

Correlation Matrix Memories: Improving Performance for Capacity and Generalisation

STEPHEN HOBSON

Ph.D. Thesis

This thesis is submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy.

University of York
Computer Science

September 2011

Abstract

The human brain is an extremely powerful pattern recogniser, as well as being capable of displaying amazing feats of memory. It is clear that human memory is associative; we recall information by associating items together so that one may be used to recall another. This model of memory, where items are associated as pairs rather than stored at a particular location, can be used to implement computer memories which display powerful properties such as robustness to noise, a high storage capacity and the ability to generalise. One example of such a memory is the Binary Correlation Matrix Memory (CMM), which in addition to the previously listed properties is capable of operating extremely quickly in both learning and recall, as well as being well suited for hardware implementation. These memories have been used as elements of larger pattern recognition architectures, solving problems such as object recognition, text recognition and rule chaining, with the memories being used to store rules. Clearly, the performance of the memories is a large factor in the performance of such architectures.

This thesis presents a discussion of the issues involved with optimising the performance of CMMs in the context of larger architectures. Two architectures are examined in some detail, which motivates a desire to improve the storage capacity and generalisation capability of the memories. The issues surrounding the optimisation of storage capacity of CMMs are discussed, and a method for improving the capacity is presented. Additionally, while CMMs are able to generalise, this capability is often ignored. A method for producing codes suitable for storage in a CMM is presented, which provides the ability to react to previously unseen inputs. This potentially adds a powerful new capability to existing architectures.

Contents

1	Introduction	21
1.1	Motivation	21
1.2	Chapter overview	22
1.3	Summary of contributions	23
2	Associative memory	25
2.1	Introduction	25
2.2	Associationism	25
2.3	Models of memory	27
2.4	Associative memory in computing	28
2.5	Implementing Associative Memories	29
2.5.1	Conventional Methods	29
2.5.2	Neural Methods	30
2.6	Hopfield Networks	31
2.7	Correlation Matrix Memories	33
2.7.1	Description	33
2.7.2	Crosstalk	35
2.7.3	Binary CMMs	36
2.7.4	Thresholding and Representation	37
2.7.5	Weaknesses	38
2.8	Bidirectional Associative Memory	38
2.9	The N-tuple method	40
2.10	Advanced Distributed Associative Memory	41
2.11	Advanced Uncertain Reasoning Architecture	42
2.12	Summary	45
3	Applications of Correlation Matrix Memories	47
3.1	Introduction	47
3.2	Cellular Associative Neural Network	47

3.2.1	Introduction	47
3.2.2	Cellular Automata	48
3.2.3	Syntactic Pattern Recognition	50
3.2.4	The Static Architecture	52
3.2.5	Incorporating Uncertainty	57
3.2.6	Further work	58
3.3	Associative Rule Chaining Architecture	60
3.3.1	Introduction	60
3.3.2	Rule Chaining	60
3.3.3	Parallel Distributed Computation	62
3.3.4	Challenges	63
3.3.5	Overview of the Associative Rule Chaining Architecture	64
3.3.6	Learning	68
3.3.7	Recall	68
3.3.8	Robustness	72
3.3.9	Experiments	72
3.3.10	Results	74
3.3.11	Further work	76
3.4	Performance and Data Representation	80
3.5	Summary	81
4	Storage Capacity of Correlation Matrix Memories	83
4.1	Introduction	83
4.2	Motivation	83
4.3	Theoretical results	84
4.4	Generating Sparse Fixed Weight Codes	89
4.4.1	Random codes	90
4.4.2	Baum codes	90
4.4.3	Turner codes	96
4.5	Thresholding	97
4.6	Improving the Storage Capacity with Baum Codes	103
4.7	Results for L-wta	103
4.8	Further Work	106
4.9	Summary	109
5	Coding for Generalisation in Correlation Matrix Memories	111
5.1	Introduction	111
5.2	Motivation	111

5.3	Generalisation and storage capacity	113
5.4	Similarity	115
5.5	Existing methods	116
5.5.1	Thermometer codes	117
5.5.2	Fixed weight Gray codes	118
5.5.3	CMAC-Gray	118
5.5.4	Discussion	119
5.6	Multi-dimensional inputs	120
5.7	Overlapped binary code construction	123
5.7.1	Production of an overlap matrix	124
5.7.2	Code size optimisation and generation	125
5.7.3	Create fixed weight code	129
5.8	Clique selection strategy	129
5.9	Analysis of OBCC	133
5.10	Further work	139
5.11	Summary	140
6	Conclusions and Further Work	141
6.1	Review	141
6.2	Further Work	144
6.2.1	Cellular Associative Neural Network	144
6.2.2	Associative Rule Chaining Architecture	145
6.2.3	Usage of Baum codes in CMMs	146
6.2.4	Overlapped Binary Code Construction	146
A	Character Recognition Data Set	149

List of Tables

4.1	Experimental results for L-wta when varying input size	105
4.2	Experimental results for L-wta when varying the weight of the input code	105
4.3	Experimental results for L-wta when varying the size of the output code	105
4.4	Experimental results for L-wta when varying the weight of the output code	105
5.1	Details of codes resulting from application of OBCC	136
A.1	The 16 features in the character recognition data set	149
A.2	The features of the selected 26 characters from the data set	150

List of Figures

2.1	Kohonen’s model of associative memory	28
2.2	A Hopfield network with four neurons	32
2.3	A correlation matrix memory with continuous weights	34
2.4	A bidirectional associative memory	39
2.5	The N-tuple method, with $N = 2$	41
2.6	The ADAM network architecture	42
2.7	The layout of AURA	43
2.8	An example of superposition	44
2.9	An example of storing different arity rules in one CMM	45
3.1	A single transition of a cellular automaton	49
3.2	An example of parsing using a grammar	51
3.3	An example of the operation of a CANN	52
3.4	The architecture of an associative processor	54
3.5	The spread of information through the CANN when using different neighbourhoods	55
3.6	An example set of single arity rules, and the resulting search tree	61
3.7	An example of parallel distributed computation: A state machine implemented in a correlation matrix memory	63
3.8	The Associative Rule Chaining Architecture (ARCA)	65
3.9	A visualisation of the columns within the matrices in ARCA	67
3.10	An example of ARCA running on a set of rules	67
3.11	A single iteration of a recall operation in ARCA	71
3.12	Contour plots showing the recall error performance for ARCA where the branching factor is 1 and 2	77
3.13	Contour plots showing the recall error performance for ARCA where the branching factor is 3 and 4	78
3.14	The implementation of multiple arity CMMs in ARCA	79
4.1	An example of the generation of Baum codes	91

4.2	Information content for random fixed weight binary codes and Baum codes over a variety of code lengths	93
4.3	Number of Baum codes produced compared to number codes which can theoretically be stored in a CMM	94
4.4	An example of Willshaw and L-max thresholding	98
4.5	The performance of L-max and Willshaw thresholds with a noise free input	100
4.6	The performance of L-max and Willshaw thresholds with 0.5% input noise	100
4.7	The performance of L-max and Willshaw thresholds with 1% input noise	101
4.8	The performance of L-max and Willshaw thresholds with 5% input noise	101
4.9	The performance of L-max and Willshaw thresholds with 10% input noise	102
4.10	Two comparisons of the storage capabilities of a CMM when using L-max and L-wta	107
4.11	Two further comparisons of the storage capabilities of a CMM when using L-max and L-wta	108
5.1	A simple example of generalisation in a CMM	113
5.2	Code space when designing codes for capacity or generalisation	114
5.3	Examples of thermometer, fixed weight Gray and CMAC-Gray codes	117
5.4	A comparison of the number of codes which can be generated for a given code length using various methods	121
5.5	Similarity matrices for codes representing the numbers 1-50 generated by thermometer codes, fixed weight Gray codes and CMAC-Gray codes.	122
5.6	Construction of an overlap matrix for the numbers 1 to 5	125
5.7	A naïve translation from overlap matrix to code matrix	126
5.8	Three columns merging into one, whilst maintaining code overlap	127
5.9	An example of the optimisation of a code matrix by removing cliques from the graph representation	127
5.10	Code is made fixed weight with the addition of single bit columns	129
5.11	A code optimisation selecting cliques using a greedy strategy	130
5.12	A code optimisation selecting cliques using a more optimal strategy	131
5.13	A similarity matrix showing the distance classes between characters	134
5.14	A similarity matrix for a code generated for the characters “A” to “Z” using OBCC, with $p = 3$	136

5.15 A similarity matrix for a code generated for the characters “A” to
“Z” using OBCC, with $p = 6$ 137

5.16 A similarity matrix for a code generated for the characters “A” to
“Z” using OBCC, with $p = 10$ 137

5.17 A similarity matrix for a code generated for the characters “A” to
“Z” using thermometer codes 139

Acknowledgements

There are a number of people to whom I am indebted for their support in this project. Firstly, I would like to thank my wife Katie for her unending support throughout the completion of this work, and my parents for their constant encouragement. My gratitude is also extended to my supervisor Jim Austin, who provided guidance, feedback and understanding, as well as many interesting and entertaining conversations. In addition, I would like to thank my assessors Simon O’Keefe and Bruce Graham for giving up their valuable time in order to assist with this project. My brother, Andrew, was also a huge help over the past years helping with mathematics, and also aiding with the proof reading of this thesis. I would also like to acknowledge the White Rose Grid (<http://www.wrgrid.org.uk/>), which was used extensively in the simulations presented in this work, and Aaron Turner who provided valuable support in the usage of this service. Finally, the UCI Machine Learning Repository (<http://archive.ics.uci.edu/ml>) was a helpful source of data.

Declaration

This thesis has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree other than Doctor of Philosophy of the University of York. This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by explicit references. Some of the material in Chapter 4 has been previously published [42] [43].

For Katie and Joshua

Chapter 1

Introduction

1.1 Motivation

Correlation matrix memories (CMMs) are neural associative memories which are able to store a large number of associations between binary patterns, and can store and recall these patterns at great speed. They are also tolerant to noise on the input and are capable of reacting appropriately to previously unseen inputs which bear a similarity to known inputs. The binary weighted form of this network has been used to great effect in a number of novel architectures in various domains. These include rule based systems [9] [12], image recognition [21] [71], and graph matching [56]. The capabilities of CMMs in these architectures are very dependent upon the representations which are used for the data items being stored. There are a number of difficulties and trade-offs involved in the selection of a data representation. However, this problem has not always been studied in detail in the production of these architectures, possibly resulting in sub-optimal performance.

There are two primary areas of performance in CMMs which are focused on in particular. The first is storage capacity. A binary weighted CMM is capable of storing a very large number of associations between input and output pairs providing an appropriate data encoding is chosen. The variation in capacity varies enormously over the space of possible encodings, so the choice is very important indeed. A sparse fixed weight representation gives a very good capacity, and a number of algorithms exist for generating codes of this form. In addition, the thresholding function which is used to convert from a continuous output activity to a binary code has an effect on the storage capacity of a CMM. A well chosen threshold will result in fewer errors

on the output of the memory, which allows a larger number of associations to be stored in the memory before the errors become intolerable.

The second performance area we focus upon is *generalisation*, the ability of the CMM to make appropriate recalls when presented with previously unseen inputs. While other neural networks, such as multi-layer perceptrons, learn an internal representation which allows for generalisation, in a CMM similar input items must be assigned similar input codes. The ability to generalise would be extremely beneficial in the architectures listed above. However, this capability of CMMs has remained largely unexplored in these pieces of work.

This thesis aims to investigate the storage properties of CMMs, and to develop new techniques for improving the performance of these memories in terms of both storage capacity and generalisation. These benefits can then be passed on to any architecture which utilises CMMs in its structure.

1.2 Chapter overview

In Chapter 2 a discussion of associative memories is conducted. Some background is given on the theory of “associationism”, which explores the idea that human memory consists of associations between memories. This theory is the basis for a computational model of associative memory, which has also been called a *mapping memory*. Two contrasting methods of constructing such memories are discussed: content addressable memories and neural associative memories. A number of examples of associative memories are described, with particular focus on the aforementioned correlation matrix memory (CMM).

Chapter 3 contains detailed descriptions of two architectures which use CMMs as a primary component. These are the Cellular Associative Neural Network (CANN) and the Associative Rule Chaining Architecture (ARCA). The CANN is an object recognition architecture based upon ideas taken from cellular automata. It consists of a regular grid of cells, each of which takes an input from a small subsection of an image. By exchanging information within a neighbourhood each cell builds up a local representation of the object to which it belongs. Objects are represented as a hierarchy of labels, and each cell is able to select an appropriate label through the use of rules. The rules are stored in CMMs, enabling their quick learning and recall, as well as allowing partial matching for rules which are not completely fulfilled.

ARCA aims to solve the problem of forward chaining; applying a set of rules to a number of symbols, and determining the consequences. While traditional approaches to this problem involve searching through a tree using a method such as depth-first search, ARCA is able to process each layer of the tree in parallel. Again, CMMs are used to store rules in the architecture. An experimental analysis of the architecture is conducted, examining how the recall performance is affected by the size of the CMMs.

In Chapter 4 a discussion of the issues involved with determining and maximising storage capacities of CMMs is presented. A review of theoretical work on the subject is conducted, identifying some important features which codes to be stored in CMMs require in order to maximise capacity. Specifically, codes should be sparsely coded, fixed weight and close to being an orthonormal set. A number of methods for generating such codes are discussed, along with some appropriate thresholding techniques. A novel thresholding technique, L-wta, is then presented for one particular code generating algorithm. Results are presented demonstrating the improvement in performance provided by L-wta.

Chapter 5 focuses on the idea of generalisation, and how codes might be generated for CMMs which allow the memory to display this property. Some discussion of how this property interacts with storage capacity is presented. Existing methods which can be used to generate codes which are suitable for providing generalisation in a CMM, and their strengths and weaknesses are examined. A novel method for optimising suitable codes, Overlapped Binary Code Construction (OBCC), is described. This method involves finding cliques within a generated graph to minimise the size of the generated codes, and so a variety of strategies for selecting cliques are discussed. The method is demonstrated generating codes for examples from a character recognition dataset.

Finally, Chapter 6 contains a review of the work conducted in the thesis. A summary of the further work which has been identified is also presented.

1.3 Summary of contributions

The following is a brief list of the contributions made in this thesis:

- The development of a pessimistic view of the prospect of true parallel

performance in the Cellular Associative Neural Network architecture (Section 3.2.6.3).

- An experimental analysis of the Associative Rule Chaining Architecture, the first such investigation (Section 3.3.9).
- A discussion of a number of methods for generating sets of coprime numbers suitable for use in the algorithm of Baum et al. [14] (Section 4.4.2.1).
- The use of L-wta thresholding in correlation matrix memories storing fixed weight codes generated by the algorithm of Baum et al., which increases the storage capacity of the memory (Sections 4.6 and 4.7).
- The introduction of the Overlapped Binary Code Construction (OBCC) optimisation technique, which utilises graph theory to produce short codes which display a predefined degree of overlap with one another (Section 5.7).

Chapter 2

Associative memory

2.1 Introduction

Associations are a concept which we are very familiar with us as humans, since our memories clearly operate using associations to link events, people and other concepts together. In this chapter we examine this idea, termed “associationism”, and explore the possibilities it presents for designing computational models. A formal definition is given differentiating an associative memory from a traditional computer memory. We then examine some methods which can be used to implement associative memories, identifying two major types: Conventional content addressable memories and neural associative memories. A variety of neural approaches to implementing associative memories are then explored.

2.2 Associationism

The concept of associative memory is not a new idea. Indeed, the observation that human memory is associative can be seen as early as 350BC, in Aristotle’s essay *On Memory and Reminiscence* [4], when he made the observation that when we attempt to remember something, we do so by first attempting to recall some other thing which is associated to it:

Accordingly, therefore, when one wishes to recollect, this is what he will do: He will try to obtain a beginning of a movement whose sequel shall be the movement he desires to reawaken.

Based upon our own experience it is clear that our memories are made up of links between facts, places, people, events and so on. For example, visiting a location may remind you of a specific person. “Associationism” is the theory of such a model of memory. Aristotle’s observations were an early example of this idea, and were compiled into the “Classical Laws of Association”, as given by Kohonen in [60]:

Mental items (ideas, perceptions, sensations or feelings) are connected in memory under the following conditions:

- 1) If they occur simultaneously (“spacial contact”).
- 2) If they occur in close succession (“temporal contact”).
- 3) If they are similar.
- 4) If they are contrary.

In other words, items which occur close to one another (either in space or time) are associated in memory somehow. In addition, our memories are capable of recognising items which have a high positive or negative correlation with some input ‘key’ item.

While the detail of the theories within associationism have varied greatly, a number of features are universal [3]:

- Ideas, sense data, memory nodes, or similar mental elements are associated together in the mind through experience. Thus, associationism is *connectionistic*.
- The ideas can ultimately be decomposed into a basic stock of “simple ideas.” Thus, associationism is *reductionistic*.
- The simple ideas are to be identified with elementary, unstructured sensations. Because it identifies the basic components of the mind with sensory experience, associationism is *sensationalistic*.
- Simple, additive rules serve to predict the properties of complex associative configurations from the properties of the underlying simple ideas. Thus, associationism is *mechanistic*.

We shall see that the associative memories described in this chapter align with the majority of these ideas. The memories primarily follow a connectionist philosophy, being examples of artificial neural networks. These memories store a large number of associations between simple data items, using simple additive rules to learn the

mappings. A thorough review of associationism is given by Anderson & Bower in [3].

2.3 Models of memory

When discussing differing methods which can be used to store information, Palm defines two differing types of memory; *listing memories* and *mapping memories* [72]. A listing memory can be defined as follows. If we define the information to be stored as a set of messages M , then a listing memory can be thought of as storing a sequence of messages, in which each message is taken from some alphabet A . We can define this sequence as:

$$M_l(A) = (s_1, \dots, s_n) : n \in \mathbb{N}, s_1, \dots, s_n \in A \quad (2.1)$$

Groups of messages can be combined by concatenating lists of them together. This paradigm is very familiar in computing; traditional computer memory works like a filing cabinet in that there are a series of locations in which data can be stored, essentially a sequence. We can retrieve these items if we know the *address* of the location at which it is stored. A mapping memory is defined very differently, and is perhaps a less intuitive concept. Rather than the messages being simply elements of an alphabet, they are elements taken from a mapping between a set of questions Q and a set of answers A . This means that for a mapping memory, M is defined as follows, where P is the set of possible questions:

$$M_m(P, A) = m : Q \rightarrow A, Q \subseteq P \quad (2.2)$$

In this case we can combine groups of messages by joining the mappings m and m' together so that the appropriate answer is retrieved for a question taken from either Q or Q' (so long as the sets of questions are disjoint i.e. $Q \cap Q' = \emptyset$).

In other words, the distinction between a listing memory and a mapping memory is made by the contents of the messages which it stores and the existence or lack of any sequential information. A listing memory stores only elements from an alphabet in sequence, and as mentioned above is a familiar structure in computing. A mapping memory works differently, building up a mapping between questions and answers, with the recall operation upon presentation of a known question returning

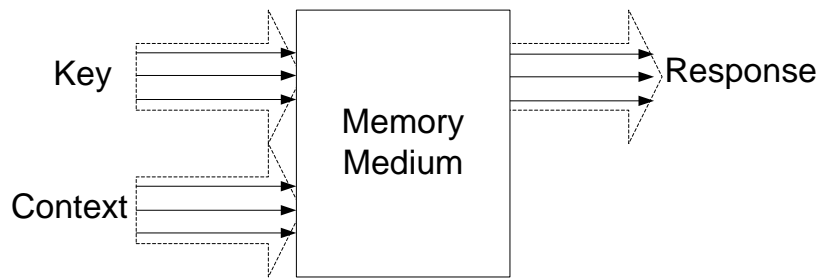


Figure 2.1: Kohonen's model of associative memory

the associated answer. The items are not stored sequentially. Associative memory is based upon this mapping memory model.

2.4 Associative memory in computing

We can take inspiration from associationism in order to develop memory in computer systems which conforms with Palm's idea of a mapping memory. Such memories display very different properties from traditional computer memory. In an associative memory, we do not use the concept of an address, as in a listing memory structure. Instead, data is stored as an association between an input item A and an output item B . This means that presenting data item A at the input will result in the data B being retrieved at the output. This is illustrated by the model proposed by Kohonen [59], shown in Figure 2.1.

For example, imagine that we want to store an association between the data items "1984" and "Orwell". We would store these in an associative memory by presenting "1984" at the input and "Orwell" at the output. In the future, if we wished to retrieve the data associated with "1984" we would present the value to the input, and would retrieve "Orwell" at the output. Contrast this with a conventional computer memory. If we wanted to store the data "1984", we would get the address of some available memory, say position 250, and place the data at that address. If we wanted to retrieve this data at any time in the future, this would be achieved by presenting the address 250 to the memory. Note the address chosen is completely arbitrary- it has no relationship with the data stored. In order to maintain associations between pairs of items we would have to retain some form of lookup table.

Associative memories can serve many useful purposes. They are commonly used for data caching, and in database engines. Furthermore, they can be utilised in pattern recognition engines, as we will be exploring. Chapter 3 contains extended examples

of some applications of an associative memory.

2.5 Implementing Associative Memories

2.5.1 Conventional Methods

There is more than one way to implement an associative memory, depending upon the properties which are desired. A conventional method would be to store all the data items in a normal computer memory system, then create a table with columns for values, and the addresses of the locations of the associated values. So, if “A” is associated to “B”, with “B” being stored at memory location 100, we would store “A” and 100 adjacent to one another in the table. Then, if we wish to recall the item associated with “A” we look-up “A” in the table, find the address of the associated item (100), and lookup that memory location to find “B”. However, such a system is slow to access data, due to the requirement to lookup the initial value in a table, and then to make an additional access to memory to retrieve the associated item. While there are methods which can be used to speed up the initial lookup (such as hashing [57]), the requirement for multiple memory reads remains.

A *content addressable memory (CAM)* is a memory system which speeds the recall of associated items. Whereas a conventional memory system stores data by reference, in a CAM it is stored by value [7]. A simple example of this would be to take the binary value of the input data and to use that as the address in which to store the associated value. So, if we were associating 81 to 9 we would use the address $81 = 01010001$, and store the value for 9 there. The success of this method is clearly largely dependent upon the form which the input data takes, and is likely to result in an unnecessarily large memory. More commonly, a CAM will operate using a list of input items paired with specialised hardware which enables all associated items for a given input to be returned, with the recall for all items in the list being conducted simultaneously. This hardware can potentially also support partial matching, allowing the memory to provide some level of *generalisation* [7] and fault tolerance. Generalisation is defined as the capability to react appropriately to previously unseen inputs, and is explored in detail in Chapter 5. However, specialised hardware solutions are expensive and have a low storage capacity in general.

2.5.2 Neural Methods

A contrasting method of implementing an associative memory is to use an *artificial neural network*. Artificial neural networks are inspired by biological neural networks, such as the human brain. The cerebral cortex, for example, consists of a huge number of neurons (approximately 10^{11}), with many connections between them. Each biological neuron communicates through short pulses of activity, taking input from a number of neurons and transmitting its own signal to other neurons. By comparing the speed at which neurons transmit information (in the order of milliseconds) and the speed of human decisions, it has been calculated that problems such as face recognition must be achieved using chains of neurons approximately 100 long [28]. For such complex tasks to be achieved in so few “steps”, it follows that some highly parallel computation must be involved. It is this idea which inspires artificial neural networks, known simply as *neural networks* from here onwards.

A neural network consists of a number of simple artificial neurons with many connections between them. As with biological neural networks, they are highly parallel, with each neuron acting locally and transmitting a signal to those neurons it is connected to. Each connection between neurons will commonly have a *weight* associated with it, and it is these which “store” data. McCulloch & Pitts proposed a computational model for a neuron, specifically a binary threshold unit. Their neuron model takes a weighted sum of inputs and will output a “1” if the sum is above some threshold [67]. Other simple functions can be used in place of the sum and threshold, such as a piecewise linear function, a sigmoid function or a Gaussian function. For any of these models, a network is able to exhibit complex behaviour despite each neuron being capable of performing only simple calculations. Such behaviour requires the network adjust the weights of the connections, usually through the presentation of a set of examples, a process known as *training* or *learning*. A thorough introduction to neural networks is given by Jain et al. in [50].

Neural networks are capable of solving many classes of problem. These include pattern classification, function approximation, data clustering and control. Any neural network which can perform pattern recognition can act as an associative memory. This can be seen in that pattern recognition involves the association of a data vector with a class. If we view the data as our input, and the class as our output then the equivalence is clear. However, some types of network are more suited to the task of acting as an associative memory than others, and some well suited examples are presented in the remainder of this chapter.

An important property of neural networks is their ability to generalise. Such a capability in a conventional memory system requires specialised hardware or a very expensive search. In general, in a neural based memory responding to a previously unseen input is no slower than responding to a known input.

In contrast to a conventional memory, in a neural network the stored data is distributed over the network. This enables increased fault tolerance, as an error in one part of the network is less likely to have a large effect on the output of the network for any given input. However, it does also mean that there is the possibility that data items will interfere with one another. For example, interference may occur if two very similar inputs are associated with dissimilar outputs. This highlights an issue with neural based memories; that the storage properties are dependent on the data being stored. Because of this, *data representation* becomes very important.

Neural associative memories can be *autoassociative* or *heteroassociative*. In an autoassociative network data items are associated with themselves. Such a network can be used for cleaning up noisy vectors. For example, if complete pictures are stored in the associative memory then the presentation of a noisy or incomplete version of a stored image to the network should result in the recall of the original picture. In contrast, a heteroassociative network associates an input vector with a different output vector. For example this might be the an association between book titles and their authors.

In summary, neural associative memories are able to operate with noisy or incomplete data (they generalise), they can operate very quickly, and can store data efficiently [7]. The associative memories which are examined in the remainder of this chapter are all neural associative memories.

2.6 Hopfield Networks

The *Hopfield network* [46] is an example of an autoassociative memory. It consists of a single layer of neurons, which are fully connected and have symmetric weights. The Hopfield network is known as a *recurrent* network, which means its outputs are fed back into its inputs. This can be seen in the example of a Hopfield network shown in Figure 2.2.

The storage algorithm for the Hopfield network is given in equations 2.3 and 2.6 [15]. In this case we are using a bipolar representation (binary values are either 1

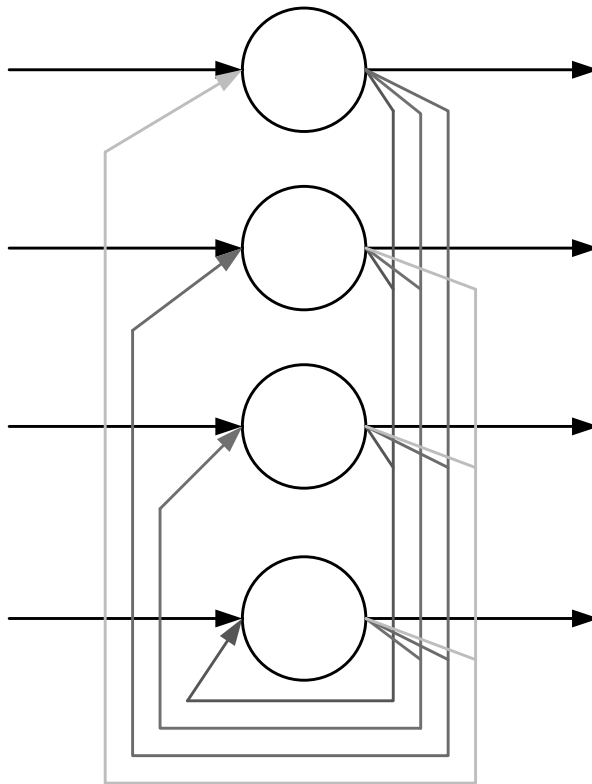


Figure 2.2: A Hopfield network with four neurons

or -1). The weight matrix W is built from the correlations between all pairs in the data vector to be learned x .

$$W_{ij} = \sum_{s=0}^{M-1} x_i^s x_j^s \text{ where } i \neq j \quad (2.3)$$

$$W_{ij} = 0 \text{ where } i = j$$

This is a form of *Hebbian* learning. This is a simple paradigm for learning, in which the strength of correlated synapses (weights) is increased. In this case negatively correlated synapses are also weakened, although this was not a part of Hebb's original rule [41]. This learning rule has the advantage that it is very quick to calculate, especially when compared to learning methods such as *back propagation* [77].

Recall of a stored pattern is an iterative process. The pattern is set at the nodes and the network updates its state until it is stable, using the update rule shown in equation 2.4. The function $f_h(x)$ returns 1 if $x > 0$ and -1 if $x < 0$.

$$x_i(t+1) = f_h \left[\sum_{j=0}^{N-1} W_{ij} x_j(t) \right] \quad (2.4)$$

The network is essentially an elegant implementation of the hill climbing local search algorithm. Equation 2.4 defines a search over an energy landscape of solutions. The stored patterns represent minima in this landscape. Because of this, the Hopfield network has properties of local search—it is guaranteed to converge to a solution. However, this is not guaranteed to be a stored pattern. There may be many local optima in the energy landscape in which the “search” can become stuck.

A major weakness of the Hopfield network is that it has limited storage capacity. In his original paper, Hopfield commented that approximately $0.15N$ items could be stored before the error in recall was severe, where N is the number of neurons. A later paper showed that a theoretical maximum was $0.138N$ [2]. This poor storage is due to noise in the network caused by *crosstalk* between the patterns, and the appearance of spurious states as data is stored in the network [41].

It has already been noted that the Hopfield network can become stuck in local minima which do not represent stored patterns. A network which attempts to resolve this weakness is the *Boltzmann machine* [41]. This uses a stochastic update rule to help the network “jump” out of local minima. It can be seen as analogous to *simulated annealing*, which performs the same function for hill climbing [55].

2.7 Correlation Matrix Memories

Another example of a neural network based associative memory is the *correlation matrix memory (CMM)*. CMMs are examined extensively in this thesis, and for this reason we shall describe them in a little more detail than other associative memories in this review.

2.7.1 Description

The correlation matrix memory [58] is an example of a heteroassociative memory (although it can also be used in an autoassociative fashion). It is a single layer network, with the input and output neurons being fully connected. Therefore, the network can be viewed as a matrix of weights with size equal to the length of the

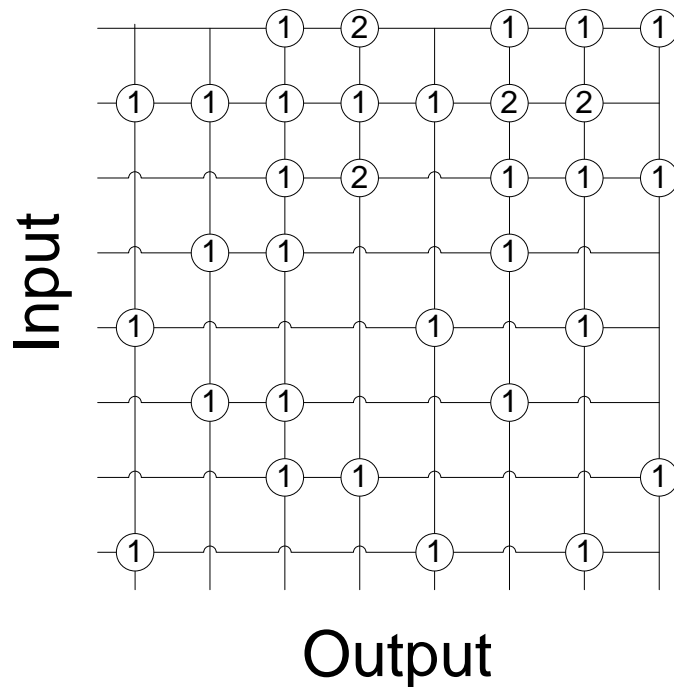


Figure 2.3: A correlation matrix memory with continuous weights

input vector multiplied by the length of the output vector, as shown in Figure 2.3. A CMM is capable of storing a large number of pairs of input and output data vectors, especially when considering it also offers properties such as fault tolerance and generalisation.

The weight matrix stores the correlations between input and output pairs which are stored in the network. There are multiple methods for learning this matrix. Some possibilities are to calculate the *pseudoinverse* based upon the input and output pairs, to use *gradient descent* to minimise the output error, or to use Hebbian learning [76] (as described in Section 2.6). Calculating the pseudoinverse is non-trivial and requires a complete set of input and output pairs, and therefore the entire weight matrix has to be recalculated in full every time a new pair is added to the network. This makes it an impractical approach in most cases. The gradient descent approach does not experience this same problem, but upon the presentation of any new pair the whole weight matrix will still require some adjustment. While the Hebbian learning approach does not minimise the error as effectively as the previously mentioned methods in general, in the case that the inputs are orthogonal it is no worse. In addition, the Hebbian approach only requires local updates to the CMM each time a new association is learnt. For this reason, this approach is commonly used as it enables learning to be performed extremely quickly.

In CMMs Hebbian learning utilises a standard binary representation using 0 and 1. A binary vector is presented at the input and the output of the network. For each pair of input and output nodes, if both nodes have the value 1, the weight of the connection which links them is increased by 1. This process is described by Equation 2.5 where W is the weights matrix, x is the set of input vectors, y is the set of output vectors, and z is the number of examples to learn.

$$W = \sum_{k=1}^z y_k x_k^T \quad (2.5)$$

The recall process is described by Equation 2.6. Essentially we calculate the product of the input vector and the weight matrix, producing an activity pattern on the output neurons. It should be noted that the term Wx_j is not a binary vector, and hence a thresholding function f is required to produce a binary output. The nature of this function will be discussed in a moment.

$$y = f [Wx_j] \quad (2.6)$$

This type of network has a number of benefits. Firstly, when the training is performed using simple Hebbian learning it is possible to train a CMM very rapidly. This is in stark contrast to error correction techniques such as back propagation [77], which uses an iterative procedure to reduce the output error and suffers from slow convergence. In addition, the recall operation is also very simple. Furthermore, these memories are able to display a robustness to noise on the input, enabling accurate recall for an incomplete input [13], and generalisation to unseen inputs. They also provide a relatively large storage capacity. These properties are examined in detail in Chapters 4 and 5.

2.7.2 Crosstalk

Because the data is stored in a distributed manner, when we recall data from the memory the activity returned is a mixture of the data that we wish to recall and other data that has been stored. If we attempt to recall the data associated with the stored vector x_j , we obtain the vector shown in equation 2.7.

$$y = y_j + \sum_{k=1, k \neq j}^m \cos(x_k, x_j) y_k \quad (2.7)$$

The term y_j is the stored vector we are trying to recall, whilst the other term is *crossstalk*. The magnitude of this term is defined by the similarity between the input vector x_j and the other stored input vectors. We can clearly see that if the set of input vectors are an orthonormal set then there will be no noise term, and hence perfect recall, since this would result in $\cos(x_k, x_j)$ always being 0. However, doing this would reduce the number of items we are able to store in the memory, and may even lose the advantage of distributed storage. For example, an orthonormal set from an input vector of size 3 would be 100, 010, 001. If this input set was used, each output pattern would be a row of the matrix, and so would not be distributed across it. In fact, the memory would be essentially acting as a listing memory. When distributing storage across a CMM it is capable of storing more vectors than the number of input neurons, as detailed in Chapter 4.

In practice then, it is important to use a distributed representation (multiple bits set to 1 in the input vectors), with the vectors being as close to an orthonormal set as possible. This reduces the noise on the output, and hence allows a larger number of pairs to be stored in the memory without error.

2.7.3 Binary CMMs

The learning algorithm given in 2.5 calculates continuous values for the CMM weight matrix. However, it is possible to make a network with binary weights, and a network of this type was suggested by Willshaw et al. [89]. In order to achieve this, we would learn as shown in equation 2.8, where \cup is the logical OR function.

$$W = \bigcup_{k=1}^z y_k x_k^T \quad (2.8)$$

Taking this approach has a number of advantages. Firstly, the storage requirement for the CMM is hugely reduced, with the exact factor depending on the number of bits used to represent each weight in the continuously weighted CMM. Secondly, having binary weights allows for an extremely efficient hardware implementation, such as in [54]. These benefits come at only a small cost to the efficiency with which the memory is able to store information, with the continuous weighted and binary

weighted CMMs being able to store information at 72% and 69% of the efficiency of a standard random access memory respectively [32] [89]. Unfortunately, because the use of an OR function rather than an addition effectively clips the weights it becomes very difficult to remove previously learned associations from the memory, an operation which is extremely simple in a continuously weighted CMM.

2.7.4 Thresholding and Representation

As previously mentioned, the recall operation in a CMM does not return a binary vector; the output activity must have a thresholding function applied to it in order to obtain the output. There are a number of different thresholding functions which can be used. The choice of function will depend on the application, and upon the representation used for the data which is being stored in the CMM.

One possible thresholding function for a binary CMM is to take the number of bits set to 1 in the input (known as the *weight* of the code), and to set any output which is equal to this value to 1 [89]. This is known as Willshaw thresholding. The method gives reliable recall when the input pattern is error free, since the activity in all the correct positions will equal this value. However, this method fails if the pattern used in recall differs from that used in learning [13]. This makes this form of thresholding very vulnerable to noisy input vectors.

A representation which has been used to great success with CMMs is *fixed weight* coding. This means that each data vector has a fixed number of elements set to 1. While this reduces the number of items which can be represented in a given vector size, it has great advantages for the performance of the CMM. Using this encoding we can use *L-max* thresholding [13], setting the l largest values in the output vector to 1, where l is the fixed weight of the output vectors. This enables improved recall performance in the presence of noise. However, it should be noted that there is an implicit assumption here, that each input code is associated with exactly one output code. In the case that multiple outputs are associated with the same input the number of bits on the output is unknown (since some bits may have been set in the same position in multiple output codes), so the value of l cannot be determined.

Not only is a fixed weight coding desirable, but optimal values for the code weights exist for any given code length. Palm et al. showed that optimal storage efficiency in a binary CMM is achieved using a sparse coding (a small number of bits set to 1 in a vector) [74]. Using such a coding ensures that the memory becomes saturated

more slowly than with a more dense coding scheme, although codes which are overly sparse reduce the degree to which information is distributed across the memory. The optimal weights of the input and output codes were shown by Palm to be $\log_2 n$ and $\log n$ respectively, where n is the length of the code [73]. These results are explored in more detail in Section 4.3.

Another facet to the representation used in CMMs is that in contrast to a conventional memory system which has a clearly defined limit to the amount of data it can store, a CMM shows a gradual degradation of performance as the weight matrix becomes saturated. The weight of the input and output codes has a significant effect on the storage capacity of a CMM. While there is not a clear point at which the memory becomes “full”, an understanding of the probability of recall errors based on the amount of data stored allows us to create memories of the appropriate size for a given task.

We see then that the capabilities of CMMs are dependant on the data representation used. It defines many properties of the network, in particular the storage capacity and the ability of the network to generalise. These ideas are further explored in depth in Chapters 4 and 5.

2.7.5 Weaknesses

Some weaknesses of correlation matrix memories should be highlighted. Firstly, since they are single layer networks CMMs are not able of solving linearly inseparable problems, such as the XOR problem. However, preprocessing methods (such as that mentioned in Section 2.9) or augmentations to the memory (such as mentioned in Section 2.10) can be used to overcome this limitation. Secondly, the size of a CMM is dependent on the size of the input and output data. This can cause practical difficulties, although again there are methods to overcome this (such as that detailed in Section 2.10). Another significant problem is the inability to determine when a CMM is full, since at this point the memory will simply return erroneous results. This problem is explored in detail in Chapter 4.

2.8 Bidirectional Associative Memory

The Bidirectional associative memory was introduced by Kosko [63], and is very similar in structure to a CMM as shown in Figure 2.4. The network is a single

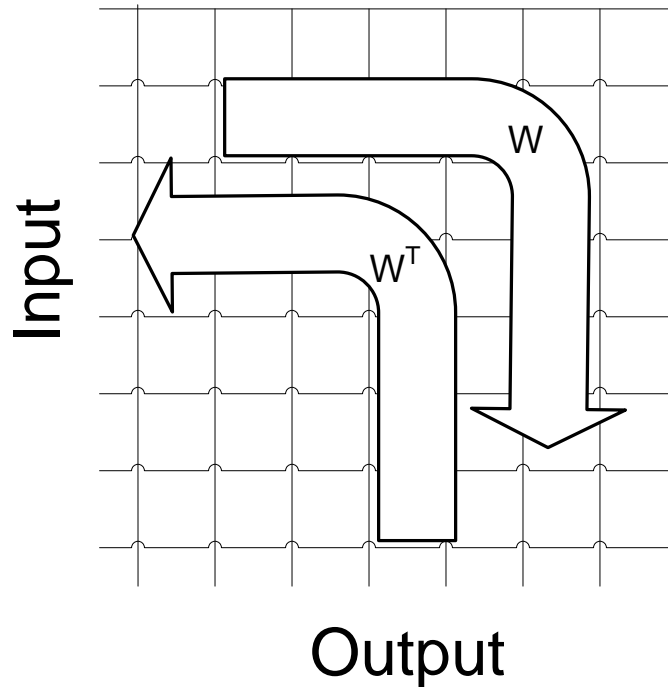


Figure 2.4: A bidirectional associative memory

layer heteroassociative memory, which is fully connected. The learning algorithm is identical to that for a continuously weighted CMM using Hebbian learning, as shown in Equation 2.5, although a bipolar representation is used rather than a standard binary representation. The major difference between the two networks is the recall operation, which introduces feedback from the output neurons to the input neurons. When an input is presented to the network, an initial output is produced at the output neurons using the learned weight matrix W , and a thresholding function. The values at the output neurons are then used to produce a new set of values at the input neurons, by passing through the transpose of the weight matrix, W^T and thresholding again. These new values are then returned forwards through the network using the weight matrix W , and so the process continues until a stable pair of states at the input and output neurons are reached. Kosko showed that the network will always converge to such a state.

This method should allow for recall even when there is significant noise on the input. Consider the case that we have some input x'_i , which is a noisy version of a true learnt input x_i . When this value is input to the network the output will be y'_i , and will likely be close to the true output y_i . The principal idea in a BAM is that when y'_i is used as an input through the transpose of the weight matrix W^T that the new value on the input neurons x''_i will be closer to x_i than x'_i was. As the recall procedure continues the input and output should be “cleaned up” until the correct

input/output pair are represented at the input and output neurons.

The BAM is similar in principal to the Hopfield network. The network is essentially performing a minimization over an energy landscape in a very similar way, but generalised to a heteroassociative case. The storage capacity of the network is not as large as a CMM, generally being smaller than $\min(m, n)$ where m is the number of input neurons and n the number of output neurons. Clearly a BAM also takes a greater amount of time to perform the recall process when compared to a CMM. However, it does offer additional robustness to noise.

2.9 The N-tuple method

The N-tuple method is a preprocessing technique first described by Bledsoe & Browning [18]. The method produces a fixed weight binary code, and so is suitable as a preprocessor for CMMs, amongst other networks. It was designed for image recognition, and was first demonstrated on a character recognition problem in the original paper.

The method works by grouping pixels in the input array into groups of size N , as shown in Figure 2.5. Typically, each group of pixels is chosen randomly, and are connected to an output address group. The address group calculates a mapping over the group of pixels which results in a single bit being set to 1 on its output. Each address group has an output of size 2^N bits. For example, for a two digit input code, the following mappings might be used; $00 \rightarrow 1000$, $01 \rightarrow 0100$, $10 \rightarrow 0010$, $11 \rightarrow 0001$. When the image is presented onto the input array, the mapping is performed at each address group, and the outputs are concatenated to produce the final code.

Performing preprocessing this way provides a level of generalisation in the system proposed by Bledsoe & Browning. When a character which is a variation of a previously seen image is presented to the input array the output should be similar as for the previously seen image itself, even if the variations include small scale, rotational or positional variation. This is primarily because the shape of any given character will ensure certain groups of pixels are in a certain state. The value of N affects the level of generalisation which is provided. As N increases the capability of Bledsoe & Browning's network to learn and recognise a larger number of examples of a given set of characters increases. However, the size of the vectors also increases, as well as their sparsity (which is an important consideration with CMMs as we shall

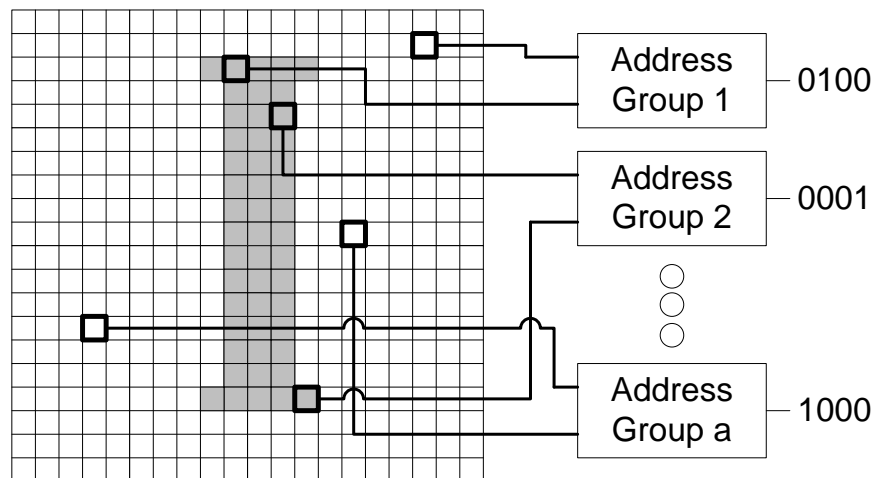


Figure 2.5: The N -tuple method, with $N = 2$

see in Chapter 4).

The N -tuple method is a general approach to code preprocessing, applicable to any form of input pattern. It provides a level of generalisation capability, and some positional, orientation and scale invariance in image processing. In addition, since the method performs a non-linear mapping over the input it also enables a CMM to solve problems which are not linearly separable.

2.10 Advanced Distributed Associative Memory

The Advanced Distributed Associative Memory (ADAM) [5] is primarily used with the N -tuple method in image processing problems, and was designed to address some of the weaknesses of the CMM network. Specifically, the size of a CMM is dependent upon the sizes of the input and output vectors. If these two values are large then the CMM will be very large also. In addition, the vulnerability of a CMM to crosstalk means that encodings which are close to orthogonal are required in order to maximise capacity. The ADAM network provides a capability to overcome these limitations somewhat.

ADAM essentially consists of two CMMs, connected via a third pattern in the middle, as shown in Figure 2.6. The additional intermediate pattern is known as the *class* pattern. When the input and output codes to be associated are presented to the network a class pattern is generated. This pattern is both unique and sparse, as well as being as close to orthogonal as possible from other class patterns. The two CMMs are then trained appropriately, with the first network learning the association

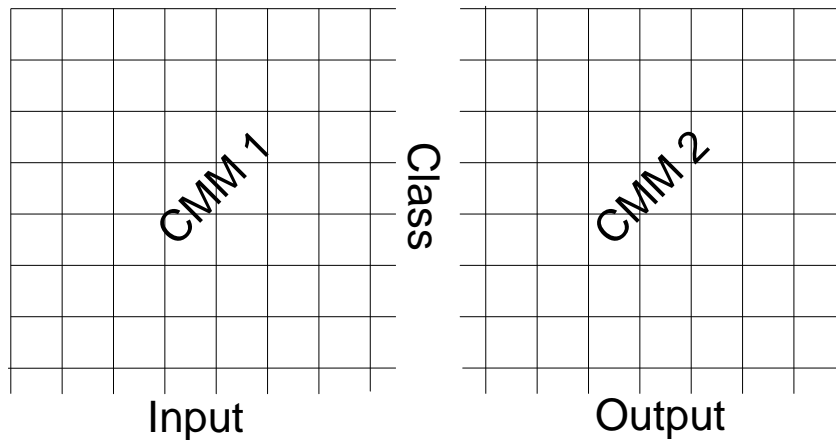


Figure 2.6: The ADAM network architecture

between the input pattern and the class pattern, and the second network learning the association between the class pattern and the output pattern.

Structuring the memory in this way has a number of benefits over a simple CMM. Firstly, if both the input and output vector lengths m and n were very large then the number of bits required for the CMM mn would also be very large. However, by constructing two CMMs with the intermediate class pattern the number of bits required becomes $mc + nc$. If $c \ll m$ and $c \ll n$ then this value will be much smaller than mn , hence greatly reducing the memory requirement for the network. In addition, because the class patterns are chosen to be close to orthogonal, the storage capacity should also be increased in cases where the input and output patterns do not have this property. Because the addition of an intermediate pattern effectively implements a two layer neural network, ADAM is also capable of distinguishing between linearly inseparable patterns.

ADAM has been applied to the recognition of features and textures in aerial photographs [80] and to texture discovery [79].

2.11 Advanced Uncertain Reasoning Architecture

The Advanced Uncertain Reasoning Architecture (AURA) [11] [12] was developed to provide a fast rule matching capability based upon correlation matrix memories. It allows the creation of rule based systems which are capable of reasoning with

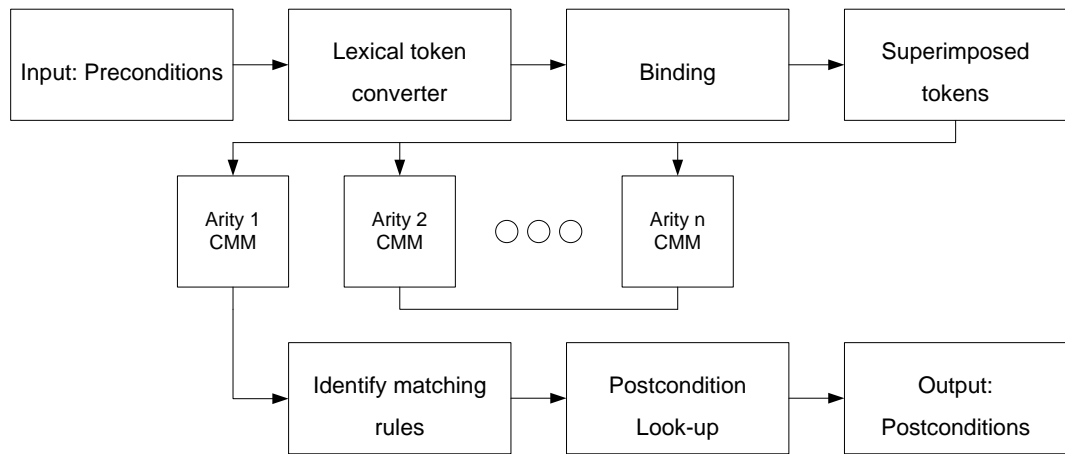


Figure 2.7: The layout of AURA

uncertain data, which can store a very large number of rules, can store rules very quickly and can perform very high speed reasoning. AURA allows the association of a number of preconditions to a number of postconditions, and is able to generalise when provided with incomplete inputs. Figure 2.7 shows the layout of the architecture.

Rules take the form of sets of preconditions and postconditions. Values in the preconditions and postconditions do not simply stand on their own, but are associated to *attributes*. This allows wildcard searches for given attributes. So, for example, a rule which allows the identification of a piece of fruit might be constructed as in Equation 2.9, with the notation Attribute : value:

$$\text{Colour : yellow} \wedge \text{Type : fruit} \wedge \text{Shape : curved} \rightarrow \text{Object : banana} \quad (2.9)$$

In this example, the rule contains three preconditions. If each of these is represented by a vector, how do we combine them into a single token to be input to a CMM? One option would be to concatenate the vectors together, but this means that the order of the attributes becomes important. A better option is to use *superposition*. This essentially means that the vectors are combined using a logical OR function, as shown in Figure 2.8. This preserves the commutativity of the inputs. So, in order to store the rules in AURA we take the tokens representing the preconditions and superimpose them; this is the input vector for the CMM. We then take the vector for the related postcondition, and associate this with the input vector in the CMM.

Up to this point the method for combining an attribute and a value into a vector has not been described. The process by which this is accomplished is called *binding*.

Pattern 1	1	1	0	0	1	0	0	0	0	0
Pattern 2	0	0	0	1	0	0	1	1	0	0
Superimposed Pattern	1	1	0	1	1	0	1	1	0	0

Figure 2.8: An example of superposition

Binding between attributes and values takes place through a binary tensor product, which is the cross product of the vectors representing the attribute and the value. For example, if the attribute 1001 is bound to the value 1100, the resulting matrix would be:

$$\begin{vmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{vmatrix}$$

This matrix can then be reshaped into a vector to be stored in the CMM. Using this process allows values to remain associated to attributes when stored in the CMM. In addition, if we wish to perform a recall where one of the attributes is unavailable we can simply bind that attribute to a vector of all ones. This will result in a “don’t care” recall for that attribute, returning all the appropriate vectors which match the other attribute value pairs in the input superposition.

There is a further issue which needs to be addressed with the architecture, which is that if all rules were stored in a single CMM there would be many erroneous recalls. In order to highlight the problem, Figure 2.9 shows a CMM which has learnt two rules. If we attempt to recall the input Colour : yellow from this network we might expect to get the result lemon, because all the preconditions for this rule have been met. However, the network returns lemon \wedge banana. Whilst this may seem to be fine on a cursory inspection, note that the rule which results in banana has the precondition Colour : yellow \wedge Shape : curved. Since the second part of this precondition has not been met, the rule should not be recalled. This problem occurs because both of the preconditions have been associated with the same output postcondition, and the partial match is returned as a full match. In order to overcome this problem, arity networks are introduced. This simply means

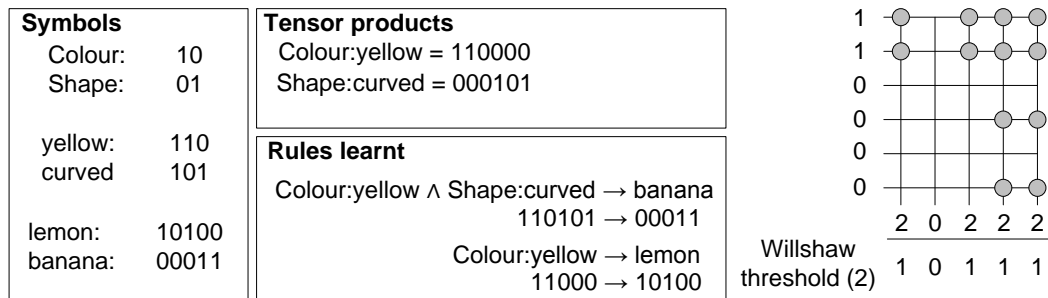


Figure 2.9: An example of storing different arity rules in one CMM. The network recalls both lemon and banana superimposed. However, only lemon should have been recalled, since both preconditions for banana were not met in the input.

that a separate CMM is used for each different rule arity. The arity is defined by the number of variable/value pairs present in the precondition. For the example given in Figure 2.9 the recall would be conducted on the arity one network, and would not make the erroneous recall we have observed. The multiple arity CMMs can be observed in the architecture shown in Figure 2.7.

The AURA approach does inherit the same limitations which occur for CMMs, in that the data representation must be carefully chosen and that rules cannot be easily removed once they have been trained. It does however offer an extremely powerful and rapid method for performing rule based reasoning, which can also be implemented efficiently in hardware [10].

2.12 Summary

After a discussion of the idea of associationism, we examined a theoretical definition of what a computational model of an associative memory would look like. Two paradigms for implementing associative memories were identified; content addressable memories and neural associative memories. The neural approach offers a robust solution which can learn and recall associations very quickly and in the presence of noise on the input. A variety of neural associative memories were described, with particular focus on the correlation matrix memory (CMM). This simple memory offers impressive storage capabilities and robustness to noise, providing an appropriate data encoding is chosen. In particular, the binary weighted version of the CMM is capable of a high storage capacity (as shall be seen in Chapter 4), as well as extremely rapid learning and recall, and efficient implementation in hardware. This memory is the subject of the remainder of this

thesis, and from here forward the term CMM shall refer to binary weighted CMMs specifically.

Chapter 3

Applications of Correlation Matrix Memories

3.1 Introduction

Having examined a variety of associative memories in Chapter 2, it would now be instructive to examine examples of how they can be used as part of a more complex system. In particular we will examine applications of correlation matrix memories (CMMs). CMMs have been applied to a variety of applications, including graph matching [56], spell checking [44] and expert systems [47]. In this chapter, two systems based upon correlation matrix memories will be examined in detail. The systems aim to solve two different pattern recognition problems; object recognition and rule chaining. Both systems store a set of rules in CMMs which are used to perform recognition. By examining these systems, we will gain a greater understanding of the issues involved with using correlation matrix memories in practice.

3.2 Cellular Associative Neural Network

3.2.1 Introduction

The Cellular Associative Neural Network (CANN) [8] is a hybrid architecture, combining elements from cellular automata, syntactic pattern recognition and

associative memory. Associative memory was discussed in Chapter 2, so this section begins by examining the remaining two elements; cellular automata and syntactic pattern recognition. The architecture is then examined in some detail, including extensions which have been applied to incorporate uncertain data.

3.2.2 Cellular Automata

A cellular automaton is a mathematical model of a physical system, introduced first by von Neumann [87]. It consists of an array of cells, each of which may take a finite number of states. Often the state will be binary, taking on only the values 1 or 0. These states update over a series of discrete time steps, according to a number of rules. These rules are all based on the state of each cell's *neighbourhood*, informally defined as the area around the cell. The cells which define the neighbourhood are chosen as part of the model. An obvious example for a regular shaped grid might be the 8 cells immediately adjacent to the current cell. The rules take the form of a look-up table. Each combination of neighbourhood state values will define a new state for the cell.

An example of a cellular automaton can be seen in Figure 3.1, based on the famous Game of Life [33]. The cells which are black are set to 1 and the cells which are white are set to 0. The neighbourhood is defined as the eight cells surrounding a given cell. In this case the rules that the automaton follows can be summarised as follows:

- If the current state is 1:
 - If zero or one neighbours are set to 1, the state becomes 0 (as if dying by loneliness)
 - If four or more neighbours are set to 1, the state becomes 0 (as if dying by overpopulation)
 - Each cell with two or three neighbours set to 1 remains as a 1.
- If the current state is 0:
 - Each cell with three neighbours set to 1 becomes 1.
 - Otherwise, the cell remains as a 0

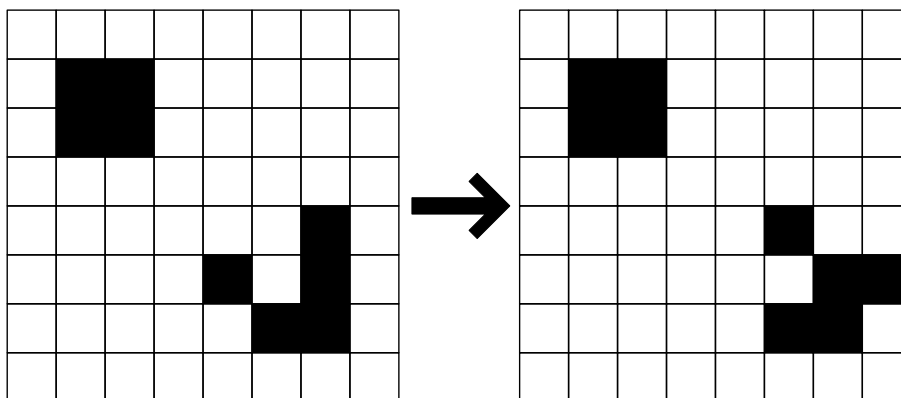


Figure 3.1: A single transition of a cellular automaton

Cellular automata are capable of showing complex global behaviours, despite being controlled through simple local rules. Indeed, it has been shown that they can be used as general purpose computers [90]. A particular benefit of cellular automata lies in their close correspondence to “single instruction, multiple data” (SIMD) hardware [69], a class of parallel computers which contain multiple processing elements that perform the same operation on different pieces of data at the same time. This means that it is possible to produce a very efficient hardware implementation of a cellular automata, enabling extremely rapid operation. For example, field programmable gate arrays (FPGAs) have been used to implement cellular automata in hardware [40]. Alternatively, modern graphics cards contain multiple programmable pixel shaders which can be used to implement such a system. However, designing a cellular automaton for a given purpose is a challenge, since the global effect of changes to local rules isn’t immediately obvious. This problem can potentially be solved by learning the rules through an algorithm though, as we shall see in Section 3.2.4.

Cellular automata (and architectures inspired by them) have been employed for a variety of applications. Fey and Schmidt propose an architecture for image processing in which each cell of the automata corresponds to a pixel in the image [29]. Their architecture is capable of detecting objects and their centre points. This architecture in fact bears much similarity to the CANN, described in Section 3.2.4. Another application is particle simulations, as demonstrated by Burstedde et al. [24] who used cellular automata to model pedestrian dynamics. They have also been used to perform image compression, data mining and fault diagnosis, demonstrating a low memory overhead and quick operation [66].

3.2.3 Syntactic Pattern Recognition

The field of pattern recognition can be broadly separated into three approaches; the statistical method, the model based (prototype matching) method and the syntactic method. The former approach uses statistical information in the data to perform recognition. An example of this would be the use of a multi-layer perceptron to recognise patterns based on features of those patterns [77]. Prototype matching uses a database of known examples of patterns, and performs recognition by finding the closest match to the input pattern in the database. An example of this would be the use the k-nearest neighbour algorithm [26]. Syntactic pattern recognition, on the other hand, uses structure which is present in the data in order to perform pattern recognition. In applying syntactic pattern recognition to an image, the implicit assumption is that there is there is clear structure to the patterns which can be recognised.

Syntactic pattern recognition is based on formal language theory, and recognition is performed by *grammars*. We would have one such grammar for each object to be recognised. According to Denning, Dennis and Qualitz [27] a formal grammar G is a four-tuple:

$$\mathbf{G} = (\mathbf{N}, \mathbf{T}, \mathbf{P}, \Sigma) \quad (3.1)$$

In this definition, \mathbf{N} is a finite set of non-terminal symbols, \mathbf{T} is a finite set of terminal symbols, \mathbf{P} is a finite set of productions and Σ is the sentence symbol (also known as the starting state). An important fact is that \mathbf{N} and \mathbf{T} are disjoint:

$$\mathbf{N} \cap \mathbf{T} = \emptyset \quad (3.2)$$

The starting state Σ is not a member of either the terminal or non-terminal symbol sets:

$$\Sigma \notin (\mathbf{N} \cup \mathbf{T}) \quad (3.3)$$

The productions P are rules which associate a group of non-terminals and terminals (with at least one non-terminal) to a new set of non-terminals and terminals. More formally, a production is an ordered pair of strings (α, β) where:

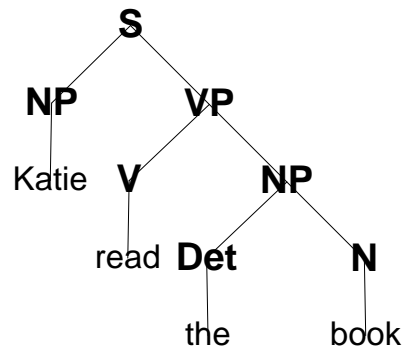


Figure 3.2: An example of parsing using a grammar

$$\alpha = \varphi A \psi \quad (3.4)$$

$$\beta = \varphi \omega \psi \quad (3.5)$$

Here, ω , φ and ψ are strings in $(\mathbf{N} \cup \mathbf{T})^*$, and A is either Σ or a member of \mathbf{N} .

Beginning in the starting state Σ , we can follow a series of these productions in order to arrive at a *sentence* of terminals. Any sentence which can be produced from the starting state Σ is said to belong to the *language* produced by the grammar. Recognition using a grammar is the reverse process to the one we have just described. The process could be termed as *recognising* the pattern; discovering whether a pattern belongs to a given grammar. Furthermore, we can attempt to find the derivation tree which produces the pattern; this is known as *parsing*. The tree has the starting state S at the root, and the pattern terminals at the leaves. Parsing can be top-down or bottom-up; top-down begins with the starting state and works down the tree to the terminals, while bottom-up parsing works from the terminals towards the starting state. An example of a parsing tree can be seen in Figure 3.2.

Of course, one of the largest issues in syntactic pattern recognition is how the grammar is constructed. A grammar can be constructed by hand, although this would be very difficult for a problem of any complexity. Often, the grammar is derived from a series of example patterns, potentially positive and negative examples of patterns which belong to the language. The problem of deriving the grammar from these examples is known as *grammatical inference*. This is a challenging task, and solutions tend to be application specific, sensitive to noise and computationally complex [83]. Indeed, these weaknesses can be applied to syntactic pattern recognition as a whole. Tanaka comments that syntactic pattern

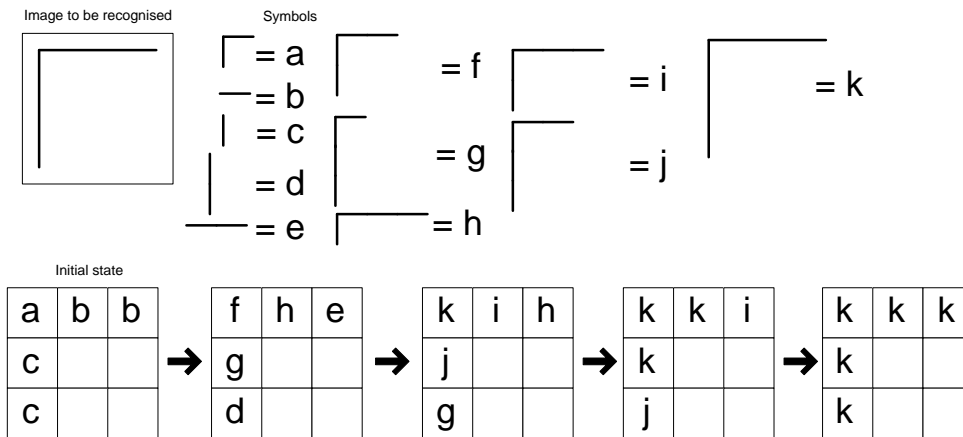


Figure 3.3: An example of the operation of a CANN

recognition suffers from high computational complexity and a lack of “expressive power” [82], citing the example of a famous chromosome pattern which cannot be described by a context free grammar [88]. Furthermore, Tanaka points out that non-terminals and productions in a generated grammar will not necessarily correspond to visual features or the relations between them. This makes the task of adjusting the grammar by hand extremely difficult. However, these weaknesses can perhaps be somewhat overcome by using syntactic pattern recognition as a constituent part of a hybrid approach, as we shall see in the following section.

3.2.4 The Static Architecture

The *Cellular Associative Neural Network (CANN)* [8] is a novel architecture for pattern recognition which has been developed based on cellular automata. The architecture has been successfully applied to image recognition [71], graph matching [56] and text processing [21]. As well as taking inspiration from cellular automata, the architecture incorporates elements from syntactic pattern recognition and associative memory. The CANN consists of a grid of cells containing *associative processors*, which are all identical. Each processor has a state, which at the start of a recognition would represent a basic feature of the item to be recognised. Each cell passes information about its contents to its neighbours, and these messages propagate further through the grid at further time steps. As cells become aware of the states of the cells around them, the states are updated to represent sub-patterns, and eventually objects. An example of this process is shown in Figure 3.3.

We saw in Section 3.2.3 that a pattern can be recognised through bottom-up parsing; that is, taking a group of terminals, applying a series of rules which transform them

into non-terminals until we eventually arrive at a single non-terminal, the starting state. The CANN model works in an analogous way. Instead of terminals we begin with a group of basic features. These features are transformed into intermediate labels (which can be thought of as the non-terminals) until we eventually arrive at an object label (the starting state). This represents a three level hierarchy:

$$\text{basic features} \rightarrow \text{intermediate labels} \rightarrow \text{object labels} \quad (3.6)$$

One problem with using a cellular automaton to do pattern recognition is the large number of rules which are required. This problem becomes worse when using a more complex rule based system such as that in the CANN. A very large rule set means that a large amount of memory will be required, and that learning and recalling rules will be slow. However, the AURA architecture introduced in Section 2.11 provides an efficient and elegant solution to this problem. Rules can be stored in CMMs, which means they can be learnt rapidly, and also recalled quickly and reliably. In addition, the storage capacity of a CMM is potentially very large, as we will see in Chapter 4.

In Section 3.2.3 it was highlighted that one of the problems with syntactic pattern recognition is the computational complexity of recognising patterns. Two aspects of the CANN architecture aim to solve this problem. The first is that the use of AURA memories to store and recall rules allows the process to be performed efficiently [6]. Secondly, being based on a cellular automaton, the architecture is highly parallel. This allows a very efficient hardware implementation, and also means the architecture can make the most of multiple processors or processor cores.

Another weakness of syntactic pattern recognition is that systems will show a lack of generality. The use of the AURA memory also helps to solve this problem. CMMs have an innate ability to generalise, and by careful selection of a threshold function a great deal of generality can be added to the system.

Each associative processor in the CANN has the same structure, as shown in Figure 3.4. There are three types of module; the spreader, the combiner and the passers. Each of these modules is based on an AURA associative memory, and the rules stored in each of these associative memories are the same in every cell in the CANN. Their functions are as follows:

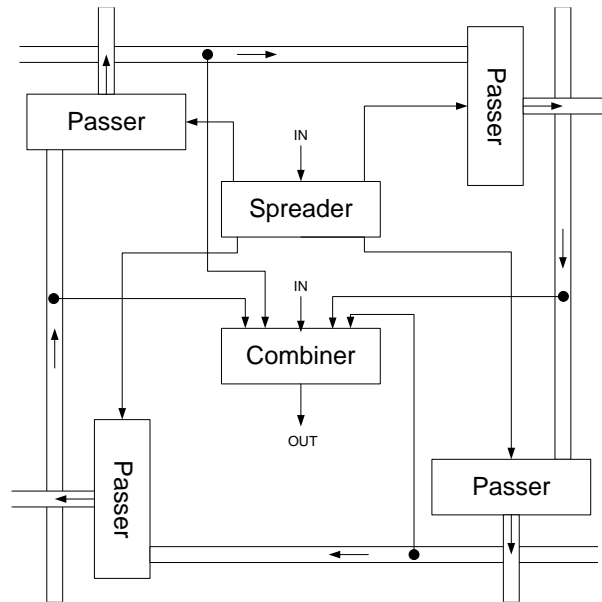


Figure 3.4: The architecture of an associative processor

- Spreader** The spreader takes the input (the current cell state) and adds to it information about where the information is coming from, so that it is ready to be passed to other cells.
- Combiner** The combiner takes the previous cell state and the information received from other cells and calculates a new state for the cell.
- Passer** The passer modules combine incoming messages with the cell state information from the spreader and pass it to the neighbours of the cell. They can also filter which messages are passed.

The architecture is flexible with regard to different neighbourhood definitions. The cell shown in Figure 3.4 would have four neighbours, but further passer modules could be added to include more neighbours. Consider the cell labelled 5 in the following grid:

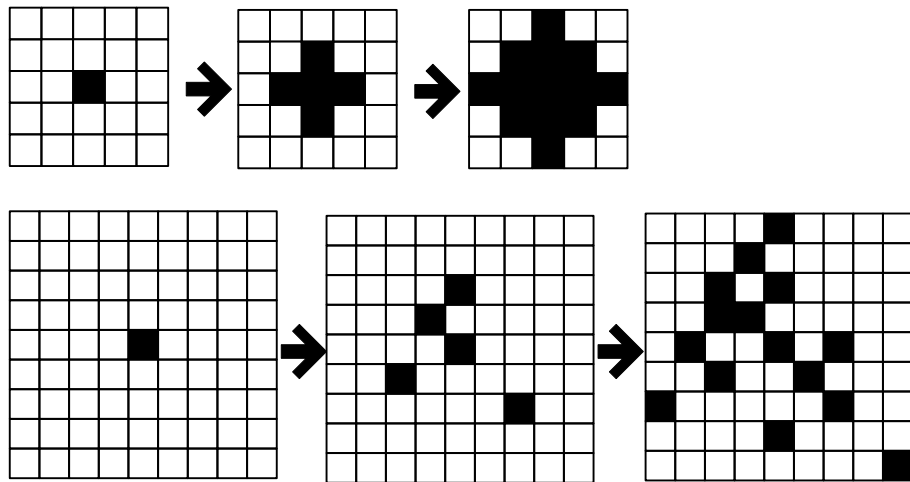


Figure 3.5: The spread of information through the CANN when using different neighbourhoods

1	2	3
4	5	6
7	8	9

Obvious neighbourhoods might be the cells labelled 2, 4, 6 and 8, or perhaps the cells labelled 1, 2, 3, 4, 6, 7, 8, 9, but this does not have to be the case. In fact, a neighbourhood can be defined as any set of surrounding cells, an arbitrary distance away. The neighbourhood controls the spread of the information through the network. Figure 3.5 shows the effect of using different neighbourhoods. A small, regular neighbourhood results in slow but dense information flow through the network. A larger, less regular neighbourhood results in a faster but sparser spread of information. Also, with a larger neighbourhood, there will be more routes for the information to reach a given point. This will increase the possible number of rules which could be applied.

Learning is performed through the following algorithm. First, the image is presented to the network as a set of low level features. These low level features must be obtained from the image through pre-processing. The network then begins to pass the information around. At each cell, if the input received already matches the precondition of a rule, then the answer is retrieved as the output. Otherwise a new intermediate symbol is created and assigned as the output, and a new rule associating the cell state to the new symbol is learnt. Through this method, new rules are created only as needed. Importantly, this rule is learnt by the corresponding associative memory in every cell in the network. This means that intermediate symbols can

be shared where they represent different parts of the same image. For example, an object might have two long vertical edges, and both of these can be described with the same set of rules. This learning process will continue until a predefined stopping condition is reached, at which point all cells associate their current state to a label representing the object being learnt. The earlier the learning is stopped, the more general the learning of the object will be. This is because each cell has only received information about a subset of the object, but will associate that subset with the whole object. On the other hand, if the learning continues until all cells have exchanged information then we will have achieved “rote” learning of the object. In this case, since each cell has become aware of the whole object, the object label will only be recalled upon a presentation of the exact object that has been learnt, and as such the system will be less capable of generalisation. Orovas suggests that a good strategy is to learn until each cell has a unique state [70], as this is an easily checked condition. Since high speed of operation is key to this system, this is a very important consideration. This method of learning certainly mitigates one of the concerns Tanaka [82] had with syntactic pattern recognition in general; namely that terminals and productions may do not correspond to visual features. Rules in the CANN directly express the relationships between features that are local to one another.

The recall process is similar to the learning process, except in the case that no matching rule is found we replace the learning of a new rule with a process of relaxation. Again, the image will be presented to the network as a set of low level features. At each iteration of the network, the inputs to the modules are checked to see if the preconditions of any rules are met. If a rule matches, then the output of the cell is set to be the postcondition of that rule. If no rules match then at this point the AURA memories may relax, depending on how much relaxation the user has allowed. Relaxation is achieved through a lowering of the thresholds of the CMMS, allowing matches to be achieved even in cases where the full conditions of a rule precondition have not been met. Essentially, the tolerance of the network can be gradually increased to whatever level the user has set as a maximum until a rule matches. If no rule can be found after relaxation, the output of the module remains as it was at the previous iteration. This recall process continues until a stopping condition is reached. This might be that there are no changes in the network after an iteration, that the number of changes in an iteration is below some threshold, or that a maximum number of iterations has been reached.

3.2.5 Incorporating Uncertainty

In the initial CANN model, a symbol was either present or not present. The CANN was further developed by Brewer [21] to deal with uncertain inputs. *Spiking* neural network memories were used to accomplish this, creating the Spiking Cellular Associative Neural Network (SCANN).

Information in the brain is not encoded as numeric values, as it is in traditional artificial neural networks. Bursts of energy are periodically emitted from each neuron, called spikes. Information is encoded in the rate of these spikes, or in the relative positions of the spikes. The model used in the SCANN is the Leaky Integrate and Fire (LIF) Neuron. This model uses an activity value, which increases whenever a spike is received at the input. The activity value gradually drains away over time. If the activity value goes over a set threshold, then the neuron will fire.

Such neurons can model uncertainty by relating firing rate to confidence. If a symbol is more likely then the neuron will fire very frequently. If it is unlikely then the neuron will fire only occasionally. By incorporating this type of neuron into the CMMs used in the architecture, a new model of operation is achieved. An example of a spiking CMM is given in [65].

A stream of low level symbols are input to each cell, and each cell then passes information to its neighbours in the form of streams of higher level symbols. As these symbols are received even higher level labels can be applied and so on. With this architecture there is no clear stopping point when recognising an object; the system can be run for any length of time. The accuracy of the output should stabilise and become more accurate over time. Also, if the input changes during operation, the system will adapt to the new input over time.

In order to apply the system to real images, a pre-processor is required. For this to work with the SCANN, the output of the processor needs to be in the form of spike trains. Brewer created a spiking edge detector for this purpose [20]. The system checks for a variety of *edge profiles* in each cell, and produces a spike rate for each. Areas which have a high contrast between pixel intensities (a strong edge) will provide high spike rates, and areas of constant contrast (no edge) will produce a low spike rate. The system has been shown to produce the expected results on real images.

3.2.6 Further work

This architecture is not without fault, and there are a number of remaining issues with the CANN architecture which require further work. We now examine some of these which have been identified:

3.2.6.1 Invariance

Ideally, object recognition systems should be translation, rotation and scale invariant; that is, objects should be recognised no matter where they are in the image, whatever their orientation and whatever size they are. While the CANN provides translation invariance, it is not scale or rotation invariant. While this problem could partially be solved by learning multiple copies of an object at different sizes and rotations, this would quickly become a very impractical solution.

3.2.6.2 Limitations in Learning

Although the SCANN is able to operate on uncertain data when performing a recall operation, the learning process can not. Real training data is likely to take the form of objects with a pre-processor applied to them (such as the edge detector we have just seen), and so an improved learning process would be able to incorporate this data. In addition, the learning process is not capable of learning multiple instances of the same object class. Incorporating this capability would potentially provide a system which was capable of improved generalisation.

3.2.6.3 Parallelism

There is also a serious problem with CANN architecture which has been “glossed over” up to this point. One of the largest benefits of the CANN architecture is that it is designed to operate in parallel. Each cell should be able to operate independently of all other cells, allowing them to execute on different hardware, giving truly parallel performance. In the case of recall, the architecture as described is indeed capable of this. However, there is an issue when learning is performed.

Firstly, recall that the rules stored in all the associative memories in the network are to be in common between all cells. This can be achieved in practice either by sharing a single version of each of these memories between cells, or by copying new

rules between all cells at the end of each iteration of the system. In the case of the former solution we cannot perform in parallel, because each memory can clearly only be in use by a single cell at any given time. In the latter case a more subtle problem presents itself. Consider the case in which two cells have the same state, and observe the same set of inputs from their neighbourhood during a single iteration of the system. If this combination of inputs has been previously unseen, then both cells will attempt to create a new intermediate symbol and to learn a rule with this symbol as the output. The question now arises; how is the code for this new intermediate symbol chosen? If it is taken from a central source then the system cannot operate fully in parallel, since all cells in the CANN will be attempting to access the same memory at the same time. However, if the source is local to the cell, how can it be guaranteed that the two cells choose the same code for this intermediate symbol? Either the symbol would need to be generated locally in a fashion that guarantees that the same code would be used for any two cells having the same state, or all cells must communicate with one another, greatly slowing down the operation of the system.

In fact, we can show that any solution to this problem which does not involve communication between the cells is doomed to failure. For the cell to operate entirely in isolation, we require a code to be generated for the intermediate symbol which will always be the same given a certain set of inputs to the memory. The state of the cell is defined as the set of all inputs to the memories in the cell at a given time step. Let us say that the number of inputs is defined as J . Furthermore, we require this operation to be undertaken in isolation from all other cells. In other words, we require a code generator in which the generated codes are a *function* of a given set of input codes. This function would also need to be *injective*, with each possible code being produced only by a single set of inputs. This is to avoid the problem in which two different symbols are assigned the same code. The size of the domain of this function is defined by the number of input codes which define the state of a cell. If the number of unique codes is defined as U , then the size of the input space is U^J . The size of the range of the function, however, is defined only by the number of codes which can be generated, in other words it is simply U . Clearly, we can see that such a function cannot exist for $J > 1$; there are not enough elements in the range for a full mapping from the domain. Since the combiner requires multiple inputs (the current cell state and at least one neighbour), it will always be the case that $J > 1$, and hence the function we require cannot exist.

Taking this into consideration, any possible solution to the parallelism problems

in the CANN needs to be focused on mitigating the problems with either sharing memories, or passing messages between cells.

3.3 Associative Rule Chaining Architecture

3.3.1 Introduction

The problem of rule chaining is one which is of particular interest in the field of neural networks, since it is a process which the brain must routinely perform. The Associative Rule Chaining Architecture (ARCA) presented in this section, put forward by Austin [9], attempts to perform rule chaining using CMMs to store and recall rules in a recursive fashion. The data uses a distributed representation, making use of tensor products and superposition to maintain very efficient use of space in the network, with the space required remaining constant. Learning is performed using simple Hebbian learning, as previously seen with CMMs.

3.3.2 Rule Chaining

Rule chaining is a common problem in artificial intelligence; that of applying a set of rules to a number of symbols in order to determine if any consequences can be applied. In this case we are only considering a very reduced logic, in which a state consists of a set of symbols which are asserted to be true. Rules take the form of one or more precondition symbols, paired with one or more consequence symbols. In this simple case, negation is not supported.

There are two forms of inference which can be applied in rule chaining; *forward chaining* and *backward chaining*. In forward chaining a set of symbols are defined as the starting state, and the rules of the system are searched. If the symbols in the precondition of any rules are all present in the current state of the system then the symbols in the consequence of the rule are added to the state, and the system then iterates. In addition to the rules, one or more symbols are defined as a goal state. If a goal state is reached then the search is complete. If the system reaches a state in which no further rules can be applied, and that state is not a goal state, then the search finishes in failure.

The alternative method for rule chaining is backward chaining. This method takes the desired goal state and searches for matching symbols in the consequences of the

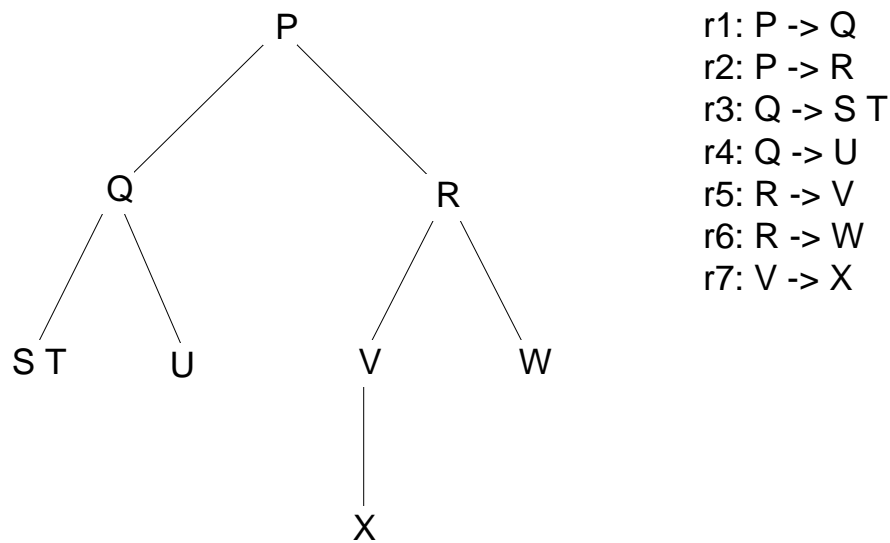


Figure 3.6: An example set of single arity rules, and the resulting search tree

system rules. The system state is then updated to include the preconditions of those rules. This process continues iteratively until either no further rules can be applied, or the state reaches a set of symbols which are known to be true.

The decision as to which chaining method should be used is application specific, with both methods having advantages and disadvantages. A discussion of this is beyond the scope of this work, but further information on this subject can be found in Russell & Norvig [78]. The implementation given here is of forward chaining, although there is no reason that the same techniques could not be used to implement backward chaining. What is important is that both forward and backward chaining are implemented through a depth-first search [78], as we shall see in a moment.

We will examine the case in which all rules are single arity; that is, each rule has exactly one precondition symbol. We will focus on rules of this type because it slightly simplifies the required system architecture (The reasons for this are explained in Section 3.3.11.2). Figure 3.6 contains a simple example of a set of rules of this type. If at any point in the search the symbol at the head of any rule exists in the current system state, it is replaced by the consequence symbols given at the tail of all rules for which it is a precondition. So, for example, if the symbol P is in the current state, it will be replaced in the subsequent state by the symbols Q and R . This results in a search tree such as that given in Figure 3.6. We wish to apply these rules to an initial system state in order to determine if any goal state can be reached.

For example, given the rules in Figure 3.6, imagine a starting state of P , with a

desired goal state of X . Determining if this is possible via a depth first search would involve finding first that P entails Q , which in turn leads to the states $S + T$ and U . Next we would search down the other subtree, finding first R , then V and finally our desired goal state X . The ARCA architecture is able to perform a search of this type, except it can search down these multiple subtrees in parallel.

3.3.3 Parallel Distributed Computation

Parallel distributed computation (PDC) is an idea presented by Austin, whereby a number of computations are distributed over a single neural network, without those computations being localised to any part of the network [6]. This idea is central to the implementation of ARCA. PDC is made possible in a correlation matrix memory due to the unique properties of the recall operation in the network; specifically, if we have a memory in which $A \rightarrow C$ and $B \rightarrow D$, a presentation of A and B (combined with logical OR) at the input will result in the recall of C and D at the output (also combined with logical OR). Exploitation of this fact allows multiple computations to be performed in a constant time operation.

A simple example of PDC can be demonstrated with a single CMM; the CMM has feedback from the output to the input of the network. In addition, the output can be tested to see whether any final condition has been reached. Consider a state machine, implemented using this simple architecture as shown in Figure 3.7. The state transitions are stored in the memory as associations, from one state to the subsequent states. For example in this case, the following associations will be stored in the network: $A \rightarrow B$, $A \rightarrow C$, $B \rightarrow D$, $C \rightarrow B$ and $C \rightarrow C$. The machine is initialized by presenting the starting state to the input of the network, in this case A . Since A has transitions to both B and C , both of these symbols are recalled by the network, combined in the output vector by a logical OR. This vector is then fed back to the input of the network, resulting in the recall of the symbols associated with B and C ; in this case B , C and D . Hence, we see that the processing is indeed parallel, with multiple state transitions being traversed in a single constant time operation. Since recall from a CMM is a highly parallel operation in itself, which can be efficiently implemented in hardware, this is potentially a very powerful observation. Furthermore, the computation is also distributed. Since the representations used in the CMM are distributed representations, the computation is not performed in a localised section of the network.

It should be noted that there is nearly always a trade off between time complexity

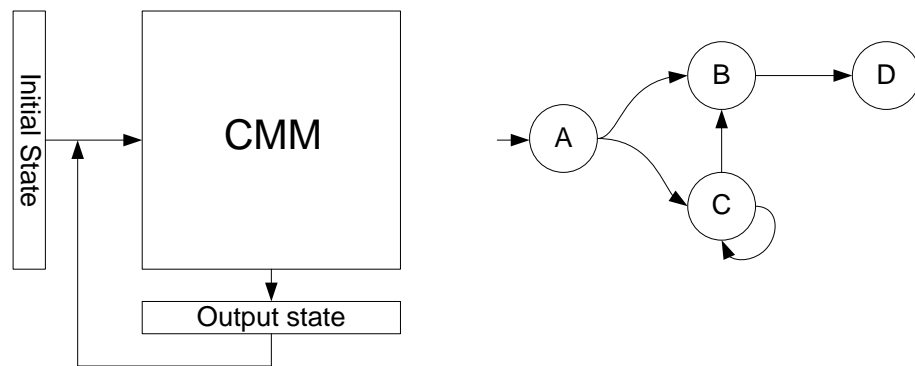


Figure 3.7: An example of parallel distributed computation: A state machine implemented in a correlation matrix memory

and space complexity, and that is no different in the case of PDC. The ability to perform multiple calculations in parallel requires a CMM large enough to store the required rules in order to perform PDC as described above. However, for a system with a rule set small enough to store in a memory this approach shows much promise.

3.3.4 Challenges

ARCA approaches the problem of rule chaining by applying a forward chaining approach, implementing the rules into correlation matrix memories and performing PDC on the system state. While this approach brings the benefits of CMMs, in terms of speed and storage capacity, it also presents its own challenges. Perhaps the primary challenge in achieving this goal is the representation of the state space. While in the simple PDC example outlined above there was only a single system state (the set of current nodes in the state machine), in this case we wish to maintain multiple branches of the search in parallel. Essentially, multiple nodes of the search tree must be stored within a single vector to be stored and recalled by correlation matrix memories, and separation maintained between them. This must be achieved while maintaining other requirements for the data to be stored in CMMs: in order for the recall process to be effective the data should be sparse and fixed weight. We shall see an elegant solution to this problem in Section 3.3.5.

A forward chaining search is not a linear operation in the general case. As we observed in Section 3.3.2, forward chaining is performed through a depth first search. Given a search tree with branching factor b (the maximum number of successors of any node in the search tree) and maximum depth m , depth-first search has modest space complexity of $O(bm)$. However, the time complexity is $O(b^m)$ in the worst case [78]. This is a potentially significant cost. By storing the entire state of the system

in a vector, and updating the state through only constant time recall operations, the ARCA architecture seems to offer the possibility to reduce the complexity of forward chaining to $O(m)$, eliminating the branching factor from consideration.

The space complexity is a more complicated issue; although the space used by the system is constant during operation, if insufficient space is pre-allocated then the memories will become saturated and the system will begin to fail. In other words, the space complexity could perhaps be defined by the amount of memory required to perform an error free recall. This is the cost of the ability to perform the rule chaining in linear time. The storage capacity of a CMM cannot be easily pre-determined, and will depend on the data that is stored in the network. This is problematic for a system such as this, because the point at which the system will begin to fail may be difficult to detect. We shall further examine this issue in Section 3.3.9.

3.3.5 Overview of the Associative Rule Chaining Architecture

ARCA uses two correlation matrix memories to perform rule chaining on the input symbols, as shown in Figure 3.8. These form a state machine, with the system in a constant feedback loop until either a goal state is found, or there are no further states to search. A number of states are entered as input to the precondition CMM. The output of the CMM is the rules which will fire given the input states. The rules are then given as input to the postcondition CMM, which outputs the new system states. If a goal state is found then the search stops, otherwise the new system states are input into the precondition CMM and the search continues. A defining feature of ARCA is its ability to maintain multiple system states in a data structure of constant size, and to search these states in parallel.

Before further describing the architecture, two important concepts need to be recalled from Section 2.11. The first of these is the idea of *superposition*, which is key to the ability of the architecture to process multiple states in parallel. Superposition involves packing multiple vectors into a single vector, by combining them with a logical OR. This operation is represented using the “+” operator. For example, consider the vectors 101000 and 000110. These codes can be superimposed to give the vector 101110. A CMM can process inputs combined in this way, and will output an OR of the resulting outputs for each of the constituent vectors in the input. For example, let us say that we have a CMM which has been trained with the associations $A \rightarrow X$ and $B \rightarrow Y$. If we input $A + B$, then the output of the

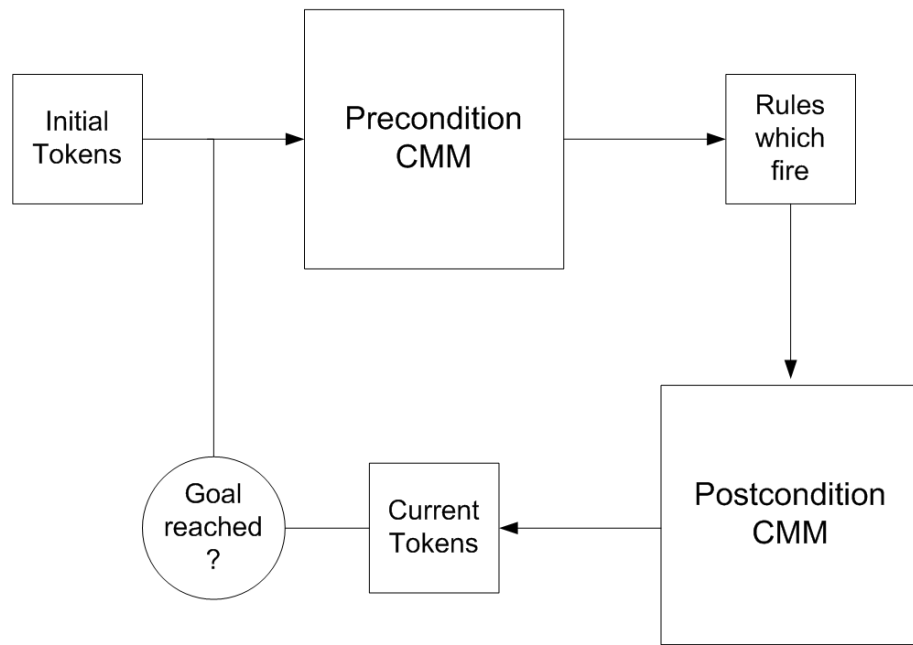


Figure 3.8: The Associative Rule Chaining Architecture (ARCA)

CMM will be $X + Y$.

The second concept to recall is the idea of *binding*. This term is used to describe the use of an outer product to combine two vectors together, producing a matrix called a *tensor product*. This operation is denoted using “:” as an operator. Hence, $A : B$ represents the binding of the variables A and B . For example, if $A = 1001$, and $B = 1100$ then:

$$A : B = \begin{vmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{vmatrix}$$

Having revisited these concepts, we can proceed to examine ARCA in further detail. There are two data types in the system, which we will term tokens and rules. Both of these are represented using fixed weight binary vectors which, as we have previously seen when using CMMs, allow for improved recall performance when compared to variable weight codes and for distributed coding within the memories. Following the example given in Figure 3.6, let us consider a search from an initial state P looking for a goal state X . In this case, the tokens are P, Q, R, S, T, U, V, W and X . Each rule also has a vector associated with it, and hence our rules are those labelled in Figure 3.6; $r1, r2, r3, r4, r5, r6$ and $r7$.

The concept of keeping multiple branches of the search concurrently within a single matrix warrants further explanation. Figure 3.9 contains a method for visualising the matrices used within ARCA. We know that the binding between a rule vector and any other vector will result in a matrix in which the only columns which will contain non-zero values are those where the rule vector was set to 1. This means that the number of non-zero columns in the matrix will be equal to the weight of the rule vector. Since the rule vectors are fixed weight this number is both constant and known. Assuming a fixed weight of 2 (for the sake of simplicity) we can visualise the matrices as shown, with each vertical line representing a column of data, labelled with the contents of that column. The positions of these columns are defined by the rule vector that they were bound to, and this is labelled at the base of the columns. The final matrix in Figure 3.9 particularly demonstrates how multiple states are maintained within a single matrix, with the two tokens Q and R stored as separate branches of the search. Since they are bound to different rules, the tokens are separated within the matrix.

The system is initialised with all the initial tokens bound to a rule¹. In this case we initialise the system with $P : r1$. Each column in this matrix is then input into the precondition CMM individually, and the outputs are formed into a matrix again at the output. This gives a matrix of the rules which will fire, bound against a set of rule vectors. In this case, the output matrix will be $r1 : r1 + r2 : r1$. That is, the two rules that fire are $r1$ and $r2$, and both of these are bound to the same rule which P was bound to, $r1$.

Each column of this matrix is then input in turn into the postcondition CMM. This CMM outputs a series of tensor products between rules and tokens, each representing the new tokens in the system, bound to the rule which produced them. These tensors are then superimposed upon one another. In this example, this means that the tensor $Q : r1 + R : r2$ is produced. Note that the produced tokens are bound to the rule that produced them. This simple fact means that even though the data is all stored within a single matrix, the separate branches of the search remain separate, allowing the concurrent processing. The system then loops, with $Q : r1 + R : r2$ input to the precondition CMM. Figure 3.10 shows the complete trace of the example we have been considering.

¹In the case of the initial tokens, the rule vector chosen is not important. The reasoning for this is given in Section 3.3.7



Figure 3.9: A visualisation of the columns within the matrices in ARCA. The matrices contain columns of various types bound to rule vectors. These columns are labelled at the top with the tokens contained within that column, with the remainder of the matrix containing zeros. The positions of the columns are defined by the rule token they were bound to (in this case of weight 2, hence each column appearing twice), and this token is labelled at the base of the column.

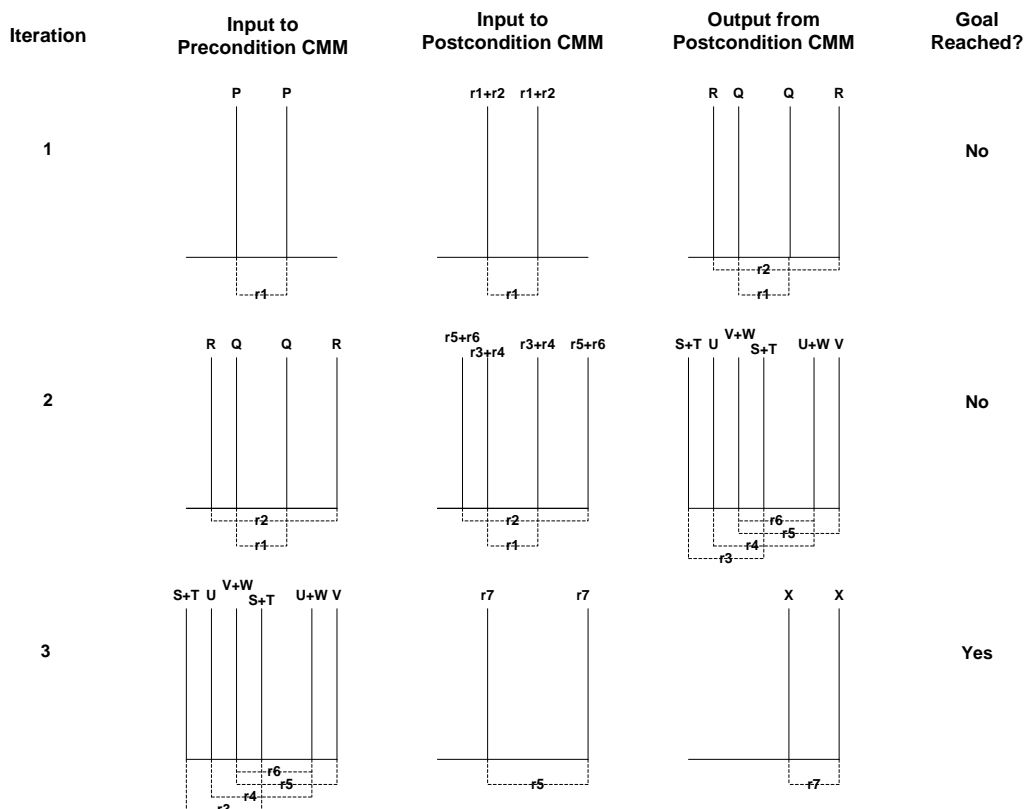


Figure 3.10: An example of ARCA running on the rules given in Figure 3.6, with a starting token P and a goal state of $\{X\}$.

3.3.6 Learning

The learning algorithm for ARCA is actually very simple. Each rule is labelled separately, and code vectors are generated for these rules and for the tokens within the rules. We will call the length of these tokens n_r and n_t respectively. We then associate the head of each rule to the corresponding rule vector, storing these associations in the precondition CMM. This means that the tokens in the precondition of each rule are associated with the rule which should fire if the tokens are present.

The training of the postcondition CMM is slightly more complex, and for each rule the following must be accomplished. Firstly, we create a tensor of the rule vector, and the tokens in the postcondition of the rule. This involves performing a superposition of the postcondition tokens, and forming the tensor product of this and the rule vector. This matrix is then considered as a vector with length $n_t n_r$, and is associated with the rule vector in the postcondition CMM. It should be noted that what is happening here is that the rule vector is being associated with a “pre-bound” tensor of the postcondition tokens bound to the rule vector itself. This means that when the precondition CMM determines that a rule should fire, the postcondition CMM will produce the tokens which will result from that rule, bound to the rule that fired them.

3.3.7 Recall

The recall process is initialised with one or more initial tokens, input by the user, alongside one or more goal tokens. The initial tokens are bound to a rule vector; the choice of rule vector here is not important. Recall that the purpose of the binding to rule vectors is to maintain separation between different concurrent branches of the search; since in the case of the initial tokens we have only a single state of the search to be concerned with, the binding is not important. After performing this binding, we are left with a tensor of size $n_t \times n_r$. We will use a very simple example to illustrate the recall process, which is illustrated in Figure 3.11 The example has three tokens, which are defined as follows:

$$\begin{aligned} A &= 101000 \\ B &= 010010 \\ C &= 100001 \end{aligned}$$

Furthermore, the system has two rules, labelled $r1$ and $r2$:

$$\begin{aligned} r1 : A \rightarrow B &= 10001 \\ r2 : A \rightarrow C &= 01010 \end{aligned}$$

So, in this case $n_t = 6$ and $n_r = 5$. Taking the vectors which we have just defined, if we initialise the system with the starting token A , and choose (arbitrarily) to bind it to $r1$, we bind A and $r1$ to achieve this input matrix:

$$\begin{matrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{matrix}$$

The next stage in the recall process is to find which rules have heads matched by tokens in the current state of the system. In order to accomplish this, each column of the tensor is input to the precondition CMM in turn. The CMM will output a vector of size n_r . Since there are n_r columns in the tensor, this means that at the output of the precondition CMM we will have n_r columns of size n_r , giving us a new matrix of size $n_r \times n_r$. Since data is distributed across the input tensor, each column does not simply contain a single token, but the superposition of potentially many tokens. However, since a superposition of inputs input to a CMM produces a superposition of their associations at the output, each new column is simply the superposition of all the rules which match the tokens in the input column. Now, since the order of these columns is maintained from input to output, the bindings are also maintained. This means that the process we have just outlined results in a matrix of rules, with rules which originated from separate paths of the search bound against different rule vectors.

Let us make this clearer by continuing the example. Taking each column of the matrix given previously, and inputting it to the precondition CMM gives the following matrix:

$$\begin{matrix} 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{matrix}$$

In this case, since both rules $r1$ and $r2$ match the token A both columns 1 and 5 contain the superposition of the matching rules $r1$ and $r2$. Each column from this intermediate matrix is now input into the postcondition CMM. Recall that the postcondition CMM has an output size of $n_t n_r$, outputting what is effectively a reshaped matrix of tokens bound against rule vectors. We input each of the columns from our intermediate matrix, giving a new matrix of size $n_t n_r \times n_r$. In this matrix each column is the superposition of potentially many pre-bound tensors of tokens, bound to the rules which fired to produce those tokens. This is perhaps best illustrated by continuing the example (the following matrix is shown transposed, for ease of display):

```

0 1 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0 1 0

```

We now wish to reduce this matrix to a single binary column vector, containing all the bindings which are present in the columns in the matrix. There are two obvious methods which could be used to achieve this; a logical OR, or a sum and threshold approach. The latter approach was suggested by Austin [9], and has been demonstrated to be the superior alternative [23]. In this case the threshold will be equal to the weight of the rule vectors since we want to preserve anything which has been bound to a rule vector. Anything which has been bound to a rule vector will appear in the matrix a number of times equal to the weight of the rule vector. So, in our example, a sum across the columns produces the following vector (again, transposed for ease of display):

```

0 1 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0 1 0

```

This vector can now be reshaped into a matrix of size $n_t \times n_r$, which in our continuing example results in the following matrix:

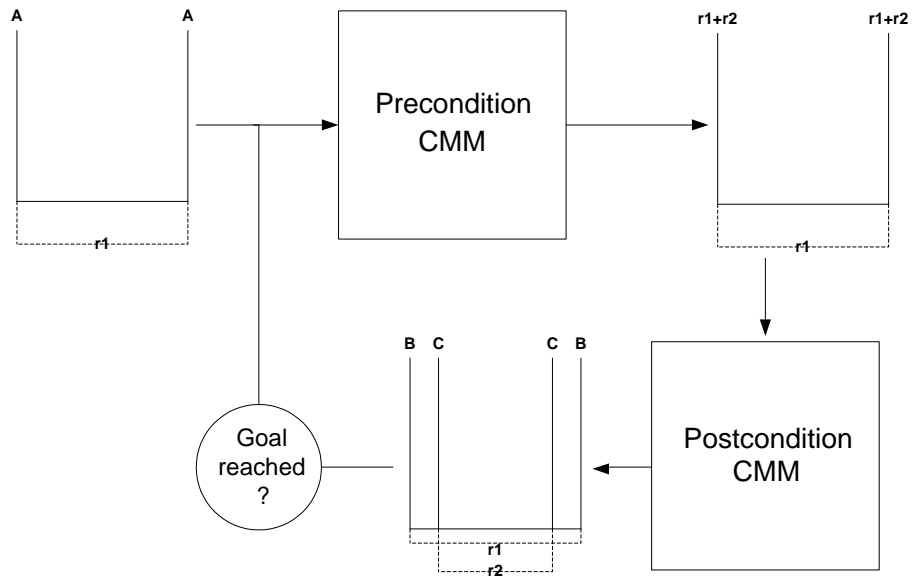


Figure 3.11: A single iteration of a recall operation in ARCA

0	1	0	1	0
1	0	0	0	1
0	0	0	0	0
0	0	0	0	0
1	0	0	0	1
0	1	0	1	0

Note that in the above matrix we can observe that we have $B : r1$ and $C : r2$, precisely the result we would expect.

Before the system iterates, there are two conditions which must be checked. Firstly, we must check whether any rules have been matched, and therefore whether the search needs to continue. Secondly, we must check whether a goal state has been reached. The former is achieved by observing whether the matrix above contains any non-zero values. If the matrix consists of all zeros then no patterns have been recovered, and so the search is completed, with no goal state reached.

The latter can be achieved by treating the above matrix as a CMM, with a superposition of all the goal tokens treated as an input, and the threshold set to the product of the number goal tokens and weight of those tokens. If the resulting binary vector contains any rule token then we can conclude that the goal state has been reached. This works because it indicates that the rule token was bound to the goal token.

3.3.8 Robustness

As has been previously mentioned in Section 3.3.4, one of the issues with ARCA is that the use of CMMs means that the point at which the system fails is difficult to determine. As the precondition and postcondition CMMs become increasingly saturated, the amount of crosstalk on the output increases. This eventually means that thresholding of the output activity regularly results in an incorrect recall, and the system begins to fail. The point at which this becomes a serious problem is difficult to define, and depends on a number of factors. Firstly, the point at which the CMMs will become saturated is difficult to determine, and will depend upon the representation used and the rules stored. Secondly, even as the memories begin to produce errors, the system may continue to operate correctly. Let us examine the reasoning behind this.

Willshaw thresholding is used in both the precondition and postcondition CMMs. Since more than one fixed weight code can be associated with a given input, L-max thresholding is unsuitable as outlined in Section 2.7.4. Willshaw thresholding has the benefit that the bits associated with the input code are guaranteed to be set in the output. However, further bits may also be set. This means that as the memories become saturated, an increasing number of incorrect bits will be set in addition to the correct output bits. These codes will then be used as input to the subsequent memory (be that either the precondition or the postcondition CMM). Because the extra bits which have been set are unlikely to fully correlate with any particular input code, these extra input codes are unlikely to produce extra bits at the output. Essentially, the feedback system between the memories should be capable of “cleaning” the erroneous output codes which begin to occur, at least up to a point. Clearly as the memory becomes very saturated and the number of incorrect bits set becomes large the chances of the extra bits corresponding to an input code on the subsequent memory increase.

All of this means that the robustness of ARCA is extremely challenging to analyse. For this reason, Section 3.3.9 focuses on performing simulations to examine the capabilities of the system in practice.

3.3.9 Experiments

In order to better understand the performance of ARCA in practice, a number of experiments have been performed. A simulation of the ARCA architecture has been

implemented in MATLAB, and applied to a variety of problems. The experimental methodology involved generating a set of rules which form a tree with a given depth m and maximum branching factor b . This allows some comparison between the system and depth first search. These rules were then trained into the architecture, and rule chaining was performed upon them.

The experiments have been performed for a range of values for the depth m and branching factor b of the search tree, and for the memory required to implement the CMMs in ARCA. This allows an examination of the required memory usage of the system, and how it alters as m and b are varied. It should be noted at this point that the fact that a MATLAB implementation was used in the experiments has limited the sizes of CMM which can realistically be operated upon, due to restrictions in terms of memory requirements and running time. This means that the largest CMMs used in the experiments would require only 2MB of memory if they were optimally implemented, with each weight represented by only a single bit. Nonetheless, the experimental results give a good idea of the relationship between m , b and the memory required by the architecture.

The specifics of the generation of the rule sets in the experiments is important. Trees were constructed in an iterative manner, beginning with the root symbol, which was also defined as the starting token for the recall process. Further layers of the tree were then added, with the total number of layers being equal to m . In the simple case of $b = 1$ this simply produces a chain of rules $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$ etc. In the case of a value of b greater than one, the decision was taken to randomly select the number of children which any given node has, sampling uniformly from the range $[1, b]$. This gives a tree with a maximum branching factor of b which should be more realistic than one in which all nodes have exactly b children. In addition, this method produces trees which are of a more reasonable size.

A range of values of m and b were selected for examination, along with a range of values of memory demands for ARCA E . E is defined by the token and rule vector lengths n_t and n_r , with the required memory in bits being $n_r n_t + n_r^2 n_t$. In order to simplify the experiment, the weight of the token and rule vectors was set to $\log_2 n$ (rounding down), where n is the vector length. This value gives a sparse representation which should provide good performance in the CMMs. For each selected value of m , b and E the following experiment was performed:

1. Generate a rule tree with depth m and maximum branching factor b .
2. Train ARCA with the generated rules.

3. Take the root of the rule tree as the starting token and a token in the bottom layer of the tree as the goal token.
4. Perform recall in ARCA with the given starting and goal tokens.
5. Note whether the recall was successful.
6. Iterate the previous steps 100 times.

This process then gives a success rate for recall in ARCA for a given combination of m , b and E . Determining whether the recall is successful is not a trivial task, since the system does not “fail” in a traditional manner. In many cases, the system will arrive at the required goal erroneously, due to over-saturation of the CMMs causing “ghost” recall values. In these cases the system could be viewed as providing the right answer for the wrong reasons. In this experiment a recall was defined as successful if and only if the goal token was found at the correct depth (i.e. after m iterations of the system).

3.3.10 Results

The graphs in Figures 3.12 and 3.13 are contour plots showing the recall error rates for the ARCA architecture for a given depth of search tree and memory requirement. Figure 3.12 shows the cases where the maximum branching factor is 1 and 2, and Figure 3.13 shows the cases where it is 3 and 4. As an initial observation, note that once the system begins to fail the performance falls off extremely rapidly as the depth of the tree increases. Even in the case where the branching factor is only 1 it only takes a very small increase in the depth of the search tree to take the method from experiencing only a tiny level of error to over 90% error. This suggests that there is a clear “breaking point” for the method, which is in contrast to the more general case of CMM storage in which performance decreases very gradually (as we shall see in Chapter 4).

Examining the case where the branching factor is 1 (Figure 3.12, top) the first observation one might make is that there is considerable structure in the performance changes, in particular a sudden fall in performance just above the memory requirement of 0.2MB. This can be attributed to the method chosen for selecting code weights, which was to take $\log_2 n$ (rounding down) as the weight for a code of length n . The decrease in performance occurs as the weight of the code increases, moving to a code length of 64 and therefore a weight of 6. The previous

code had weight of 5 whilst only being very slightly shorter, and therefore was a more sparse code. It would seem that the less sparse code has caused a slight drop in performance at this point. Moving onto a more general analysis of the graph, a disappointing result is that the the memory required to achieve correct recall appears to increase slightly more rapidly than linearly with respect to the depth of the tree. This can likely be attributed to the size of the post-condition CMM, and the choices made in the design of this experiment. Specifically, for a token code length of n_t and a rule code length of n_r the number of bits required to represent the post-condition CMM is $n_t n_r^2$. As such, it is heavily dependent upon the value of n_r . It may be the case that better results are possible if n_t is increased and n_r is decreased. The relationship between these values currently remains unexplored. In this experiment n_r and n_t were simply given the same value, which may have resulted in a less efficient use of space than possible.

In addition, the weight chosen may not be optimal, which could also cause the slightly worse than expected performance. As mentioned in Section 2.7.4, when attempting to maximise storage capacity of a CMM the optimal weight for an input code is $\log_2 n$, but for an output code it is $\log n$ (The theoretical results behind this are examined in more detail in Chapter 4). In a system such as ARCA where codes are used as both input and output codes, this is problematic. It would seem to be a reasonable hypothesis that optimal value will lie between $\log_2 n$ and $\log n$. In work by Graham & Willshaw [37] the relationship between these values is explored for a large CMM, with input and output code size $m = n = 2^{18}$. It can be observed in this case that storage capacity falls off particularly quickly with increasing output weight, more-so than with decreasing input weight. This might suggest that the optimal weight for this case may lie closer to $\log n$ than $\log_2 n$, and that perhaps this offers a better starting point for future investigations.

The results for the branching factors 2, 3 and 4 (Figure 3.12, bottom and Figure 3.13 top and bottom) show the massive decrease in performance which occurs as the branching factor increases. This is to be entirely expected given the explosion in the number of rules and tokens which occurs. The exponential nature of this increase is very visible in the increase of the incline of the 1% error contour as the branching factor increases. By the time the branching factor reaches 4 we are only able to achieve reasonable results for trees of depth 4 for the ranges of code lengths used in these experiments, simply due to the huge numbers of tokens and rules which are produced even at this depth. This problem is similar to that experienced by traditional methods such as depth first search, as seen by the time complexity of

that method, $O(b^m)$. In this case, however, the explosion in the number of rules results in extreme memory requirements rather than running time.

3.3.11 Further work

There are a number of unanswered questions and potential extensions to ARCA which have been identified. These are outlined below:

3.3.11.1 Effect of rule and token sizes and weights

The choices of the length of the codes used to represent rules and tokens (n_r and n_t respectively) are very important in the use of ARCA. In particular, the memory requirement of the system is defined by these values as $n_r n_t + n_r^2 n_t$ bits. This memory requirement is dominated by n_r , so ideally this value in particular should be kept as small as possible. While both of these values must be large enough so that the tokens and rules in the system can be represented, further work is required to understand the effect that varying these lengths relative to one another has on recall performance.

In addition, the choice of weights when codes are used as both input and output codes, as they are in ARCA, is problematic. This is because the optimal weight for input and output codes in CMMs are different, as will be further explored in Chapter 4. In ARCA the output to the postcondition CMM is the tensor of rule and token vectors, further complicating the issue. Investigations into the optimal choices of weights in the case of ARCA should be conducted.

3.3.11.2 Dealing with multiple preconditions

In this examination of ARCA we have only considered the case where each rule has only a single symbol as the precondition; arity one rules. This enabled a clear and simple examination of the architecture, but there is no reason why the system cannot handle rules with more than one symbol at the head of the rule, so long as a maximum number is predefined.

In order to implement this extension to the system, the introduction of “arity networks” is required. As discussed in Section 2.11, it is not possible for a CMM to perform recall correctly if rules of multiple arities are stored in the same network.

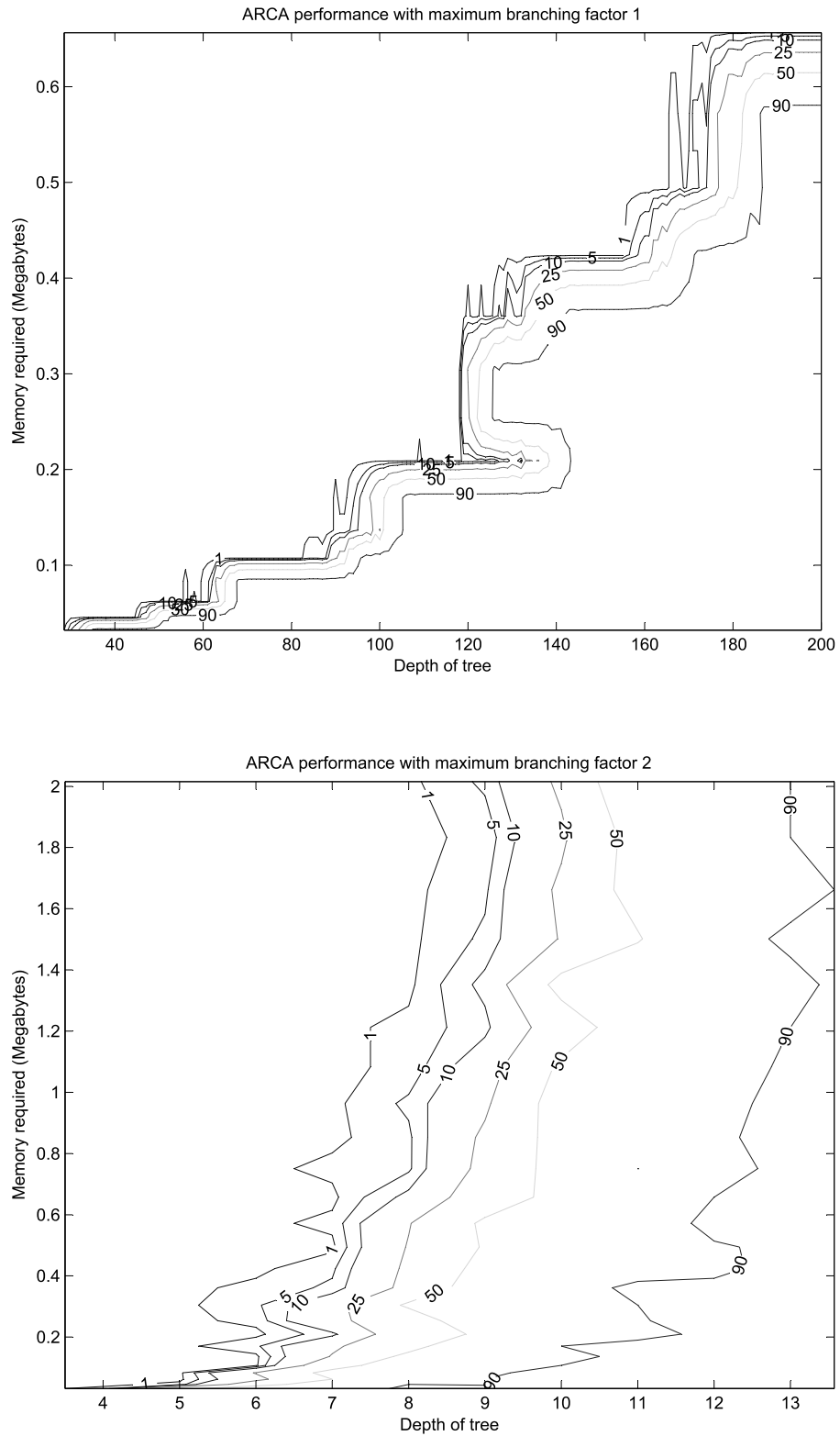


Figure 3.12: Contour plots showing the recall error performance for ARCA where the branching factor is 1 (top) and 2 (bottom)

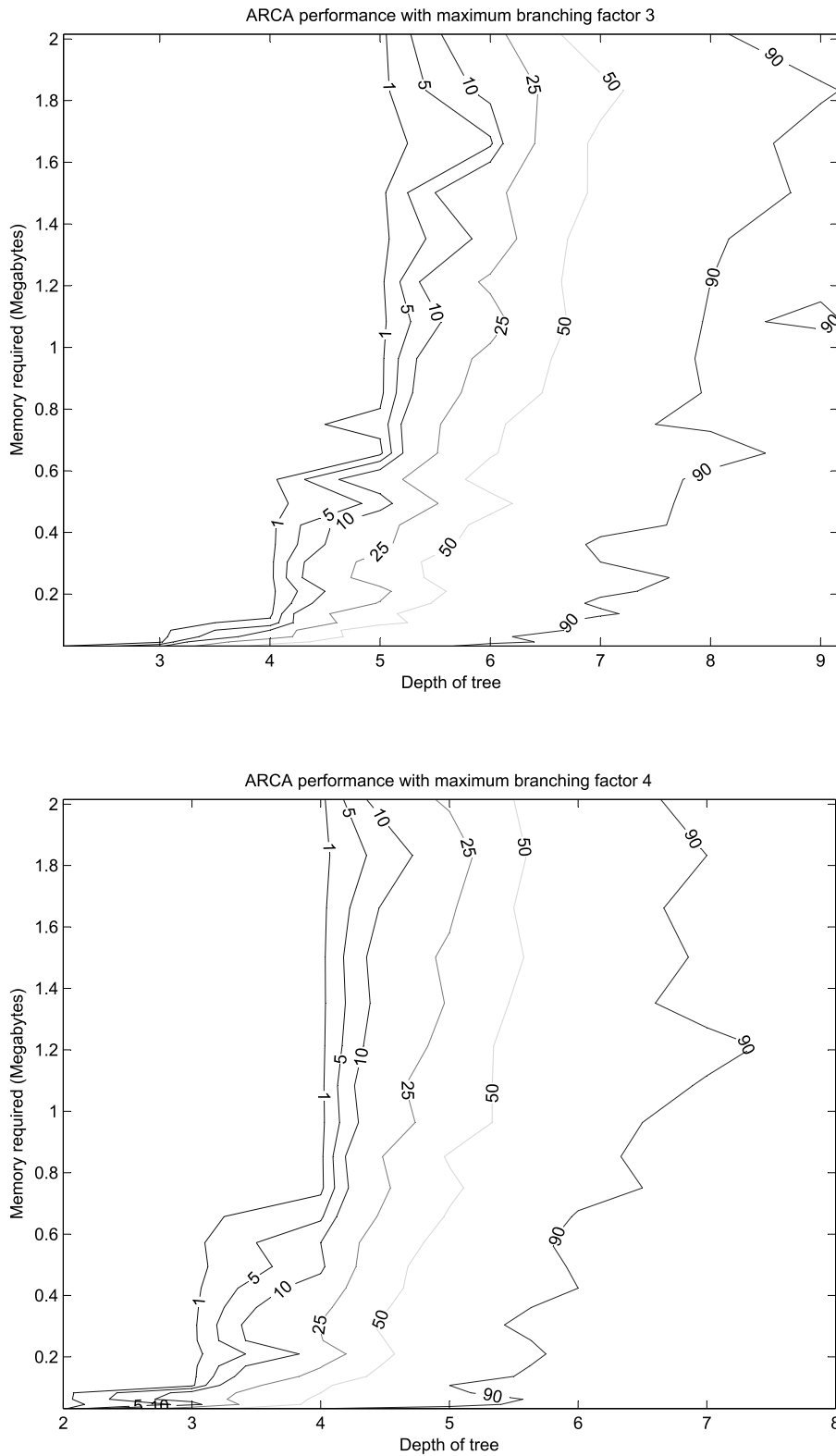


Figure 3.13: Contour plots showing the recall error performance for ARCA where the branching factor is 3 (top) and 4 (bottom)

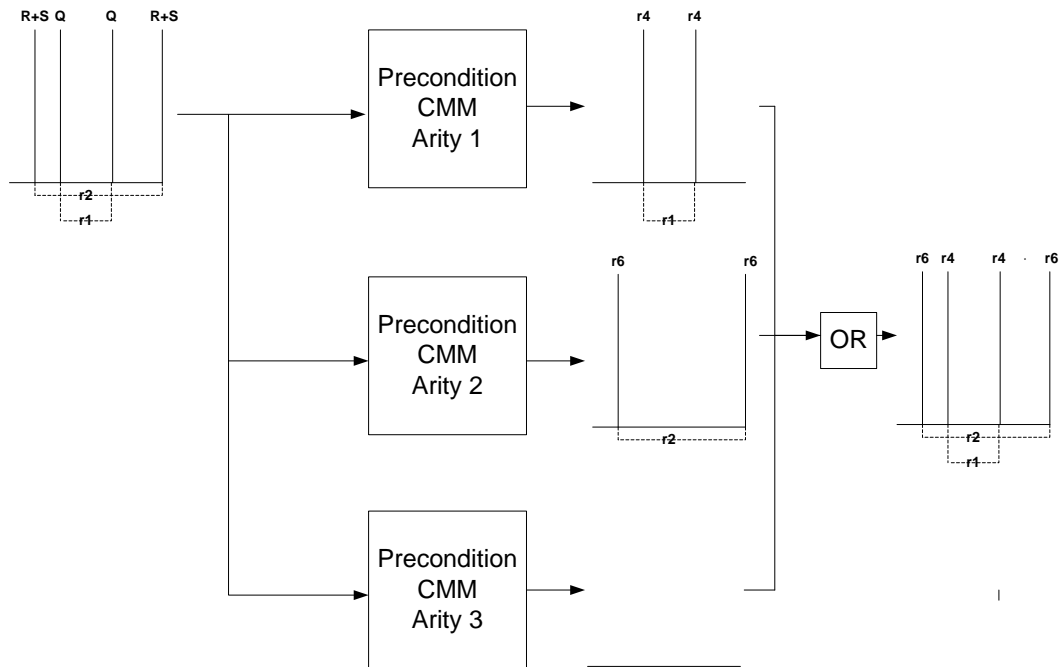


Figure 3.14: The implementation of multiple arity CMMs in place of the precondition CMM in ARCA. Each CMM matches rules with the appropriate arity, and the results are combined with a logical OR.

This is because rules which have multiple preconditions will be recalled if only a subset of those preconditions are met. However, by using a separate network to store rules of each arity this problem is overcome. In the case of ARCA this would involve replacing the precondition CMM with a set of CMMs, one for each required rule arity. When performing training, each rule is trained into only the CMM which corresponds to the arity of that rule. On recall, each vector is presented to every network, thresholded, and the results superimposed.

An example of arity networks is shown in Figure 3.14. In this case the arity 1 CMM matches a rule with Q as a precondition, the arity 2 CMM matches a rule with $R+S$ as a precondition while the arity 3 CMM finds no matching rules. These results are superimposed (combined with logical OR), giving a single tensor product containing the two matched rules bound correctly to the appropriate rule tokens.

Any further effects of implementing this addition to the architecture are currently unknown, and warrant further investigation.

3.3.11.3 Extensions to ARCA logic

Further work is required to extend this system to operate with more complex logics. Firstly, the system as described here has no ability to deal with logical negation in either the rules or the system state. Implementation of this would allow full representation of basic propositional logic [78]. Furthermore, in order to perform more complex rule systems, an implementation of a rule chaining system supporting first-order logic would be required. First-order logic is described by Russell & Norvig as “sufficiently expressive to represent a good deal of our common sense knowledge”. However, this is a significantly more complex logic, and to undertake this is a considerable task.

3.3.11.4 Application to real problems

Up to this point ARCA has only been tested on synthetic problems, and so the many potential issues which may occur when it is applied to a real problem have not been investigated. In particular, the structure of rules in a real problem is likely to deviate from those which have been examined in the experiments in this work. While we have examined theoretical limits to performance using large trees of unique rules, a real system may well have tokens appearing at multiple points in such a tree, as well as multiple trees. Performance in these cases would be of great interest.

3.4 Performance and Data Representation

Although they solve very different problems, both of the architectures examined in this chapter rely on correlation matrix memories to perform pattern recognition. The memories define the performance of the systems to a large extent. The benefits of CMMs, specifically speed and robustness to noise, allow the methods to perform in a very different way to many traditional methods of solving the respective problems of object recognition and rule chaining. On the other hand, the weaknesses of CMMs need to be acknowledged. The data representation needs to be very carefully chosen, and the point of failure is difficult to determine.

Of particular interest is the data representation used in these networks. As mentioned in Section 2.7.4, the data representation defines the performance of a CMM. Yet, this is something that previous work on these architectures has largely

not addressed. Through careful selection of a data representation, and corresponding thresholding function, it may be possible to improve the performance of these systems.

This observation motivates the remaining work in this thesis. While performance can be defined in many ways, here we choose to focus on two aspects; the storage capacity of CMMs, and the ability of CMMs to generalise. Chapter 4 examines a novel method for improving the storage capacity of correlation matrix memories, based on a specific data representation and thresholding function pairing. Chapter 5 then gives an overview of a novel method for generating codes that allow a correlation matrix memory to better respond to previously unseen inputs, and to perform generalisation on the output. It is hoped that by implementing these techniques into existing architectures such as those outlined in this chapter, increased performance will be observed.

3.5 Summary

We have examined two architectures which use correlation matrix memories to store rules. The CANN (Cellular Associative Neural Network) architecture is able to perform object recognition using an array of associative processors, using ideas inspired by cellular automata. ARCA (Associative Rule Chaining Architecture) is able to perform forward chaining upon large sets of rules. The performance of ARCA was examined experimentally with regard to its memory usage for trees of various depth and branching factor. The lack of study into data representation for both of these architectures was highlighted, and this observation motivates the remaining work in this thesis.

Chapter 4

Storage Capacity of Correlation Matrix Memories

4.1 Introduction

The storage capacity of a correlation matrix memory (CMM) is difficult to define in advance. As a result, considerable work has been conducted to attempt to explore this issue. In this chapter a review of such methods is undertaken, giving a variety of techniques which can be used to evaluate the storage capacity of CMMs. This leads on to a discussion of thresholding and data encoding in CMMs, and the benefits and drawbacks of various methods which can be employed.

In particular, we explore the use of fixed weight codes generated using the algorithm of Baum et al. [14]. This method has the benefit of generating unique codes which are well separated in pattern space, which makes them well suited for storage in a CMM. We will subsequently term the codes generated from this algorithm *Baum codes*, for ease of reference. While existing thresholding techniques can be applied to Baum codes, we present a novel method which improves the storage capacity of a CMM when using these codes.

4.2 Motivation

In Chapter 3, two architectures which utilise CMMs to store rules were described. In previous work on these architectures, the issue of data encoding has been largely

unexplored. If a data encoding is chosen such that the storage capacity of the CMMs is maximised, then the performance of the previously described architectures is improved. Therefore, an understanding of the issues involved with representation in CMMs in regards to storage capacity is important, as well as the implementation issues which are faced when applying these techniques to CMMs in a real system.

4.3 Theoretical results

It is impossible to exactly define the storage capacity of a CMM of a given size in advance, even if the data encoding is known. This is because the capacity will depend upon the pairs of data which are associated with one another. In addition, a CMM does not reach a clear point at which it has become “full”, but rather the probability that a recall will contain an error becomes larger as the memory becomes increasingly saturated. However, previous results have been achieved which aid our understanding of the storage capacity of CMMs, as well as informing appropriate choices for the length and weight of the codes used.

When recalling data associated with an input vector in a CMM, the output activity is a sum of any vectors associated with the input vector, as well as a noise component. This noise comes from associations between similar input items and various outputs, and is known as crosstalk. A discussion of crosstalk for continuously weighted CMMs was given in Section 2.7.2. The problem is similar in the binary weighted case, in that the angle between input codes is the defining factor in the amount of crosstalk, and therefore very important to the storage capacity of the network. The problem cannot be as simply expressed as in Equation 2.7 due to the fact that weights are clipped, but the principal is similar. In order to minimise the noise, the input codes should be an orthonormal set, although as we shall see this is not an optimal choice for other reasons. The issues associated with generating orthogonal sets of codes will be examined in Section 4.4. In addition, the choice of threshold function will also relate directly to the representation used. Threshold functions will be examined in detail in Section 4.5.

Willshaw et al. examined the information capacity of CMMs, and found that the theoretical maximum information per weight is $\ln 2 \approx 0.69$ bits [89]. This means that a CMM is capable of storing approximately 69% of the maximum amount of information which could possibly be stored in a traditional random access memory of that size, i.e. an *information efficiency* of 69%. This value is high considering

the distributed nature of the storage, and the associative properties of the memory. The information efficiency is important to consider as it defines a slightly different property to storage capacity. A memory may be able to store a large number of items whilst being very inefficient in terms of how the storage has been used, in which case the storage capacity would be high but the information efficiency would be low, suggesting that the information content of the codes stored was also low.

Palm [72] [73] took this information theoretic approach further, applying it to choosing an encoding which would maximise the storage capacity of a correlation matrix memory. His method involves maximising the average information which can be retrieved from a CMM by choosing a fixed weight encoding appropriate to the number of input and output neurons. Firstly, let us define the size of the input and output codes as n (giving an $n \times n$ CMM), the input weight as k , the output weight as l and the number of stored pairs as z . Then, we can define the information I stored in a CMM by the following equation, where N_t is the number of incorrect ‘‘ones’’ set on the output t :

$$I = \sum_{t=1}^z \left[\log_2 \binom{n}{l} - \log_2 \binom{l + N_t}{l} \right] \quad (4.1)$$

Equation 4.1 gives the sum of the information for all input/output pairs, using two terms. The first term, $\log_2 \binom{n}{l}$ gives the information in a fixed weight output code using a l of n encoding, since $\binom{n}{l}$ different codes can be represented using such an encoding. However, simply summing this term over all the pairs of data does not give the information stored in the CMM, since not all outputs will be recalled perfectly. Assuming a Willshaw threshold is applied, outputs may have extra bits set. Therefore, the second term in the sum reduces the information accordingly, being the information lost in the selection of l bits to set to one from $l + N_t$. Palm goes on to give the following theorem in [73].

Theorem. *For any $n \in \mathbb{N}$ and any $\epsilon > 0$, we can find parameters k, l, z such that*

$$\frac{I(n, k, l, z)}{n^2} \rightarrow \frac{\ln 2}{1 + \epsilon}$$

and

$$\frac{\text{Information stored per output string}}{\text{total information of one output string}} \rightarrow 1$$

for $n \rightarrow \infty$

The parameters are chosen in such a way that $k = (1 + \epsilon) \log_2 n$, $l = O(\log n)$, $z \sim \left(\frac{n}{\log n}\right)^2$

Breaking this down, the term $\frac{I(n,k,l,z)}{n^2}$ represents the information content of each weight in the memory (the total information divided by the number of weights in the memory). So, the theorem states that the parameters can always be set such that the information per bit is close to the theoretical maximum for a CMM, $\ln 2 \approx 0.69$. This is in agreement with the value given by Willshaw [89] for the theoretical maximum information efficiency in a CMM. In addition, the information stored for each output string will be close to the total information contained in that string. In order to obtain these results, the values k , l and z are set as given in the theorem. This gives a series of parameters which define the most efficient use of the memory as the size of n tends towards infinity.

Palm et al. give further asymptotic results for the storage of randomly generated pairs of codes in a CMM [74]. In this case, each position in the input and output codes has an independent probability of being set to 1, with the probability being p for the input codes and q for the output codes. In these circumstances, a high memory capacity is achieved when the number of 1s and 0s in the weight matrix are equal, or equivalently when the memory is 50% saturated. Achieving this means keeping the product zpq low, where z is the number of patterns being stored. Therefore, if we wish to store a large number of items in the memory, both p and q should be small. In other words, the input and output patterns should be *sparse* coded. In fact, the optimal information capacity of $\ln 2 \approx 0.69$ can be achieved when the number of 1s in each pattern is $O(\log n)$.

Two approaches to defining the storage capacity of a CMM were presented by Austin [13]. Firstly, a probabilistic approach allows us to define a probability of recall failure to the memory, P . This gives the chance of a recall which is imperfect (i.e. contains any error). Defining z as the number of patterns learnt, m as the length of input code, k as the weight of the input code, n as the length of the output code and l as the weight of the weight of the output code the value of P can be calculated as:

$$P = 1 - \left\{ 1 - \left[1 - \left(1 - \frac{lk}{nm} \right)^z \right]^k \right\}^n \quad (4.2)$$

Equation 4.2 is derived by calculating the number of weights set to 1 in a memory

which has learnt z associations, which allows the calculation of the probability of any outputs going above the output threshold (assuming a Willshaw threshold). For the full derivation see Appendix A in [13].

In addition, an instructive exercise is to calculate the number of patterns which may be stored in a memory before the probability of a single bit error (and only a single bit) becomes maximum. This allows an examination of the storage of the memory without reference to probabilities. This number, z can be defined as follows:

$$z = \frac{\ln\left(1 - \frac{1}{n^{1/k}}\right)}{\left(1 - \frac{lk}{nm}\right)} \quad (4.3)$$

This allows choices to be made for the weights and lengths of the output codes of a memory based upon the number of items to be stored, although again it assumes the use of Willshaw thresholding. Of course, at capacity such a memory would be subject to occasional erroneous recalls, as defined by Equation 4.2.

Graham & Willshaw calculated storage capacity and information efficiency values for a large CMM, with equal numbers of input and output neurons ($n = 2^{18} = 262,144$) [37]. They found that both the storage capacity (defined as the number of codes which can be stored before there is a one bit error in a recalled pattern) and information efficiency were highly non-linear functions of the input weight k and output weight l of the codes. Both the information efficiency and storage capacity were found to be maximal with an input weight $k = \log_2 n = 18$. For a given input weight, the capacity was found to reduce exponentially with increasing $\log_2 l$, with the information efficiency decreasing linearly. The information efficiency I can be approximated with the following function, where C is a constant for a given input weight, in the range $0 < C \leq 1$:

$$I = C \left(1 - \frac{\log_2 l}{\log_2 n}\right) \ln 2 \quad (4.4)$$

Interestingly, both the maximum efficiency and storage capacity were found to be at the same point; with input weight $k = 18$ and output weight $l = 8$. This gave an information efficiency $I = 0.58$ and a storage capacity of $z = 313,000,000$. Note that this value of z is orders of magnitude larger than the number of input neurons, $n = 262,144$. Comparing these results to the theorem of Palm given in [73] and repeated above, we see considerable agreement. The value of k is precisely as given by Palm, and the optimal output weight $l = 8$ is indeed in the order of $\log n = 5.42$. In terms

of the storage capacity Palm gave the value $z \sim \left(\frac{n}{\log n}\right)^2 = 2,340,000,000$, which is of a similar order of magnitude to the observed value. The information efficiency is clearly lower than the maximum possible $I = 0.69$, though not considerably so. Given that the storage capacity was defined by Graham and Willshaw to allow for no error, it is perhaps not surprising that the capacity and information efficiency are lower than the theoretical maxima.

All the work which has just been described is focused upon the storage capabilities of a single CMM attempting to recall associated pairs of data. However, when using CMMs in a more complex architecture, such as AURA [11], the situation becomes less clear. Turner and Austin [86] developed a probabilistic framework for analysis of AURA memories, which complicate the CMM model we have been examining by allowing partial matches, one-to-many associations and memories consisting of multiple CMMs.

One further issue which affects the capacity and coding issues we have discussed is the connectivity of the CMM. Up to this point CMMs have only been considered to be fully connected. However, Graham & Willshaw have shown that high information efficiencies can be achieved with codes which are more densely coded through the use of a partially connected CMM [36]. Such an approach is more biologically plausible, due to the connectivity between neurons in the brain. Making such a change to the connectivity of a CMM also requires different approaches to thresholding when compared to the fully connected version [35]. In practice their approach is able to achieve a high information efficiency for codes which are 2-3 times more densely coded than in the fully connected case.

In summary then, the theoretical work points to the use of sparse coding for both the input and output codes in a fully connected CMM. Using such codes appears to maximise both the information efficiency and the storage capacity of the memories. There appears to be general agreement that input weight should be in the order of $\log_2 m$, where m is the number of input neurons. Ideally, output codes are sparser than this, in the order of $\log n$, where the number of output neurons is n . Using this form of coding, it is possible to store $z \gg m$ codes in a CMM, while allowing for a small possibility of error. In addition, the crosstalk (and output noise) is reduced through using a set of input codes which are close to orthonormal.

4.4 Generating Sparse Fixed Weight Codes

As we have observed in Section 4.3, optimal storage capacities in a CMM are achieved when input and output codes are sparsely coded. The use of fixed weight codes is also beneficial, meaning that all codes have equal (hopefully optimal) sparsity, and allowing the use of L-max thresholding. In addition, we wish for the input codes to approximate an orthonormal set. Therefore, a relevant question is “How do we generate such codes?”.

Firstly, the concept of orthogonality between sets of codes should be made more precise. Given binary codes of length n , the dot product between two codes gives the cosine of the angle between them. When the dot product is 0, the two codes are orthogonal ($\cos \frac{\pi}{2} = 0$). Generating a large number of codes which represent an orthonormal set is often not possible, since the largest orthonormal set possible for a binary code of length n is n . Therefore, we are interested in sets of codes which are *close* to orthonormal. This means that the average dot product between all codes is as small as possible.

Perhaps the simplest representation of all would be the use of unary input codes (a single bit set to 1 in n bits). Such codes are maximally orthogonal and are simple to generate. This provides a storage capacity for a CMM of exactly n code pairs, as well as allowing exactly n different codes to be generated. Each output code will be stored in exactly one row of the matrix, and given a correct input code there will be no error on recall. However, the fault tolerance capability of the network is lost, since the storage is no longer distributed. It is necessary to use input and output codes with more than one bit set to 1 to distribute storage over the network. In addition, the theoretical results outlined in Section 4.3 suggest that the sparsity of codes should be in the order of $\log n$ or $\log_2 n$, so such codes have been shown to be highly suboptimal in terms of storage capacity and information efficiency.

Another issue which needs to be highlighted is that in some cases, input and output encodings may be the same. For example, this occurs in both ARCA and the CANN architecture outlined in Chapter 3. In the CANN rules are stored which associate between pairs of the same types of data, and in ARCA both the state and token encodings are involved in the input and output of CMMs. This makes the choice of an encoding more difficult, since the optimal weights described in Section 4.3 cannot be applied for both input and output codes.

We will now examine some methods for generating sparse fixed weight codes for

storage in correlation matrix memories.

4.4.1 Random codes

One obvious method for approaching the problem of generating fixed weight sparse codes is to randomly select k bits to set to 1 in the n bits of the code. This allows codes to be quickly generated, but makes no guarantees regarding the orthogonality of the generated codes. In addition, every new code must be checked against all existing codes to ensure that it is unique. This is not an inconsiderable cost, and only becomes more expensive as the number of codes to be generated increases. There is no real reason to generate codes for CMMS this way in practice, since Turner codes (described in Section 4.4.3) offer superior codes for little additional cost in comparison to random codes.

Alternatively, codes can be randomly selected from a list of all possible fixed weight codes for given values of k and n . This is certainly a superior alternative, since it is trivial to ensure that each code is only selected once from the list, and hence there is no need to check for uniqueness. However, the problem that there are no guarantees on the orthogonality between a set of codes generated this way is still an issue. Other methods exist which allow fixed weight codes to be generated in such a way that we can be sure that a set of codes are close to orthonormal.

4.4.2 Baum codes

Baum et al. proposed an algorithm which generates fixed weight codes which have a small amount of overlap [14]. The code is divided into l sections which are relatively prime (coprime) in length, with each section i having length p_i . For example, a code of length 32 where $l = 3$ could be divided into sections of length 16, 9 and 7. The size of l defines the weight of the code. To generate code number c , we set the bit in position j as follows (where x is the code to output):

$$\begin{aligned}
 x_j^c &= 1 \text{ if } j - \sum_{k=1}^{i-1} p_k \equiv c \pmod{p_i} \\
 &= 0 \text{ otherwise}
 \end{aligned}
 \tag{4.5}$$

1	0	0	0	0	1	0	0	1	0
0	1	0	0	0	0	1	0	0	1
0	0	1	0	0	0	0	1	1	0
0	0	0	1	0	1	0	0	0	1
0	0	0	0	1	0	1	0	1	0
1	0	0	0	0	0	0	1	0	1
0	1	0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	0	0	1
0	0	0	1	0	0	0	1	1	0
0	0	0	0	1	1	0	0	0	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 4.1: An example of the generation of Baum codes. Here, $l = 3$ and $p_1 = 5, p_2 = 3, p_3 = 2$, giving a code of length 10

Essentially, what is happening is that a single bit will be set to 1 in each section of the code. As subsequent codes are generated, the next bit in each section will be set to 1 instead, wrapping around to the beginning of the section when the end is reached. For example, Figure 4.1 shows a code with $n = 10$ and $l = 3$, taking $p_1 = 5, p_2 = 3, p_3 = 2$.

Using this mechanism $p_1 \times p_2 \times \dots \times p_s$ unique codes can be generated, which is substantially fewer than it is possible to represent with a general fixed weight coding scheme ($\frac{n!}{(n-l)!l!}$). However, the overlap between the codes is guaranteed to be small, which improves recall accuracy if they are used in a CMM. Since the method is deterministic, we can be certain about the amount of overlap between generated codes. With no loss of generality we can consider a Baum code with section lengths $p_1 < p_2 < \dots < p_s$. The first p_1 codes generated will have no overlap at all. The first $p_1 p_2$ overlap by at most 1 bit (a Hamming distance¹ of at least $2l - 2$). In their analysis Baum et al. [14] state that if $\prod_{i=1}^t p_i$ codes are used (where $t \leq l$), the minimum Hamming distance between any two codes will be $d = 2(l - t + 1)$. For this reason, it is beneficial for $p_i \approx n/l$, since this maximises the product between the section lengths p_i , and hence the number of codes which can be generated with minimal overlap.

As was mentioned above, when using Baum codes, fewer codes can be generated when compared to methods which use the full code space available for a binary fixed weight code (such as random codes or Turner codes). This is reflected in the information contained in a Baum code of a given size. As Nadal & Toulouse

¹The Hamming distance between two codes is the number of bits which differ between them.

pointed out, the goal of an increased storage capacity cannot be solely based upon the number of patterns stored, but also on the information contained within those patterns [68]. The information content of the codes stored in a CMM will affect the information efficiency I . Since fewer codes can be generated for a given code length, the information content in a Baum code of a given length is lower than for a random code (or indeed a Turner code as we shall soon see in Section 4.4.3) of the same length. Example information contents for Baum codes and random codes of various lengths are given in Figure 4.2. We see that while the information content for Baum codes is lower than for random fixed weight codes, the difference is not too huge for small values of l . Since we are using the algorithm to generate sparse codes, l will always be a relatively small value. Exactly what effect storing Baum codes has on the value of I is a question which should be tackled in future.

Having said all of this, an alternative viewpoint is offered when we consider the number of codes which will practically be stored in a CMM. The theorem of Palm given in Section 4.3 states that as n tends to infinity the number of stored codes which maximises information efficiency and storage capacity will tend to $(\frac{n}{\log n})^2$ with the appropriate selection of parameters. Taking this value as the number of stored pairs z , we can compare it to the number of Baum codes which can be generated for the appropriate values of n and l . If z associations are stored and the input and output spaces are different, then we require z unique codes on the input and up to z unique codes on the output. If the input and output codes are from the same space then we may require up to $2z$ unique codes from a single codespace. In other words, in order to store z associations in a CMM we will need to generate between z and $2z$ unique codes. Figure 4.3 shows the number of Baum codes which can be generated when the sparsity of the codes is set to be (a) $\log_2 n$ (the optimum sparsity for an input code) and (b) $\log n$ (the optimum sparsity for an output code), with the weight rounded to the nearest integer. Note that the breaks in continuity of the number of Baum codes are caused when the sparsity of the code changes. Taking $z = (\frac{n}{\log n})^2$ and plotting both z and $2z$ on the graphs, we see that sufficient codes are generated for all reasonable values of n when the weight is set to $l = \log_2 n$, and for $n > 316$ (or, more pertinently when $l > 2$) when the weight is set to $l = \log n$. This suggests that for practical applications, Baum's algorithm is capable of producing a sufficient number of codes to fill a CMM to capacity, provided the entire code space is utilised.

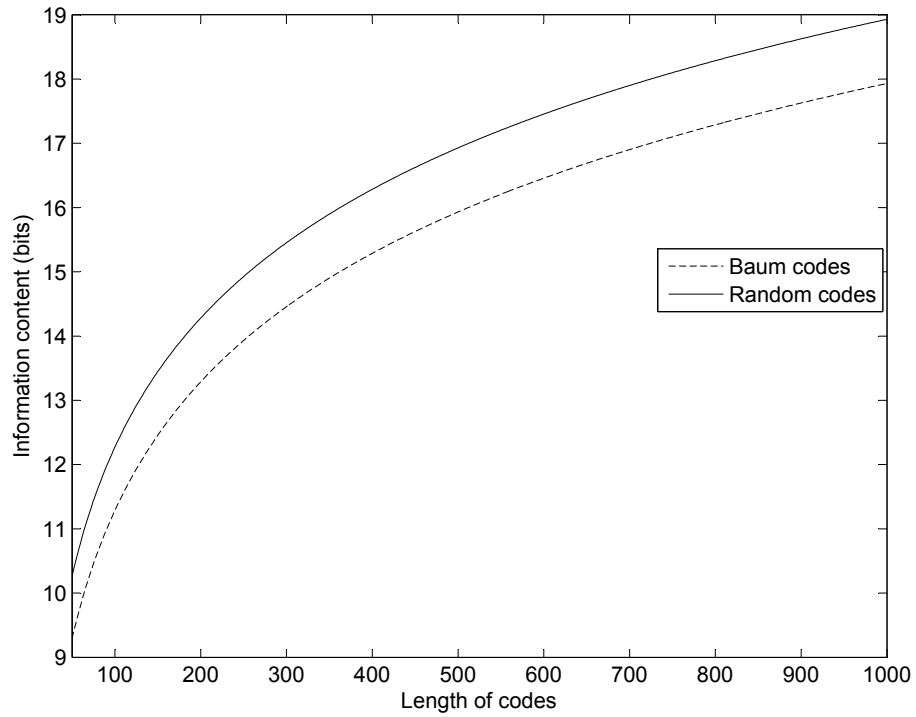
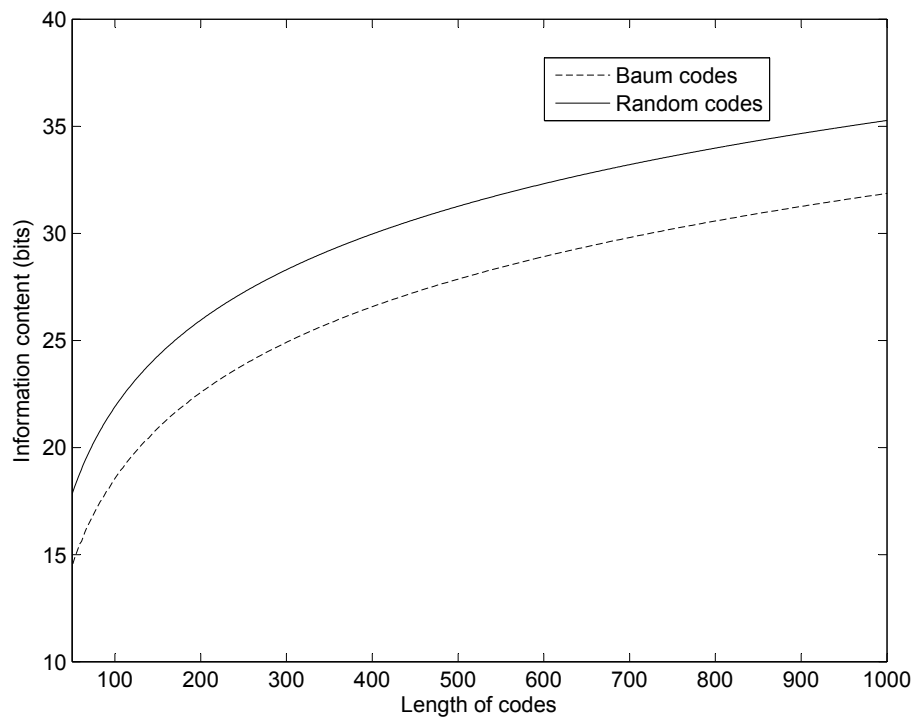
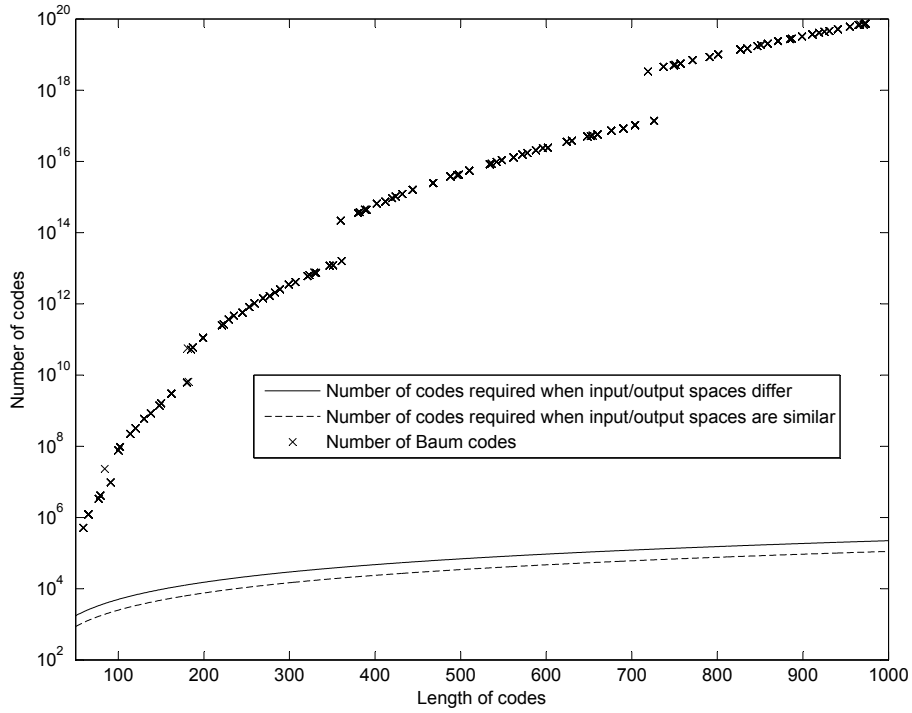
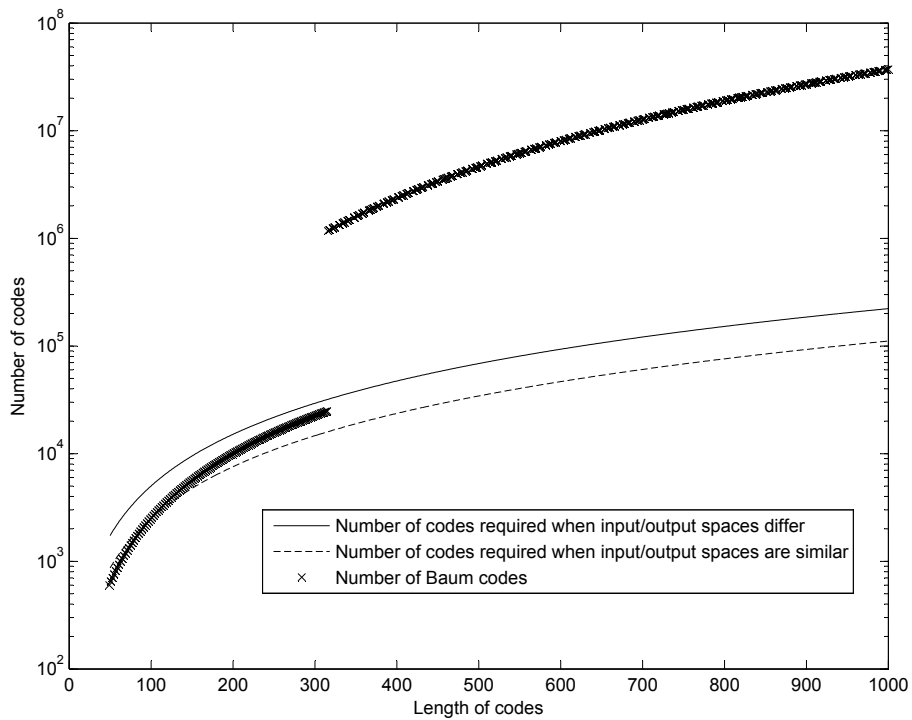
(a) Information content with code weight $l = 2$ (b) Information content with code weight $l = 4$

Figure 4.2: Information content for random fixed weight binary codes and Baum codes over a variety of code lengths



(a) Number of Baum codes produced with weight $\log_2 n$



(b) Number of Baum codes produced with weight $\log n$

Figure 4.3: Number of Baum codes produced compared to number codes which can theoretically be stored in a CMM

4.4.2.1 Generating coprimes for Baum codes

One issue with using Baum codes is that a set of coprime numbers must be generated before the method can be used. Generating such a set for a given code length n and weight l is not a trivial task. In addition, the set of numbers chosen should ideally be optimised such that the product between the numbers $E = \prod_{i=1}^l p_i$ is maximised, since this ensures the largest possible number of codes can be generated.

The naïve approach is to exhaustively generate all possible combinations of l coprimes which sum to n , and so select the set with the largest product. Such an algorithm is simple to write. However, as l increases the search space becomes extremely large, and the search quickly becomes intractable even for relatively small values of l . For this reason, this approach is not suitable for practical values of l and n .

If a set of numbers are prime, they are guaranteed to also be coprime. By using this fact, an alternative strategy can be devised. There is no need to calculate prime numbers, since huge lists of primes are readily available, certainly to a large enough number to be usable for Baum codes of a practical length. In this case, a set of primes of size l can be selected such that they sum to n . There may not always be a set of primes which sum to exactly n , but in practice it would seem reasonable to accept solutions which are close to n . While this technique could operate extremely quickly, especially when compared to an exhaustive search, the quality of the solution in terms of maximising the product of the coprimes E is likely to be far from optimal due to the rarity of primes.

An alternative is to use an approximation algorithm, which might work as follows. This assumes that sets of coprimes whose sum is close to n are acceptable, as this enables solutions which provide a much higher value of E . A starting seed p_1 is selected to be the first element of P . This value should be slightly smaller than $\frac{n}{l}$. The next value, p_2 is then produced by finding the smallest value larger than p_1 which is coprime with it. Each subsequent value in P is produced the same way, finding a number larger than the previous entry into P which is coprime with all the existing members of P . Since these numbers will generally be close together, and the first number was selected to be slightly smaller than $\frac{n}{l}$, we should find that $\sum_{x=1}^l P_x \approx n$. In addition, since all the values of P are close to $\frac{n}{l}$, the value of E is high. If a more precise approximation to n is required, a small local search could be applied for various values of p_1 .

4.4.3 Turner codes

An alternative method for generating fixed weight codes which display high orthogonality is described by Turner & Austin [85]. The method produces a set of codes which are closer to an orthonormal set than Baum codes, although at considerable computational expense.

The heart of the algorithm is that each new code is generated based upon minimising the dot product with the sum of all previously generated codes. This gives a code which minimises the average dot product between all pairs of codes C . For example, consider the case in which the following codes have already been generated, with code weight 2:

$$\begin{aligned} c_1 &= 100010 \\ c_2 &= 011000 \\ c_3 &= 000101 \\ c_4 &= 010100 \\ c_5 &= 001001 \end{aligned}$$

In this case, the sum over all the codes in C gives the vector 122212. The code which minimises the dot product with this vector can be determined by finding the smallest k values in the vector. These are the positions which should be set to 1 in the new code. This means that in this case the code which minimises the dot product is 100010. However, note that this code is already in the set C , as c_1 . This motivates the second key part of the algorithm, which is determining uniqueness. Note that this is a problem shared in common with randomly generated codes. For each candidate solution up to $\#C$ comparisons must be made. As C becomes larger and fewer codes are available to use, many such checks may be required each iteration of the algorithm.

The primary strength of this algorithm is that a set of codes is generated which are close to an orthonormal set, closer in fact than Baum codes. In addition, a larger number of codes can be generated. This means that the information content of Turner codes is larger, and hence fewer bits are required to produce a given number of codes. In addition, if only a subset of all codes are required, the first codes generated provide maximal orthogonality. Unfortunately though, the requirement to check for uniqueness reduces the usefulness of the algorithm, since the running time increases quickly as the number of codes to be generated increases.

4.5 Thresholding

The choice of threshold function is an important one, and relates to the encoding which has been chosen. Two thresholding techniques which can be applied to sparse codes stored in CMMs are Willshaw thresholding and L-max thresholding, as mentioned in Section 2.7.4. We will now examine these methods, and their strengths and weaknesses.

The name Willshaw thresholding refers to the form of thresholding which was described by Willshaw et al. in [89]. The thresholding function uses the number of 1s in the input code k as a threshold value, setting any output neuron with activity equal to k to 1. This form of thresholding has one very important property; it is guaranteed that no bits set to 1 on the output will be missed, given a correct input code.

If codes with fixed weight are used, an alternative threshold function is available; L-max thresholding [13]. This sets the l neurons with the highest output activity to 1 and the rest to 0, where l is the weight of the output code. Casasent and Telfer [25] experimented with various output encodings, including binary codes, Hamming codes and fixed weight codes, albeit with analog input codes. They found that in the presence of noise, fixed weight codes with L-max thresholding gave the greatest storage capacity for a given code length. This capability to make correct recalls from noisy input codes is the defining advantage of L-max thresholding.

To illustrate why, consider the recall shown in Figure 4.4. The recall from the CMM on the left is from a perfect input code. There are three output neurons which are associated with all three inputs which are set to 1, so these three neurons have activity 3. Using Willshaw thresholding, only those bits which are associated with all the inputs set to 1 will be set on the output. In this case this produces a correct recall. The recall on the right is based upon a noisy input code, with noise which has resulted in two bits being flipped. Note that the output activity has been reduced for two of the correct output bits. However, by taking the $l = 3$ highest output activities a correct recall is achieved even in the presence of noise.

In the simple case that inputs are noise free and only a small number of inputs have been stored, both of the described methods will have very similar performance. In short, with a perfect input L-max and Willshaw thresholds will perform identically as a memory becomes saturated until “extra” ones appear in the output. At this time, Willshaw thresholding will return the l correct bits set to 1, as well as an extra

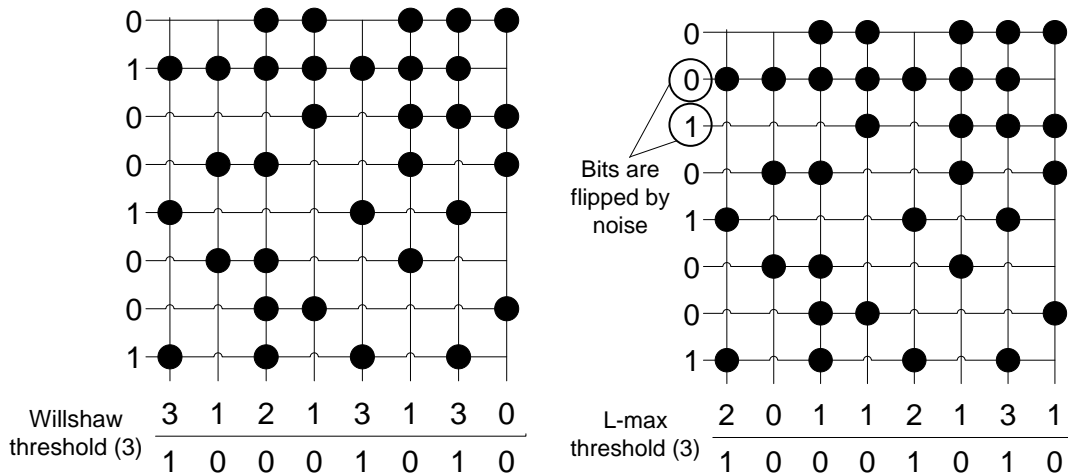


Figure 4.4: An example of Willshaw and L-max thresholding. The left image shows a recall using Willshaw thresholding from a perfect input code. The right image shows a recall using L-max thresholding from a noisy input code.

N erroneous bits, where N is the number of extra bits which have met the threshold. L-max thresholding will return an arbitrary choice of l bits from the $l + N$ bits set in the output. So while Willshaw thresholding will guarantee the correct bits have been set, L-max thresholding will more often get a perfect recall because occasionally the correct output will be “guessed” from the $l + N$ bits. Which behaviour is desirable is application dependent.

However, as previously mentioned, the real benefit of L-max thresholding occurs when there is noise on the input. This benefit can be clearly observed experimentally, examining the recall abilities of L-max and Willshaw thresholding for a variety of input noise levels. An input code size of $m = 256$ was chosen, with an input weight of $k = \log_2 256 = 8$. The selected output code size was $n = 256$ with an output weight of $l = \log 256 \approx 2$. Since the input will contain noise, the number of bits may be affected. For this reason we define two forms of Willshaw thresholding to compare. The first is standard Willshaw thresholding, where the threshold is set to the number of bits set to 1 which appear in the input code. We term the second “Fixed Willshaw” thresholding, in which the threshold is set to the number of 1s which were set in all inputs which were trained. In each experiment an empty CMM was created, and the following steps were then undertaken.

1. Generate a unique fixed weight input code.
2. Generate a random fixed weight output code (this need not be unique).

3. Train the CMM using the generated input/output pair.
4. Present every input which the CMM has learnt (altered by noise) and compare the correct output to the actual output using Willshaw, Fixed Willshaw and L-max. The input is subject to a different level of noise in each experiment.
5. If average error (defined below) exceeds 10% for all thresholding techniques then exit, otherwise return to 1.

For each experiment the above steps were run with twenty CMMs. A different integer was used to seed the random generator for each CMM, resulting in differing output patterns being trained, and differing noise. The noise level represents the probability that any given input bit will have its value “flipped” (changed from 1 to 0, or from 0 to 1). After each iteration the average error was calculated for all twenty CMMs. The recall error was defined as the percentage of recalled patterns which contained an error in any bit.

The results of these experiments can be seen in Figures 4.5, 4.6, 4.7, 4.8 and 4.9 for noise levels ranging from 0% to 10%. In the case that there is no input noise the performance of L-max and Willshaw is very similar, as one would expect. This can be seen in Figure 4.5. As the error increases, there is a small improvement in the number of perfect recalls when using L-max, caused by the occasions that L-max happens to “guess” the correct bits from those whose activity is maximal. Since there is no noise, the performance of Willshaw and Fixed Willshaw is identical.

As the level of noise increases, as seen in Figures 4.6, 4.7, 4.8 and 4.9 we see an increasing benefit in the use of L-max thresholding. We also see that Fixed Willshaw is a far superior alternative to standard Willshaw thresholding. Even with only a small amount of input noise, such as in Figures 4.6 and 4.7, we see that Willshaw thresholding has considerable error even with very few codes stored, whereas L-max thresholding enables a number of codes to be stored before errors begin to appear. This clearly demonstrates that L-max is by far the superior choice in applications where input noise is to be expected.

One issue with L-max thresholding is that it cannot be used in the case that outputs are superimposed. An underlying assumption of the method is that all input codes are associated with exactly one output code. For example, if an input code is associated with two output codes, and the weight of those codes is l then the correct output code would have $2l$ bits set to 1 (assuming that the codes are orthogonal). However, since there is no way to know that this is the case, L-max will still set l

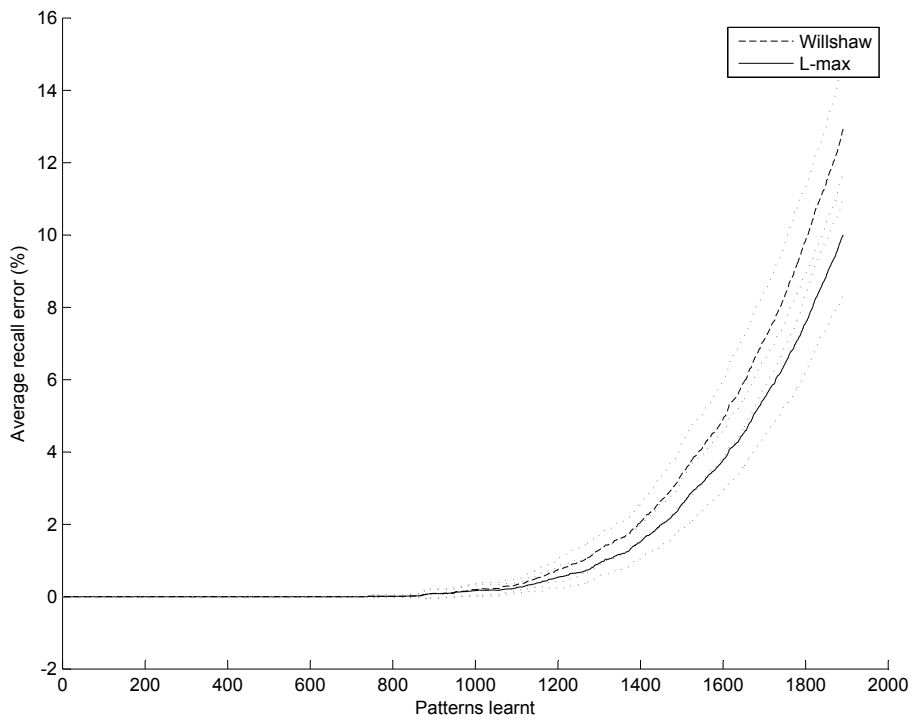


Figure 4.5: The performance of *L-max* and Willshaw thresholds with a noise free input. Dotted lines show the standard deviation of recall error.

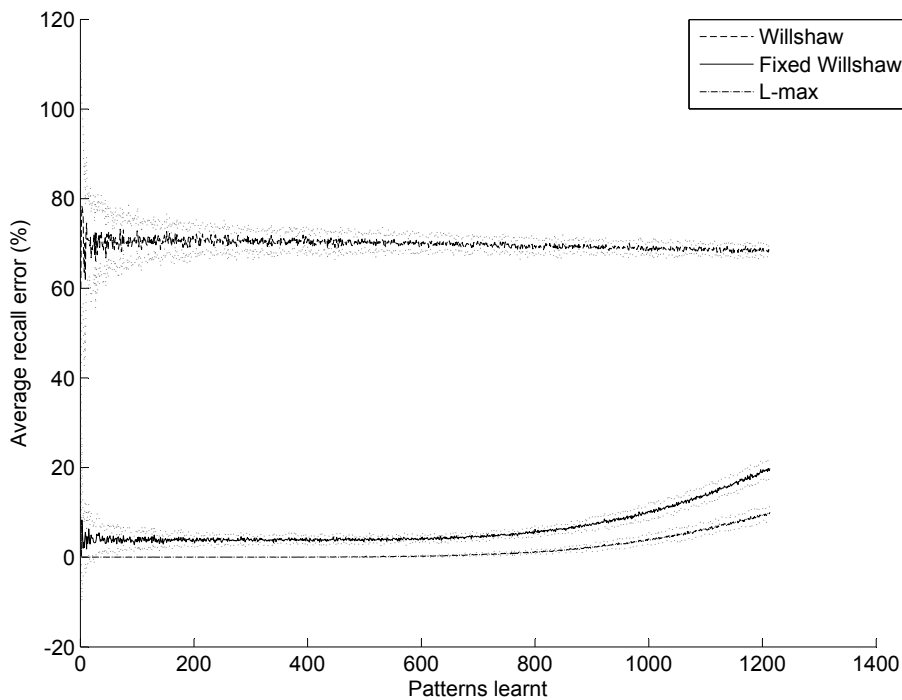


Figure 4.6: The performance of *L-max* and Willshaw thresholds with 0.5% input noise. For clarity, note that the top line is Willshaw thresholding, the middle is Fixed Willshaw and the bottom is *L-max*. Dotted lines show the standard deviation of recall error.

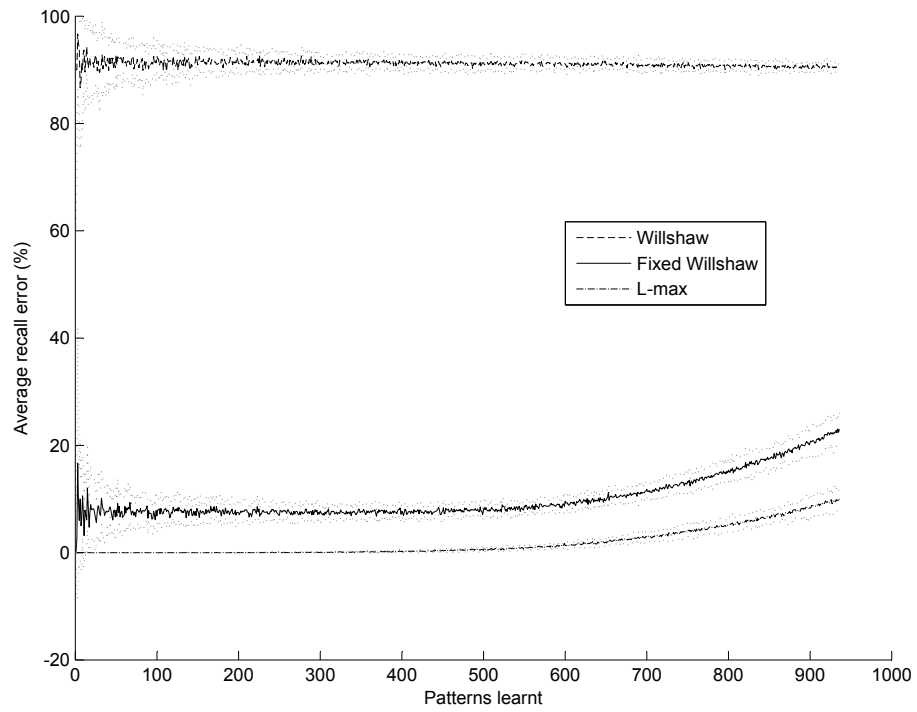


Figure 4.7: The performance of L-max and Willshaw thresholds with 1% input noise. Dotted lines show the standard deviation of recall error.

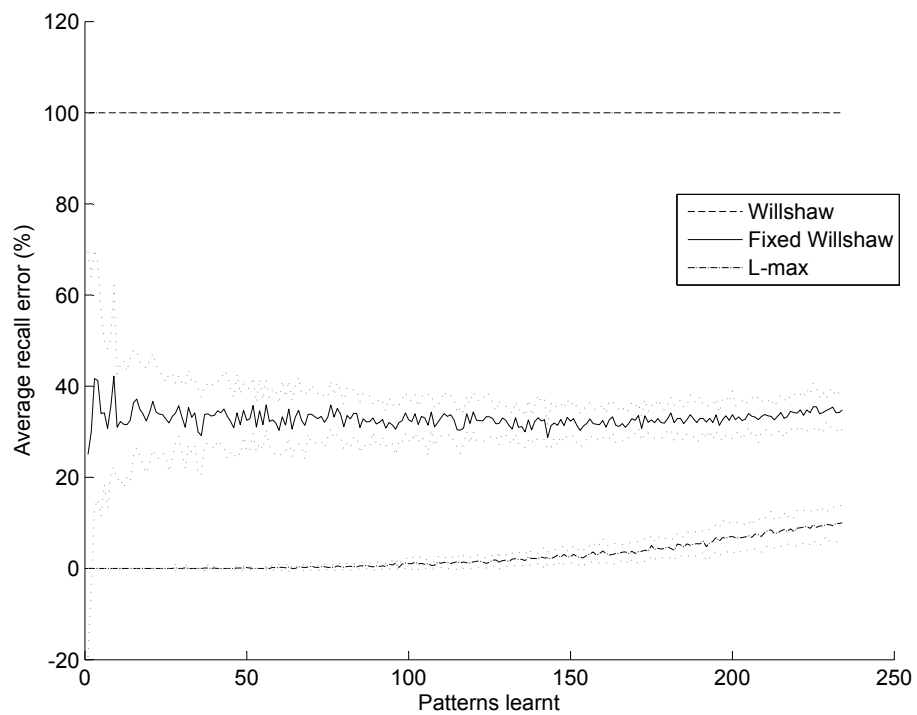


Figure 4.8: The performance of L-max and Willshaw thresholds with 1% input noise. Dotted lines show the standard deviation of recall error.

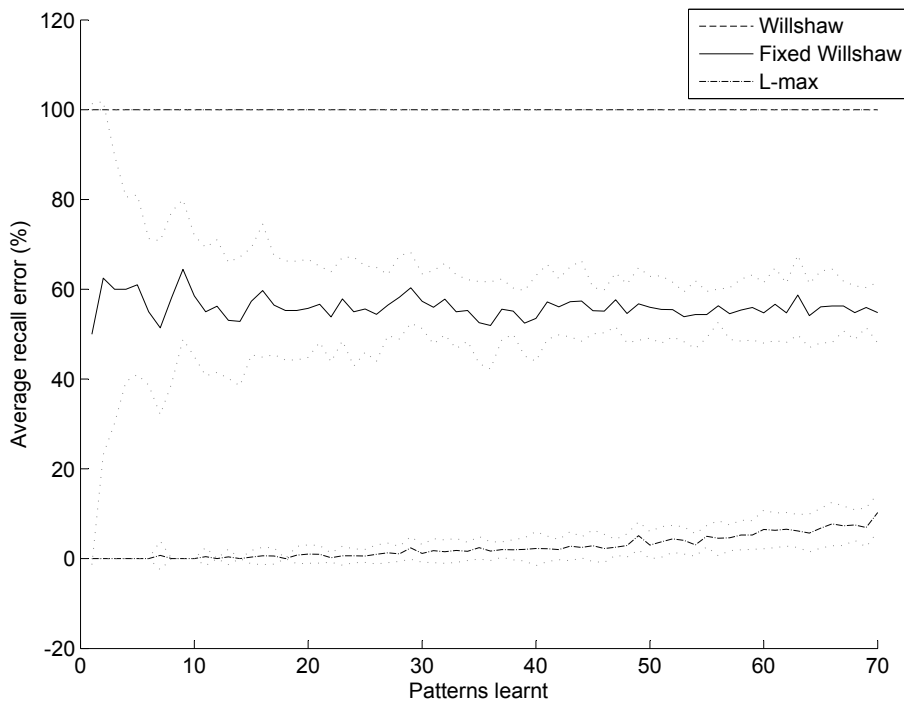


Figure 4.9: The performance of L-max and Willshaw thresholds with 1% input noise. Dotted lines show the standard deviation of recall error.

bits to 1. Worse, these bits may not even be those that belong to one of the two codes, but will be an arbitrary subset of both. For this reason, L-max thresholding is only suitable in the case where each input code is associated with exactly one output code.

Another issue with L-max is that a result is always returned, regardless of the strength of the output activity. In the case of Willshaw thresholding, if there were no outputs associated with a given input it is likely that a vector of all 0s would be returned. There may be occasions when using L-max we wish to only return a result when there is a level of confidence that an output was associated with the given input. For example, in the CANN architecture [8] (discussed in Section 3.2) the learning algorithm requires a different course of action when a given input has an output already associated with it compared to when the input has not previously been associated to anything. A decision on the likelihood that an input has an output associated with it can be reached by examining the output activity before thresholding. If the l highest output activities are close to the input weight k then we can be reasonably confident that an output code was associated with the input code being applied.

L-max thresholding requires communication between output neurons. This is in

contrast to traditional thresholding techniques which are entirely local to each individual neuron. This means that an efficient implementation of L-max is a slightly troublesome proposition.

4.6 Improving the Storage Capacity with Baum Codes

In the AURA library [7] the algorithm of Baum et al. [14] has been used to generate fixed weight codes, with L-max and Willshaw thresholding often used when recalling these codes. This represents an oversight, since both of these thresholding functions may produce output codes which are not possible under the Baum algorithm. By constraining the threshold so that only the codes generated by the algorithm are output, an increased storage capacity can be achieved.

It has already been mentioned that the algorithm divides the code into a series of sections. Baum et al. point out in the appendix to their paper that a useful property of the algorithm is that there is exactly one 1 in each section of the code [14]. This means that a winner-takes-all (WTA) threshold can be applied to each section of the code, rather than taking the l highest values from the whole code, as we would with L-max. We introduce this concept to thresholding in a CMM. This thresholding technique incorporates more information about the output encoding into the thresholding function, and therefore provides a more robust thresholding. We shall call this thresholding technique L-wta.

4.7 Results for L-wta

To demonstrate the improved storage capacity of a CMM when using L-wta compared to L-max a series of simulations were conducted. The storage of a CMM is affected by the size of the input and output codes, and also by the weight of the coding system used. For this reason the L-wta technique was compared to L-max for a variety of choices of these values. In each experiment an empty CMM was created for the appropriate code sizes. The following steps were then undertaken.

1. Generate an input code according to the algorithm of Baum et al. [14] This code will be unique.

2. Generate a random output code. This code is a random code from the entire space of possible Baum codes for the given set of coprimes, and so may not be unique.
3. Train the CMM using the generated input/output pair.
4. Present every input which the CMM has learnt and compare the correct output to the actual output using L-max and L-wta.
5. If average error (defined below) exceeds 10% for all thresholding techniques then exit, otherwise return to 1.

For each experiment these steps were run with twenty CMMs. A different integer was used to seed the random generator for each CMM, resulting in differing output patterns being trained. After each iteration the average error was calculated for all twenty CMMs. The recall error was defined as the percentage of recalled patterns which contained an error in any bit.

Tables 4.1 to 4.4 summarise the results which were achieved. In order to illustrate the performance of the thresholding techniques at a variety of error tolerances we examine the number of codes learnt in each memory before recall error exceeded 0.1%, 1%, 5% and 10%, together with the percentage increase in capacity provided by the L-wta method. The significance of these results was calculated by performing a Student's t-test, and therefore assumes that the values were normally distributed².

Table 4.1 shows the results when the size of the input was varied, whilst size and weight of the output code remained constant. Similarly, Table 4.2 shows the results when the weight of the input code was varied. In both cases it can be seen that the use of L-wta results in an increase of approximately 15% in storage capacity, and that this increase is highly statistically significant in the majority of cases. L-wta appears to provide the largest increase in storage over L-max when the input code is sparse; that is, when the code size is increased or the weight is decreased. This advantage appears less pronounced as the amount of output error increases, but the statistical significance of the results becomes greater.

The case is similar when examining Tables 4.3 and 4.4. The effect of output sparsity on the effectiveness of the technique is less clear. However, the storage capacity

²Because the experimental results gave a range of output errors rather than number of codes stored, the standard deviations for the Student's t-test were approximated by interpolating across the standard deviations for the output errors. It is important that the significances of the results in the tables, therefore, are viewed as an approximation.

Input	Output	0.1% error				1% error				5% error				10% error			
		L-max	L-wta	%	Sig.	L-max	L-wta	%	Sig.	L-max	L-wta	%	Sig.	L-max	L-wta	%	Sig.
64/4	256/4	70	78	11.4	ns	115	129	12.2	*	153	171	11.8	**	179	207	15.6	***
128/4	256/4	139	141	1.4	ns	197	234	18.8	***	289	334	15.6	***	345	406	17.7	***
256/4	256/4	259	286	10.4	ns	424	473	11.6	**	600	696	16.0	***	719	831	15.6	***
512/4	256/4	496	589	18.8	*	814	927	13.9	***	1182	1369	15.8	***	1416	1630	15.1	***

Table 4.1: Experimental results for L-wta when varying the size of the input code. All tables show the number of codes learnt before errors at given levels. Codes are given in the format length/weight. Note that in some cases the given code lengths are approximate. This is due to the complexity of generating large sets of coprime numbers which sum to a given target. Bold numbers show the percentage increase in storage capacity when using L-wta rather than L-max. * represents significance at the $P < 0.05$ level, ** at $P < 0.01$ and *** at $P < 0.001$ level. Results labelled ns have significance $P > 0.05$.

Input	Output	0.1% error				1% error				5% error				10% error			
		L-max	L-wta	%	Sig.	L-max	L-wta	%	Sig.	L-max	L-wta	%	Sig.	L-max	L-wta	%	Sig.
512/2	256/4	260	260	0.0	ns	282	304	7.8	ns	482	555	15.1	***	577	707	22.5	***
512/4	256/4	496	589	18.8	*	814	927	13.9	***	1182	1369	15.8	***	1416	1630	15.1	***
512/8	256/4	1023	1036	1.3	ns	1453	1603	10.3	***	1811	1989	9.8	***	2028	2229	9.9	***
512/16	256/4	1186	1267	6.8	**	1408	1512	7.4	***	1673	1824	9.0	***	1829	1989	8.7	***

Table 4.2: Experimental results for L-wta when varying the weight of the input code

Input	Output	0.1% error				1% error				5% error				10% error			
		L-max	L-wta	%	Sig.	L-max	L-wta	%	Sig.	L-max	L-wta	%	Sig.	L-max	L-wta	%	Sig.
256/4	64/4	91	120	31.9	*	178	203	14.0	*	245	286	16.7	***	287	337	17.4	***
256/4	128/4	148	179	20.9	ns	265	289	9.1	*	362	429	18.5	***	436	508	16.5	***
256/4	256/4	259	286	10.4	ns	424	473	11.6	**	600	696	16.0	***	719	831	15.6	***
256/4	512/4	373	415	11.3	ns	608	712	17.1	***	950	1099	15.7	***	1137	1327	16.7	***

Table 4.3: Experimental results for L-wta when varying the size of the output code

Input	Output	0.1% error				1% error				5% error				10% error			
		L-max	L-wta	%	Sig.	L-max	L-wta	%	Sig.	L-max	L-wta	%	Sig.	L-max	L-wta	%	Sig.
256/4	512/2	564	709	25.7	**	1259	1435	14.0	***	1932	2138	10.7	***	2310	2599	12.5	***
256/4	512/4	373	415	11.3	ns	608	712	17.1	***	950	1099	15.7	***	1137	1327	16.7	***
256/4	512/8	257	267	3.9	ns	353	400	13.3	***	495	569	14.9	***	580	677	16.7	***
256/4	512/16	71	89	25.4	ns	138	157	13.8	*	197	222	12.7	***	219	266	21.5	***

Table 4.4: Experimental results for L-wta when varying the weight of the output code

when using L-wta is consistently a considerable improvement over that achieved using L-max.

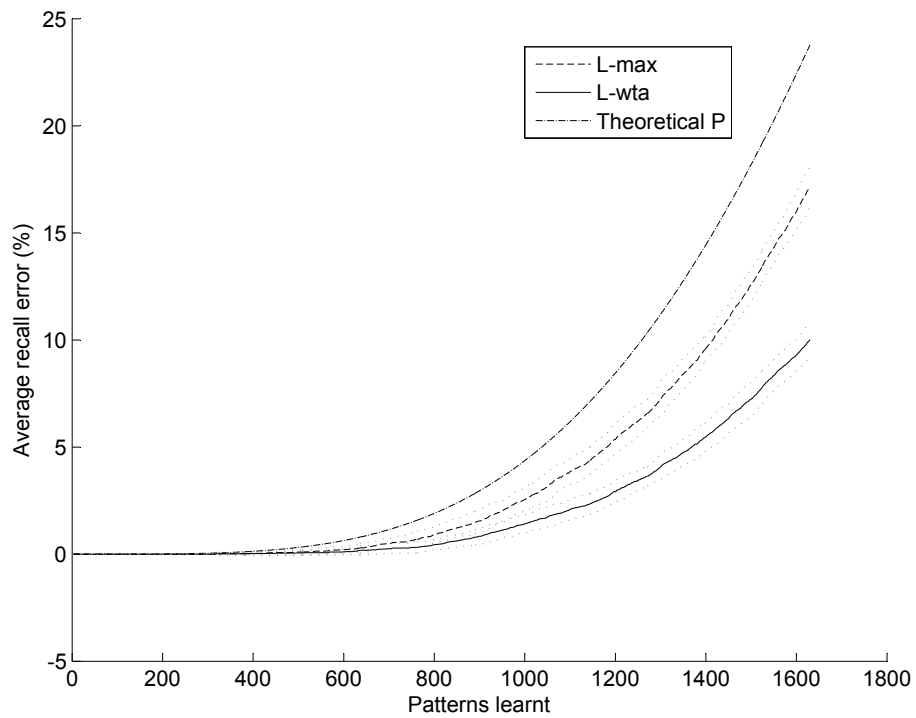
Figure 4.10 and Figure 4.11 show four examples of the performance of the two thresholding techniques as codes are trained into the memories. It can clearly be seen that as the memory becomes increasingly saturated, the use of L-wta provides an ever larger benefit over L-max thresholding. In addition, the error probabilities predicted by Equation 4.2 are given on these graphs. Note that the use of Baum codes has resulted in recall errors which are consistently an improvement on those given by Equation 4.2.

In addition, note that because there is no noise on the input vectors, the performance of L-max in this case is nigh on identical to the performance of Willshaw thresholding under the same conditions, as was demonstrated in Section 4.5. Thus we can also conclude that L-wta thresholding provides a performance gain over Willshaw thresholding when applied to Baum codes. Further tests are required to discover the benefits of L-wta thresholding when input codes are subject to noise.

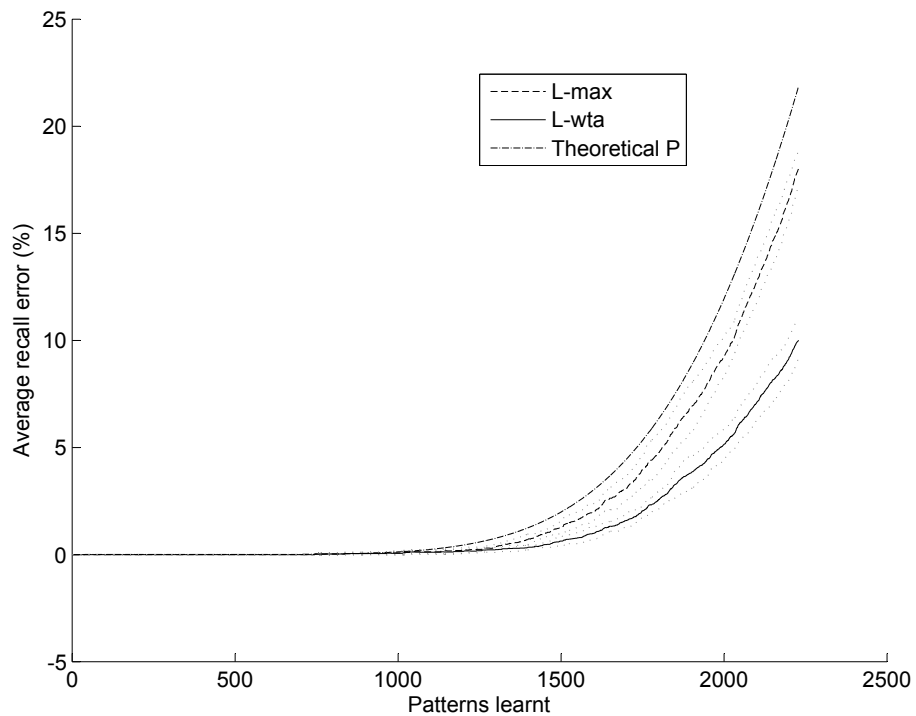
Comparing these results with other theoretical results given in Section 4.3 we see much agreement. While the values chosen for the lengths and weights of the codes were not optimal, in general the largest storage capacities appear where the theory would suggest. For example, when varying the output weight (Table 4.4), we would expect the largest capacity when the output weight is $\log_{10} 512 \approx 3$. We observe that the largest tested capacities occur when the tested weight is 2. When varying the input weight (Table 4.2) the result is similar. We would expect the largest capacity to be achieved when the input weight is $\log_2 512 = 9$. While at the 0.1% error level the larger capacity is actually when the weight is 16, the results at this level are the least significant. As the error increases (and the significance of the results increases as well) the largest capacity occurs when the weight is 8, as we would expect.

4.8 Further Work

There are two main pieces of work which remain to be undertaken based upon the work in this chapter. Firstly, while some preliminary investigation into the information content of Baum codes has been conducted, a more in depth exploration of the effect storing Baum codes in a CMM has on the information efficiency I is required. While Baum codes allow codes to be generated which display high orthogonality to one another, their use results in a lower information content in

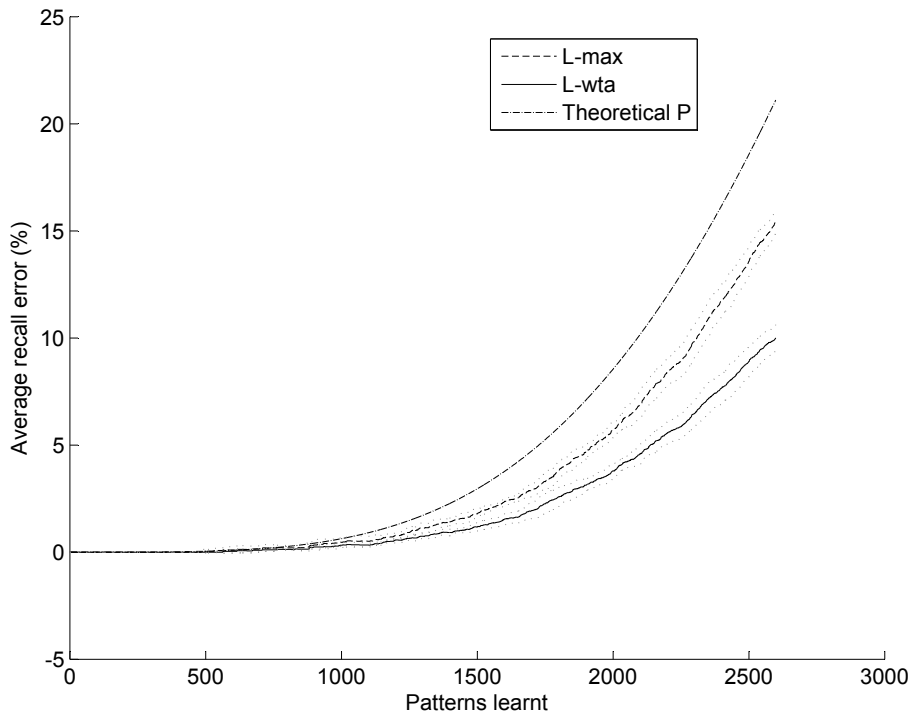


(a) Recall error with input length 512 & weight 4, and output length 256 & weight 4

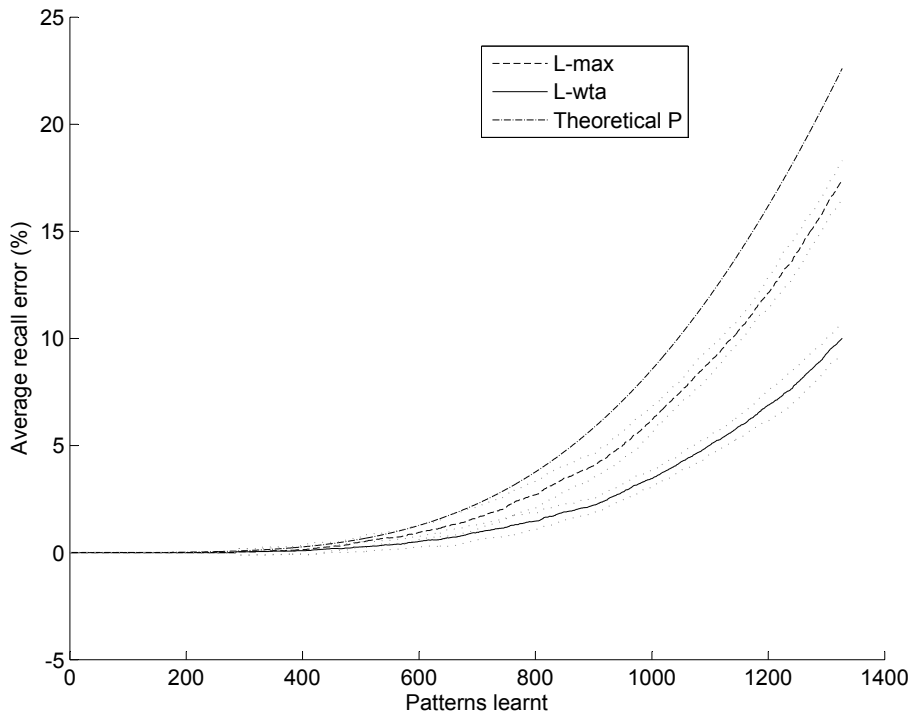


(b) Recall error with input length 512 & weight 8, and output length 256 & weight 4

Figure 4.10: Two comparisons of the storage capabilities of a CMM when using *L-max* and *L-wta*. Dotted lines show the standard deviation of recall error between runs of the experiment. Theoretical probability of recall error calculated using Equation 4.2.



(a) Recall error with input length 256 & weight 4, and output length 512 & weight 2



(b) Recall error with input length 256 & weight 4, and output length 512 & weight 4

Figure 4.11: Two further comparisons of the storage capabilities of a CMM when using L-max and L-wta. Dotted lines show the standard deviation of recall error between runs of the experiment. Theoretical probability of recall error calculated using Equation 4.2

each code. This means that although a larger number of Baum codes can be stored in a CMM for a given error tolerance, this may not mean that the information efficiency is actually higher.

Secondly, while L-wta has been shown to improve the storage capacity of a CMM storing Baum codes when compared to L-max and Willshaw thresholding, the performance of the technique in the presence of noise on the input code remains unexplored. Further work is required to determine how such noise affects the benefit offered by L-wta thresholding.

4.9 Summary

In summary, a survey of existing methods for exploring the storage capacity of CMMs was provided. The results given by these pieces of work point the way to the use of sparse fixed weight codes for storage in CMMs. The question of how such codes should be generated was addressed, and led to a discussion on the use of Baum codes.

The benefits of L-max and Willshaw thresholding were discussed, and results presented demonstrating the improvement in performance offered by L-max thresholding in the presence of input noise. It has been demonstrated that when using codes generated by the algorithm of Baum et al. [14] a novel technique, L-wta, provides an increase in storage capacity over thresholding using L-max. This increase in storage capacity is generally in the order of 15%, but has been observed to be as high as 30%.

Chapter 5

Coding for Generalisation in Correlation Matrix Memories

5.1 Introduction

The ability to sensibly respond to previously unseen inputs is one of the features which defines neural networks. Correlation matrix memories are able to display this capability, but in order to do so a coding scheme must be carefully chosen so that appropriate codes are assigned to input items which are similar. In this chapter we propose a framework for generating sparse fixed weight binary codes which exhibit a pre-defined similarity to one another, as determined by an input matrix of similarities. Two codes are defined to be similar when the Hamming distance between them is small. This method offers a flexible approach to code generation for correlation matrix memories in cases where generalisation is required.

5.2 Motivation

It has previously been shown that the use of sparse fixed weight codes allows a CMM to store a large number of associations, greater than the number of input neurons, provided a small possibility of recall error is tolerated [42]. This is achieved through the use of coding systems which produce sparse codes, and maximise the orthogonality between the codes. This is a very powerful capability, and in many applications it may be the best choice for the data representation. For other applications, however, it may be a requirement that the CMM is capable of

responding correctly to previously unseen inputs based upon the known inputs. This capability is termed *generalisation*. The question arises, is it possible to generate codes for input to a CMM which meet such a requirement?

In Section 2.7 it was noted that CMMs are “able to display a robustness to noise on the input, enabling accurate recall for an incomplete input”. This capability has been demonstrated [13], but perhaps not fully exploited. Consider the following definition of generalisation, given by Haykin [41]:

A network is said to generalize well when the input-output mapping computed by the network is correct (or nearly so) for test data never used in creating or training the network.

It can be observed that the ability of a CMM to make a correct recall when given an incomplete input is in fact a form of generalisation. Since the input has been altered from the code that was originally learnt, the ability to recall the correct output matches the definition given above.

This capability can actually be taken further, given an appropriate set of codes used as the input to a CMM. Consider the case that an input code is complete, yet previously unseen by the network. If that code is sufficiently similar to a code which has been previously learnt by the network, the CMM may be able to make a “correct” recall for the code for the same reasons the network is able to respond correctly to an incomplete input.

Consider the following example. We construct a memory which identifies a few fruits by their colours, as shown in Figure 5.1. A number of associations have been stored in the memory: yellow \rightarrow banana, red \rightarrow strawberry, purple \rightarrow plum and orange \rightarrow tangerine. The codes used to represent the colours have been chosen in such a way that colours which are similar are assigned similar codes (this idea will be further discussed in Section 5.3). The memory is capable of recalling any of the fruits it has been taught by presenting the appropriate colour at the input. When the previously unseen code for green is presented to the memory as shown, the code for banana is recalled. This occurs because the input code most similar to green is yellow; hence the output code associated with yellow is recalled. This is an example of generalisation.

It was mentioned in Chapter 3 that the performance of the ARCA and CANN architectures was largely dictated by the performance of the CMMs used by the two

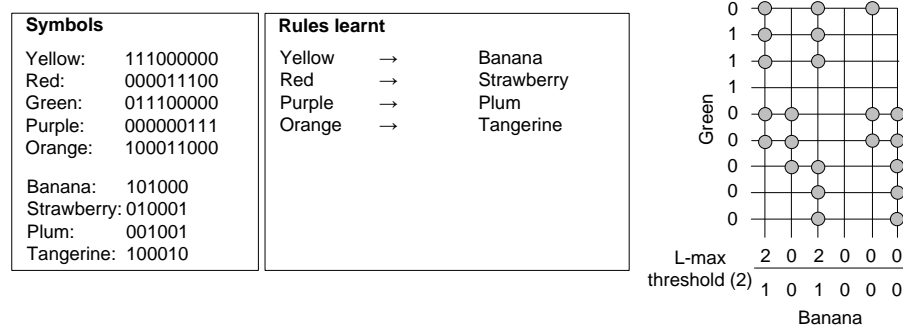


Figure 5.1: A simple example of generalisation in a CMM. This memory associates various colours to fruits of that colour. When an unseen input is presented (in this case green), the memory recalls the fruit associated with the input that is most similar to green. In this example the most similar input is yellow, hence banana is recalled at the output.

systems. Therefore, if the codes used in the CMMs in these systems enabled them to generalise, the two architectures would both gain the ability to respond to unseen inputs. Specifically, ARCA would be able to perform “fuzzy” rule chaining, making inferences for a set of symbols which are a close but inexact match to a learnt rule. In addition, the CANN would be able to recognise low level features which are close to learnt features, but not identical. Indeed, any architecture which employs CMMs would potentially benefit from this capability.

5.3 Generalisation and storage capacity

The multi-layer perceptron (MLP) [77] uses back propagation learning to learn the boundaries between classes of data. In effect, an internal representation is learnt, which replaces the representation of the original data. The outputs are then generated from this internal representation. In a CMM, generalisation is performed based on the similarity between input symbols. This contrasts with the MLP in that in a CMM we need to choose the data representation carefully.

The requirements for input codes for a CMM which maximise storage capacity and those which enable generalisation are quite different. When attempting to maximise the storage capacity of the CMM, as discussed at length in Chapter 4, the input codes should be (amongst other things) as far apart as possible in the code space. That is to say, they should be as close to an orthonormal set as possible. This condition is opposed to the idea outlined in the previous section; that in order to enable generalisation similar items should be assigned similar codes. Figure 5.2

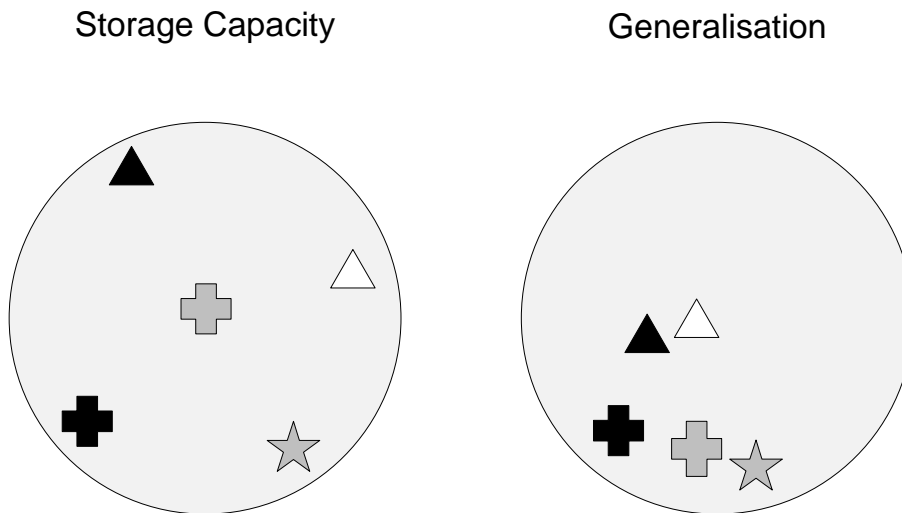


Figure 5.2: Designing codes to maximise storage capacity or to maximise generalisation require the codes to be arranged differently in code space. In the former case we desire codes which are far apart in the code space, in the latter case we desire codes which are close together where items are similar.

gives a visualisation of this concept, considering the problem of generating a set of input codes representing shapes of various shades. If the storage capacity is to be maximised the codes will be “pushed” as far apart as possible as shown in the left image. If we wish the CMM to be able to generalise, the codes for items which are similar (i.e. the same shade, or the same shape) should also display similarity.

In the case of CMM input codes, this similarity manifests itself as overlaps between codes; that is, 1s in the same position in both codes. Equivalently, we can consider the inner product of two codes to represent the level of similarity between them. For example, the codes 1100 and 0011 are orthogonal (inner product = 0), while the codes 1100 and 0110 are somewhat similar in code space (inner product = 1). For use in a CMM we desire a code generation technique that provides a mapping between the input space and the code space distances that is monotonic, preserving the order of the distances. Furthermore, an ideal mapping should also have a linear relationship; as distances increase between items in the input space the distances between their codes should increase at the same rate. In Section 5.7 we present a framework and novel optimisation procedure for the generation of such codes, based upon a predefined matrix of distances (a *distance matrix*).

It should be noted that the dynamic range of the Hamming distance in a sparse fixed weight code is going to be smaller than the input space distance in the general case. There is a choice therefore in the choice of weight for the code: a code with a smaller weight will (in general) provide greater storage capacity in the CMM due to

its sparsity, but there is a trade-off against the dynamic range of the similarity which can be represented. Additionally, a *strictly* monotonic mapping over the whole input space is impossible in most cases; the weight of the code provides a strict limit to the distances which can be represented by the encoding. However a monotonic mapping is possible if as the distance between two items in the input space increases and the dynamic range of the code is saturated, the distance between the codes remains strictly at the maximum.

Of course, one cannot truly separate the criteria of storage capacity and ability to generalise in CMMs. Recall that the output of a CMM is a mixture of the correct output code, and some other activity from other output codes related to similar inputs (crosstalk). If the input representation is carefully chosen then this “noise” may be viewed as providing a useful measure of the similarity between the input vector and other learned vectors, allowing for generalisation to occur. If the data representation is chosen such that similar inputs are related to similar outputs the noise will in general contribute only a small amount to the network output. This should mean that the recall performance of the network is high in cases where this criteria is met. Thus it may be possible to achieve a high storage capacity in some cases despite the input codes being far from orthonormal.

In summary, when choosing an input encoding for a CMM we are presented with two options. If the desired result is that similar inputs are to be associated to similar outputs then we wish for a code in which similar input codes are assigned codes with a high overlap. If this is not the case then an encoding such as that presented in Chapter 4 should be employed, which will maximise the storage capacity, whilst reducing the generalisation and error tolerance capabilities of the memory.

5.4 Similarity

Before discussing generalisation further, the concept of “similarity” needs to be addressed. What does it mean to say that two items are similar? And how is similarity measured? The Merriam-Webster dictionary defines similarity as follows:

adj: having characteristics in common

This natural way to think of the term leads us to a simple method for attempting to represent the abstract concept of similarity; through comparing a number of

characteristics, or *features*. Each feature represents some aspect of the item in question. A comparison can then be made between the set of features of two objects, allowing their similarity to be determined by some metric. This technique is commonly used throughout the pattern recognition domain.

For example, consider the example given in Figure 5.2. These objects can be represented by a large number of features. Some examples might be brightness, number of sides, area, perimeter or ratio of perimeter to area. The features chosen will define the similarity between the objects.

If an object is represented by n different features, and we make the assumption that each feature is represented by a continuous variable, we can think of each object as a point in n dimensional space. We can then define the similarity between two objects through their relationship in this space; specifically the distance between two objects. For any given application this distance might be calculated differently. For example, it might be desired that certain features are more heavily weighted, or that Euclidean or city block distance are used to calculate the distance between objects. Regardless of the method chosen, two objects with a small distance between them should be considered to be similar, whereas those which are distant from one another should be considered dissimilar. A truly general method for code generation should be flexible to these requirements.

5.5 Existing methods

We now present a discussion of some existing code generation techniques which are available for CMMs. In order to meet the requirements we have set out the methods must be able to generate fixed weight codes which maintain the distance relationship between points in the input space and the code space. In addition, we require the weight of the codes to be sparse, since such codes are necessary for CMM performance, as detailed in Chapter 4. Finally, since the memory requirements for a CMM are based upon the product of the length of the input and output codes, we wish for as many codes as possible to be represented for a given code length.

There are existing methods which generate fixed weight codes with a defined amount of overlap between codes: here we examine thermometer codes (as seen in [64]), Gray codes [38] and CMAC-Gray codes [61]. Each of these methods produces a series of codes in which adjacent codes are similar in code space. As such, each method is only suitable for representing a single dimension of the input space, representing a

1 1 1 0 0 0 0	1 1 0 0	1 1 0 0 1 1 0 0
0 1 1 1 0 0 0	0 1 1 0	0 1 1 0 1 1 0 0
0 0 1 1 1 0 0	1 0 1 0	0 1 1 0 0 1 1 0
0 0 0 1 1 1 0	0 0 1 1	1 0 1 0 0 1 1 0
0 0 0 0 1 1 1	0 1 0 1	1 0 1 0 1 0 1 0
	1 0 0 1	0 0 1 1 1 0 1 0
(a)	(b)	(c)

Figure 5.3: Examples of (a) thermometer codes, (b) fixed weight Gray codes and (c) CMAC-Gray codes

single feature. In Section 5.6 we examine how such methods could be applied to an input space with multiple features.

5.5.1 Thermometer codes

In a traditional thermometer code each subsequent code has an additional 1 added in the leftmost available position, e.g. 10000, 11000, 11100, 11110 etc. A fixed weight version of the code, as used by Kustrin [64] has all the bits set to 1 in a single block. Subsequent codes can be generated by shifting this block along the code, as shown in Figure 5.3(a). If the codes generated by this method are stored in an ordered list, the overlap between codes is monotonically decreasing as the distance between codes in the list increases. This allows a very natural representation for linear lists, such as the natural numbers. For example, if the codes given in Figure 5.3(a) were used to represent the numbers 1 – 5 then the overlap between 1 and 2 would be 2, the overlap between 1 and 3 would be 1 and between 1 and 4 or 5 it would be 0.

The weight of the code defines the level of similarity which can be represented between two objects. If the weight is larger then the distance between codes which share an overlap will also be larger.

Thermometer codes have the large disadvantage that very few different codes can be generated for a given code length. For a code of length n and weight k , only $n - k + 1$ codes can be generated. This means that to represent a large number of input items, the code must become very long. This has the effect of making the CMM very large, increasing memory requirements and processing time.

5.5.2 Fixed weight Gray codes

Gray codes [38] are binary codes with the property that adjacent codes differ by exactly one bit. While Gray codes are not fixed weight, it is possible to produce fixed weight Gray codes by simply generating a series of Gray codes and selecting only those codes with the desired fixed weight. Furthermore, such codes can be generated in constant time [17]. Adjacent codes within this new coding differ by exactly two bits (the minimum possible in a fixed weight code). An example of a fixed weight Gray code with weight 2 is shown in Figure 5.3(b).

The number of codes which can be generated by a fixed weight Gray code is maximal for a fixed weight binary code, specifically $\binom{n}{k} = \frac{n!}{(n-k)!k!}$. However, the distance between codes does not increase monotonically. For this reason, codes which are not close in the input space may be assigned codes which are close in the code space. This property is very undesirable for our purposes.

5.5.3 CMAC-Gray

The Cerebellar model articulation controller (CMAC) [1] is a neural network which was designed to perform complex control based on many input variables by taking inspiration from the manipulator control system in the brain. Of interest to us in this case is the non-linear mapping which is applied on the input to the network.

An input variable x is assigned a C dimensional vector $M[m_0, m_1 \dots, m_C - 1]$. Each field position can then be calculated with the following equation [62]:

$$m_i = \left\lceil \frac{x + C - 1 - i}{C} \right\rceil \quad (5.1)$$

This results in C values in which a single element shifts with respect to its neighbours. For use in a binary network it is then necessary to convert these values into binary codes; in the case of a CMM binary fixed weight codes. For CMAC-Gray codes, a Gray code is used to make this conversion. For the purposes of use in a CMM, more precisely, a fixed weight Gray code could be used. These codes are then concatenated in order to form the final codes. An example of a CMAC-Gray code with $C = 2$ is shown in Figure 5.3(c).

The use of the CMAC encoding reduces the issues with the non-monotonically increasing distances in the Gray code, improving the distance preservation from

input space to code space. The greater the value of C , the greater the local preservation of monotonically increasing distances is. Having said this, an overly large value of C is not desirable for two reasons: the number of codes which can be generated for a given code length decreases as C increases, and we cannot generate a sparse code if C is large. The code weight must be spread across the C elements of the code, and if C is large then the weight of the code must also be large. It is also still the case that codes which are distant from one another will overlap with one another, as with fixed weight Gray codes.

The method is very quick in operation, and a large number of codes can still be generated for a given code length, albeit many fewer than with a fixed weight gray code. In a similar fashion to Baum Codes (see Section 4.4.2), dividing the code into sections results in a reduction in the number of codes which can be generated for a given code length. Compared to $\binom{n}{k}$ codes for the fixed weight Gray code, we can generate only $\left(\frac{n}{C}\right)^C$. This value becomes increasingly smaller than $\binom{n}{k}$ as the size of C increases.

5.5.4 Discussion

In summary, the methods discussed in this section provide a trade-off between code size, the number of items that can be represented and the quality of the representation. Thermometer codes provide a code in which the overlap between codes decreases monotonically as the distance between inputs increases. However, the codes generated by the method are relatively large, since only a small number of codes can be generated for a given code length. Gray codes allow a much larger number of codes to be generated for a given code length, but the mapping from input space to code space is not monotonic, resulting in a poorer quality representation. The use of CMAC-Gray codes improves the monotonicity of the mapping from input to code space when compared to fixed weight Gray codes, although the number of codes which can be generated is lower than with fixed-weight Gray codes (while still significantly higher than thermometer codes). Thus, it provides the middle-ground between the first two methods.

Figure 5.4 shows the number of unique codes which can be generated by the discussed methods for a given code length and two different code weights (16 and 32). In order to demonstrate the effect of varying the value of C , multiple values are given. Since CMAC-Gray codes can only be generated where both k/C and n/C are whole numbers their values are only plotted on the graph at these points. Note that

the vertical axis has a logarithmic scale. Hence, we see that the number of codes which can be generated using fixed weight Gray codes is orders of magnitude higher than the other methods. When using CMAC-Gray codes, the number of codes which can be generated also decreases by orders of magnitude as C increases. Using thermometer codes results in the ability to generate only a tiny number of codes for a given code length when compared to the other methods.

Figure 5.5 shows some visualisations of the similarity between codes generated using each of the discussed methods for the numbers 1-50. The grey-scale value at each point in the matrix represents the overlap between the codes generated for that pair of numbers, with a maximal overlap represented by white (i.e. the codes are the same) and no overlap being represented by black. An ideal representation will move smoothly from white down the diagonal to black as the distance from the diagonal increases. It is clear to see that the thermometer codes provide the best representation, followed by CMAC-Gray, with Gray codes giving the worst result. This supports the claims made about the monotonicity of the mapping from code space into pattern space for each method.

5.6 Multi-dimensional inputs

The code generation methods examined in Section 5.5 only generate codes for a single continuous variable, which means that they can only represent an input which consists of a single feature. In order to generate codes in the case where an input is multi-dimensional (i.e. it has multiple features) we must generate a code for each dimension individually and concatenate them to create the final code. In this way, items with similar features will be assigned similar codes.

However, constructing codes in this fashion has an effect on the quality of the representation of the code; the distance mapping is only monotonic in the case that the dynamic range of the code is not exceeded in any dimension. In other words, in the case in which two items have a large distance between them caused by only a few features being significantly different, the resulting coding will have the items relatively close in code space.

This problem is perhaps best explained with an example. Consider the case in which we have three input items, A , B and C . Each of these consists of a three dimensional feature vector as follows:

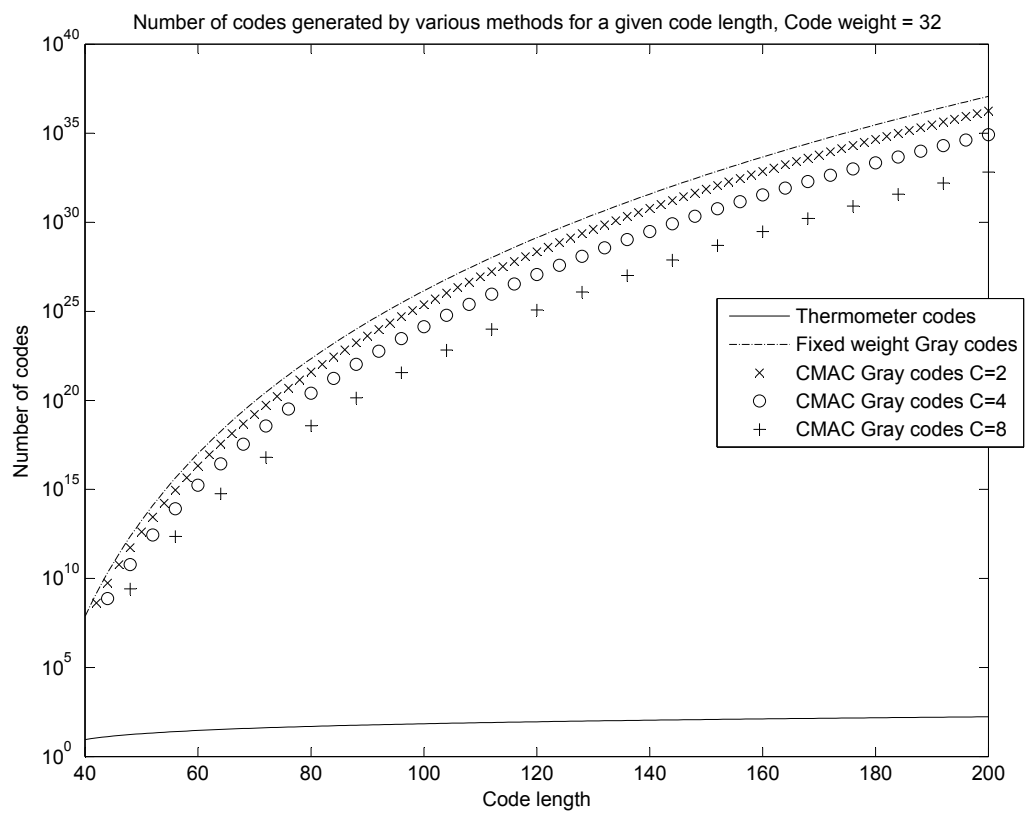
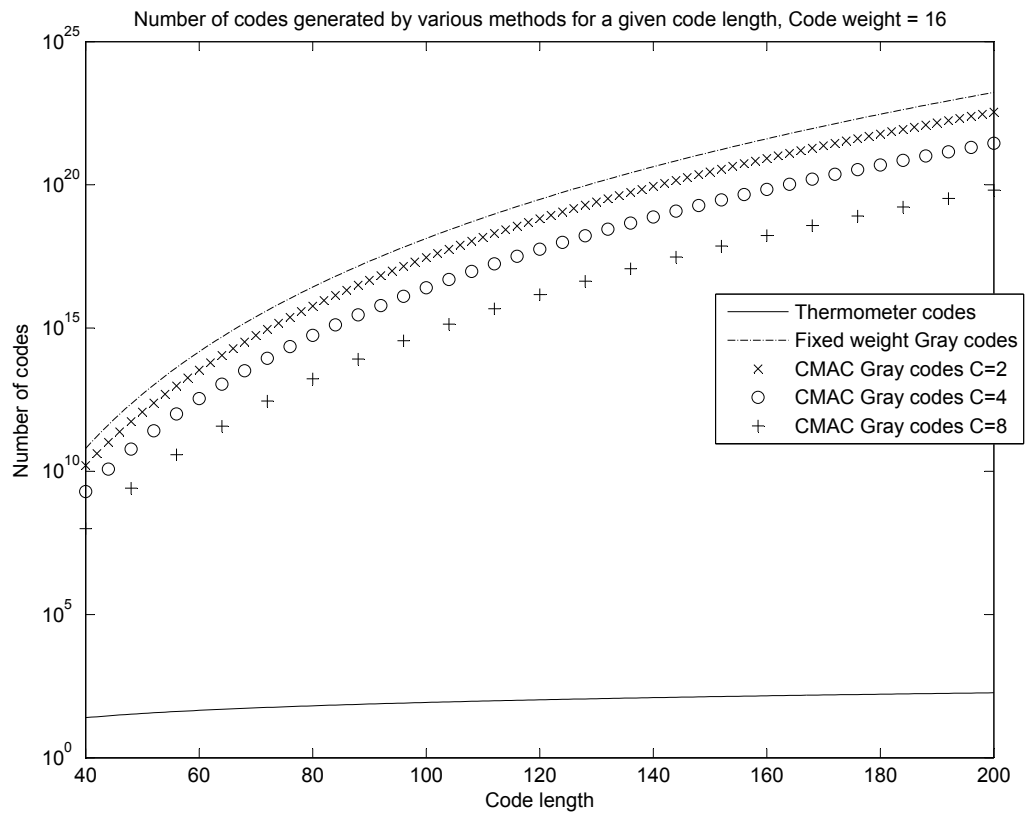


Figure 5.4: A comparison of the number of codes which can be generated for a given code length using thermometer codes, fixed weight Gray codes and CMAC Gray codes (for different values of C).

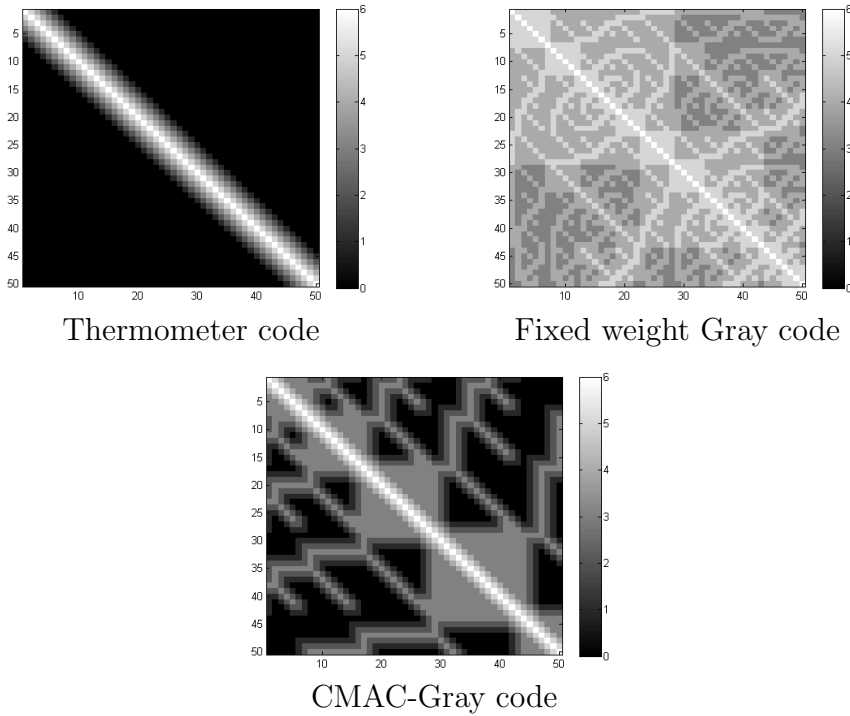


Figure 5.5: Similarity matrices for codes representing the numbers 1-50 generated by thermometer codes, fixed weight Gray codes and CMAC-Gray codes.

$$\begin{aligned}
 A &= [1, 3, 3] \\
 B &= [3, 1, 6] \\
 C &= [1, 3, 12]
 \end{aligned}$$

Observe that the distances between the three possible pairings of the input items (as measured by city block distance $\text{dist}_{\text{city}}$) are $\text{dist}_{\text{city}}(A, B) = 7$, $\text{dist}_{\text{city}}(A, C) = 9$, $\text{dist}_{\text{city}}(B, C) = 10$.

Consider the simple case in which we choose to represent each of these three input dimensions with thermometer codes of weight 3. This gives the following representation:

$$\begin{aligned}
 A &= [11100] + [00111] + [00111000000000] \\
 B &= [00111] + [11100] + [00000111000000] \\
 C &= [11100] + [00111] + [00000000000111]
 \end{aligned}$$

In terms of Hamming distance (dist_{ham}) between the generated codes (obtained through code generating function F) we observe that the distances are $\text{dist}_{\text{ham}}(F(A), F(B)) = 14$, $\text{dist}_{\text{ham}}(F(A), F(C)) = 14$, $\text{dist}_{\text{ham}}(F(B), F(C)) = 10$. Note that the ordering of the distances has changed during the mapping. Whereas before the mapping $\text{dist}_{\text{city}}(A, C) > \text{dist}_{\text{city}}(A, B)$, after generating the codes we have the case that

$\text{dist}_{\text{ham}}(F(A), F(B)) > \text{dist}_{\text{ham}}(F(A), F(C))$. Thus we can say that the mapping between distances in the input space and the code space was not monotonic, since the ordering of distances has not been maintained.

Furthermore, there are many metrics one might want to use to define the distances between multi-dimensional input items. The previous example used city block distance, but one could equally use Euclidean distance or any number of other distance metrics. When using thermometer codes, Gray codes or CMAC-Gray codes it is not possible to use an arbitrary distance metric. We now present a code production framework, *Overlapped Binary Code Construction* (OBCC), which allows construction of codes based on a distance matrix of the user's choosing.

5.7 Overlapped binary code construction

Similar observations to those made above regarding the requirements for the generation of binary codes which preserve a predefined overlap were made by Palm et al. in [74]. They presented a method for generating *Sparse Similarity Preserving Codes* (SSPC), which meet the requirements of being both binary and sparse, despite not being fixed weight codes. The Overlapped Binary Code Construction (OBCC) method presented in this section uses similar techniques as a framework for generating fixed weight binary codes, together with a novel optimisation procedure for minimising the length of the codes. We would prefer that the codes were not too large, since the size of a CMM is the product of the sizes of its input and output codes.

OBCC is a code construction framework which provides an alternative to the methods discussed in Section 5.5. Whereas those methods generated codes based on a multi-dimensional input of features, OBCC allows the generation of fixed weight codes based directly upon a matrix of distances between the items to be encoded. It offers a method for the production of codes from this information.

Before summarising the OBCC method, the similar technique, SSPC, given by Palm et al. should be described. SSPC begins with a matrix of distances between input items, which is converted into an overlap matrix through a very similar method to that which will be described in Section 5.7.1. Essentially, the distances are converted into the desired overlap between each pair of codes. At this point the overlap matrix is converted into a binary code, using the same procedure which is described in Section 5.7.2, and shown in Figure 5.7. This simple process produces a code

which perfectly preserves the similarities described in the similarity matrix, but it can produce very long codes. Therefore an optimisation procedure is desirable, to shorten the codes whilst maintaining the overlap between all pairs of codes. The need for such a process was acknowledged by Palm et al., but no process was described. OBCC improves on the method with a novel optimisation procedure, as well as an additional process to ensure the codes have a fixed weight.

The OBCC method proceeds with the following steps:

1. Convert the input *distance matrix* into a *overlap matrix*.
2. Optimise the code length by finding groups of items which display mutual similarity.
3. Generate codes based on the overlap matrix and the groups found.
4. Make the codes fixed weight.

The steps which primarily differentiate this method from the existing SSPC method are steps 2 and 4. Every stage of the OBCC method will now be explored in detail.

5.7.1 Production of an overlap matrix

The method takes a *distance matrix* as input. The matrix is of size $k \times k$, where k is the number of items to be encoded. This matrix is symmetric with 0s along the diagonal, and represents the distances between all the input items as defined by the user. These distances can be calculated by any metric the user requires, but must be positive integer values. In [74] Palm et al. suggested that the values in the distance matrix could be grouped into classes, with each value in the matrix then being replaced by an integer representing its class. This allows a distance matrix to be constructed with real numbers, preserves the order of the values in the matrix and allows the user to control the degree of similarity which needs to be represented. An example of a distance matrix can be seen in Figure 5.6. In this example the items to be encoded are the numbers 1 to 5, as labelled in bold. The distances in the matrix have been defined for any two items x and y as $|x - y|$. The first step in the method is to take the input matrix and to convert it into a matrix of the desired overlap between codes.

When using binary codes there is a finite level of precision which can be used to measure similarity, since only a finite number of bits may overlap between pairs of codes. This limitation can be viewed as the “dynamic range” of the code. If very distant items are to be considered similar then a very large weight would be required.

Initial distance matrix	Overlap matrix with $p = 2$																																																																								
<table style="border-collapse: collapse; text-align: center;"> <tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>2</td><td>1</td><td>0</td><td>1</td><td>2</td><td>3</td></tr> <tr><td>3</td><td>2</td><td>1</td><td>0</td><td>1</td><td>2</td></tr> <tr><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>1</td></tr> <tr><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr> </table>		1	2	3	4	5	1	0	1	2	3	4	2	1	0	1	2	3	3	2	1	0	1	2	4	3	2	1	0	1	5	4	3	2	1	0	<table style="border-collapse: collapse; text-align: center;"> <tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>3</td><td>2</td><td>1</td><td>0</td><td>0</td></tr> <tr><td>2</td><td>2</td><td>3</td><td>2</td><td>1</td><td>0</td></tr> <tr><td>3</td><td>1</td><td>2</td><td>3</td><td>2</td><td>1</td></tr> <tr><td>4</td><td>0</td><td>1</td><td>2</td><td>3</td><td>2</td></tr> <tr><td>5</td><td>0</td><td>0</td><td>1</td><td>2</td><td>3</td></tr> </table>		1	2	3	4	5	1	3	2	1	0	0	2	2	3	2	1	0	3	1	2	3	2	1	4	0	1	2	3	2	5	0	0	1	2	3
	1	2	3	4	5																																																																				
1	0	1	2	3	4																																																																				
2	1	0	1	2	3																																																																				
3	2	1	0	1	2																																																																				
4	3	2	1	0	1																																																																				
5	4	3	2	1	0																																																																				
	1	2	3	4	5																																																																				
1	3	2	1	0	0																																																																				
2	2	3	2	1	0																																																																				
3	1	2	3	2	1																																																																				
4	0	1	2	3	2																																																																				
5	0	0	1	2	3																																																																				

Figure 5.6: Construction of an overlap matrix for the numbers 1 to 5

For this reason, we may want to limit the distance between items which is considered to be “similar”, so that the weight of the code is kept to a reasonable level. We approach this problem by providing an additional parameter to the method, which we shall call p , the maximum distance at which items are still to be considered similar. This parameter affects the maximum weight which the produced code will have, although it does not directly control it; a larger value of p will tend to produce codes with a larger weight. The conversion from distance matrix D to overlap matrix M can then take place as follows:

$$M_{ij} = \begin{cases} 1 + p - D_{ij} & \text{if } (p + 1) \geq D_{ij} \\ 0 & \text{otherwise} \end{cases}$$

Figure 5.6 shows an example of this conversion with $p = 2$. This matrix now represents the desired number of bits which should overlap between all pairs of input items.

5.7.2 Code size optimisation and generation

The remainder of the OBCC process is dedicated to converting from the overlap matrix into a fixed weight code in which each pair of codes overlap by the amount specified in the overlap matrix. At this stage it is possible to generate a code through a naïve translation from the overlap matrix to a *code matrix* (the matrix of codes to be output by the method). All pairs of codes would have overlaps as specified in the matrix, but the length of the resulting codes would be much larger than necessary. However, we will examine this process as it informs the remainder of the method.

A naïve conversion is demonstrated in Figure 5.7. We consider all the values below the diagonal in the overlap matrix. For each of these values we create a number of columns in the code matrix equal to the value. These columns then have a 1 placed

	1	2	3	4	5
1	3	2	1	0	0
2	2	3	2	1	0
3	1	2	3	2	1
4	0	1	2	3	2
5	0	0	1	2	3

1:	1	1	1	0	0	0	0	0	0	0	0	0
2:	1	1	0	1	1	1	0	0	0	0	0	0
3:	0	0	1	1	1	0	1	1	1	0	0	0
4:	0	0	0	0	0	1	1	1	0	1	1	0
5:	0	0	0	0	0	0	0	0	0	1	1	1

Figure 5.7: A naïve translation from overlap matrix to code matrix

in positions in the code matrix corresponding to our current position in the overlap matrix. For example, the entry in position (2,1) in the matrix is 2. This would produce two columns in the new code with a 1 in positions 2 and 1.

As mentioned, this code is longer than is necessary. However, it is possible to reduce the length of the code, without affecting the number of bits that overlap between each code. Groups of columns can be merged together to reduce the length of the codes whilst maintaining the overlap between all pairs of codes. While in some cases a long code may be acceptable, a shorter code will lead to more compact CMMS.

Note that to start with all columns in the naïvely generated code matrix contain precisely two bits. Columns from this matrix can be represented, therefore, as two element sets. Now, for example, examine the code in Figure 5.7. The first, third and fourth columns contain overlaps between the following pairs of bits; $\{1, 2\}$, $\{1, 3\}$ and $\{2, 3\}$. These columns could all be merged into a single column, with bits $\{1, 2, 3\}$ all set. This preserves the amount of overlap between the codes, and reduces the number of columns in the code, as shown in Figure 5.8. This procedure can also be applied to further columns in this example.

Having demonstrated that it is possible to merge columns together while maintaining the overlap between all codes, we need to formalise the rules for such merges. In order to do this we can use the $\binom{n}{k}$ function, with $k = 2$ and n as the number of bits in the new column. We can enumerate all the two element combinations from a set of bits, and look for columns containing these pairs. For example, to create a new column with 4 bits set we calculate all $\binom{4}{2} = 6$ combinations, which gives the following general result $\{\{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \{c, d\}\}$. A new 4 bit column could then be created if a set of columns which matched this pattern could be found.

Having made the conditions under which columns can be merged precise, we require a method to quickly find such sets of columns. This can be achieved through examination of the overlap matrix, represented as a graph. Under this

```

1: 1 1 1 0 0 0 0 0 0 0 0
2: 1 1 0 1 1 1 0 0 0 0 0
3: 0 0 1 1 1 0 1 1 1 0 0
4: 0 0 0 0 0 1 1 1 0 1 1
5: 0 0 0 0 0 0 0 0 0 1 1 1

1: 1 1 0 0 0 0 0 0 0 0
2: 1 1 1 1 0 0 0 0 0 0
3: 1 0 1 0 1 1 1 0 0 0
4: 0 0 0 1 1 1 0 1 1 0
5: 0 0 0 0 0 0 0 1 1 1
    
```

Figure 5.8: Three columns (top) merging into one (bottom), whilst maintaining code overlap

Naïve code matrix	Graph representation	Optimised code matrix
<pre> 1: 1 1 1 0 0 0 0 0 0 0 0 2: 1 1 0 1 1 1 0 0 0 0 0 3: 0 0 1 1 1 0 1 1 1 0 0 4: 0 0 0 0 0 1 1 1 0 1 1 5: 0 0 0 0 0 0 0 0 0 1 1 1 </pre>		<pre> 1: 1 2: 1 3: 1 4: 0 5: 0 </pre>
<pre> 1: 1 0 0 0 0 0 0 0 0 2: 1 1 1 0 0 0 0 0 0 3: 0 1 0 1 1 1 0 0 4: 0 0 1 1 1 0 1 1 5: 0 0 0 0 0 1 1 1 </pre>		<pre> 1: 1 0 2: 1 1 3: 1 1 4: 0 1 5: 0 0 </pre>
<pre> 1: 1 0 0 0 0 2: 1 0 0 0 0 3: 0 1 1 0 0 4: 0 1 0 1 1 5: 0 0 1 1 1 </pre>		<pre> 1: 1 0 0 2: 1 1 0 3: 1 1 1 4: 0 1 1 5: 0 0 1 </pre>
<pre> 1: 1 0 2: 1 0 3: 0 0 4: 0 1 5: 0 1 </pre>		<pre> 1: 1 0 0 1 0 2: 1 1 0 1 0 3: 1 1 1 0 0 4: 0 1 1 0 1 5: 0 0 1 0 1 </pre>

Figure 5.9: An example of the optimisation of a code matrix by removing cliques from the graph representation

representation, each item to be encoded (the labels of the overlap matrix) is a vertex of the graph, and the values in the overlap matrix are the weights on the edges. Now, when examining this graph, sets of columns which can be merged will appear as *cliques*. A clique C is defined as a subset of the vertices V such that all the members of C are pairwise adjacent [75].

This is a powerful observation, as it allows the reduction of our problem to the well understood problem of finding cliques in an arbitrary graph [19]. Furthermore, we wish to find *maximal* cliques; cliques which are not a subset of any other clique. This is because we wish to reduce the size of the code by as much as possible, and by removing a clique which is not maximal we pass up the opportunity to reduce the size of the code by a larger amount. Beyond this, the strategy which should be used to select the clique to remove each iteration is an important and complex question which will be further examined in Section 5.8. For the remainder of this example we will simply greedily select the largest clique in the graph.

Having found a clique, we produce the resulting column in the code matrix, and subtract 1 from each edge involved in the clique (if the weight of an edge is reduced to 0 the edge is removed). The column created has a 1 set in each position corresponding to a vertex which was a member of the clique. By iteratively applying this process the code is gradually optimised. Eventually we are left only with cliques of size 2, and so columns are generated for these in the same fashion as for the other cliques. This gives the following optimisation algorithm.

1. Convert the overlap matrix into a graph.
2. Select a maximal clique in the graph
3. Produce a code column from the clique.
4. Subtract 1 from each edge in the clique.
5. If no cliques remain, finish. Otherwise return to 2.

An example of this process can be seen in Figure 5.9. The first column shows a conversion from the current state of the graph to a code matrix, with the largest clique highlighted in bold. The second column shows the graph representation of the overlap matrix, again with the chosen clique in bold. The final column shows the code as it is built up. In the first step, the largest clique found is $\{1, 2, 3\}$. This produces the first column in the optimised code matrix as shown. This process continues in the following two steps, as the cliques $\{2, 3, 4\}$ and $\{3, 4, 5\}$ are also found, and corresponding columns placed in the code matrix. In the fourth and fifth iterations of the algorithm the last two (single edge) cliques are removed and

1:	1	0	0	1	0	1	0
2:	1	1	0	1	0	0	0
3:	1	1	1	0	0	0	0
4:	0	1	1	0	1	0	0
5:	0	0	1	0	1	0	1

Figure 5.10: Code is made fixed weight with the addition of single bit columns

the appropriate columns added to the code (shown together in the final stage of Figure 5.9). At this stage the optimisation of the code matrix is complete.

An important note at this point is that it is not essential to find all cliques in order to generate a legitimate code. Each clique found represents an optimisation of the code, folding multiple columns into one, and as such reduces the size of the code. However, if time constraints do not allow for the optimisation process to complete, it is possible to generate codes based only on those reductions already found. This would simply involve adding column(s) to the code for each remaining edge in the graph according to their weight.

5.7.3 Create fixed weight code

Up to this point, the requirement that codes should be fixed weight has been ignored. The final step in the code generation process is to ensure that codes are fixed weight. This is accomplished simply by finding the code with the largest weight, and adding bits to each code to bring it to that same weight, each in an individual column. This ensures that all codes have the same weight, whilst not affecting the number of bits which overlap. Continuing with the same example, this process can be seen in Figure 5.10. In this case, the codes for 2, 3 and 4 have the highest weight, 3, whilst the codes for 1 and 5 have a weight of only 2. For this reason, we add a column with a single 1 in position 1, and another column with a single 1 in position 5.

5.8 Clique selection strategy

The strategy which is used to select the next clique to remove each iteration is extremely important. It will affect both the running time of the optimisation process and the length of the generated code. Essentially, the strategy is defined by the algorithm which is used to find the next clique in the graph, and so this section provides an overview of the available techniques for finding maximal cliques.

Graph representation	Optimised code matrix
	1: 1 2: 1 3: 1 4: 1 5: 0 6: 0 7: 0 8: 0
	1: 1 0 0 0 0 0 0 1 1 2: 1 1 1 0 0 0 0 0 0 3: 1 0 0 1 1 0 0 0 0 4: 1 0 0 0 0 1 1 0 0 5: 0 0 0 0 0 0 1 1 0 6: 0 1 0 0 0 0 0 0 1 7: 0 0 1 1 0 0 0 0 0 8: 0 0 0 0 1 1 0 0 0

Figure 5.11: A code optimisation selecting cliques using a greedy strategy

In general, prioritising the selection of larger cliques would appear to be advantageous since it will reduce the size of the code by the largest amount. This results in a greedy approach to clique selection, as used in the previous example. However, this may not always be optimal. Consider the example graph given in the first line of Figures 5.11 and 5.12. Figure 5.11 shows the optimisation of the code represented by this graph using a greedy strategy to select the next clique to remove. In this case, the clique $\{1, 2, 3, 4\}$ is the largest. Having removed this clique, all remaining cliques are of size 2, so we can quickly remove these, completing the optimisation process and leaving a code of length 9. However, in Figure 5.12 we see the same code represented by the same graph optimised using a different clique selection strategy; the smallest clique of size greater than 2 is selected. This results in the multi-stage optimisation process shown, which results in a final code of length 6; significantly shorter than that produced using the greedy strategy.

This raises the question of what the optimal strategy for clique selection truly is. A step in the right direction might be to remove the largest cliques in the graph, except in the cases where a certain number of the edges in the clique are also members of other cliques. This would help to deal with the problem we have just highlighted. However, there is another issue involved with clique selection beyond

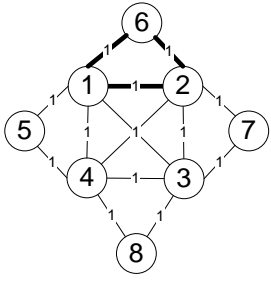
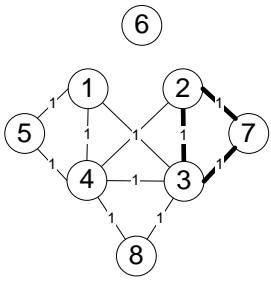
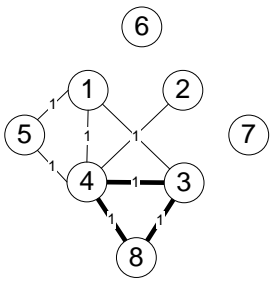
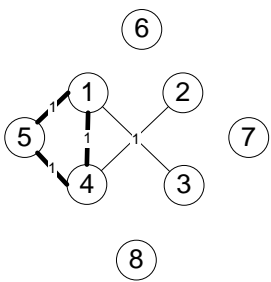
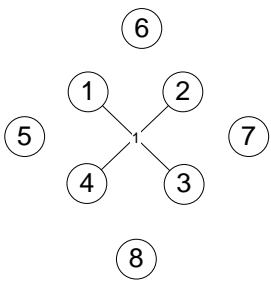
Graph representation	Optimised code matrix
	1: 1 2: 1 3: 0 4: 0 5: 0 6: 1 7: 0 8: 0
	1: 1 0 2: 1 1 3: 0 1 4: 0 0 5: 0 0 6: 1 0 7: 0 1 8: 0 0
	1: 1 0 0 2: 1 1 0 3: 0 1 1 4: 0 0 1 5: 0 0 0 6: 1 0 0 7: 0 1 0 8: 0 0 1
	1: 1 0 0 1 2: 1 1 0 0 3: 0 1 1 0 4: 0 0 1 1 5: 0 0 0 1 6: 1 0 0 0 7: 0 1 0 0 8: 0 0 1 0
	1: 1 0 0 1 1 0 2: 1 1 0 0 0 1 3: 0 1 1 0 1 0 4: 0 0 1 1 0 1 5: 0 0 0 1 0 0 6: 1 0 0 0 0 0 7: 0 1 0 0 0 0 8: 0 0 1 0 0 0

Figure 5.12: A code optimisation selecting cliques using a more optimal strategy than that used in Figure 5.11

simply finding the strategy which results in an optimal code, which is the complexity of the clique finding algorithm. In addition to the fact that using a greedy clique selection strategy is not necessarily optimal, it should also be noted that its use would require application of the *maximum clique problem*; that is, finding the largest clique in an arbitrary graph (note the contrast to a *maximal* clique, which is a clique whose vertices are not a subset of any other clique).

The maximum clique problem is known to be NP-Complete [52], and as such has exponential time complexity in the number of vertices in the worst case. One of the first exact algorithms for solving the maximum clique problem was the Bron–Kerbosch algorithm [22], and it is still widely used. Modern versions of the algorithm apply pruning techniques to improve the performance of the algorithm, and a worst case time complexity for finding all maximum cliques of $O(3^{\frac{n}{3}})$ has been achieved [84].

Even optimised exact algorithms for solving the maximum clique problem such as these are still subject to the inherent complexity of the problem. Given this, together with the fact that the strategy of finding the largest clique is not even necessarily optimal for use in OBCC, we shall consider approximate solutions to the problem which provide maximal cliques which are not necessarily the maximum. There are many heuristic algorithms which can be used to achieve an approximate solution to the maximum clique problem in polynomial time [75] [19]. These algorithms seem to offer more promise, since iterative application of an exact algorithm would seem to be impractical.

There are a huge number of heuristic approaches to the maximum clique problem, which fall within a number of classes. Sequential greedy heuristic methods begin with a set of vertices which are not a clique and find a clique through either repeated addition of vertices or repeated removal of vertices. These algorithms differ in the heuristic which is used to decide upon the vertex to be added or removed, and can operate very quickly [19]. However, if the running time is to be kept low then the quality of solutions which are found is generally not of a high standard [51].

An alternative approach is to use local search heuristics. While the sequential greedy approach only finds a single maximal clique which may or may not be a maximum clique, local search methods attempt to improve such an approximation by expanding the search space around the neighbourhood of candidate solutions. The solution is then improved until some local optimum is reached. An example of such a method is given by Katayama et al. [53]. One of the weaknesses of local

search methods is that any local optimum may be significantly smaller than the global optimum. Methods such as simulated annealing and tabu search can be used to at least partially alleviate this problem. Simulated annealing has been shown to outperform greedy methods and local search methods when applied to the maximum clique problem [45], particularly in dense graphs. Tabu search has also been applied to the problem with good results [81] [34].

Also of interest are neural approaches to the maximum clique problem. The general approach is to formulate the problem as an energy minimisation, and then to use a network to find the minimisers of the function. For example, the well known Hopfield network [46] has been used to solve the maximum clique problem [48]. Furthermore, Grossman demonstrates that a network based on the Hopfield network and supplemented with an annealing procedure significantly outperforms greedy heuristic methods [39]. Further example of neural approaches to the maximum clique problem can be found in [49], [16] and [31].

In short, there are a number of classes of clique finding algorithms which need to be evaluated with regard to OBCC. These methods vary from those which take considerable time to ensure they find the maximum clique in the graph to those which simply accept the first clique they find but can run extremely quickly. Clearly the algorithms which run in polynomial time show the greatest promise, since in practice we are likely to be applying OBCC to large graphs. The primary question is how much benefit is gained by using a more computationally expensive method which aims to find a closer approximation to the largest clique in the graph? Experimental analysis is required to answer this question.

Beyond this, work then needs to be done to refine the clique selection strategy in order to further improve the performance of OBCC, through techniques such as avoiding cliques whose removal will reduce the total number of cliques in the graph by a significant amount. In addition, given that the graph does not alter significantly between iterations of the optimisation procedure in OBCC, it may be possible to make improvements to the clique finding algorithms to utilise this in order to improve performance.

5.9 Analysis of OBCC

In order to analyse the OBCC method it is helpful to have an application, since it is difficult to represent realistic feature relationships in randomly generated data. One

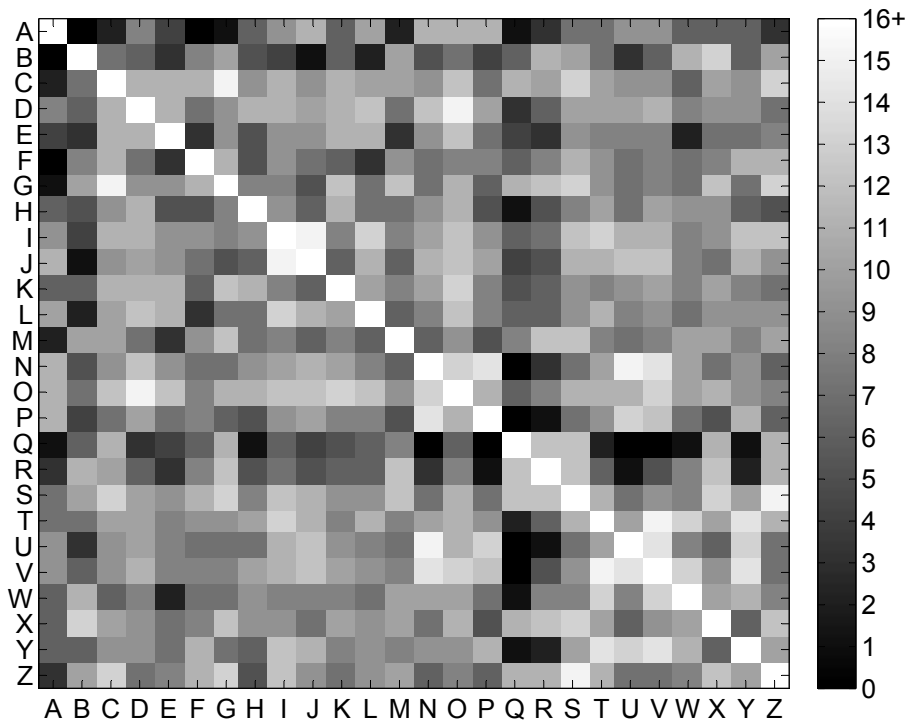


Figure 5.13: A similarity matrix showing the distance classes between characters

suitable problem is that of generating codes to represent characters in the alphabet, represented by a number of features. A character recognition data set was obtained from [30], which contains examples of capital letters along with 16 features. Each of these features is scaled to be an integer between 0 and 15. Further details of the data set are given in Appendix A.

One example of each letter was taken from this dataset (feature values are given in Appendix A), and a distance matrix created by summing the distances between all the features for each pair of letters. This gives a wide range of distances, too large to realistically represent using overlap between binary codes. For this reason, the distances were grouped into 16 classes, through application of k-means clustering. This gives 16 distinct distances between letters, represented by the integers 1-16. A similarity matrix showing the classes to which each distance was assigned is shown in Figure 5.13, with lighter shades representing shorter distances.

At this point, the OBCC framework was applied to the problem. With a problem of this scale it is practical to use the Bron–Kerbosch algorithm [22] to find the maximum clique in the graph in the optimisation process. We used this approach in lieu of a proven alternative, although a variety of other algorithms should be examined in future. We applied the method for a variety of values of p , the maximum distance at which the method considers items to be similar.

Table 5.1 shows the results which were obtained when varying p between 3 and 10. The table shows the length of the code which would have been produced before any optimisation, the length of the code after optimisation has finished, and then the final length of the code n once it has been made fixed weight. The final weight of the code is also given. As the value of p increases further distance classes are taken into account. As one might expect, this also increases the length of the resulting code. The huge decrease in code length provided by the optimisation procedure should be noted, providing almost an order of magnitude decrease when $p = 10$.

One interesting feature of these results is that despite the fact that the unoptimised code becomes very much larger as p increases, the final code length after optimising and making the code fixed length plateaus, with only the weight increasingly significantly after $p = 5$. It appears that the optimisation procedure is able to compress the length of the code increasingly efficiently as p becomes larger, which suggests that many further optimisations are becoming available.

It should also be observed that the final codes are fairly sparse, with the weights being fairly close to the ideal weight of $\log_2 n$ for usage as an input code for a CMM (see Chapter 4 for details of this). This value is shown in the final column. This fact makes the codes quite suitable for usage in a CMM. The sparsity is largely due to the procedure which makes the codes fixed weight.

However, it should also be noted that the codes are very long. Despite the fact that only 26 unique items are being represented, the codes are in the order of 200 bits in length. In order to store 26 unique input codes a CMM may need fewer than 26 input neurons. This suggests that further optimisation procedures may be necessary if the codes are to be stored efficiently in a CMM. We shall make further comment regarding this in a moment.

Figures 5.14, 5.15 and 5.16 show similarity matrices for three of the output codes. As the value of p increases the increased representation of similarity can be observed. Note that some very intuitive similarities can be seen. For example, in Figure 5.14 “G” can be seen to be similar to “C”, “O” to “D”, “J” to “I” and “S” to “Z”. In addition, even once the fidelity of representation is much higher in Figure 5.16 there is still no similarity represented between “H” and “Q” or “C” and “A”.

The OBCC framework operates in a very different way to the existing methods which were outlined in Section 5.5. This makes a comparison between OBCC and these methods very difficult. While codes can be generated using these methods for the problem examined above through the methods described in Section 5.6, they

MaxDist (p)	Length Before Optimisation	Length After Optimisation	Final Length (n)	Weight (l)	Ideal weight ($\log_2 n$)
3	17	13	63	3	6
4	43	26	122	6	7
5	93	46	218	11	8
6	190	63	251	14	8
7	321	80	254	16	8
8	493	101	252	18	8
9	703	109	252	20	8
10	949	129	277	23	8

Table 5.1: Details of codes resulting from application of OBCC to a set of features representing capital letters

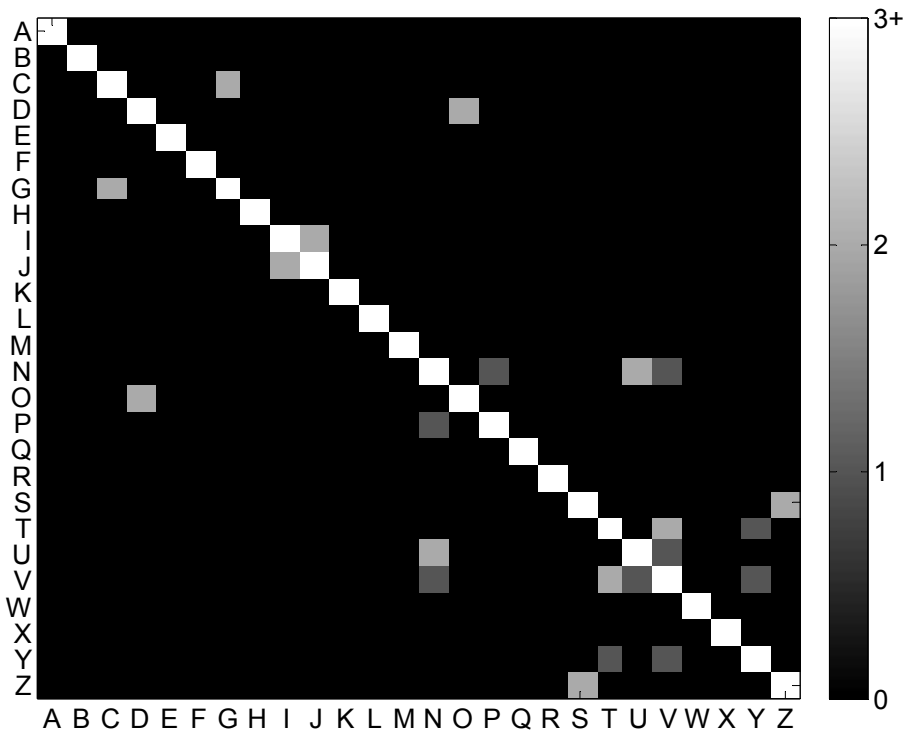


Figure 5.14: A similarity matrix for a code generated for the characters “A” to “Z” using OBCC, with $p = 3$

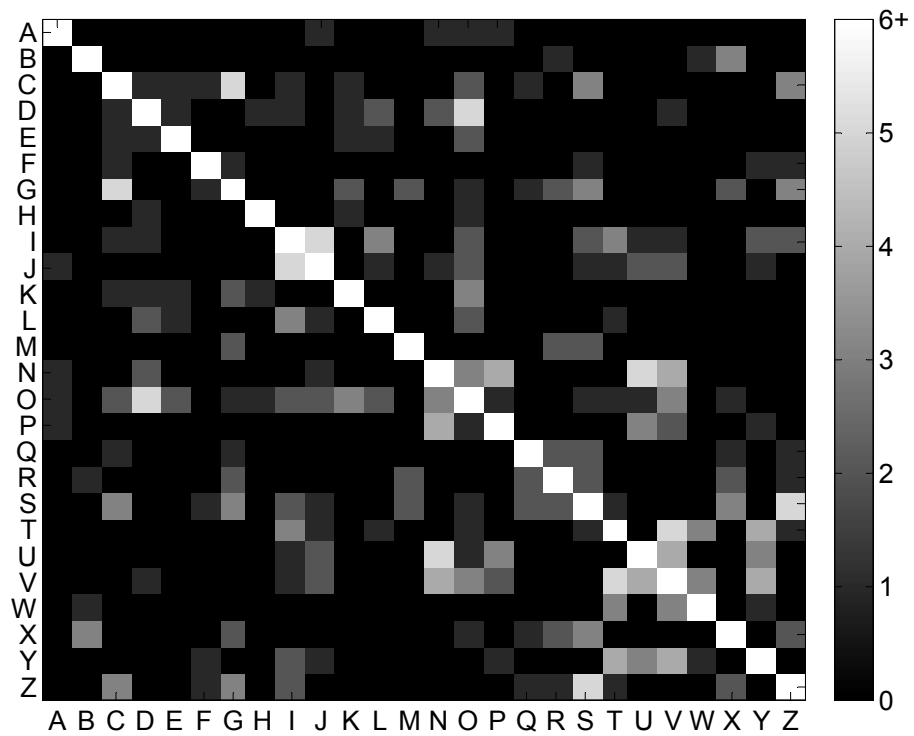


Figure 5.15: A similarity matrix for a code generated for the characters “A” to “Z” using OBCC, with $p = 6$

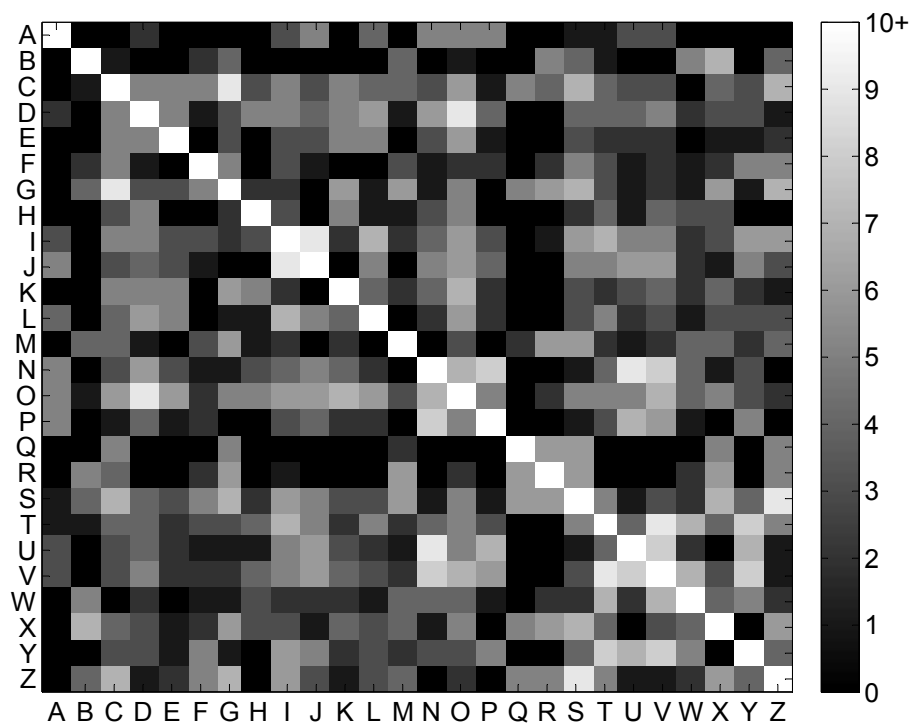


Figure 5.16: A similarity matrix for a code generated for the characters “A” to “Z” using OBCC, with $p = 10$

are not as flexible as OBCC and the resulting code is less sparse. In the given data set there are 16 features, and 17 bits must be used to encode each feature in order to represent them as thermometer codes (in addition, this gives a code length of $16 \times 17 = 272$, comparable to those given in Table 5.1). Such a code must have a minimum weight of 32 in order to represent any similarity at all, allowing 2 bits per feature. This is less sparse than the codes generated using OBCC, making it less suitable for storage in a CMM.

Generating a multi-dimensional thermometer code with the code length and weights described, the similarity matrix shown in Figure 5.17 is achieved. While the representation is reasonable in some ways, the code has a number of disadvantages compared to those generated by OBCC. Firstly, as mentioned, it is significantly less sparse. Secondly, almost every pair of characters has at least some overlap, reducing the ability of a CMM to differentiate between items which are quite different. The represented similarity between some items is higher than one might expect, with “P” and “Z” being an example. These characters are quite different in a number of features, but because a few features are very close the overlap is relatively high. Regardless, it is the flexibility of OBCC which is most compelling. By varying the value of p , the degree to which items are considered similar can be determined optimally for a particular application. This is not so easy to achieve using the alternatives.

There is a further comment which should be made on the comparison between OBCC and the existing methods. As mentioned in Section 5.5, the existing methods make trade-offs between the length of the output code and the quality of the preservation of distances from input space to code space. It should be highlighted that this trade-off is inevitable. Put simply, the reason that thermometer codes are able to preserve distance so well when generating codes for a series of integers, such as seen in Figure 5.5, is because the code is long. Given that the optimisation technique used in OBCC has been designed to perfectly preserve distances between input and code spaces, it also suffers from this problem, with codes becoming quite long. This has been observed in the experiment we have just presented. Whether this is regularly the case needs to be investigated further. In Section 5.10 we will discuss the possibility of a development to the framework which would allow the generation of shorter codes at the expense of the quality of the distance preservation properties of the method.

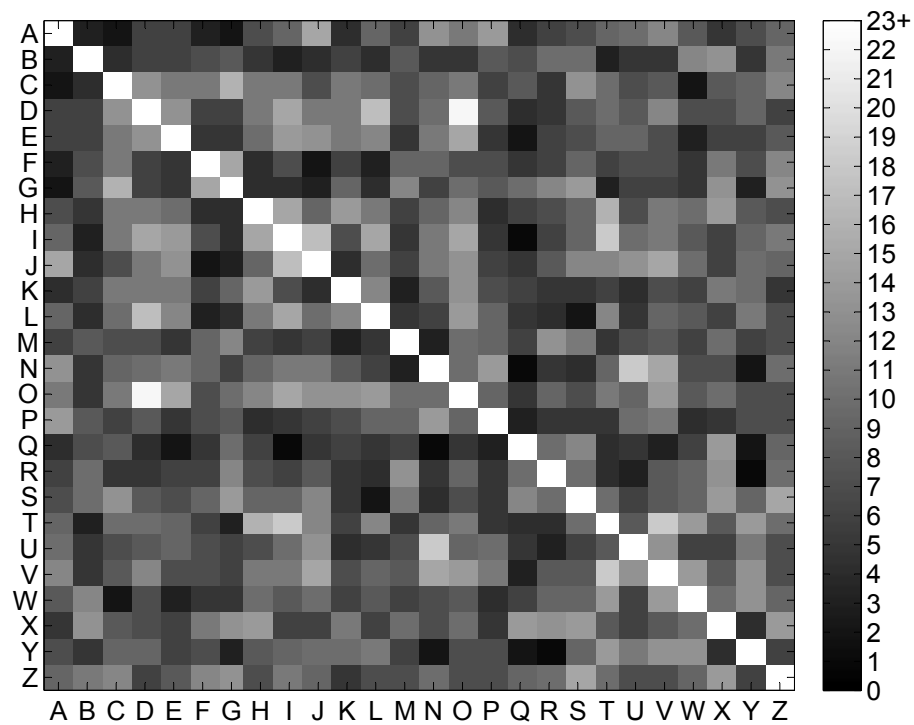


Figure 5.17: A similarity matrix for a code generated for the characters “A” to “Z” using thermometer codes

5.10 Further work

There is much further work required to understand the benefits and limitations of the OBCC optimisation process and framework. As outlined in Sections 5.9 and 5.8, much experimental analysis of OBCC is needed in order to understand both the effect of using various algorithms to find cliques in the optimisation process, and to compare the method further with existing techniques. Having accomplished this, there is a lot of scope for improving the clique finding algorithm, through optimising it specifically for OBCC. This might involve taking advantage of the fact that the graph changes only slightly between iterations, and attempting to prioritise cliques in order to minimise the length of the output code.

In addition to this, there is a further development of OBCC which we alluded to in Section 5.9; altering the technique so that producing a short code is prioritised above perfect preservation of distances from input space to code space. Implementation of this would involve the use of a clique selection strategy which uses methods that find sets of vertices which approximate cliques (i.e. sets of vertices in which the vast majority are pairwise adjacent, but not necessarily all). Use of such methods for finding cliques in the optimisation process would result in the the overlap between

the eventual output codes to be slightly different to that specified in the overlap matrix, but it would also result in a shorter code. In many cases, this may be of benefit. This possibility warrants further investigation.

Finally, the OBCC method should be further applied to real problems in order to evaluate the benefits of the method to generalisation and storage capacity in CMMs. The technique should be applied to the storage of associations in a single CMM in the first instance, but beyond this it should be evaluated on some more complex architectures such as the CANN and ARCA described in Chapter 3. Such an investigation will reveal the true benefits of the technique.

5.11 Summary

We have discussed the relationship between storage capacity and generalisation, and explored the concept of similarity as it relates to the problem of code generation. A survey of existing methods which are able to generate codes in which adjacent items are assigned similar codes has been conducted, and the methods examined and compared. While each method displays its own strength, they do not offer a large degree of flexibility when applied to multi-dimensional input spaces.

The OBCC framework for generating fixed weight codes based on a distance matrix has been presented. The method can be used to produce optimised binary codes in which the overlap between all pairs of codes has been pre-specified in a matrix. This allows the representation of an input with a multi-dimensional feature space, in which the similarity between codes is defined by an arbitrary metric defined by the user. Further work is required in order to experimentally compare OBCC with existing methods, and to improve the clique finding strategy used in the optimisation stage of the process.

Chapter 6

Conclusions and Further Work

6.1 Review

This thesis examined correlation matrix memories (CMMs), and their role within larger architectures. Two architectures were described in detail: the Cellular Associative Neural Network (CANN) [8] and the Associative Rule Chaining Architecture (ARCA) [9]. The CANN is an object recognition system which operates on principles taken from cellular automata. ARCA is capable of performing forward chaining on a set of rules, moving down each layer of the search tree in parallel. In both systems CMMs are used to store and recall the rules which are used to make the various required inferences. The storage properties of CMMs were examined, including the effect of utilising different encodings and thresholding functions. A novel method for thresholding codes generated using the algorithm of Baum et al. [14] was described. Finally, methods for generating codes which enable CMMs to react appropriately to previously unseen data were examined. A novel technique for optimising codes which maintain a predefined level of similarity with one another was presented.

Chapter 3 discussed the Cellular Associative Neural Network (CANN) and Associative Rule Chaining Architecture (ARCA). The CANN consists of a regular grid of associative processors, and is able to perform object recognition through an iterative update procedure. Each iteration every cell exchanges information with its neighbours, eventually resulting in each cell building up a local representation of the object to which the cell belongs. One of the major capabilities of the network is the promise of highly parallel operation. However, there is one issue

which must be overcome in order to achieve this. Since the rules stored at each cell are to be the same, the cellular processors must either share memory or somehow coordinate in order to avoid inconsistent rules being produced in the case that two processors see the same input during the same iteration. However, both of these possibilities present barriers to truly parallel operation. A pessimistic view of the possibility of parallel operation in the CANN was presented, concluding that cells must have additional communication with one another, or share memory (with all the additional difficulties which that presents).

ARCA is able to perform forward chaining on a set of rules using a novel methodology. CMMs are used to store the rules, split across two separate memories for the rule preconditions and the postconditions. Crucially, separate nodes of the search tree are kept separate through the use of *binding* between the tokens and the system states. This allows each level of the tree to be searched in parallel, in contrast to depth first search which must explore each branch of the tree individually. The performance limitations of this architecture had not previously been explored. The architecture was tested on a number of rule sets which involved search trees with various depths and branching factors. Each depth and branching factor was tested on ARCA using a variety of token and rule vector lengths, resulting in a number of different CMM sizes (and therefore memory requirements). It was found that for any given branching factor there is a clear tree depth at which the method fails, with the recall performance falling off extremely rapidly once errors began to appear. While one might hope that in the case that the branching factor is 1 the performance would increase linearly with memory requirement, this was not observed to be the case. It is speculated that this may be due to suboptimal choices of the relative sizes of the token and rule vectors, or their weight. As the branching factor of the search tree increases, the performance of the method becomes worse very quickly indeed, as might be expected due to the huge increase in the number of rules which occurs.

In Chapter 4 the storage capacity of CMMs was considered, with a review of theoretical results being presented. The theory suggests that the optimal storage capacity is achieved when the codes to be stored are sparse, and the set of input codes are close to orthonormal. Specifically, the input weight should be in the order of $\log_2 m$ and the output weight should be in the order of $\log n$, where m and n are the input and output vector lengths respectively. A discussion of methods for generating these codes was presented. The algorithm of Baum et al. for generating sparse fixed weight codes quickly generates codes with a high orthogonality to one another, but produces codes with a lower information content than for other methods. However,

it was shown that a sufficient number of codes can be generated in order to saturate a CMM in the vast majority of cases, provided the full code space is utilised. This suggests that the algorithm is a reasonable choice for a variety of applications.

An analysis of the Willshaw and L-max thresholding mechanisms demonstrated the benefits offered by L-max thresholding in the presence of noise on the input to the CMM. As the level of noise increases, the benefits of L-max thresholding become increasingly pronounced. In addition, two interpretations of Willshaw thresholding were presented, with “Fixed Willshaw” being demonstrated to be the clearly superior alternative; that is, the threshold should be set to the known fixed input weight of the network rather than the weight of the current input vector.

The idea of L-wta thresholding in CMMs storing Baum codes was introduced, and the benefits of the method demonstrated over a variety of input and output vector sizes and weights. The increase in storage capacity when using L-wta rather than L-max was shown to be in the order of 15%. The experimental results support the theoretical results on the storage capabilities of CMMs. In addition, it was demonstrated that networks storing associations between Baum codes are able to store a greater number of associations than the theory suggests is possible for randomly generated codes.

Chapter 5 focused upon the idea of generating codes for storage in a CMM which maximise the capability of the network to generalise. While there are existing methods which exist for solving this problem they have significant weaknesses when faced with the representation of multi-dimensional features spaces. The OBCC method was introduced, allowing the generation and optimisation of codes suitable for storage in a CMM with a predefined degree of overlap between them. This method, therefore, is flexible to feature spaces with any number of dimensions. The optimisation process uses a graph representation of the code and involves finding cliques in this graph. Each clique represents an optimisation to be made in the code, with the size of the clique representing the magnitude of the optimisation. A number of strategies for finding cliques in the graph were discussed, varying from exhaustive approaches which are extremely computationally expensive to methods which find smaller cliques but are able to run extremely quickly. It was speculated that in general finding larger cliques is preferable, but it is certainly not always optimal to pursue the largest clique at the expense of everything else. The method is flexible in that the optimisation need not be completed for the code to be used, and by varying the strictness with which cliques are accepted (i.e. accepting some sets of nodes which are close to being cliques) some quality of representation can be

sacrificed in order to achieve a more compact code. The method was demonstrated on a character recognition data set.

6.2 Further Work

A number of areas for further work have been identified throughout the thesis. The following is a summary of these issues.

6.2.1 Cellular Associative Neural Network

A number of problems with the CANN architecture were discussed. Firstly, the architecture does not display scale or rotational invariance. These are both very desirable properties in an object recognition system. When an object has been learnt by the system, that same object should be recognised whenever it appears regardless of any changes to its size or orientation. Methods for solving this problem need to be developed.

The learning process of the CANN is also limited. Currently the system is only capable of learning a single example of each object. This means that the system is incapable of learning the full gamut of variations within an object class, being only able to recognise very small deviations from the learnt example. In addition, even though the architecture was augmented to perform recalls on uncertain data, the learning process is not capable of using this information. Since real training examples would require a pre-processor applied to image data, adding this capability would allow learning based upon the edge detector presented by Brewer [20].

Finally, the issue of parallelism in the CANN requires further investigation. While a pessimistic view of this possibility was presented, there may be methods which will allow the architecture to operate in a fairly efficient parallel manner. However, it is believed that this will require either a system for efficiently sharing memories between the cellular processors, or a message passing system which ensures that conflicting rules are not learnt during any iteration of the system.

6.2.2 Associative Rule Chaining Architecture

The performance capabilities of ARCA have not been previously studied in detail, so there are many open questions on this subject. There are two different data types used in the system: rule vectors and token vectors. The former are used to represent the rules which have been learnt in the system, with the latter representing the symbols which are used in the rules. The choices made regarding the representations used for these two data types are crucial, since they define the memory requirements of the system and the recall capabilities of the CMMs. The observation was made that the length of the rule vectors n_r is particularly important, since the number of bits required for the two CMMs is $n_r n_t + n_r^2 n_t$ (where n_t is the length of the token vectors). For this reason it is highly preferable to keep the value of n_r as small as possible. However, clearly reducing the size of the vector will also reduce the size of the input to the postcondition CMM, and therefore reduce its storage capacity. In addition, the values chosen should depend upon the number of rules and symbols in the set of rules. The trade-offs involved need to be thoroughly investigated, so that appropriate vector lengths can be chosen for any given problem.

An additional problem is that of selecting the weight of the rule and token vectors. As described in Chapter 4, the optimal weight for an input and output code in a CMM is not the same. This causes a problem in systems such as this, in which codes are used as both input and output codes, since the optimal choice is not clear. In addition, the output of the postcondition CMM is not even simply one of the vectors, but a tensor product of rule and token vectors. This means that weights need to be selected very carefully in order to maximise the performance of the system, and an investigation into how this could be achieved would be very useful.

While the version of ARCA which was discussed and implemented in Chapter 3 was only capable of operating on rules which had a single symbol in the precondition, there is no reason that the system could not be extended to operate on rules with an arity greater than 1. The concept here is basically the same as in AURA, discussed in Section 2.11. A different precondition CMM would be required for each rule arity, with the outputs of these CMMs being combined with a logical OR. This prevents partial matches being returned when only a subset of a rules preconditions have been met. Of course, making such a change is bound to alter the performance of the system, and so this is another possibility which should be investigated.

While ARCA is a very interesting and novel system, it is only capable of performing forward chaining on a very restricted logic. Firstly, the system is not capable of

processing logical negation. Adding this capability would be sufficient to allow the representation of basic propositional logic. While this would increase the power of the system, an even greater improvement would be to extend the system to first-order logic. First-order logic is considerably more expressive than propositional logic, but it is also significantly more complex. Thus, this would be a challenging task.

Finally, although ARCA has been implemented and demonstrated to work on synthetic problems, it has not yet been applied to any real world problems. There are potentially a number of issues which such an undertaking would reveal. For this reason, the identification of an appropriate problem should be made a priority.

6.2.3 Usage of Baum codes in CMMs

The usage of Baum codes for generating codes to store in CMMs has been shown to be effective. However, whether it provides optimal information efficiency is yet to be determined. Although a larger number of associations between Baum codes can be stored in a CMM than randomly generated codes, each code contains less information. This means that the overall information efficiency may be lower, despite the higher storage capacity. It should be determined whether the usage of Baum codes is truly more efficient than other choices for generating codes for storage in CMMs.

In addition, while L-wta was shown to improve the recall capability of a CMM which stores associations between Baum codes when compared to L-max thresholding, this was only confirmed in the absence of noise on the input. Further investigations should be conducted to confirm the benefit L-wta offers when the input is subject to noise. In the absence of noise the improvement to storage capacity was around 15% on average, but this figure could be larger or smaller in this additional case.

6.2.4 Overlapped Binary Code Construction

The Overlapped Binary Code Construction (OBCC) method was presented, but many questions remain regarding the benefits and limitations of the technique. Firstly, there are a wide variety of options available for the clique selection strategy involved in the code optimisation process. Experimental analysis of a variety of these methods is required to assess their relative merits in terms of computational expense, code length minimisation and overlap reproduction accuracy. In addition,

these techniques need to be compared to existing methods such as thermometer codes, fixed weight Gray codes and CMAC Gray codes.

In addition, there is the possibility of optimising a clique finding algorithm specifically tailored for the OBCC method. There are specific properties of the graph which could be exploited, such as the fact that it only changes in a small way between iterations of the algorithm. In addition, it may be possible to prioritise cliques for removal which will result in a code of minimal length.

A further development to OBCC would be to alter the technique to prioritise a shorter code above the perfect preservation of distances between output codes. Accomplishing this would involve the selection of sets of nodes which approximate cliques in the optimisation process, rather than selecting only sets of nodes which form complete cliques. Combining such sets of nodes would result in greater optimisations to the length of the code at the expense of the quality of the representation. The degree to which representation is sacrificed could be controlled by a parameter of the algorithm. This would enable a large increase in flexibility to the method.

Finally, the OBCC method should be further evaluated on real problems. In particular, the effect of the method on storage capacity and generalisation in a single CMM should be investigated, with further studies being conducted into the benefits of the method to more complex architectures. The CANN and ARCA systems described in Chapter 3 would both be suitable for such investigations.

Appendix A

Character Recognition Data Set

A character recognition data set titled “Letter Recognition Data Set” was obtained from the UCI Machine Learning Repository [30]. The data set consists of 20,000 instances of characters, together with 16 features, each of which has been normalised to be an integer between 0 and 15. The features are shown in Table A.1. For the experiment shown in Chapter 5, a single instance of each letter of the alphabet was chosen. Table A.2 gives the feature values for the selected instances.

Name	Description
x-box	Horizontal position of box
y-box	Vertical position of box
width	Width of box
high	Height of box
nopix	Total number of pixels
x-bar	Mean x of pixels in box
y-bar	Mean y of pixels in box
x2bar	Mean x variance
y2bar	Mean y variance
xybar	Mean $x y$ correlation
x2ybr	Mean of $x^2 \times y$
xy2br	Mean of $x \times y^2$
x-ege	Mean edge count, left to right
xegvy	Correlation of x -edge with y
y-ege	Mean edge count, bottom to top
yegvx	Correlation of y -edge with x

Table A.1: The 16 features in the character recognition data set

Letter	x-box	y-box	width	high	nopix	x-bar	y-bar	x2bar	y2bar	xybar	x2ybr	xy2br	x-egge	xegy	y-egge	yegy
A	1	1	3	2	1	8	2	2	2	8	2	8	1	6	2	7
B	5	9	7	7	10	9	8	4	4	6	8	6	6	11	8	7
C	7	10	5	5	2	6	8	6	8	11	7	11	2	8	5	9
D	2	6	3	4	2	6	7	10	8	6	6	6	3	8	3	8
E	3	4	3	6	2	3	8	6	10	7	6	15	0	8	7	8
F	6	9	8	7	4	6	12	6	6	13	7	3	2	9	2	5
G	7	10	7	7	5	5	7	7	6	9	8	10	2	9	5	9
H	4	8	5	5	2	7	5	15	1	7	9	8	3	8	0	8
I	3	7	4	5	2	7	7	0	7	13	6	8	0	8	1	7
J	2	4	4	3	1	8	8	2	7	15	5	8	0	7	1	8
K	4	5	5	8	2	3	6	7	3	7	8	12	3	8	3	10
L	3	6	4	4	2	5	5	1	9	6	2	11	0	8	3	7
M	6	9	9	6	6	5	7	3	5	10	10	9	8	6	3	8
N	2	3	2	1	1	6	9	6	3	8	7	8	4	9	1	7
O	2	6	3	4	2	8	7	7	5	7	6	9	2	8	3	8
P	1	1	2	2	1	5	10	4	4	10	8	4	1	9	3	7
Q	8	15	7	8	4	8	5	4	9	11	4	10	3	6	9	9
R	8	12	8	7	6	9	6	3	5	9	4	8	6	8	6	8
S	4	10	6	7	4	7	8	3	7	10	5	7	2	6	5	8
T	3	7	5	5	3	7	11	1	8	7	11	8	1	10	1	8
U	2	3	3	1	1	4	8	5	6	11	10	8	3	10	1	6
V	2	5	4	4	1	6	11	3	4	8	11	8	2	10	1	8
W	4	6	7	4	4	9	11	2	3	5	9	8	7	11	1	8
X	4	9	6	7	5	8	5	3	5	6	7	8	3	9	9	9
Y	4	5	5	4	2	4	11	2	7	11	10	6	1	11	2	5
Z	5	10	7	8	5	7	8	2	9	12	7	8	1	9	6	7

Table A.2: The features of the selected 26 characters from the data set

Nomenclature

I	The information efficiency of a CMM.
k	The weight of an input code (the number of bits set to 1).
l	The weight of an output code (the number of bits set to 1).
m	The number of input neurons in a CMM. If the number of input and output neurons is the same, n is used to represent both.
n	The number of output neurons in a CMM.
W	The weight matrix of a CMM.
x	An input vector to be stored in a CMM.
y	An output vector to be stored in a CMM.
z	The number of pairs of data stored in a CMM.

Bibliography

- [1] J. S. Albus. A new approach to manipulator control: The cerebellar model articulation controller (CMAC). *Journal of dynamic systems, measurement and control*, 97(3):220–227, 1975.
- [2] D. J. Amit, H. Gutfreund, and H. Sompolinsky. Storing infinite numbers of patterns in a spin-glass model of neural networks. *Physical Review Letters*, 55(14):1530–1533, 1985.
- [3] J. R. Anderson and G. H. Bower. *Human associative memory : a brief edition*. Experimental psychology series. L. Erlbaum Associates, 1980.
- [4] Aristotle. On memory and reminiscence, 350BC.
- [5] J. Austin. ADAM: A distributed associative memory for scene analysis. In M Caudhill and C Butler, editors, *First International Conference on Neural Networks*, volume IV, page 285, San Diego, 1987.
- [6] J. Austin. Parallel distributed computation. In *International Conference on Artificial Neural Networks*, 1992.
- [7] J. Austin. Associative memories and the application of neural networks to vision. In R Beale, editor, *Handbook of Neural Computation*. Oxford University Press, 1995.
- [8] J. Austin. The cellular neural network associative processor, C-NNAP. In *Image Processing and its Applications, 1995., Fifth International Conference on*, pages 622–626. IET, 1995.
- [9] J. Austin. A production system architecture using a neural associative memory. Unpublished manuscript, 2007.

- [10] J. Austin and J. Kennedy. Presence, a hardware implementation of binary neural networks. In *Eighth International Conference on Artificial Neural Networks, Sweden*, volume 1, pages 469–474, 1998.
- [11] J. Austin, J. Kennedy, and K. Lees. A neural architecture for fast rule matching. In *Artificial Neural Networks and Expert Systems Conference*, 1995.
- [12] J. Austin, J. Kennedy, and K. Lees. The advanced uncertain reasoning architecture, AURA. *RAM-based Neural Networks, Ser. Progress in Neural Processing*. World Scientific Publishing, 9:43–50, 1998.
- [13] J. Austin and T. J. Stonham. Distributed associative memory for use in scene analysis. *Image & Vision Computing*, 5(4):251–260, 1987.
- [14] E. B. Baum, J. Moody, and F. Wilczek. Internal representations for associative memory. *Biological Cybernetics*, 59(4):217–228, 1988.
- [15] R. Beale and T. Jackson. *Neural Computing, An Introduction*. IOP Publishing, 1990.
- [16] A. Bertoni, P. Campadelli, and G. Grossi. A neural algorithm for the maximum clique problem: analysis, experiments, and circuit implementation. *Algorithmica*, 33(1):71–88, 2002.
- [17] J. R. Bitner, G. Ehrlich, and E. M. Reingold. Efficient generation of the binary reflected gray code and its applications. *Communications of the ACM*, 19(9):521, 1976.
- [18] W. W. Bledsoe and I. Browning. Pattern recognition and reading by machine. In *Proceedings of the Eastern Joint Computer Conference*, 1959.
- [19] I. M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo. The maximum clique problem. *Handbook of combinatorial optimization*, 4(1):1–74, 1999.
- [20] G. Brewer. A spiking neural edge detector for a neural object recognition system. In *Proceedings of BICS 2006*, 2006.
- [21] G. Brewer. *Spiking Cellular Associative Neural Networks for Pattern Recognition*. PhD thesis, University of York, 2008.
- [22] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

- [23] N. Burles. Quantum parallel computation with neural networks. Master's thesis, University of York, 2010.
- [24] C. Burstedde, K. Klauck, A. Schadschneider, and J. Zittartz. Simulation of pedestrian dynamics using a two-dimensional cellular automaton. *Physica A: Statistical Mechanics and its Applications*, 295(3-4):507–525, 2001.
- [25] D. Casasent and B. Telfer. High capacity pattern recognition associative processors. *Neural Networks*, 5:687–698, 1992.
- [26] T. Cover and P. Hart. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, 13(1):21 – 27, jan 1967.
- [27] P. J. Denning, J. B. Dennis, and J. E. Qualitz. *Machines, Languages and Computation*. Prentice Hall Professional Technical Reference, 1978.
- [28] J. Feldman, M. Fandy, N. Goddard, and K. Lynne. Computing with structured connectionist networks. *Communications of the ACM*, 31(2):170–187, 1988.
- [29] D. Fey and D. Schmidt. Marching-pixels: a new organic computing paradigm for smart sensor processor arrays. In *Conf. Computing Frontiers'05*, pages 1–9, 2005.
- [30] A. Frank and A. Asuncion. UCI machine learning repository, 2010.
- [31] N. Funabiki, Y. Takefuji, and K. Lee. A neural network model for finding a near-maximum clique. *Journal of Parallel and Distributed Computing*, 14(3):340 – 344, 1992.
- [32] E. Gardner. Maximum storage capacity in neural networks. *Europhysics Letters*, 4:481–485, 1987.
- [33] M. Gardner. Mathematical games: The fantastic combinations of John Conway's new solitaire game "Life". *Scientific American*, 223(4):120–123, 1970.
- [34] M. Gendreau, P. Soriano, and L. Salvail. Solving the maximum clique problem using a tabu search approach. *Annals of Operations Research*, 41(4):385–403, 1993.
- [35] B. Graham and D. Willshaw. Improving recall from an associative memory. *Biological Cybernetics*, 72:337–346, 1995.
- [36] B. Graham and D. Willshaw. Information efficiency of the associative net at arbitrary coding rates. In *Proceedings of ICANN96*, pages 35–40, 1996.

- [37] B. Graham and D. Willshaw. Capacity and information efficiency of the associative net. *Network*, 8:35–54, 1997.
- [38] F. Gray. Pulse code communication, 1953. US Patent 2,632,058.
- [39] T. Grossman. Applying the INN model to the MaxClique problem. *DIMACS Series: Second DIMACS Challenge*, pages 125–145, 1993.
- [40] T. R. Hartka. Cellular automata for structural optimization on reconfigurable computers. Master’s thesis, Virginia Polytechnic Institute and State University, 2004.
- [41] S. Haykin. *Neural Networks, A Comprehensive Foundation*. Prentice Hall, second edition, 1999.
- [42] S. Hobson and J. Austin. Improved storage capacity in correlation matrix memories storing fixed weight codes. *Artificial Neural Networks–ICANN 2009*, pages 728–736, 2009.
- [43] S. Hobson and J. Austin. Improved storage capacity in correlation matrix memories storing fixed weight codes. *Journal of Artificial Intelligence and Soft Computing Research*, 1(2), 2011.
- [44] V. Hodge and J. Austin. A comparison of standard spell checking algorithms and a novel binary neural approach. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1073–, 2003.
- [45] S. Homer and M. Peesfado. Experiments with polynomial-time clique approximation algorithms on very large graphs. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, 26:147, 1996.
- [46] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America*, 79(8):2554–2558, 1982.
- [47] Austin J. and Filer R. Using neural networks for inferencing in expert systems. In Taylor J. G., editor, *Neural Networks and Their Applications*. Wiley-Blackwell, 1996.
- [48] A. Jagota. Approximating maximum clique with a Hopfield network. *Neural Networks, IEEE Transactions on*, 6(3):724 –735, may 1995.

- [49] A. Jagota, L. Sanchis, and R. Ganesan. Approximately solving maximum clique using neural network and related heuristics. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, 26:169, 1996.
- [50] A. K. Jain, J. Mao, and K. M. Mohiuddin. Artificial neural networks: A tutorial. *Computer*, 29(3):31–44, 1996.
- [51] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256–278, 1974.
- [52] R. M. Karp. Reducibility among combinatorial problems. *50 Years of Integer Programming 1958-2008*, pages 219–241, 1972.
- [53] K. Katayama, A. Hamamoto, and H. Narihisa. An effective local search for the maximum clique problem. *Information Processing Letters*, 95(5):503 – 511, 2005.
- [54] J. Kennedy and J. Austin. A hardware implementation of a binary neural associative memory. 1994.
- [55] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
- [56] S. Klinger. *Chemical Similarity Searching with neural graph matching methods*. PhD thesis, University of York, 2006.
- [57] D. E. Knuth. *The art of computer programming, Vol. 3*. 1973.
- [58] T. Kohonen. Correlation matrix memories. *IEEE Transactions on Computers*, 100(4):353–359, 1972.
- [59] T. Kohonen. *Associative Memory: A System-Theoretical Approach*. Berlin, 1977.
- [60] T. Kohonen. *Self-Organization and Associative Memory*. Springer-Verlag, third edition, 1989.
- [61] A. Kolcz and N. Allinson. Enhanced N-tuple approximators. In *Weightless Neural Network Workshop*, pages 38–45, 1993.
- [62] A. Kolcz and N. Allinson. Application of the CMAC input encoding scheme in the N-tuple approximation network. *IEE Proceedings-Computers and Digital Techniques*, 141(3):177–183, 1994.

- [63] B. Kosko. Bidirectional associative memories. *Systems, Man and Cybernetics, IEEE Transactions on*, 18(1):49–60, 1988.
- [64] D. Kustrin. *Forecasting Financial Time Series with Correlation Matrix Memories for Tactical Asset Allocation*. PhD thesis, University of York, 1998.
- [65] D. Kustrin and J. Austin. Spiking correlation matrix memory. In *International Workshop on Emergent Neural Computational Architectures Based on Neuroscience*, 1999.
- [66] P. Maji, C. Shaw, N. Ganguly, B. K. Sikdar, and P. P. Chaudhuri. Theory and application of cellular automata for pattern classification. *Fundamenta Informaticae*, 58(3):321–354, 2003.
- [67] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 5(4):115–133, 1943.
- [68] J. P. Nadal and G. Toulouse. Information storage in sparsely coded memory nets. *Network: Computation in Neural Systems*, 1(1):61–74, 1990.
- [69] H. Niesche. Introduction to cellular automata. In *Seminar Organic Computing SS2006*, 2006.
- [70] C. Orovas. *Cellular Associative Neural Networks for Pattern Recognition*. PhD thesis, University of York, 1999.
- [71] C. Orovas and J. Austin. A cellular neural associative array for symbolic vision. In *NIPS*, 1998.
- [72] G. Palm. On associative memory. *Biological Cybernetics*, 36(1):19–31, 1980.
- [73] G. Palm. Neural assemblies. chapter On the Storage Capacity of Associative Memories, pages 192–199. Springer-Verlag, 1982.
- [74] G. Palm, F. Schwenker, F. Sommer, and A. Strey. Neural associative memories. In A. Krikelis and C. Weems, editors, *Associative Processing and Processors*, pages 307–326. IEEE Computer Society, 1997.
- [75] P. M. Pardalos and J. Xue. The maximum clique problem. *Journal of Global Optimization*, 4(3):301–328, 1994.
- [76] H. Ritter, T. Martinetz, and K. Schulten. *Neural computation and self-organizing maps: an introduction*. Addison-Wesley, 1992.

- [77] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning internal representations by error propagation*, pages 318–362. MIT Press, Cambridge, MA, USA, 1986.
- [78] S. J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Prentice Hall Englewood Cliffs, NJ, second edition, 2003.
- [79] G. Smith and J. Austin. Discovering textures with a metric based on image structure. In *Proceedings of International Conference on Artificial Neural Networks, Brighton (Sept. 1992)*.
- [80] G. Smith and J. Austin. Analysing aerial photographs with adam. In *Neural Networks, 1992. IJCNN., International Joint Conference on*, volume 3, pages 49–54. IEEE, 1992.
- [81] P. Soriano and M. Gendreau. Diversification strategies in tabu search algorithms for the maximum clique problem. *Annals of Operations Research*, 63(2):189–207, 1996.
- [82] E. Tanaka. Theoretical aspects of syntactic pattern recognition. *Pattern Recognition*, 28(7):1053–1061, 1995.
- [83] K. Tombre. Structural and syntactic methods in line drawing analysis: To which extent do they work? *Structural and Syntactical Pattern Recognition (SSPR)*, 1996.
- [84] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363(1):28 – 42, 2006. Computing and Combinatorics, 10th Annual International Conference on Computing and Combinatorics (COCOON 2004).
- [85] A. Turner and J. Austin. A novel method for producing near orthogonal codes. Technical report, University of York, May 1995.
- [86] M. Turner and J. Austin. Matching performance of binary correlation matrix memories. *Neural Networks*, 10(9):1637–1648, 1997.
- [87] J. Von Neumann and A. W. Burks. Theory of self-reproducing automata. 1966.
- [88] P. S. Wang. An application of array grammars to clustering analysis for syntactic patterns. *Pattern recognition*, 17(4):441–451, 1984.

- [89] D. J. Willshaw, O. P. Buneman, and H. C. Longuet-Higgins. Non-holographic associative memory. *Nature*, 222(7):960–962, 1969.
- [90] S. Wolfram. *Cellular Automata and Complexity: Collected Papers*. Perseus Books Group, 1994.