquad (35)$$

Table 4 shows that the stored patterns are stored correctly i.e. the memory is successfully associating Y^1 with X^1 and Y^2 with X^2. The other input patterns were used to demonstrate the ability of the memory to generalize. X^4 is closer to X^1 than X^2 and its output is the same as that of X^1. X^5 is closer to X^2 and its output is the same as X^2. X^3 is similar to both X^1 and X^2, its output being that which was associated with X^2.

Table 4 The response of the SDM to each input. The output is shown for each of the five inputs. Also shown are the local fields at the 3 output units, which are used to calculate the outputs.

Input	Local fields			Output
X^l	$h_1^{[2]l}$	$h_2^{[2]l}$	$h_3^{[2]l}$	$O^{[2]l}$
X^1	-4	2	2	-1 1 1
X^2	-9	-1	-1	-1 -1 -1
X^3	-8	0	0	-1 -1 -1
X^4	-7	1	1	-1 1 1
X^5	-6	0	0	-1 -1 -1

3.3 The performance of the SDM

Kanerva (1988) defined the term *fidelity* as a measure of performance of the SDM. Fidelity is the probability that, when an association, (X, Y) is stored in memory, the output pattern, Y, can be recalled when the input is I. The closer the input I is to X, the higher the fidelity. The maximum fidelity occurs when the input I is the same as the input pattern, X, of the association. Kanerva derived an expression for the fidelity of the SDM as a function of:

- the number of associations stored, N_p,
- the number of inputs, N,
- the number of location units, s, and
- the access radius, r.

A major part of Kanerva's derivation for the expression for fidelity involves deriving an

expression for the probability that an element of an output pattern of a stored association is stored incorrectly. This expression gives the theoretical error rates for the stored output elements as a function of the number of patterns stored. As is outlined below, these error rates are easy to verify experimentally. The results of my experiments revealed that there was a significant difference between the experimental and theoretical error rates (Ryan & Andreae, 1995). An analysis of Kanerva's derivation leads to a new expression for the error rate which closely matches the experimental results. Section 3.3.1 describes an alternative method of reading from memory which improves the error rates.

Kanerva's expression for the probability of an element in an output pattern in a stored association being stored incorrectly is:

$$P_{Error} = \Phi\left(\frac{s}{\sqrt{N_p s\left(1+\frac{h_p^2 s}{4p^2 N}\right)}}\right) \qquad (36)$$

where Φ is the standard normal cumulative distribution function, p is the probability that a location unit will be active for a given input (a function of r and N), N_p is the number of associations stored in the SDM and h_p is[4]:

$$h_p = -\frac{1}{\pi} e^{-(\Phi^{-1}(p))^2} \qquad (37)$$

An experiment was performed to confirm the validity of (36). The experiment was designed to obtain error rates for output elements in stored associations (described below) so that the actual error rates could be compared to those predicted by (36). In the experiment a SDM was simulated on a computer. The SDM had N=48 inputs, s=500 location units, M=8 outputs and

[4]see Appendix A.2 for a description of the meaning of h_p.

an access radius of r=17. The experiment consisted of 100 trials. For each trial 500 random[5] associations (pairs of randomly selected inputs and outputs[6]) were stored using (33). The associations were stored in groups of 50 and after each group of 50 all the stored associations were checked. The percentage of elements in the output patterns which were not recalled correctly was calculated and averaged over 100 trials. The fixed weights and the associations were randomized for each trial.

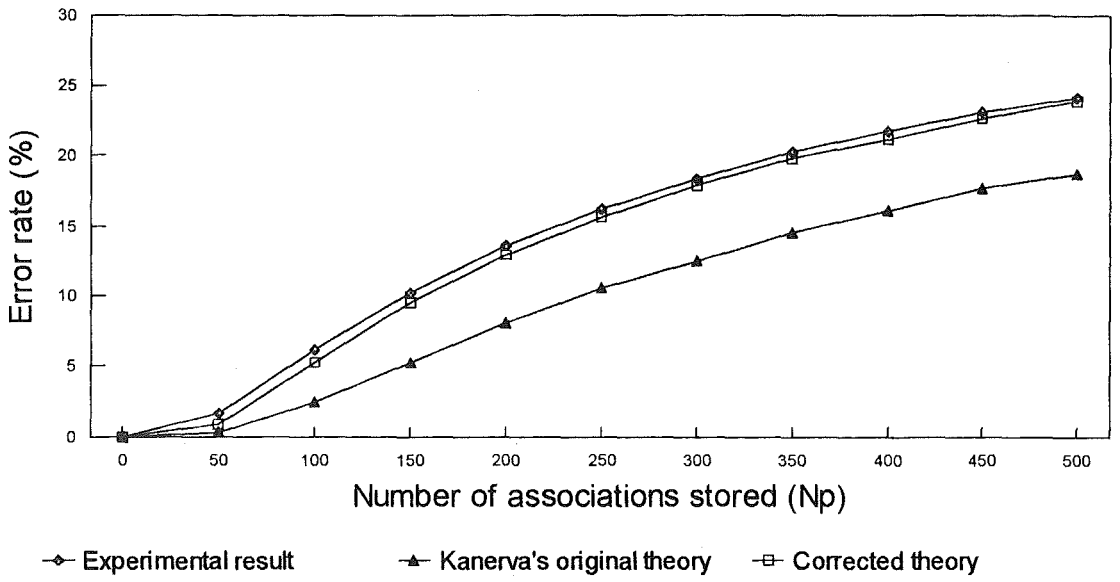


Figure 3.2 There is a discrepancy between the error rate predicted by Kanerva's theory and the experimental results. A corrected theory is closer to the experimental results.

The experimental results and those predicted by (36) are shown in Figure 3.2. This graph shows there is a significant difference between the theoretical and actual results with the actual results giving a higher error rate. Inspection of the derivation of (36) shows that the variance of the output data was not taken into account (it was assumed to be 0). When this

[5]the random number generator used to generate the random associations and weights is a standard random number generator known as the additive congruential generator [Rubinstein, 1981; Knuth, 1969].

[6]A check was made when the associations were created that none of the inputs were the same. This check was done because in the unlikely even that two or more inputs were the same the set of associations would be impossible to store under any circumstances.

is taken into account the probability of an error becomes[7]:

$$P_{Error} = \Phi\left(\frac{s}{\sqrt{N_p s\left(1+\frac{h_p^2 s}{4p^2 N}+p^2 s\right)}}\right) \tag{38}$$

Figure 3.2 also shows the error rate predicted by (38). This prediction is close to the experimental results[8].

3.3.1 An alternative method of reading from the SDM

A significant improvement can be made to the SDM by changing the way the output units produce their values. For this alternative method the local field is the *average* of the weights (rather than the sum of the weights as in (31)) from the active location units:

$$h_i^{[2]}(t+1) = \frac{\sum_{j=1}^{s} O_j^{[1]}(t) w_{ij}^{[2]}}{\sum_{j=1}^{s} O_j^{[1]}(t)} \tag{39}$$

Whereas in the original SDM the output units have a threshold of T=0 (see equation (30)), for this method each output unit has its own threshold equal to the average weight of the connections from *all* of the location units to that unit. Thus the output is given by:

[7]see Appendix A.2 for a derivation

[8]A explanation for the small difference between the corrected theory and the experimental results is given in the appendix 3.3 with the derivation of the corrected theory.

$$O_i^{[2]}(t)=\begin{cases} 1 & \text{if } h_i^{[2]}(t)>T_i \\ -1 & \text{if } h_i^{[2]}(t)\le T_i, \end{cases} \tag{40}$$

where the threshold, T_i, for output i is the average of the weights from all s of the location units to output i:

$$T_i=\frac{\sum_{j=1}^{s} w_{ij}^{[2]}}{s} \tag{41}$$

This means that the output units are now comparing the average weight from the active location units to the average weight from all the location units. Previously the output units were comparing the sum of the weights from the active units to zero, (30). Using this new method of reading from memory the probability of an error is[9]:

$$P_{Error}=\Phi\left(\frac{s}{\sqrt{N_p s\left(1+\frac{h_p^2 s}{4p^2 N}+p-p^2\right)}}\right) \tag{42}$$

This expression is very close to that derived by Kanerva, (36) because the terms p and p^2 are small compared to 1. p is the probability that a location unit will be active. For the SDM used in these experiments approximately a tenth of the location units would be active for any input, i.e. $p\approx 0.1$. For the memory Kanerva (1988) often used for an example only a thousandth of the location units would be activated, $p\approx 0.001$. Another experiment was performed, this time to confirm (42). The experiment was the same as above (in section 3.3) but used the alternative method of reading from the SDM. The results of this experiment are shown in Figure 3.3.

Figure 3.3 also shows that the alternative method of reading from the SDM (equation (40))

[9] see Appendix A.3 for a derivation

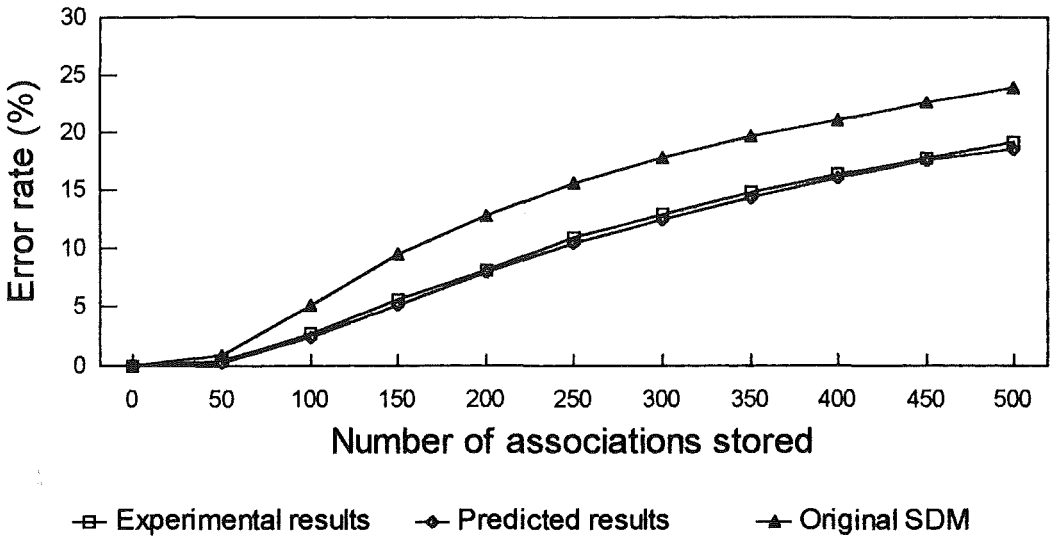


Figure 3.3 Experimental results using the alternative method of reading are close to the theoretical results, (42). The error rate is significantly better than that for original SDM.

is significantly better (the error rate is lower) than the original, (equation (30)). In these experiments the patterns were chosen so that on average there was the same number of 1's and -1's in the patterns. A further experiment was performed to demonstrate the superiority of the alternative method of reading from SDM. In this experiment the patterns were chosen so that 75% of the elements of the output patterns were 1's. The experiment was performed using both the alternative method of reading and the original method. The results are shown in Figure 3.4.

Figure 3.4 shows that the original method of reading from memory does not work well at all when the probability of a 1 in the output patterns varies from 0.5. The reason for this is that when there are different numbers of 1's and -1's the average weight (from the location units to the output units) becomes non-zero. The average weight is a measure of the noise from the other stored patterns. In this case the average weight will be positive and so as more associations are stored, more outputs which were originally stored as -1 will be read incorrectly (using the original method of reading from memory) as 1. Eventually (as more associations are stored) the average weight will become so positive that every bit will be read as a 1. The alternative method of reading from memory, presented here, is not significantly affected by having data with a non-zero mean. This is because, although the average weight

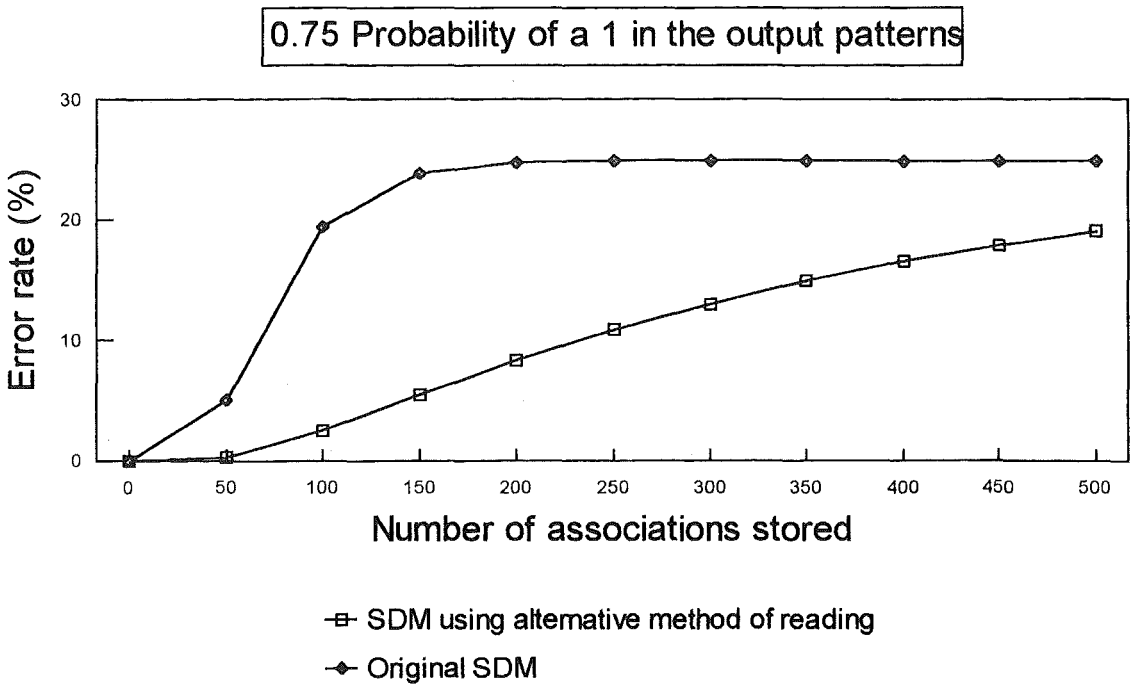


Figure 3.4 When there are unequal numbers of 1s and -1s, the alternative method of reading from the SDM is much better than the original method.

is still positive, the weights are less positive from location units where a -1 has been stored.

The experimental and theoretical work outlined above shows that, when (33) is used to assign the variable weights, the number of associations the SDM can store before the error rates start to become significant is approximately one tenth of the number of location units, 0.1s. The capacity of the SDM when (33) is used has also been analyzed by Chou (1989) and Keeler (1988). The capacity can be increased if an iterative algorithm is used to assign the weights. This is outlined in the following section.

3.4 Assigning the variable weights iteratively

The SDM, like the Hopfield network, can store a lot more associations if the associations are stored by assigning the weights using an iterative training algorithm rather than using (33). In (33), for each association, the weights from the active location units to each output unit are either incremented or decremented by a fixed amount, depending on whether the output is to be a 1 or -1. Prager & Fallside (1989) suggested an iterative algorithm which differs from

(33) in three major ways:

- The set of associations to be stored is presented to the SDM repeatedly until all of the associations are stored correctly.

- When an association (X^l, Y^l) is presented to the SDM only the weights to the output units which are in the wrong state are adjusted. An output unit (unit i) is in the wrong state if the output using (30), $O_i^{[2]l}$ is not the same as the desired output, Y_i^l. i.e. the weights from the active location units to output unit i are adjusted if:

$$O_i^{[2]l} \neq Y_i^l \tag{43}$$

- The weights which are adjusted when (43) is true, are not adjusted by a fixed amount, but are adjusted by just enough to change the polarity of the output. The polarity of the output depends on the sign of the local field, $h_i^{[2]l}$ (31). The sign of the local field can be changed by adjusting each of the weights from the active location units by $-\beta \frac{h_i^{[2]l}}{N_A^l}$ where N^l_A is the number of active location units when the input is X^l,

$$N_A^l = \sum_{j=1}^{s} O_j^{[1]l} \tag{44}$$

and β is a constant greater than one. Then when the association (X^l, Y^l) is being presented, the weight from location unit j to output unit i changes by

$$\Delta w_{ij}^{[2]l} = \begin{cases} -\beta O_i^{[1]l} \frac{h_i^{[2]}}{N_A^l} & \text{if } O_i^{[2]l} \neq Y_i^l \\ 0 & \text{if } O_i^{[2]l} = Y_i^l \end{cases} \tag{45}$$

This is very similar to the iterative training algorithm for the Hopfield network, (19), which was proposed by Gardner (1988). In fact, as shown below, Gardner's analysis of the capacity of the Hopfield network and the iterative algorithm can be applied to the SDM. For the original SDM[10] an association (X^l, Y^l) is stored correctly if for all of the output units, the local field, $h_i^{[2]l}$ has the same sign has the desired output, Y_i^l:

$$Y_i^l h_i^l > 0 \tag{46}$$

for all i=1..M (M is the number of output units). The stronger condition that Gardner (1988) used for storing associations in the Hopfield memory (see chapter 2) can also be used for the SDM. Namely, a pattern is stored correctly if:

$$Y_i^l h_i^l > \kappa \sqrt{\sum_{j=1}^{s} \left(w_{ij}^{[2]h}\right)^2} \tag{47}$$

for all i=1..M, where κ is a non-negative constant. Like the Hopfield memory larger values of κ should imply larger basins of attraction (see page 15 for a description of basins of attraction).

Storing a set of N_p associations in the SDM is equivalent to finding solutions to the inequalities, (46) (or (47) if larger basins of attraction are desired). For each output unit there are N_p inequalities in s variables (the weights from the s location units) to solve. The number of variables, s, restricts the number of inequalities for which solutions will exist and hence restricts the number of associations which can be stored. Gardner demonstrated that, when the patterns are uncorrelated and κ=0, the maximum number of inequalities for which a solution exists (i.e. the number of associations that can be stored) is twice the number of variables in the inequalities. For the Hopfield network there are N variable weights to each output unit and so the capacity (the number of associations which can be stored) is 2N. For the SDM there are s variable weights (one from each of the s location units) to each output unit and so the

[10]The method of reading has little effect when an iterative method is used to store the weights, so the Kanerva's original method of reading (which is simpler) is used here. The experiments detailed on page 39 were performed using both methods of reading and there was no significant difference in the results.

capacity is 2s.

Furthermore, the iterative algorithm Gardner used, (18) and (19), that converges to a solution of the inequalities, (14), for the Hopfield network can also be used in the SDM to converge to a solution to the inequalities in (47). This algorithm defines errors at every output unit for each pattern and gives an expression to adjust the weights iteratively to reduce these errors. For the SDM, the error at output unit i for input pattern X^l is:

$$E_i^l = \kappa \sqrt{\sum_{j=1}^{s} \left(w_{ij}^{[2]l}\right)^2} - Y_i^l h_i^l \qquad (48)$$

The algorithm attempts to make all these errors non-positive. The change for weight w_{ij} when (X^l, Y^l) is presented is:

$$\Delta w_{ij}^l = \begin{cases} \dfrac{\alpha O_j^{[1]l} Y_i^l E_i^l}{\sum_{k=1}^{s} O_k^{[1]l}} & \text{if } E_i^l > 0 \\ 0 & \text{if } E_i^l \le 0 \end{cases} \qquad (49)$$

where α is a positive constant, termed the learning rate. Gardner's algorithm is applied by iteratively changing the weights using (49) until all the errors are non-positive. The iterative algorithm proposed by Prager and Fallside, (45), is the same as Gardner's, (49), when $\alpha=\beta$ and $\kappa=0$.

An experiment was performed to see if the capacity, 2s, predicted by applying Gardner's theory is correct. In this experiment a memory was used that had N=48 inputs, and M=1 output. Five tests were performed with 100, 200, 300, 400 & 500 location units respectively. In these tests random associations were stored using Gardner's iterative algorithm, (49). For each test the number of associations stored, N_p was increased until no more associations could be stored. Each test was repeated 50 times and the number of associations that could be stored was recorded and averaged. For each test the fixed weights and associations were randomized.

The results are shown in Figure 3.5. As can be seen the experimental results give a capacity of almost exactly 2s, confirming Gardner's theory.

3.5 Associating real valued inputs and outputs

Section 3.2 described how binary inputs and outputs can be associated in the SDM. The SDM can use real inputs and outputs, as is described below in sections 3.5.1, and 3.5.2 respectively.

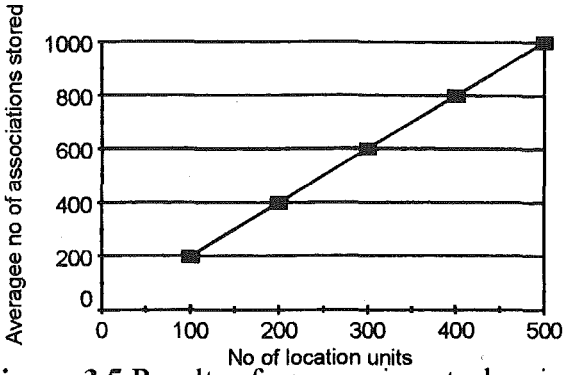


Figure 3.5 Results of an experiment, showing the number of associations that can be stored in an SDM as a function of the number of location units

3.5.1 Real valued inputs

If the elements in the input patterns are real then the fixed weights of the connections going to the location units will also be real. The addresses of the location units represent random points in the continuous input space. For example, in Figure 3.6 the input space of a two-input SDM is represented by a plane. Each dot on the plane in Figure 3.6 represents the address of one location unit.

A location unit becomes active if its address is "close" to the input pattern. For binary input patterns, the Hamming distance metric was used to measure distance. For real inputs, a number of metrics can be used to measure distance. Two metrics are described here:

- one creates an N- dimensional sphere of active location units around the input point, and
- the other creates an N-dimensional cube of active location units.

An N-dimensional sphere is created if the Euclidean distance is calculated between the input points and the location unit addresses, and then only those units that are less than r away from the input point become active:

$$O_i^{[1]}(t+1)=\begin{cases} 1 \text{ if } \sqrt{\sum_{j=1}^{N}\left(O_j^{[0]}(t)-w_{ij}^{[1]}\right)^2}>r \\ 0 \text{ otherwise} \end{cases} \quad (50)$$

where $O_j^{[0]}(t)$ the state of input unit j at time t (a real number, given by (28)), $w_{ij}^{[1]}$ is the fixed weight of the connection from input unit j to location unit i, and r is the access radius. Note that in (50) the fixed weights are not used in the standard way. Normally connection weights scale the strength of the signal through the connection. This method creates an N-dimensional sphere of active location units. The sphere is centred around the input point and has a radius, r. This is shown graphically in Figure 3.6, where a circle is shown centred around an input point, (0.5, 0.5) with a radius of r=0.2. The points within the circle represent the set of active location units when the input is (0.5, 0.5). Also shown in Figure 3.6 is a square. This square represents the set of active location unit if the activation is given by the second method:

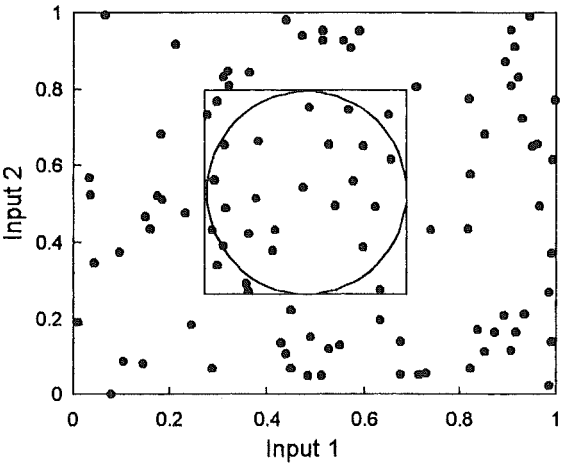


Figure 3.6 A graphical representation of the fixed weights and active location units of an SDM with two real inputs.

$$O_i^{[1]}(t+1)=\begin{cases} 1 \text{ if } \left|O_j^{[0]}(t)-w_{ij}^{[1]}\right|<r, \text{ for } j=1..N \\ 0 \text{ otherwise} \end{cases} \quad (51)$$

The expression in (51) requires significantly less computation than (50) and so is more desirable. The extra location units activated by this method will have a negligible effect on the performance of the memory.

(50) and (51) show how real inputs can be used to obtain a set of active location units. The

SDM then operates in exactly the same way as it does when there are binary inputs. The output is given by (30), if the outputs are binary, or (52) if the outputs are real. Similarly the weights are given by (33) or (49) if the outputs are binary and (54) if they are real.

3.5.2 Real valued outputs

Real valued outputs can be obtained by the output units averaging the weights from the active location units, i.e. the output of unit i is:

$$O_i^{[2]}(t) = \frac{h_i^{[2]}(t)}{\sum_{j=1}^{s} O_j^{[1]}} \tag{52}$$

where $h_i^{[2]}(t)$ is given by (31). A different method of determining the values of the variable weights is required if real valued outputs are to be stored. One way of determining the weights is through an iterative algorithm which works by defining an error for every training pattern at every output, E_i^l:

$$E_i^l = Y_i^l - O_i^{[2]l} \tag{53}$$

where Y_i^l is the desired output and $O_i^{[2]l}$ is the actual output when the input is X^l. This error is corrected if it is too large by adjusting the weights:

$$\Delta w_{ij}^l = \begin{cases} \dfrac{\alpha O_j^{[1]l} E_i^l}{\sum_{k=1}^{s} O_k^{[1]l}} & \text{if } |E_i^l| > \xi_i \\ 0 & \text{if } |E_i^l| \le \xi_i \end{cases} \tag{54}$$

where ξ_i is an acceptable error for output i and α is a learning rate.

When real outputs are stored, the number of associations that can be stored in the SDM is less than when binary outputs are stored, as will now be shown. Each association, (X^l, Y^l), to be

stored puts the following restraint on the weights:

$$|Y_i^l - O_i^{[2]l}| < \xi_i \tag{55}$$

For small ξ_i this is approximately equivalent to:

$$O_i^{[2]l} = Y_i^l \tag{56}$$

If (56) is true then the association will be stored correctly. When there are N_p associations to be stored there are N_p linear equations, (56), in s variable weights which must be true for each output unit. Standard linear algebra shows that the maximum number of independent equations in s variables for which a solution exists is s, so the maximum number of associations that can be stored in the SDM when the outputs are real is equal to the number of location units, s. This is half the number of binary associations that can be stored.

3.6 A comparison between the SDM and Hopfield memory.

The main difference between the SDM and the Hopfield memory is in the number of associations that can be stored. The SDM can store many more associations than the Hopfield memory. The number of associations that can be stored by the Hopfield memory is restricted by the number of inputs, N, whereas for the SDM it is restrained by the rather arbitrary number of location units, s ($s < 2^N$ for binary inputs).

The SDM achieves this greater capacity by performing a random mapping from the N inputs to the s location units. This mapping increases the number of variable weights to each location unit from N (for the Hopfield memory) to s (for the SDM).

Another significant difference between the two memories is that there is no guarantee that the SDM will converge to a stable pattern when it is configured as a content addressable memory. The Hopfield memory does have this guarantee if Hopfield's original method of assigning weights is used, (7).

3.7 Summary

The sparse distributed memory is an associative memory consisting of three layers of units: input units, location units and output units. There are two sets of connections, a set of connections with fixed, random weights from the input units to the location units, and a set of connections with variable weights from the location units to the output units. The memory can be configured with either binary or real inputs and outputs.

Computer-based experiments were performed to verify the theoretical performance of the SDM. These experiments showed that there was a significant difference between the theoretical and actual error rates. This difference occurs because there was an incorrect assumption that the data has a variance of zero. A corrected theory is proposed which accurately predicts the experimental error rates.

An improved method of reading from the SDM was proposed which reduces the effect of the non-zero variance of the data. This method performs just as well when the mean of the data is not zero, unlike the original method which deteriorates quickly.

Gardner's analysis of the capacity of the Hopfield memory applies equally well to the SDM. The iterative algorithm proposed by Gardner can also be adapted to the SDM, predicting that the maximum number of associations that can be stored will be 2s. This was verified experimentally.

The maximum number of associations that can be stored in the SDM is 2s which is much greater than the 2N associations that can be stored in the Hopfield network. This is because the number of location units can be much greater than the number of inputs. The capacity is only restrained by $s<2^N$ for binary inputs. For real inputs there are no theoretical restraints on the number of location units (and hence the capacity).

CHAPTER 4

CMAC

4.1 Introduction

Proposed by Albus (1975a & 1975b), the cerebellar model articulation controller (CMAC) is an associative memory whose structure is based on that of the cerebellum in the human brain. This chapter describes the CMAC which is used in an experiment described in chapter 7. The format of this chapter is as follows:

- section 4.2 describes how the CMAC associates input and output patterns consisting of sets of real numbers, (an example is given in section 4.2.1),
- section 4.3 describes how binary patterns can be stored in the CMAC, and
- the CMAC is compared to the SDM in section 4.4.

4.2 The cerebellar model articulation controller (CMAC)

Albus (1975a) proposed the CMAC, outlined below, as a realistic method for controlling robot manipulators. Put simply, controlling a manipulator involves finding what each actuator should do at every point in time and under every set of circumstances. The signal to any one actuator is a function of time and a number of other variables, such as the state of other joints and sensory feedback data. Calculating the signals to carry out manipulator tasks routinely performed by biological organisms would require an enormous amount of computation (Albus gave the example of squirrels leaping from tree to tree).

As an alternative to explicitly calculating the complex functions Albus proposed using a form of look-up table, where the input variables used in the function point to a location in the look-up table, which contains the outputs. These outputs would be connected to actuator control circuits if the look-up table was being used to control manipulators. The structure of the look-up table was modeled on the way the cerebellum stores data, which Albus believed to be intimately involved in the motor control process. The resulting CMAC network is able

to store complex functions with many input and output variables. It is also able to interpolate between stored points (generalize). The CMAC is described below and an example is outlined in section 4.2.1.

The input to the CMAC network consists of a set of N numbers which together are called the input pattern. The set of all possible input patterns is an N-dimensional space called the input space. An input pattern can also be referred to as a point in input space or, simply, an input point. The output of the CMAC is a set of M numbers called the output pattern. The CMAC can have binary outputs (see section 4.4) but for the moment only output patterns with real valued numbers will be considered.

The CMAC is able to store a set of associations. Each association consists of an input pattern, X (X_1, X_2, ... X_N) and a corresponding output pattern, Y (Y_1, Y_2, ... Y_M). If there are N_p associations to be stored then this set of associations is $\{(X^1, Y^1), ..., (X^{Np}, Y^{Np})\}$, where ($X^I$, Y^I) is one association.

The CMAC, like the SDM, can be viewed as an associative memory consisting of three layers of units:

- a layer of N input units, one for each number in the input pattern, layer 0,
- a layer of s location units, layer 1, and
- a layer of M output units, one for each number in the output pattern, layer 2.

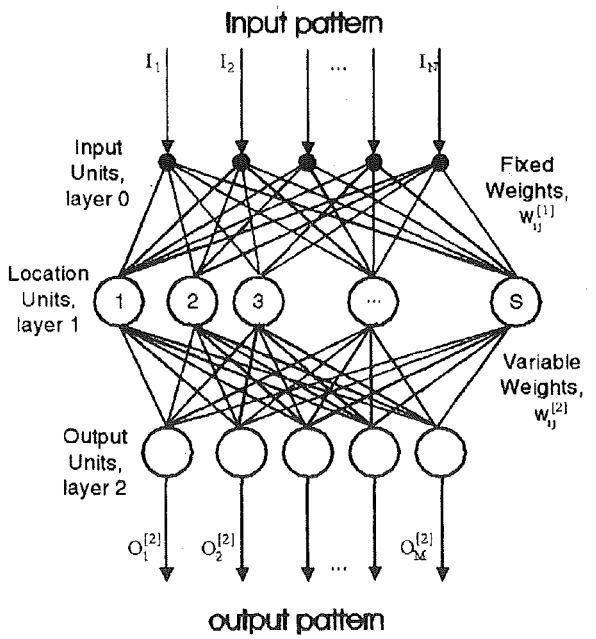


Figure 4.1 A diagram of Albus's cerebellar model articulation controller (CMAC).

There are two sets of connections between these layers of units:

- a set of connections from the input units to the location units, the weights of which are fixed, and

- a set of connections from the location units to the output units, the weights of which change according to the associations stored in memory as described below.

A diagram of the connections between the units is shown in Figure 4.1. Note that this structure is exactly the same as that of the SDM described in the previous chapter (Figure 3.1).

The input units operate in the same was as the input units in the SDM, (28).

There are N connections from the input units to each location unit. The N weights of these connections specify a point (or address) in the N-dimensional input space for each location unit. The set of weights of the connections going to a location unit are referred to as the *location* (in input space) of that unit. The fixed weights are chosen when the network is set up. For the CMAC the weights are chosen so that the weights of the connections going to the location units form a *regular* grid over the input space (see Figure 4.2). This is the major difference between the CMAC and the SDM; in the SDM the weights are chosen randomly.

For example, Figure 4.2 shows the input space for a CMAC with N=2 numbers in the input pattern. The possible values of each number in the input pattern varies between 0 and 1. Each dot in Figure 4.2 represents the weights of the two connections going to a location unit from the two input units. For example the dot in the top left-hand corner represents the weights {0.1, 1.0} going to a location unit. In this case the weights form a regular 10x10 grid over the input space.

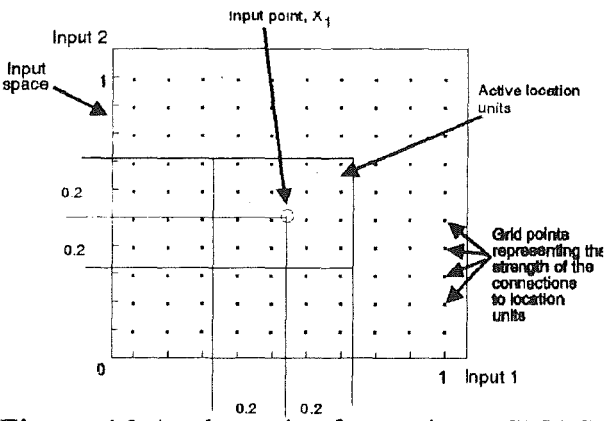


Figure 4.2 A schematic of a two input CMAC. Each dot represents the weights to one location unit.

An input pattern activates a set of location units which have locations that are "close" to the input pattern. The location units have two possible outputs: 1 (active) and 0 (not active). A

location unit is active if its location (given by the fixed weights) is close to the input pattern, I. For the CMAC the active location units form an N-dimensional rectangle in input space around the input point (see Figure 4.2). The activation of the location units is given by the expression (57):

$$O_i^{[1]}(t+1)=\begin{cases}1 \text{ if } |O_j^{[0]}(t)-w_{ij}^{[1]}|<r_j \text{ for } j=1..N \\ 0 \text{ otherwise}\end{cases} \quad (57)$$

where $O_i^{[k]}(t)$ is the output of the *ith* unit in layer k on time step t, $w_{ij}^{[1]}$ is the fixed weight of the connection from input unit j to location unit i, i=1..s. r_j is termed the access distance for the *jth* number of the input pattern. Note that weights are not used in the normal way in (57). Normally weights modify the strength of the signal travelling through the connection.

The access distances are analogous to the access radius in the SDM. In the example in Figure 4.2 the access distances are the same for both inputs, $r_1=r_2=0.2$. With a spacing of 0.1 between the location units this means that for each input pattern a block of 4x4=16 location units are active, as shown.

In the CMAC an output pattern is generated by reading the variable weights from the active location units. Each output unit simply sums the weights connected to it from the active location units to produce an output:

$$O_i^{[2]}(t+1)=h_i^{[2]}(t+1)=\sum_{j=1}^{s} w_{ij}^{[2]} O_j^{[1]}(t) \quad (58)$$

where $h_i^{[2]}(t+1)$ is the local field of unit i at step t+1 and $w_{ij}^{[2]}$ is the weight of the connection from location unit j to output unit i.

Associations are stored in the CMAC by adjusting the variable weights from the location units to the output units. The weights are adjusted using an iterative training algorithm outlined by Albus (1975b). For this algorithm an error is defined for each association (X^l, Y^l) at each output unit:

where Y_i^l is the desired output of unit i for association *l*, and O_i^l is the actual output of unit

$$E_i^l = Y_i^l - O_i^{[2]l} \tag{59}$$

i (using (58)) when the input pattern is $I=X^l$. The weights are then adjusted by Δw_{ij}^l to correct this error if it is too large:

$$\Delta w_{ij}^l = \begin{cases} \dfrac{\alpha E_i^l O_j^{[1]}}{\sum_{k=1}^{s} O_k^{[1]}} & \text{if } |E_i^l| > \xi_i \\ 0 & \text{if } |E_i^l| \le \xi_i \end{cases} \tag{60}$$

where α is a learning rate, and ξ_i is the size of an acceptable error for output i. The size of ξ_i will depend on what the output is being used for.

This training method has been criticised for being slow to converge by Parks and Militzer (1992), who suggest a number of other training algorithms. They conclude that the best algorithm in terms of convergence and computational requirements is the "maximum error" training algorithm. For this algorithm the weights to an output unit are only updated, using (60), if its error is larger than the error of the previous x iterations, where $1 \le x \le N_p$. The algorithm works best if $x=N_p$. If x=1 then this algorithm is the same as that proposed by Albus.

4.2.1 A simple example of the CMAC

This simple example shows:

- a graphical representation of active location units in the CMAC,
- how sets of neighbouring input points overlap
- the iterative learning algorithm converging to store some sample associations, and
- a surface plot of the data stored in the CMAC.

In this example the CMAC has only N=2 input units, M=1 output unit, and s=100 location

units. The numbers in the input pattern are in the range (0, 1). The weights of the connections going to the location units are distributed over the input space at regular intervals of 0.1, as shown in Figure 4.3, where each dot represents the weights to one location unit. For example the dot in the top left hand corner represents the weights {0.1, 1.0} going to one location unit.

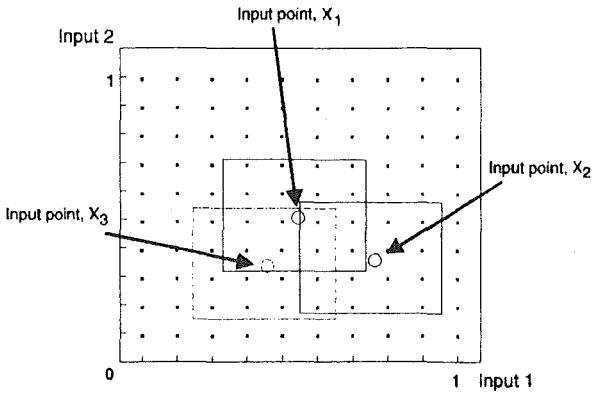


Figure 4.3 The blocks of active location units for three input points, X_1, X_2, and X_3.

Each access distance is set to 0.2, i.e. $r_1=r_2=0.2$. This results in a 4x4 block of active location units becoming active around the input point. Three blocks of active location units are shown in Figure 4.3, for the points $X^1=(0.54, 0.51)$, $X^2=(0.77, 0.36)$ and $X^3=(0.46,0.34)$.

Outputs patterns are going to be associated with the input patterns X_1 and X_2. These output patterns are Y_1= (0.4) and Y_2 =(-0.6), respectively. Input pattern X_3 is used as a test. The blocks of active location units for X_1 and X_2 overlap. The weights of the connections going from the four location units which are activated by both input patterns (X_1 and X_2) will, for this example, be referred to as W_{12}. The weights from the twelve location units activated by only X^1 (and neither of the other two inputs) will be referred to as W_1, and the weights from the twelve location units activated only by X^2 will be referred to as W_2.

Iteration	W_1	W_2	W_{12}	O^1_1	O_1^2	E_1^1	E^1_2	ΔW_1	ΔW_2
0	0	0	0	0	0	0.4		0.025	
1	0.025	0	0.025	0.4	0.1	0	-0.7	0	-0.044
2	0.025	-0.044	-0.019	0.225	-0.6	0.175	0	0.011	0
3	0.036	-0.044	-0.008	0.4	-0.556	0	-0.044	0	-0.003
4	0.036	-0.046	-0.011	0.389	-0.6	0.011	0	0.001	0
5	0.037	-0.046	-0.01	0.4	-0.597	0	-0.003	0	0

Table 5 shows the results of the iterative learning algorithm, (60). This table shows 4 iteration

using (60). Each iteration shows:

- The values of the three groups of weights, W_1, W_2 and W_{12}.
- The value of the output and errors when X^1 and X^2 are presented, O_1^1, E_1^1 and O_1^2 , E_1^2 respectively.
- The value of the change in weights as calculated by (60). These values ΔW_1 and ΔW_2 are both added to W_{12} and are added to W_1 and W_2 respectively.

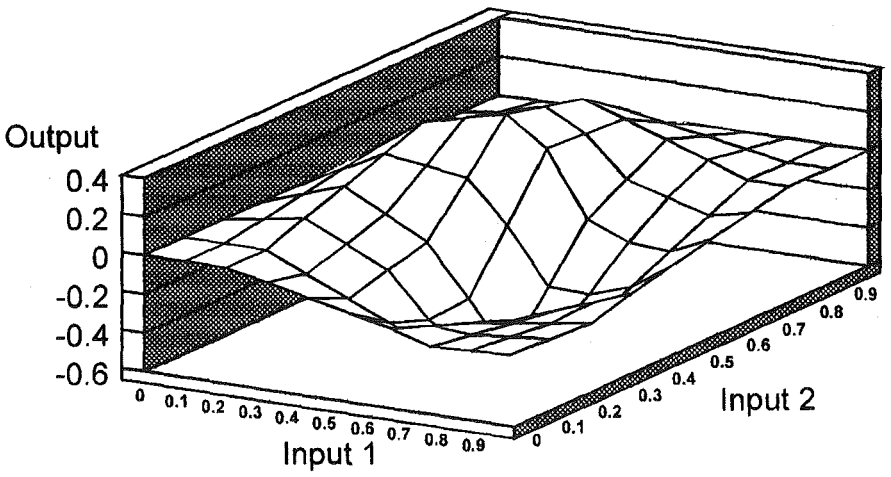


Figure 4.4 A graphical representation of the information stored in the CMAC.

The iterations alternate between using the associations (X^1, Y^1) and (X^2, Y^2). The first iteration uses (X^1, Y^1). Table 5 shows that the CMAC weights converge, for this example, in just five iterations. Figure 4.4 shows a graphical representation of the information stored in the CMAC when the learning is complete. The output has a peak of 0.4 at input X^1=(0.54, 0.51) and a trough of -0.6 at input X^2=(0.77, 0.36). The output for the rest of the input space depends on how much the active location units overlap with those of X^1 and X^2. the output for X^3 will be given by the following expression:

$$\begin{aligned} O_1^3 &= 4W_1 + 2W_{12} + 2W_2 \\ &= 4.(0.037) + 2.(-0.01) + 2.(-0.047) \\ &= 0.033 \end{aligned} \tag{61}$$

The CMAC network has a built-in ability to perform local generalization, i.e. inputs which are close to each other will have similar values. This local generalization occurs because

inputs which are close have overlapping sets of active location units. Figure 4.4 graphically shows this local generalization; inputs which are close to the stored points, X^1 and X^2, have values which are close to those stored at X^1 and X^2. For example X^3 is closer to X^1 than X^2 and its output is closer to that stored at X^1 than that stored at X^2.

4.3 Storing binary patterns

The CMAC is not suited to handling binary inputs, but it can handle binary outputs by comparing the sum of the weights from the active units to a threshold:

$$O_i = \begin{cases} 1 \text{ if } h_i > T \\ 0 \text{ if } h_i \leq T, \end{cases} \tag{62}$$

where T is the threshold and h_i is given by (58). Equation (62) is exactly the same as the equation specifying the binary output for the SDM, (30).

When binary outputs are stored an iterative algorithm can be used, such as the one proposed by Gardner for the Hopfield memory. This algorithm is exactly the same for the CMAC as it is for the SDM, (49).

4.4 Comparing the CMAC and the SDM

The SDM and CMAC are very similar. They both have three layers of units with a layer of fixed weights between the input and location units and a layer of variable weights between the location units and the output units. All the units compute the same functions in the SDM as they do in the CMAC. The only significant difference between the two is in the way the fixed weights are chosen. In the SDM the fixed weights are chosen randomly, whereas in the CMAC the weights are chosen so they are spaced regularly over the input space.

The main effect of this regular spacing rather than random is that the CMAC is easier to implement on a serial computer that the SDM. The difference arises because it is easier to find the set of active location units for the CMAC than it is for the SDM. When an input

pattern is presented to the SDM it must be compared to the address of every location unit, using (29), (50) or (51) (depending on the configuration of the SDM). This can be a lengthy process if there are many location units. The CMAC however can take advantage of the regular spacing of the location units and use a simple algorithm to calculate which units are active. This is much faster than comparing the input pattern with the weights going to every location unit. The ability of the CMAC to run quickly on a serial computer has meant that it has been able to be used in real time control applications (Miller et al, 1988).

If there are a large number of inputs, then the SDM may be a better choice than the CMAC because the number of location units activated by the CMAC will be too large. For example if there are 100 inputs and the memory is configured so that the 100 dimensional rectangle of active location units is two locations wide for each input, then the number of location units activated for any input will be 2^{100}. Obviously it is not practical to implement such a memory. In the SDM however the number of location units could be chosen to reflect the number of associations which are going to be stored. For example, if there are going to be 50 associations stored then about 500 location units could be used.

The capacity of the CMAC, like the SDM is dependent on the number of location units. for binary output patterns the capacity will be 2s (for independent patterns with $\kappa=0$) and for real output patterns the capacity will be s.

4.5 Summary

The CMAC is an associative memory which has the same structure as the SDM. It has three layers of units: input units, location units and output units. There are two layers of weights: fixed weights from the input units to the location units, and variable weights from the location units to the output units.

The points represented by the fixed weights going to the location units form a regular pattern over all of the input space. The variable weights are defined by an iterative learning algorithm. Either binary or real outputs can be stored.

The CMAC and SDM are very similar. The only difference is the way the fixed weights are assigned. The regular pattern of the fixed weights in the CMAC make it easier to implement on a serial computer.

CHAPTER 5

Storing sequences on neural networks

5.1 Introduction

The artificial neural networks in the previous chapters (the Hopfield network, SDM and CMAC) have been configured as either content addressable memories, which store patterns, or associative memories, which associate pairs of patterns. In both cases the patterns used are static, i.e. they do not change over time. For many applications this is adequate, for example, if the neural network is being used to recognise still images. On the other hand many other applications use patterns which change with time; for example, in speech recognition the speech consists of a time varying sound wave. It would be useful if, for these applications, a neural network could use patterns which vary with time. This chapter shows how the Hopfield network and SDM can be adapted to operate with time varying patterns.

For the purpose of this chapter a time varying pattern will be represented as a sequence of patterns

$$(\boldsymbol{P}^1, \boldsymbol{P}^2, \ldots . \boldsymbol{P}^{N_S}) \qquad \textbf{(63)}$$

with a time of τ between consecutive patterns. N_S is the number of patterns in the sequence. An associative memory can be configured to store a sequence of patterns by storing consecutive pairs of patterns as associations. Such a memory would store the following associations: (P^1, P^2), (P^2, P^3), ... The sequence can then be recalled: with P^1 the association (P^1, P^2) recalls P^2, with P^2 the association (P^2, P^3) recalls P^3, and so on. This is explained in more detail in the following section (section 5.2).

When using associative memories to store sequences, a problem arises if a particular pattern occurs more than once in a sequence because two associations would have this pattern as their first pattern. A sequence in which a pattern occurs more than once is called a *complex*

sequence. A significant amount of work has been published on how to overcome this problem and allow Hopfield networks to store complex sequences. This work is summarised in section 5.2.

The number of associations that can be stored in a Hopfield network is limited by the number, N, of elements in the input pattern. This was discussed in section 2.3. A sequence consisting of N_S patterns requires N_S-1 associations to be stored, and so the size of a sequence that can be stored in the Hopfield is also limited by the number of elements in the patterns. This limitation can be overcome by using the SDM. The number of associations that can be stored in the SDM is not limited by the number of elements. The techniques used for storing complex sequences in the Hopfield network can also be used to enable the SDM to store complex sequences. Section 5.3 describes some tests with the SDM using one of these techniques to store complex sequences.

Section 5.4 describes how the techniques for storing complex sequences can be adapted to allow the SDM to recognise sequences. An example application is described in section 5.4.1 which involves recognising spoken digits. The performance of the SDM at recognising the spoken digits is compared to that of a multi-layer neural network trained with the back-propagation algorithm.

5.2 Storing sequences in the Hopfield network

To store sequences, the Hopfield network is configured as an associative memory with feedback as shown in section 2.4.1. That section describes how the Hopfield network can be configured as an associative memory. Sequences are stored by associating consecutive patterns. An association consists of an input pattern and an output pattern (I ,O). So for each association the input pattern is $I=P^l$ and the output pattern is $O=P^{l+1}$ giving the following set of associations: $\{(P^1, P^2), (P^2, P^3), \ldots (P^{Ns-1}, P^{Ns})$. N_s-1 associations are required to store a sequence of N_s patterns.

The set of associations can be stored by using Gardner's iterative algorithm, (26) and (27), to assign the weights. In (26) and (27) X^l is replaced with P^l, and Y^l is replace with P^{l+1}.

Using this algorithm the change of the weight of the connection from unit j in the input layer to unit i in the output layer, for association (P^l, P^{l+1}), is:

$$\Delta w_{ij}^{l}=\begin{cases} \alpha E_i^l P_i^{l+1} P_j^l & \text{if } E_i^l>0 \\ 0 & \text{if } E_i^l\le 0 \end{cases} \quad (64)$$

where E_i^l is the error at output unit i for association (P^l, P^{l+1}):

$$E_i^l=\kappa\sqrt{\sum_{j=1}^{N} w_{ij}^2}-P_i^{l+1}\sum_{j=1}^{N} w_{ij}P_j^l \quad (65)$$

κ is a positive constant.

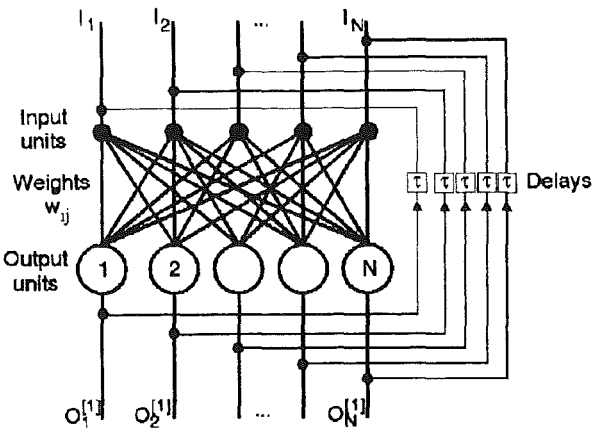


Figure 5.1 The Hopfield network as an associative memory with the same number of inputs as outputs and with feedback connections.

Once the sequence has been stored it can be retrieved by using the first pattern of the sequence as the input, $I=P^1$. When the first pattern of the sequence has been presented the network is in input mode. In input mode the state of the input units is given by the input signals, (24). After the first pattern has been presented in input mode the network goes into retrieve mode. In retrieve mode the state of the input units is given by the state of the corresponding output unit, (21). The state of the output units is given by (22) and (23).

The time steps that are represented by t in (24), (21), (22) and (23) should have a duration of τ. If the associations have been stored correctly then after the first pattern has been presented to the input the output will be pattern P_2 after a time of τ. The next output should be P_3, then P_4, until the whole sequence has been recalled.

In the format outlined above, the Hopfield memory is only able to store *simple sequences*. A *simple sequence* is one in which each pattern in the sequence occurs only once, i.e. $P^l \neq P^k$ for $l, k = 1..N_S$. In this configuration the Hopfield memory is only able to store simple sequences because the output generated using (20) , (22) and (23) depends only on the current input

pattern, and not any of the other input patterns. This is explained further in an analogy to a computer memory below.

The Hopfield memory can be compared by analogy to a computer memory in the same way that the SDM was in Chapter 3, section 3.2.1. The input patterns are analogous to addresses and the output patterns are analogous to data in a computer memory. Storing an association (P^l, P^{l+1}) is analogous to storing data P^{l+1} at address P^l. Trying to store a sequence in which a pattern occurs more than once is analogous to trying to store two different data at the same address. For example if patterns l and k in a sequence are the same, $P^l = P^k$, then the following associations would be stored as part of storing the whole sequence: (P^l, P^{l+1}) and (P^k, P^{k+1}). This is analogous to storing data P^{l+1} and data P^{k+1} at address P^l=P^k which is not possible for a normal computer memory or for the Hopfield network as described above. Hence sequences in which patterns occur more than once can not be stored on the Hopfield network in this configuration.

A sequence in which a pattern occurs more than once is called a *complex sequence*. The *order of complexity* of a sequence is the length of the largest repeating sub-sequence within the sequence. The order of complexity will be referred to as C. Some examples of sequences with different orders of complexity are given in Figure 5.2. A simple sequence has an order of complexity of C=0.

(A, B, C, D, E, F, G, H, I)
C=0

(A, B, C, D, B, E, F, B)
C=1

(A, B, C, D, E, B, C, F, G)
C=2

Figure 5.2 Letters in the 3 sequences above represent patterns. Shown are examples of sequences with C of 0, 1 and 2. The repeating sub-sequences are underlined.

At any position in a sequence, which has an order of complexity of C, it is necessary to know the previous C+1 patterns in order to unambiguously determine what the next pattern will be. For example, in simple sequence (C=0) it is only necessary to know the previous pattern to determine what the next pattern will be. In another example, the third sequence shown in Figure 5.2 has an order of complexity of C=2. For this sequence it is necessary to know the previous three patterns in order to determine the next pattern. If for example it was only known that the previous two

patterns were B and C then it would not be possible to determine if the next pattern was D or F.

One way to allow the Hopfield network to store complex sequences is to somehow allow the output to depend, not only on the current input, but also on previous inputs. Two techniques for achieving this are discussed below:

- using additional connections with delays (section 5.2.1), and
- using decaying local fields (section 5.2.2).

5.2.1 Using additional connections with delays to store complex sequences

To store complex sequences it is necessary that the output is influenced by at least the previous C+1 patterns. One method of allowing the output of the Hopfield network to depend on previous patterns is to include additional connections with delays between units (Guyon & Personnaz, 1988; Kleinfield, 1986; Kuhn et al, 1989), as shown in Figure 5.3. Each connection has an adjustable weight which indicates the strength of that connection. In this configuration the local field is:

$$h_i(t) = \sum_{j=1}^{N} \sum_{k=1}^{C+1} w_{ijk} O_j(t-k) \qquad (66)$$

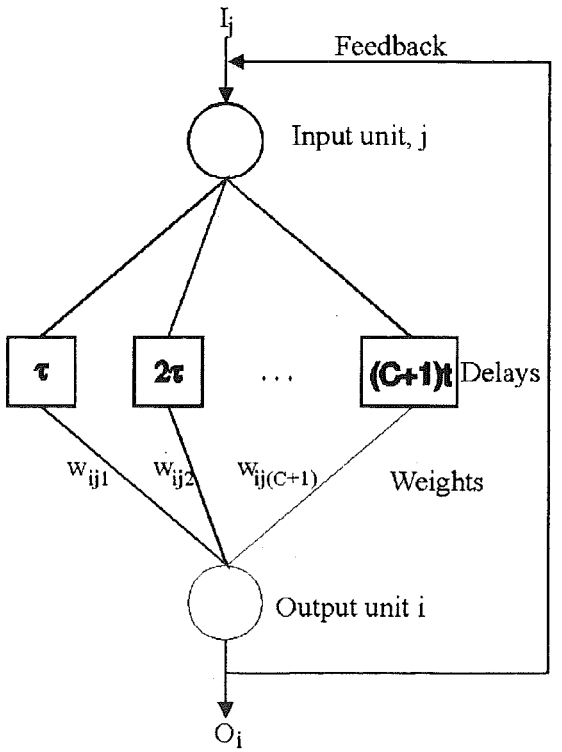


Figure 5.3 Input units have C+1 connections to each output unit. Each connection has a time delay, and a variable weight, indicating the strength of the connection.

The local field is now a weighted combination of the previous C+1 patterns. The value C is the order of complexity of the sequences being stored. C+1 is the minimum number of previous patterns which are required to unambiguously determine the next output.

The weights can be determined using an iterative algorithm based on Gardner's, (64) and (65) (as suggested by Bauer & Krey, 1991). When pattern *l* of the sequence is being stored the change of the weight of the connection from input unit j to output unit i, delayed by kτ, is:

$$\Delta w_{ijk} = \alpha E_i^l P_i^l P_j^{l-k} \tag{67}$$

where the error, E_i^l is:

$$E_i^l = \kappa \sqrt{\sum_{k=1}^{C} \sum_{j=1}^{N} w_{ijk} - P_i^l h_i^{l-1}} \tag{68}$$

where κ is a non-negative constant and h_i^{l-1} is the local field, (66), when P^{l-1} is the current input. This technique for storing sequences will be referred to as the *multiple delayed weights* technique.

Other methods of assigning weights for this type of Hopfield network are discussed by Guyon & Personnaz, (1988).

The concept of weights from delayed inputs has also been applied to back-propagation networks by Wan (1993). A derivation of the back--propagation algorithm (Rumelhart, McClelland et al, 1986; Parker 1985) is used to assign the weights. The resulting discrete time neural networks were found to be effective at time series prediction. Each set of delayed weights between units forms a finite impulse response (FIR) filter. Wan (1993) argues that the FIR filter models the processes of axonal transport, synaptic modulation and charge dissipation in biological neurons better than the single weight normally used.

Another method of allowing past inputs to affect the current output is to have only one weighted connection between each input and output unit, but give each connection a fixed random delay, $T_{ij}\tau$, where T_{ij} is a random positive integer within a specified range, $1..T_{ij}^{Max}$ (Bauer & Krey, 1990, 1991; Herz et al, 1988, 1989; Mato & Parga, 1991). The local field is then given by the following expression:

$$h_i(t)=\sum_{j=1}^{N} w_{ij} O_j(t-T_{ij}) \qquad (69)$$

The output therefore depends on a random combination of the previous outputs. A learning algorithm similar to those above can be used to assign the weights.

A network with fixed random delays can store complex sequences, if some of the delays go back at least C+1 steps (C is the order of complexity), i.e. $T_{ij}^{Max} > C$. However there are some sequences that will not be able to be stored. From (69) it can be seen that output i at time t depends on a pattern made up from elements of the previous T_{ij}^{Max} outputs:

$$(O_1(t-T_{i1}),\ O_2(t-T_{i2}),\ldots,\ O_N(t-T_{iN})) \qquad (70)$$

If this particular combination of elements in a set of patterns occurs more than once in a sequence then the sequence will not be able to be stored correctly. For example, a network with four input units has the following delays associated with the connection to the first output unit: $T_{11} = T_{12} = 2$ and $T_{13} = T_{14} = 1$. The sequence of patterns shown in Figure 5.4 could not be stored on this network because when the sequence is being recalled the local field for output unit 1, h_1, will be the same when recalling patterns P_3 and P_5 but the outputs are supposed to be different.

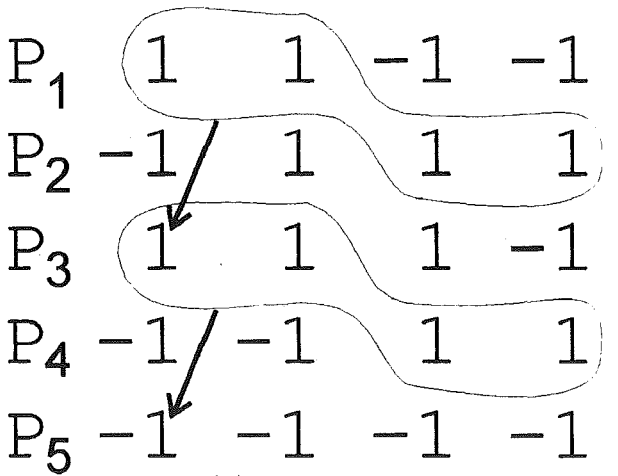


Figure 5.4 A delayed pattern occurs twice, making the sequence of patterns impossible to store (for some delays).

5.2.2 Using decaying local fields to store complex sequences.

Another method of allowing the output of the Hopfield network to depend on previous

patterns is to modify the way the local field of the output units is calculated so that it represents a decaying sum of all of the previous inputs to that unit:

$$h_i(t+1)=\sum_{j=1}^{N} w_{ij}\sum_{k=1}^{t} I_j(k)d^{t-k} \\ =dh_i(t)+\sum_{j=1}^{N} w_{ij}I_j(t) \tag{71}$$

where d is the decay rate, d<1. This modification to the Hopfield network was suggested by Bressloff & Taylor (1992). This modification allows complex sequences to be stored because the local fields and hence the state of the output units is often a function of all the previous patterns. Again the weights of the connections can be assigned using Gardner's algorithm, (64) and (65).

This technique for storing complex sequences will be referred to as the *decaying local fields* technique. A similar technique for storing complex sequences on a neural network was devised by Wang & Arbib (1990).

The decaying local fields technique for storing complex sequences has some advantages over the multiple delaved weights technique described in the previous section. The main advantage is that fewer weights are required for the decaying local field technique (N^2 weights) than multiple delayed weights technique ($(C+1)N^2$ weight). The second advantage is that if decaying local fields are being used then it is not necessary to know the order of complexity of the sequences before determining the format of the network. For the delayed weights technique the required number of delayed weights depends on the order of complexity of the sequences which are being stored on the network.

A sequence of N_s patterns requires N_s-1 associations to be stored in order to store the sequence. The number of associations that can be stored on the Hopfield network with decaying local fields will depend on the number of inputs N. Hence the maximum length of the sequences that can be stored will also depend on the number of inputs, N. One way to overcome the limitation on the maximum length of sequences that can be stored is to use an

SDM to store the sequences, as will now be described.

5.3 Storing complex sequences in the SDM

To store sequences on the SDM it should be configured with the same number of inputs and outputs, N=M, and, like the Hopfield network, the outputs should be fed back to the inputs. In this format, shown in Figure 5.5, the SDM is able to store only simple sequences because the output pattern depends solely on the input pattern. Kanerva (1988) suggested that, to store complex sequences, weights from delayed inputs should also contribute towards the output. This is equivalent to the delayed weights configuration of the Hopfield memory described above.

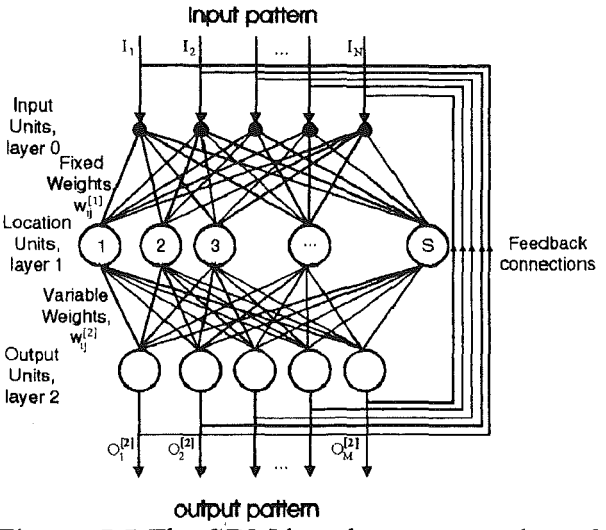


Figure 5.5 The SDM has the same number of inputs and outputs, and has feedback connections from the output to the input.

Another method of storing complex sequences on the SDM is to change the SDM so that the influence of the inputs decays exponentially. This is an adaptation of the techniques used to store complex sequences by Bressloff & Taylor (1991) and Wang & Arbib (1990). One way of allowing previous inputs to have a decaying affect is to change the behaviour so that the activations of the location units decay exponentially with time. The activation of the location units is then given by:

$$O_i^{[1]}(t+1)=\begin{cases} 1 & \text{if } \sum_{j=1}^{N} O_j^{[0]}(t)\,w_{ij}^{[1]} \geq N-2r \\ dO_i^{[1]}(t) & \text{if } \sum_{J=1}^{N} O_j^{[0]}(t)\,w_{ij}^{[1]} < N-2r \end{cases} \qquad (72)$$

where d is the decay rate, $0 < d < 1$. The output units can operate in the same way as in the

original memory, (30). The only difference is that in the calculation of the local field, (31), the activations will now have real values $0 \leq O_j^{[1]}(t) \leq 1$. In the original SDM the activations could only be 0 or 1.

The variable weights are assigned using Gardner's iterative algorithm, where the change of weight from location unit j to output unit i, for association (P^k, P^{k+1}) is:

$$\Delta w_{ij}^k = \begin{cases} \alpha O_j^{[1]k} P_i^{k+1} \dfrac{E_i^k}{\sum_{l=1}^{s} a_l^k} & \text{if } E_i^k > 0 \\ 0 & \text{if } E_i^k \leq 0 \end{cases} \qquad (73)$$

where $O_j^{[1]k}$ is the activation of location unit j when the input is P^k. The error is

$$E_i^k = \kappa \sqrt{\sum_{j=1}^{s} w_{ij}^2 - P_i^k h_i^k} \qquad (74)$$

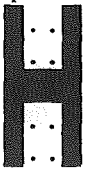
This is similar to the Hopfield network with exponential decays proposed by Bressloff & Taylor (1991), but there is an advantage with the SDM in that it has a much larger capacity (as outlined in Chapter 3).

5.3.1 An example

To demonstrate that the SDM can store complex sequences an SDM was simulated with the above configuration, using N=20 inputs and outputs, s=500 location units, an access radius of r=7, and a decay rate of d=0.5. The inputs were arranged in blocks of 5 rows of 4, so that letter shapes could be represented by the inputs. For example the pattern:

(1 -1 -1 1 1 -1 -1 1 1 1 1 1 1 -1 -1 1 1 -1 -1 1)

represents the letter 'H' on a 5x4 grid:



where ■ represents +1 and . represents -1

Two sequences, representing the words HELLO and GOODBYE, were stored using the learning algorithm described above (with κ=0). They were stored as *cyclic sequences*. A cyclic sequence is one in which the first pattern is associated with the last pattern. After learning, the presentation of an H to the input would result in the following repeating sequence:

where each block of 20 (5x4) represents the input at a different time. Similarly the presentation of a G would result in:

These sequences have an order of complexity of 2. The sequences could also be recalled if there was some noise in the original input pattern.

5.4 Recognising sequences

Up to this point it has been discussed how the SDM can be used to store and retrieve sequences. It is also possible to use the SDM to recognise sequences. This can be achieved by configuring an SDM with one output for each sequence to be recognised and by allowing the activations to decay exponentially (72). There are no feedback connections.

Sequences are learnt by first presenting the whole sequence to the network, pattern by pattern, allowing the activation of the location units to decay after each pattern (72). This results in a pattern of decaying activations over the location units which is then associated with an output pattern which has a 1 at the unit used to represent that sequence and -1's elsewhere.

The association is stored using Gardner's iterative algorithm.

A sequences is recognised by presenting it to the network, pattern by pattern, allowing the activation of the location units to decay, and then calculating the local fields for each output. There is one output for each sequence to be recognised. The sequence is recognised as the sequence represented by the output which has the highest local field.

To test the ability of the SDM to recognise sequences, an SDM was set up with the same configuration as in 5.3.1 but with only two outputs (one for each sequence to be recognised). The sequence HELLO was associated with the first output and the sequence GOODBYE was associated with the second output, using the technique outlined above.

After the two associations had been stored the memory was easily able to identify the two sequences correctly. After the sequence HELLO was presented to the network, pattern by pattern the first output had a higher local field and after the sequence GOODBYE was presented the second output had a higher local field. A more demanding test of the ability of the SDM to recognise sequences was performed. These tests involving the recognition of spoken digits are outlined in the following section.

5.4.1 A sequence recognition example - recognising speech

This task involves recognising the spoken digits 0-9. There were 8 samples of each digit (not all from the same speaker). Each sample was processed to give a formant track for that sample[11]. The formant track consists of ten points in sequence, each point given by two values which represent the two dominant formants. Figure 5.6 shows the average formant tracks for each digit (normalised to values between 0 and 1).[12]

---

[11]See Witten (1982) for an introduction to speech processing and a description of formants.

[12]Each of these formant trajectories consists of 10 points. The trajectory for the digit '6' contains several zero points because it has very little spoken sound in comparison to the other digits.

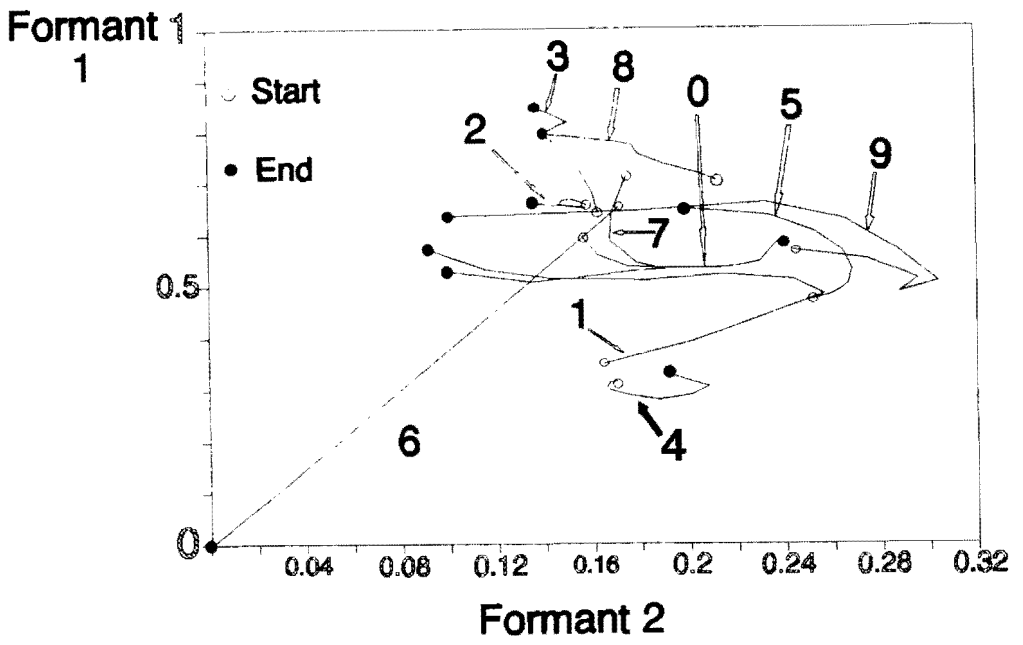


Figure 5.6 Averaged formant tracks for the digits 0-9

A memory was set up with n=2 inputs (1 for each formant), m=10 outputs (1 for each digit to be recognised), s=500 location units and an access radius of r=0.1. A decay rate of d=0.9 was used. Because the inputs are real valued, the randomly selected location units also have real values chosen randomly, R_{ij}, $0<R_{ij}<1$. After a series of experiments it was found that the best performance is obtained when the distribution of the location unit addresses matches that of the input patterns (as suggested by Keeler [26]). Figure 5.7 shows an example of randomly selected location units. With real valued inputs the activation of a location unit is given by:

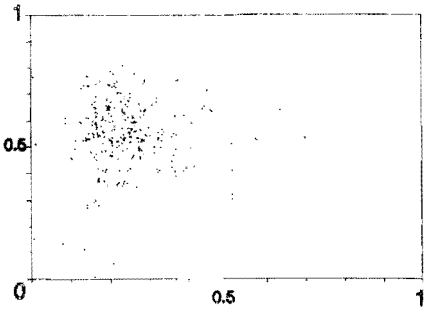


Figure 5.7 The addresses of the location units have the same distribution as the inputs. Each dot represents the address of one location unit.

$$O_i^{[1]}(t+1)=\begin{cases} 1 & \text{if } \sum_{j=1}^{N} |O_i^{[0]}(t)-w_{ij}^{[1]}| \geq r \\ dO_i^{[1]}(t) & \text{if } \sum_{j=1}^{N} |O_i^{[0]}(t)-w_{ij}^{[1]}| < r \end{cases} \quad (75)$$

In the first test the memory was trained with half of the data (selected randomly) and tested over the full set. The best performance gave 95% recognition (76/80). In a second test the memory was trained using the average formant tracks (shown in Figure 5.6) and tested using the original data. The best performance for this test was 93.75% (75/80) of the digits being recognised correctly. This performance equalled the best a back-propagation network could do with exactly the same data[13]. The 20-10-10 back-propagation network took approximately 20000 iterations[14] during its training whereas the SDM took only 61 iterations. The total error is shown in Figure 5.8 as a function of the number of iterations. The total error is the error used in Gardner's training algorithm summed over all the outputs and training sequences.

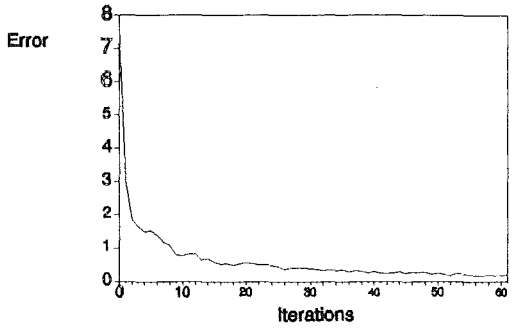


Figure 5.8 The total error decreases to approximately 0 after 61 iterations

5.5 Summary

Simple sequences of patterns can be stored in an associative memory by storing associations consisting of consecutive patterns. The first pattern in the sequence is used to recall the rest of the patterns.

A complex sequence has patterns occurring more than once in it. Complex sequences cannot

[13]Thanks to John Kirkland for supplying both the formant data and the results of his tests with a backpropagation network (personal communication).

[14] The stopping criterion was Kirland's

be stored in associative memories as described above because it would be necessary to store associations which have the same inputs but different outputs.

Several techniques for storing complex sequences in the Hopfield network have been proposed. These techniques involve either using extra sets of connections which incorporate delays or using decaying activations. These techniques used to store complex sequences in the Hopfield network can also be successfully applied to the SDM. The advantage of using the SDM over the Hopfield network is that the SDM has a larger capacity and so more or larger sequences can be stored. Simple examples confirm that sequences can successfully be stored on the SDM using the decaying activations method.

The same techniques used for storing sequences in the SDM can be used to recognise sequences. This is confirmed using both simple examples and a more complex speech recognition example. In the speech recognition example the SDM with decaying activations is able to recognise formant tracks with the same accuracy as a backpropagation network. The SDM however takes much fewer iterations to train (50 iterations) than the backpropagation network (20000 iterations).

CHAPTER 6

The PURRPUSS system

6.1 Introduction

PURR-PUSS (PP for short) is a system designed to enable a robot to learn. The PP system has been developed over several years [Andreae, 1977; Andreae & MacDonald, 1991; Andreae et al, 1993] and it continues to be developed. This chapter describes one implementation of the PP system and gives an example of its operation. There are several features that have been used in previous PP systems (such as threading events, recency productions and contexts of contexts) which are not mentioned in this thesis because they are not relevant (see Andreae 1974-1991 for a full account of the PP system). There are two features new to PP that are introduced in this chapter: learning from random actions, and the goal-setting GO command. Chapter 7 shows how the PP system can be combined with the CMAC system and together learn to effectively control continuous outputs.

There is little consensus on how much knowledge is innate in a human brain and how much is learnt through experience. In an attempt to shed some light on this, the PP system was designed to contain as little built-in (or innate) knowledge as possible. The PP system has built into it reflexes and two learning mechanisms. Reflexes are robot movements in response to some triggers (usually specific stimuli). The learning mechanisms are called production learning and leakback learning (described in sections 6.2 and 6.3 respectively).

The PP system is goal seeking, that is, it chooses actions that, according to its calculations will lead towards goals. Section 6.5 discusses how goals are given to PP. Section 4.6 demonstrates how PP operates by showing how PP can learn to move a ball in a tilting maze towards a goal.

Previous research has shown that the PP system, with its minimal built in knowledge can, among other things (Andreae, 1972-1991):

- emulate a Universal Turing machine and so can calculate any computable

function [MacDonald & Andreae, 1981],

- learn from reflexes [MacDonald, 1984], and
- learn multilevel tasks [Andreae et al, 1993].

6.2 Production learning

For this thesis a robot will be considered to consist of a body and a brain (or control centre) that controls the body. The body has several effectors that perform actions and influence the environment in which the robot exists. The robot also has a number of sensors that respond to stimuli in the environment. The PP system is the robot controller; it sends signals to the robot effectors and receives signals from the robot sensors.

The PP system operates in discrete time. In each time step PP can send signals to its effectors and receive signals from its sensors. If there are m effector signals and n sensor signals then there are m+n=S signals in any particular time step. These signals are termed events, signals to effectors are called output events and signals from sensors are called input events. For example an output event may be **move_arm** "up", where **move_arm** indicates which effector the signal is sent to (the arm) and "up" represents the signal. This signal would cause the robot's arm to move up. Figure 6.1 shows the channels of communication between the environment, effectors, sensors, short and long term memory (STM and LTM) and the reflexes. All these parts are explained below.

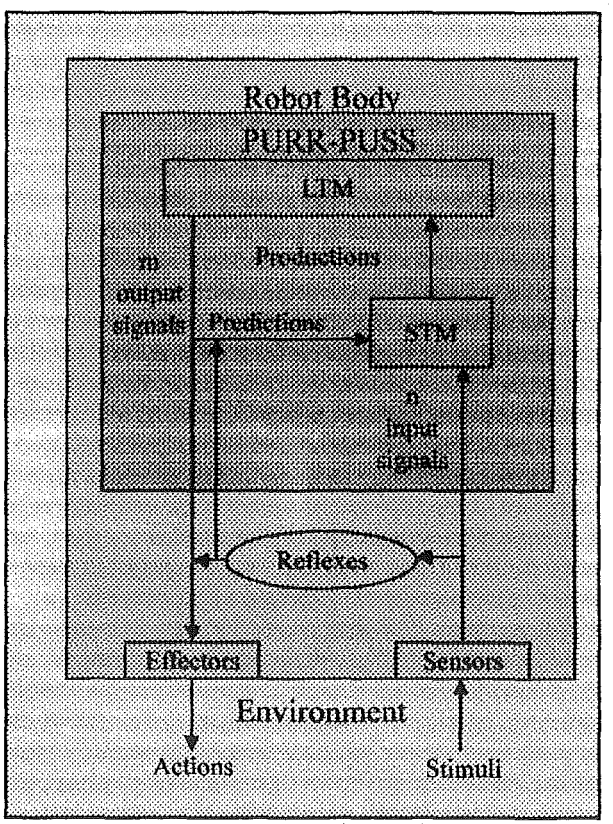


Figure 6.1 A simple schematic of the PP system. PP sends signals to the robot effectors and receives signals from the robot sensors.

6.2.1 Short Term Memory (STM)

PP learns from the S sequences of discrete events, which either go to the effectors or come from the sensors. These sequences can be represented as follows:

$$\begin{aligned} Seq_1&: \ldots E_{1,-2},\ E_{1,-1},\ E_{1,0},\ \ldots \\ Seq_2&: \ldots E_{2,-2},\ E_{2,-1},\ E_{2,0},\ \ldots \\ &\vdots \qquad\qquad \vdots \\ Seq_s&: \ldots E_{s,-2},\ E_{s,-1},\ E_{s,0},\ \ldots \end{aligned} \tag{76}$$

where sequence Seq_s, s=1..S is either an input sequence coming from the sensors or an output sequence going to the effectors. At any time, the current events (the signals currently going to the effectors and coming from the sensors) are described by the terms $E_{s,0}$. The events immediately before the current events are given by $E_{s,-1}$, and the events before that are given by $E_{s,-2}$. Thus all the events are defined relative to the current events. Each input and output event can have a range of values (including a "null" value) depending on what is being represented. The last x events, for some fixed x, of each sequence are stored in STM, i.e. the events $E_{s,0}$ to $E_{s,-x+1}$.

For example, in Figure 6.2 there are five sequences of events, two input sequences, coming from the eye and the ear, representing what is seen (e.g. '**see** square 1') and what is heard, and three output sequences, representing a body movement (**tilt**), an eye movement (**look**) and an execution action (**go**). Figure 4.2 shows how the sequences may progress.

	Time Step		1	2	3	
Stimuli						
	Sequence 1	**see**	square 1	square 1	square 2	...
	Sequence 2	**hear**	"move_2"	null	null	...
Actions						
	Sequence 3	**tilt**	east	south	east	...
	Sequence 4	**look**	null	square 2	null	...
	Sequence 5	**go**	null	null	GO	...

Figure 6.2 An example of five sequences. The discrete time steps have been labelled 1, 2, 3, ... The stimuli are sensed in the first half of each time step and the actions are performed in the second half.

In Figure 6.2 a "null" indicates that there is no signal going to the effector (if it is an output sequence) or coming from the sensor (if it is an input sequence) during that time step. In this case at step 1, the current events held in STM are:

$E_{1,0}$ = '**see** square ', $E_{2,0}$ = '**hear** "move_2"', $E_{3,0}$ = '**tilt** east',

$E_{4,0}$ = '**look** null', $E_{5,0}$ = '**go** null'.

In step 2 the current events in STM are

$E_{1,0}$ = '**see** square 1', $E_{2,0}$ = '**hear** null' $E_{3,0}$ = '**tilt** south',

$E_{4,0}$ = '**look** square 2' $E_{5,0}$ = '**go** null',

and the events prior to the current events (the previous current events) are also held in STM:

$E_{1,-1}$ = '**see** square 1', $E_{2,-1}$ = '**hear** "move_2"', $E_{3,-1}$ = '**tilt** east',

$E_{4,-1}$ = '**look** null', $E_{5,-1}$ = '**go** null'.

6.2.2 Long term memory (LTM)

PP learns by storing *productions* in LTM according to a set of predetermined *production templates*, T_k. These productions allow PP to make *predictions* which are used for planning and choosing actions. This section describes what productions are, how they are stored and how they are used to make predictions. Each production consists of a prediction event, and a set of events, called a *context*.

6.2.2.1 Contexts

A context is a group of events that have at some stage occurred together in STM. Production templates prescribe exactly which events in STM are combined to form contexts. A context, C_i, i=1..N_c (N_c is the number of contexts that have been stored), is given by:

$$C_i = E^{il}, E^{12}, ..., E^{il} \tag{77}$$

where the E_{ij} are events from STM (76) as prescribed by the production templates, and l is the number of events in the context.

For example, with the appropriate production templates, a context with two events (l=2) may be:

C_1 = '**hear** "move_2"', '**look** square 1'.

Likewise, a context with one event in it (l=1) may be:

C_2 = '**see** square 1'.

6.2.2.2 Production templates

The production templates prescribe which events from the sequences, Seq_s, are to be associated in productions. An event which is t events before the current event in the sth sequence would be called $E_{s,-t}$. Each template, T_k is of the form:

$$T_k:[E_{s_{k1},-t_{k1}}, E_{s_{k2},-t_{k2}}, \ldots, E_{s_{kl_k},-t_{kl_k}} \rightarrow E_{s_{k0},0} \ ] \qquad (78)$$
$$\textit{Context} \rightarrow \textit{Prediction}$$

where l_k is the number of events in each context which is stored as a result of template k. The events to the left of the "→" prescribe which events from the sequences Seq_s will form contexts, and the event to the right prescribes the prediction events that will be associated with the contexts in LTM. The coefficients S_{kj} and t_{kj}, $j=0..l_k$, prescribe where in the above sequences (76) each event for the productions will come from. The S_{kj} prescribe which sequence, 1..S, and the t_{kj} prescribe which step, relative to the current step, $1..t_{kj}^{Max}$, is to be used.

A simple production template may appear as follows:

T_1: $[E_{2,-1} \rightarrow E_{4,0}]$.

Productions stored as a result of this production template would consist of a context with just one event, from sequence 2, and a prediction event, from sequence 4. The prediction event would have occurred one time step after the context event, e.g. if the events had progressed as detailed in Figure 6.2 then on step two $E_{4,0}$ = '**look** square 2' and $E_{2,-1}$ = '**hear** "move_2"' and so the following production would be stored:

$P_{1,1,1}$ ['**hear** "move_2"' → '**look** square 2'].

P_{ijk} refers to the jth production stored from template i. The production consists of the jth context stored in the cluster of which template i is a part (see the following section for a definition of clusters) and the kth prediction associated with that context. In this case the

production is stored as a result of template 2, '**hear** "move 2"' is the first context stored in its cluster, and '**look** square 2' is the first prediction associated with that context.

6.2.2.3 Clusters

The templates are arranged into groups called clusters. The templates in each cluster are designed so that, of all the contexts stored as a result of the templates, no more than one context can be active at a time. A context is active if all of its events are in the correct places in the sequences in STM, as prescribed by the template from which it originated.

Section 6.3 describes how connections are formed between productions that occur consecutively. These connections can only be formed between productions in the same cluster. Templates are grouped into clusters so as to reduce the number of connections between productions. This can eliminate many unnecessary connections thereby reducing the amount of computation required in the leakback process (section 6.3).

The PP system described in Section 6.6.2 has three production templates grouped into two clusters.

6.2.2.4 Productions

A production is stored whenever all of the events prescribed by a production template have non-"null" values. In the example above, the production $P_{1,1,1}$, was stored at step two on that step because both events, $E_{2,-1}$ and $E_{4,0}$, specified by template, T_1, were non-"null". Whereas on step three no production is stored because the specified events are both "null" (although only one of the specified events needs to be null for there to be no production stored).

A context can have more than one prediction event associated with it. The context and prediction(s) forming the production(s) are stored in LTM.

6.2.2.5 Predictions

A context in LTM is said to be active for prediction if, in the following step all its events will be in the places in the sequences prescribed by the template from which it originated. i.e. context i which originated from template k is currently active if: If context i is active then, in the following step, all of its events will be in the places prescribed by template k. A context active for prediction is able to predict an event to be used in the following step. The prediction event can be used as either:

1) an output, if it is an output event, or

2) a prediction of the next input (used for planning), if it is an input event[15].

If, in the example above, the robot hears "move_2", then the current event in sequence two will be $E_{2,0}$ = '**hear** "move_2"'. This will cause the context of $P_{1,1}$ to become active for prediction because in the following step $E_{2,-1}$ will be '**hear** "move_2"', i.e. in the following step the event "move_2" will be in the same time step (relative to the current step) it was when the production was first stored. The prediction event of $P_{1,1}$ is an output event and so it will be used as an output in the following step, that is, in the following step the robot will do the action '**look** square 2'. If there was more than one prediction associated with the context then leakback would be used to determine which prediction would be chosen. Section 4.3 describes the operation of leakback.

6.3 Leakback learning

Often one context will have two or more different prediction events, for example the context '**see** square 1' could be associated with the prediction events '**tilt** east' and '**tilt** west'. All stored prediction events will be possible predictions and when the context occurs PP must choose which of these events to predict. To do this PP uses what is known as leakback and a network of probabilities to choose prediction events most likely to reach goals. Leakback estimates the expectation that a particular event will lead to a goal. The event with the highest expectation is chosen as the prediction. This is the way actions (output events) are chosen which will lead to goals. Goals are marked contexts; Section 4.5 discusses how contexts are marked as goals.

---

[15]See Andreae & MacDonald (1991) for more information on planning

Assigning numbers, i to each context which has occurred in cluster l, and j to each prediction associated with context i, the expectation of the jth prediction of context i, X_{ij} is calculated using the following expression:

$$X_{ij}=DF \times \sum_{k=1}^{Nc_l} p_{ijk} X_k \tag{79}$$

where Nc_l is the number of contexts that have been stored in cluster *l*, X_{ki} is the expectation of context k:

$$X_k=\begin{cases} 1 & \text{if context k is a goal} \\ \underset{j=1..NC_l}{Max}(X_{kj}) & \text{otherwise} \end{cases} \tag{80}$$

and p_{ijk} is the probability that context k will be active next, given that the events forming context i and prediction j have just occurred. This probability is estimated by calculating the ratio of

- the number of times context k has become active immediately after the production consisting of context i and prediction j have occurred, N_{ijk}, and
- the number of times the production of context i and prediction j has occurred,

$$\sum_{k=1}^{Nc_l} N_{ijk} \tag{81}$$

$$p_{ijk}=\frac{N_{ijk}}{\sum_{k=1}^{Nc_l} N_{ijk}} \tag{82}$$

N_{ijk} is stored as a weight associated with a connection from the production consisting of context i and prediction j to context k. Most N_{ijk}, and hence p_{ijk} will be zero because only a few of the Nc_l contexts will have ever been active after the events of any context i and prediction j have occurred. There are no connections between contexts in different clusters

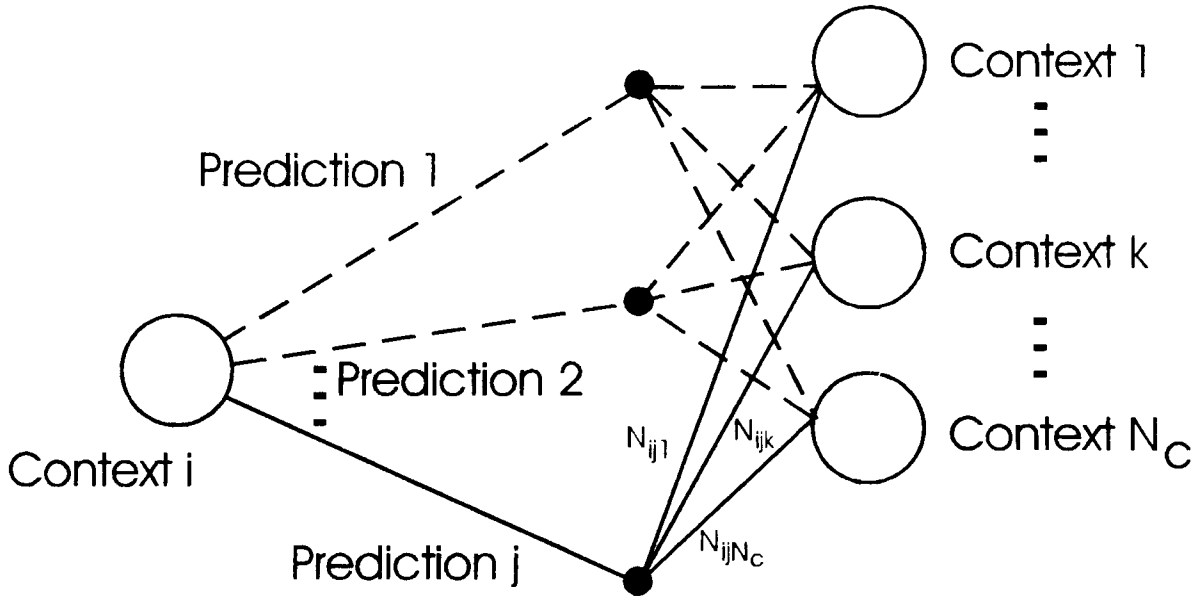


Figure 6.3 A schematic of the connections made between context i and all the other contexts in the cluster.

(see 6.2.2.3). The updating of these weights is what was referred to as *leakback learning*.

The discount factor (DF) in (79) is a number between 0 and 1. This factor gives more weight to actions which have shorter paths to a goal.

6.3.1 An example of leakback learning

The example is taken from the PP system described in section 4.6. The production: $P_{1,4}$ ['**see** 6' → '**move** north'] can be followed by productions with several different contexts, e.g. '**see** square 2', '**see** square 5', or '**see** square 7'. At one particular stage of the learning the production $P_{1,4}$ has occurred 429 times and of those 429 occurrences the context '**see** square 2' has been active in the following step 354 times. PP calculates (using (82)) the probability of the context '**see** 2' being active after the production $P_{1,4}$ has occurred to be 354/429 = 0.825. Figure 6.4

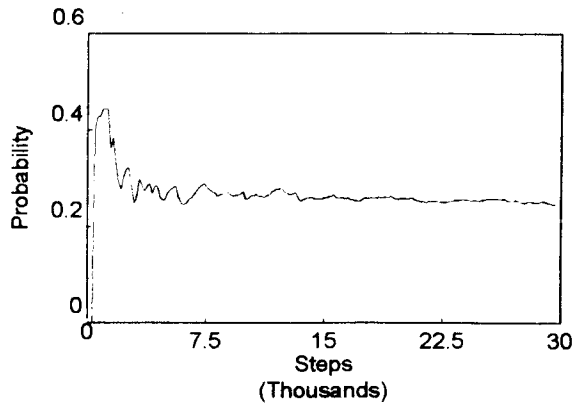


Figure 6.4 Graph showing the changing probability that a North tilt action will result in a movement from square 6 to square 2.

shows how this particular probability changes over time.

6.4 Performing actions for the first time

The PP system performs actions which are the prediction events of productions stored in LTM. The PP system begins with no stored productions and so the question arises, how can PP perform an action that is not stored in a production? There are a number of ways in which actions can be performed initially, before they are stored as prediction events in productions:

- a teacher can perform actions for the PP system,
- the actions can be performed by built in reflexes,
- PP can mimic the actions of others,
- PP can perform random actions for a period of time.

In order for a teacher to perform actions for the PP system, the teacher must either be able to send the signals to the robot's effectors, or physically lead the robot's limbs. PP can easily learn actions if the teacher generates the effector signals, although the human brain could not learn to perform actions in this fashion because no teacher can send signals through another human's nervous system. In order to learn actions when being lead, PP must be able to calculate what effector signals would generate the movement it senses [MacDonald, 1984]. Both these methods are useful for teaching PP to perform tasks and has been used often [Andreae 1977; Andreae & MacDonald 1993].

MacDonald (1984) showed that PP can learn from actions which are initially performed as reflexes. This is important because human babies are born with many reflexes and could possibly learn to move from them in a similar manner. Section 6.6.3.3 discusses how reflexes are used to perform actions in the demonstration PP system.

Mimicking in PP has only been implemented as a type of reflex which was used in learning to say words [Andreae 1977]. In order to be able to mimic the actions of others the PP system must know or learn the relationship between the stimuli (e.g. what is heard) and the command signals which perform the action (e.g. the signals sent to the speech effectors).

Section 6.6 shows an example of a PP system which learns from random actions. In this example a random action is selected if there is no production predicting an action or if there is a production but the number of times its context has occurred is below a certain amount. This technique for initially performing actions does not require a teacher to know anything about how the robot body operates and so could possibly occur in human brains.

There is some evidence that random-like (chaotic) behaviour could occur in the brain in unfamiliar situations. One of the conclusions of Skarda & Freeman's (1987) study of the olfactory system of rats was that an unfamiliar odour would produce chaotic activity in the olfactory neurons, whereas a familiar odour would produce a more ordered activity.

6.5 Marking contexts as goals

The network of probabilities and the expectation calculations allow PP to choose actions that will lead towards goal contexts. Goals are contexts which are marked as a goal and given an expectation of 1. There have been three different ways of marking contexts as goals:

- novelty learning [Andreae 1977],
- external reward [Andreae 1977], and
- internal reward or self set goals.

With novelty learning PP marks each new context as a goal. The goal mark is removed when the context occurs for a second time. This encourages PP towards new experiences and it has been found that this is very useful when teaching PP to perform new tasks [Andreae, 1977].

Goals can be given to PP using external reward [Andreae 1977]. This typically consists of a button which when pushed will cause all the contexts currently active to be marked as goals.

A new, third way of contexts being marked as goals is through internal reward. This comprises of a command which marks all the contexts currently active as goals. The command is called the GO command. The GO command will strictly be classed as an output event which will not influence the environment through an effector but will influence the state

of the PP system. An example of internal reward being used is given in 6.6.

This new way of marking goals is introduced to allow a robot to be controlled through verbal commands rather than through an external reward mechanism. When the robot is learning a task it is still necessary to have an external reward mechanism to set goals. But rather than the external reward marking contexts as goals directly it will trigger the GO command which will then mark the goals. If the GO command is the prediction event in a template then this means that at some latter date PP can perform the GO command itself and thus set its own goals.

In the example in section 6.6 there is a template that associates an audio stimulus with the GO command. This means that after learning has occurred PP is able to set its own goals in response to an audio stimulus, i.e. in response to a verbal command.

A context which is marked as a goal will have the mark removed when the goal is reached, i.e. when the context becomes active (the events in STM match those stored in the context).

6.6 An example of the PP system

This demonstration shows how PP can select actions to achieve goals. It also shows how actions can be learnt from random actions and from reflexes, and how contexts can be marked as goals through internal reward. The task used is that of controlling a ball rolling in a maze by tilting the maze. This task is also used in the following Chapter where a hybrid PP-CMAC system is able to provide superior control.

This section proceeds by describing the maze which PP learns to control (6.6.1), the configuration of the PP system (6.6.2), and then a summary of how PP learnt and the results of that learning (6.6.3).

6.6.1 The maze

The tilting maze is shown in Figure 6.5. The maze is divided into 12 squares with walls

between some of the squares. The maze can be tipped along two axes of rotation. The angles of tilt are given by two variables, θ_x and θ_y. The PP system must learn to control the tilting of the maze so as to move the ball from wherever its current position is to some goal square. The ball and maze are simulated. The details of the simulation are given in appendix A.4.

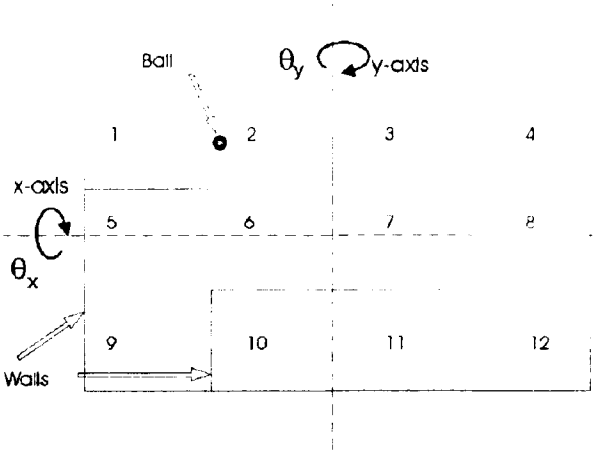


Figure 6.5 A simple maze, divided into 12 squares. The two axes of rotation are shown.

6.6.2 The configuration of the PP system

Figure 6.6 shows the channels of communication between the PP system, maze, eye, ear, command processor, visual processor and the teacher. For this exercise PP has five sequences of events, two input and three output:

Input sequences

Sequence 1: A sequence of visual inputs from an eye, which give the square which the ball is in, from square 1 to square 12.

Sequence 2: A sequence of audio inputs from an ear. These inputs can be anything that is said to PP, e.g. "move_1". Audio inputs are always written with double quotes.

Output sequences

Sequence 3: A sequence of tilt actions, North, South, East or West. These actions tilt the maze more towards the North, South East or West if the ball is not already moving in that direction. The action is translated into θ_x and θ_y increments (± 0.05 radians) by the command processor (Figure 6.6). The command processor uses a direction of velocity signal from a visual processor to decide if the ball is already travelling in the direction specified by the tilt action.

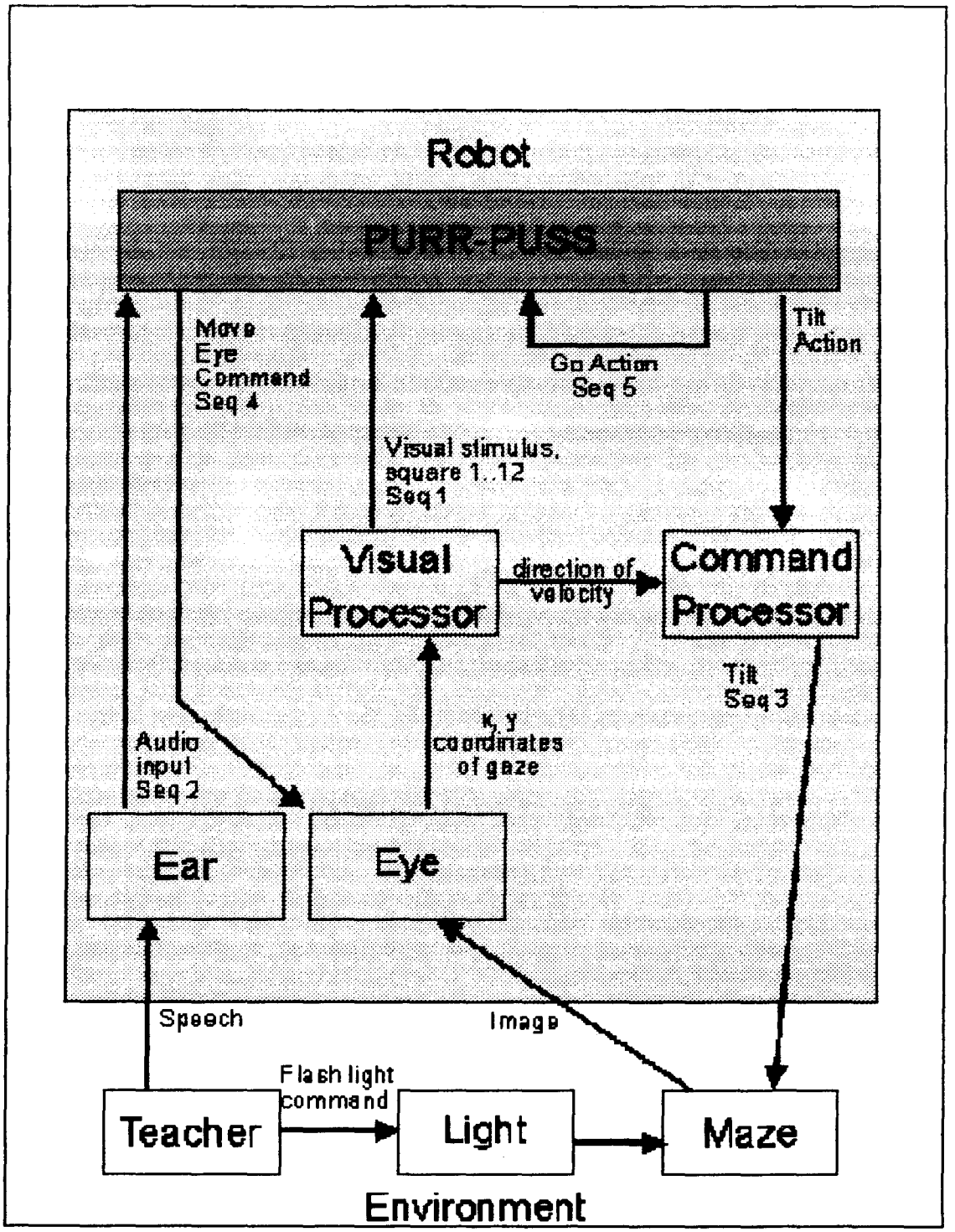


Figure 6.6 The channels of communication between the teacher, the maze, and the robot.

Sequence 4: A sequence of eye movement actions, 1..12. These actions make the eye glance briefly at the given square and then back to its original position.

Sequence 5: A sequence of GO actions. This is an internal reward command which does not have a direct effect on the outside body. This command marks, as goals, any contexts that are active in cluster 1 in the following step.

The output from the visual processor (Figure 6.6) is a representation of the square (1..12) in which the gaze of the eye lies. The eye operates almost entirely by reflex. If there is no eye movement command (Sequence 4) then the eye follows the ball by reflex and so the visual processor gives the position of the ball in the maze. There is another reflex which is triggered by a flashing light. This reflex causes an eye movement command (Sequence 4) to glance at the square where the light flashed and then move back to the ball. The flashing light also causes a reflex GO command (Sequence 5) which is executed in the step immediately after the eye movement command.

The actions of moving the eye to the flashing light and performing the GO action are examples of actions which will be learnt from reflexes. Initially these actions are performed in response to a stimulus (the flashing light) but after learning, the actions will be performed because they are prediction events of productions in LTM.

PP is configured with three templates, arranged in two clusters:

$$\begin{array}{c} \textit{Cluster } 1 \\ T_1{:}[E_{1,0} \rightarrow E_{3,0}] \\ \\ \textit{Cluster } 2 \\ T_2{:}[E_{2,-1} \rightarrow E_{4,0}] \\ T_3{:}[E_{2,-2},\ E_{4,-1} \rightarrow E_{5,0}] \end{array} \qquad (83)$$

Productions stored in the first cluster allow PP to use leakback to calculate which actions are most likely to lead to goals. Template, T_1 means that the current visual stimulus is used to predict the next tilt action. This template would apply, for example, when the ball is in square 6, the visual input is '**see** square 6', and a '**tilt** north' action is performed resulting in the following production being stored:

$P_{1,2}$: ['**see** square 6' '**tilt** north'].

The next time the visual stimulus is '**see** square 6', PP will predict a '**tilt** north' action.

Productions stored in the second cluster allow PP to respond to verbal commands from the teacher and set goals in the first cluster (see 6.6.3.1). T_2 predicts an eye movement action in response to an audio stimulus, e.g. $P_{2,1}$: ['**hear** "move_2"' → '**look** square 2']. T_3 predicts a GO action in response to an audio stimulus followed by an eye action, e.g. $P_{3,1}$: ['**hear** "move_2"', '**look** square 2' → 'GO']

As the PP system is configured here, it reacts whenever its inputs or outputs change or after a certain time (100 steps) if there has been no change. When the PP system reacts it updates its short term memory. This may cause some previously stored contexts to become active and allow some actions to be chosen as described in section 4.3. Updating STM may also cause some new productions to be stored and the weights between contexts to be updated.

Every 10ms the ball position, velocity and acceleration are updated using equations (116)-(118) (in A.4) and the command processor updates θ_x and θ_y if necessary according to the current PP tilt command.

The PP system is configured so that at first its tilting actions are random but after a time they become goal directed. If PP is unable to choose an action, or an action is chosen using a context with an age less than 50 then a random action is used. The age of a context is the number of times it has occurred. The number 50 was chosen so that there was enough time for the probabilities to be estimated roughly.

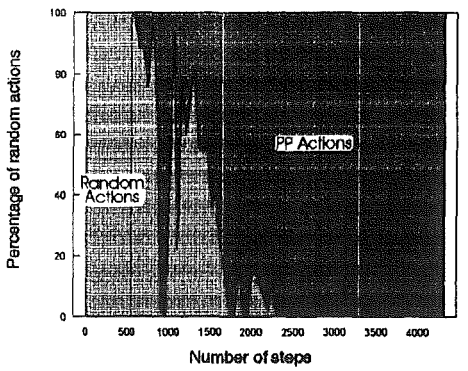


Figure 6.7 Graph showing the transition from random to goal directed actions.

6.6.3 Learning in the PP system

With the PP system configured as above, PP begins choosing random actions and learning. PP goes through two phases which are outlined below: choosing actions randomly (Figure 6.8)

and using productions to choose goal directed actions (6.6.3.2). The change from random to goal directed actions is gradual. The actions are random until the contexts have an age of 50 or more. As more contexts age over 50, the actions become more goal directed.

Time Step	Step Length	tilt	see	GO	look	hear	Flash	Production No	Production
1	1	east	1			"move_2"	F	P1,1,1	[1 >> east]
2	1	south	1		2			P1,1,1	[1* >> south] ["move_2"* >> 2*]
3	1	east	2	GO				P1,1,2	[2* >> east*] ["move_2" 2 >> GO]
4	1	west	1					P1,1,1	[1 >> west]
5	100	east	1					P1,1,1	
6	1	east	2			"move_3"	F	P1,1,2	[2 >> east]
7	1	north	2		3			P1,1,2	[2 >> north] ["move_3" >> 3]
8	1	west	3	GO				P1,1,3	[3 >> west] ["move_3" 3 >> GO]
9	1	north	2					P1,1,2	[2 >> north]
10	40	west	1					P1,1,1	
11	100	west	1					P1,1,1	
12	100	north	1					P1,1,1	[1 >> north]
13	100	east	1					P1,1,1	
14	30	north	2					P1,1,2	
15	100	east	2					P1,1,2	
16	100	west	2					P1,1,2	[2 >> west]
17	30	east	1					P1,1,1	
18	13	east	2					P1,1,2	
19	100	east	2					P1,1,2	
20	100	east	2					P1,1,2	
21	100	east	2					P1,1,2	
22	100	north	2					P1,1,2	
23	100	west	2					P1,1,2	
24	30	west	1					P1,1,1	
25	100	south	1					P1,1,1	
26	100	west	1					P1,1,1	
27	100	west	1					P1,1,1	
28	100	east	1					P1,1,1	
29	30	east	2					P1,1,2	
30	2	north	6					P1,1,4	[6 >> north]

Figure 6.8 The first 30 steps of PP's learning during the demonstration.

When PP is learning, the teacher stipulates the goal square by flashing the light in the square where s/he wants PP to move the ball. At the same time the teacher says the number of the square where the light is flashed. In the future the teacher will only need to say the number of the square rather than flash the light to get PP to move the ball there. The mechanism PP uses to achieve this responsiveness to verbal commands is described in section 6.6.3.3.

6.6.3.1 Learning from random actions

While PP is choosing tilt actions randomly it is storing productions and updating connections between contexts in each cluster. The connections in cluster 1 show the probable consequence of each tilt action in every context. For example, after a short time PP has learnt that after a '**tilt** north' action, when the visual stimulus is '**see** square 6', there is an 82.5% chance the next visual stimulus will be '**see** square 2' (as shown in figure 4.6), a 7% chance it will be '**see**

square 6' a 0.9% chance it will be '**see** square 7', etc. There are a number of reasons why the '**tilt** north' action does not always result in a transition from square 6 to square 2 such as, if there is a large velocity component in the east or west directions then the ball could move to square 5 or square 7 respectively or if the velocity is low then the ball will remain in square 6 for longer than 100 steps. As time goes by these probabilities converge to stable values. Figure 6.4 shows how one of the probabilities stabilises over time.

6.6.3.2 Choosing goal directed actions.

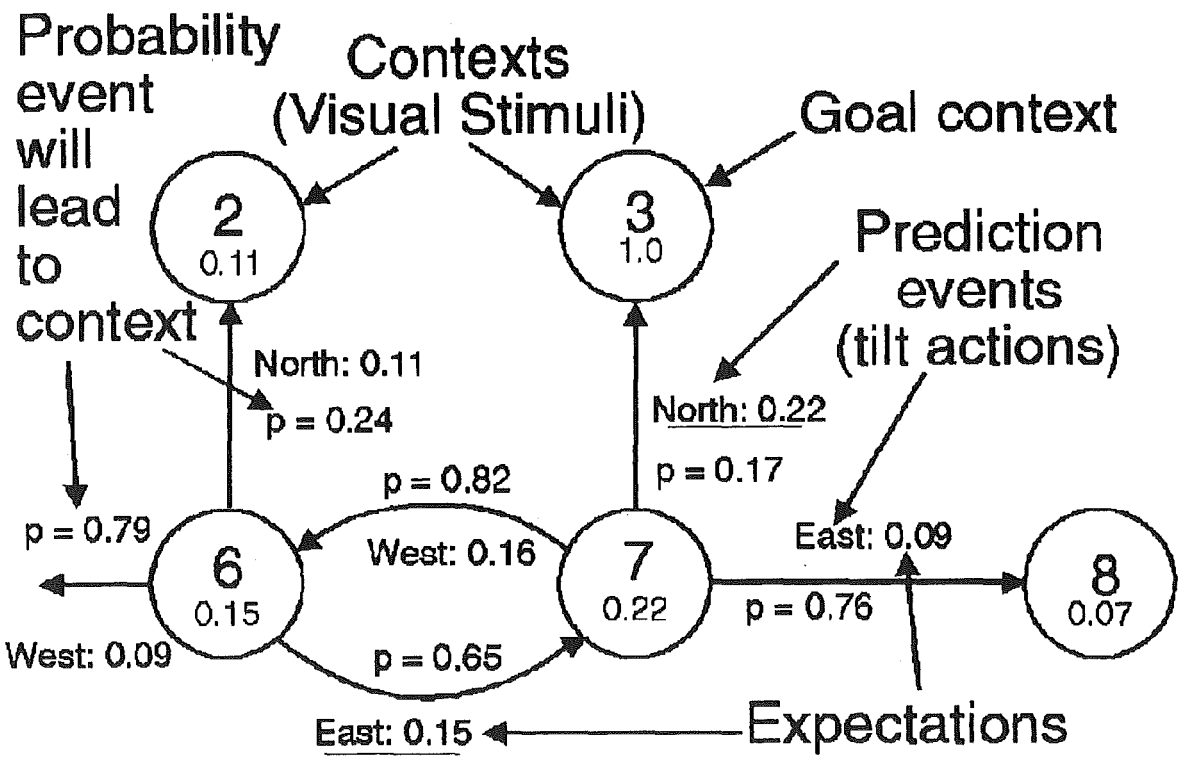


Figure 6.9 Some connections between contexts in cluster 1, showing transition probabilities, p_{ijk} and expectations, X_{ij}, that different actions will lead to a goal.

Figure 6.9 demonstrates how leakback is used in cluster 1 to choose tilt actions which lead to goals. PP's other actions, move eye (sequence 4) and GO (sequence 5), are chosen using templates from cluster 2, as described in the following section. At any time there are 4 possible tilt actions to choose from. In square 6 the east action has the highest expectation of leading to the goal context, square 3, and so the west action is chosen. Similarly the north action is chosen is square 7. The expectations shown in Figure 6.9 were calculated using (80) and the probabilities were calculated using (82). Appendix A.6 shows the full PP memory

after a period of learning.

6.6.3.3 Learning from reflexes to respond to verbal commands

The move eye and GO reflexes and the templates in the second cluster (see (83)) were designed to allow PP to learn to respond to verbal commands. Productions stored from template T_2 associate a verbal command with a move eye action. Productions from template T_3 associate the same verbal command and eye action with performing the GO command. The eye action and GO command are originally performed as a reflex to a flashing light. When a light flashes two actions are performed by reflex in the following steps:

- In the step immediately following the flashing light a move eye action will be performed by reflex. The eye moves to where the light flashed as described in Section 6.6.2 above. For example, if in Step 1 the light was flashed in square 3, then in step 2 the action '**move** eye 3' would be performed by reflex.
- In the next step the GO action is performed by reflex. In the previous example the GO action would be performed on step 3. The GO command sets the current context in cluster 1 as a goal context. In this case the context 'see square 3' would be marked as a goal.

The teacher can take advantage of these reflexes to train the system to respond to commands to move the ball to any square. To do this the teacher has to, for each square, say the command to move to that square and flash the light in the square. It is necessary to do this once for each square that is going to be a goal square. After that the teacher only has to repeat the command and the productions in cluster 2 will respond, causing the eye to move to the desired square, the GO command to be executed, and the goal square to be set.

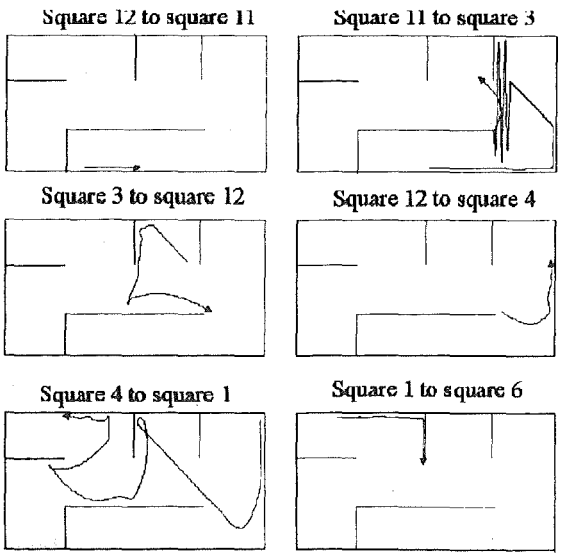


Figure 6.10 The path of the ball is shown for a sequence of goals, beginning in square 10 the goals are squares 11, 3, 13, 4, 1 and 6.

It is important to note that the verbal commands issued by the teacher and heard by PP have no meaning themselves to PP. The only meaning given to the words is given by the context in which they occurred.

When the eye moves to the goal square the visual stimulus will indicate that square and so the active context in cluster 1 will be the one consisting of the goal square stimulus. When the GO command is executed it will cause the active context in cluster 1 to be marked as a goal which will cause PP to choose actions which should lead to the goal, and hence move the ball to the goal square.

For example if the teacher says "move_3" and flashes the light in square 3 then as a reflex to the light, the eye will move to square 3 and production $P_{2,2}$ will be stored according to template 2: $P_{2,2}$: ['**hear** "move_3" → '**move_eye** 3']. In the next step the visual stimulus will be '**see** square 3' and PP will do the GO command (also as a reflex response to the flashing light). On execution of the GO command production $P_{3,2}$ will be stored according to template 3:

$P_{3,2}$: ['**hear** "move 3"', '**move_eye** 3' → GO].

Meanwhile in cluster 1 the context ' **see** square 3' will be active if it exists and will be marked as a goal by the GO command. This will influence the leakback calculations (82) and PP should begin choosing actions which will lead to square 3 as outlined in section 6.6.3.2. Appendix A.5 shows PP's interaction with the world for the first 790 steps.

Using the tilt actions the PP system is able to learn to control the position of the ball in the maze but the control is not necessarily very good. The next chapter describes how a hybrid PP-CMAC system can control the ball position better.

Note: Another experiment was performed, in which there was no command processor. The purpose of this experiment was to show that the command processor was not crucial to the previous demonstration. PP is able to perform the function of the command processor. In the experiment PP's tilt actions were increment/decrement θ_x/θ_y, and there were some inputs representing the direction of the velocity. In the experiment with no command processor more contexts were stored in cluster 1 (~80 compared to 12). The simpler system with the

command processor and fewer productions was chosen to be presented here so that the operation of the system could be followed more easily.

6.7 Summary

The PP system is able to learn through its interaction with the world the consequences of its actions. The learning occurs by storing productions according to a predetermined set of production templates and by updating connections between consecutively occurring contexts. Through leakback PP can choose events which have the best chance of leading to a goal. A working PP system has been presented which demonstrated:

- how PP can learn to control the movements of a ball rolling in a tilting maze,
- how contexts can be marked as goals using an internal reward command, and
- how PP can learn from random actions.

CHAPTER 7

A hybrid PP-CMAC system

7.1 Introduction

This chapter demonstrates how the two learning systems PP and CMAC can be combined and how together the two systems can learn to effectively control smooth movement.

There is evidence (Gowitzke & Milner, 1980; Bloedel, 1992) that one of the functions of the cerebellum is to control smooth movement following discrete instructions from the cerebral cortex. Albus proposed CMAC as a model of the cerebellum and it has been used to produce smooth movement in control systems (Millar, Glanz & Kraft, 1988). The previous chapter described the PP system which is able to learn from its environment and produce plans which lead to goals. When executing these plans PP produces a series of discrete actions. The proposed PP-CMAC system is, perhaps, analogous to the cerebral cortex-cerebellum system in the brain, with PP providing discrete actions from plans and CMAC using those actions to produce smooth movements.

The PP-CMAC system is introduced through an example in section 7.2. In that section the system is configured to learn to control a ball in a tilting maze (a control problem that was introduced in the last chapter). That section describes the system and then analyses the learning that takes place in a specific experiment. The features of the combined system are summarized in section 7.2.3. The PP-CMAC system is compared to other forms of motor control in section 7.3.

7.2 A demonstration of the hierarchical PP-CMAC system

To demonstrate the design of this hierarchical system a simple sequential problem which needs smooth control was required. Ideally, a model would have been constructed of some human bodily part such as an arm or the speech organ. For simplicity, the task used is that

of controlling a ball rolling in a maze. The movement of the ball is controlled by tilting the maze. The maze and ball are the same as that which PP learned to control in Chapter 6. The dynamics of the model are described in appendix A.4. Although PP learned to control the ball in the maze in chapter 6, it did not control it very well, as can be seen in Figure 6.10. PP-CMAC learns to control the ball in the tilting maze better than PP did by itself. When learning is complete in PP-CMAC, the control of the ball is smoother and the ball moves more directly to the goal square from its current position.

For a robot to move an arm smoothly to a new position, the dynamics of the arm, constraints imposed by the joints, and loading of the arm have to be taken into account. Similar constraints must be taken into account when using speech organs. The ball and maze are not a good model of an arm, the speech organs, or any bodily function but they do present the PP-CMAC combination with a dynamic situation, requiring complex control and having constraints (the ball bounces off the walls). Controlling the position of the ball is fairly complex because the position of the ball at time $t+\tau$ depends not only on the control signals at time t but also on the position and velocity of the ball at time t.

The following sections describe the configuration of the system (7.2.1) and what is happening during the learning process (7.2.2).

7.2.1 The configuration of the PP-CMAC system

Figure 7.1 shows the interactions between a human teacher, PP, CMAC and the maze. Figure 7.1 is similar to Figure 6.6 which shows how PP is set up to control the maze by itself. The major difference is that the CMAC system is in Figure 7.1.

CMAC receives inputs from the eye and from the visual processor. The inputs from the eye give the x and y coordinates of the gaze. The input from the visual processor is a number, 1..12, indicating the square that the eye was looking at when the GO action was last performed by PP. It will be seen later (in section 7.2.1.2) that the input from the visual processor to CMAC is in fact the goal square. The GO action and the setting of goals is discussed further below. The output of CMAC is the two components of the tilt, θ_x and θ_y.

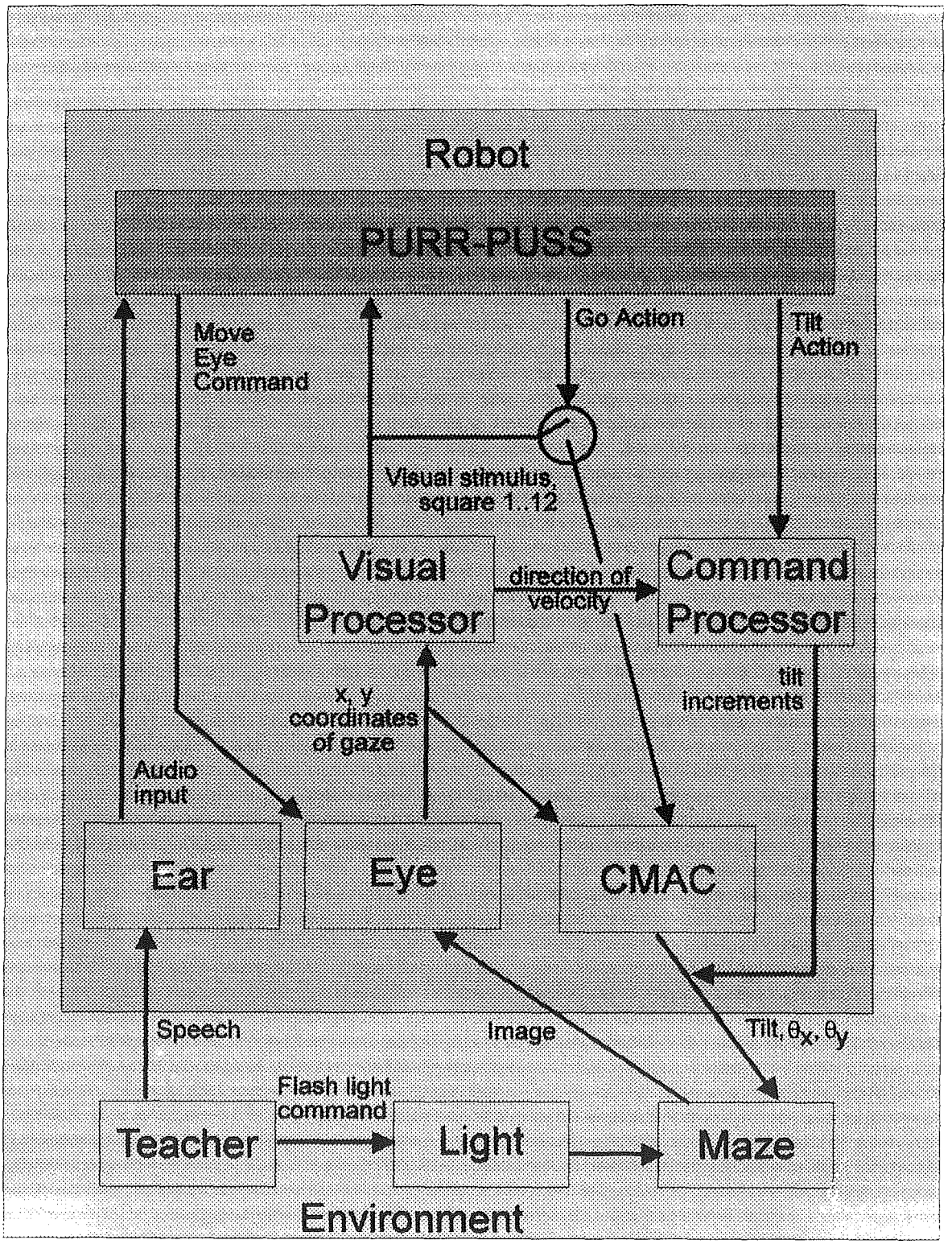


Figure 7.1 An outline of the PP-CMAC system

The tilt from CMAC forms one input to the maze, the other is from the light which flashes in a square. The configuration of the CMAC system is discussed in section 7.2.1.1.

PP is configured in the same way it was in Chapter 6, with two types of stimulus (audio and visual) and three types of action (tilt, move eye, and GO). All PP's actions perform the same function as they did previously, except the GO action has an additional function of routing goal information to the CMAC. The way this happens will not be obvious from Figure 7.1, but it is explained clearly in section 7.2.1.2 below. The GO action sets the goals for both PP and CMAC.

PP and CMAC communicate in two ways. The first (described in 7.2.1.3) is through the command processor which, by incrementing θ_x and θ_y, supplies training examples for CMAC. The second channel of communication is the GO command which routes the visual stimulus from the visual processor to the CMAC input.

7.2.1.1 CMAC configuration

For the experiment, CMAC had s=4800 location units (arranged in a 12x20x20 block - goal, x-coordinate, y coordinate). The access distances were chosen so that a set of 1x4x4=16 location units became active for any input. Note that the block of active units has a width of only 1 in the goal dimension which means there is no generalisation possible between different goals. This was chosen deliberately because goals that are neighbours should not necessarily be treated in a similar way (eg square 9 and square 10). A learning rate of α=0.6 was used. At the beginning of the experiment all the weights in CMAC are set to 0.

Like the PP system, CMAC reacts whenever its inputs or outputs change. If its inputs change, then new outputs are calculated using equation (58). If its outputs are changed by the command processor, then its weights are updated using equation (60). The command processor can increment or decrement the tilt in either the x- or y-direction. The learning that occurs in CMAC is discussed in more detail in section 7.2.2.

7.2.1.2 Routing goals to the CMAC - the GO action

In the experiment described in chapter 6 the GO action was first performed as a reflex to the

flashing light. After learning, PP was able to perform the GO action in response to a verbal command. The GO action set the goals in PP, and PP operates in exactly the same way here as it did in the experiment in that chapter.

In this experiment the GO action has an additional effect. When the GO action is executed, the current output of the visual processor (PP's visual stimulus) is routed to the input of CMAC and remains there until the next GO action is performed. In Chapter 6, when the GO action was performed the visual stimulus was the goal square. PP is operating in exactly the same way and so in this experiment the visual stimulus is also the goal square when the GO action is executed. This means that the GO action sends the visual stimulus of the goal square to CMAC. Section 7.2.2 describes how, after learning is complete, sending the goal visual stimulus to CMAC causes CMAC to move the ball to the goal square.

The GO action therefore sets goals, based on the visual stimulus, in CMAC and PP at the same time. This ensures that PP and CMAC are coordinated, that is they are each consistently trying to achieve the same goal. This allows the CMAC system to use a combination of PP's current corrective actions and knowledge of previous actions for a particular goal to produce output commands that will lead to that goal.

The GO command was inspired by its namesake in Bullock & Grossberg (1988), which has some similarities to the system presented here but also significant differences; these are discussed in 7.3.

7.2.1.3 Which part has control, PP or CMAC?

The tilt that is sent to the maze is the output of CMAC modified by the increments or decrements of the command processor. The command processor only produces these changes if the ball is not rolling in the direction indicated by PP's tilt action. In the beginning CMAC's weights are all zero and so it produces no output. The only effect on the output is due to the tilt changes from the command processor. So in the beginning PP has complete control.

As CMAC learns, it begins preempting PP's actions, causing the ball to roll in the desired

direction. In this case the command processor produces no increments in response to PP's actions. So PP's actions have less and less effect as the CMAC learns and in this way CMAC takes control gradually from PP.

7.2.1.4 The teacher

The teacher has to perform the same combination of light flashing and verbal commands, as in the previous chapter, to teach the system to respond to audio commands. For this demonstration the teacher is automated; audio commands are chosen to ensure that the ball travels every path between possible goals. There are 12*11 possible paths, ie from square 1 to squares 2..12; from square 2 to squares 1, 3..12; etc. When the ball moves into a goal square the automated teacher selects the least travelled path from that square and says the appropriate command indicating the new goal square. If there is a set of paths which have been travelled the least, then one of that set is chosen at random.

This procedure ensures that the system learns every possible movement that can be specified by the teacher. This is a more rigorous procedure than just choosing random goals.

7.2.2 Learning in the PP-CMAC system

The PP-CMAC system begins with little knowledge of the maze problem it is about to learn. The only knowledge in the system is within the structure and the built-in learning mechanisms of PP and CMAC. During learning the system goes through three stages:

- random PP actions,
- goal-directed PP actions, and
- goal-directed CMAC actions.

Initially learning progresses in the same way as it did in Chapter 4, with PP performing random actions and building up a network of transition probabilities. As the age of PP's contexts goes over 50, PP begins performing goal-directed actions using the network of probabilities and leakback (see Chapter 4 for a full description of PP's learning). At this stage CMAC begins learning.

CMAC will only learn if a PP action results in the command processor incrementing one of the control signals (θ_x or θ_y), i.e. if the ball does not already have a component of movement in the direction specified by PP's action. PP will only produce a new tilt action when the ball moves from one square to another (see Figure 7.2) because this is when the visual inputs to PP change. When a tilt action results in the command processor incrementing one of the control signals, the new control signal provides a new target for the CMAC network. The CMAC uses this target to adjust its weights using its learning algorithm, equation (60).

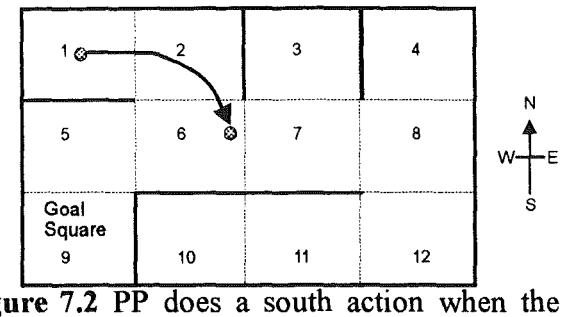


Figure 7.2 PP does a south action when the ball enters square 2. With no south component of movement the command processor tilts the maze south and CMAC learns.

For example, suppose that for a particular ball position and goal, CMAC's θ_x output is 0 radians (as it would be in the beginning, before any CMAC learning, when all the weights are 0) and that PP performs a South action. If the ball does not already have a component of motion in the south direction then the command processor will produce a tilt increment of 0.05 radians and in CMAC, the weights of the connections to the θ_x output will be updated, using equation (60). The error in the θ_x output (output 2) is E_2=0.5-0=0.5. Sixteen location units are active at any one time, so, in this case, the weights of the connections from the sixteen active units will change by $\Delta w=\alpha 0.5/16$. α is 0.6 so the weights change by 0.3/16.

This means the new CMAC θ_x output will be 0.3 radians. The next time CMAC's inputs are the same (ie the ball is in the same position and the goal is the same), then the θ_x output will again be 0.3. Note that, if the ball is in a nearby position, then many of the same location units will be active and so, the output will probably be close to 0.3. This means that as the ball approaches the same position the x-axis tilt will approach 0.3. If, as it is approaching this position, the ball's movement does not have a very large component in the north direction then this tilt will be large enough to start it moving in the south direction. If this is the case then, when PP performs the south action, the command processor will not increase the tilt and so no CMAC learning will take place. However, if the ball continues moving in the north direction then the command processor will increase the tilt. The weights of the connection to

the CMAC output will increase and so the next time the CMAC will tilt the maze even more.

As learning progresses PP's actions will start to be preempted more and more often by CMAC, e.g. by the time the ball moves into a square where PP does a South tilt action CMAC will have already tilted the maze to make the ball move in the south direction and so the command processor will not try to increment the tilt. As time goes by, PP's actions have less and less effect as CMAC takes control. With less effectual PP actions (*i.e. more actions which do not* cause the command processor to adjust the tilt) there are fewer updates of the CMAC weights, until eventually CMAC has learnt the whole problem and PP's actions never have an effect. This reduction in the number of CMAC weight updates for this experiment is shown in Figure 7.3.

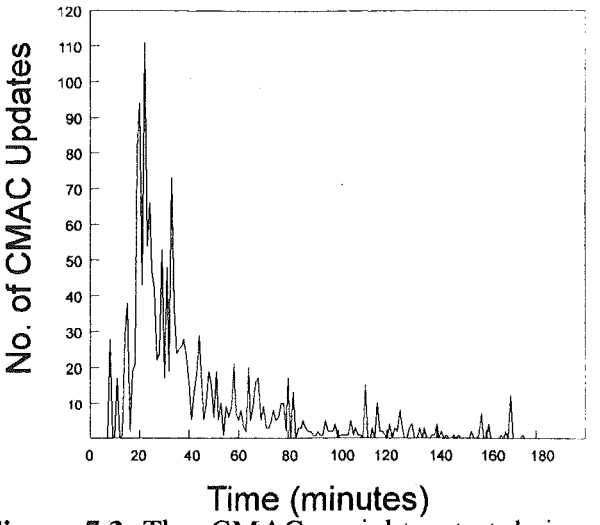


Figure 7.3 The CMAC weights start being updated as PP actions start becoming goal-directed. The number of updates per minute decreases as CMAC takes control.

When CMAC is learning, it is effectively constructing a different surface for each goal. These surfaces cause the ball to roll towards the goal which is at the lowest point on the surface. Figure 7.4 shows the surface which results when the goal is "square 9", the bottom left square of the maze[16]. This surface indicates the tilt of the maze at all ball positions in the maze. For example when the ball is in the middle of square 12 the inputs to the CMAC are (x=0.875, y=.0167 ,goal=9) and the tilt of the maze is given by the outputs (θ_x= 0.314, θ_y=0.192).

After CMAC has completed training, PP's actions cease to have any effect at all. This is because at any stage the output of the CMAC will cause the ball to roll with a component of its motion in the direction specified by PP's tilt action and so the command processor will be

[16] The surface is constructed by generating the angle controls for each point, θ_x, θ_y (giving the slope in two directions), and integrating with respect to ball position. The slopes between points with walls between them are ignored.

doing nothing.

7.2.2.1 The quality of control improves

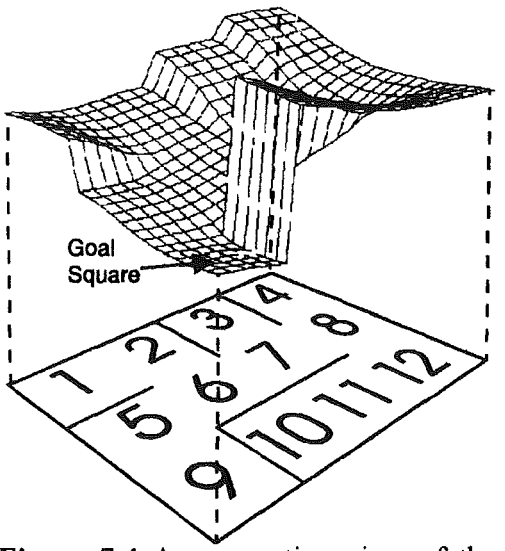


Figure 7.4 A perspective view of the surface obtained from the CMAC when the goal is the bottom left square (9). The surface shows the tilt of the maze as a function of the ball position over the whole of the maze

As the system moves through each of the stages outlined above, the quality of control improves. Initially when all the actions are random, the ball will only move to the goal square by chance. In this stage the system is finding out what the actions do and how to use them to solve the given problem. When the actions are goal-directed, under PP control, the ball moves to the goal square fairly reliably, although there can be problems with overshoot as is the case in Figure 7.5 (top). However, when CMAC is in control there are none of these problems, as is evident in Figure 7.5 (bottom). As the control transfers from PP to CMAC the system is improving with practice.

The reason for these improvements can be explained in terms of degrees of freedom (Bernstein, 1967). In this task there are two degrees of freedom, θ_x and θ_y. The PP system, through the command processor, can only adjust one degree of freedom at a time, but to skilfully perform this task the operator must be able to adjust both degrees of freedom at the same time. For example, in going through squares 6 & 7 to square 3, in square 7 the operator needs to adjust θ_x so there is an acceleration in the north direction and at the same time adjust θ_y so there is an acceleration in the west direction. The western acceleration reduces any velocity component in the east direction and ensures that the ball does not overshoot and go into square 8 rather than 3. With two outputs, the CMAC system is able to adjust both these degrees of freedom simultaneously.

This ability to adjust θ_x and θ_y simultaneously is shown graphically in Figure 7.4. For example when moving from square 11 to 7 the CMAC has to change the θ_y tilt from giving

an acceleration in the east direction to one which gives an acceleration in the west direction while at the same time adjusting θ_x to give an acceleration in the north direction (to move from square 12 to 8) followed by an adjustment to give an acceleration in the south direction (to avoid overshooting into squares 4).

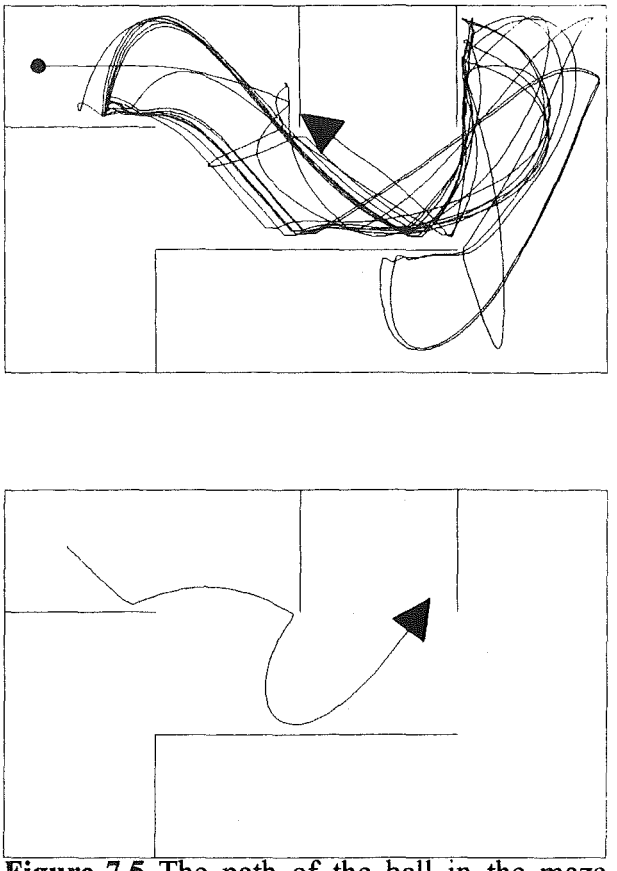


Figure 7.5 The path of the ball in the maze from square 1 to 3 under PP control (top) and CMAC control (bottom).

It has been observed (Turvey, Flitch & Tuller, 1982) that when people learn a skill they simplify the learning process by eliminating some degrees of freedom. As they become more skilled the number of degrees of freedom used increases. For example when a child learns to hit a baseball s/he stands quite rigid, only moving the arms when swinging the bat. This reduces the number of degrees of freedom being controlled, simplifying the problem and allowing learning to take place. As the child improves, he or she will include shoulder movements in the swing, increasing the number of degrees of freedom. A good batter will have movement in the hips, shoulders and wrists. The example given in the present paper shows that the PP-CMAC system can learn in the same way, adjusting only one degree of freedom at a time initially and gradually adjusting two degrees as the skill is learnt.

The system improving as it practises can be likened to a person learning to hit a tennis ball. In the beginning the person may use high level commands, given by a coach, such as to hold the racket in a certain way and swing. People are generally not very good at hitting tennis balls using these higher level commands, but with practice they improve and find they are doing it automatically. This automatic hitting of the ball is analogous to the stage when CMAC is controlling the system.

The concept of a goal being used as an input to a control system may explain the disruptive effect of a tennis player focusing on the stroke production. Instead of concentrating on the position where the ball is supposed to go, the player focuses on the movement of the tennis racket, so the control system is deprived of the goal information (where the ball is supposed to go) and the shot is not played well.

7.2.3 Summary of features of combined system

Two learning systems are combined, PP and CMAC. The PP system is capable of performing complex tasks in an unknown environment, and produces goal-directed movements. However the PP system is not suited to controlling smooth movements. The CMAC system is suited to controlling smooth movements. The systems communicate in two ways:

- The GO command routes goal information to the CMAC. In section 7.2 the information was routed from the sensors. The information could also come from a memory of the sensors (if the direct sensory information is not available).
- The CMAC system learns from PP's performing error-based, incremental actions. The actions are termed error-based because commands will be sent to the effectors only when there is an error (or difference) between the desired movement and the sensed movement (the visual processor senses which direction the ball is moving in.). This allows control to gradually transfer from PP to CMAC. The actions are incremental in that they adjust the output from CMAC by a small amount. This allows previous knowledge stored in CMAC to have an effect and for performance to improve with practice.

When the CMAC is fully trained, the sequence of commands which are sent to the effectors are accessed from memory using a combination of the goal and sensory information.

7.3 Comparisons with other systems.

7.3.1 Open-loop vs closed-loop control.

Human motor control has been characterized in two ways: open-loop models and closed-loop

models (Stelmach, 1982; Abernethy & Sparrow, 1992). These models take goal information and produce a sequence of motor commands for muscles.

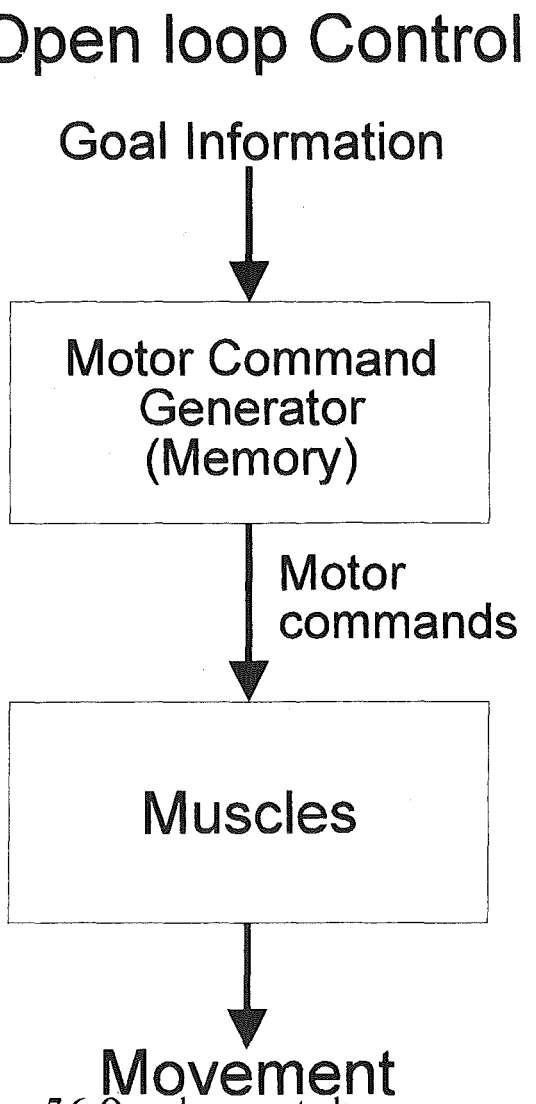


Figure 7.6 Open-loop control: a sequence of commands are retrieved from memory in response to a goal cue.

7.3.1.1 Open-loop control

In open-loop models, the sequence of commands sent to the muscles are retrieved from memory in a way somewhat similar to how a computer executes commands in a program (see Figure 7.6). In its purest form there are no means for correcting errors using feedback. There are problems concerning these models with how many variations of motor programs can be stored and how motor programs are generated for novel movements. These problems have been addressed by Schmidt (1982) who suggested that we store generalized motor programs, termed schema.

Some forms of open-loop control do make use of feedback. One example of this, shown in Figure 7.7, is Neilson, Neilson and O'Dwyer's (1992) Adaptive Model Theory (AMT). In this theory a desired response trajectory is converted to movement commands through an inverse model of the controlled system (the effectors). Feedback is used to update a forward model of the controlled system from which inverse parameters are extracted for the inverse model. In this model the feedback is used to adapt the controller to any changes in the effectors' responses.

7.3.1.2 Closed-loop control

Closed-loop models (e.g. Adams, 1971) use sensory feedback to produce an error between the desired movement and the detected movement (see Figure 7.8). The error is then converted to muscle commands which should reduce the error. Although this form of control gives very good accuracy, these models are undermined by their limited ability to explain control of rapid actions (Kelso & Stelmach, 1976) and the ability of both humans and animals to produce movement in the complete absence of sensory information (Taub, 1977).

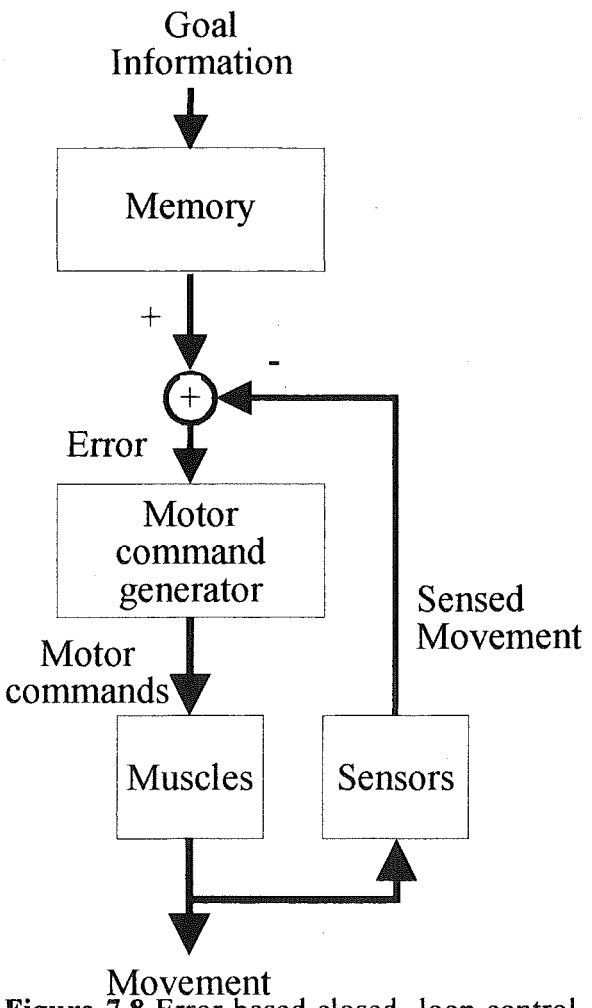


Figure 7.8 Error based closed -loop control.

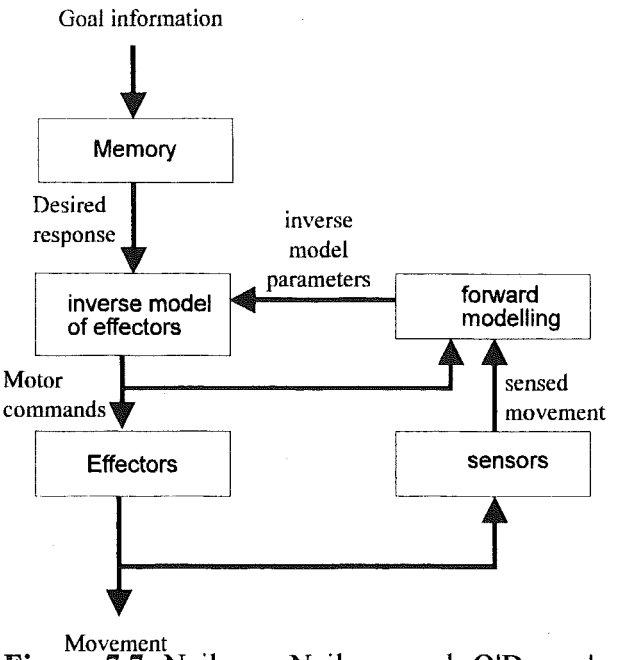


Figure 7.7 Neilson, Neilson and O'Dwyer's (1992) Adaptive Model Theory.

7.3.1.3 The PP-CMAC system

When the PP-CMAC system is under PP control, the motor control is closed-loop, because movement commands are only sent to the effectors if there is a difference (or error) between the desired movement and the sensed movement. In the example given in 7.2 the maze was tilted more only if the ball was not already moving in the desired direction. This is a very simple form of feedback control because the motor commands are not augmented by the size

of the error; nevertheless it is a form of closed-loop control.

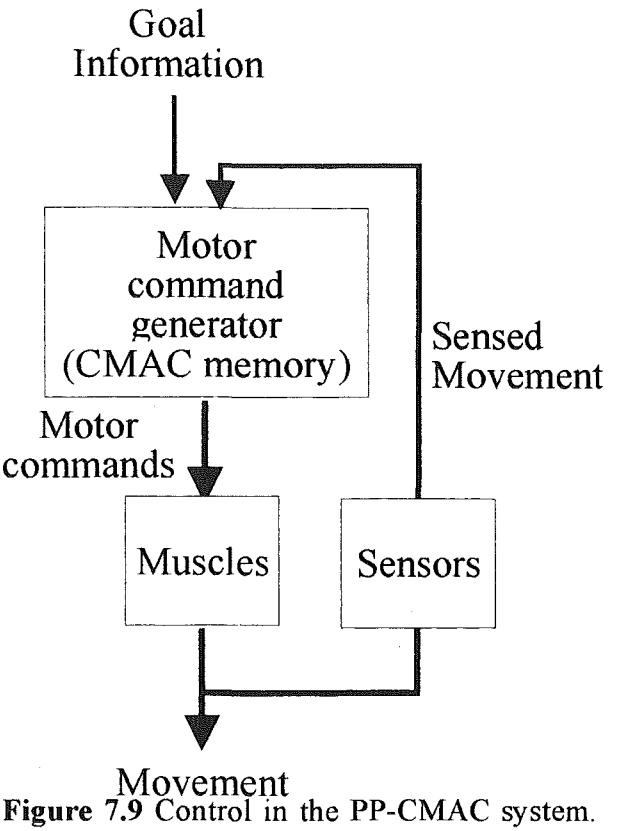


Figure 7.9 Control in the PP-CMAC system.

When the PP-CMAC system is under CMAC control, there is no error calculated at all (see Figure 7.9), which indicates that it is not using closed-loop control. However, the type of motor control is not open-loop because it depends not only on the input (the goal information) but also on the sensed output (the movement). It can be classed as a type of closed-loop control which does not calculate an error but uses a combination of goal and sensory information to generate movement commands from memory. The commands accessed from the memory are an accumulation and generalization of PP's error-based commands.

One problem with the closed-loop control used in the PP-CMAC system (Figure 7.9) is that, because it uses sensory information, it is subject to the same criticisms as other closed-loop systems outlined in section 7.3.1.2. In particular it would not be possible to produce movement if there was no sensory information. One possible solution, shown in Figure 7.9, would be to have another neural network which modeled the muscles, estimating their movement. The model would use the motor commands (which are sent to the muscles) as inputs and it would produce an estimation of the movement (from those commands) as an output. If the model was 100% accurate then the estimated movement would be the same as the sensed movement and so the input to the CMAC in Figure 7.9 would be the same as it is in Figure 7.9. The model would use the actual movement (detected by the sensors) to train itself.

If a model of the muscles was included in the system, as described above, then it would be possible to produce movement in exactly the same way as before but without sensory

feedback. The system would then be using open-loop control. No experiment has been done to test this theory.

In the open-loop control depicted in Figure 7.6 a sequence of motor commands are generated from memory in response to a cue representing the goal. In the closed-loop control depicted in Figure 7.8 the sequence of motor commands is generated from the error between the desired movement and the sensed movement, but a sequence of desired movements must be generated from a cue representing the goal. In the model of Neilson et al a sequence representing the desired response trajectory must be generated, again in response to a goal cue, from which the motor commands are then derived. Although it is possible to store sequences on neural networks (see Chapter 5) the PP-CMAC system avoids this problem because it does not require a memory to produce a sequence of commands or desired movements in response to a single cue. Instead the constantly changing sensory information is used to generate a sequence of motor commands.

The closed-loop control used in the PP-CMAC system also has an advantage over open-loop control in that the feedback provides verification that the movements were successful and the commands are adjusted in response to the feedback.

7.4 Extending the PP-CMAC combination - future work.

The simple task used in this demonstration did not require much of PP with only 36 contexts stored. The combined PP-CMAC system could learn more complicated tasks which the CMAC system would not be able to learn in full and so PP would have a larger role.

The primary role of the CMAC is to provide smooth control of the robot movements. This can be done by configuring the CMAC in the same way as it was in this experiment: with inputs representing the goal and current state of the body and outputs representing the signals necessary to control the body. We propose that the goal input will come from sensory information as directed by PP through the use of GO commands and the current state will be given by either actual or estimated stimuli.

It has been shown that this configuration is capable of learning to control the position of a ball in a tilting maze. This is analogous to controlling any body function, e.g. the movement of an arm. A control system for an arm could have proprioceptive and/or visual inputs, giving the position of the arm, and a visual input giving a goal. These are analogous to the x-y position coordinates and the visual goal respectively.

A complicated task using an arm will involve moving through a sequence of different positions. To do this using the control system outlined above it will be necessary to have a sequence of visual goals directed to the input. It will be the role of the PP system to coordinate the visual system to do this so that the goals arrive in the correct order and at the correct time. Similarly a complicated task involving the ball and maze could involve moving the ball through a series of positions in sequence. For example, move the ball from its current position to square 1 then to square 3. An example of PP performing complex tasks using a set of simple movements is given in (Andreae et al, 1993)

7.5 Summary

This chapter has shown how two learning systems, PP and CMAC, with different capabilities, can be combined. It has been shown that PP, which is able to discover how to use its actions to achieve goals, can train a CMAC network to perform smooth skilled movements necessary to complete a simple task. The CMAC is configured with inputs representing the goal and current state. We have demonstrated how the system improves with practice, initially controlling one degree of freedom at a time and finally controlling both. The quality of control of the PP-CMAC system is better than what PP achieves by itself.

CHAPTER 8

Conclusions

The conclusions drawn in this thesis can be grouped into two main areas:

- the sparse distributed memory, SDM
- the combined PURR-PUSS and CMAC system.

The work in Chapter 3 analyzed the capacity of the SDM. The work stemmed from an experiment which was performed to determine if the theoretical error rates, put forward by Kanerva (1988), were correct (section 3.3). This experiment showed that there was a discrepancy between the theoretical and actual rates. A modified theory was proposed which agreed better with the experimental results.

This first experiment led to further investigations into the capacity of the SDM. An alternative method of reading from the memory was proposed (section 3.3) which decreased the error rates. Section 3.4 described how the capacity could be improved by an order of magnitude by using an iterative algorithm to store information in the SDM. The iterative algorithm was derived from Gardner's (1988) iterative algorithm for the Hopfield network. It was confirmed experimentally that Gardner's (1988) analysis of the capacity of the Hopfield network could be applied to the SDM.

Section 3.6 outlines the differences between the SDM and the Hopfield memory. The capacity of the Hopfield network is proportional to the number of input units, N, whereas the capacity of the SDM is proportional to the number of location units, s. The number of location units in the SDM can be much larger than the number of input units in the Hopfield memory and so the capacity of the SDM can be much larger than that of the Hopfield memory.

Chapter 5 showed how the SDM could be used to store, retrieve and recognise sequences. A significant amount of work had been done which showed how the Hopfield network could

store sequences. Chapter 5 showed how the techniques used to store sequences in the Hopfield network could be applied to the SDM. The advantage of using the SDM rather than the Hopfield network is that the SDM has a larger capacity, as was pointed out in earlier chapters.

Chapter 4 describes the CMAC network which is very similar to the SDM. It is argued that the capacity limitations of the CMAC are the same as those of the SDM. The major difference between the two networks from an operational point of view is that it is easier to get the CMAC to run quickly on a computer. For this reason the CMAC network was used in the experiments in Chapter 7.

The second major section of work involves combining the PURR-PUSS system with CMAC to give a system which has the benefits of both. First, Chapter 6 describes the PURR-PUSS system. Two features which are new to the PURR-PUSS system are described in Chapter 6, learning from random actions, and the goal-setting command, GO.

Chapter 7 shows how the PURR-PUSS and CMAC systems can be combined to form a system which is able to learn how to perform skilled movement tasks proficiently. In the combined system PP discovers how to use its actions to achieve goals and it trains the CMAC network to perform skilled smooth movements. The system improves its skill at a task with practice.

References

Abernethy B. & Sparrow, W. A. (1992), "The rise and fall of dominant paradigms in motor behaviour", in J/ J. Summers (ed), *Approaches to the Study of Motor Control and Learning*, Amsterdam: North-Holland, 3-45.

Adams, J. A. (1971), "A closed-loop theory of motor learning", *Journal of Motor Behaviour*, **3**, 111-150.

Albus, J. S. (1975a), "A new approach to manipulator control: the Cerebellar Model Articulation Controller (CMAC)", ASME Transaction Series, *Journal of Dynamic Systems, Measurement, and Control*, **97**, 220-227, 1975.

Albus, J. S. (1975b), "Data storage in the Cerebellar Model Articulation Controller (CMAC)", ASME Transaction Series, *Journal of Dynamic Systems, Measurement, and Control*, **97**, 228-233.

Amit, D. J. & Gutfreund, H. (1985), "Spin-glass models of neural networks", *Physical Review A*, **32**, 2, 1007-1018.

Anderson, J. A. (1972), "A simple neural network generating an interactive memory", *Mathematical Biosciences*, **14**, 197-220.

Andreae, J. H. (1972-1991), *Man-Machine Studies Progress Reports UC-DSE/1-41*, Dept. of Electrical and Electronic Engineering, University of Canterbury, Christchurch, New Zealand.

Andreae, J. H. (1977), "Thinking with the teachable machine" Academic Press Inc. (London) Ltd.

Andreae, J. H. (1985), "The first moment", *Man-Machine Studies Progress Report UC-DSE/26*, 44-55, Dept. of Electrical and Electronic Engineering, University of Canterbury, Christchurch, New Zealand.

Andreae J. H. & MacDonald, B. A. (1991), "Expert control for a robot body", *Kybernetics*, **20**, 4, 28-54.

Andreae, J. H., Ryan, S. W., Tomlinson, M. L. & Andreae, P. M. (1993), "Structure from Associative Learning", *Int. Journal Man-Machine Studies*, **39**, 1031-1050.

Barro, S., Bugarin, A., Yáñez, A. (1991), "Systolic implementation of Hopfield networks of

arbitrary size", *Lecture notes in Computer Science 540, Artificial Neural Networks, International Workshop IWANN '91, Granada, Spain, September 17-19, 1991, Proceedings*, ed A. Prieto, Springer-Verlag, 268-276.

Bauer, K. & Krey, U. (1990), "On learning and recognition of temporal sequences of correlated patterns", *Zeitschrift für Physik B - Condensed Matter, 79, 461-474.*

Bauer, K. & Krey, U. (1991), "On the storage capacity for temporal pattern sequences of correlated patters", *Zeitschrift für Physik B - Condensed Matter,* **84**, 461-474.

Bernstein, N. (1967), *The Coordination and Regulation of Movement.* London: Pergamon Press.

Bloedel, J. R. (1992), "Functional heterogeneity with structural homgeneity: How does the cerebellum operate?", *Behavioral and Brain Sciences*, **15**, 4, 666-678.

Bressloff, P. C. & Taylor, J. G. (1991), "Temporal sequence storage capacity of time-summating neural networks", *Journal of Physics A: Mathematics & General*, **25**, 833-842.

Bullock, D. & Grossberg, S. (1988), "Neural dynamics of planned arm movements: Emergent invariants and speed accuracy properties during trajectory formation", *Psychological Review*, **95**, 1, 49-90.

Chou, P. A. (1989), "The capacity of the Kanerva memory", *IEEE Transactions on Information Theory*, **35**, 281-298.

Cottrell, G. W., Munroe, P. & Zipser, D. (1987), "Learning internal representations from gray-scale images:An example of extentional programming". In *Proceedings of the 9th Annual Conference of the Cognitive Science Society*, pp 461-473.

Freeman, W. J. (1987), "Simulation of chaotic EEG patterns with a dynamic model of the olfactory system", *Biological Cybernetics* **56**, 139-150.

Fukushima, K. (1987), "A neural network model for selective attention in visual pattern recognition and associative recall", *Applied Optics*, **26**, 23, 4985-4992.

Fukushima, K. (1988), "Neocognitron: A hierarchical neural network capable of visual pattern recognition", *Neural Networks*, **1**, 2, 119-130.

Fukushima, K. & Imagawa, T. (1993), "Recognition and segmentation of connected characters with selective attention", *Neural Networks*, **6**, 1, 33-41.

Fukushima, K. (1993), "Neural network for connected character recognition", *Proceedings of the first New Zealand two-stream conference on artificial neural networks and expert*

systems", ed N. K. Kasabov, IEEE Computer Society Press.

Gardner, E. (1988), "The space of interactions in neural network models", *Journal of Physics A: Mathematics and General*, **21**, 257-270.

Glover, D. E. (1988), "A hybrid optical fourier/electronic neurocomputer machine vision inspection system", *Proceedings of Vision 1988 Conference, sponsored by SME/MVA.*

Gorman, R. P. & Sejnowski, T. J. (1988), "Analysis of hidden units in a layered network trained to classify sonar targets", *Neural Networks*, **1**, 75-89.

Gowitzke, B. A. & Milner, M. (1980), *Understanding the Scientific Bases of Human Movement*, Williams & Wilkins.

Grossberg, S. "Adaptive pattern classification and universal recoding: I. Parallel development and coding of neural feature detectors", *Biological Cybernetics*, **23** 121-134.

Guyon, I. & Personnaz, L. (1988), "Storage and retrieval of complex sequences in neural networks", *Physical Review A*, **38**, 12, 6365-6372.

Herz, A., Sulzer, B., Kuhn, R. & J. L. van Hemmen (1988), "The Hebb rule: Storing static and dynamic objects in an associative neural network", *Europhysics Letters, 7*, 7, 663-669.

Herz, A., Sulzer, B., Kuhn, R. & J. L. van Hemmen (1989), "Hebbian learning reconsidered: representation of static and dynamic objects in associative neural nets", *Biological Cybernetics, **60***, 457-467.

Hopfield, J. J. (1982), "Neural networks and physical systems with emergent collective computational properties", *Proceedings of the National Academy of Sciences*, USA, **79**, 2554-2558.

Hopfield, J. J. (1984), "Neurons with graded response have collective computational properties like those of two-state neurons", *Proceedings of the National Academy of Sciences*, USA, **81**, 3088-3092.

Kanerva, P. (1988), *Sparse Distributed Memory*, MIT Press.

Keeler, J. D. (1988), "Comparison between Kanerva's SDM and Hopfield-type Neural Networks", *Cognitive Science* **12**, 229-239.

Kelso, J. A. S. & Stelmach, G. E. (1976), "Central and periphial mechanisms in motor control", in G. E. Stelmach (ed), *Motor Control: Issues and Trends*, 1-49, New York: Academic press.

Kleinfield, D. (1986), "Sequential state generation by model neural networks", *Proceedings*

of the Natural Academy of Sciences, USA, **83**, 9469-9473.

Knuth, D. E. (1969), *The Art of Computer Programming*, vol 2, Addison-Wesley.

Kuhn, R., van Hemman, J. L. & Reidel, U. (1989), "Complex temporal association in neural networks", *Journal of Physics A: Mathematics & General*, **22,** 3123-3135.

Lapedes, A. & Farber, R. (1987),, "Non-linear signal processing using neural networks: prediction and system modelling", Los Alamos National Laboratory report LA-UR-87-2662.

MacDonald, B. A. (1980), "Turing machine power for a multiple context learning system", *Man-Machine Studies Progress Report UC-DSE/16*, Dept. of Electrical and Electronic Engineering, University of Canterbury, Christchurch, New Zealand, 11-38.

MacDonald, B. A. (1984), "Designing teachable robots", Ph.D. Thesis, Department of Electrical and Electronic Engineering, University of Canterbury, Christchurch , New Zealand.

MacDonald, B. A. & Andreae, J. H. (1981), "The competence of a multiple context learning system", *Int. J. General Systems*, 7, 123-137.

Mato, G. & Parga, N. (1991), "Sequences in neural networks with temporal association", *Zeitschrift für Physik B - Condensed Matter*, **84**, 483-486.

McCulloch, W. S. & Pitts, W. H. (1943), "A logical calculus of the ideas immanent in neural nets, *Bulletin of Mathematical Biophysics*, **5**, 115-133.

Millar, W. T. III, Glanz, F. H. & Kraft, L. G. III, (1988) "Application of a general learning algorithm to the control of robotic manipulators", *The International Journal of Robotics Research*, **6**, 2, 84-98.

Minsky, M. L. & Papert, S. S. (1969), *Perceptrons*, MIT Press, Cambridge, MA.

Neilson, P. D., Neilson, M. D., & O'Dwyer, N. J. (1992), "Adaptive Model Theory: Application to disorders of motor control", in J. J. Summers (Ed) *Approaches to the Study of Motor Control and Learning*, 495-548, Elsevier Science Publishers.

Parker, D. B. (1985), "Learning logic". Report TR-47, Cambridge, MA: Massachusetts Institute of Technology, center for computational research in economics and management science.

Parks, P. C. Militzer, J. (1992), "A comparison of five algorithms for the training of CMAC memories for learning control systems", *Automatica*, **28**, 5, 1027-1035.

Prager, R. W. & Fallside, F. (1989), "The modified Kanerva model for automatic speech

recognition", *Computer Speech and Language*, **3**, 61-81.

Rosenblatt F. (1958), "The perceptron: A probabilistic model for information storage and organization in the brain", *Psychological Review*, **65** 386-408.

Rosenblatt, F. (1962), *Principles of Neurodynamics*, New York: Spartan.

Rubinstein, R. Y. (1981), *Simulation and the Monte Carlo method*, John Wiley & Sons.

Rumelhart, D. E., McClelland, J. L. & the PDP research group (1986), *Parallel Distributed Processing. Explorations in the Microstructure of Cognition*, volumes 1 & 2, MIT Press, Cambridge, MA.

Ryan, S. W. & Andreae, J. H. (1995), "Improving the performance of Kanerva's associative memory", *IEEE Transactions on Neural Networks*, **6**, 1, 125-130.

Schmidt, R. A. (1982), "The schema concept", in J. A. S. Kelso (ed), *Human Motor Behavior: An introduction*, 219-238, Lawrence Erlbaum Associates.

Sejnowski, T. J. & Rosenberg, C. R. (1987), "Parallel networks that learn to pronounce English text", *Complex Systems*, 1:145-168.

Skarda, C. A. & Freeman, W. J. (1987), "How brains make chaos in order to make sense of the world", *Behavioral and Brain Sciences*, **10**, 161-195.

Stelmach, G. E. (1982), "Motor control and motor learning: The closed-loop perspective", in J. A. S. Kelso (ed), *Human Motor Behavior: An introduction*, 93-116, Lawrence Erlbaum Associates.

Taub, E. (1977), "Movements in nonhuman primates deprived of somatosensory feedback", *Exercises and Sport Science Reviews*, **4**, 335-374.

Turvey, M. T., Flitch, H. L. & Tuller, B. (1982), "The Bernstein perspective: I. The problems of degrees of freedom and context-conditioned variability", in J. A. S. Kelso (ed), *Human Motor Behaviour: An Introduction*, Hillsdale, NJ: Lawerence Erlbaum Associates, 239-252.

Wan, E. A. (1993), "Discrete time neural networks", *Journal of Applied Intelligence*, **3**, 91-105.

Wang, D. & Arbib, M. A. (1990), "Complex temporal sequence learning based on short-term memory", *Proceedings of the IEEE*, **78**, 9, 1536-1543.

Witten, I. H. (1982), *Principles of Computer Speech*, London: Academic Press.

APPENDIX A.1

Derivation of error rates in Hopfield memory, (13)

When pattern P^k is presented to a Hopfield memory the output of unit i is correct if the following is true:

$$P_i^k h_i^k > 0 \tag{84}$$

substituting (3) and (7), (84) becomes:

$$\begin{aligned}
&P_i^k \sum_{\substack{j=1 \\ j\neq i}}^{N} \left(\sum_{l=1}^{N_P} P_i^l P_j^l \right) P_j^k > 0 \\
&\rightarrow \sum_{\substack{j=1 \\ j\neq i}}^{N} \left(\sum_{l=1}^{N_P} P_i^l P_j^l \right) \cdot \left(P_i^k P_j^k \right) > 0 \\
&\rightarrow \sum_{\substack{j=1 \\ j\neq i}}^{N} \left(\left(\sum_{\substack{l=1 \\ l\neq k}}^{N_P} P_i^l P_j^l \right) \cdot \left(P_i^k P_j^k \right) + \left(P_i^k P_j^k \right)^2 \right) > 0 \\
&\rightarrow \sum_{\substack{j=1 \\ j\neq i}}^{N} \sum_{\substack{l=1 \\ l\neq k}}^{N_P} P_i^l P_j^l P_i^k P_j^k > -N \qquad \text{since} \sum_{\substack{j=1 \\ j\neq i}}^{N} \left(P_i^k P_j^k \right)^2 = N-1
\end{aligned} \tag{85}$$

Assuming the patterns are uncorrelated and the probability of a 1 in a pattern is the same as that for a -1, then the product, $P^l_i P^l_j P^k_i P^k_j$ will be a random variable with the probability of there being a 1, p(1)=p(-1)=0.5. If another variable is introduced, which represents the number of times the product, $P^l_i P^l_j P^k_i P^k_j$, equals 1, N_1 , then the sum of the products in (85) will be:

$$\sum_{\substack{j=1 \\ j\neq i}}^{N} \sum_{\substack{l=1 \\ l\neq k}}^{N_P} P_i^l P_j^l P_i^k P_j^k = 2N_1 - (N-1)(N_P - 1) \tag{86}$$

Using (85) there will be an error if:

$$N_1 < \frac{N(N_p - 1) - N}{2} \tag{87}$$

N_1 is a binomial variable. There are $N(N_p-1)$ terms in the sum so N_1 will have an expectation

of:

$$E[N_1]=\frac{(N-1)(N_P-1)}{2} \tag{88}$$

The variance of N_1 will be:

$$\sigma^2[N_1]=\frac{(N-1)(N_p-1)}{4} \tag{89}$$

Using the normal approximation to the binomial distribution, the probability of an error is:

$$\begin{aligned} P_{Error}=P\left(N_1<\frac{(N-1)(N_P-1)-(N-1)}{2}\right)&=\phi\left(\frac{\frac{(N-1)(N_P-1)-(N-1)}{2}-E[N_1]}{\sigma[N_1]}\right)\\ &=\phi\left(\frac{\frac{(N-1)(N_P-1)-(N-1)}{2}-\frac{(N-1)(N_P-1)}{2}}{\sqrt{\frac{(N-1)(N_P-1)}{4}}}\right)\\ &=\phi\left(-\sqrt{\frac{N-1}{N_P-1}}\right)\\ &\sim\phi\left(-\sqrt{\frac{N}{N_P}}\right)=\phi\left(-\frac{1}{\sqrt{\alpha}}\right) \end{aligned} \tag{90}$$

where $\alpha=N_p/N$.

APPENDIX A.2

Derivation of the correction to Kanerva's theory, equation (38)

An expression for the probability of an error is obtained by measuring the variance of the local field of output i, when the input is X^j. This local field is given by the following expression:

$$h_i^j = \sum_{k=1}^{N_p} L_k^j Y_i^k \tag{91}$$

where L^j_k is number of location units which are active for both X^j and X^k.

The variance of h^j_i is

$$Var\{h_i^j\} = N_p Var\{L_k^j Y_i^k\} \tag{92}$$

because the terms $L^j_k Y^k_i$ are independent. At this stage of the derivation Kanerva treats Y^k_i as a constant (zero variance) which results in the incorrect assumption that

$$Var\{L_k^j Y_i^k\} = Var\{L_k^j\} \tag{93}$$

But by definition,

$$\begin{aligned} Var\left\{L_k^j Y_i^k\right\} &= E\left\{\left(L_k^j Y_i^k\right)^2\right\} - \left[E\left\{L_k^j Y_i^k\right\}\right]^2 \\ &= E\left\{\left(L_k^j Y_i^k\right)^2\right\} - \left[E\left\{L_k^{j'}\right\}.E\left\{Y_i^k\right\}\right]^2 \quad \text{since } L_k^j \text{ and } Y_i^k \text{ are independent} \\ &= E\left\{\left(L_k^j\right)^2\right\} \quad \text{since } Y_i^k = \pm 1 \text{ and } E\left\{Y_i^k\right\} = 0 \\ &= Var\left\{L_k^j\right\} + \left[E\left\{L_k^j\right\}\right]^2 \end{aligned} \tag{94}$$

where E(x) is the mean of variable x.

From Kanerva[17],

$$Var\left\{L_k^j\right\}=p^2s\left(1+\frac{h_p^2 s}{4p^2N}\right) \quad (95)$$

where h_p is the value of the slope of the normalised function L^j_k when the distance between X^j and X^k is 0.5N. An expression for h_p is given in (37).

The expectation of L^j_k is:

$$E\left\{L_k^j\right\}=p^2s \quad (96)$$

so,

$$\begin{aligned} Var\left\{h_i^j\right\} &= N_p Var\left\{L_k^j Y_i^k\right\} \\ &= N_p p^2 s\left(1+\frac{h_p^2 s}{4p^2N}+p^2 s\right) \end{aligned} \quad (97)$$

If bit Y^k_i is a 1 then there will be an error if $h^j_i < 0$. Assuming a normal distribution,

$$p\left(h_i^j<0\right)=\Phi\left(\frac{0-E\left\{h_i^j\right\}}{\sqrt{Var\left\{h_i^j\right\}}}\right) \quad (98)$$

$E\{h^j_i\}=ps$, so,

---

[17]in deriving this expression for the variance, Kanerva writes L^j_k as:

$$L_k^j=E\left(L_k^j\right)+e_k^j \quad (1)$$

and makes two simplifying assumptions:

1. that the expectation of L^j_k is a linear function of the Hamming distance between X^j and X^j, and
2. that the variance of e^j_k is constant.

These two assumptions could account for the small, but consistent, difference between the experimental and theoretical results in Figure 3.2.

$$P_{Error} = \Phi\left(\frac{s}{\sqrt{N_p s\left(1+\frac{h_p^2 s}{4p^2 N}+p^2 s\right)}}\right) \tag{99}$$

APPENDIX A.3

Derivation of equation (42)

If the output of unit i is supposed to be 1 then, using the alternative method of reading from the SDM, there will be an error if the average weight from the active location units, μ^i_{active} is less than the average weight from all the location units, μ^i_{total}. Therefore the probability that an output unit has an incorrect output is given by the probability that $\mu^i_{active} - \mu^i_{total} < 0$. To calculate this probability expressions for Var$\{ \mu^i_{active} - \mu^i_{total}\}$ and E$\{ \mu^i_{active} - \mu^i_{total}\}$ are required.

$$Var\{\mu^i_{active} - \mu^i_{total}\} = Var\{\mu^i_{active}\} + Var\{\mu^i_{total}\} - 2Cov\{\mu^i_{active}, \mu^i_{total}\} \qquad (100)$$

where Cov{x,y} is the covariance of variables x and y. The average weight from the active location units to output unit i is:

$$\mu^i_{active} = \frac{h^j_i}{\sum_{k=1}^{N_p} a^j_i} = \frac{\sum_{k=1}^{N_p} L^j_k Y^j_i}{\sum_{k=1}^{N_p} a^j_i} \qquad (101)$$

where a^j_i is the activation of location unit i from input X^j. To simplify the calculations the

number of location accessed by X^j, $\sum_{k=1}^{N_p} a_i^j$ is treated as a constant[18] of value $E\left\{\sum_{k=1}^{N_p} a_i^j\right\}=ps$.

The variance of the average weight from all the location units to output unit i is:

$$Var\left\{\mu_{active}^{i}\right\}=\frac{N_p s\left(1+\frac{h_p^2 s}{4p^2 N}+p^2 s\right)}{s^2} \tag{104}$$

The average weight from all the location units to output unit i is:

$$Var\left\{\mu_{total}^{i}\right\}=\frac{\sum_{j=1}^{N_p}\sum_{k=1}^{s} a_k^j Y_i^j}{s} \tag{105}$$

Using the same reasoning as above the variance of μ^i_{total} will be:

$$Var\left\{\mu_{total}^{i}\right\}=N_p\frac{Var\left\{\sum_{k=1}^{s} a_k^j Y_i^j\right\}+\left[E\left\{\sum_{k=1}^{s} a_k^j Y_i^j\right\}\right]^2}{s^2} \tag{106}$$

$\sum_{k=1}^{s} a_k^j Y_i^j$ can be described by the binomial distribution so

---

[18]Note: this will have some variance but the effect of this variance in this equation is small. This is because $\sum_{k=1}^{N_p} a_i^j$ is large compared to $\sum_{k\neq j} L_k^j Y_i^k$ (which has a mean of 0) and $L_j^j=\sum_{k=1}^{N_p} a_i^j$. However neglecting the effect of this variance means the error rate should be slightly higher than (42). This probably explains why the experimental rate is slightly higher than the theoretical rate in 34.

$$Var\left\{\sum_{k=1}^{s} a_k^j Y_i^j\right\} = sp(1-p) \tag{108}$$

and

$$E\left\{\sum_{k=1}^{s} a_k^j Y_i^j\right\} = ps \tag{109}$$

so,

$$Var\left\{\mu_{total}^i\right\} = \frac{N_p p(1-p+ps)}{s} \tag{110}$$

The covariance of μ^i_{active} and μ^i_{total} is given by:

$$\begin{aligned} Cov\left\{\mu_{active}^i,\ \mu_{total}^i\right\} &= E\left\{\mu_{active}^i \mu_{total}^i\right\} - E\left\{\mu_{active}^i\right\} E\left\{\mu_{total}^i\right\} \\ &= E\left\{\mu_{active}^i \mu_{total}^i\right\} \qquad \text{since } E\left\{\mu_{total}^i\right\} = 0 \end{aligned} \tag{111}$$

Therefore,

$$Cov\left\{\mu_{active}^i,\ \mu_{total}^i\right\} = E\left\{\sum_{k=1}^{N_p} \frac{L_k^j Y_i^k}{\sum_{l=1}^{s} a_l^k} \cdot \sum_{k=1}^{N_p} \frac{\sum_{l=1}^{s} a_l^k Y_i^k}{s}\right\} \tag{112}$$

In the product of the sums in the above equation all the terms will average to 0 except those where the Y^k_i are the same. This gives the covariance as:

$$Cov\left\{\mu^i_{active},\ \mu^i_{total}\right\}=E\left\{\sum_{k=1}^{N_p}\frac{L_k^j\sum_{l=1}^{s}a_l^k}{\sum_{l=1}^{s}a_l^k s}\right\}$$
$$=N_p\frac{E\left\{L_k^j\right\}.E\left\{\sum_{l=1}^{s}a_l^k\right\}}{\sum_{l=1}^{s}a_l^k s} \qquad (113)$$
$$=N_p\frac{p^2 s.ps}{ps.s}$$
$$=N_p p^2$$

Therefore the variance of μ^i_{active} - μ^i_{total} will be:

$$Var\left\{\mu^i_{active}-\mu^i_{total}\right\}=\frac{N_p s\left(1+\frac{h_p^2 s}{4p^2N}+p^2 s+p(1-p+ps)-2p^2 s\right)}{s^2} \qquad (114)$$
$$=\frac{N_p s\left(1+\frac{h_p^2 s}{4p^2N}+p-p^2\right)}{s^2}$$

$E\{\mu^i_{active}\}$ will be 1 and $E\{\mu^i_{total}\}$ will be 0. So $E\{\mu^i_{active}$ - $\mu^i_{total}\}=1$. Therefore, using (98), the probability of an error is:

$$P_{Error}=\Phi\left(\frac{s}{\sqrt{N_p s\left(1+\frac{h_p^2 s}{4p^2N}+p-p^2\right)}}\right) \qquad (115)$$

APPENDIX A.4

The dynamic maze model

The model is of a ball within a tilting maze, shown in Figure 7.1. The movement of the ball is controlled by tilting the maze along two axes of rotation. At time t, the angles of tilt along the y-axis and x-axis, $\theta_y(t)$ and $\theta_x(t)$, give the components of acceleration of the ball in the plane of the maze, in the x and y directions respectively:

$$a_x(t)=g.\sin\theta_y(t) \text{ and}$$
$$a_y(t)=g.\sin\theta_x(t), \quad (116)$$

where g is the gravitational acceleration.

Acceleration normal to the plane of the maze is ignored.

The velocity and acceleration are updated at regular intervals, τ. τ=10ms was used for this demonstration. The x and y components of velocity at time $t+\tau$ are given by:

$$v_x(t+\tau)=v_x(t)+\tau a_x(t) \text{ and}$$
$$v_y(t+\tau)=v_y(t)+\tau a_y(t) \quad (117)$$

Similarly the x and y components of position at time $t+\tau$ is given by:

$$p_x(t+\tau)=p_x(t)+\tau v_x(t) \text{ and}$$
$$p_y(t+\tau)=p_y(t)+\tau v_y(t) \quad (118)$$

When the ball collides with a wall the component of velocity normal to the wall changes sign and is halved. This gives a bouncing effect off the walls.

APPENDIX A.5

PP's interaction with the maze - the first 256 steps.

Time Step	Step Length	tilt	see	Go	look	hear	flash	Production	Production
1	1	east	1			"move_2"	F	P1,1,1	[1 >> east*]
2	1	south	1		2			P1,1,1	[1 >> south]
								P2,1,1	["move_2" >> 2]
3	1	east	2	GO				P1,1,2	[2 >> east]
								P2,2,2	["move_2" 2 >> GO]
4	1	west	1					P1,1,1	[1 >> west]
5	100	east	1					P1,1,1	
6	1	east	2			"move_3"	F	P1,1,2	[2 >> east]
7	1	north	2		3			P1,1,2	[2 >> north]
								P2,1,3	["move_3" >> 3]
8	1	west	3	GO				P1,1,3	[3 >> west]
								P2,2,4	["move_3" 3 >> GO]
9	1	north	2					P1,1,2	[2 >> north]
10	40	west	1					P1,1,1	
11	100	west	1					P1,1,1	
12	100	north	1					P1,1,1	[1 >> north]
13	100	east	1					P1,1,1	
14	30	north	2					P1,1,2	
15	100	east	2					P1,1,2	
16	100	west	2					P1,1,2	[2 >> west]
17	30	east	1					P1,1,1	
18	13	east	2					P1,1,2	
19	100	east	2					P1,1,2	
20	100	east	2					P1,1,2	
21	100	east	2					P1,1,2	
22	100	north	2					P1,1,2	
23	100	west	2					P1,1,2	
24	30	west	1					P1,1,1	
25	100	south	1					P1,1,1	
26	100	west	1					P1,1,1	
27	100	west	1					P1,1,1	
28	100	east	1					P1,1,1	
29	30	east	2					P1,1,2	
30	2	north	6					P1,1,4	[6 >> north]
31	13	west	7					P1,1,5	[7 >> west]
32	1	north	3			"move_4"	F	P1,1,3	[3 >> north]
33	1	east	3		4			P1,1,3	[3 >> east]
								P2,1,5	["move_4" >> 4]
34	1	south	4	GO				P1,1,6	[4 >> south]
								P2,2,6	["move_4" 4 >> GO]
35	1	west	3					P1,1,3	[3 >> east]
36	100	east	3					P1,1,3	[3 >> east]

37	100	north	3					P1,1,3	
38	100	north	3					P1,1,3	
39	100	north	3					P1,1,3	
40	100	west	3					P1,1,3	
41	100	west	3					P1,1,3	
42	100	south	3					P1,1,3	[3 >> south]
43	100	south	3					P1,1,3	
44	100	west	3					P1,1,3	
45	100	north	3					P1,1,3	
46	100	east	3					P1,1,3	
47	100	south	3					P1,1,3	
48	100	south	3					P1,1,3	
49	100	north	3					P1,1,3	
50	100	west	3					P1,1,3	
51	100	north	3					P1,1,3	
52	100	west	3					P1,1,3	
53	100	south	3					P1,1,3	
54	34	south	7					P1,1,5	[7 >> south]
55	3	west	6					P1,1,4	[6 >> west]
56	18	west	5					P1,1,7	[5 >> west]
57	11	east	9					P1,1,8	[9 >> east]
58	100	east	9					P1,1,8	
59	100	south	9					P1,1,8	[9 >> south]
60	100	north	9					P1,1,8	[9 >> north]
61	25	east	5					P1,1,7	[5 >> east]
62	3	east	6					P1,1,4	[6 >> east]
63	7	north	2					P1,1,2	
64	100	south	2					P1,1,2	[2 >> south]
65	25	west	6					P1,1,4	
66	4	west	7					P1,1,5	
67	11	east	6					P1,1,4	
68	20	east	7					P1,1,5	[7 >> east]
69	15	south	8					P1,1,9	[8 >> south]
70	5	north	12					P1,1,10	[12 >> north]
71	10	west	8					P1,1,9	[8 >> west]
72	1	south	4			"move_5"	F	P1,1,6	[4 >> south]
73	1	east	4		5			P1,1,6	[4 >> east]
								P2,1,7	["move_5" >> 5] 7
	1	south	5	GO				P1,1,7	[5 >> south]
								P2,2,8	["move_5" 5 >> GO]
75	1	west	4					P1,1,6	[4 >> west]
76	100	west	4					P1,1,6	
77	100	south	4					P1,1,6	
78	22	west	8					P1,1,9	
79	3	north	7					P1,1,5	[7 >> north]
80	15	north	6					P1,1,4	
81	5	east	2					P1,1,2	
82	100	west	2					P1,1,2	
83	30	south	1					P1,1,1	
84	100	north	1					P1,1,1	
85	100	south	1					P1,1,1	
86	100	east	1					P1,1,1	
87	25	west	2					P1,1,2	
88	2	north	6					P1,1,4	

Time Step	Step Length	tilt	see	Go	look	hear	flash	Production	Production
89	23	east	2					P1,1,2	
90	100	east	2					P1,1,2	
91	100	west	2					P1,1,2	
92	30	north	1					P1,1,1	
93	100	east	1					P1,1,1	
94	25	west	2					P1,1,2	
95	13	north	1					P1,1,1	
96	100	north	1					P1,1,1	
97	100	east	1					P1,1,1	
98	30	south	2					P1,1,2	
99	25	south	6					P1,1,4	[6 >> south]
100	11	south	7					P1,1,5	
101	17	north	8					P1,1,9	[8 >> north]
102	2	north	12					P1,1,10	
103	20	east	8					P1,1,9	[8 >> east]
104	10	west	4					P1,1,6	
105	100	north	4					P1,1,6	[4 >> north]
106	100	north	4					P1,1,6	
107	100	north	4					P1,1,6	
108	100	east	4					P1,1,6	
109	100	north	4					P1,1,6	
110	100	east	4					P1,1,6	
111	100	south	4					P1,1,6	
112	100	north	4					P1,1,6	
113	100	south	4					P1,1,6	
114	100	north	4					P1,1,6	
115	100	north	4					P1,1,6	
116	100	south	4					P1,1,6	
117	100	east	4					P1,1,6	
118	100	west	4					P1,1,6	
119	100	east	4					P1,1,6	
120	73	east	8					P1,1,9	
121	100	west	8					P1,1,9	
122	30	east	7					P1,1,5	
123	13	north	8					P1,1,9	
124	12	west	4					P1,1,6	
125	100	south	4					P1,1,6	
126	20	south	8					P1,1,9	
127	3	north	7					P1,1,5	
128	15	west	6					P1,1,4	
129	5	north	2					P1,1,2	
130	10	east	1					P1,1,1	
131	23	east	2					P1,1,2	
132	100	west	2					P1,1,2	
133	30	north	1					P1,1,1	
134	100	north	1					P1,1,1	
135	100	south	1					P1,1,1	
136	100	south	1					P1,1,1	
137	100	west	1					P1,1,1	
138	100	south	1					P1,1,1	
139	100	east	1					P1,1,1	

Time Step	Step Length	tilt	see	Go	look	hear	flash	Production	Production
140	30	east	2					P1,1,2	
141	2	north	6					P1,1,4	
142	13	south	7					P1,1,5	
143	26	west	8					P1,1,9	
144	3	west	12					P1,1,10	
145	13	south	11					P1,1,11	[11 >> south]
146	15	east	10					P1,1,12	[10 >> east]
147	24	north	11					P1,1,11	
148	15	east	12					P1,1,10	[12 >> east]
149	13	east	8					P1,1,9	
150	10	north	4					P1,1,6	
151	100	north	4					P1,1,6	
152	100	north	4					P1,1,6	
153	100	west	4					P1,1,6	
154	100	west	4					P1,1,6	
155	100	south	4					P1,1,6	
157	3	east	7					P1,1,5	
158	15	east	8					P1,1,9	
159	14	east	12					P1,1,10	
160	100	east	12					P1,1,10	
161	100	east	12					P1,1,10	
162	100	north	12					P1,1,10	
163	20	east	8					P1,1,9	
164	10	south	4					P1,1,6	
165	13	west	8					P1,1,9	
166	10	north	12					P1,1,10	
167	20	east	11					P1,1,11	[11 >> east]
168	13	east	12					P1,1,10	
169	3	north	8					P1,1,9	
170	10	south	4					P1,1,6	
171	19	west	8					P1,1,9	
172	10	north	12					P1,1,10	
173	21	west	8					P1,1,9	
174	2	east	7					P1,1,5	
175	8	east	3					P1,1,3	
176	100	south	3					P1,1,3	
177	100	west	3					P1,1,3	
178	100	east	3					P1,1,3	
179	100	south	3					P1,1,3	
180	100	west	3					P1,1,3	
181	100	south	3					P1,1,3	
182	100	west	3					P1,1,3	
183	100	west	3					P1,1,3	
184	41	north	7					P1,1,5	
185	1	north	3					P1,1,3	
186	100	east	3					P1,1,3	
187	100	north	3					P1,1,3	
188	100	north	3					P1,1,3	
189	100	west	3					P1,1,3	
190	100	west	3					P1,1,3	
191	100	east	3					P1,1,3	

Time Step	Step Length	tilt	see	Go	look	hear	flash	Production	Production
192	100	north	3					P1,1,3	
193	100	west	3					P1,1,3	
194	100	north	3					P1,1,3	
195	100	east	3					P1,1,3	
196	100	east	3					P1,1,3	
197	100	south	3					P1,1,3	
198	100	south	3					P1,1,3	
199	100	south	3					P1,1,3	
200	100	east	3					P1,1,3	
201	74	east	7					P1,1,5	
202	30	east	8					P1,1,9	
203	100	west	8					P1,1,9	
204	30	north	7					P1,1,5	
205	13	south	3					P1,1,3	
206	10	north	7					P1,1,5	
207	13	west	6					P1,1,4	
208	8	south	2					P1,1,2	
209	9	east	1					P1,1,1	
210	13	west	2					P1,1,2	
211	8	west	6					P1,1,4	
212	1	east	5		"move_6"	F	P1,1,7		[5 >> south]
213	1	west	5		6			P1,1,7	[5 >> south]
								P2,1,9	["move_6" >> 6]
214	1	south	6	GO				P1,1,4 P2,2,10	["move_6" 6 >> GO]
215	1	west	5					P1,1,7	
216	2	south	9					P1,1,8	
217	100	west	9					P1,1,8	
218	100	west	9					P1,1,8	
220	100	north	9					P1,1,8	
221	100	east	9					P1,1,8	
222	100	south	9					P1,1,8	
223	100	north	9					P1,1,8	
224	28	east	5					P1,1,7	
225	100	north	5					P1,1,7	[5 >> north]
226	100	south	5					P1,1,7	
227	25	north	9					P1,1,8	
228	13	west	5					P1,1,7	
229	100	south	5					P1,1,7	
230	100	west	5					P1,1,7	
231	100	east	5					P1,1,7	
232	100	north	5					P1,1,7	
233	100	north	5					P1,1,7	
234	100	east	5					P1,1,7	
235	100	east	5					P1,1,7	
236	100	west	5					P1,1,7	
237	100	east	5					P1,1,7	
238	1	north	6			"move_7"	F	P1,1,4	
239	1	west	6		7			P1,1,4 P2,1,11	["move_7" >> 7]
240	1	east	7	GO				P1,1,5 P2,2,12	["move_7" 7 >> GO]
241	1	west	2					P1,1,2	
242	15	north	1					P1,1,1	

Time Step	Step Length	tilt	see	Go	look	hear	flash	Production	Production
243	100	north	1					P1,1,1	
244	100	north	1					P1,1,1	
245	100	east	1					P1,1,1	
246	30	west	2					P1,1,2	
247	13	north	1					P1,1,1	
248	100	south	1					P1,1,1	
249	100	north	1					P1,1,1	
250	100	north	1					P1,1,1	
251	100	north	1					P1,1,1	
252	100	west	1					P1,1,1	
253	100	west	1					P1,1,1	
254	100	west	1					P1,1,1	
255	100	east	1					P1,1,1	
256	100	east	1					P1,1,1	

APPENDIX A.6

PP's memory

Actions and stimuli

Action	Instances
tilt	west* east* north south
Go	GO
Move_Eye	1* 2* 3* 4* 5* 6* 7* 8* 9* 10 11 12

Stimulus	Instances
see	1* 2* 3* 4* 5* 6* 7* 8* 9* 10 11 12
Hear	"move_1"* "move_2"* "move_3"* "move_4"* "move_5"* "move_6"* "move_7"* "move_8"* "move_9"* "move_10" "move_11" "move_12"

Templates

Cluster 1
No of templates = 1
1 [.see >> tilt]

Cluster 2
No of templates = 2
1 [.Hear >> Move_Eye]
2.[..Hear, .Move_Eye >> Go]

Memory map

Contexts stored for cluster 1

Context	Context no
1*	(1)
2*	(2)
3*	(3)
6*	(4)
7*	(5)
4*	(6)
5*	(7)
9*	(8)
8*	(9)
12	(10)
11	(11)
10	(12)

Memory map for cluster 1

Context no	Production no	Prediction	Next Context	Number of times context has followed production
1	P1,1,1	east*	1	57
			2	112
		south	1	16
		west*	1	29
		north	1	26
2	P1,1,2	east*	2	20,
			4	1
		north	2	30,
			4	1,
			1	2
		west*	1	86,
			2	24
		south	4	307,
			2	76,
			1	23
3	P1,1,3	north	3	30
		east*	3	25
		west*	3	25
		south	5	111,
			3	60
4	P1,1,4	north	5	107,
			2	101,
			7	195,
			4	23
		west*	7	118,
			2	10,
			4	11,
			5	10
		south	2	5,
			5	14,
			7	8,
			4	2
		east*	2	192,
			4	104,
			5	565,
			7	15
5	P1,1,5	west*	3	36,
			9	14,
			4	512,
			5	63
		south	4	21,

			9	11,
			5	30,
			3	3
		east*	3	5,
			9	204,
			4	12,
			5	48
		north	3	67,
			5	24,
			4	150,
			9	146,
			2	1
6	P1,1,6	south	9	828,
			6	73
		west*	6	16,
			9	1
		north	6	31
		east*	6	25,
			9	1
7	P1,1,7	north	7	17,
			8	2,
			4	4
		east*	4	328,
			7	116,
			8	50
		west*	8	4,
			7	16
		south	7	19,
			8	74,
			4	4
8	P1,1,8	east*	8	18,
			7	1
		south	8	33
		north	7	157,
			8	60
		west*	8	21
9	P1,1,9	north	10	20,
			6	84,
			5	2,
			9	16
		east*	6	14,
			10	14,
			9	6
		west*	6	731,

Context no	Production no	Prediction	Next Context	Number of times context has followed production
			10	607,
			5	372,
			9	37
		south	10	158,
			9	23,
			5	1,
			6	1
10	P1,1,10	north	9	798,
			10	90,
			11	138
		west*	10	33,
			11	139,
			9	5
		east*	10	22,
			9	4
		south	10	22,
			11	2,
			9	1
11	P1,1,11	east*	10	258,
			11	53
		south	12	10,
			10	7,
			11	2
		north	10	14,
			12	10,
			11	1
		west*	12	89,
			11	16
12	P1,1,12	east*	11	109,
			12	60
		west*	12	27
		north	12	23
		south	12	31

Productions stored in cluster 2

Template 1

Context	Production No	Prediction (Move Eye action)
"move_2"*	P2,1,1	2
"move_3"*	P2,1,2	3

Context no	Production no	Prediction	Next Context	Number of times context has followed production
"move_4"*	P2,1,3		4	
"move_5"*	P2,1,4		5	
"move_6"*	P2,1,5		6	
"move_7"*	P2,1,6		7	
"move_8"*	P2,1,7		8	
"move_9"*	P2,1,8		9	
"move_10"	P2,1,9		10	
"move_11"	P2,1,10		11	
"move_12"	P2,1,11		12	
"move_1"*	P2,1,121		1	

Template 2

Context		Context no	
"move_2"*	2*	P2,2,1	GO
"move_3"*	3*	P2,2,2	GO
"move_4"*	4*	P2,2,3	GO
"move_5"*	5*	P2,2,4	GO
"move_6"*	6*	P2,2,5	GO
"move_7"*	7*	P2,2,6	GO
"move_8"*	8*	P2,2,7	GO
"move_9"*	9*	P2,2,8	GO
"move_10"	10	P2,2,9	GO
"move_11"	11	P2,2,10	GO
"move_12"	12	P2,2,11	GO
"move_1"*	1*	P2,2,12	GO