Pattern Recognition Using Associative Memories

Nathan John Burles

Submitted for the degree of Doctor of Philosophy

University of York

Department of Computer Science

April 2014

Dedication

For Victoria

Abstract

The human brain is extremely effective at performing pattern recognition, even in the presence of noisy or distorted inputs. Artificial neural networks attempt to imitate the structure of the brain, often with a view to mimicking its success. The binary correlation matrix memory (CMM) is a particular type of neural network that is capable of learning and recalling associations extremely quickly, as well as displaying a high storage capacity and having the ability to generalise from patterns already learned. CMMs have been used as a major component of larger architectures designed to solve a wide range of problems, such as rule chaining, character recognition, or more general pattern recognition. It is clear that the memory requirement of the CMMs will thus have a significant impact on the scalability of such architectures.

A domain specific language for binary CMMs is developed, alongside an implementation that uses an efficient storage mechanism which allows memory usage to scale linearly with the number of associations stored. An architecture for rule chaining is then examined in detail, showing that the problem of scalability is indeed settled before identifying and resolving a number of important limitations to its capabilities. Finally an architecture for pattern recognition is investigated, and a memory efficient method to incorporate general invariance into this architecture is presented—this is specifically tested with scale invariance, although the mechanism can be used with other types of invariance such as skew or rotation.

Contents

Abstract **3**

List of Figures **9**

List of Tables **13**

Acknowledgements **15**

Declaration **17**

1 Introduction **19**

1.1 Motivation 19

1.2 Thesis Summary 20

1.3 Chapter Overview 22

2 Literature Review **25**

2.1 Introduction 25

2.2 Artificial Neural Networks 26

2.2.1 Feed-Forward Neural Networks 27

2.3 Associative Memories 28

2.3.1 Hopfield Networks 30

2.3.2 Tensor Products 32

2.3.3 Correlation Matrix Memories 34

2.3.4 Capacity of Correlation Matrix Memories 38

2.3.5 Threshold Functions 40

2.3.6 Alternatives to Tensor Products 43

2.4 Distributed Computation 46

2.4.1 Tensor Product Production System 47

2.4.2 Parallel Distributed Computation 47
2.5 Architectures for Pattern Recognition 50
2.5.1 Neocognitron 50
2.5.2 Multiple Interacting Instantiations of Neural Dynamics 51
2.5.3 Cellular Associative Neural Network 52
2.6 Summary 57

3 The Extended Neural Associative Memory Language 59
3.1 Introduction 59
3.2 The Extended Neural Associative Memory Language 59
3.2.1 Simple Operators 60
3.2.2 Advanced Operators 61
3.2.3 Compound Operators and Additional Functions 61
3.2.4 Combining Operations 61
3.3 Storage Mechanisms 62
3.3.1 Experimentation 64
3.3.2 Results 67
3.3.3 Independently Varying the Vector Weights 72
3.4 Further Work 75
3.4.1 Independently Varying the Vector Weights 75
3.4.2 Mathematical Analysis of CMM Capacity 75
3.5 Summary 76

4 The Associative Rule Chaining Architecture 77
4.1 Introduction 77
4.2 Rule Chaining 77
4.3 The Associative Rule Chaining Architecture 79
4.3.1 Training 82
4.3.2 Recall 82
4.3.3 Experimentation 86
4.3.4 Results 88
4.3.5 Using a Sparse Matrix Representation 94
4.4 Reducing the Memory Requirements 98
4.4.1 Training 100

4.4.2 Recall . . . 100
4.4.3 Experimentation . . . 102
4.5 Multiple Arity Rules . . . 109
4.5.1 Training . . . 112
4.5.2 Recall . . . 112
4.5.3 Experimentation . . . 113
4.6 Pathfinding . . . 116
4.7 Application to Solitaire . . . 118
4.7.1 Configuration . . . 119
4.7.2 Experimentation . . . 120
4.8 Further Work . . . 124
4.8.1 Vector Lengths and Weights . . . 124
4.8.2 Graphs . . . 124
4.8.3 Weighted Edges . . . 125
4.9 Summary . . . 125

5 The Cellular Associative Neural Network 127
5.1 Introduction . . . 127
5.2 The Architecture . . . 127
5.2.1 Learning . . . 129
5.2.2 Recall . . . 130
5.3 Removing Arity Networks From the CANN . . . 131
5.3.1 Use of a `NULL` Vector . . . 132
5.3.2 Relaxation . . . 132
5.4 Incorporating Scale Invariance . . . 134
5.4.1 Recall . . . 135
5.4.2 Initial Experimentation . . . 135
5.4.3 Further Experimentation . . . 137
5.5 Noisy Inputs . . . 139
5.6 Photographs . . . 142
5.7 Further Work . . . 144
5.7.1 Image Scaling . . . 144
5.7.2 Rotation Invariance . . . 144
5.7.3 Primitive Features Used . . . 145

5.7.4 Superposition of Primitives . . . 145
5.8 Summary . . . 146

6 Conclusions **147**
6.1 Introduction . . . 147
6.2 The Extended Neural Associative Memory Language . . . 148
6.3 The Associative Rule Chaining Architecture . . . 149
6.3.1 Addressing the Memory Requirement of the Architecture . . . 149
6.3.2 Incorporating Multiple Arity . . . 151
6.3.3 Demonstrating Pathfinding . . . 152
6.3.4 Summary . . . 152
6.4 The Cellular Associative Neural Network . . . 153
6.4.1 Simplifying the Architecture . . . 153
6.4.2 Addition of Tensor Products . . . 154
6.4.3 Summary . . . 155
6.5 General Conclusions . . . 155
6.6 Further Work . . . 156
6.6.1 The Extended Neural Associative Memory Language . . . 157
6.6.2 The Associative Rule Chaining Architecture . . . 158
6.6.3 The Cellular Associative Neural Network . . . 159

Appendix **161**

A ENAMeL Code **161**
A.1 Minimal Example of ENAMeL Code . . . 161
A.1.1 MATLAB Equivalent . . . 162
A.2 ENAMeL Code Used for ARCA . . . 164
A.2.1 MATLAB equivalent . . . 167

Abbreviations and Nomenclature **171**

References **173**

List of Figures

2.1 A multilayer perceptron 27
2.2 A Hopfield network with four neurons 31
2.3 A binary correlation matrix memory 35
2.4 A small binary CMM, represented as a matrix and a feed-forward neural network 35
2.5 Three tensor products, given in both matrix form and tensor product representation 39
2.6 A set of associations represented as a tree 40
2.7 Circular convolution with vectors of length 3 45
2.8 Circular correlation with vectors of length 3 45
2.9 Example non-deterministic finite state automaton 48
2.10 A CMM as a state machine 49
2.11 Baum codes generated with $n = 5$, $s = 2$, $p_1 = 2$, $p_2 = 3$, demonstrating a potential issue distinguishing overlapping vectors 49
2.12 A typical architecture of the Neocognitron 51
2.13 Decomposition of part of a square into simple primitives 53
2.14 The "Original Architecture" CANN module configuration 56

3.1 An example of the Yale format 64
3.2 An example of the binary Yale format 64
3.3 An example of the hybrid format 65
3.4 Scatter plots showing the memory requirements of a CMM using different storage formats, with short and long vectors 68
3.5 Scatter plots showing the memory requirements of a CMM using different storage formats, with different input and output vector lengths 69

3.6 Scatter plots showing the memory requirements of a CMM using different storage formats, with different input and output vector weights 71
3.7 Heat map showing how the capacity of a CMM changes with vector weights 73

4.1 An example set of rules represented as a list and a tree 78
4.2 Block diagram of ARCA 80
4.3 A visualisation of the tensor product $(\mathbf{b} : \mathbf{r_1}) \vee (\mathbf{c} : \mathbf{r_5})$ 83
4.4 A visualisation of two iterations of the rule chaining process within ARCA . 84
4.5 Contour plots showing the recall performance of ARCA for branching factors 1 and 2 (non-sparse), plotting memory required against tree depth . . . 89
4.6 Contour plots showing the recall performance of ARCA for branching factors 1 and 2 (non-sparse), plotting memory required against number of rules 90
4.7 Scatter plots showing the recall performance of ARCA for branching factors 1 and 2 (non-sparse) 92
4.8 Scatter plots showing the recall performance of ARCA for branching factors 3 and 4 (non-sparse) 93
4.9 Scatter plots showing the recall performance of ARCA for branching factors 1 and 2 (binary Yale format) 96
4.10 Scatter plots showing the recall performance of ARCA for branching factors 1 and 2 (hybrid format) 97
4.11 Block diagram of the single CMM ARCA 99
4.12 A visualisation of two iterations of the rule chaining process within the single CMM ARCA 101
4.13 Scatter plots showing the memory requirements of one- and two-CMM ARCA for branching factors 1 and 2 104
4.14 Scatter plots showing the memory requirements of one- and two-CMM ARCA for branching factors 3 and 4 105
4.15 Scatter plots showing the memory requirements of one- and two-CMM ARCA for branching factor 1 with changing vector lengths 107
4.16 Scatter plots showing the memory requirements of one- and two-CMM ARCA for branching factor 2 with changing vector lengths 108
4.17 Block diagram of multiple arity networks with ARCA 111
4.18 Block diagram of multiple weight networks with ARCA 111

4.19 Scatter plots showing the memory requirements of ARCA with multiple arity rules . . . 114
4.20 Scatter plots comparing the memory requirements of ARCA with single and multiple arity rules . . . 115
4.21 An example tree of rules, where only the path to be found is labelled . . . 116
4.22 Mechanisms used in order to find the path between two nodes . . . 117
4.23 An English solitaire board . . . 119
4.24 Plots showing the number of operations required by a DFS to determine if a Solitaire state is valid or invalid, compared to the single operation ARCA requires . . . 121
4.25 Scatter plot comparing the number of operations required by a DFS to that required by ARCA to find a solution for a given state . . . 123

5.1 An example of the CANN recognising a simple shape . . . 128
5.2 The "Corner Turning 2" CANN module configuration . . . 129
5.3 Shapes to demonstrate the CANN's arity network limitations . . . 132
5.4 The 8 patterns trained into the scale invariant CANN . . . 135
5.5 Feature extraction for the scale invariant CANN . . . 136
5.6 An example of the noisy inputs presented to the CANN for recall . . . 139
5.7 Plots showing the recall performance of the CANN with noisy inputs presented at scales 25%–100% . . . 140
5.8 Plots showing the recall performance of the CANN with noisy inputs presented at scales 125%–200% . . . 141
5.9 Application of the scale invariant CANN to photographs of the Eiffel Tower 143

List of Tables

2.1 Example tokens, each allocated a binary vector generated using Baum's algorithm with $n = 12$, $s = 2$, $p_1 = 5$, $p_2 = 7$ 37

2.2 Baum codes generated with $n = 12$, $s = 3$, $p_1 = 3$, $p_2 = 4$, $p_3 = 5$ 43

3.1 Simple operators available in ENAMeL 60

3.2 Advanced operators available in ENAMeL 61

3.3 Additional and compound operators available in ENAMeL 62

3.4 The memory required to associate vector pairs in CMMs of varying size . . 69

3.5 Capacity of a CMM with independently varied input and output weights . . 74

4.1 A subset of the rules given in Figure 4.1, with a binary vector assigned to each individual vector and rule 81

4.2 A set of rules to demonstrate the difficulty with multiple arity, with a binary vector assigned to each individual token vector 110

5.1 Relaxation options for the CANN 133

5.2 Recall results for the scale invariant CANN 137

5.3 Recall results for the scale invariant CANN using feature extraction 138

Acknowledgements

I wish to thank my supervisors, Jim Austin and Simon O'Keefe, for their invaluable guidance, feedback, and support. I would also like to thank my assessors, Dimitar Kazakov and Marc de Kamps, for giving up their valuable time to assist with this project. My gratitude is extended to the Department and the EPSRC for providing financial support for this research. Finally I would like to thank my mother, Tina, for help with proof-reading this thesis, and most of all my wife Victoria, for her support and encouragement throughout.

Declaration

I declare that the research described in this thesis is original work, which I undertook at the University of York during 2010 - 2014. Except where stated, all of the work contained within this thesis represents the original contribution of the author.

Some of the material in Chapters 3, 4, and 5 have been previously published; where items were published jointly with collaborators, the author of this thesis is responsible for the material presented here.

- N. Burles, S. O'Keefe, and J. Austin. Incorporating scale invariance into the cellular associative neural network. In *Artificial Neural Networks and Machine Learning–ICANN 2014*, pages 435–442. Springer, 2014

- N. Burles, S. O'Keefe, J. Austin, and S. Hobson. ENAMeL: A language for binary correlation matrix memories. *Neural Processing Letters*, 40(1):1–23, 2014

- N. Burles, S. O'Keefe, and J. Austin. Improving the associative rule chaining architecture. In *Artificial Neural Networks and Machine Learning–ICANN 2013*, pages 98–105. Springer, 2013

- N. Burles, J. Austin, and S. O'Keefe. Extending the associative rule chaining architecture for multiple arity rules. In *Neural-Symbolic Learning and Reasoning Workshop*, pages 47–51, Beijing, 5 August 2013

- J. Austin, S. Hobson, N. Burles, and S. O'Keefe. A rule chaining architecture using a correlation matrix memory. In *Artificial Neural Networks and Machine Learning–ICANN 2012*, pages 49–56. Springer, 2012

Chapter 1

Introduction

1.1 Motivation

Binary correlation matrix memories (CMMs) are a type of artificial neural network, specifically they are associative memories. They have been shown to have a high capacity, storing a large number of associations between binary vectors [112], and their simplicity allows for very fast storage and recall. CMMs have an ability to *generalise*, to recall appropriate associations when presented with an unseen input that is similar to known inputs, as well as to perform correctly in the presence of noisy inputs. Binary CMMs, and architectures based on them, have been effectively applied to problems in a wide range of domains, including traffic management [53, 73], pattern recognition [16, 77], rule-based systems [9, 10], and graph matching [62]. Although these architectures have been shown to be very capable, some of them have a number of problems or limitations which reduce their applicability.

There are three main foci in this thesis. The first covers the problems which may be encountered when attempting to incorporate binary CMMs into an architecture. Although binary CMMs are capable of storing a large number of associations between input and output vectors, their memory requirement is often unnecessarily high due to the use of inefficient storage mechanisms—especially so when the CMM is not saturated. Implementing CMMs efficiently, both in terms of memory and time, requires domain specific knowledge that can act as a barrier for entry to the field. A simple implementation which stores the matrices efficiently can greatly increase the scalability of architectures which incorporate CMMs.

The second area focused upon is a rule-based architecture—the Associative Rule Chaining Architecture (ARCA). This architecture is able to infer knowledge from a set of rules,

given a starting state, and can be applied to any rule-based problem such as rule chaining. It is limited, however, in terms of both applicability and memory requirement. As the number of rules is increased the memory requirement increases exponentially, meaning that the architecture will not scale to larger problems. In addition to this, all the rules in a system must have the same arity—the same number of antecedents. Although this may be sufficient in many cases, the ability to store multiple arity rules would be extremely beneficial to allow the architecture to be applied to more complicated sets of rules or other problems where the arity is likely to vary between rules, such as feature matching.

The final focus of this thesis is an architecture for distributed symbolic computation—the Cellular Associative Neural Network (CANN). The CANN is another example of a rule-based architecture, and is specifically applied to syntactic pattern recognition. Previous work has shown that the architecture is capable of performing pattern recognition successfully [76], even in the case of noisy inputs and when using photographic images rather than symbolically encoded data [16]. While the design of the CANN ensures that it supports translation invariant pattern recognition innately, it is distinctly limited in its suitability for this general task due to the lack of support for scale or rotation invariance. A solution to incorporating general purpose invariance would enable the CANN to be applied to a far wider range of pattern recognition problems.

The subject of this thesis is pattern recognition using rule-based CMM architectures, aiming to develop and extend these architectures in order to enhance their suitability to their respective applications. The project also aims to reduce the memory requirements of CMMs in order to improve the scalability of any architectures using them, including those designed for pattern recognition.

1.2 Thesis Summary

The previous section presented a brief discussion of the motivation behind this project. It explained the limitations of two CMM-based architectures, and modifications or extensions that are required in order that the architectures—and CMMs in general—may become more suited to their intended purposes.

The overall aim of this project is to resolve the problems identified in the ARCA and CANN systems, two architectures capable of performing rule-based pattern recognition. Although these architectures have previously been shown to be sufficiently capable, the overarching hypothesis of this work is that CMM-based architectures can be effective

and proficient for use with rule-based systems, demonstrating efficient memory usage and potential to scale.

Below is a short list of any contributions made in this thesis:

- The development of a domain specific language for use with binary correlation matrix memories (Section 3.2), and an interpreter for this language that utilises an efficient storage mechanism to allow large-scale simulations to be run (Section 3.3.2).

- An experimental analysis of the Associative Rule Chaining Architecture, comparing the memory requirement to the number of rules stored (Section 4.3.4), as well as when changing the relative length of different vector types in the system (Section 4.4.3).

- Two methods to reduce the growth of memory required with respect to the number of rules from exponential to linear—using a sparse storage mechanism (Section 4.3.5) and modifying the architecture (Section 4.4.3).

- Identification of a limitation of the "arity networks" proposed in previous work and provision of an alternative (Section 4.5) that has been tested experimentally—allowing the Associative Rule Chaining Architecture to be successfully applied to any directed acyclic graph or forest (Section 4.5.3).

- Demonstration that it is possible to find the path between two nodes using the Associative Rule Chaining Architecture, the first time this has been addressed (Section 4.6).

- Application of the Associative Rule Chaining Architecture to a real problem, that of creating a Solitaire solver, providing a comparison between the number of operations required by alternative methods of tree search (Section 4.7).

- Simplification of the architecture of the Cellular Associative Neural Network, in order that further improvements could be made (Section 5.3).

- Incorporation of a general purpose solution to handling invariance in the Cellular Associative Neural Network (Section 5.4), including a test of this applied to scale invariance with symbolic images (Section 5.4.2), noisy images (Section 5.5), and images using generic feature extraction (Sections 5.4.3 and 5.6).

1.3 Chapter Overview

Chapter 2 begins with a discussion of associative memories, introducing some background on artificial neural networks before presenting *associationism* and types of associative memory—particularly the correlation matrix memory (CMM). Subsequently two neural-network based architectures for object recognition are reviewed, presenting their salient features as a means to reveal the motivations behind their development.

Chapter 3 introduces a domain specific language (DSL) for use with binary CMMs—the Extended Neural Associative Memory Language (ENAMeL). This DSL is intended to lower the requirements for entry to research using binary CMMs, as well as to provide an efficient and scalable mechanism for storing binary CMMs. As such, following the description of the language is an investigation of storage methods that may be employed.

In Chapter 4, a detailed description of a CMM-based architecture is given—the Associative Rule Chaining Architecture (ARCA). The aim of ARCA is to be an efficient method to perform forward chaining—inference of information by repeated application of a set of rules to a state. More generically ARCA implements a tree search; whereas a traditional method such as depth-first or breadth-first search looks at each node in turn, ARCA inspects an entire layer of the tree simultaneously. After describing the existing architecture, a number of important improvements are introduced. Firstly the use of an efficient storage method in ENAMeL reduces the rate of growth of memory required, with respect to the number of rules stored, from exponential to linear. The architecture is then modified in order to reduce the memory requirements further. Notably this also reduces the rate of memory growth from exponential to linear, in the event that the use of an efficient storage mechanism is not possible. Next an extension to ARCA is introduced which allows multiple arity rules to be successfully stored and recalled, before a method is described to perform pathfinding using ARCA. Finally, ARCA is applied to the game of Solitaire. This demonstrates that ARCA can successfully scale to large problems—in this case 185.9M rules—and that the pathfinding method works correctly.

Chapter 5 focuses on another CMM-based architecture—the Cellular Associative Neural Network (CANN). This architecture was developed to perform translation-invariant syntactic pattern recognition, using ideas from cellular automata. The CANN consists of a network of simple processing cells, arranged in a regular grid, which in essence is overlaid on an image. Each cell uses a small section of the image as its input, and by exchanging information with its neighbours gradually builds a view of the object. Recognition is

performed hierarchically; tokens represent individual features within an image, and rules combine them until an object is recognised. Rules are stored in CMMs in order to take advantage of their speed during both training and recall, as well as to provide support for partial matching in the case of distorted or occluded objects. While the CANN has previously been shown to be effective, it is limited in its utility as it is not able to perform scale or rotation invariant recognition. In order to introduce an extension allowing such invariance the architecture is first modified to remove the "arity networks". The capabilities of the CANN are then augmented to allow invariant pattern recognition—this is applied specifically to scale invariance, however the mechanism is suitable for adaptation to other types of invariance such as rotation, skew, or stretch.

Finally, Chapter 6 conducts a review of the work presented in this thesis, discussing and evaluating the degree to which the work completed meets the motivations of the project. This chapter also combines any conclusions from the work presented throughout this thesis, and draws any final and overarching conclusions from these. The thesis concludes with a summary of any area which would benefit from further study.

Chapter 2

Literature Review

2.1 Introduction

Traditional computing tends to place emphasis on obtaining the exact solution to a problem; neural networks are better suited to a "fuzzy" approach, finding an approximate or good enough result. The cost of certainty and precision in some problems can be large—they can become exponentially more difficult as the required precision increases. An example given by Zadeh [116] is that of a human parking a car. With a large space to park in, the problem is simple; as the space is reduced, or the required precision increased, the difficulty of parking grows rapidly. As such, being imprecise can greatly reduce the computation required to solve a problem, if the situation can tolerate some uncertainty.

Human cognitive abilities, such as understanding distorted speech and recognising images, stem from our ability to tolerate imprecision and uncertainty [116]. Pattern recognition in particular is an area that is well suited to a brain-inspired approach—our ability to recognise similar objects in different positions, invariant of scale and rotation, is due to our ability to generalise.

In this chapter we first examine artificial neural networks, and in particular associative memories. Following this we discuss distributed computation, and the use of a correlation matrix memory as a state machine—a platform for symbolic computing. Finally, a number of neural network-based architectures for pattern recognition are investigated. The work reviewed in the following sections provides the background necessary to understand and illustrate the limitations of current approaches, and the problems which this thesis has addressed.

2.2 Artificial Neural Networks

The human brain is incredibly complex, composed of around one hundred billion neurons and many times more synapses [111]. These can be mapped during dissection to give a good idea of the cellular and network structure of the brain. Alternatively, functional magnetic resonance imaging (fMRI) can be used to determine the areas within the brain that are used for particular tasks. These techniques do not tell us how the brain functions, but in the connectionist approach it is assumed that since individual neurons are very basic that the higher-level capabilities come from the structure.

Artificial Neural Networks (ANNs), often simply called neural networks, are systems designed to mimic the structure of the brain. McCulloch and Pitts introduced simplified neurons, as a model of biological neurons, in 1943 [68]. Research interest in ANNs grew with the advent of back-propagation [89], as hardware development increased the size and capabilities of these networks [65]. ANNs simulate the behaviour or structure of animal brains, with many self-contained artificial neurons operating independently of each other. Although ANNs are inspired by nature, they are not constrained by it and many different types of ANN have been developed—some of these are designed as an attempt to solve a specific problem, where others are an attempt to create a more general parallel computation architecture.

There are many different types of neural network, generally separated into three classes of learning—supervised learning, unsupervised learning, and reinforcement learning. Each of these classes has particular types of problems to which it is better suited [59]. Supervised learning requires pairs of input and output patterns to be presented to a network. The network attempts to minimise the error between its own output and the expected output, when presented with a particular input. In unsupervised learning, on the other hand, only input patterns are presented. The network is provided with a cost function which it must attempt to minimise—this function can be based on the input data and the network's output.

Reinforcement learning often uses a very different approach and explicit input or output data may not presented at all, instead an agent learns from interactions with an environment. Numerical "rewards" denote the success of a particular action and an agent seeks to maximise the reward obtained in the long-term. The cost associated with an interaction is often also recorded, and taken into account when calculating the total reward.

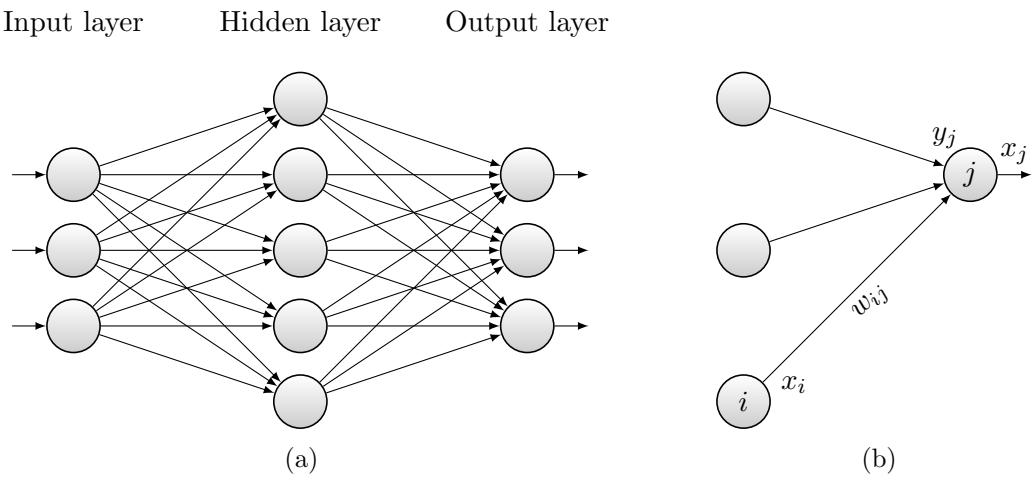


Figure 2.1: (a) A multilayer perceptron with a single hidden-layer and (b) inputs to a single neuron.

2.2.1 Feed-Forward Neural Networks

A feed-forward neural network is one of the simplest types of network to understand, and is often employed for classification. The neurons are arranged in layers, and connections must always go forward from a layer, creating a directed acyclic graph—there are no connections that go backwards or between nodes within a layer [47].

The multilayer perceptron (MLP) is a commonly used feed-forward neural network, able to distinguish linearly inseparable data. Each neuron in the network is a very simple element, which takes one or more inputs, applies a function, and produces an output. An example MLP is shown in Figure 2.1a, with a single "hidden" layer (a layer of neurons which is not exposed as an input or output). Any function may be used in a neuron, but the most common functions are linear, sigmoidal, or a simple threshold; in order to distinguish linearly inseparable data the neuronal activation function of one or more layers of neurons must be nonlinear [47].

As each of the neurons is independent of the others, it is possible for every neuron to have a different function—during training, each of the parameters to this function can then be gradually updated until the difference between the expected and actual output is minimised. More commonly, every neuron in a layer will use the same, fixed function [92], and instead the "weight" of a connection between two neurons is modified. The weight of a connection is simply a factor by which the output of a neuron is multiplied before it is taken as the input to another neuron, as shown in Figure 2.1b. The total input to a

neuron j, y_j, is thus:

$$y_j = \alpha_j + \sum_i w_{ij} x_i \tag{2.1}$$

$$x_j = f(y_j) = f(\alpha_j + \sum_i w_{ij} x_i) \tag{2.2}$$

where x_i is the output of a neuron i, w_{ij} is the weight of the connection from neuron i to neuron j, and α_j is a constant offset or bias. The output of neuron j, or x_j, is the result of applying the neuron's function to y_j, as in Equation 2.2.

It is common for literature which introduces neural networks to discuss pattern recognition, which may lead to a misapprehension that this is the only task to which neural networks are applied. Although they are particularly well suited to it, they are not limited to this application and have been used in various fields including control, database retrieval, and fault-tolerant computing [95].

The learning algorithm typically used with an MLP, error back-propagation, can limit the practical use of these networks. The training period commonly occurs offline, meaning the network is unavailable, and so this algorithm is not suitable for real-time applications [47]. Additionally it is possible for the back-propagation algorithm to become stuck in a *local minimum*, rather than finding the *global minimum* as is the aim [44, 70]. Finally, Tesauro and Janssens [102] showed an exponential relationship between the number of neurons in an MLP's input layer and the time required to train it. This demonstrated a scaling problem—although MLPs have been successfully applied to small problems in many areas, as problems grow larger they become computationally infeasible.

2.3 Associative Memories

In traditional computer memories, data is accessed using an *address*. This type of memory is also known as a *listing memory*. Palm [80] defines a listing memory M_l as storing a sequence of messages, where each message is a symbol s taken from an alphabet Σ:

$$M_l(\Sigma) = (s_1, \ \ldots, \ s_n) : n \in \mathbb{N}, \ s_1, \ \ldots, \ s_n \in \Sigma \tag{2.3}$$

If we know the arbitrary address at which data is stored, then we can retrieve it. In

order to combine two sequences of messages M and M', they are simply concatenated:

$$\begin{aligned} M &= (s_1,\ \ldots,\ s_n) \\ M' &= (s'_1,\ \ldots,\ s'_k) \\ M + M' &= (s_1,\ \ldots,\ s_n,\ s'_1,\ \ldots,\ s'_k) \end{aligned} \tag{2.4}$$

Associative memories, or mapping memories M_m, operate in a very different manner—essentially they provide a system able to answer questions. Formally they are defined:

$$M_m(P, A) = \{m : Q \to A,\ Q \subseteq P,\ |Q| < |\mathbb{N}|\} \tag{2.5}$$

where a message m is a mapping from a question Q to an answer A, and Q is a finite subset of the set of all possible questions P. In this type of memory, messages are combined as a set union—as long as the sets of questions are disjoint—with the answer to a question q being provided by the appropriate mapping:

$$\begin{aligned} M &= \{m : Q \to A\} \\ M' &= \{m' : Q' \to A'\} \\ (M + M')(q) &= \begin{cases} M(q) & \text{if } q \in Q \\ M'(q) & \text{if } q \in Q' \end{cases} \quad \text{where } Q \cap Q' = \emptyset \end{aligned} \tag{2.6}$$

Stated more simply, a listing memory sequentially stores elements selected from an alphabet and data is referenced directly by an address. An associative memory, on the other hand, creates mappings between inputs and outputs—or questions and answers—and data is referenced by the presentation of a known input causing the return of its associated output.

Although their properties are very different, Palm [80] showed that it is trivial to implement a mapping memory using a listing memory and vice versa. In order to store mapped data in a listing memory, we can simply create a list where:

$$s_1 = q_1,\ s_2 = a_1,\ \ldots,\ s_{2n-1} = q_n,\ s_{2n} = a_n \tag{2.7}$$

and to implement a listing sequence in an associative memory we use the mapping:

$$\{1 \to s_1, \ldots, n \to s_n\} \tag{2.8}$$

While using a conventional listing memory to implement an associative memory in this way is simple, it is also very slow and inefficient. In order to recall the output value associated with an input q_n, we must inspect the data stored at every address in turn until we find the input and its associated output. Various methods, such as hashing [63], may be used in order to improve upon this and reduce the number of memory accesses required. Because the input must still be compared to that stored at a particular location, however, a minimum of two memory accesses will always be needed—firstly to read the input stored at s_i and then to retrieve the output associated with this input from s_{i+1}.

A *content addressable memory* (CAM) is one that is designed to be fast for use with mapping memories, storing data by value rather than by reference [1]. The simplest implementation of this would be to use the binary value of an input as the address in which to store the associated output. For example if we wanted to associate an input '2001' with an output 'Kubrick', we would store 'Kubrick' in address 2001 or `0b11111010001`. It is clear that this method is not practical, however, as it is likely to lead to inefficient memory utilisation and relies on input data having a binary representation that is short enough to suitably use as an address. CAMs are therefore often implemented in specialised hardware which searches the entire memory in parallel for a given input, and returns a list of all outputs associated with it. Ternary CAMs provide a form of partial matching [79] by introducing a value "don't care" (`X`) in addition to the usual binary (`0` and `1`).

In general, however, both binary and ternary CAMs implemented using specialised hardware are expensive and have a low capacity. As such, we will now investigate alternative methods based on artificial neural networks.

2.3.1 Hopfield Networks

Recurrent networks differ from feed-forward neural networks in that they form a directed cyclic graph. They may contain feedback loops, which can be used to cause the network to have a *state*—essentially a form of storage [88]. The Hopfield network, originally proposed in 1982 [54], is one of the better known recurrent neural networks. Instead of having layers of neurons like an MLP, every neuron in a network is connected to all of the others—although some of the connections may have a weight of 0, which is equivalent to having no connection.

Figure 2.2 shows an example Hopfield network with four neurons, in which the output of every neuron is connected as an input to every other neuron. As the output of every

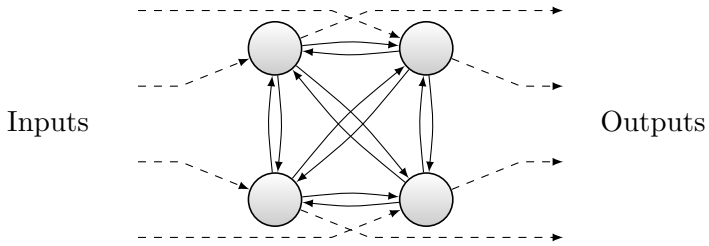


Figure 2.2: A Hopfield network with four neurons

neuron may affect the input of every other neuron, these connections create feedback loops that greatly increase the complexity of the network dynamics. The *energy* of a Hopfield network, E, is calculated using Equation 2.9:

$$E = -\frac{1}{2}\sum_{i,j} w_{ij} s_i s_j - \sum_i i_i s_i \tag{2.9}$$

where w_{ij} is the weight of the connection between neurons i and j, s_i is the current state of neuron i (a binary value), and i_i is the value on the input to neuron i. Binary values in this system can either be typical binary (`0`, `1`) or bipolar (`-1`, `1`). Typically Hopfield networks contain no self-connections, and all other connections are symmetric:

$$\forall i : w_{ii} = 0 \tag{2.10}$$

$$\forall i, j : w_{ij} = w_{ji} \tag{2.11}$$

Upon presentation of an input to the network, the activity of the neurons will gradually update until the energy function has been minimised [54], this is then the final and stable output state. In Hopfield's original work the update function is asynchronous—a single neuron is chosen at random, and its value is updated in order to minimise the energy of the network. Hopfield networks can be updated synchronously, where all neurons are updated simultaneously, however this is not biologically realistic as it requires a global clock [45].

One differentiating factor of these neural networks is that they are designed to continue to run indefinitely; the outputs of a Hopfield network are influenced by the network's current state. In a feed-forward neural network an output is generated in direct response to the presentation of an input, this means that changing the inputs will result in a series of corresponding outputs—as the feed-forward network has no state, repeat presentation of an input will always generate the same response. The Hopfield network, on the other hand, will always attempt to minimise its energy until it reaches a stable configuration.

Due to the nature of the update operation, it is not designed to process a series of inputs—with unstable inputs, the network may not reach a minimum before inputs are changed. Additionally, the minimum reached with a given input may differ from one presentation to the next as both the input and the current state of the neurons affect the next state. The network continuously provides an output throughout its operation, which can potentially give an estimation of a result as the optimisation is under way.

Training and operation of Hopfield networks are distinct, and so these networks are required to be taken offline in order to train new information—depending on the application this may be problematic. The training method is simple and fast, however, especially when compared to error back-propagation [89]. They have, therefore, been successfully used—particularly in pattern recognition (for example [75]), due to their ability to act as a hetero-associative or auto-associative memory through careful training of the minima of the network's energy function.

The capacity of a Hopfield associative network has been studied rigorously in [69], finding that under certain conditions the capacity is moderate. In contrast, however, Baum et al. find that using a single layer matrix memory and a local representation can result in a higher capacity [13]. One problem with Hopfield networks used in this fashion that does not seem to have been adequately resolved stems from the nature of the recall operation. For any given input, the network will continuously update until a minimum of the energy function has been reached. There is the potential for the network to reach a minimum that was not originally trained—although with careful training the probability of this can be reduced.

Perhaps more interesting is the assumption that for any given input, a stable output is desired. In alternatives, such as the Correlation Matrix Memory or Holographic Reduced Representation that we will see later, presentation of an untrained input should result in either an empty output, or an output with low confidence. Using a Hopfield network there is no way to distinguish between items that have or have not been trained, or to determine how much confidence should be assigned to the output.

2.3.2 Tensor Products

The concept of tensor products was introduced by Smolensky, described in detail in [96]. In this work, Smolensky discusses various vector representations: local, quasi-local, and distributed. In a local representation, only one input neuron is active at a time—each

input neuron stores a single concept, and each concept is stored by only a single input neuron. Local representations thus suffer from two limitations: they fail to efficiently scale, as the number of input neurons increases with the number of concepts to be stored, and the system may fail to operate if only a single neuron fails.

The quasi-local representation helps to resolve the second limitation of local representations, but at the expense of the first. In this case each neuron is used by only a single concept, but a number of neurons represent each concept. In this way, multiple neuron failures are required before the system will fail, however it multiplies the length of the input and so compounds the problem of scaling. Finally, in a distributed representation a concept is represented by the pattern of activity over a number of neurons. These patterns may overlap, meaning that up to 2^n patterns may be represented with only n binary neurons.

Smolensky [96] uses real-valued vectors and matrices, and presents two methods for producing the matrix of associations—simple Hebbian learning, and a modified version known as the "delta rule" which incorporates error-correction. Hebb [48] hypothesised that:

> When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.

In Hebbian learning the tensor product is found by calculating the outer product of two vectors—that is to say that the value, or activation, of a neuron is set to the product of its input and output activations. The superposition of two real-valued tensor products is then found by using the sum operation. Using the delta rule, the activation of a neuron is set to the product of its input activation and the difference between the output activation and an expected value provided by an external teacher.

In the case that the vectors used are orthogonal, the two training methods produce identical matrices [96]. Smolensky claims that the original Hebb's method is therefore somewhat limited—using the delta rule will result in a correct matrix for both local and distributed representations, where Hebb's method will only work correctly in local representations (or a distributed representation where all the vectors have been carefully selected to be orthogonal). Smolensky does not specify what is considered to be an "incorrect" matrix, however it is a reasonable assumption that in this context such a matrix

would produce the wrong result when a vector is recalled from it.

A particular problem with the use of tensor products in cognition is that the number of matrix dimensions increases for every level of additional binding required, quickly causing the tensor product to become too large to be feasible—biologically or otherwise—for all but the smallest of problems [107]. It is argued (e.g. [43,99]) that this limitation can be avoided through the use of reduced representations [49], such as convolution [84] (discussed further in Section 2.3.6), elementwise multiplication [41, 60], or permutation-based thinning [87].

Jackendoff [58] presented four challenges for cognitive neuroscience, positing that the connectionist approaches at the time—such as tensor products—failed to resolve the problems. Gayler [42] presented responses to each of these challenges, demonstrating that Vector Symbolic Architectures—a development of Smolensky's tensor products—are able to meet the challenges successfully. Further than this, Gayler emphasises that tensor product networks are able to cope with *The Problem of Variables*. The productivity of language leads to the difficulty that humans can generate and understand an infinite number of sentences using a finite neural network. The use of variables allows for this productivity, assuming each variable may contain an arbitrary value [58]—allowing for complex structures such as recursion.

In summary, although tensor products are relatively simple, they can be used to solve certain sophisticated problems. Networks which make use of tensor products in more complex architectures, or those which have developed from—and enhanced—tensor products, have been shown to be capable of solving a range of complex challenges (e.g. [19, 20, 42]).

2.3.3 Correlation Matrix Memories

Correlation Matrix Memories (CMMs) and tensor products are very similar, both storing the associations between pairs of vectors using the outer product. One distinction is their intended purpose—CMMs are designed to be used as associative memories; tensor products are intended to provide a distributed representation of a variable/value pair [97].

CMMs are thus a special type of feed-forward neural network that store associations between pairs of vectors in a number of neurons, often represented as a matrix [64, 112]. They are a single layer, fully connected network, as every input neuron is directly connected to every output neuron. Although CMMs may be "real-valued"—where the matrix of connection weights may contain any real number—we are interested in a sub-class of CMMs known as binary CMMs.

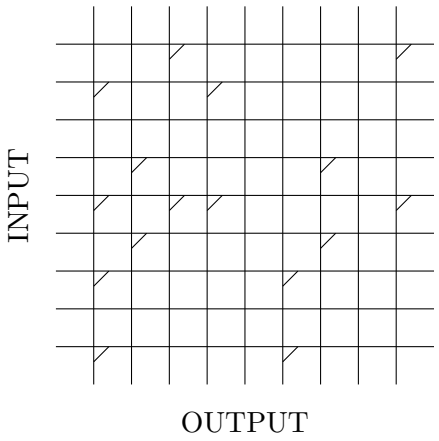


Figure 2.3: A binary correlation matrix memory

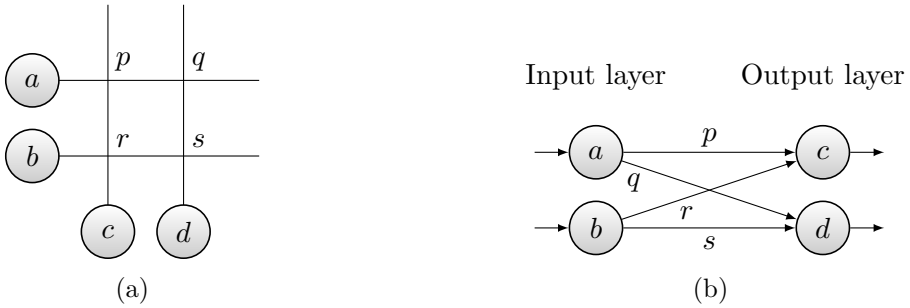


Figure 2.4: A small binary CMM, represented as (a) a matrix and (b) a feed-forward neural network. The neurons are labelled *a*–*d*, and the connections *p*–*s*. Neurons are generally not explicitly presented in matrix form, rather they are implicitly understood to be present.

Binary CMMs are a form of hetero-associative memory, or content-addressable memory [13]—information is recalled based on an association with input information, rather than the knowledge of a particular storage location [74]. Depending on the content trained, they can also act as an auto-associative memory—where the presentation of partial information causes the recall of the complete information, for example coping with occlusion when performing pattern recognition [110].

Figure 2.3 shows an example binary CMM. Each horizontal line, or wire, represents an input neuron, and each vertical wire an output neuron. This representation shows the clear relationship between a CMM and its matrix—each intersection between wires is a cell in the matrix of weights. In this binary CMM, a weight of 1 is marked, and a weight of 0 is unmarked.

To aid comparison with other neural networks Figure 2.4 shows a very small (2×2) binary CMM, firstly as a matrix and then as a feed-forward neural network. In these figures, the neurons are labelled *a*–*d* and the connections are labelled *p*–*s*. All connections

between neurons have a weight of either 1 or 0, distributing an input or blocking the signal respectively. The output neurons act in the manner common to perceptron networks, by summing their inputs before applying a threshold.

A significant benefit of CMMs compared to many other neural networks is that they may be trained while online, using high-speed, Hebbian learning. As mentioned earlier, associations are formed using the outer product of input and output vectors; in a binary CMM, tensor products can simply be superimposed onto an existing matrix using a logical OR operation. The equation for training is formalised in Equation 2.12:

$$\mathbf{M} = \bigvee_{i=1}^{n} \mathbf{x}_i {\mathbf{y}_i}^T \tag{2.12}$$

where $\mathbf{M}$ is the resulting matrix of binary weights (the CMM), $\mathbf{x}$ is the set of input vectors, $\mathbf{y}$ is the set of output vectors, n is the number of training pairs, and $\bigvee$ indicates the logical OR of binary matrices.

Considering Smolensky's claim that Hebbian learning is not suitable for use with distributed vectors, except if they are orthogonal, binary CMMs might be considered to be somewhat limited when compared to real valued matrices. The delta rule cannot be adapted for binary networks, as the activations of all neurons can only be set to either one or zero—adjusting an activation to minimise error is simply not possible. In Willshaw's original work [112], however, it was demonstrated that using an appropriate threshold method and value can overcome this apparent limitation and allow binary CMMs to effectively store a number of associations even with non-orthogonal vectors.

The recall process is similarly rapid, as shown in Equation 2.13. A non-binary vector is first calculated by performing a matrix multiplication between the transposed input vector $\mathbf{x}^T$ and the CMM $\mathbf{M}$. A threshold function f must then be applied to this vector, in order to produce the final binary output vector.

$$\mathbf{y} = f(\mathbf{x}^T \mathbf{M}) \tag{2.13}$$

This recall operation may be greatly optimised, using the fact that the input vector contains only binary components. For the j^{th} bit in the output, the result of a matrix multiplication is the vector dot product of the transposed vector $\mathbf{x}^T$ and the j^{th} column

Token	Binary vector^T	Token	Binary vector^T	Token	Binary vector^T
a	000100001000	**e**	000010000100	**i**	010000100000
b	000100100000	**f**	100000000010	**j**	000010010000
c	001000010000	**g**	010000000001	**k**	100000001000
d	100001000000	**h**	001001000000	**l**	100000100000

Table 2.1: An example set of tokens, with a fixed-weight binary vector allocated to each token. These binary vectors were generated using Baum's algorithm [13] with a length of 12 and weight of 2 (with partition lengths 5 and 7), and assigned to the tokens randomly. The column vectors are shown here transposed for practical reasons.

of matrix $\mathbf{M}$, represented as $\mathbf{M}[:,j]$:

$$\mathbf{y}_j = \mathbf{x}^T \cdot \mathbf{M}[:, j] \tag{2.14}$$

The vector dot product is defined in Equation 2.15, where l is the input vector length and $\mathbf{M}_{i,j}$ is the value stored in the j^{th} column of the i^{th} row of matrix $\mathbf{M}$.

$$\mathbf{y}_j = \sum_{i=1}^{l} \mathbf{x}_i \mathbf{M}_{i,j} \tag{2.15}$$

Given the binary nature of $\mathbf{x}$, it is clear that this dot product is equal to the sum of all values $\mathbf{M}_{i,j}$ where $\mathbf{x}_i = 1$. The complete recall operation (without application of a threshold) is thus formalised in Equation 2.16, where $\mathbf{M}[i,:]$ represents the i^{th} row of matrix $\mathbf{M}$:

$$\mathbf{y} = \sum_{i=1}^{l} \begin{cases} \mathbf{M}[i,:] & \text{if } \mathbf{x}_i = 1 \\ 0 & \text{if } \mathbf{x}_i = 0 \end{cases} \tag{2.16}$$

2.3.3.1 Tensor Product Representation

To describe the contents of a CMM or tensor product, without resorting to the use of binary matrices, we use a pictorial representation. Using the example tokens given in Table 2.1, the tensor product $\mathbf{a} : \mathbf{b}$[1] is shown in Figure 2.5a. Below this matrix, in Figure 2.5d, is our higher-level representation of a tensor product—each displayed column is labelled at the top with the input token it contains, and at the bottom with the output token to which the input was bound.

Figures 2.5b and 2.5e show the result of superimposing this first tensor product with

[1] $\mathbf{a} : \mathbf{b}$ indicates that vector $\mathbf{a}$ is *bound* to vector $\mathbf{b}$, through the use of the outer product operation. This may also be represented as $\mathbf{a} \otimes \mathbf{b}$, however here we use ":" for practical reasons that we shall encounter in Chapter 3.

e : **i**. As there is overlap between the output vectors **b** and **i**, the seventh column contains both vectors **a** and **e** superimposed. Finally, Figures 2.5c and 2.5f show a CMM trained with all of the associations in Figure 2.6. Although the matrix contains a large number of trained vector pairs, in the pictorial representation it is still possible to quickly determine which columns contain a particular vector. Columns which are not displayed in a tensor product are assumed to be filled entirely with zeros; it is possible that they contain extraneous noise, however this is irrelevant for the purposes of the diagram.

2.3.4 Capacity of Correlation Matrix Memories

In [13], Baum et al. compared the capacity of an associative memory created using a Hopfield network to the capacity of a binary CMM using a local representation. Quoting [69], they state that in a Hopfield network with an input of length N, $N/4 \ln N$ pairs of vectors may be associated while still allowing for correct recall.

In comparison, using a local representation stored in a binary CMM provides a capacity that "is at least comparable to that of the Hopfield model" [13]. The lack of a definitive capacity is due to the lack of fault-tolerance provided by a local representation—as only a single neuron is used to represent each individual concept, if it were to fail then that concept would be lost. In order to make the network more robust, the use of a quasi-local representation is proposed—as long as every copy of a neuron does not fail, no content is lost and the memory can be repaired.

Austin and Stonham [11] calculated a conservative estimate for the probability of recall failure of a particular binary CMM trained with a number of associations to be:

$$P = 1 - \left\{ 1 - \left[1 - \left(1 - \frac{NI}{HR} \right)^{T} \right]^{I} \right\}^{H} \tag{2.17}$$

where P is the probability of a recall failure (the recalled vector differs in some way from the expected result), R and I are the length and weight of the input vector, H and N are the length and weight of the output vector, and T is the number of pairs of vectors stored in the matrix. Equation 2.17 says that as the "capacity" of an associative memory is reached, when further associations are stored in the matrix, the probability of accurately recalling any given vector will gradually decrease.

Turner and Austin [104] developed this further, creating a probabilistic framework to estimate the performance of binary CMMs. This framework allows for networks of

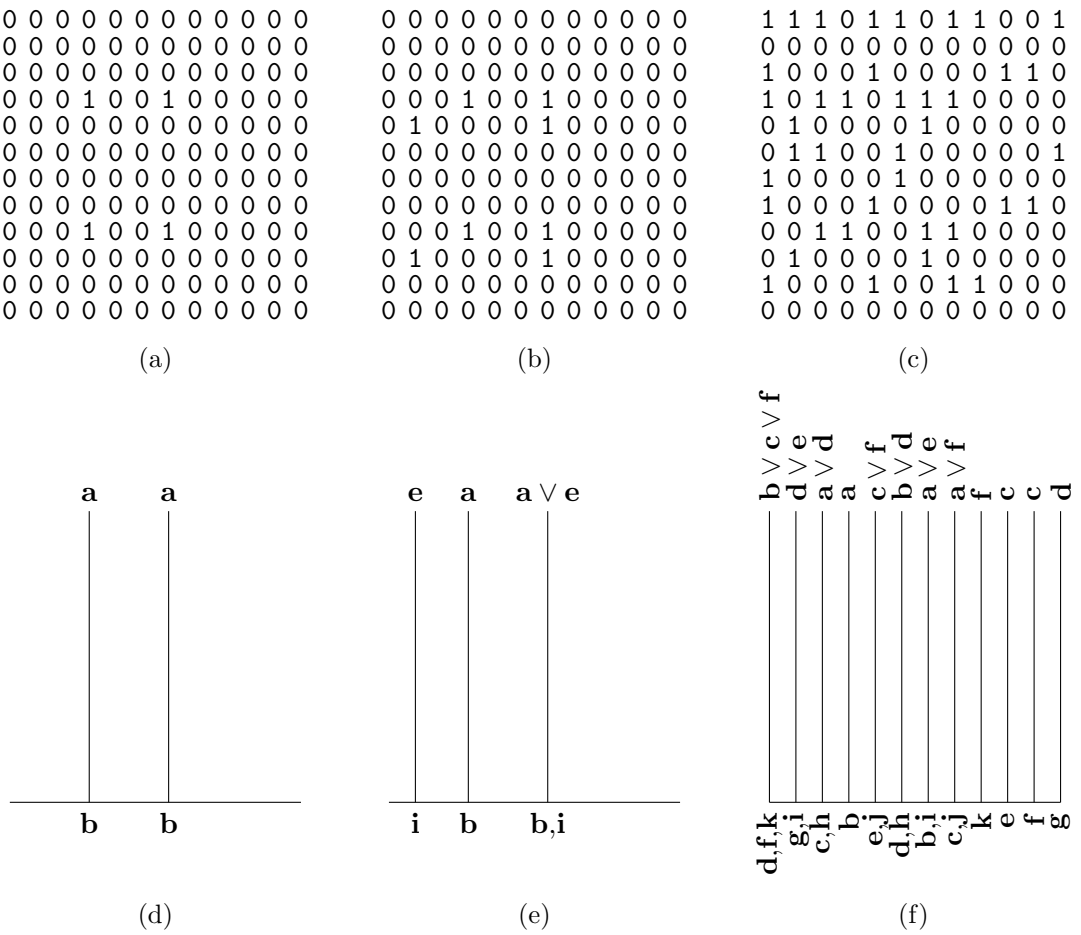


Figure 2.5: Three tensor products, or CMMs, given first in matrix form (above) and then in a pictorial representation (below). (a) and (d) show the tensor product formed between **a** and **b**; formally this tensor product is represented as **a** : **b**. (b) and (e) show the superposition of this tensor product with that formed between **e** and **i**; formally these figures show (**a** : **b**) ∨ (**e** : **i**). Finally, (c) and (f) show the CMM formed by training all of the rules found in Figure 2.6 using the method given in Equation 2.12. In all of these diagrams, the presence of a particular input vector within a column is indicated by the label at the top of that column. The output vector with which this input was bound is labelled at the bottom of each column.

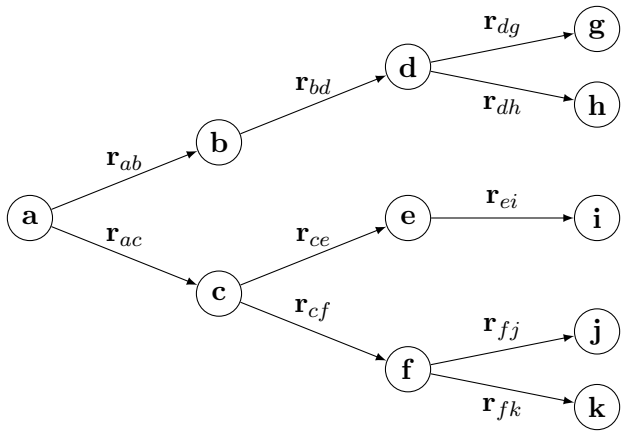


Figure 2.6: A set of associations represented as a tree, where the nodes represent vector tokens and the edges represent associations between them. In this tree the branching factor is 2, meaning that each node has at least one and at most two children.

any depth, as well as allowing for matching of partially presented inputs. A limitation of all capacity estimations to date, however, is that they calculate the performance of binary CMMs storing randomly-generated fixed-weight vectors. As we shall see in the next sections, the capacity of a CMM is affected by the choice of threshold function and increased significantly by increasing the orthogonality of vectors.

2.3.5 Threshold Functions

There are a number of functions which may be used as the threshold during a recall from a binary CMM, although the choice of function may be limited by the application and the data representation used. In a typical feed-forward neural network, the threshold can be applied locally and independently in each individual output neuron. In a CMM there may instead be a global function, applied to the values across all of the output neurons.

2.3.5.1 Willshaw's Threshold

One of the simplest threshold functions is *Willshaw's threshold* [112]. When training the CMM, all input vectors are required to have the same fixed weight—the number of bits set to `1`. During a recall, this fixed input weight is used as a simple threshold: any output neuron with a value greater than or equal to this weight is set to `1`, and neurons with values below the limit are set to `0`. When using this threshold with a complete input, the CMM is guaranteed to successfully recall all of the expected output bits. If a CMM becomes saturated (with too many vector pairs associated in it) there may be additional *ghosts* in the output. These are outputs bits that are incorrectly set due to unwanted

interactions and interference between similar patterns within the matrix.

Willshaw's threshold allows for partial matching; relaxation is achieved by simply reducing the threshold. For example if the fixed input weight is 4, then using a value of 3 as the threshold will allow patterns missing a single input bit to be recognised. Similarly, reducing the threshold further will allow recall with fewer correctly set input bits. Reducing the threshold value can, however, have the negative side effect of increasing the number of ghosts recalled—particularly if the matrix is approaching saturation. Without employing relaxation the method will fail to recall if the presented input differs in any way from that trained, meaning that it is susceptible to errors if recall inputs may be noisy [11].

Finally, when using Willshaw's method of thresholding, binary CMMs have the unusual property of continuing to operate correctly when input vectors are superimposed [5]. The resultant output vector then contains the superposition of the expected output vectors. To illustrate this, consider the example set of vector tokens given in Table 2.1, and rules in Figure 2.6. Figure 2.5c shows the matrix that results if these rules are trained into a CMM. A recall of the vector **b** (or `000100100000`) results in an output of `201102110000`. After applying Willshaw's threshold with a value of 2 (the weight of an input vector), we recover the vector `100001000000`—the expected vector **d**. Similarly, recalling **e** (or `000010000100`) results in an output of `020000200000`, or `010000100000` after a threshold—again this is the expected vector **i**.

The superposition of these input vectors, $\mathbf{b} \vee \mathbf{e}$, is `000110100100`. The result of recalling this from the CMM is `221102310000`; it can be clearly seen that after applying the threshold, still using a value of 2, the vector `110001100000` contains the superposition of both of the expected outputs, $\mathbf{d} \vee \mathbf{i}$. Due to overlap between the vectors, however, all of the set bits that form vector **l** are present in the superimposed output. When using a distributed encoding, it is often not possible to determine which individual vectors are actually present in a superimposed vector, and which simply appear to be present due to overlap with other vectors.

2.3.5.2 L-max Threshold

An alternative threshold function is known as the *L-max threshold* [6]. In this case, when training the CMM, all output vectors are required to have the same fixed weight. The threshold is applied as a global function across the output neurons, setting the highest l

values to `1` and the rest to `0`—where l is the fixed weight of an output vector.

Austin and Stonham [11] showed that compared to Willshaw's threshold this method provides improved recall performance, and hence capacity, when presenting noisy input vectors. It similarly provides an implicit ability for relaxation—by setting the highest values to `1`, the output values do not necessarily need to be as high as the weight of an input vector and so not all of the input bits need to be correctly set.

L-max suffers from one potential limitation—that an input may be associated with only a single output—due to the requirement that l must be known. If an input vector is individually associated with two output vectors, then the number of set bits in the expected output cannot be determined. Similarly, input vectors cannot be superimposed prior to a recall, as l is unknown.

2.3.5.3 L-wta Threshold

The final threshold function requires that vectors are generated using a specific algorithm, known as *Baum's algorithm* [13]. Baum et al. recognised that if the orthogonality between vectors can be maximised, then the interference between pairs of stored vectors should be minimised. To generate vectors of length n, and weight s (where the weight of a vector is the number of ones it contains), their algorithm is as follows.

Choose s co-prime numbers such that they sum to n and each number is greater than the last:

$$p_1 < p_2 < \cdots < p_s \ , \ n = \sum_{i=1}^{s} p_i \ , \ \forall ij, i \neq j \Rightarrow p_i \perp p_j$$

Every generated vector then contains a single "one" in the first p_1 bits, another in the next p_2 bits, etc. As an example, with $n = 12$, $s = 3$, $p_1 = 3$, $p_2 = 4$, and $p_3 = 5$, the vectors—or codes—would be generated as shown in Table 2.2. Hobson [52] notes that the requirement that $p_1 < p_2 < \cdots$ is not strictly necessary, it is sufficient that the numbers are all co-prime. Including the requirement in the algorithm does not lose any generality, although it does not provide any benefit either.

Using the requirement, if only $p_1 \times p_2$ codes are generated, the minimum Hamming distance between them will be $2s - 2$, as there will be at most a single `1` overlapping between any two codes. Similarly if $p_1 \times p_2 \times p_3$ codes are generated, there will be at most two `1`s overlapping between two codes, giving a minimum Hamming distance of $2s - 4$ [13].

Without the monotonically increasing ordering, the same property holds—amending the calculation to use the smallest p_i-values, rather than assuming these are the first

1.	1	0	0	1	0	0	0	1	0	0	0	0
2.	0	1	0	0	1	0	0	0	1	0	0	0
3.	0	0	1	0	0	1	0	0	0	1	0	0
4.	1	0	0	0	0	0	1	0	0	0	1	0
5.	0	1	0	1	0	0	0	0	0	0	0	1
6.	0	0	1	0	1	0	0	1	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 2.2: Baum codes generated with $n = 12$, $s = 3$, $p_1 = 3$, $p_2 = 4$, $p_3 = 5$

values. In general, if up to $\prod_{i=1}^{t} p_i$ *Baum codes* are generated, for the smallest values of p_i, the minimum Hamming distance between any two codes will be $2(s - t + 1)$.

Baum et al. briefly mention the ability to exploit the known structure of the codes, using a "winner takes all" approach to the threshold. This is developed by Hobson [52] and termed the *L-wta threshold*. This method of thresholding is essentially an extension of the L-max threshold—for each section of the recall result with length p_i, L-max is applied with an l-value of 1. Thus, in each of the s sections the highest value is mapped to a one, and everything else maps to zero.

Casasent and Telfer had previously experimented with the capacity of CMMs and determined that, of those threshold methods they tried, using fixed-weight vectors and the L-max threshold method allowed a CMM to have the highest capacity [23]. As such, Hobson compared the storage capacity of a CMM when using L-max and L-wta. His experimental results showed that when using Baum codes, the capacity achieved in a CMM with the L-wta threshold is always the same or greater than that achieved with the L-max threshold [52].

As the L-wta threshold is very similar to L-max, it suffers the same limitation: requiring that l is known. For each of the s sections, the expected output is a single one, and so output vectors cannot be superimposed—neither due to the superposition of input vectors, nor association of a single input vector with multiple output vectors.

2.3.6 Alternatives to Tensor Products

One of the criticisms of tensor products is the potentially large amount of memory required to store vector associations. Storing associations between two vectors of lengths l and m requires a matrix of size $l \times m$. If a third vector is to be associated, of length n, then this increases the requirement to a size of $l \times m \times n$. Although there are effective methods

to store sparse matrices using a compressed representation, it is clear that the storage requirements can increase rapidly for compositional structures, or for non-sparse codes.

Plate [84] proposes an alternative method of storing associated vectors. Rather than simply using an outer product operation, associations are constructed using circular convolution. The result of this operation is effectively a compressed form of the outer product, a vector of the same length as those originally associated, known as the Holographic Reduced Representation (HRR).

The general equation for circular convolution ($\circledast$) is given in Equation 2.18 and shown in Figure 2.7. The figure is an illustration of convolution applied to a pair of 3-bit vectors. The 3×3 matrix is the outer product of vectors $\mathbf{x}$ and $\mathbf{y}$, and vector $\mathbf{z}$ is the result of compressing—or convolving—this matrix [85].

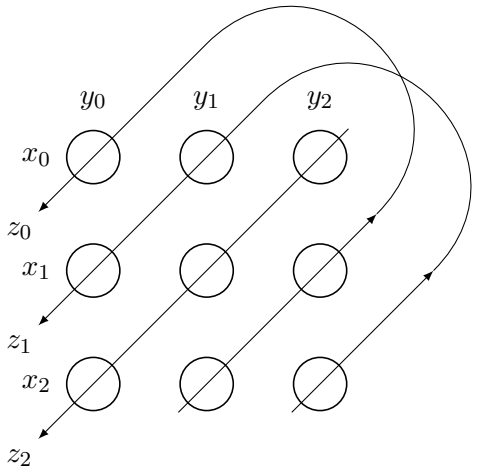
$$\begin{aligned} \mathbf{z} &= \mathbf{x} \circledast \mathbf{y} \\ \mathbf{z}_i &= \sum_{j=0}^{n-1} \mathbf{x}_j \mathbf{y}_{(i-j) \bmod n} \end{aligned} \tag{2.18}$$

Under certain conditions, given an HRR storing only a single associated pair of vectors, circular correlation ($\circledcirc$) will work as the approximate inverse operation—performing a recall operation to recover an associated vector, and giving a somewhat noisy result [84]. Equation 2.19 gives the general equation for this operation, and Figure 2.8 demonstrates the process of reversing the previous circular convolution.

$$\begin{aligned} \mathbf{y} &\approx \mathbf{z} \circledcirc \mathbf{x} \\ \mathbf{y}_i &\approx \sum_{j=0}^{n-1} \mathbf{z}_{(i+j) \bmod n} \mathbf{x}_i \end{aligned} \tag{2.19}$$

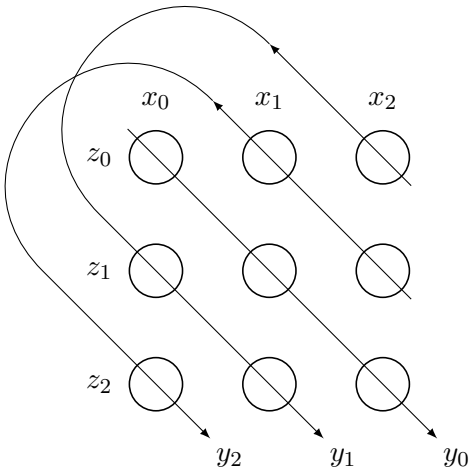
In order to compose HRRs, Plate suggests simply adding them together. Having done so, the recall operation will only work effectively to determine the presence of an associated vector within the HRR, rather than to obtain the associated vector, due to the additional noise that will be output. For a system using HRRs to be able to perform a full recall, and actually retrieve the associated vectors, Plate proposed an additional stage after the circular correlation—passing the retrieved vector through an auto-associative memory to correct any errors.

Plate claims that "the exact method of implementation of the [auto-associative] memory is unimportant" [84]. Using a matrix memory as this auto-associative memory, how-



$$\mathbf{z} = \mathbf{x} \circledast \mathbf{y}$$
$$z_0 = x_0y_0 + x_1y_2 + x_2y_1$$
$$z_1 = x_0y_1 + x_1y_0 + x_2y_2$$
$$z_2 = x_0y_2 + x_1y_1 + x_2y_0$$

Figure 2.7: Circular convolution with vectors of length 3 ($n = 3$) [85]. Each of the circles is an element of the outer product of $\mathbf{x}$ and $\mathbf{y}$. Each of the elements of the final circular convolution are sums along the diagonal lines.



$$\mathbf{y} \approx \mathbf{z} \circledcirc \mathbf{x}$$
$$y_0 \approx z_0x_0 + z_1x_1 + z_2x_2$$
$$y_1 \approx z_0x_2 + z_1x_0 + z_2x_1$$
$$y_2 \approx z_0x_1 + z_1x_2 + z_2x_0$$

Figure 2.8: Circular correlation with vectors of length 3 ($n = 3$) [85]. Each of the circles is an element of the outer product of $\mathbf{z}$ and $\mathbf{x}$. Each of the elements of the final circular correlation are sums along the diagonal lines.

ever, could counteract any memory-saving benefit of using HRRs. Therefore, the choice of error-correcting memory may be important, as it must be able to effectively perform error-correction on the output of an HRR, while still requiring less memory than an equivalent distributed vector recall system built entirely using a matrix memory.

HRRs are of questionable utility in architectures based on binary vectors; although a binary HRR could be created, the convolution process reduces the fidelity of even real-valued associations to a degree that requires an error-correcting memory to recover from. As well as this, the composition of HRRs is performed by summing—resulting in a non-binary output. To use a logical OR operation instead will further decrease the accuracy of a recall.

2.4 Distributed Computation

Using fMRI it has been shown that different areas of the ventral stream are used when recognising different categories of object [57]. From the results obtained, it was concluded that rather than the brain containing a separate area for each category, it seems that the representations of objects are in fact distributed. The mechanism or representation used for the distribution was not known, however it is noted that the representation appears to be organised in a way that reflects the differences between categories. It is proposed that the distinguishing factors between representations of different object categories may be similar to those proposed by Tanaka [101], including changes in luminosity in particular areas of an image, or specific sub-shapes such as circles and squares.

On the other hand, it has been found in [83] that stimuli from each of a rat's whiskers is processed by a small but separate module, or group of neurons, in the rat's brain. They also concluded that the timing of spikes (an electrical pulse from one neuron to another) plays a large part in the encoding of information. Previously it had been assumed that the timing of spikes was not sufficiently accurate, and that only the number of spikes in a given time could be used to pass information [93]. This difference of findings seems to indicate that it is possible that some functions in the brain are performed by exclusive modules, where others—particularly higher-order functions—are performed in parallel using a distributed representation.

As discussed in Section 2.3.2, local representations are a straightforward way to store content in a number of nodes, where each individual node stores one piece of information. A local representation is ideal for parallel processing, as the content of each node

is independent of the other nodes and therefore each node may be processed separately by dedicated hardware or software [50]. While this may be easy to implement, it is more susceptible to errors from noisy inputs, and each module must be implemented specifically for a node.

Some systems built with neural networks use a distributed vector representation instead [50]. Using a distributed representation, content is stored as a pattern of activity across a number of nodes rather than in the activation of a single node. This gives various benefits, such as the ability to generalise and gracefully degrade with a partial or noisy input, and the use of a number of simple homogeneous nodes that can be processed in parallel [84].

2.4.1 Tensor Product Production System

Productions are simple rules that may be used in various fields, such as artificial intelligence and expert systems. They contain antecedents and consequents—the consequents "fire" if all the antecedents of a production are met.

In their most basic form, a production system may be thought of as being similar to a state machine, with the antecedent of a rule being the current state, and the consequent of a rule becoming the new state. In this form productions are limited to having only a single antecedent, referred to as arity-1 rules. Productions with two antecedents are arity-2 rules, etc.

Dolan and Smolensky developed a simple production system in 1989, using tensor products [29]. The system required third-order tensors (or three-dimensional matrices), and successfully operated on a given set of productions. The experiment was limited to a set of six productions, and six tokens (out of a possible 35), represented by seven-bit vectors. Despite their very limited size, the results were encouraging—particularly those involving the injection of faults, as the system was able to continue operating successfully for a reasonable number of injected faults.

2.4.2 Parallel Distributed Computation

Parallel Distributed Processing (PDP) was a concept presented by the PDP Research Group at the University of California, San Diego [90]. Essentially this work expands upon the use of distributed vector representations, developing and expounding upon a number of models suited to varying uses.

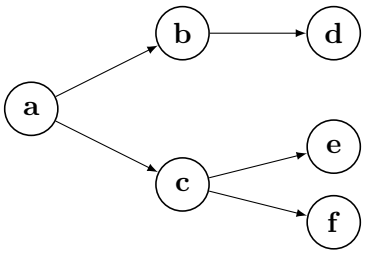


Figure 2.9: Example non-deterministic finite state automaton

Parallel Distributed Computation (PDC) is a similar idea presented by Austin [5], specifically applying to CMMs. PDC has an important feature that is not shared by PDP—the ability to process more than one computation simultaneously, using only a single neural network.

An alternative way to consider a CMM, rather than as an associative memory, is as a state machine [5]. This extends the use of associations as a form of storage to allow them to be used for computation. Operation of the network is exactly the same—namely that an input vector is presented, and the associated vector is output—but considering it in this way can encourage inspiration for different applications.

Developing Dolan and Smolensky's production system further, Austin [6] proposed a two-layer CMM to implement productions. One of the particular aims of this work was to develop a neural network-based production system that could be practically implemented on available systems. Kustrin and Austin [66] extended the notion, creating a CMM-based system capable of performing connectionist propositional logic, and demonstrating its ability to resolve queries using the trained axioms. This work was limited, however, in that it provided no way for a production to have more than one consequent. If a rule like this were to exist, then distinguishing individual vectors in a superimposed output would have been a problem.

As well as a potential for further application, some of the capabilities of CMMs become more apparent. When considering a particular application of CMMs, for instance performing pattern recognition, the benefits of parallel operation seem obvious—if two images are processed at once, then the overall execution time will be halved compared to only processing one at a time. Similarly, in a classical system designed to process a finite state automaton, parallel operation will reduce the processing time—but sequential operation is sufficient. When considering a CMM to be a state machine, however, the ability to operate correctly on multiple states simultaneously becomes not only desirable, but essential if the limitation shown in Kustrin and Austin's work is to be avoided.

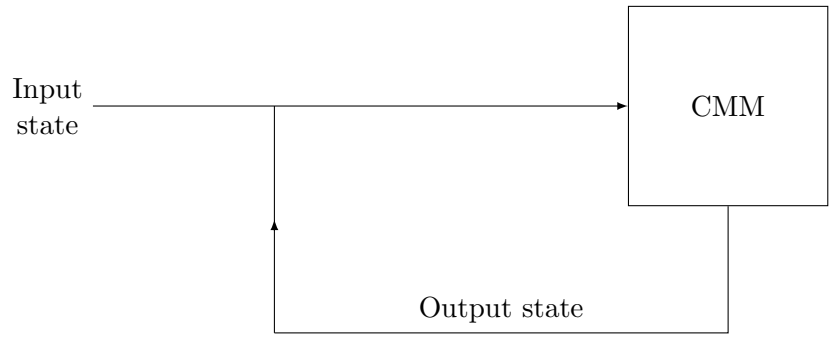


Figure 2.10: A CMM as a state machine

a	1	0	1	0	0
b	0	1	0	1	0
c	1	0	0	0	1
d	0	1	1	0	0
e	1	0	0	1	0
f	0	1	0	0	1

(a)

$\mathbf{a} \vee \mathbf{b} = \texttt{11110}$

$\mathbf{a} \vee \mathbf{c} = \texttt{10101}$

$\mathbf{b} \vee \mathbf{c} = \texttt{11011}$

$\mathbf{e} \vee \mathbf{f} = \texttt{11011}$

(b)

Figure 2.11: (a) Baum codes generated with $n = 5$, $s = 2$, $p_1 = 2$, $p_2 = 3$, and (b) demonstrating a potential issue distinguishing overlapping vectors.

As an example, consider the states and transitions in Figure 2.9. A system may traverse the state space using a number of algorithms, for instance breadth-first or depth-first search. The state transitions can be trained into a CMM, representing each state token by a distributed vector and then associating the appropriate vectors. Operation of the state machine is iterative, with the output of a recall operation becoming the input to the next iteration, as shown in Figure 2.10.

Upon presentation of the input vector **a**, in an associative memory that converges to a single output state—such as the Hopfield network [54]—only a single output state would be selected, either **b** or **c**. As we saw in Section 2.3.5.1, however, both states may be recalled simultaneously when using a CMM. Due to the nature of matrix memories all of the output neurons involved with mapping **a** to **b**, as well as **a** to **c**, would fire—after applying an appropriate threshold the output would then be the superposition of the two states **b** and **c**.

Both the L-max and L-wta thresholds require knowing the exact weight of an output vector, with the L-wta method also requiring knowledge about the distribution of bits set to one. If multiple vectors are superimposed then neither the weight nor the bit distribution can be guaranteed, as vectors may overlap (as can be seen in Figure 2.11b

with $\mathbf{a} \vee \mathbf{c}$). This leaves the original Willshaw threshold as the only appropriate option, in this case using a threshold value equal to the weight of a single input vector. The problem now faced is to be able to distinguish between the superimposed vectors that are output in such a situation. Using the tokens in Figure 2.11a, the superposition of vectors **b** and **c** is `11011`. The superposition of vectors **e** and **f**, however, is exactly the same. Although this is a contrived example, it serves as a demonstration of the potential difficulty. A solution to this problem will be described in Section 4.3.

2.5 Architectures for Pattern Recognition

A large number of architectures have been developed to perform pattern recognition using neural networks, often with greatly different motivations and intended applications. This section reviews the salient features of the Neocognitron and an architecture developed using the Multiple Interacting Instantiations of Neural Dynamics (MIIND) framework, two architectures that remain actively developed and used. I will then discuss syntactic pattern recognition and cellular automata before finally introducing the Cellular Associative Neural Network (CANN), which is investigated in greater detail in Chapter 5.

2.5.1 Neocognitron

The Neocognitron is a hierarchical, multi-layered neural network proposed by Fukushima in 1980 [34]. The architecture was originally inspired by structures found in the visual cortex of the brain by Hubel and Wiesel in 1965 [55], using alternating layers of S-cells and C-cells (simple and complex cells). The S-cells perform feature extraction, such as detecting edges, or recognising small parts of an image. The C-cells then combine the output of a number of S-cells, firing if at least one of its inputs also fires, in order that a distorted or shifted image will still be recognisable.

Figure 2.12 shows an example of the hierarchical architecture. Each layer (U_x) contains multiple sub-layers, or "cell-planes", each of which are trained to respond to different features in an input image.

Continued research has focused on improvements to pattern recognition quality and robustness [37], as well as the development of further specific applications such as handwritten digit recognition [35] or the restoration of partly occluded patterns [36]. The motivation behind the Neocognitron is clearly to develop an effective pattern recognition

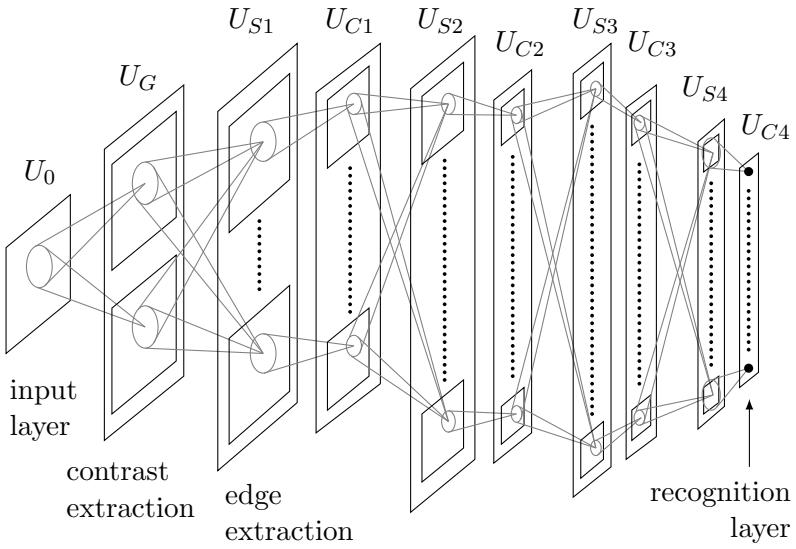


Figure 2.12: A typical architecture of the Neocognitron [38]

architecture with a biological basis. An investigation of the Neocognitron by Shouno [94] found that the architecture is reasonably effective at pattern recognition, even when presented with moderately scrambled input.

A possible limitation of the Neocognitron, identified by Gupta and Singh [46], is its complexity. In their implementation the Neocognitron had 10 layers (U_0–U_{C4} as shown in Figure 2.12), containing over 20,000 cells and over 3,000,000 connections between them. Configuration and training of this network is therefore a slow process, although more recently new training methods have been developed to improve upon this [39, 40].

2.5.2 Multiple Interacting Instantiations of Neural Dynamics

Multiple Interacting Instantiations of Neural Dynamics (MIIND) [26] is an open source framework designed to aid the implementation and analysis of cognitive models. As such, MIIND is not an architecture on its own, and is intended to fulfil a very different purpose. MIIND has been used for a wide range of modelling, however, including the simple pattern recognition architecture to be described here [106].

Mishkin and Ungerleider [71] experimentally distinguished two areas of the brain particularly involved in vision—the ventral stream identifies what an object is, and the dorsal stream identifies where it is. The MIIND-implemented architecture models these streams, connecting the artificial neurons in such a way as to be biologically plausible—for instance creating specific inhibition circuits, rather than allowing negative weights on synapses.

Often, in pattern recognition, the task is to detect the presence of a pattern within an

image. This architecture differs from this, in that it not only detects the presence of the pattern, but also where in the image that pattern was found. The architecture is composed of layers corresponding to the primary visual cortex (V1) as well as the V2 and V4 areas of the ventral stream, and finally parts of the inferior temporal lobe—the PIT and AIT. In addition to feed-forward connections between the various layers, feedback connections are used to carry information about a target object to be located.

This architecture may seem very limited against alternatives such as the Neocognitron or classical pattern recognition algorithms. Comparing this to other pattern recognition systems would be unfair, however, as the architecture is actually a neural model of visual object-based attention and not intended as a general purpose pattern recognition tool.

2.5.3 Cellular Associative Neural Network

Before the Cellular Associative Neural Network (CANN) can be discussed, two topics—syntactic pattern recognition and cellular automata—must first be introduced.

2.5.3.1 Syntactic Pattern Recognition

Within pattern recognition, classification algorithms can be loosely categorised into two general approaches: statistical and syntactic [103]. The first approach extracts statistical information about features in the data in order to perform recognition, and includes methods such as Linear or Quadratic Discriminant Analysis [14]. Model-based classification (prototype matching) also uses statistical methods, however it is sometimes considered as a distinct approach [33]. An example is the k-Nearest Neighbour algorithm [25], which attempts to perform recognition by finding the closest match to an input pattern in a database of known patterns. Finally, syntactic (or structural) methods rely on the structure present within patterns in order to perform pattern recognition.

The majority of syntactic pattern recognition techniques are based on the transformation of complex patterns into simpler subpatterns, using hierarchical decomposition [31]. The decomposition may be performed a number of times, until *pattern primitives* are obtained. An example of this is shown in Figure 2.13, where part of a square is gradually decomposed into a number of simple primitives—short edges and corners.

Formal language theory is the basis of this decomposition; each object to be recognised has a *language*, and the rules which define the composition of primitives into patterns is

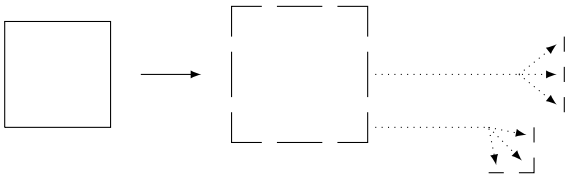


Figure 2.13: Decomposition of part of a square into simple primitives

the *grammar*. Anderson [2] defines a grammar as a 4-tuple:

$$\Gamma = (N, \Sigma, S, P) \tag{2.20}$$

In this tuple N is a finite set of non-terminal symbols, Σ is a finite set of terminal symbols, S is the starting symbol, and P is a finite set of productions. Σ is also known as the language's alphabet, as all valid *words* within the language will be some combination of these symbols Σ^*. In the case of syntactic pattern recognition, the selected pattern primitives form the set of terminal symbols.

The starting symbol S is a member of the non-terminal symbols:

$$S \in N \tag{2.21}$$

Additionally, N and Σ are disjoint:

$$N \cap \Sigma = \emptyset \tag{2.22}$$

A production in the set P is an ordered pair of strings (v, v')—both elements of $(N \cup \Sigma)^*$, where the first element must contain a symbol from N:

$$v = (N \cup \Sigma)^* N^+ (N \cup \Sigma)^* \tag{2.23}$$

$$v' = (N \cup \Sigma)^* \tag{2.24}$$

Finally, if W and W' are elements in $(N \cup \Sigma)^*$, where $W = uvw$, $W' = uv'w$, and $(v, v') \in P$, then $W \Rightarrow W'$ is known as a derivation.

In at least one production, $v = S$ in order that derivation may begin. After following a series of productions, we arrive at a word containing only terminal symbols. Any word in Σ^* that is able to be produced by the set of productions P is said to belong to the language L generated by the grammar Γ, which is denoted by $\Gamma(L)$.

Recognising a word using a grammar is the opposite of deriving it. In this process, we *parse* the word—attempting to find a tree of derivations which produces it. Parsing can be either top-down, beginning with the starting state and working towards the terminals, or bottom-up, working in the other direction.

Fu [32] notes two particular difficulties with the use of syntactic pattern recognition. The first is the initial selection of primitives to use—in some applications it is not obvious what the primitives should be. As an example individual strokes are considered good primitives for handwriting, however their extraction from an image is not an easy task for a machine. Primitive selection is thus a compromise between its use as a basic component of the pattern, and its ease of extraction from an image.

When primitives have been selected, the second difficulty becomes apparent—that of constructing the grammar. Grammars may be constructed manually, however this becomes unrealistic for all but the simplest of problems. As such, *grammatical inference* is more commonly used—deriving a grammar from a series of positive and negative examples of patterns belonging to the language. Solutions to this are often computationally complex, while being limited by a sensitivity to noise. A number of approaches exist in an attempt to mitigate or bypass these limitations, including the use of stochastic grammars and the combination of grammatical inference with other techniques [27]. The CANN is an example of this, integrating grammatical inference with features from cellular automata and CMMs.

2.5.3.2 Cellular Automata

The cellular automaton was introduced by von Neumann [109] as a reductionist model for physical self-replicating systems. A cellular automaton is formed by connecting simple processing units, known as cells, into a grid or array. Each cell may be in one of a finite number of states; often cell states are binary, taking the value `1` or `0`.

Cell states are updated in discrete time steps, according to a set of rules which take into account the state of each cell's *neighbourhood*. The neighbourhood of a cell is selected as part of the particular model used; common two dimensional neighbourhoods are the Moore neighbourhood, the 8 cells surrounding a cell, and the von Neumann neighbourhood, the 4 cells directly adjacent to a cell. Given a specific neighbourhood, each possible combination of neighbouring state values defines a rule which determines the new state for a cell.

Although each individual cell is simple, exchanging information with their neighbours

to update their state can lead to complex or emergent behaviour from the cellular automata. Wolfram [113] showed that further than this, cellular automata can in fact be used as a Turing-complete general purpose computer. Their ability to show complex global behaviour from simple local rules makes them well suited to use for parallel and distributed processing [86].

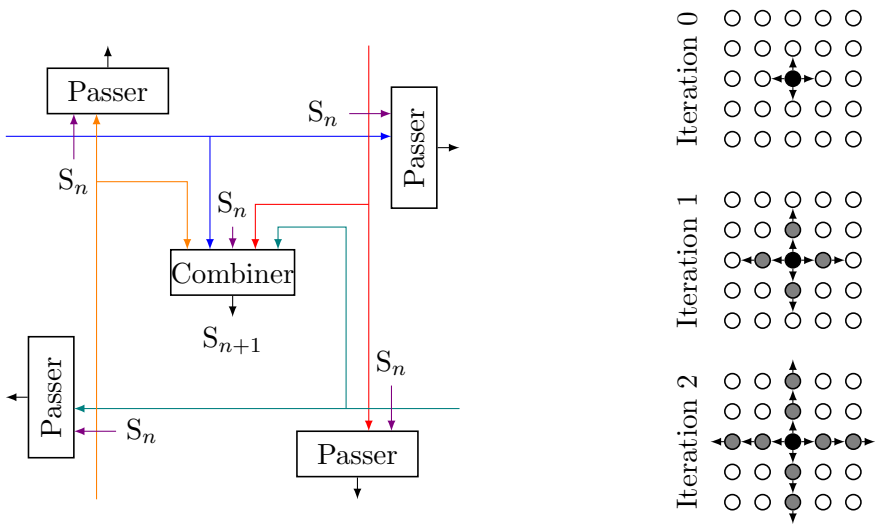
Architectures based on cellular automata have been used successfully to solve low and medium level problems in computer vision, such as gap filling, segment detection, and template matching [28]. They have also been extended to incorporate features from neural networks, with similar applications, using weight matrices and continuous time dynamics to replace the simple automaton rules [24]. As will be discussed in the next section, Orovas [78] introduced the CANN as an alternative architecture which uses simple correlation matrix memories in each cell, in order to provide fast and efficient processing.

2.5.3.3 The Cellular Associative Neural Network

The CANN builds upon previous work by Bledsoe and Browning [15] which used n-tuples to recognise features in a monochrome image. This earlier work was limited as it was unable to recognise a pattern presented in a position other than that in which it was originally taught. The n-tuple method was adapted by Austin [3] to use associative memories, and extended further to allow for processing of grey-scale images [4].

The Advanced Distributed Associative Memory (ADAM) was applied to scene analysis, and combines a CMM with an n-tuple pre-processor to produce sparse, fixed-weight inputs [11]. Using a number of ADAM cells communicating in parallel, Austin et al. proposed a system for map matching [8]. Each CMM uses only a small section of the image as its input—in the order of 10^2 pixels—rather than attempting to use a single CMM for the entire image. This provided easier relaxation and partial matching, as it could be applied on a local scale, rather than across the entire image. The Cellular Neural Network Associative Processor (C-NNAP) was developed as a high-performance parallel implementation of ADAM to remove the obstacle to its use in real-time image processing—namely the large quantity of data required to be processed [61].

The CANN was proposed in [78] as an architecture for pattern recognition, with the suggestion that it could be implemented using C-NNAP in order to improve performance and capabilities. In this architecture, a number of "cells" are arranged in a fashion similar to that used in a cellular automaton [109]. Each cell contains a number of "modules"—a



(a) (b)

Figure 2.14: (a) The "Original Architecture" CANN module configuration, where S_n is the cell state (the output of the combiner module) after the n^{th} iteration and (b) information flow of this configuration over two iterations [16].

CMM or ADAM memory—designed to distribute information to the cell's neighbours, or combine information from its neighbours. The system is designed such that modules may be connected in varying configurations, in order to suit different problems. An example configuration used by Orovas [78] for 2-dimensional pattern recognition is shown in Figure 2.14.

The CANN uses a hierarchical approach to pattern recognition but, rather than requiring a number of distinct layers, it uses an iterative approach. Features are represented symbolically, and these symbols are gradually combined using the rules from a formal grammar in order to build up a representation of the objects in an image—syntactic pattern recognition. The CANN performs bottom-up parsing in order to gradually combine the basic structural elements (such as simple line segments) into higher-level features, and finally into the recognised object. The iterative approach also means that the architecture can be applied to general parallel computing, rather than simply pattern recognition, for any problem which may be represented symbolically.

Orovas' work showed that the CANN was able to recognise patterns with translation invariance, but it was not capable of coping with scale or rotation invariance [76]. Additionally, the patterns recognised were limited to simple line drawings consisting of overlapping rectangles.

Brewer [16] extended the original CANN to include features found in natural neural

networks, such as spike trains, with an aim to improving the applicability of the CANN to photographic input images. The use of spike trains allowed the Spiking CANN to assign a confidence level to potential features represented by the rate of firing of a given pattern, and also allowed the incorporation of leaky integrate and fire (LIF) neurons to provide relaxation at a local level.

A large amount of this work was tested using similar line drawings as in the original CANN, however a small experiment was performed that used photographs in order to gauge the viability of the CANN with real-world data [16]. The results of this showed that the CANN has potential to be used, but that a number of limitations—in particular the lack of scale and rotation invariance—still need to be resolved. A solution to incorporating general invariance into the CANN is described in Section 5.4.

2.6 Summary

After an introduction to artificial neural networks, and particularly feed-forward neural networks, we examined the concept of associative memories. Specialised hardware-based associative memories—content addressable memories—were discussed, before introducing artificial neural network-based alternatives—Hopfield networks, tensor products, and in particular Correlation Matrix Memories (CMMs). These neural approaches offer fast learning and recall, as well as robust operation when presented with a noisy input. Following this the notion of distributed computation is described, and parallel distributed computation—processing multiple superimposed inputs simultaneously—is introduced in preparation for the Associative Rule Chaining Architecture (ARCA) in Chapter 4. Finally a number of neural network-based architectures for pattern recognition are discussed, in particular the Cellular Associative Neural Network (CANN) which will be investigated further in Chapter 5. The remainder of this thesis focuses on the binary CMM, and architectures based on it—hereafter the term CMM will refer specifically to the binary CMM.

Chapter 3

The Extended Neural Associative Memory Language

3.1 Introduction

Domain specific languages (DSLs) can provide many benefits to those working in a domain, by allowing solutions to be developed at the abstract level of the domain [108]. The Extended Neural Associative Memory Language (ENAMeL) has been designed to allow efficient applications using correlation matrix memories (CMMs) to be more easily developed. ENAMeL also provides the potential to greatly reduce the memory requirement of CMM-based systems, due to the use of compact vector representations, which is important to allow these systems to scale and become able to be applied to real-world problems.

Previous work has shown that the abstraction provided by a DSL can often lead to greater developer productivity, when compared to a general purpose language [108]. On the other hand, depending on the domain, it may not be appropriate to design and implement a new DSL due to the time and costs involved. The volume of research using CMMs is sufficient that the development of a DSL for this domain is worthwhile. Additionally, the existence of a DSL in this domain may help to spur further research as it lowers the requirements for entry to the field.

3.2 The Extended Neural Associative Memory Language

While there are only a few basic operations required to work with simple CMMs—the ability to create matrices and to train and recall vectors from these matrices—more complex

CMM-based architectures require additional operators. Further to this, a number of compound operations are included as an extension to the language, as they are commonly used in CMM-based architectures and can be significantly more efficient if performed internally.

In some cases, operands are required to be a specific type—either a vector or a matrix. In these cases, ENAMeL will convert between types wherever it is possible, using the given rules to disambiguate between possible alternative interpretations. In the tables below, an emboldened lowercase character is used to represent a vector, an emboldened uppercase character represents a matrix, and an italicised character indicates an integer. In other cases an operation can be applied equally to both vectors and matrices, and so a type is not specified. Variable names are not case sensitive and can contain any alphabetic characters except "v", as this is used as an operator.

3.2.1 Simple Operators

The simple operators are shown in Table 3.1 below.

Operation	Description
$\mathbf{a}$ = Pattern l $x_0 \ldots x_{n-1}$	Create a vector $\mathbf{a}$, with length l and bits $x_0 \ldots x_{n-1}$ set to one, implying a weight n. This allows arbitrary external vector generation algorithms to be used.
$\mathbf{A}$ = Matrix r c	Create a matrix $\mathbf{A}$ with r rows and c columns.
$\mathbf{A} = \mathbf{b}{:}\mathbf{c}$	Bind vector $\mathbf{b}$ to vector $\mathbf{c}$, forming their outer product and storing the result as matrix $\mathbf{A}$. If either of $\mathbf{b}$ or $\mathbf{c}$ are matrices, then they will first be converted into a vector in a row-major order.
$\mathbf{A} = \mathbf{B}\mathrm{v}\mathbf{C}$	Superimpose $\mathbf{B}$ and $\mathbf{C}$, storing the result in $\mathbf{A}$. If $\mathbf{B}$ and $\mathbf{C}$ are not of the same type, then the type of $\mathbf{A}$ will be the same as that of $\mathbf{B}$.
$\mathbf{a} = \mathbf{b}.\mathbf{C}$	Recall vector $\mathbf{b}$ from matrix $\mathbf{C}$, storing the result an a non-binary vector $\mathbf{a}$. If $\mathbf{b}$ is a matrix with total length equal to the height of $\mathbf{C}$, then $\mathbf{b}$ is first converted into a vector in a row-major order. If $\mathbf{C}$ is a vector with a length that is a multiple of the length of $\mathbf{b}$, then $\mathbf{C}$ is first converted into a matrix in a row-major order.
$\mathbf{A} = \mathbf{B}\vert n$	Apply Willshaw's threshold function to $\mathbf{B}$ with a value of n, storing the result in $\mathbf{A}$.
$\mathbf{A} = \mathbf{B}$	Duplicate $\mathbf{B}$, storing the copy as $\mathbf{A}$.
Print $\mathbf{A}$	Print the contents of $\mathbf{A}$ to the standard output stream, using a compact representation.

Table 3.1: Simple operators available in ENAMeL

3.2.2 Advanced Operators

To allow more advanced architectures to be developed using CMMs, the operators shown in Table 3.2 are also required.

Operation	Description
a = Vector **B**	Create a vector **a** by converting matrix **B** into a single-row vector in a row-major order.
A = Matrix r c **b**	Create a matrix **A** with r rows and c columns, by converting vector **b** in a row-major order. If **b** is a matrix, then it is first converted into a vector in a row-major order.
A = **B**+**C**	Sum the individual bits of **B** and **C**, storing the result in **A**. The result of a sum operation is non-binary. If **B** and **C** are not of the same type, then the type of **A** will be the same as that of **B**.
a = **b**/n	Apply the L-max threshold function to vector **b** with a desired output weight of n, storing the result in vector **a**.
a = **B**#n	Extract the n^{th} column from matrix **B**, storing the result as vector **a** (column indexing is zero-based).
a = n#**B**	Extract the n^{th} row from matrix **B**, storing the result as vector **a** (row indexing is zero-based).
A = Transpose **B**	Transpose the matrix **B**, storing the result as matrix **A**.
a = Append **b** **c** ...	Append the vectors **b**, **c**, etc, storing the result as vector **a**. If any of the operands is a matrix, then it is first converted into a vector in a row-major order.

Table 3.2: Advanced operators available in ENAMeL

3.2.3 Compound Operators and Additional Functions

The remaining operations in the ENAMeL language include a number of useful functions and two compound operators, as shown in Table 3.3.

3.2.4 Combining Operations

The binary operations can be combined, using parentheses, to reduce the quantity of code required and in some cases improve the performance. For example to associate two vectors **b** and **c** in a pre-existing CMM **M**, the explicit instructions would be **A** = **b**:**c** followed by **M** = **M**v**A**. To avoid the overhead of storing the temporary CMM **A** and then merging this with **M**, the combined command **M** = **M**v(**b**:**c**) can instead be used.

Another common example is the recall of a vector **b** from a CMM **M**, followed immedi-

Operation	Description
Generator $l_0 \ldots l_{n-1}$	Create a vector "generator" that uses Baum's algorithm [13] to generate vectors with weight n and length $\sum_{i=0}^{n-1} l_i$. The vectors are divided into n sections of length l_i, each with a single bit set to one, and each length l_i should be co-prime with every other.
$\mathbf{a}$ = Vector w l	Generate a vector $\mathbf{a}$ with weight w and length l, using an appropriate vector generator. This generator must have already been created.
$\mathbf{a} = \mathbf{b} \backslash n$	Apply the L-wta threshold function to vector $\mathbf{b}$ with a desired output weight of n, storing the result in vector $\mathbf{a}$. The lengths of individual sections of the vector are taken from an appropriate generator.
a = Weight $\mathbf{B}$	Calculate the weight of $\mathbf{B}$, storing the result as integer a. The weight of a vector or matrix is the number of bits set to one that it contains.
$\mathbf{A} = \mathbf{B}.\mathbf{C}$	Recall matrix $\mathbf{B}$ from matrix $\mathbf{C}$, storing the result as a non-binary matrix $\mathbf{A}$. This alternative recall is used in the case that $\mathbf{B}$ and $\mathbf{C}$ are both matrices with the same height. In this compound recall, each column of $\mathbf{B}$ is recalled in turn from the matrix $\mathbf{C}$, with the output non-binary vector being placed in the same column position in $\mathbf{A}$ as the input vector was taken from $\mathbf{B}$.
$\mathbf{a} = \mathbf{B}$,$\mathbf{C}$,n	Recall matrix $\mathbf{B}$ from matrix $\mathbf{C}$ as above, in this case applying Willshaw's threshold function with a value of n to each recalled vector. Each of the recalled binary vectors are then summed together, resulting in a single non-binary vector output $\mathbf{a}$.
Exit	Exits the ENAMeL interpreter.

Table 3.3: Additional and compound operators available in ENAMeL

ately by a threshold operation. Instead of $\mathbf{a} = \mathbf{b}.\mathbf{M}$ followed by $\mathbf{a} = \mathbf{a}|n$, the combination $\mathbf{a} = (\mathbf{b}.\mathbf{M})|n$ can be used. Not all combinations of commands will improve the efficiency of execution, however they can reduce the quantity of code required and help to improve the "self-documenting" nature of code written using this DSL. A minimal example of ENAMeL code is given in Appendix A.1.

3.3 Storage Mechanisms

One important benefit of CMMs is considered to be that they offer fast learning and recall. Selection of the storage mechanisms used for the data structures in ENAMeL is thus important, as this can have a great bearing on the time complexity of each operation and the memory requirement of a CMM-based system.

There are a number of storage mechanisms that could be employed to store binary

vectors and matrices. They would not all be appropriate, however, due to the significant detrimental effect they would have on the speed of operations. An example of this is run-length encoding (RLE). RLE might seem to be an ideal choice—in particular the storage of sparse binary vectors and matrices would be expected to be very efficient as long runs of zeros can be compressed to an integer representing the length of the run. In order to perform a recall, we must be able to quickly access individual rows and cells of the matrix. Unfortunately fast access to a row is not possible when using RLE; in order to find the start of a particular row, a linear scan from the beginning of the matrix must be used.

Due to these limitations, imposed by many compression algorithms, the focus has been directed on three possible storage mechanisms—non-sparse storage, a binary form of the Yale format [30], and a hybrid of these. Non-sparse storage is the simplest mechanism that may be employed—every bit is stored explicitly as either a `0` or `1`. This provides a number of benefits such as the direct access to specific rows and bits in vectors and matrices, as the address can be easily calculated. Additionally, the memory required by a structure is well-defined and fixed throughout the life of that structure. On the other hand this fixed memory requirement may be considered as a limitation, as a structure will require the same memory regardless of how much information it stores—whether it is sparsely filled or fully saturated.

The Yale format is a sparse storage mechanism, and as such it only stores non-zero elements [30]. It uses three arrays for this—array A stores the non-zero values, JA stores the column positions for each of these values, and IA stores indexes into the A and JA arrays to denote the beginning of each matrix row. Figure 3.1 shows a matrix represented both non-sparsely and using the Yale format. As an example, the locations of the non-zero elements of the 4^{th} row (which contains the values `00052`) are found by accessing locations 3 and 4 of IA (zero-based indexing)—6 and 8 respectively. The column positions and values of the non-zero elements in the 4^{th} row can then be found in JA and A between locations 6 and 7 (zero-based indexing), showing that the rightmost two columns contain `5` and `2`.

If the matrix to be stored is known to contain only binary values, then the array A is no longer necessary—if a column position exists in the JA array for a given row, then the bit is set, otherwise it is zero. Figure 3.2 shows a binary matrix represented both non-sparsely and using the binary Yale format. It is accessed in the same way as the original Yale format, except that the value of a non-zero element is not stored in the array

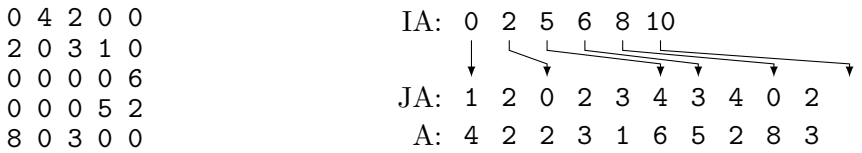


Figure 3.1: An example matrix represented non-sparsely (on the left) and using the Yale format (on the right). In the Yale format the first array (IA) stores indexes into the other arrays to denote the beginning of each matrix row. The second array (JA) stores the column indices of non-zero elements. Finally, the main array (A) stores the values of each of the non-zero elements.

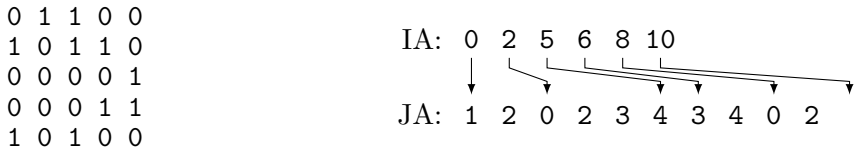


Figure 3.2: An example binary matrix represented non-sparsely (on the left) and using the binary Yale format (on the right). In the binary Yale format the first array (IA) stores indexes into the second array (JA) to denote the beginning of each matrix row. JA then stores the column indices of bits set to one.

A as it is known to be `1`. To store a binary vector only the JA array is used, storing the positions of all set bits in the vector.

The final storage mechanism investigated is a hybrid solution [21], which uses the smallest of the two preceding mechanisms on a row-by-row basis. A single array contains pointers to each of the rows, which may be stored either non-sparsely or using the binary Yale format—selected independently of every other row. Figure 3.3 shows a binary matrix represented both non-sparsely and using the hybrid format. In this case the maximum value that may need to be stored in a JA array is `5`, and so 4 bits can be used to store the column index of each set bit. This means that any row with more than one set bit is smallest when stored non-sparsely. This hybrid format allows a sparsely filled matrix to use minimal memory, while enforcing an upper bound on the memory required for a saturated matrix—that of the non-sparse storage (plus the fixed overhead required to store an array of pointers to each row).

3.3.1 Experimentation

Previous work, for example [51,52,104], has focused on the storage capacity of CMMs under various conditions—using different threshold functions, varying the length and weight of input and output vectors, and so on. Although this is an important consideration when experimenting with memory usage, it is unlikely that a CMM-based system would always

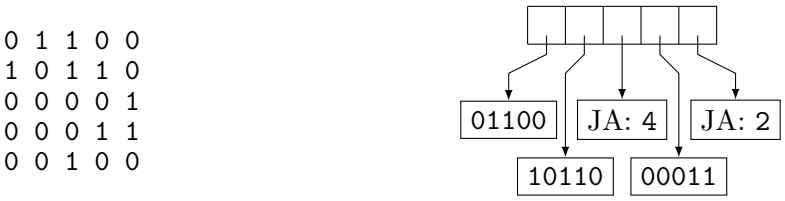


Figure 3.3: An example binary matrix represented non-sparsely (on the left) and using the hybrid format (on the right). In the hybrid format the array stores a reference to each row of the matrix. Each individual row is then stored either non-sparsely, or using an array (JA) which stores the column indices of bits set to one. In this case, a column index in JA would be stored using 4 bits.

operate with fully-saturated matrices. As such, it is more important to determine how the memory requirement changes with respect to the number of pairs of vectors associated within the matrix.

A number of experiments have been performed in order to understand how the memory requirement of a CMM is affected by its contents. The ENAMeL interpreter has been used to implement a simple CMM using various different parameters—vector lengths, l_{in} and l_{out}, and vector weights, w_{in} and w_{out}. For each experiment a set of n vector pairs was randomly selected, where n was chosen as 1.5 times the estimated capacity of the CMM (predetermined experimentally given the particular parameters in use). Each pair of vectors was then associated in the CMM in turn, calculating after each storage operation the recall reliability and the memory requirements for the resulting CMM if it were stored non-sparsely, using the binary Yale format, and using the hybrid format. Using a larger than necessary value for n is intended to provide a more realistic simulation than simply using the estimated capacity as n. In a larger system it would be unusual for every vector to be used, and so a greater number of input and output vectors are allocated than the capacity of the system.

When a non-sparse representation is used, the memory required M_N is determined by the input and output vector lengths—l_{in} and l_{out} respectively—as shown in Equation 3.1.

$$M_N = l_{in} l_{out} \tag{3.1}$$

The memory required when using the binary Yale format is more difficult to calculate, as it is dependent on the number of set bits within the CMM which in turn is dependent on the interactions and overlaps between different pairs of vectors associated. By assuming that there are no overlaps within the CMM an upper bound for the number of set bits,

S can be calculated as shown in Equation 3.2, where n is the number of vector pairs associated, and w_{in} and w_{out} are the respective weights of input and output vectors.

$$S = \min(w_{in}w_{out}n, l_{in}l_{out}) \tag{3.2}$$

Given this upper bound on the number of set bits within a CMM, we can calculate an upper bound on the memory required to associate n vector pairs, M_Y. The memory required to store a single bit in the binary Yale format is dependent on the vector lengths used. The binary Yale format uses two arrays, IA and JA, as shown in Figure 3.2. IA denotes the beginning of each row of the matrix as an index into the JA array, as well as a final index pointing to one past the end of the JA array in order to record the end of the matrix. It must therefore be able to store a value equal to the maximum number of elements in the JA array. This is calculated simply by multiplying the lengths of the input and output vectors of the matrix—$l_{in} \times l_{out}$.

The JA array stores column indices of any bits set to one, and as such must be able to store a value equal to the number of columns (the length of the output vector), l_{out}, minus one (as zero-based indexing is used).

Having calculated the maximum values that may be required to be stored, it is simple to calculate the minimum number of bits required to store S bits as shown in Equation 3.3. The ceiling of the binary logarithm of a value n finds the number of bits required to represent integers in the range $[0, n-1]$ in binary.

$$M_Y = (l_{in}+1)\lceil\log_2(l_{in}l_{out}+1)\rceil + S\lceil\log_2(l_{out})\rceil \tag{3.3}$$

In reality this is not practical, as it will often result in storing an integer using a number of bits that is not a power of two—significantly impacting the performance as values will be stored across byte- and word-boundaries. In order to resolve this we can simply round the number of bits up to the nearest power of two, as shown in Equation 3.4.

$$M_Y = (l_{in}+1)2^{\lceil\log_2\lceil\log_2(l_{in}l_{out}+1)\rceil\rceil} + S * 2^{\lceil\log_2\lceil\log_2(l_{out})\rceil\rceil} \tag{3.4}$$

The hybrid storage format is harder to analyse, as the memory required is dependent on the number of set bits within each row of the matrix. The matrix has an overhead of one pointer for each row—typically using 32-bit pointers, although 64-bit pointers can be used

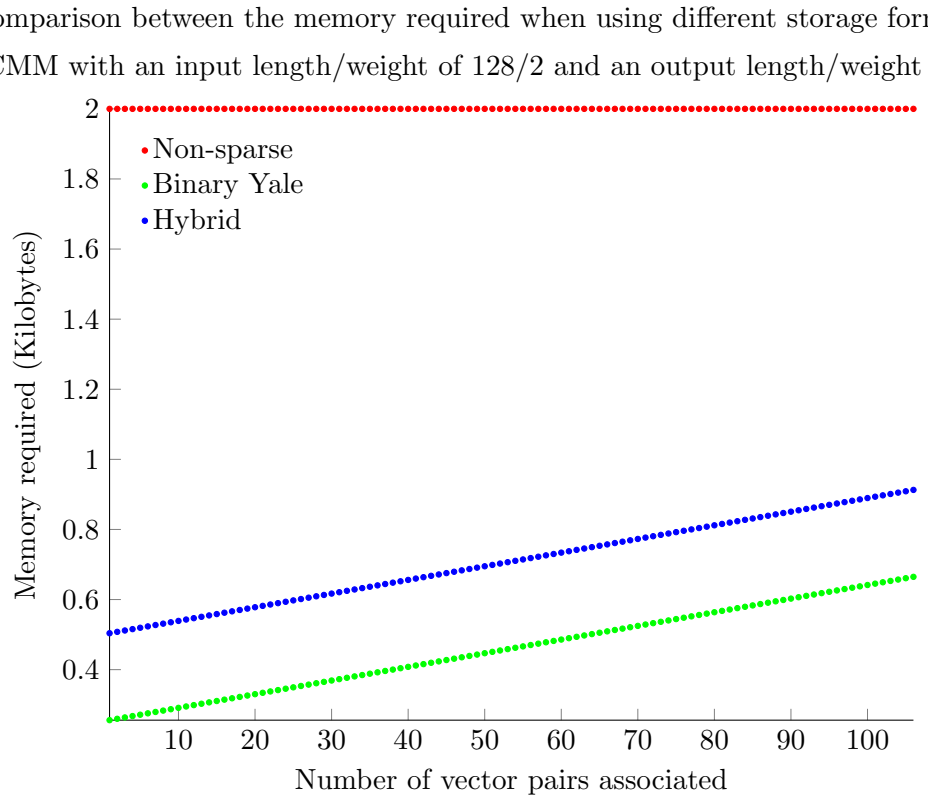
if the data size exceeds the 32-bit limit and the system architecture allows. The individual rows are then stored using either a non-sparse format, or the binary Yale format—in this case requiring only the JA array for each row. This variability means that as the number of associated vector pairs increases, the memory required by the hybrid format is expected to increase in line with the binary Yale format—until the point at which a non-sparse representation requires less memory, at which point it will switch to a fixed maximum size.

3.3.2 Results

Each of the experiments has been run 100 times, using a different seed for the random number generator, and the results averaged across all runs. The graphs in the following figures are scatter plots showing the memory requirements of a single CMM when using different storage formats, as the number of associated vector pairs increases. Each point is shown if the CMM was able to successfully recall at least 99% of all trained vectors.

Figure 3.4 shows clearly the benefit that using sparse storage for CMMs can provide. At the top, the vector lengths are 128-bits with a fixed weight of 2. In this case the binary Yale format requires consistently less memory than the hybrid format. Both formats store column positions in a similar way using the JA array, however the IA array of the binary Yale format is able to store 16-bit integers as indices whereas the hybrid format must use 32-bit pointers. At the bottom of Figure 3.4, the vector lengths are 1024-bits with a fixed weight of 2. As the total size of the matrix has surpassed 2^{16}-bits, the IA array is required to use 32-bit integers and so the binary Yale format requires the same amount of memory as the hybrid format.

The next results, shown in Figure 3.5, help to illustrate the effect that the storage format used can have on the memory requirement of a CMM when changing the input and output vector lengths. Previous work has shown that a $l_{in} \times l_{out}$ CMM can have a higher capacity if l_{in} is greater than l_{out} rather than vice versa, even when the total size of the CMM is equal [52]. For example a CMM with an input length of 512-bits and output length of 256-bits has a higher capacity than the alternative 256- by 512-bits CMM. If a non-sparse storage format is used then the memory requirement of both matrices will be identical and selecting parameters which lead to the highest capacity is therefore the most sensible option. If a sparse storage format is used, then the memory requirement will be affected by the length of IA array—as the input vector length increases, so too does the



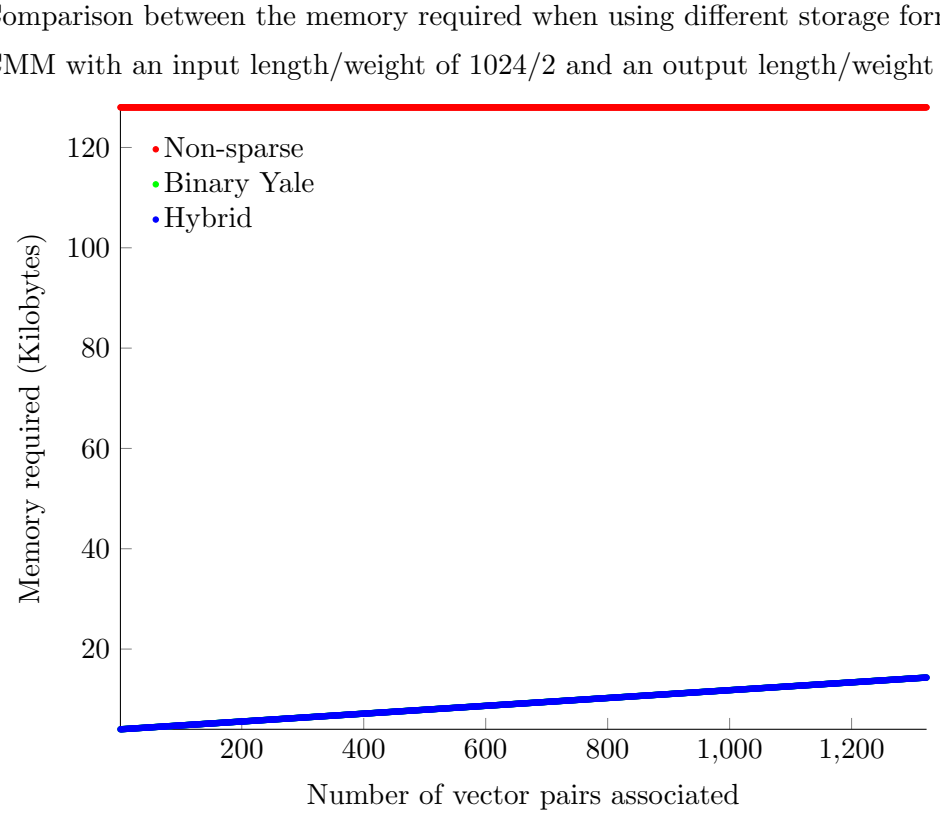


Figure 3.4: Scatter plots showing the memory requirements of a CMM using different storage formats, as the number of associated vector pairs increases. Each point is shown if the CMM was able to successfully recall at least 99% of all trained vectors. All vector weights are 2; the vector lengths are 128-bits (top) and 1024-bits (bottom).

Table 3.4: The memory required to associate vector pairs in CMMs of varying size. The maximum capacity of a CMM is calculated as the highest number of vector pairs trained where at least 99% of them were successfully recalled. The weight of all vectors used was set as 2.

Vector lengths (bits)		Vector pairs	Memory required (KB)		
Input	Output	(max. capacity)	Non-sparse	Binary Yale	Hybrid
256	128	100 (160)	4.0	0.9	1.4
128	256	100 (112)	4.0	0.6	0.9
1024	512	750 (988)	64.0	9.9	9.9
512	1024	750 (788)	64.0	7.9	7.9
2048	1024	1800 (2466)	256.0	22.1	22.1
1024	2048	1800 (1856)	256.0	18.1	18.1
4096	2048	4200 (6240)	1024.0	48.8	48.8
2048	4096	4200 (4508)	1024.0	40.8	40.8

Comparison between the memory required when using different storage formats for a CMM with vector weights of 2, and input/output lengths of 512/1024 or 1024/512

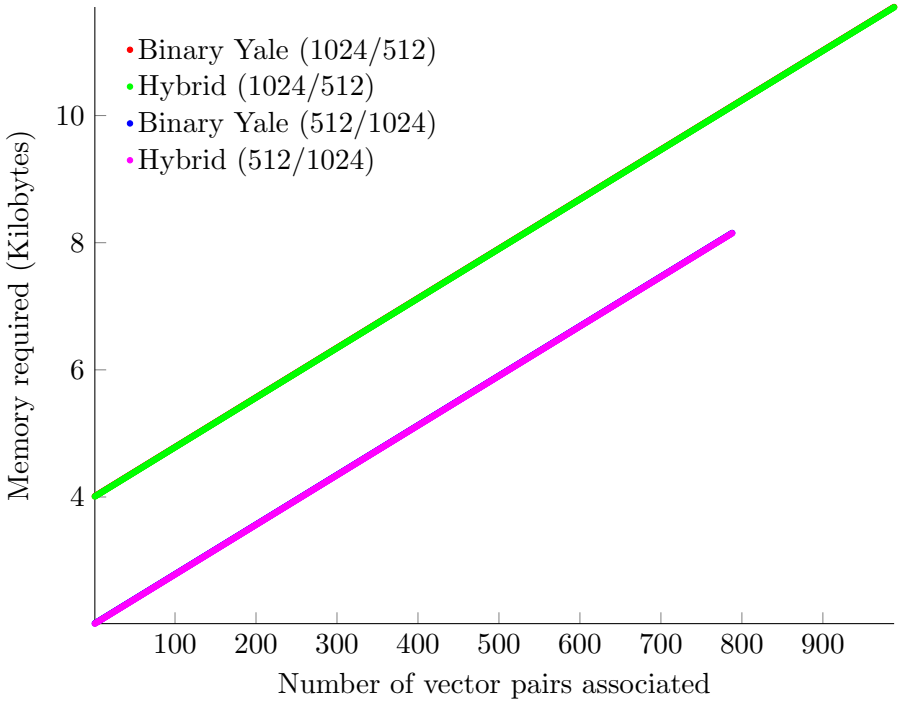


Figure 3.5: Scatter plots showing the memory requirements of a CMM using different storage formats, as the number of associated vector pairs increases. Each point is shown if the CMM was able to successfully recall at least 99% of all trained vectors. All vector weights are 2; the input/output vector lengths are either 1024/512-bits or 512/1024-bits. The memory required when using non-sparse storage is not shown, as this was a constant 64KB throughout.

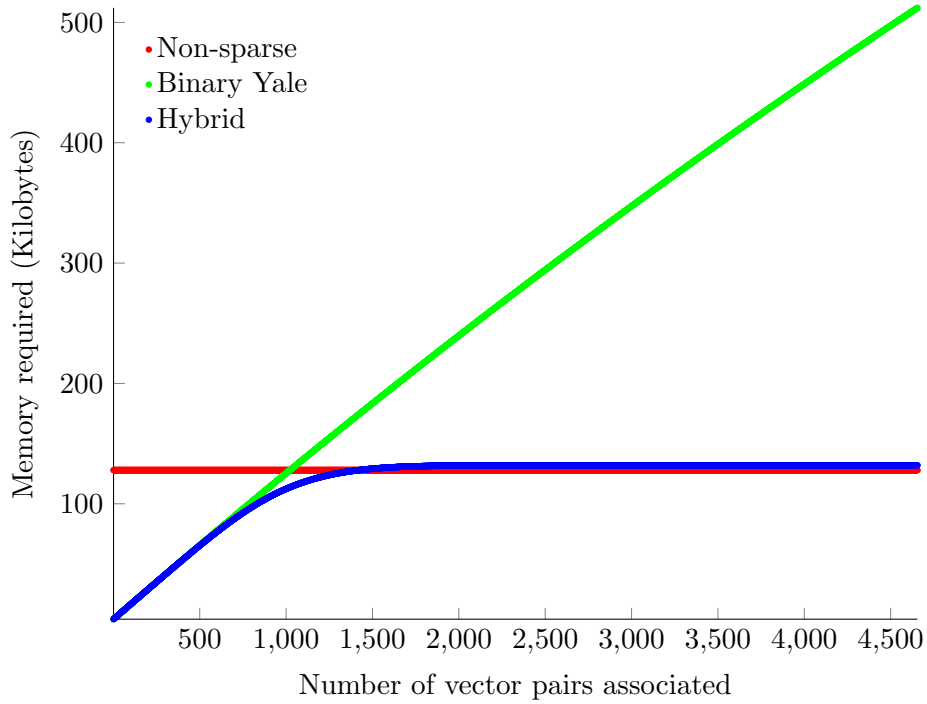
number of indices or pointers which must be stored. However, as the number of vector pairs associated increases, the size of the IA array is expected to become less important to the overall memory requirement.

Figure 3.5 shows the memory required when the vector weights are 2, and either the input vector length is 1024-bits with an output vector length of 512-bits or vice versa. In each case, the memory required by the Binary Yale format is equal to that of the Hybrid format. The memory required when using non-sparse storage is constant, at 64KB, and so this is not shown on the plot. These results demonstrate that, as expected, the capacity is greatest with the longest input length and least with the shortest input length. However they also show that the memory requirement does not always follow this pattern. As an example, Table 3.4 shows the memory required to associate 750 pairs of vectors in each of these CMMs—as 750 is the approximate capacity of the 512 / 1024 CMM while still achieving at least 99% recall success. In this case the size of the IA array has a significant effect on the memory requirement—the 1024 / 512 CMM requires 25% more memory than the 512 / 1024 CMM.

In order to determine how the size of the IA array affects the memory requirements of different sizes of CMM, the table also shows the results for three other pairs of input and output lengths. As expected, the significance of the IA array decreases as the size of the CMM increases. For the hybrid format the additional memory required to use a longer input vector than output vector, compared to the opposite, falls from 56% for a 256 / 128 CMM to 20% for a 4096 / 2048 CMM. When the number of associated vector pairs is also taken into consideration, the decreasing significance of the IA array is demonstrated further—the overhead decreases from 40.9-bits per associated vector pair for a 256 / 128 CMM to only 15.6-bits per pair for a 4096 / 2048 CMM. Given these results, and the large increase in capacity available when increasing the input vector length rather than the output vector length, it is clear that the memory requirement of overheads such as the IA array is not significant when selecting the parameters of a CMM.

The final results, shown in Figure 3.6, show the benefit of using a hybrid format rather than choosing to use non-sparse storage or the binary Yale format. The relationship between the weight of vectors and the capacity of a CMM is a complex issue. It has been shown that the maximum capacity of a CMM can be achieved when vectors have a weight approximately equal to $\log_2 l$, where l is the vector length [82]. At the top of Figure 3.6 all vectors have a length of 1024-bits and a weight of 8. In this size of CMM it is clear

Comparison between the memory required when using different storage formats for a CMM with an input length/weight of 1024/8 and an output length/weight of 1024/8



Comparison between the memory required when using different storage formats for a CMM with an input length/weight of 1024/16 and an output length/weight of 1024/2

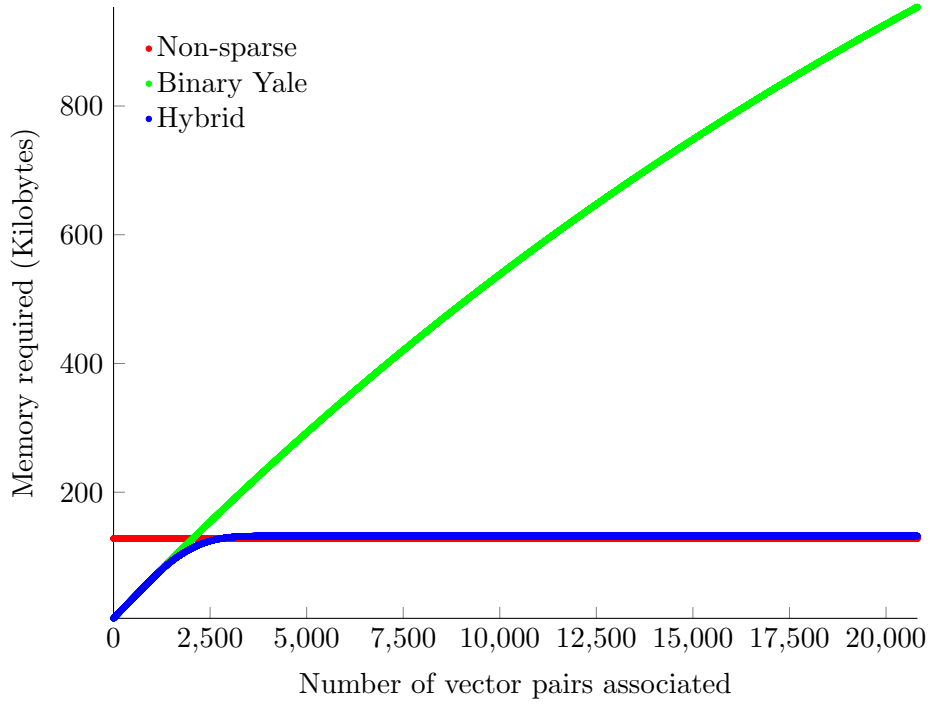


Figure 3.6: Scatter plots showing the memory requirements of a CMM using different storage formats, as the number of associated vector pairs increases. Each point is shown if the CMM was able to successfully recall at least 99% of all trained vectors. All vector lengths are 1024-bits; the input/output vector weights are 8/8 (top) and 16/2 (bottom).

that the use of sparse storage is suitable for low numbers of associated vector pairs, but quickly becomes inefficient as the number of associated vector pairs increases. The hybrid format is ideal in this situation, as it limits the maximum memory requirement to that of the non-sparse storage.

3.3.3 Independently Varying the Vector Weights

Choosing the weight of vectors as $\log_2 l$ in order to achieve the maximum capacity for a CMM operates under an assumption that input and output vectors must be of equal length and weight. If this is not a requirement, and the vectors may be selected from different vector spaces, then it is possible to achieve even greater capacity—as shown at the bottom of Figure 3.6. In this case the input vector weight is 16 and the output vector weight is 2, and the capacity has increased to 20813 pairs of vectors from 4655 pairs.

The reason for this is two-fold. Firstly, decreasing the weight of vectors will similarly decrease the number of correlations to be stored within the CMM for every vector pair that is associated, thus using the lowest possible weight will reduce interference within the CMM and increase the capacity. On the other hand, as we saw in Section 2.3.3, a recall operation also includes the application of a threshold—in this case using Willshaw's threshold at a value equal to the weight of an input vector. This threshold mechanism "cleans" the output of a CMM, allowing only those values which meet the threshold value to pass through as a `1`. Increasing this threshold value by increasing the input vector weight helps to counteract the interference within the matrix.

It is anticipated that using a weight of 2 for output vectors will maximise the capacity as this will not affect the threshold value, but will decrease the number of correlations to be stored within the CMM for every associated vector pair—reducing interference within the CMM. The input vector weight must take both of the influences into account and so increasing this weight is expected to increase the capacity of the CMM up to a point at which the additional interference introduced exceeds the additional capability of the threshold to "clean" the output.

Figure 3.7 shows a heat map of the capacity of a CMM where all vectors have a length of 1024-bits, and the input and output weights have been independently varied. A CMM is deemed to have reached its capacity at the point that training another vector pair would mean that it is no longer possible to successfully recall at least 99% of all the vectors which have been associated within the CMM. For this length of vector, the weights are limited

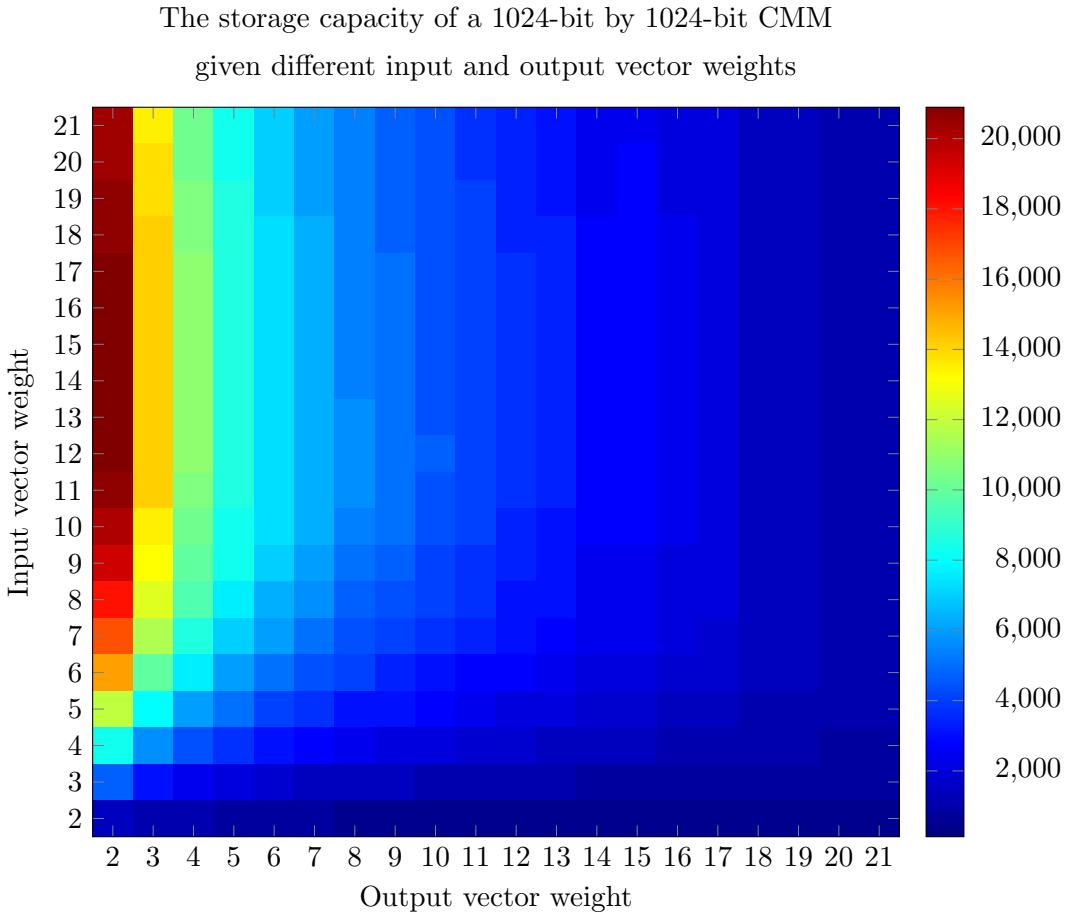


Figure 3.7: Heat map showing how the capacity of a CMM changes as the input and output vector weights are independently varied. The capacity is considered as the point at which the number of associated vector pairs cannot be increased without the recall success rate falling below 99%. All vector lengths are 1024-bits; the input and output vector weights are independently varied between 2 and 21.

to 21 due to the difficulty of finding suitable coprime numbers to be used as the basis for vector generation using Baum's algorithm.

It is clear to see that, as expected, decreasing the output vector weight increases the capacity of a CMM—the highest capacity achieved at any given input vector weight was always with an output vector weight of 2. Selecting an input vector weight in order to maximise the capacity of the CMM is less clear. For the majority of output vector weights the highest capacity was achieved with an input vector weight of either 13 or 15, as can be seen in Table 3.5. Given the trends—increasing capacity until the input vector weight of 13, and decreasing after 15—it would seem that the input vector weight of 14 is anomalous. Looking in more detail at the vector generation can, however, help to explain this.

As we saw in Section 2.3.5.3, Baum's algorithm can be used to generate vectors with

Input vector weight	Output vector weight												
	2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
21	20023	13371	10112	8133	6776	5846	5016	4516	4030	3647	3230	2954	2311
20	20133	13504	10175	8180	6808	5835	5042	4545	4060	3689	3245	2987	2346
19	20238	13621	10260	8274	6928	5931	5110	4608	4118	3735	3292	3016	2359
18	20551	13730	10371	8349	6962	5985	5163	4671	4166	3776	3343	3046	2400
17	20826	13965	10560	8487	7090	6114	5280	4737	4228	3843	3405	3109	2438
16	20818	13978	10558	8519	7103	6092	5261	4729	4260	3872	3424	3140	2473
15	20876	14063	10634	8531	7147	6175	5320	4810	4314	3899	3458	3160	2481
14	20694	13902	10511	8483	7097	6114	5313	4770	4290	3887	3449	3169	2484
13	20842	14065	10609	8567	7151	6175	5359	4846	4327	3971	3496	3216	2520
12	20695	13946	10524	8537	7121	6175	5358	4859	4355	3942	3502	3186	2523
11	20369	13721	10433	8389	7052	6157	5348	4817	4307	3859	3438	3147	2501
10	19839	13406	10168	8202	6966	6038	5193	4665	4164	3749	3370	3033	2450
9	19101	12853	9855	8055	6728	5769	4989	4486	4006	3607	3207	2901	2383
8	17901	12237	9356	7543	6309	5373	4641	4177	3739	3373	3017	2742	2283
7	16616	11409	8590	6845	5716	4911	4268	3784	3391	3078	2784	2512	2105
6	14716	9809	7388	5931	4969	4264	3714	3332	2976	2710	2444	2243	1884
5	11783	7862	5971	4777	4016	3478	3008	2746	2429	2239	1985	1875	1584
4	8241	5558	4190	3423	2834	2415	2111	1994	1742	1595	1481	1393	1218
3	4420	2926	2302	1885	1573	1429	1252	1094	1088	977	908	817	746
2	1262	969	918	709	575	467	409	366	298	292	273	243	237

Table 3.5: Capacity of a CMM with independently varied input and output weights. The capacity is considered as the point at which the number of associated vector pairs cannot be increased without the recall success rate falling below 99%. All vector lengths are 1024-bits; the input and output vector weights are independently varied between 2 and 21 (not all are shown).

an increased orthogonality compared to randomly generated vectors. In order to do this, the vector length l is divided into w sections (where w is the vector weight) each with a length p_i that is coprime to all others. The algorithm then proceeds to set one bit in each of these sections, incrementing the position for every new vector and wrapping around when necessary. It is guaranteed that if only the first p_1 vectors are used that they will not overlap, if the first $p_1 p_2$ vectors are used, they will overlap with each other in the position of a single set bit, etc. Because of this, the length of the shortest sections can have a bearing on the capacity by affecting the number of vector interactions.

Weight 13: $61, 64, 67, 69, 71, 73, 77, 79, 83, 85, 89, 97, 109$

Weight 14: $43, 49, 55, 59, 61, 67, 69, 71, 73, 79, 89, 97, 103, 109$

Weight 15: $47, 49, 53, 55, 57, 59, 61, 64, 67, 71, 73, 79, 89, 97, 103$

Looking at the coprime numbers above, used to generate vectors with length 1024-bits and

weight 13–15, it is clear that there is a large drop in the length of the first p_i sections when moving from 13 to 14, and a slight recovery when moving from 14 to 15. This will have caused more interference between vectors with an input vector weight of 14, and hence has reduced the capacity of the CMM compared to those with input vector weights of 13 and 15.

3.4 Further Work

Although ENAMeL is complete, and the hybrid storage format has been shown to provide reasonable memory efficiency while still allowing fast access to individual rows and bits of a CMM, there are a number of unanswered questions that have developed during experimentation.

3.4.1 Independently Varying the Vector Weights

The choice of vector weight is very important in determining the capacity of a CMM, affecting both the number of correlations stored within the CMM for every associated vector pair and the threshold used during a recall. Section 3.3.3 showed that using $\log_2 l$ as the vector weight, where l is the vector length, does not provide the highest possible capacity when input and output vectors can have different lengths.

Experimentation has shown that a minimal output vector weight results in the highest CMM capacity, however further work is required to fully understand the relationship between the input vector weight and the capacity of a CMM when vectors are generated using a method such as Baum's algorithm.

3.4.2 Mathematical Analysis of CMM Capacity

In this work CMMs have been trained with synthetic data, in order to determine capacity or memory requirement. When CMMs are applied to a real problem, the required capacity is likely to be determined in advance and the memory requirement may be a constraint. Determining the vector parameters to use in order to achieve this capacity is a difficult problem. An approximation of a CMM's capacity can be obtained using equations in [11] or [104], however these work with randomly generated vectors rather than those created using an algorithm that attempts to increase the orthogonality between vectors—such as Baum's algorithm.

A complete mathematical analysis of the capacity of a CMM storing Baum vectors could be of great use, creating a model that estimates the capacity of a CMM with given input and output vector lengths and weights. In order to be most useful this should then be modified to work from the opposite angle: a user would give the target capacity as an input, as well as any constraints—for example input and output vectors must be the same length—and the model would suggest vector parameters to use in order to achieve this capacity.

3.4.2.1 Analysis of Vector Interactions

As an extension to the mathematical analysis of CMM capacity, modifying the model in order to determine interactions and overlaps between vector pairs associated within a CMM would allow it to be used to estimate the memory requirement of a CMM using different storage mechanisms.

3.5 Summary

A domain specific language for correlation matrix memories was presented. A number of storage mechanisms for CMMs were investigated, in particular a binary form of the Yale format [30] and a hybrid format that uses a combination of sparse and non-sparse storage. This led to a discussion of why alternative storage mechanisms such as run-length encoding are not suitable for CMMs.

The benefit of a hybrid storage mechanism was discussed, and results have been presented that demonstrate the memory efficiency it can provide for CMMs—especially when they are not saturated—using different values for the length and weight of vectors.

Finally, it has been shown that using a weight of $\log_2 l$ for vectors of length l does not provide the maximum capacity for a CMM when input and output vectors are not required to be taken from the same vector space. In this case, reducing the weight of output vectors and increasing the weight of input vectors greatly increases the capacity. In the experimentation, the capacity was able to be increased by up to 400% compared to using $\log_2 l$ as the weight for all vectors.

Chapter 4

The Associative Rule Chaining Architecture

4.1 Introduction

Rule chaining is a common problem in the field of artificial intelligence, searching a tree of rules to determine if there is a path from the starting state to the goal state. Rule chaining is also believed to be performed by the brain, and so a solution that uses biologically-inspired neural networks is of particular interest.

The Associative Rule Chaining Architecture (ARCA) was first proposed by Austin [7], using CMMs to store the rules. Vectors use a distributed representation, allowing for efficient memory use and a greater possibility for fault tolerance than a local representation [13]. States are superimposed throughout the recall operation to reduce the time complexity of rule chaining.

4.2 Rule Chaining

Rule chaining involves the repeated application of a set of rules to a state, in order to infer further information which can be added to that state. The representation of symbols used in ARCA allows only for the assertion of facts as being true, negation is not currently possible.

There are two forms of rule chaining: forward chaining and backward chaining. The choice of which to use is typically governed by the application, with each method having particular advantages and disadvantages [91].

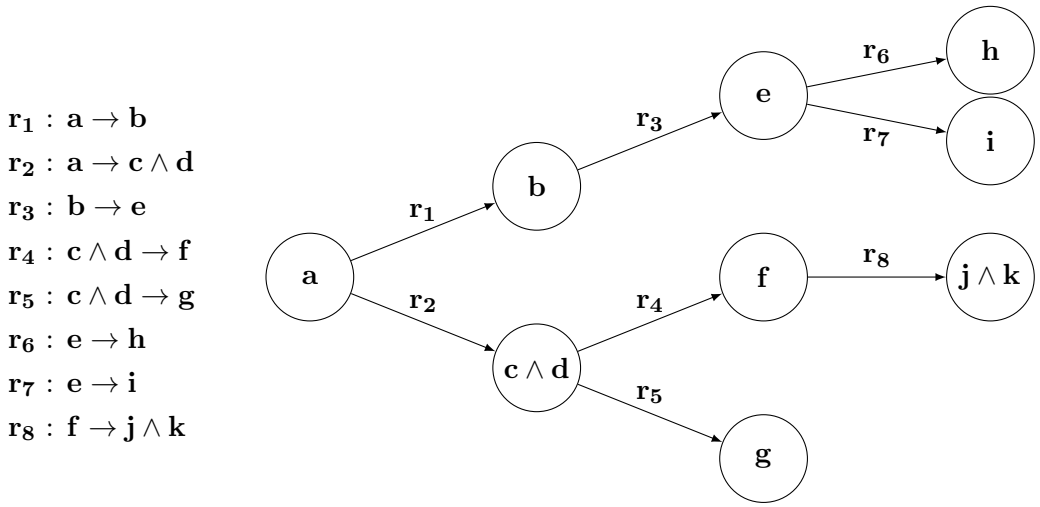


Figure 4.1: An example set of rules represented as a list and a tree

Forward chaining begins by creating an initial state containing any symbols we know to be true, and a goal state containing one or more symbols that we wish to infer. The set of rules is searched to find any for which the antecedents are all present within this state, and the consequents of these matching rules are added to the state. At this stage, the current state can be tested to determine whether the goal state has been reached, and if it has not then the system iterates. If no further rules can be applied to the current state, then the search has completed in failure.

Backward chaining begins in a similar fashion by creating a known state and a goal state, however in this method the search begins at the goal state. The set of rules is searched to find those in which the consequents match the goal state, and the antecedents of these rules are added to the state. The search continues iteratively until the current state has reached a set of symbols contained within the known state, or until no further rules can be applied.

Work on ARCA has focused on implementing forward chaining, but there is no reason that the ARCA system could not be applied to backward chaining as both methods of rule chaining are typically implemented as a depth-first search (DFS) [91]. ARCA implements a tree search that in essence is very similar to breadth-first search (BFS)—examining all of the nodes at a given depth before descending to the next level. All nodes at a given depth are inspected simultaneously, however, and so this aspect bears a similarity to a DFS—the search quickly traverses down the branches of a tree. ARCA therefore implements neither a DFS nor a BFS, but is suitable to be used in place of either.

Figure 4.1 provides a simple example set of rules of the form *antecedents* → *consequents*,

where the majority of the rules are single-arity—meaning that they have exactly one antecedent. Rules 4 and 5 are the exception to this, as they are both 2-arity rules. During a search, if the antecedents of a rule are present in the current state then they are replaced by the consequents of all rules containing the same antecedents. For example, if **e** is present in the current state then both rules $\mathbf{r_6}$ and $\mathbf{r_7}$ will be applied—meaning **e** is replaced by both **h** and **i** in the new state. This is represented more clearly as an unordered search tree, where rules that share the same antecedents branch from a single node.

To demonstrate forward chaining on the rules given in Figure 4.1, we can select **a** as our initial state and **f** as a goal state. Using a DFS, we would first find a match with rule $\mathbf{r_1}$ and follow this branch to **b**. We would continue in turn through the further states of **e**, **h**, and **i** before backtracking to the root and traversing the second sub-tree—finding $\mathbf{c} \wedge \mathbf{d}$ before finally reaching **f**. ARCA has been designed to perform a search in a similar fashion, however it searches each of the sub-trees simultaneously.

In the general case, performing a DFS on a tree has a time complexity of $O(b^d)$, where b is the branching factor of the tree and d is the depth. By searching every branch simultaneously, ARCA can eliminate the branching factor and reduce the time complexity of a tree search—and hence forward chaining—to $O(d)$. A DFS has a space complexity of $O(bd)$, however the issue of space complexity within ARCA is more difficult to analyse. The memory required is constant throughout, as the rules must be initially trained into CMMs. If these CMMs are not sufficiently large, however, then they will become saturated and may recall rules incorrectly. Determining the capacity of a CMM in advance depends on a number of factors, including the representation used for data, as well as the data itself.

4.3 The Associative Rule Chaining Architecture

ARCA performs forward chaining by storing the rules in CMMs and iteratively recalling the current state in order to replace any matched antecedents with the appropriate consequents. Although this provides the advantages of CMMs, such as constant-time rule lookup, it also presents a major challenge that must be overcome. ARCA is designed to perform an unordered tree search similar to that of a breadth-first search, with an important distinction that it examines all nodes at a given depth of the tree simultaneously. This means that we must be able to store multiple branches superimposed within a single vector. More than this, we must ensure that the superimposed branches remain

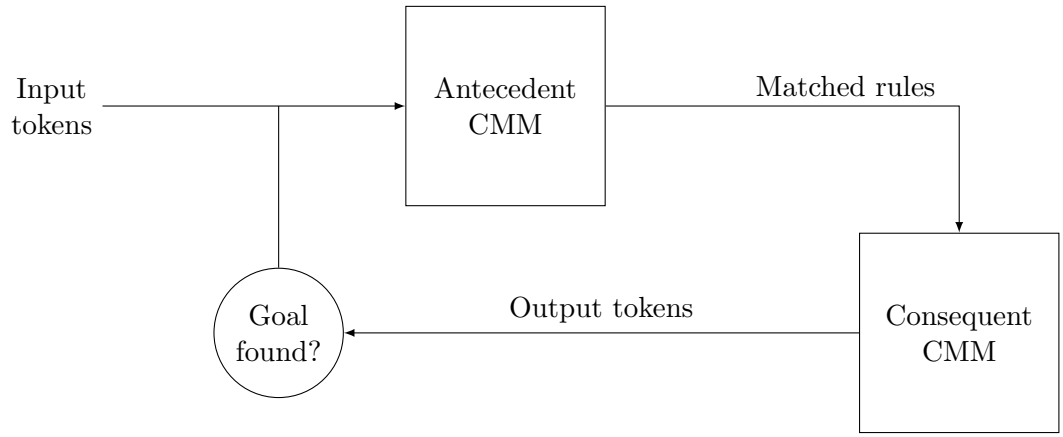


Figure 4.2: Block diagram of ARCA

distinguishable throughout the search.

The architecture and control structure of ARCA is intentionally simple and consists of a simple state machine, as shown in Figure 4.2. Rules are separated into two CMMs—one stores the mapping of antecedents to a rule, and the other stores the mapping of a rule to its consequents. ARCA uses two distinct sets of binary vectors, both with a fixed weight. The "token vectors" represent a single antecedent or consequent. In addition to this each rule is assigned a unique "rule vector", which serves as the connection between the two CMMs. One or more tokens form an initial state and are presented as an input to the antecedent CMM, resulting in an output containing all rules for which the antecedents are matched. These rules are presented as an input to the consequent CMM, which results in a new state containing the consequent tokens of all matched rules. This output can be tested to determine whether the goal state has been found, and if it has not then the system iterates using this state as the new input.

At this point we must revisit the concept of vector superposition from Section 2.3.5.1, in order to illustrate the difficulty of maintaining the separation of individual branches throughout a search. Vector superposition is denoted using the $\vee$ operator—a logical OR—and means that multiple vectors are overlaid to create a single vector.

If a CMM has been trained with two pairs of vectors, $\mathbf{p} \rightarrow \mathbf{q}$ and $\mathbf{s} \rightarrow \mathbf{t}$, then presenting either $\mathbf{p}$ or $\mathbf{s}$ as an input to the CMM will result in either the vector $\mathbf{q}$ or $\mathbf{t}$ being recalled respectively. If the superposition $\mathbf{p} \vee \mathbf{s}$ is presented as an input to the CMM, then the output before thresholding will be the superposition of the expected output vectors, $\mathbf{q} \vee \mathbf{t}$. Willshaw's method of thresholding [112] involves setting any output bit to `1` that has a value greater than or equal to the trained input weight, and every other bit to `0`. When

Table 4.1: A subset of the rules given in Figure 4.1, with a binary vector assigned to each individual vector and rule

Token vector	Binary representation
a	1001000
b	0100100
c	0010001
d	1000010
e	0010010
f	0100001
g	0010100

Rule	Binary representation
$\mathbf{r_1} : \mathbf{a} \rightarrow \mathbf{b}$	10100
$\mathbf{r_2} : \mathbf{a} \rightarrow \mathbf{c} \wedge \mathbf{d}$	01010
$\mathbf{r_3} : \mathbf{b} \rightarrow \mathbf{e}$	10001
$\mathbf{r_4} : \mathbf{c} \wedge \mathbf{d} \rightarrow \mathbf{g}$	01100
$\mathbf{r_5} : \mathbf{c} \wedge \mathbf{d} \rightarrow \mathbf{f}$	10010

using this threshold superimposed recall can operate correctly, as the output weight of each expected bit will still equal or surpass the trained input weight—during a recall this output value cannot be reduced by setting any other input bits to 1.

If we consider vectors **b** and **c** given in Table 4.1, then the result of superimposing them is $0100100 \vee 0010001 \rightarrow 0110101$. This now provides an ideal example of the difficulties inherent in using superposition. The vector 0110101 has just been shown to be the superposition of vectors **b** and **c**. Given the encoding, however, this is ambiguous as it is also the superposition of vectors **f** and **g**. We can be sure the vector does not contain certain vectors such as **a**, as one or more of their set bits are not present, but we cannot be sure of those vectors where all of their set bits are present.

To resolve this difficulty, we must revisit another concept—tensor products—that were introduced in Section 2.3.2. A tensor product is formed by binding two vectors together to form their outer product, represented by the ':' operator.[1] For example $\mathbf{M} = \mathbf{a} : \mathbf{r_3}$ represents binding the vector **a** to the vector $\mathbf{r_3}$ and storing the result as matrix **M**. Using the vectors in Table 4.1 we can illustrate this:

$$\mathbf{a} : \mathbf{r_3} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Tensor products provide separation between superimposed vectors, which in turn allows differentiation between superpositions that are indistinguishable when superimposed

[1] In some work the outer product operation is represented as $\mathbf{a} \otimes \mathbf{b}$, however here it is $\mathbf{a} : \mathbf{b}$ for practical use in ENAMeL.

as vectors. This means that it is possible to know exactly which vectors are present within a superposition. Considering the example earlier, the superposition of vectors **b** and **c** is identical to that of the superposition of vectors **f** and **g**. If we form tensor products before superimposing, however, this problem is resolved—even if we use the same vectors for binding in each pair:

$$(\mathbf{b} : \mathbf{r_2}) \vee (\mathbf{c} : \mathbf{r_3}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad (\mathbf{f} : \mathbf{r_2}) \vee (\mathbf{g} : \mathbf{r_3}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

4.3.1 Training

The antecedent CMM is trained with the associations between the superimposed antecedent tokens of a rule, and the rule vector assigned to that rule. This means that if the antecedent tokens of a rule are present in a particular input, then the rule should fire.

A slightly more complex method is used to train the consequent CMM. Firstly we superimpose the consequents of a rule and bind this to the rule vector, resulting in a tensor product. This tensor product is flattened in a row-major order to form a vector with length equal to $l_t l_r$, where l_t and l_r are the lengths of a token and rule respectively. The consequent CMM is now trained with the associations between the rule vector and this flattened tensor product. This means that when a rule fires from the antecedent CMM, the consequent CMM will produce a tensor product containing the consequent tokens bound to the rule that caused them to fire.

4.3.2 Recall

The recall process is best described with the aid of a diagram, and so we must revisit the visualisation of tensor products introduced in Section 2.3.3.1. Binding a token vector to a rule vector results in a matrix which contains the token vector in every column where the rule vector was set to `1`—every other column contains only `0`s. Figure 4.3 shows a visualisation of a tensor product containing $(\mathbf{b} : \mathbf{r_1}) \vee (\mathbf{c} : \mathbf{r_5})$. Each of the rule vectors has a fixed weight of 2, and so each token vector is present in two of the columns. Columns containing only `0`s are absent from the visualisation. Due to the overlap between rules $\mathbf{r_1}$ and $\mathbf{r_5}$ only three columns contain non-zero elements, however it is still possible to

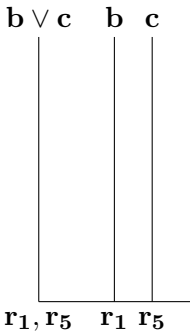


Figure 4.3: A visualisation of the tensor product $(\mathbf{b} : \mathbf{r_1}) \vee (\mathbf{c} : \mathbf{r_5})$ using the binary representations given in Table 4.1. The fixed weight of each rule vector is set as 2, and so each bound token vector is present in two of the columns. The labels at the top of columns describe the tokens contained within that column, and the labels at the base of columns show to which rule vector these tokens are bound.

distinguish between the vectors **b** and **c** as the rule vectors also contain distinct set bits.

Figure 4.4 shows two iterations of a recall process performed on our continuing example. As before, we wish to perform forward chaining on the rules given in Figure 4.1 using **a** as our initial state and **f** as the goal state.

We must firstly create an input state, containing the tokens with which we are starting our search. A state is simply a tensor product containing one or more tokens bound to rules, and so we bind our input token vector **a** to a "dummy" rule $\mathbf{r_0}$ to form $\mathbf{TP_{in1}}$. Each input token vector will exist in this tensor product a number of times equal to the fixed weight of a rule vector, in our example this is twice. The choice of which rule vector with which to bind our initial input token vectors does not have any effect on the recall process, as the rule vectors serve only to maintain separation between multiple superimposed branches of a search.

The first stage of recall is to determine which rules are matched by the tokens contained within our current state. Each column of $\mathbf{TP_{in1}}$ must be recalled in turn from the antecedent CMM, resulting in a series of vectors that contain the rule vectors representing any rules which have fired. These vectors are then used as the columns of a new tensor product, $\mathbf{TP_{r1}}$. As can be seen in Figure 4.4, each token vector in $\mathbf{TP_{in1}}$ has been replaced by a vector containing the rules which fired due to that token, and hence each rule vector exists twice in the intermediary tensor product (because the weight of a rule vector is 2). Each column which contained only zero elements in the input tensor product similarly contains only zero elements in this intermediary tensor product.

The next stage of a recall iteration is to find the consequents of any rules which have

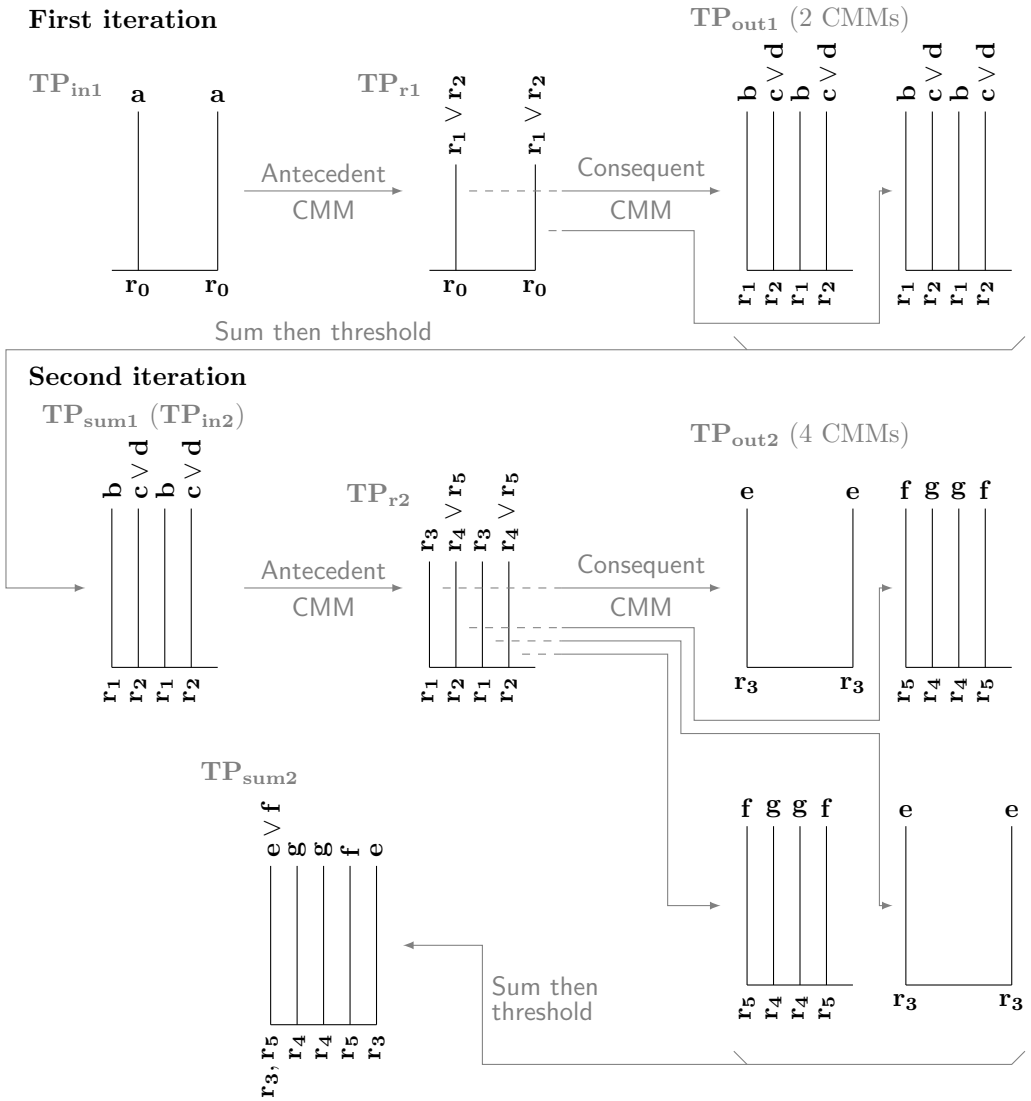


Figure 4.4: A visualisation of two iterations of the rule chaining process within ARCA. The process is initialised by creating a tensor product, $\mathbf{TP_{in1}}$, binding the input tokens (in this case $\mathbf{a}$) to a rule vector ($\mathbf{r_0}$). Each column of this tensor product is then recalled in turn from the antecedent CMM, to form a new tensor product, $\mathbf{TP_{r1}}$, containing the rules which have fired. Each column of $\mathbf{TP_{r1}}$ can then be recalled in turn from the consequent CMM, resulting in a number of output tensor products ($\mathbf{TP_{out1}}$—one tensor product for every non-zero column of $\mathbf{TP_{r1}}$). These output tensor products can be summed to form a non-binary CMM, before a threshold is applied using a value equal to the weight of a rule vector and resulting in $\mathbf{TP_{sum1}}$. The second iteration continues in the same fashion, using $\mathbf{TP_{sum1}}$ as the new input tensor product.

been matched. To do this, we must recall each column of $\mathbf{TP_{r1}}$ in turn from the consequent CMM. In this case, however, the result of recalling a vector is an entire flattened tensor product containing token vectors bound to rule vectors. As we know the width and height of the desired tensor product, we can reform each recalled vector to recover a series of tensor products, $\mathbf{TP_{out1}}$. As the weight of a rule vector in this example is set to two, every rule vector existed twice within $\mathbf{TP_{r1}}$ and so we know that every output token vector will be found bound to a particular rule within two of the output tensor products.

To complete this first iteration of our recall, we must now combine the output tensor products $\mathbf{TP_{out1}}$ into a single tensor product ready to be used again as an input. Each of the output tensor products has the same dimensions as an entire state, and so we have two possible methods of combining them: either simply superimposing them using a logical OR, or summing them to form a non-binary tensor product and then applying a threshold. The latter method allows us to exploit the knowledge that each expected output token will be repeated within a number of tensor products equal to the weight of a rule vector—we can use this as our threshold. This method has been shown to provide superior capacity, as any erroneously recalled bits that do not reach this threshold will be removed from the final output [17]. As such, in our example we sum the $\mathbf{TP_{out1}}$ tensor products and apply a threshold using Willshaw's method [112] and a value of 2.

Having obtained a single tensor product, $\mathbf{TP_{sum1}}$, we must now check whether the search has completed. Firstly we can check whether any rules have been matched, and hence whether the search should continue. If $\mathbf{TP_{sum1}}$ consists only of zeros then we know it cannot contain any token vectors, and hence the search has completed in failure.

If, on the other hand, $\mathbf{TP_{sum1}}$ is not empty then we must check whether our goal state has been reached. To achieve this we consider $\mathbf{TP_{sum1}}$ as a CMM. We can superimpose our goal token vectors, and recall this from $\mathbf{TP_{sum1}}$ using a threshold equal to the weight of these superimposed token vectors. If the result of this recall contains a rule vector then we know that the goal tokens were bound to this rule, and we can conclude that the goal state has been reached. In the absence of any such rule, the system simply iterates.

In our worked example, it is clear that the tensor product $\mathbf{TP_{sum1}}$ is not empty, and similarly that recalling the vector $\mathbf{f}$ from this tensor product will not result in any rule being output. The second iteration is therefore started, using $\mathbf{TP_{sum1}}$ as our new input state—$\mathbf{TP_{in2}}$. The operation of the ARCA system continues in exactly the same fashion with each iteration, first recalling each column of the input state from the antecedent

CMM to form $\mathbf{TP_{r2}}$, before recalling each column of this intermediary tensor product to result in a series of output tensor products—$\mathbf{TP_{out2}}$. In this second iteration there are four non-zero columns in our input and intermediary tensor products, and so we recover four output tensor products.

To complete the example, we sum and then threshold the $\mathbf{TP_{out2}}$ tensor products, and are left with a single output state—$\mathbf{TP_{sum2}}$. In this case, recalling the token vector $\mathbf{f}$ from our output state will result in a vector containing the rule $\mathbf{r_5}$. Thus we know that our goal state has been reached, as well as which rule fired—as it is possible that a token may appear as a consequent more than once in a particular set of rules.

4.3.3 Experimentation

The capacity of an ARCA network is very difficult to analyse, due to the operation of CMMs and the large number of parameters that using CMMs introduces. As CMMs become saturated, their outputs contain an increasing number of incorrectly set bits. Determining the point at which a CMM becomes saturated is difficult as it depends on the vector lengths and weights used, as well as the vector generation algorithm used. Adding to this difficulty, the ARCA network may continue to operate correctly even when the CMMs may individually be considered as saturated. This is largely due to the sum-then-threshold operation performed at the end of each iteration described in Section 4.3.2. This operation helps to clean some of the unwanted noise from the output, as it filters out any incorrectly set bits that do not reach the required threshold. Because of all these challenges, the most reliable way to examine the performance of ARCA is through experimentation to determine the capacity under different conditions.

Experimentation by Hobson [51] demonstrated that ARCA is able to search multiple branches of a tree of rules simultaneously, while maintaining separation between each of the branches. As the branches are searched simultaneously, the time complexity of the search is reduced from $O(b^d)$ to $O(d)$, where b is the branching factor and d is the depth of the tree. A simple implementation in MATLAB was used for these experiments, which put significant limitations on the experiments due to high memory usage and experimental execution time—although the number of steps in the search is reduced, when CMMs are implemented using a classical computing architecture each step of ARCA takes longer than those of a traditional tree search.

Although the results demonstrate ARCA works as expected, the contour plots pre-

senting these results might be somewhat misleading. With a branching factor of 1 the memory required by ARCA grows at a rate that is almost linear with respect to the depth of the tree. With higher maximum branching factors (where each node has between 1 and *max* children inclusively), the plots show that the memory required by ARCA grows exponentially with respect to the depth of the tree. While this is indeed the case, it masks the more important question of how the memory required by ARCA grows with respect to the number of rules. Presenting the results in this fashion would also allow a comparison between the memory required by ARCA when storing the same number of rules but with different maximum branching factors in the tree of rules.

Further than this, the decision to use a vector weight of $\lfloor \log_2 l \rfloor$ (where l is the vector length) rather than a fixed weight for all vector lengths adds complexity to the results. The results presented by Hobson [51] with a branching factor of 1 demonstrate this—there are a number of areas in the graph where the memory required increases while the number of rules stored actually decreases, due to a sudden increase in vector weight.

In order to better understand the relationship between the number of rules stored and the memory required, further experiments have been performed. To ensure that the results may be compared with those previously obtained, the experimental design remains very similar to that presented in [9]. ARCA has been implemented using the ENAMeL interpreter [21], and applied to a variety of problems. For each experiment a tree of rules was generated with a given depth d and maximum branching factor b. These rules were then learned by the system, and rule chaining was performed on them. Appendix A.2 contains a sample of the ENAMeL code used to run an experiment.

Search trees were constructed in an iterative manner, beginning with the root token, which was also used as the starting token for the recall process. Further layers of the tree were then added, with the total number of layers being d. In the simple case of $b = 1$, this produces a chain of rules $\mathbf{a} \rightarrow \mathbf{b}$, $\mathbf{b} \rightarrow \mathbf{c}$, etc. In the case of $b > 1$, the number of children of a given node was uniformly randomly sampled from the range $[1, b]$. This results in a tree with maximum branching factor b which is more realistic than one in which all nodes have exactly b children.

The experiments have been performed over a range of values for d, b, and the memory required to implement the CMMs in ARCA. Although ENAMeL has been used, its ability to use a sparse representation has not been used in order that a direct comparison to previous results can be performed. When a sparse representation is not used, the memory

required E is determined by the token and rule vector lengths—l_t and l_r respectively—as shown in Equation 4.1.

$$E = l_t l_r + l_t {l_r}^2 \text{ bits} \tag{4.1}$$

In order to simplify the experiments, the length of both token and rule vectors were set equally as l. Additionally, to aid interpretation of the results, the weight of all vectors has been set to 4. This value gives a sparse representation over the entire range of vector lengths used, and should provide good performance in the CMMs [81]. For each value of d, b, and l, the following experiment was performed:

1. Generate a rule tree with depth d and maximum branching factor b.
2. Train ARCA with the generated rules, with codes being generated by Baum's algorithm [13]. Baum codes are assigned to each token and rule randomly.
3. Take the root of the rule tree as the starting token, and select 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. Repeat the previous steps 100 times.

This gives a success rate for recall in ARCA, for a given combination of d, b, and l. A recall is defined as successful if and only if the goal token was found at the correct depth (i.e. after d iterations of the system). In unsuccessful recall, the system does not always fail in a traditional manner. Often the system will arrive at the desired goal erroneously, due to saturation of the CMMs causing the recall of extra patterns that were not trained—"ghosts". In these cases, the recall was deemed to have been unsuccessful.

4.3.4 Results

The graphs in Figure 4.6 are contour plots showing the recall success rates for the ARCA architecture for a given number of rules and memory requirement (related to the vector length). Each contour designates the boundary between recall success ranges—for example any point in the area above and to the left of the 99 contour represents at least 99% recall accuracy, any point below and to the right represents less than 99%.

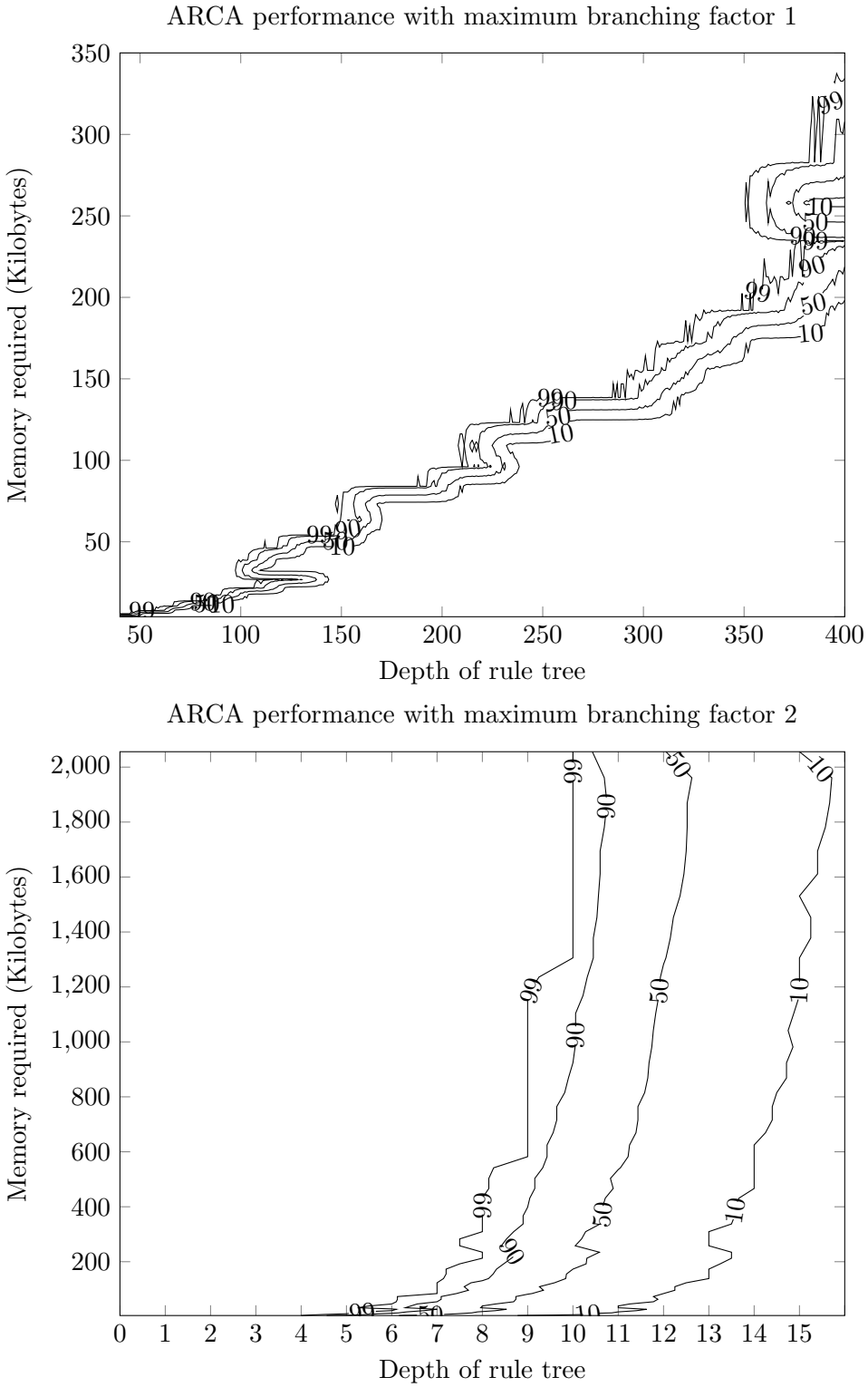


Figure 4.5: Contour plots showing the recall performance of ARCA, where the branching factor is 1 (top) and 2 (bottom). Contours summarise the recall accuracy with various depths of the rule tree and with various memory requirements, effectively showing the percentage of recalls which are successful for a given size of CMM and depth of tree.

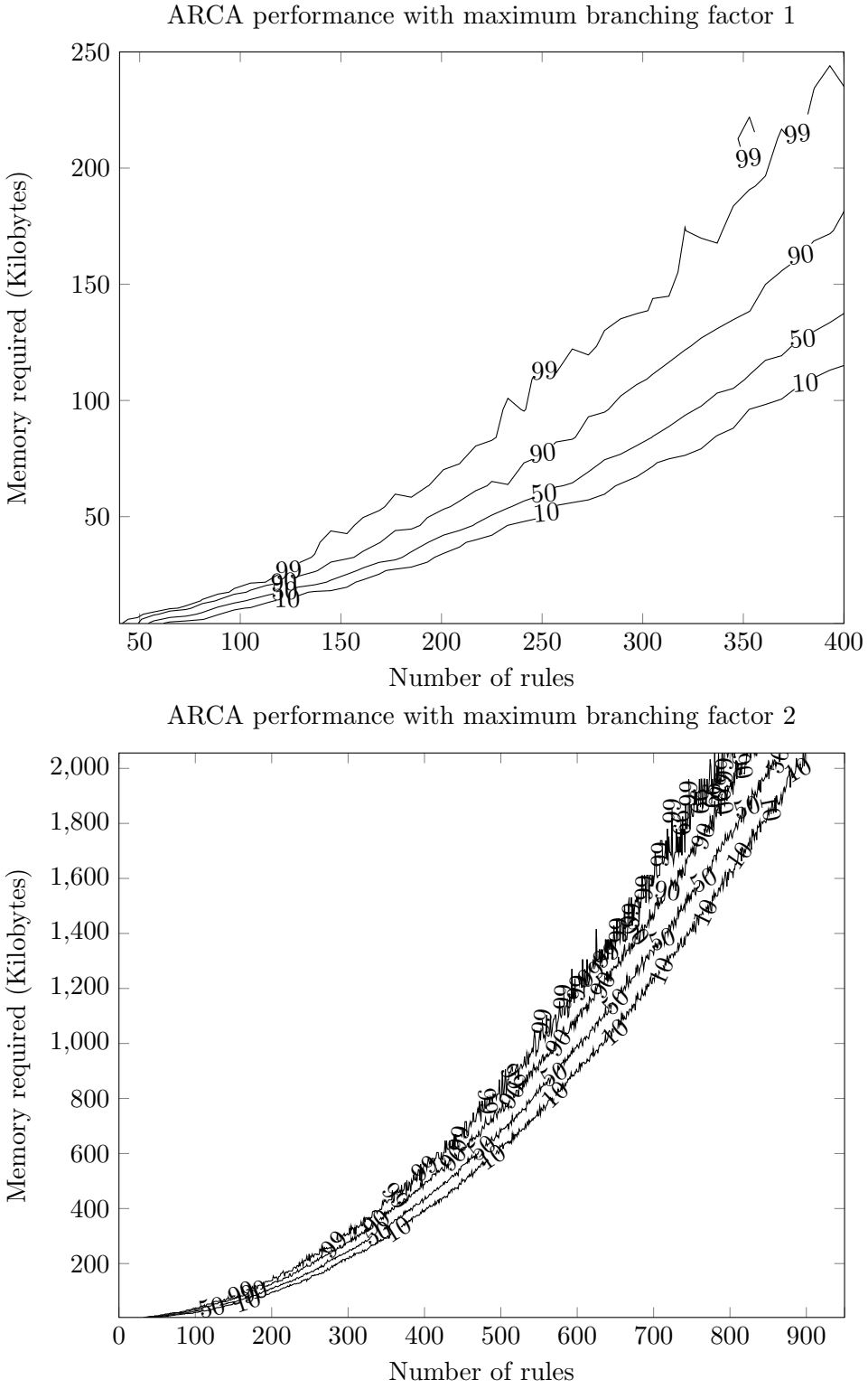


Figure 4.6: Contour plots showing the recall performance of ARCA, where the branching factor is 1 (top) and 2 (bottom). Contours summarise the recall accuracy with various numbers of rules and with various memory requirements, effectively showing the percentage of recalls which are successful for a given size of CMM and number of rules.

The contour plot with maximum branching factor 1 shows that the memory required if ARCA is to achieve correct recall grows slightly faster than linearly when using non-sparse storage. The reason for this will be explored in Section 4.4. As expected the graph is significantly clearer to read than in the previously presented results [51] (shown in Figure 4.5), due to the use of a fixed weight of 4 rather than setting the weight as $\log_2 l$, where l is the vector length.

The results for a maximum branching factor of 2 show that even when the memory requirement is plotted against the number of rules, rather than against the depth of the tree, the performance begins to decrease significantly. This means that as the branching factor increases, so too does the memory requirement—even when storing the same number of rules. This is not explained by the state explosion that occurs when increasing the branching factor, as that is countered by plotting the number of rules rather than the depth of the tree. As such, there are two limitations in this case.

Firstly the capacity of the input and intermediary tensor products used in ARCA—$\mathbf{TP_{in1}}$ and $\mathbf{TP_{r1}}$ in Figure 4.4. This limitation is affected by the relative lengths of the token and rule vectors explored in Section 4.4, as varying these affects the size and storage capacity of the tensor products. Secondly, due to the nature of the training process the antecedent CMM is significantly smaller than the consequent CMM. This means that while the antecedent CMM may become saturated, the consequent CMM remains sparse—and so the memory is inefficiently utilised. A resolution for this will be presented in Section 4.4.

The graphs in Figures 4.7 and 4.8 are scatter plots showing the same results as the contour plots—the memory requirement for the ARCA architecture for a given number of rules and recall success rate. Each point represents a successful recall in at least 10% of the experimental runs, with the colour denoting the range from at least 10% to at least 99% recall success rate. When the data is represented in this format, it is clearer to see the relationship between the length of vectors and the memory required. With a non-sparse memory storage this relationship is fixed, as given in Equation 4.1, and clearly increases at a rate that is faster than linear with respect to the increasing length of vectors.

Although the performance decreases significantly between branching factors 1 and 2, it is interesting to note that the results are very similar for a branching factor of 2, 3, or 4. This indicates that although moving from a list of rules (branching factor 1) to a tree (branching factor 2) results in a memory increase, increasing the branching factor of that tree has minimal effect. In fact the graphs appear to indicate that as the branching factor

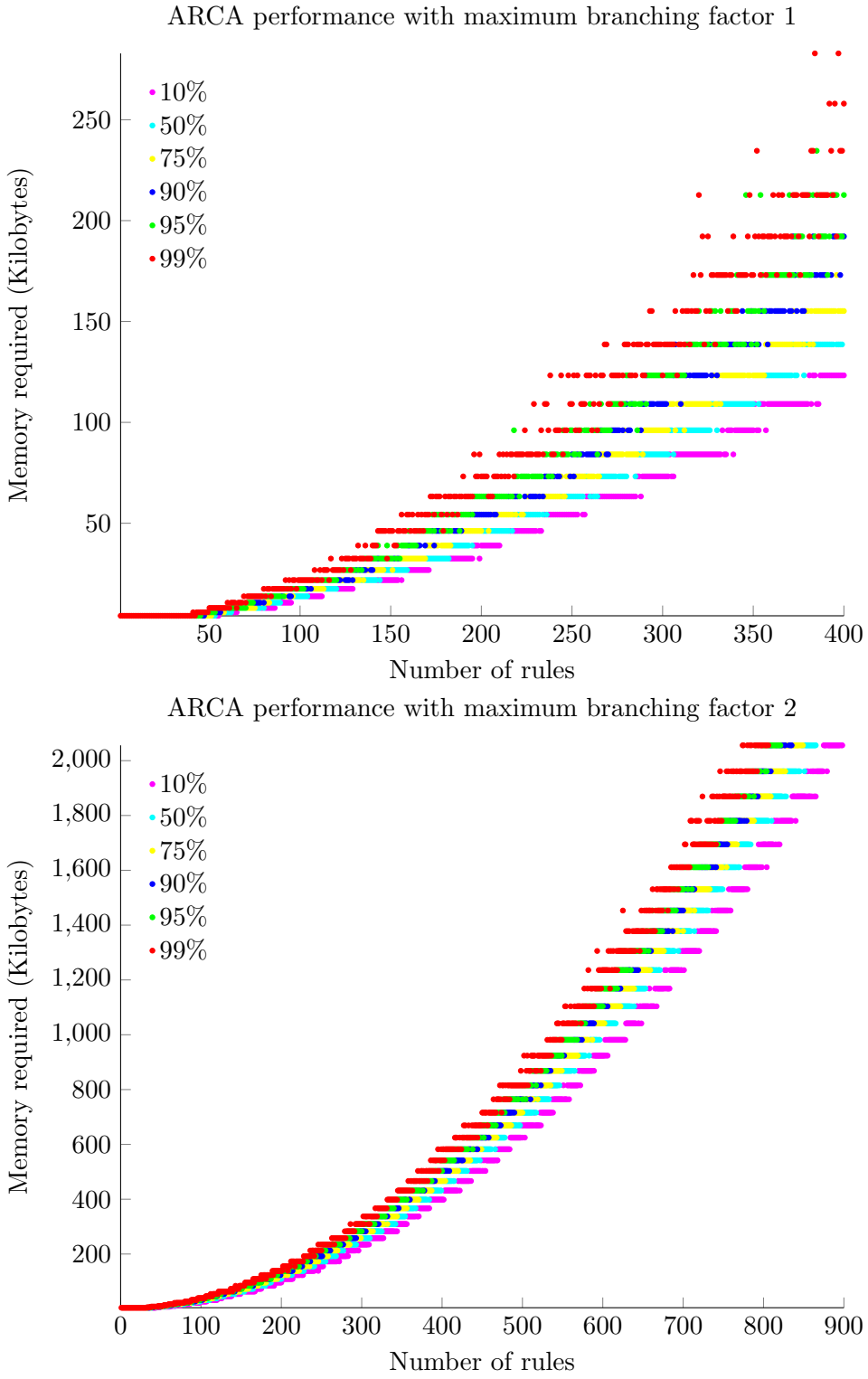


Figure 4.7: Scatter plots showing the recall performance of ARCA, where the branching factor is 1 (top) and 2 (bottom). Each point represents a successful recall in at least 10% of the experimental runs, with the colour denoting the range from at least 10% to at least 99% recall success rate.

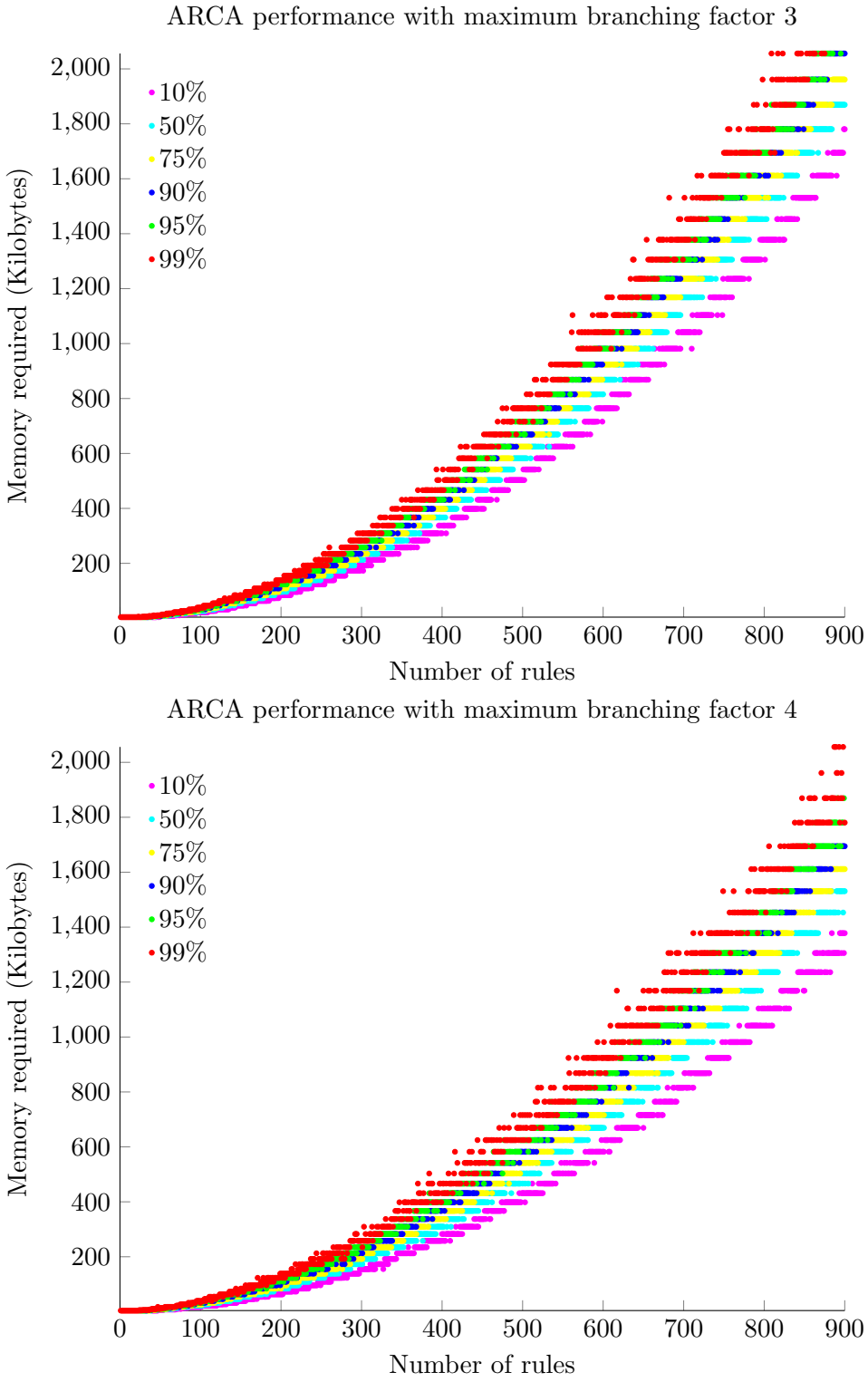


Figure 4.8: Scatter plots showing the recall performance of ARCA, where the branching factor is 3 (top) and 4 (bottom). Each point represents a successful recall in at least 10% of the experimental runs, with the colour denoting the range from at least 10% to at least 99% recall success rate.

increases from 2, the memory requirement is reduced. This is caused by the number of iterations decreasing as the branching factor increases. A tree with 500 rules, for example, contained an average of 13.16 levels with a branching factor of 2, compared to an average of 8.08 levels with a branching factor of 3. Reducing the number of iterations improves the recall reliability of the system, because the number of erroneously recalled "ghosts" increases with every iteration until a failure occurs.

Finally, it is significant that there is only a very small difference between the memory required for 100% success and that for 0%—especially with a branching factor greater than 1. We saw in Section 2.3.4 that CMMs usually degrade gradually when reaching saturation, which indicates that this is not the sole cause of ARCA failing. As mentioned earlier, given the structure of ARCA, the size of the antecedent CMM is much smaller than that of the consequent CMM. As such, the antecedent CMM will approach saturation much faster than the consequent CMM, and so the input to this second CMM becomes too noisy to result in an accurate recall. Additionally, the tensor products used during recall are a similar size to the antecedent CMM. These will also become saturated, as the number of simultaneous branches of the search increases quickly when the branching factor is greater than 1.

4.3.5 Using a Sparse Matrix Representation

ENAMeL has the facility to measure the memory required when storing the matrices using a sparse representation. The memory required to implement ARCA with sparse storage is more difficult to calculate as it is dependent on the number of set bits within the matrices, which in turn is dependent on a large number of variables. When using the binary Yale format, an upper bound for the number of set bits, S, can be calculated as shown in Equations 4.2–4.4, where n is the number of rules stored, and w_t and w_r are the respective weights of token and rule vectors.

$$S_1 = nw_t w_r \tag{4.2}$$

$$S_2 = nw_t {w_r}^2 \tag{4.3}$$

$$S = S_1 + S_2 \tag{4.4}$$

Given these upper bounds on the number of set bits within each CMM, we can then calculate upper bounds on the memory required, E_{sparse}. The memory required to store

a single bit in the binary Yale format is dependent on the vector lengths used. The binary Yale format uses two arrays, IA and JA. IA denotes the beginning of each row of the matrix as an index into the JA array, and as such must be able to store a value equal to the maximum number of elements in the JA array. This is calculated simply by multiplying the lengths of the input and output vectors to the matrix. JA, on the other hand, stores only column indices of any bits set to `1`, and as such must be able to store a value equal to the number of columns (minus one as zero-based indexing is used)—or the length of the output vector. This was discussed in more detail in Section 3.3.1.

Combining Equation 3.4 with the previous equations for S_1 and S_2, gives us the very conservative Equations 4.5–4.7 for the maximum memory requirements, where l_t and l_r are the respective lengths of token and rule vectors.

$$E_{\text{sparse1}} = (l_t + 1)2^{\lceil \log_2 \lceil \log_2 (l_t l_r + 1) \rceil \rceil} + (n w_t w_r) 2^{\lceil \log_2 \lceil \log_2 (l_r) \rceil \rceil} \tag{4.5}$$

$$E_{\text{sparse2}} = (l_r + 1)2^{\lceil \log_2 \lceil \log_2 (l_t {l_r}^2 + 1) \rceil \rceil} + (n w_t {w_r}^2) 2^{\lceil \log_2 \lceil \log_2 (l_t l_r) \rceil \rceil} \tag{4.6}$$

$$E_{\text{sparse}} = E_{\text{sparse1}} + E_{\text{sparse2}} \tag{4.7}$$

The actual number of set bits and memory required is likely to be lower than these upper bounds, however, due to overlaps between the associations stored. The number of overlapping bits will depend on the level of saturation of each CMM, which is affected by the vector lengths and weights selected, as well as the vector generation algorithm used.

The hybrid storage format used by ENAMeL is harder again to analyse, as the memory required is dependent on the number of set bits within each row of the matrices. Each matrix has an overhead of one pointer for each row—typically using 32-bit pointers, although 64-bit pointers can be used if the data size exceeds the 32-bit limit. The individual rows are then stored using either a non-sparse binary format, or a sparse format similar to the binary Yale format—in this case requiring only the JA array for each row. This variability means that the maximum memory requirement of each row is limited to l_{out} bits, where l_{out} is the output vector length, plus the fixed pointer overhead. It is unlikely that these upper bounds will be reached under normal operation of the system—these bounds will only take effect as the CMMs reach a certain level of saturation, at which point the network is liable to fail anyway.

As an extension to the results presented in Section 4.3.4, and to demonstrate the suitability and potential scalability of ENAMeL for practical applications, the previous

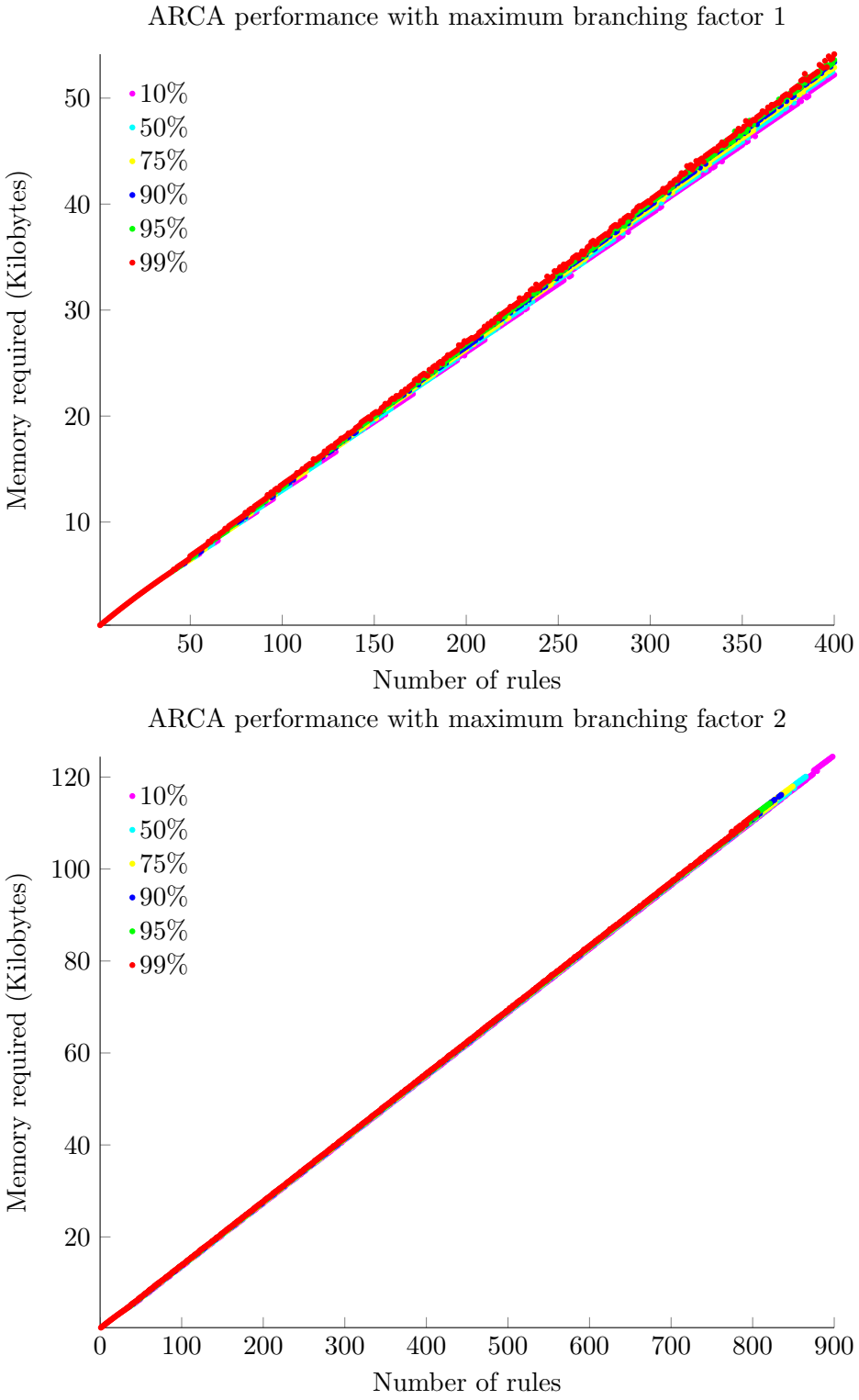


Figure 4.9: Scatter plots showing the recall performance of ARCA, where the branching factor is 1 (top) and 2 (bottom). Each point represents a successful recall in at least 10% of the experimental runs, with the colour denoting the range from at least 10% to at least 99% recall success rate. The memory required is calculated using the binary Yale format.

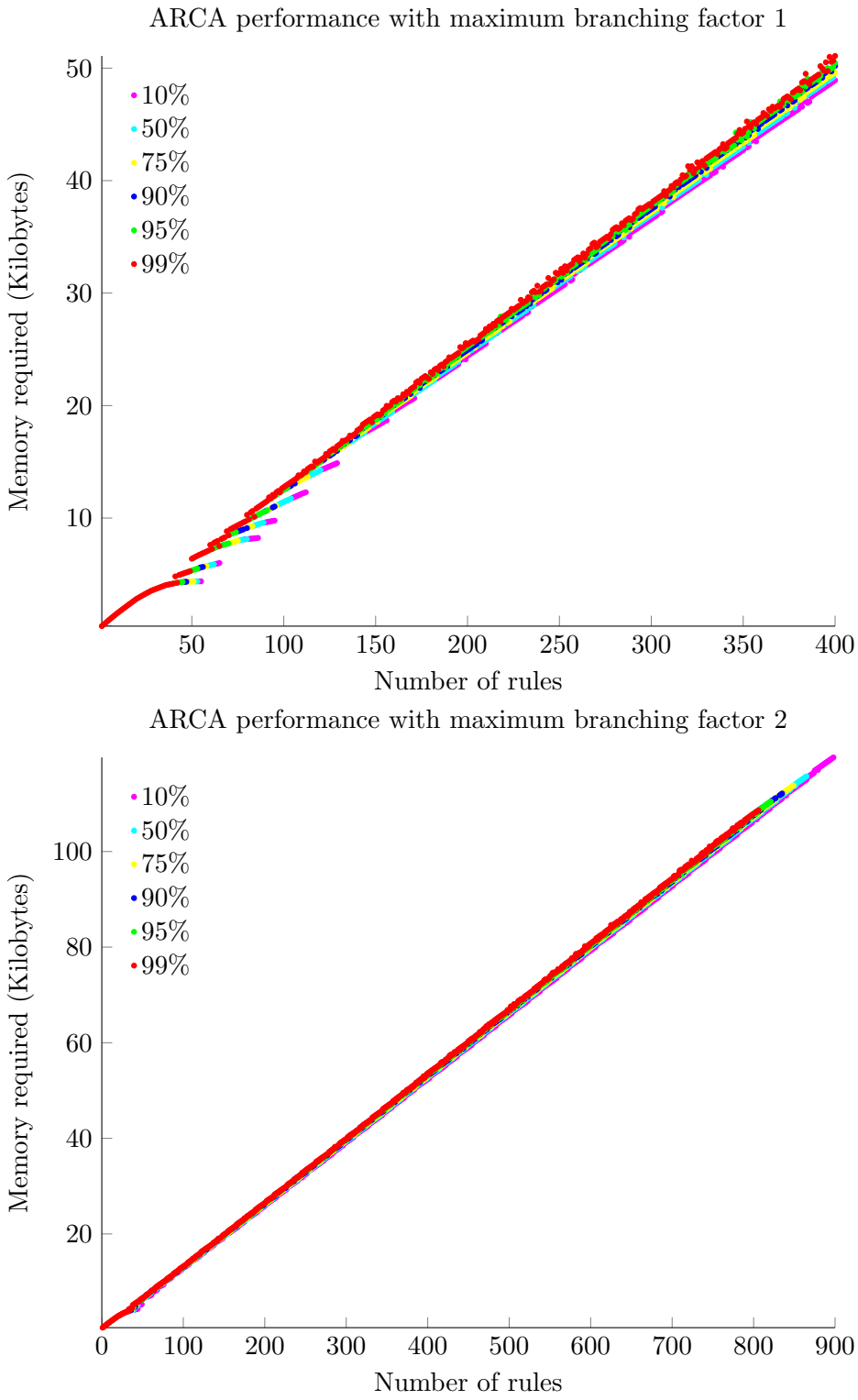


Figure 4.10: Scatter plots showing the recall performance of ARCA, where the branching factor is 1 (top) and 2 (bottom). Each point represents a successful recall in at least 10% of the experimental runs, with the colour denoting the range from at least 10% to at least 99% recall success rate. The memory required is calculated using the hybrid storage format.

experiments have been analysed further—in this instance recording the memory required when using both the binary Yale format and ENAMeL's hybrid storage. The graphs in Figures 4.9 and 4.10 are scatter plots showing the memory requirement for the ARCA architecture for a given number of rules and recall success rate, when the CMMs are stored using the binary Yale format (Figure 4.9) and the hybrid storage format (Figure 4.10). Each point represents a successful recall in at least 10% of the experimental runs, with the colour denoting the range from at least 10% to at least 99% recall success rate.

It is clear to see that the use of sparse storage has significantly reduced the memory requirement of ARCA when compared to Figure 4.7. More importantly, there is now a linear relationship between the number of rules trained and the memory required. This is to be expected—as we saw in Section 3.3, the use of sparse storage allows the size of a CMM to grow linearly with respect to the number of vector pairs associated within it.

The hybrid storage format requires fractionally less memory than the binary Yale format due to the saturation of the antecedent CMM. As the CMM becomes saturated it requires less memory to store non-sparsely, leading to the curves that can be seen in the left-hand part of the top of Figure 4.10. The consequent CMM remains sparsely populated, as it has a far greater size, and so this continues to be most efficiently stored in a sparse format.

Finally, it is notable that the memory required when the branching factor is greater than 1 is very similar to the requirement when the branching factor is equal to 1 when storing the same number of rules. Although the vectors are still required to be longer, when a sparse storage format is used this imposes only a small penalty by reducing the number of vector overlaps within the CMM.

4.4 Reducing the Memory Requirements

The original architecture, presented in Section 4.3, used two CMMs to separate the antecedents and consequents of rules. When storing a rule, for example $\mathbf{a} \rightarrow \mathbf{b}$, a unique "rule vector" must be generated. This is a label for the rule, $\mathbf{a} \xrightarrow{\mathbf{r_0}} \mathbf{b}$, but can be considered as essentially replacing the original rule with one containing an additional intermediary step $\mathbf{a} \rightarrow \mathbf{r_0} \rightarrow (\mathbf{b} : \mathbf{r_0})$. The antecedent CMM then stores the first half of this $\mathbf{a} \rightarrow \mathbf{r_0}$, and the consequent CMM stores the rest $\mathbf{r_0} \rightarrow (\mathbf{b} : \mathbf{r_0})$.

The original architecture used two CMMs to try and improve performance; in ARCA the separation of superimposed states during the recall process is actually performed by

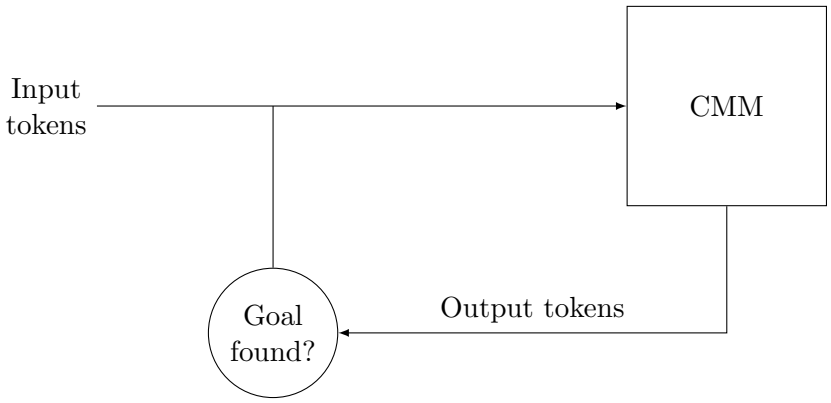


Figure 4.11: Block diagram of the single CMM ARCA

binding consequents to the rule vector prior to training them into the consequent CMM, rather than through the use of two CMMs. As such, the use of two CMMs is unnecessary (except in certain specific cases indicated below) and the ARCA system can be simplified to use a single CMM mapping directly from the antecedents to the consequents as shown in Figure 4.11. This can help by reducing both the memory required to store the rules, and the time required to perform each iteration of a recall operation [19].

It is important to note that using a single CMM may not reduce the memory requirements when a non-sparse matrix storage is used. As we saw in Equation 4.1, the memory required for ARCA using two CMMs and non-sparse storage is determined by the token and rule vector lengths—l_t and l_r respectively. The memory required for ARCA using only a single CMM with non-sparse storage E_{single} is similarly dependent on these variables, as shown in Equation 4.8:

$$E_{\text{single}} = l_r {l_t}^2 \text{ bits} \tag{4.8}$$

We can then determine a relationship between Equations 4.1 and 4.8, such that the memory required for each will be equal when Equation 4.9 holds. When token and rule vectors have the same length, the memory required should be essentially the same. If token vectors are longer than rule vectors then ARCA using a single CMM would be expected to require more memory, and if token vectors are shorter than rule vectors then ARCA using a single CMM should require less memory.

$$\begin{aligned} l_r {l_t}^2 &= l_t l_r + l_t {l_r}^2 \\ l_t &= 1 + l_r \end{aligned} \tag{4.9}$$

This is not a concern in the context of software implementations of CMMs such as

ENAMeL, as it is typical to use sparse storage for the matrices. It should remain a consideration if a CMM implementation is used that does not utilise sparse storage, such as a RAM-based hardware implementation.

4.4.1 Training

To train this simplified ARCA requires a similar operation as originally used when training the consequent CMM of the original architecture. Every rule is still assigned a unique rule vector, and the superimposed consequents of the rule are bound to this rule vector $\mathbf{b} : \mathbf{r_0}$. The single CMM is now trained using the superimposed antecedents of the rule as an input, and this tensor product as an output $\mathbf{a} \rightarrow (\mathbf{b} : \mathbf{r_0})$. Upon presentation of an input containing the antecedent tokens of a rule, the CMM will produce a tensor product containing the consequent tokens bound to the rule that caused them to fire.

4.4.2 Recall

The recall process is similar to that of the original architecture, without the intermediate steps caused by having two CMMs. It is best described with the aid of a diagram, so Figure 4.12 shows two iterations of a recall process performed on the example used in Section 4.3. As with the original example, we wish to perform forward chaining on the rules given in Figure 4.1 using **a** as our initial state and **f** as the goal state.

We must firstly create an input state, containing the tokens with which we are starting our search. A state is simply a tensor product containing one or more tokens bound to rules, and so we bind our input token vector **a** to a "dummy" rule $\mathbf{r_0}$ to form $\mathbf{TP_{in1}}$. Each input token vector will exist in this tensor product a number of times equal to the fixed weight of a rule vector, in our example this is twice. The choice of which rule vector with which to bind our initial input token vectors does not have any effect on the recall process, as the rule vectors serve only to maintain separation between multiple superimposed branches of a search.

The first stage of recall is to find the consequents of any rules which are matched by the tokens contained within our current state. Each column of $\mathbf{TP_{in1}}$ must be recalled in turn from the CMM. The result of recalling a vector is an entire flattened tensor product containing token vectors bound to rule vectors. As we know the width and height of the desired tensor product, we can reform each recalled vector to recover a series of tensor products, $\mathbf{TP_{out1}}$. As the weight of a rule vector in this example is set to two, every token

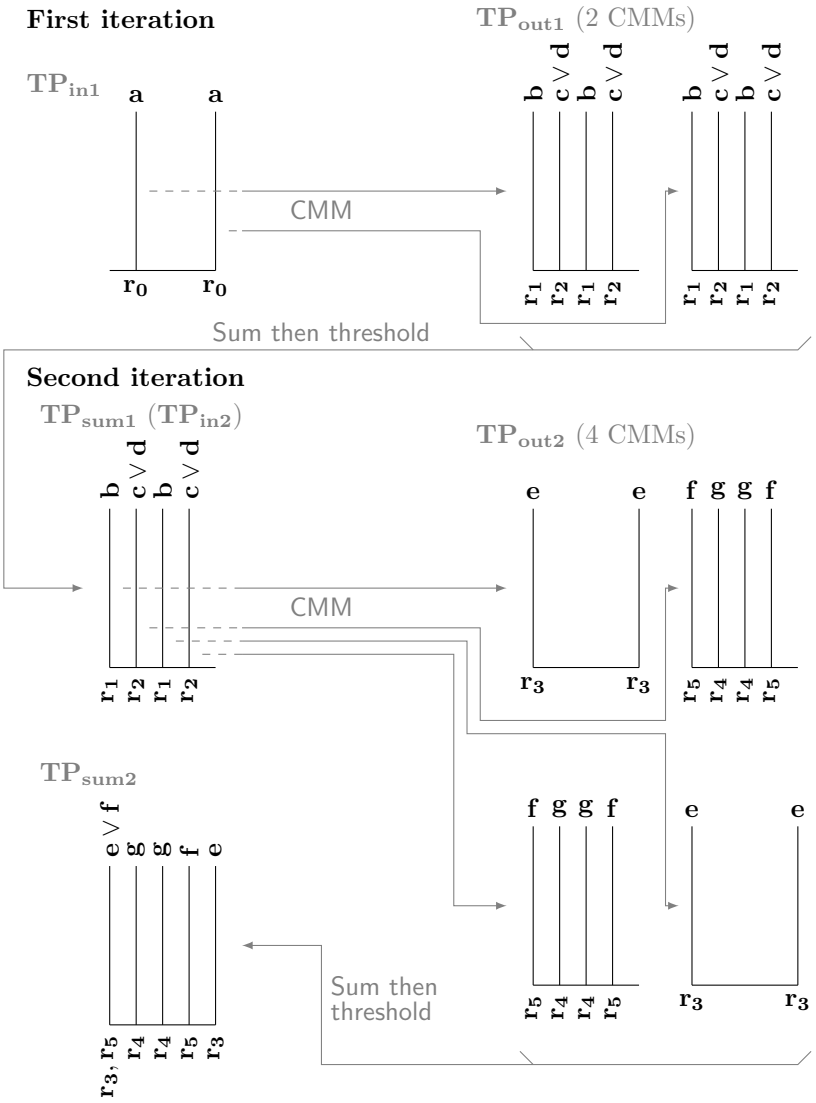


Figure 4.12: A visualisation of two iterations of the rule chaining process within the single CMM ARCA. The process is initialised by creating a tensor product, $\mathbf{TP_{in1}}$, binding the input tokens (in this case $\mathbf{a}$) to a rule vector ($\mathbf{r_0}$). Each column of this tensor product is then recalled in turn from the CMM, resulting in a number of output tensor products ($\mathbf{TP_{out1}}$—one tensor product for every non-zero column of $\mathbf{TP_{in1}}$). These output tensor products can be summed to form a non-binary CMM, before a threshold is applied using a value equal to the weight of a rule vector and resulting in $\mathbf{TP_{sum1}}$. The second iteration continues in the same fashion, using $\mathbf{TP_{sum1}}$ as the new input tensor product.

vector existed twice within $\mathbf{TP_{in1}}$ and so we know that every output token vector will be found bound to a particular rule within two of the output tensor products.

To complete this first iteration of our recall, we must now combine the output tensor products $\mathbf{TP_{out1}}$ into a single tensor product ready to be used again as an input. This is performed in the same way as with the original ARCA architecture. We sum the $\mathbf{TP_{out1}}$ tensor products and apply a threshold using Willshaw's method [112] and a value equal to the weight of a rule vector—in this case a value of 2.

Having obtained a single tensor product, $\mathbf{TP_{sum1}}$, we must now check whether the search has completed. Firstly we can check whether any rules have been matched, and hence whether the search should continue. If $\mathbf{TP_{sum1}}$ consists only of zeros then we know it cannot contain any token vectors, and hence the search has completed in failure.

If, on the other hand, $\mathbf{TP_{sum1}}$ is not empty then we must check whether our goal state has been reached. To achieve this we consider $\mathbf{TP_{sum1}}$ as a CMM. We can superimpose our goal token vectors, and recall this from $\mathbf{TP_{sum1}}$ using a threshold equal to the weight of these superimposed token vectors. If the result of this recall contains a rule vector then we know that the goal tokens were bound to this rule, and we can conclude that the goal state has been reached. In the absence of any such rule, the system simply iterates.

In our worked example, it is clear that the tensor product $\mathbf{TP_{sum1}}$ is not empty, and similarly that recalling the vector $\mathbf{f}$ from this tensor product will not result in any rule being output. The second iteration is therefore started, using $\mathbf{TP_{sum1}}$ as our new input state—$\mathbf{TP_{in2}}$. The operation of the single CMM ARCA system continues in exactly the same fashion with each iteration, recalling each column of the input state from the CMM to result in a series of output tensor products—$\mathbf{TP_{out2}}$. In this second iteration there are four non-zero columns in our input tensor products, and so we recover four output tensor products.

To complete the example, we sum and then threshold the $\mathbf{TP_{out2}}$ tensor products, and are left with a single output state—$\mathbf{TP_{sum2}}$. In this case, recalling the token vector $\mathbf{f}$ from our output state will result in a vector containing the rule $\mathbf{r_5}$. Thus we know that our goal state has been reached, as well as which rule fired.

4.4.3 Experimentation

In order to compare the memory requirements of the "single CMM" ARCA to those of the original architecture, both variants have been applied to a set of randomly generated

forward chaining problems. For each experiment a tree of rules was generated with a given depth d and maximum branching factor b, using the same procedure as detailed in Section 4.3.3. These rules were then learned by both systems, and rule chaining was performed on them.

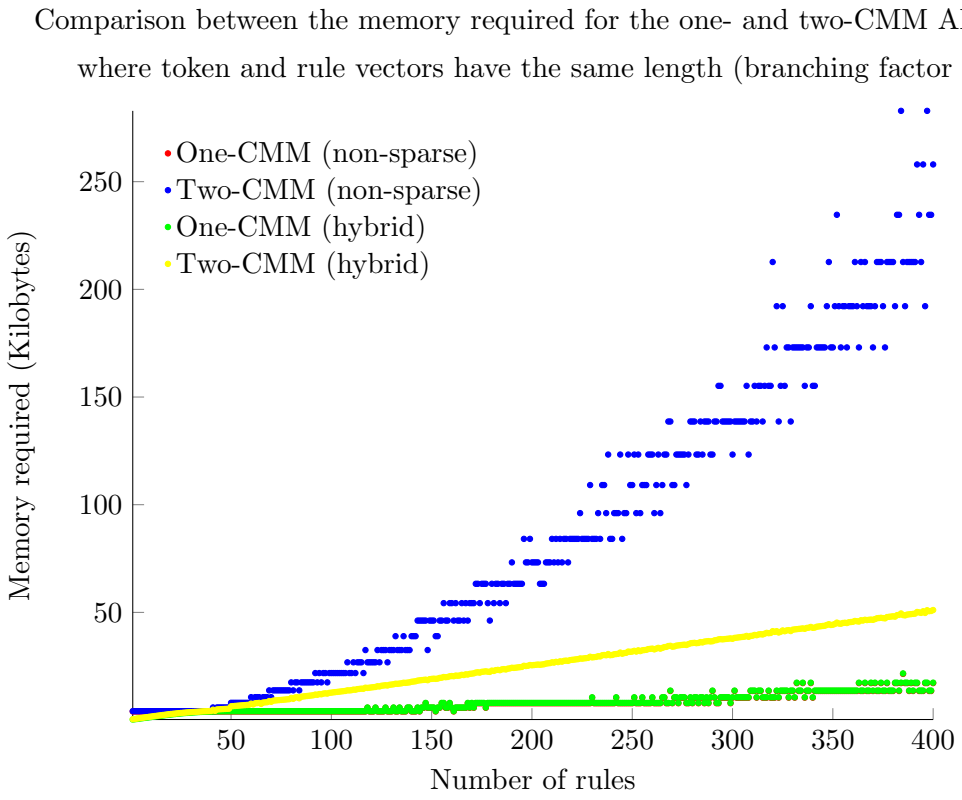
In all experiments, the weight of all vectors has been set to 4. This value gives a sparse representation over the entire range of vector lengths used, and should provide good performance in the CMMs [81]. In these experiments, the lengths of token and rule vectors have been varied independently, in order to determine whether the single CMM ARCA requires less memory under the various possible conditions.

The graphs in Figures 4.13 and 4.14 are scatter plots showing the minimum memory required for ARCA to successfully store and recall a given number of rules in at least 99% of the experimental runs. In these experiments, token and rule vectors have the same length—varied between 32 and 256 bits. The memory required has been calculated for two methods of storage—non-sparse and the hybrid format—when using both the one-CMM and the two-CMM variants of ARCA. The results do not include the binary Yale format as previous experiments have shown the hybrid format requires less memory.

It is unsurprising that in general the hybrid format requires the least memory, given that it is able to select either non-sparse or binary Yale for each individual row. When the branching factor is 1, however, it is interesting that the memory required for the one-CMM ARCA using non-sparse storage is the same as that required for the hybrid format. This shows that in this case, every row of the CMM is most efficiently stored non-sparsely. The results in Section 3.3.2 help to explain this—as a CMM reaches saturation it can require less memory when stored non-sparsely. In the two-CMM ARCA, the smaller antecedent CMM becomes saturated while the consequent CMM remains sparsely populated—in the one-CMM ARCA, this CMM is now able to be fully utilised.

When the branching factor is greater than 1 the memory required for the hybrid format is lower than for non-sparse storage, showing that the CMM is more efficiently stored using the binary Yale format and is not fully saturated at the point of failure. This limitation is caused by the capacity of the tensor products used during recall—they become saturated more quickly than the CMM as the number of superimposed states increases.

Most importantly all of these results show that when the token and rule vectors have the same length, the one-CMM variant of ARCA requires less memory than the two-CMM variant in order to successfully store the same number of rules. Due to equation 4.9 it was



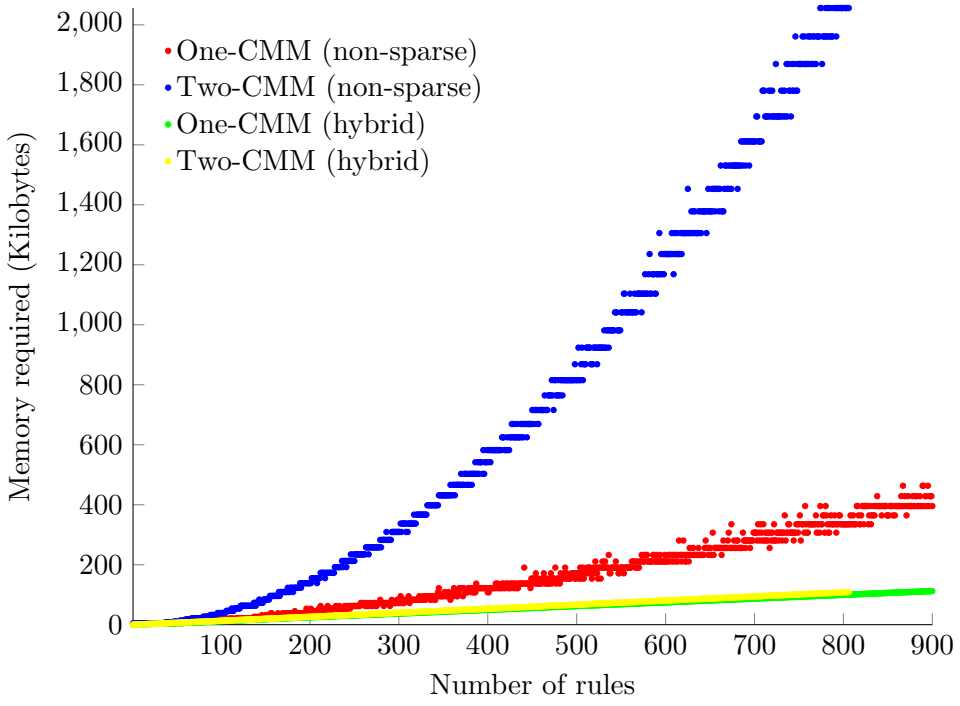
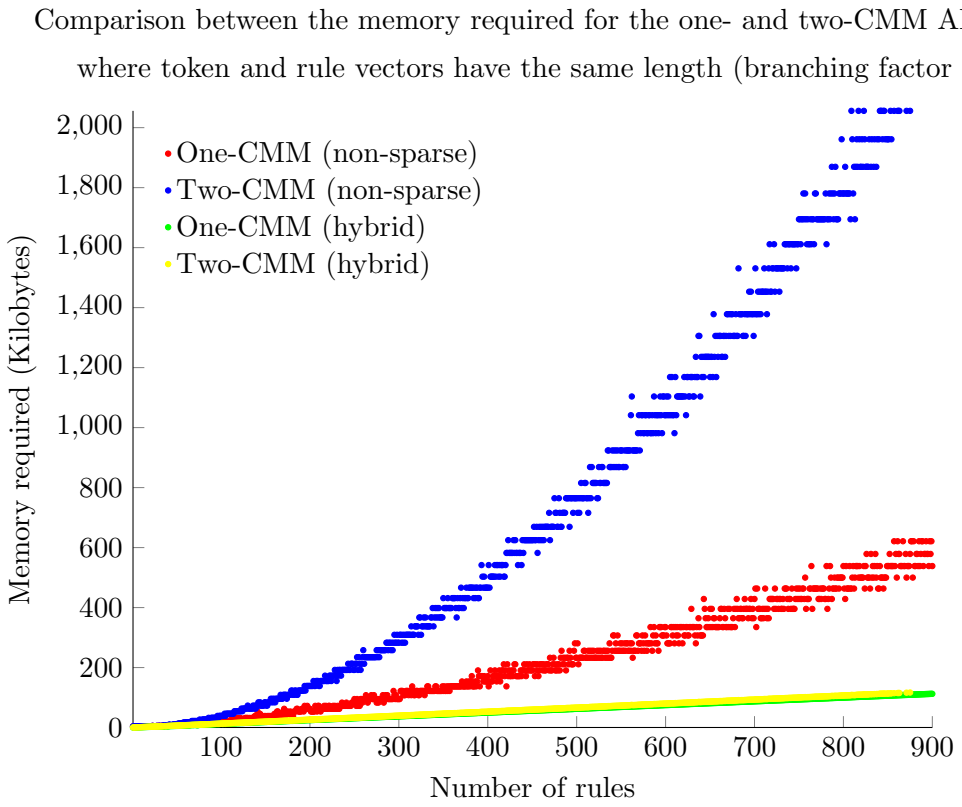


Figure 4.13: Scatter plots showing the memory requirements of ARCA, comparing the one- and two-CMM variants with various storage mechanisms, where the branching factor is 1 (top) and 2 (bottom). Each point represents the minimum memory required to result in a successful recall in at least 99% of the experimental runs. In these experiments, token and rule vectors have the same length.



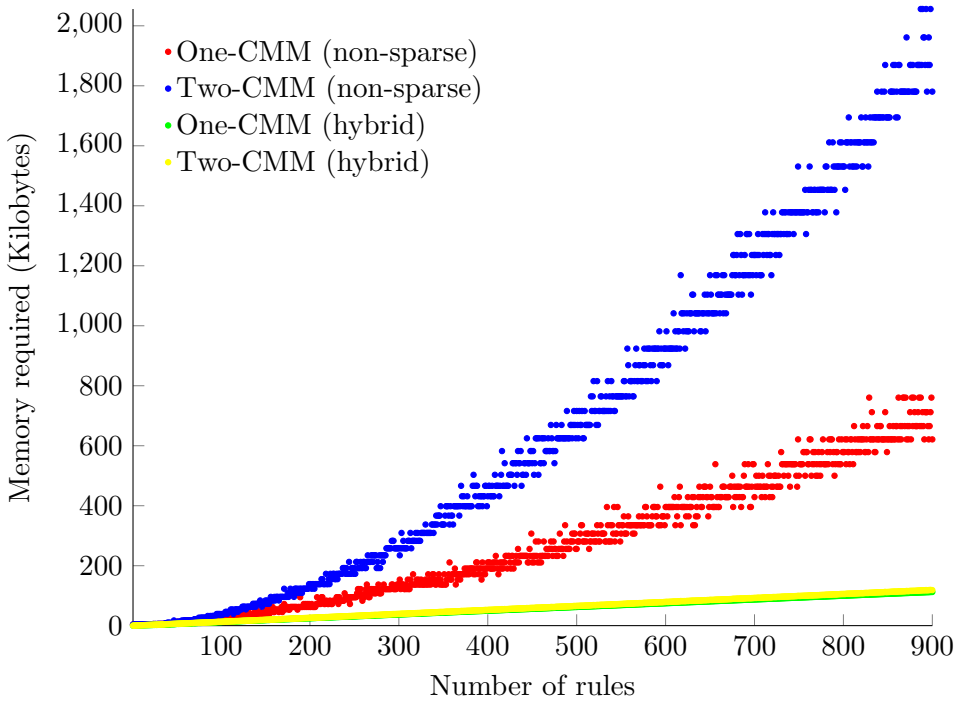


Figure 4.14: Scatter plots showing the memory requirements of ARCA, comparing the one- and two-CMM variants with various storage mechanisms, where the branching factor is 3 (top) and 4 (bottom). Each point represents the minimum memory required to result in a successful recall in at least 99% of the experimental runs. In these experiments, token and rule vectors have the same length.

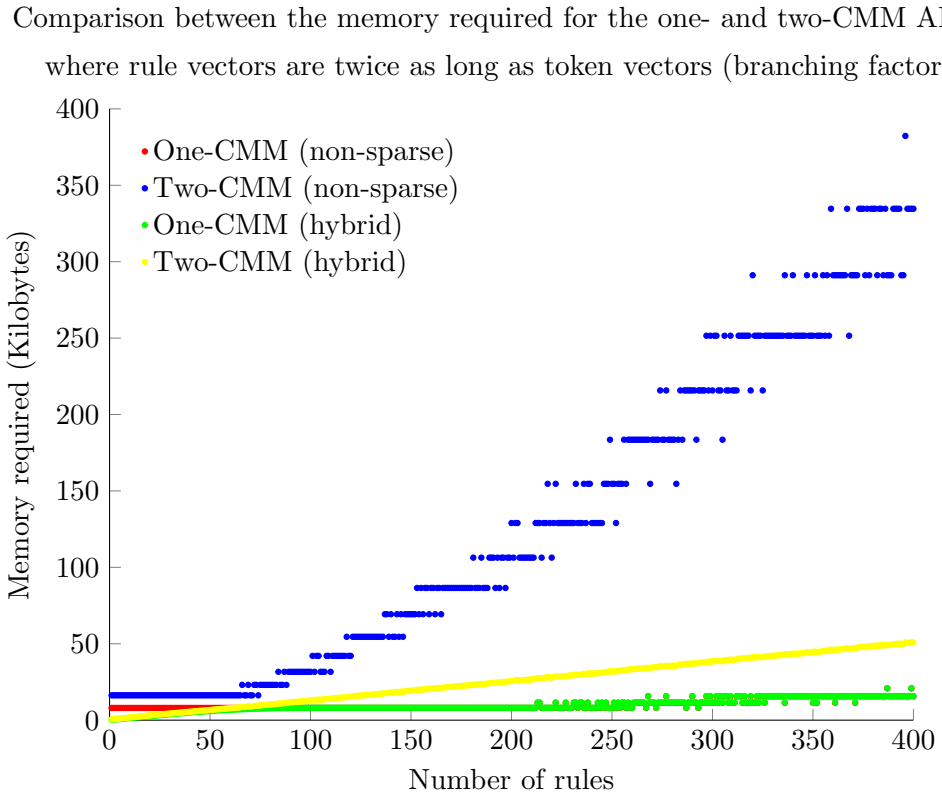
expected that when token and rule vectors have the same length, the memory required for both variants would be virtually identical when using non-sparse storage. However, this is under the assumption that both variants will be able to successfully store the same number of rules for a given vector length. The results for all branching factors demonstrate that this is not the case—the one-CMM variant of ARCA can successfully store significantly more rules, reducing the non-sparse memory growth from exponential to linear.

The graphs in Figures 4.15 and 4.16 are scatter plots showing the minimum memory required for ARCA to successfully store and recall a given number of rules in at least 99% of the experimental runs. At the top of each figure the rule vectors were twice as long as token vectors, and at the bottom this was reversed.

The results with a branching factor of 1, Figure 4.15, are exactly as would be expected. When rule vectors are twice as long as token vectors, the memory required by the two-CMM variant of ARCA using non-sparse storage is double that required when token vectors are twice as long as rule vectors. In Equation 4.1 we can see that the memory requirement in this case is dominated by the size of the consequent CMM ($l_t {l_r}^2$) and that doubling the length l_r will increase the memory requirement four-fold, compared to only two-fold when doubling l_t.

The one-CMM variant of ARCA shows the opposite effect when using non-sparse storage—when rule vectors are twice as long as token vectors, the memory required is half that required when token vectors are twice as long as rule vectors. Equation 4.8 shows that in this case the memory requirement ($l_r {l_t}^2$) is dominated by the token vector length. Doubling this length l_t will increase the memory requirement four-fold, compared to only two-fold when doubling l_r. As the number of rules trained approaches 300, it appears that the relationship breaks down as the memory requirement is the same in both graphs. The reason for this is due to the fact that the token vectors are used as the input vectors for the one-CMM ARCA. As we saw in Section 3.3.2, increasing the length of input vectors increases the capacity of a CMM faster than increasing the length of output vectors.

When the hybrid storage format is used, the results are similarly as expected. The memory required by the two-CMM variant of ARCA is virtually unaffected by the vector lengths, as it uses sparse storage which is most affected by vector weights. The one-CMM variant of ARCA does require more memory when token vectors are twice as long as rule vectors, rather than vice versa. This is due to the use of non-sparse storage—as previously discussed, the single CMM is able to be fully utilised and so it becomes more efficient to



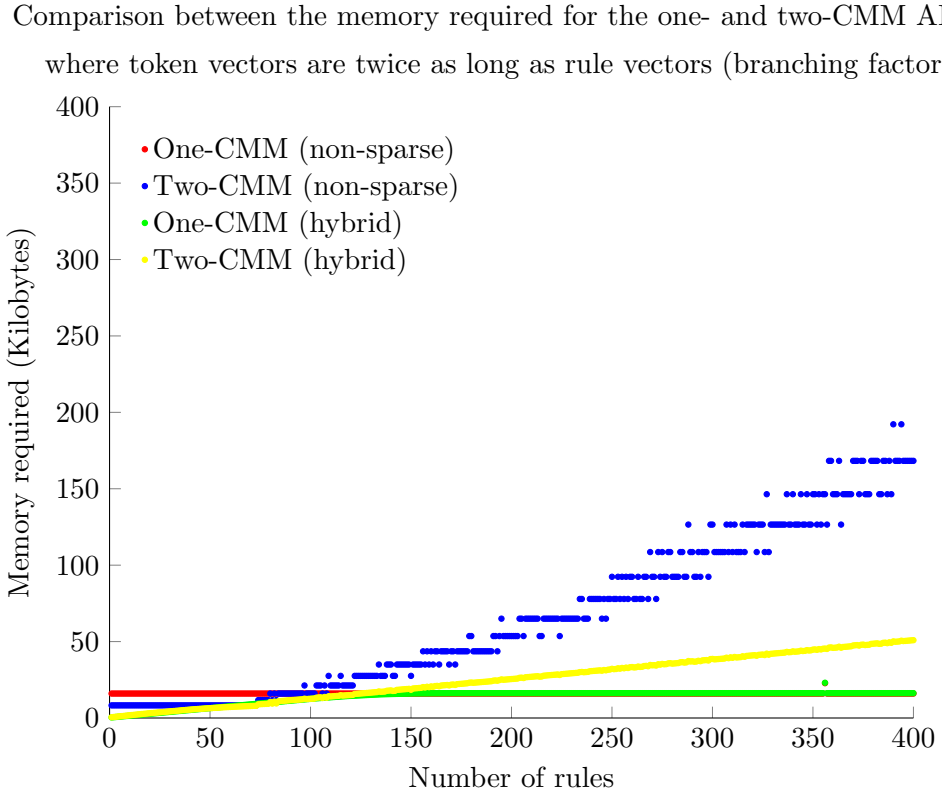


Figure 4.15: Scatter plots showing the memory requirements of ARCA, comparing the one- and two-CMM variants with various storage mechanisms, where the branching factor is 1 and either the rule vectors are double the length of the token vectors (top) or vice versa (bottom). Each point represents the minimum memory required to result in a successful recall in at least 99% of the experimental runs.

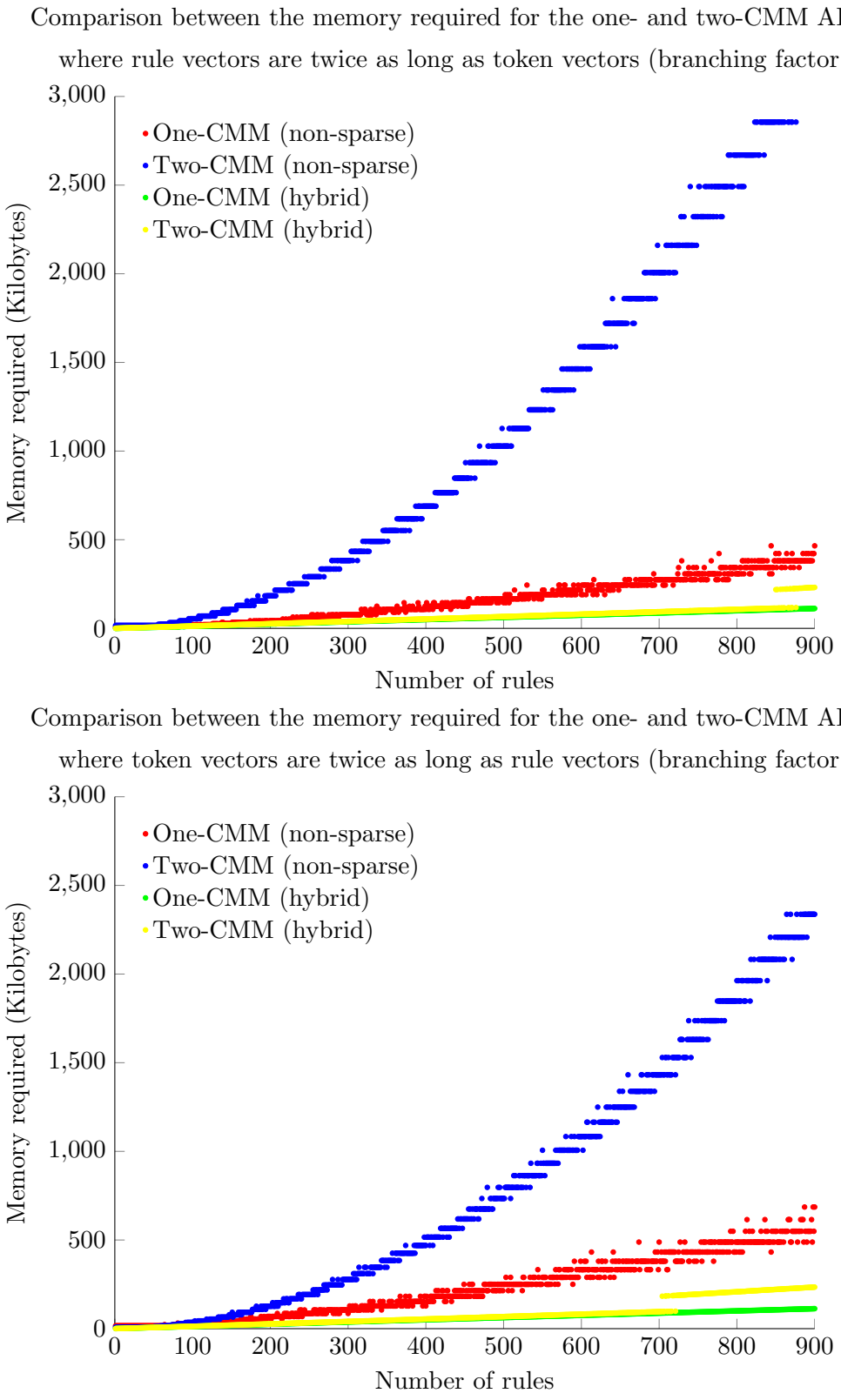


Figure 4.16: Scatter plots showing the memory requirements of ARCA, comparing the one- and two-CMM variants with various storage mechanisms, where the branching factor is 2 and either the rule vectors are double the length of the token vectors (top) or vice versa (bottom). Each point represents the minimum memory required to result in a successful recall in at least 99% of the experimental runs.

store non-sparsely.

When the branching factor is increased the results show similar trends, however there are a number of points which must be addressed. When using non-sparse storage with the two-CMM variant, the memory required is higher when rule vectors are twice as long as token vectors than the opposite. The requirement is not doubled, however, due to the limited capacity of the tensor products. As we saw in Section 4.3, there are two tensor products used in the two-CMM ARCA—$\mathbf{TP_{in}}_{\boldsymbol{x}}$ and $\mathbf{TP}_{\boldsymbol{rx}}$—with dimensions $l_t \times l_r$ and $l_r \times l_r$ respectively (input $\times$ output). This means that increasing the length of rule vectors will increase the capacity of both tensor products, allowing it to have a far greater effect on the overall capacity of the ARCA network.

When using non-sparse storage with the one-CMM variant, the memory requirement is still lower when rule vectors are twice as long as token vectors rather than vice versa, but not by half. In this case there is a single tensor product—$\mathbf{TP_{in}}_{\boldsymbol{x}}$—with dimensions $l_t \times l_r$. As we discussed in Section 3.3.2, increasing the token vector length will therefore increase the capacity of this tensor product faster than increasing the rule vector length: allowing ARCA to store more rules for a given vector length.

The memory requirement when using the hybrid storage format is largely identical—whether the one-CMM or two-CMM variant of ARCA is used, and similarly unaffected by the relative vector lengths. At around 700 rules trained, however, the two-CMM variant begins to require twice as much memory when token vectors are twice as long as rule vectors. This occurs as the length of the output of the consequent CMM ($l_t l_r$) surpasses 2^{16}—as the CMM is stored sparsely, when this threshold is met the location of each set bit is stored using 4 bytes instead of 2 bytes.

4.5 Multiple Arity Rules

The preceding sections of this chapter have developed a rule chaining architecture that is capable of performing rule chaining on *single arity* rules—those with only a single antecedent. In some cases it may be sufficient to operate under this restriction. In complex rule chaining systems, or applications such as feature matching, *multiple arity* rules—those with more than one antecedent—may be necessary [18].

Rules with different arities cannot be stored in the same CMM, because of the operation of Willshaw's threshold and the relaxation ability of CMMs. This issue can be demonstrated most clearly with an example, and so we consider an ARCA system trained

Table 4.2: A set of rules to demonstrate the difficulty with multiple arity, with a binary vector assigned to each individual token vector

Token vector	Binary representation
a	1001000
b	0100100
c	0010010
d	1000001
e	0101000
f	0010100
g	1000010
h	0100001

Rules	
$\mathbf{r_1}$	$\mathbf{a} \rightarrow \mathbf{b}$
$\mathbf{r_2}$	$\mathbf{a} \rightarrow \mathbf{c}$
$\mathbf{r_3}$	$\mathbf{b} \rightarrow \mathbf{d}$
$\mathbf{r_4}$	$\mathbf{c} \rightarrow \mathbf{e}$
$\mathbf{r_5}$	$\mathbf{c} \rightarrow \mathbf{f}$
$\mathbf{r_6}$	$\mathbf{a} \wedge \mathbf{b} \rightarrow \mathbf{g}$
$\mathbf{r_7}$	$\mathbf{a} \wedge \mathbf{d} \wedge \mathbf{g} \rightarrow \mathbf{h}$
$\mathbf{r_8}$	$\mathbf{a} \wedge \mathbf{c} \wedge \mathbf{d} \rightarrow \mathbf{f}$

with the rules shown in Table 4.2—a set of token vectors and multiple arity rules using these vectors.

Upon presentation of a vector 1101100 containing both **a** and **b**, we require the system to match every rule that contains only **a** or **b** in the antecedents: $\mathbf{r_1}$, $\mathbf{r_2}$, $\mathbf{r_3}$, and $\mathbf{r_6}$. To match the single arity rules correctly, the threshold value used must be equal to the weight of a single input vector—a value of 2. Using a threshold value of only 2 means that any rule containing an **a** or **b** in the antecedents will be matched, as this will be recognised as a partial input. This means that rules $\mathbf{r_7}$ and $\mathbf{r_8}$ will be incorrectly matched, in addition to the expected matches. Setting a threshold to resolve this partial matching is impossible, as any value allowing a single arity rule to match will also allow multiple arity rules to match.

Previous work has proposed the use of arity networks as a solution to this difficulty [10], as shown in Figure 4.17. Under this scheme, multiple distinct ARCA networks are created—one for each arity of rules used in the system. Each rule is trained into the correct n-arity ARCA network for the number of tokens in its antecedent, allowing a different threshold to be used for each ARCA network.

Although this scheme will help in many cases, it still does not fully solve the problem. Consider the 3-arity rules given in Table 4.2, $\mathbf{r_7}$ and $\mathbf{r_8}$. When recalling rule $\mathbf{r_7}$, the superimposed tokens will form a vector 1001011 with a weight of only 4, thus the threshold for the 3-arity network must be set as 4 to allow correct recall. For rule $\mathbf{r_8}$, the superposition of input tokens forms a vector 1011011. It can clearly be seen that this vector is very similar to that of rule $\mathbf{r_7}$, with the addition of only a single bit. Unfortunately, we have already determined that the threshold used with the 3-arity network must be at most 4. We can see, therefore, that presentation of the rule $\mathbf{r_7}$ input vectors ($\mathbf{a} \wedge \mathbf{d} \wedge \mathbf{g}$) will cause

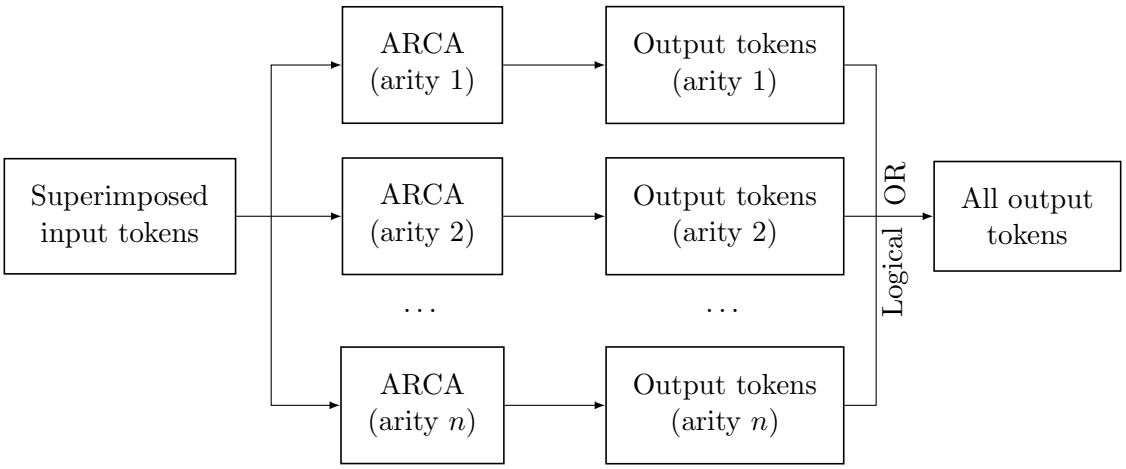


Figure 4.17: Block diagram of multiple arity networks with ARCA

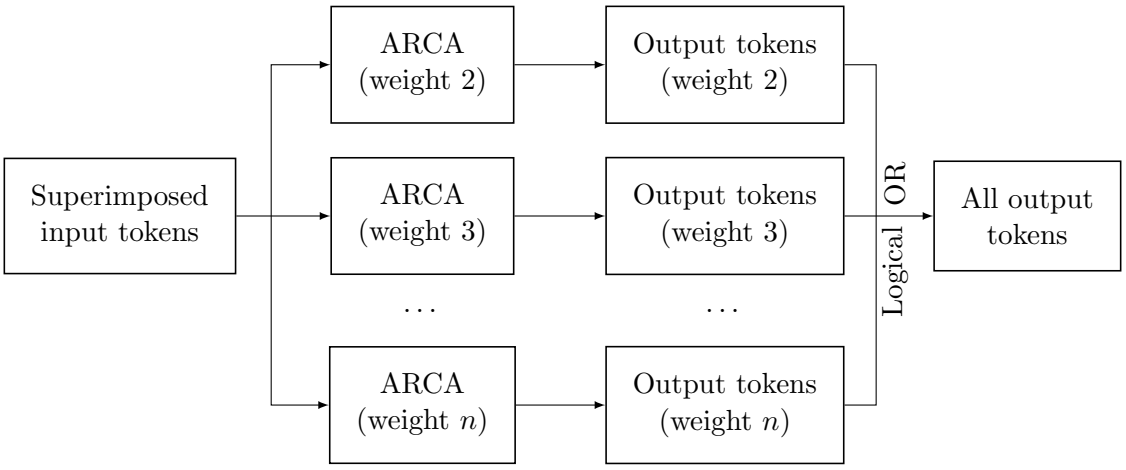


Figure 4.18: Block diagram of multiple weight networks with ARCA

rule $\mathbf{r_8}$ to match incorrectly due to partial matching.

In order to resolve this, a modification can be made to the concept such that instead of separating rules by arity, rules are separated by the combined weight of their superimposed antecedent tokens. This will operate in a similar way to the original proposal, however each ARCA network will be identified by the total weight of the superimposed antecedent tokens rather than by the number of antecedent tokens as shown in Figure 4.18. The minimum possible weight is 2, as distributed vectors are used throughout the system. This minimum may be higher in a particular application, as it is dependent on the fixed weight chosen for token vectors.

This solution is suitable for use with any implementation of CMMs, as it may be applied to both the original ARCA and the single CMM ARCA. The rest of this section is devoted to its application with the single CMM ARCA.

4.5.1 Training

When training a rule, its antecedents are first superimposed and the weight of this vector is used to determine into which of the ARCA networks the rule should be trained. The threshold value for each ARCA network is now well defined, although relaxation can still be achieved by reducing this threshold value. After determining which of the ARCA networks is suitable for the rule, the training process continues as previously detailed in Section 4.4.1.

4.5.2 Recall

A block level diagram of the recall operation is shown in Figure 4.18. To initialise the recall any input tokens are superimposed, for example **a**, **d**, and **g**. This superimposed input vector can then be recalled from each of the individual ARCA networks. As each ARCA network is distinct, this recall may happen in parallel where this is supported by the infrastructure. Given the rules and vectors shown in Table 4.2, upon presentation of this input a number of rules will be matched: rules $\mathbf{r_1}$ and $\mathbf{r_2}$ in the weight-2 ARCA network, and rule $\mathbf{r_7}$ in the weight-4 ARCA network. Rule $\mathbf{r_8}$ will not be matched, as it is stored in the weight-5 ARCA network and so requires all 5 set bits to be present in the recall input.

After recall from each of the ARCA networks, the result is a number of tensor products containing the consequent tokens for each of the matched rules, bound to the rules that caused them to fire. In order to be able to iterate, we must combine these into a single tensor product. In this instance, as the output of each ARCA network is independent of the others, we must simply superimpose them using a logical OR.

Testing for search completion on this superimposed tensor product operates in the same fashion as the single arity ARCA. If the superimposed tensor product consists solely of zeros, then no rules have been matched in any of the ARCA networks and hence the search is completed without finding a goal state. If the superimposed tensor product is not empty, then we must check whether all of the consequent tokens for the goal state are present in the output—if they are then the search has been successful, and if not then the search will iterate.

4.5.3 Experimentation

In order to show that "weight" networks are effective, and to determine the memory overhead caused by having multiple separate ARCA networks, the single CMM ARCA has been applied to a set of randomly generated forward chaining problems. For each experiment a tree of rules was generated with a given depth d and maximum branching factor b, using the same procedure as detailed in Section 4.3.3. In these experiments, the number of antecedents and consequents for each rule was uniformly randomly sampled from the range $[1, 5]$. These rules were then learned by the multiple-arity ARCA system, and rule chaining was performed on them.

As with previous experimentation, the weight of all vectors has been set to 4. In this case, the superposition of vectors means that the combined weight of the antecedents or consequents of a rule will be in the range $[4, 20]$. In order to reduce the scope of the experiment, the length of token vectors and rule vectors are equal—varied between 128 and 8192 bits.

The graphs in Figure 4.19 are scatter plots showing, in red, the minimum memory required for ARCA to successfully store and recall a given number of multiple arity rules in at least 99% of the experimental runs. Additionally, in green, an "adjusted" memory requirement is shown in order to allow comparison with the previous single arity results. The adjusted value is calculated by dividing the number of bits in each row of each matrix by the average superimposed vector weight of a given experimental run, and multiplying by 4 (the weight of a single vector). This can only be an approximation, as changing the superimposed weight of vectors will also affect the overlap between rules.

As would be expected, ARCA requires more memory to store the same number of rules when those rules are multiple arity rather than single arity. This is because each rule results in a greater number of associations being stored in the CMMs. Simply increasing the number of CMMs also has an effect on the memory requirement, as there are fixed overheads associated with each CMM when using sparse storage.

To aid comparison with the single arity results from Figure 4.13, these previous results are shown together with the adjusted multiple arity results in Figure 4.20. The adjusted memory requirement of the multiple arity network is clearly higher than that of the single arity network, however it is important that there is still a linear relationship between the number of rules and the memory required. The multiple arity results are also likely to be overestimating the adjusted memory required, as the number of row pointers is not

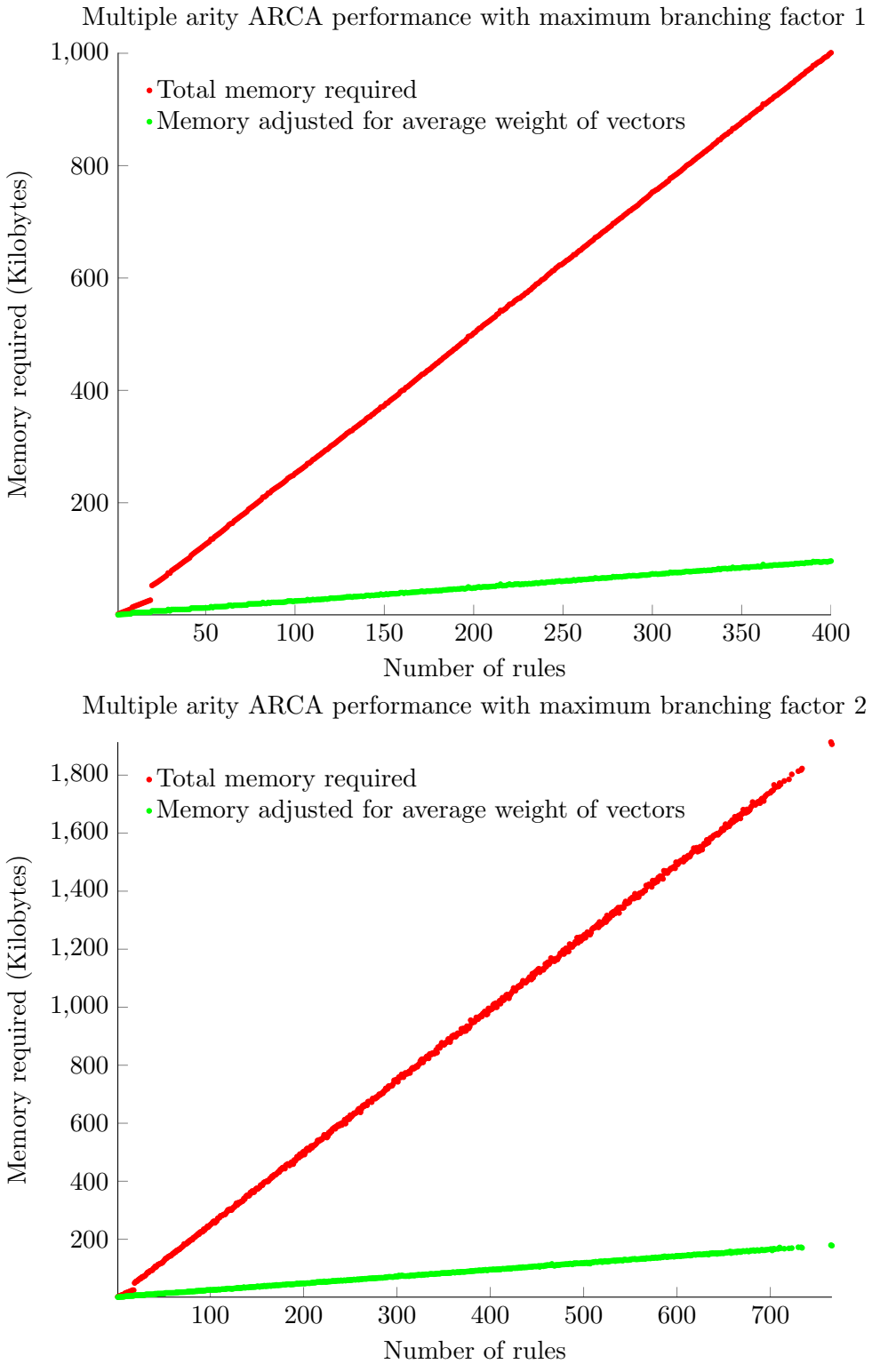


Figure 4.19: Scatter plots showing the memory requirements of ARCA with multiple arity, where the branching factor is 1 (top) and 2 (bottom). Each point represents the minimum memory required to result in a successful recall in at least 99% of the experimental runs. Each plot contains two graphs; the total memory required is shown in red, and the memory required when adjusting for the superimposed vector weight is shown in green.

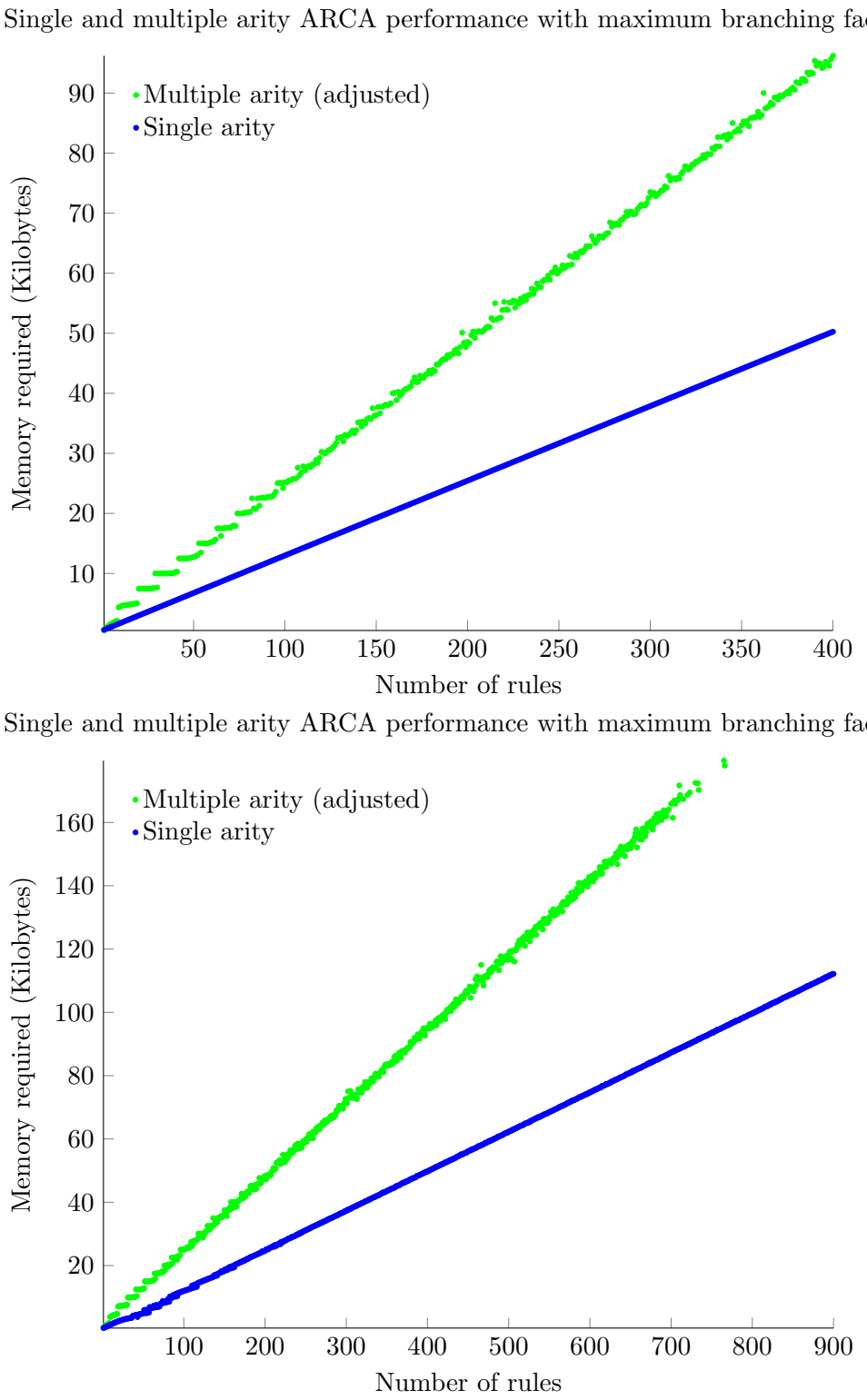


Figure 4.20: Scatter plots comparing the memory requirements of ARCA with single and multiple arity (adjusted), where the branching factor is 1 (top) and 2 (bottom). Each point represents the minimum memory required to result in a successful recall in at least 99% of the experimental runs.

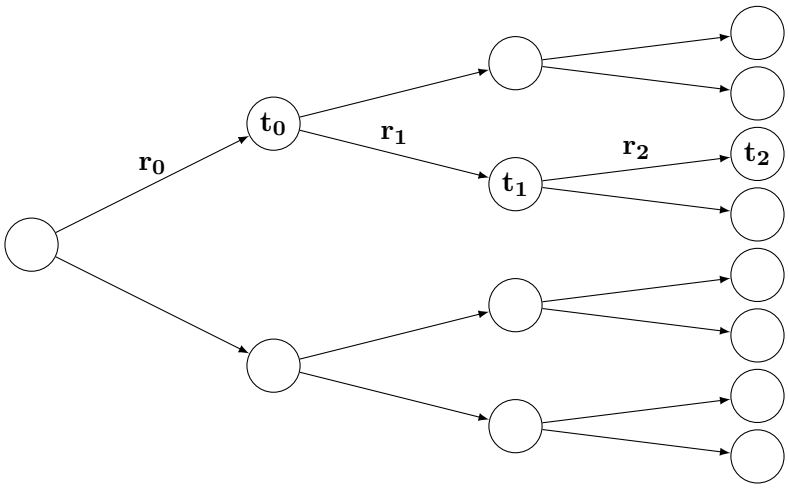


Figure 4.21: An example tree of rules, where only the path to be found is labelled

adjusted. The n CMMs used to store multiple arity rules require nl_i row pointers, where l_i is the input vector length, rather than only l_i row pointers required by single arity.

4.6 Pathfinding

When performing rule chaining, the path between the starting state and the goal state is often unimportant. In expert systems, for example, the task is to infer new information rather than to determine how that new information has been produced. This is not true in all cases—for example an application that wished to find the shortest path between two nodes, or a solitaire[2] solver designed to provide a solution to a player, rather than simply inform them whether the game can be completed. Previous work on ARCA has left this unaddressed, and so it is important to demonstrate that this is possible when using the Associative Rule Chaining Architecture.

The architecture does not require any adjustments in order to be used in this fashion, however during a recall each intermediary tensor product must be saved in order that the path can be extracted. Recalling the operation of the network shown in Figure 4.12, both the input and output of the CMM are tensor products—binding a token $\mathbf{t}$ to a rule $\mathbf{r}$. We will consider a simple example where the goal state is reached after two iterations, as this is sufficient to demonstrate that the method is effective. The path we are searching for is labelled in Figure 4.21, within a larger tree of rules.

[2]Solitaire is also known as peg solitaire or solo noble, and involves moving pegs on a board with holes, with an aim to eliminating all but one of the pegs.

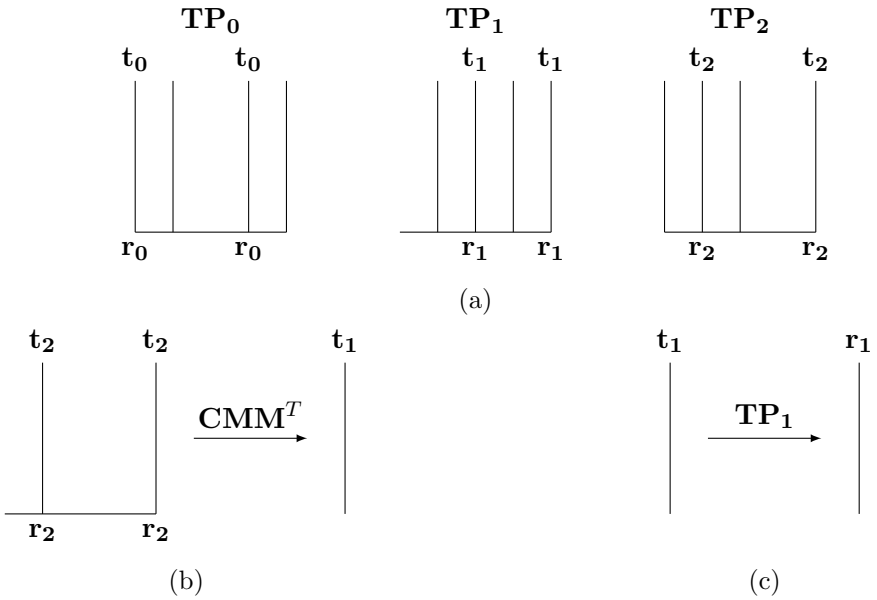


Figure 4.22: (a) The input tensor product ($\mathbf{TP_0}$) and output tensor products after each iteration ($\mathbf{TP_1}$ and $\mathbf{TP_2}$), each containing tokens in addition to those which form the path, (b) recalling a flattened tensor product from the transposed CMM to find the rule's antecedents, and (c) recalling a token vector from an intermediary tensor product to find out to which rule vector it was bound.

Figure 4.22a shows the tensor products saved during each iteration of our example—$\mathbf{TP_0}$ is the original input, and $\mathbf{TP_1}$ and $\mathbf{TP_2}$ are the output after each iteration of recall. Each of these tensor products is shown containing additional, unlabelled tokens to indicate that they may contain more tokens than those which form the path. The goal state contains only the token $\mathbf{t_2}$—recalling this token from $\mathbf{TP_2}$ resulted in the rule $\mathbf{r_2}$, and so we know the search has completed successfully. Our knowledge of the path is currently limited to the beginning and end:

$$\mathbf{t_0} \xrightarrow{\mathbf{r}_?} \ldots \xrightarrow{\mathbf{r_2}} \mathbf{t_2}$$

We also know how many iterations were required, and hence the length of the path:

$$\mathbf{t_0} \xrightarrow{\mathbf{r}_?} \mathbf{t}_? \xrightarrow{\mathbf{r_2}} \mathbf{t_2}$$

There are two stages to the procedure used to move one step backwards up the tree of rules. In the first stage, we must find the antecedents of the rule which resulted in our goal state. This is shown in Figure 4.22b: the tensor product formed between our goal token and the rule token is recalled from the transposed CMM. Transposing the CMM means

that the input is now a flattened tensor product, and the output is a token vector. In our example, recalling $\mathbf{t_2} : \mathbf{r_2}$ will result in the output $\mathbf{t_1}$—we now know that the complete rule is:

$$\mathbf{t_1} \xrightarrow{\mathbf{r_2}} \mathbf{t_2}$$

and the path is:

$$\mathbf{t_0} \xrightarrow{\mathbf{r_?}} \mathbf{t_1} \xrightarrow{\mathbf{r_2}} \mathbf{t_2}$$

The second stage determines the previous rule which fired, in order that we can create a new tensor product to iterate by repeating the first stage. This method is the same as that used to detect successful completion during an iteration of recall—we recall the tokens found in stage 1 from the intermediary tensor product for the correct iteration. In our example, this means we recall $\mathbf{t_1}$ from $\mathbf{TP_1}$, resulting in the output $\mathbf{r_1}$. We can now repeat the procedure in order to move further back up the tree of rules. In our example, however, this is unnecessary as we have completed the path:

$$\mathbf{t_0} \xrightarrow{\mathbf{r_1}} \mathbf{t_1} \xrightarrow{\mathbf{r_2}} \mathbf{t_2}$$

4.7 Application to Solitaire

Solitaire, or solo noble, involves moving pegs on a board with holes with an aim to eliminating all but one of the pegs. On an English solitaire board, shown in Figure 4.23, there are 33 holes and 32 pegs—meaning there are theoretically 2^{33} possible board positions. In actuality 23,475,688 unique positions—unique when rotational and reflectional symmetry are taken into account—are reachable when following a valid series of moves. A move involves "jumping" one peg over another into an empty hole before removing the peg which was jumped over. English solitaire has over 185.9M valid moves, with a maximum branching factor of 21.

The moves are naturally represented as an unordered tree, and so searching for a solution to a given board position is an ideal example of ARCA's capabilities. Each move consists of an input state and an output state, the affected pegs are implied by the differences between these states. A state is most simply represented as a 33-bit integer, with a `1` in each particular bit position indicating the presence of a peg in its respective hole. An alternative representation, mapping each of the unique and reachable states to a 25-bit integer, would clearly be more efficient in terms of memory used (especially so

```
    B C D F G H J
A       X X X
E       X X X
I   X X X X X X X
O   X X X O X X X
U   X X X X X X X
W       X X X
Y       X X X
```

Figure 4.23: The starting layout of an English solitaire board. An X represents a hole containing a peg, and an O represents an empty hole. The rows and columns are labelled in accordance with Wolstenholme notation[3].

if the integer sizes are rounded to the next power of 2). There is no simple mapping for these states, however, and so a lookup would be required in order to translate between a state and its representation—imposing an undesirable delay on any search.

4.7.1 Configuration

Due to the time required to train and test a simulated ARCA system storing this many rules, it was decided that vector lengths would be set at an equal length—testing each increasing power of 2 until a successful length was found. At these vector lengths it would be impractical to store ARCA's matrix non-sparsely (for example a length of 2^{18} bits results in a matrix of 2^{54} bits, or 2 petabytes), and so the vector weights are all set at 2 to reduce the memory required when using sparse storage. This very low input weight has previously been shown to reduce a CMM's capacity from its potential, however it also causes a large reduction in memory requirement when sparse storage is used. Using the simple state representation described above, the highest value which may need to be uniquely represented is 2^{33}, or 8,589,934,592. With a weight of 2, the highest possible number of vectors available when using Baum's algorithm with a vector length of 2^{17} bits is 4,294,967,295 (using co-primes 65,535 and 65,537), as such the minimum vector length attempted was 2^{18} bits.

In order to determine whether ARCA using a given vector length can successfully store Solitaire's 185.9M moves, they were first trained into a CMM using the procedure detailed in Section 4.4.1. Each state is represented using the simple 33-bit representation

[3]Wolstenholme notation, and further information regarding English solitaire, can be found at `http://www.topaccolades.com/notation/solitaire.htm`

above, and converted into a fixed weight vector using the 33-bit integer as an index into the sequence of vectors generated using Baum's algorithm (as was described in Section 2.3.5.3).

The initial state—8,589,869,055 in decimal—was then presented for recall using the procedure detailed in Section 4.4.2. After each of the 31 iterations, the target state—65,536 in decimal—was recalled from the output tensor product. Prior to the final iteration the target state should not be present in the output tensor product, as the number of pegs remaining after each iteration is fixed. The system was therefore deemed to have failed if at any point prior to the final iteration this recall resulted in any output, or if the output after the final iteration was not a valid rule vector.

ARCA was able to successfully store and recall the entire tree of moves when using a vector length of 2^{23} bits and weight of 2. If non-sparse storage was used for the matrix, this would require a total of 2^{69} bits, or 64 exabytes, of memory. Using the hybrid storage mechanism introduced in Section 3.3, however, the memory requirement is reduced to a more practical 11.11 gigabytes. As a simple comparison, the memory required to store the 23,475,688 unique states and 185.9M moves in a tree structure is 1.04 gigabytes. This assumes that each node consists of a 64-bit integer state, a 32-bit integer to store the number of children, and a 32-bit pointer to an array containing pointers to each of the children.

4.7.2 Experimentation

For the purposes of simulation, the CMM could not be stored in memory in its entirety, and so it was divided into sections stored on disk. As a result, it is not possible to perform any reasonable comparison of the time required to perform a search. If implemented using a hardware platform such as C-NNAP [61], however, recall from a binary CMM is a single, fast operation. As a result, the comparisons presented here are simply between the number of higher-level operations required—counting the number of tree nodes visited, or the number of ARCA iterations required. The time required for each of these operations will vary greatly, depending on the implementation.

4.7.2.1 Checking a State

The simplest experiment with which to first compare ARCA and a standard DFS is the number of operations required to determine if an input state is valid or invalid—i.e. whether the state may be reached by following only valid moves. This process is very

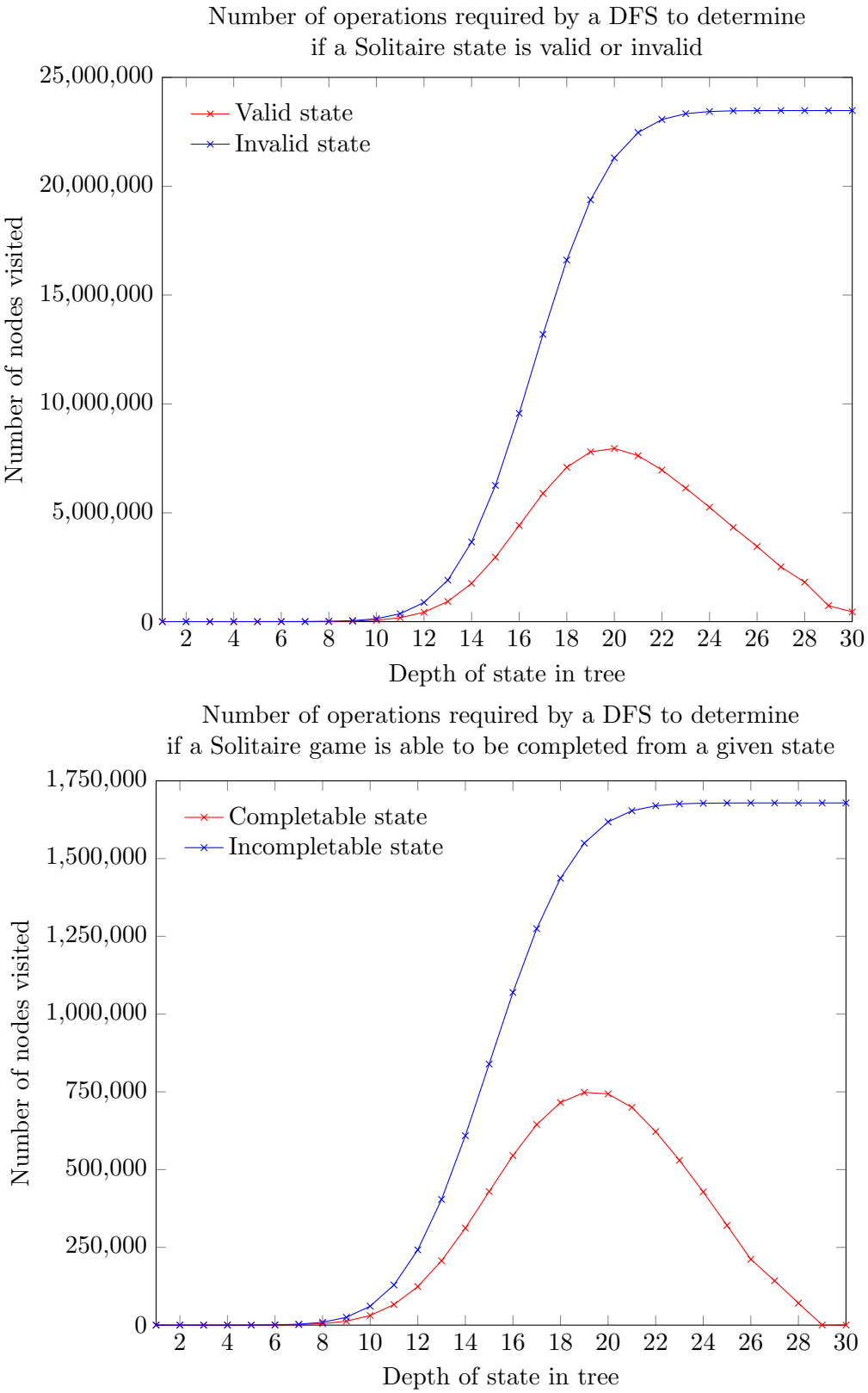


Figure 4.24: Plots showing the number of operations required by a DFS, compared to the single operation ARCA requires. For each depth of tree the plots show the average number of nodes which must be inspected to find a valid/completable state, as well as the number of nodes which must be inspected to determine that a state is invalid/incompletable.

simple using ARCA, and requires only a single recall operation. Upon presenting the state as an input, an empty output indicates that the state is invalid and a non-empty output indicates otherwise. Using a DFS, the tree of rules must be searched starting at the root and following each possible path until the state is found or all paths have been exhausted. As the state explicitly encodes the number of remaining pegs, the maximum depth of the search is easily calculated as $32 - \text{HW}(\text{state})$, where $\text{HW}(x)$ indicates the Hamming weight of the binary value x.

The top of Figure 4.24 shows the number of operations required to determine if a state is valid or invalid for each depth of the tree. The results show the number of nodes which must be inspected when using a DFS, firstly as an average number for those states which do exist in the tree, and secondly as the number for any invalid states. As the number of operations required using ARCA is always 1, these values can essentially be considered as the speedup provided by the architecture.

Removing any states from the tree from which it is not possible to complete the game leaves 1,678,935 unique states and 8.5M possible moves. Although it is no longer possible to determine if a state is valid or invalid, a reduction of the number of rules improves the performance of a DFS when attempting to determine if the game is able to be completed from a given state. As such, the bottom of Figure 4.24 shows the number of operations required to determine if a game is able to be completed from a given state for each depth of the tree. As before, the results show the number of nodes which must be inspected when using a DFS, firstly as an average number for those states which do exist in the tree, and secondly as the number for any valid state from which the game cannot be completed. Again, ARCA requires only a single operation to determine if a state exists in the tree.

4.7.2.2 Finding a Solution

More useful than an ability to simply determine if a state is valid or invalid, or if a game may be completed from a given state, is the ability to find a solution from any given valid state. In order to determine a solution using ARCA, the procedure detailed in Section 4.6 can be applied—most effectively using the smaller tree of states from which there is a solution. The current state is presented as the initial input to ARCA, and iteratively recalled until the target state is found. As the tree of rules contains only those from which it is possible to complete the game, it is guaranteed that the target state will be found. It is also known that this will occur after $\text{HW}(\text{state}) - 1$ iterations, as the state explicitly

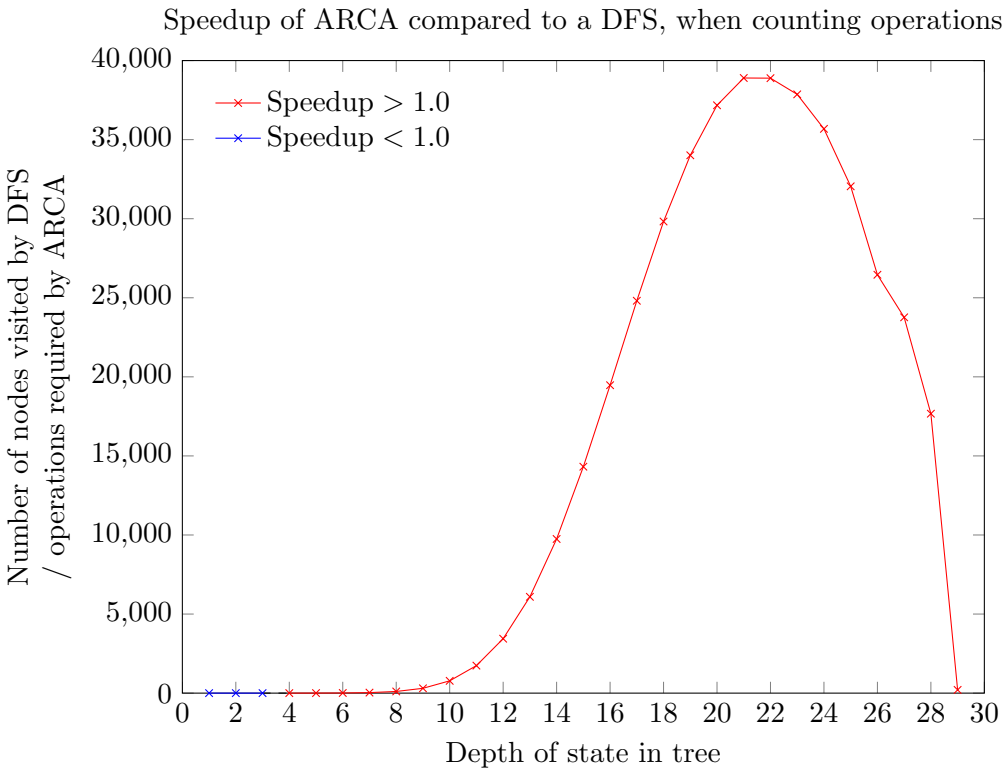


Figure 4.25: Scatter plot comparing the number of operations required by a DFS to that required by ARCA to find a solution for a given, completable state. The number of operations required by the DFS is divided by that required by ARCA to obtain a speedup value for a state given its depth in the tree.

encodes the number of pegs remaining. Having completed the search, the path is found by backtracking through the intermediary results—requiring an additional HW(state) − 1 operations.

Using a DFS, it must first be determined if the state exists in the tree as demonstrated in Section 4.7.2.1. Once the input state is found in the tree, it is trivial to find a path to the solution: it is possible to complete the game from every node which remains in the reduced tree, and so following any path will result in success. Figure 4.25 compares the number of operations required by a DFS and that required by ARCA for those states from which it is possible to complete the game. The results are averaged across all states with the same number of pegs remaining—that is, at the same level in the tree of moves. Each point is calculated as the average number of nodes which must be inspected to find a given state plus the number of nodes which must be inspected to find the solution (in this case simply HW(state) − 1). This value is then divided by the number of operations required by ARCA, which as stated above is 2(HW(state) − 1).

These results clearly show that ARCA is suitable for application to large problems and

real datasets. The speedup which may be obtained by using ARCA in place of a DFS is very dependent on the distribution of the data, however the results of this experimentation show that it can reach 4 orders of magnitude. It should also be emphasised that the tree used for the final results in Figure 4.25 was pruned prior to searching for a given node—without this pruning the DFS took an order of magnitude longer. ARCA, on the other hand, requires the same number of operations whether the tree is pruned to remove incompletable states or not (although the memory required by the pruned tree would inevitably be smaller).

4.8 Further Work

ARCA has been shown to be capable of performing rule chaining effectively, with a linear relationship between the number of rules stored and the memory requirement. Further work is still required, however, particularly in the areas described below.

4.8.1 Vector Lengths and Weights

The lengths and weights of vectors used to represent tokens and rules in ARCA are very important, as these determine the capacity and memory requirement. The weight chosen in this work may not be optimal, and so this needs further investigation. Similarly, the relationship between the token vectors and rule vectors is not well understood—further experimentation of varying their lengths and weights independently may yield performance or capacity improvements.

4.8.2 Graphs

ARCA is able to perform rule chaining with any directed acyclic graph, tree, or forest of rules, whether the rules are single or multiple arity. It is not ideal for use with cyclic graphs, however, due to the stopping condition employed. In an unsuccessful recall, the stopping condition is that the output of an iteration is all zeros. If a loop is encountered, for example with two rules $\mathbf{a} \rightarrow \mathbf{b}$ and $\mathbf{b} \rightarrow \mathbf{a}$, then an unsuccessful recall will never end—recall operations will only complete if the goal state is reached. Loop detection may therefore need to be incorporated if ARCA is to be used with cyclic graphs.

4.8.3 Weighted Edges

The pathfinding presented in Section 4.6 is limited to unweighted edges—essentially finding the smallest number of "hops" between two nodes, as traversing any edge carries the same cost. It may be possible to find an encoding to be used for the rule vectors, in order to incorporate weighted edges into ARCA. If the system is also extended to fully support graphs then ARCA would be able to find the shortest path between two nodes, providing a fast, neural-inspired alternative to Dijkstra's Algorithm or A*.

4.9 Summary

In summary, significant improvements to ARCA have been presented. Most importantly, the relationship between the number of rules stored and the memory required has been reduced from exponential to linear by modifying the architecture to use only a single CMM. It has also been shown that the use of sparse storage can provide a significant memory reduction, either individually or in addition to the saving achieved when using a single CMM.

A mechanism to allow ARCA to store multiple arity rules has been fixed and demonstrated to work effectively, with minimal memory overhead compared to a single arity network. If all rules used in a system have the same arity then if possible these should be mapped to a single arity ARCA network for performance—requiring a recall from only one CMM rather than many. If the system contains rules with various different arities, however, multiple weight networks are required.

ARCA has been shown to be able to determine the path between two nodes by backtracking after a search. Although this may be unnecessary in some applications of inference, it is an important facility in applications such as the control of autonomous agents.

Finally, ARCA has been successfully demonstrated on a very large problem requiring pathfinding. It has been shown to be able to reduce the time complexity of finding a solution to a particular Solitaire state by up to 4 orders of magnitude compared to a DFS. When the challenge is simply to determine if a state is valid or invalid, the time complexity is reduced by up to 7 orders of magnitude.

Chapter 5

The Cellular Associative Neural Network

5.1 Introduction

Pattern recognition is one of the few tasks in which computers may still be outperformed by humans. A large amount of research has been directed to specific types of pattern recognition, such as Optical Character Recognition, however these techniques and results often do not generalise well to the wider field. Recent research has developed techniques that aim to harness the human ability to perform visual pattern recognition [118], and so a solution that uses biologically-inspired neural networks is of particular interest.

The Cellular Associative Neural Network (CANN) performs syntactic pattern recognition, using elements from cellular automata and CMMs. It has been shown to be capable of distributed symbolic processing and pattern recognition with translation invariance, but without scale invariance [78]. The next sections describe the architecture, before moving on to modifications and improvements.

5.2 The Architecture

The CANN is an array of cells—known as associative processors—each of which contains a number of modules. The modules use CMMs to store rules that symbolically describe an object. Each module contains one or more CMMs, configured as an arity network, in order that the number of antecedents to a rule can be variable [18]. Whereas each cell in a cellular automata has only a single state, an associative processor in the CANN

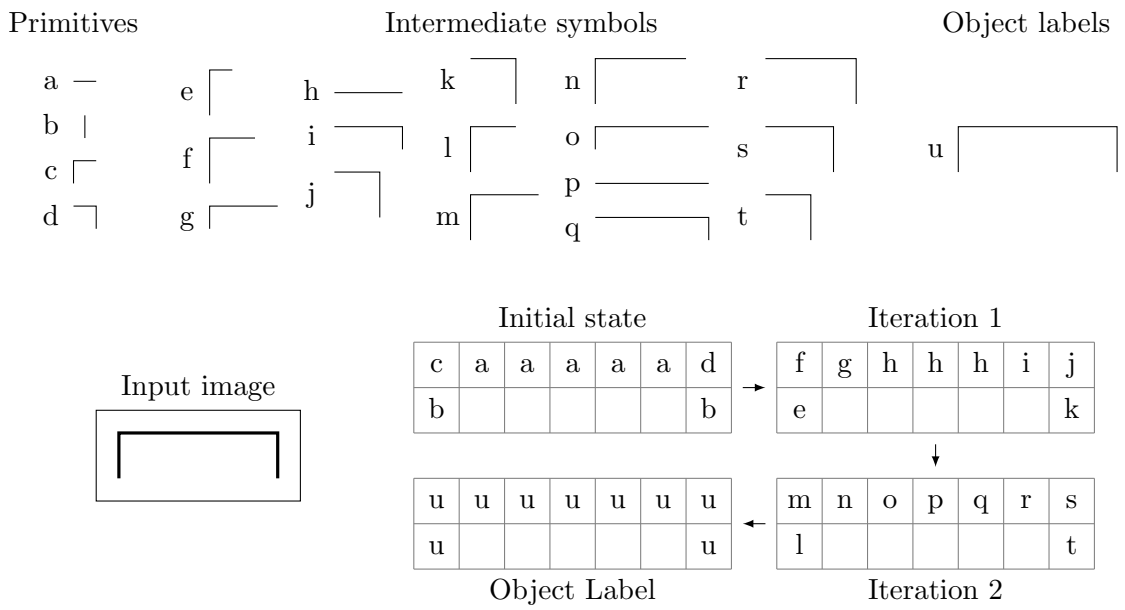


Figure 5.1: An example of the CANN recognising a simple shape. When learning an object, the training iterations end when each cell containing an input has been assigned a unique state—at this point, these cells are assigned the object label. In this example, this stage is reached after two iterations.

has additional outputs: one for each neighbour to allow the flow of information to be controlled.

Learning and recognition of an object's structure uses a hierarchical approach, and cells exchange symbolic information with their four direct neighbours during each iteration. This means that after n iterations, each cell is made aware of the state of all cells up to a Manhattan distance of n from it. Figure 5.1 shows an example of this operation, recognising a simple shape. Various module configurations for the 2D CANN have been investigated, in order to optimise this message passing.

Brewer [16] showed that the "Corner Turning 2" configuration shown in Figure 5.2a provides the best performance of those tested—allowing information to travel between any two cells, while requiring fewer total rules than alternative suitable configurations. The data flow of this configuration is shown in Figure 5.2b, where the black cell is the origin of a piece of information, grey cells are those to which the information has been passed, and white cells are those which are not yet aware of the information.

It should be noted that Brewer's experiments were somewhat limited by the small number of cells used—set as a 20×20 grid. For each of the shapes learned, and module configurations used, the shape information reached the boundary of this grid and hence artificially limited the number of rules generated. For example, when learning all of the

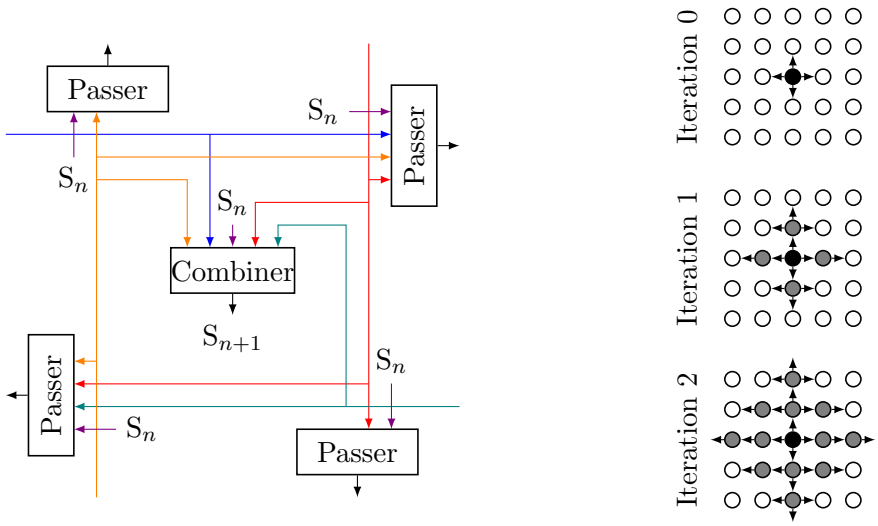


Figure 5.2: (a) The "Corner Turning 2" CANN module configuration, where S_n is the cell state (the output of the combiner module) after the n^{th} iteration and (b) information flow of this configuration over two iterations [16].

shapes shown in Figure 5.4 using the "Corner Turning 2" configuration a total of 17164 rules are generated on an "infinite grid"—rather than the 14847 stated in [16]. In general, however, this does not have a significant effect on the operation of the CANN as the information only ever flows outwards from a cell. Only when using Brewer's "Cellular Automata" configuration could this cause an issue with translation invariance, due to the reflection of information back towards a cell.

5.2.1 Learning

Before learning an object's structure, a number of "primitives" are first recognised in the image—vertical and horizontal lines, and the four types of corner: |, —, ⌜, ⌝, ⌞, ⌟. Each of these primitives is represented by a vector, and one vector forms the initial input to each cell overlaying the image. During an iteration, information is received from the relevant passer modules of each of a cell's neighbours to form the antecedents of a rule. For each module, an input vector is created by appending the information received from neighbouring cells to the cells' state (or its initial input for iteration 0), according to the module configuration. This vector is then used to recall a new cell state from the combiner module, or a new information vector from a passer module. The position that a vector is appended within the input is determined by the module configuration, which allows a cell to distinguish between information received from different neighbours.

If a recall does not result in any output then this combination of antecedents has not been seen previously. In this case a new vector—a "transition symbol"—is generated to form the consequent of a rule, and associated with the antecedents into the relevant module. This transition symbol—whether it has been recalled or newly generated—is used either as the cell's output state or to form the new information to be passed to the cell's neighbours, depending on the module. When a transition symbol is generated, the new rule must be communicated to all other cells. This ensures that all cells contain the same information which allows the CANN to be translation invariant.

Finally, when every cell that initially contained a primitive has been assigned a unique state, the termination condition for learning an object is reached. At this point each of these cells generates a final rule to be stored in the combiner module. As with all other iterations the cell's inputs are used as the antecedents, however the consequent in this case is a user provided symbol which denotes the learnt object.

Appending vectors in order to create a module input, rather than superimposing them, is feasible because each module has a fixed number of inputs n_i. This means that the module has a fixed input length of $n_i \times l$, where l is the vector length used in the system. If one of the cell's neighbours does not have any information to pass, then an empty vector will be transferred and hence included in the input to one or more modules, leading to the requirement of an arity network. For example, in Figure 5.2a, the "combiner" module has a total of 5 inputs—all four neighbours, and the state of the cell itself (the output of the combiner module from the previous iteration). If a vector weight of 4 is chosen, then when all the inputs contain information the total weight is $5 \times 4 = 20$—this is used as the value for Willshaw's threshold. If one of the cell's neighbours does not pass any information, however, then the total weight will only be $4 \times 4 = 16$. If this were to be recalled from a CMM with a threshold value of 20, then it could never result in an output. As such, it is stored in a separate CMM with a threshold value of 16. A recall operation can then present the input vector to the correct CMM in this arity network, using the relevant threshold value.

5.2.2 Recall

When a pattern is presented for recall, the operation of the CANN is similar to that of a cellular automaton. The rules which govern state transitions are stored in the various modules—with each cell containing exactly the same rules, to allow a pattern to be

recognised by any group of cells. To begin a recall, the primitives are extracted from the pattern and used as the input to each cell. As with the learning process, a recall happens iteratively; during each iteration information is received from each of a cell's neighbours and appended to its current state, before recall from each of the modules.

If the pattern is recognisable, then after a number of iterations it will be labelled with a symbol representing the object. It would be unrealistic to expect a perfect recall to happen in every case, however, due to factors such as noisy inputs, distortion, and occlusion. In these cases, the system is able to generalise by taking advantage of a CMM's ability to perform partial matching. If, at any stage, a consequent is not successfully recalled from a module, then relaxation can be employed—that is to say that the threshold value will be reduced in order that an incomplete match may be attempted. This also allows the CANN to recognise inputs which are similar to patterns which have been previously trained [16].

5.3 Removing Arity Networks From the CANN

The use of arity networks in the CANN is a significant limitation that must be addressed before further improvements may be made. In other architectures, such as ARCA, arity networks are used to allow rules with a variable number of antecedents to be stored. In this usage, the arity network is formed from a number of CMMs and a rule is trained into the appropriate CMM. When recalling a rule, however, it is recalled from all of the CMMs—with the final result being the superposition of all of the individual outputs.

Arity networks are used in the CANN because some of the inputs to a module may be empty, and hence the total weight of the combined inputs may vary. In order for a recall to be successful, without allowing unwanted partial matching, a rule may only be recalled from the appropriate CMM. Consider, for example, the shapes in Figure 5.3a. These shapes are clearly similar, and could be converted to primitives as shown in Figure 5.3b. Note that the central cell of the T-shape (highlighted) contains three primitives, as all of these basic shapes are contained within the cell in the original image.

If, during a recall of the T-shape, the inputs to the red cell are presented to each of the CMMs in the arity network then an unwanted match with the ⌜-shape will result. This is because all of the elements of the second shape are present in the input, and so the threshold for this shape will be reached. If the inputs are presented only to the appropriate CMM in the arity network, then only the expected T-shape will match. If relaxation is required, due to an incomplete or distorted input, then the inputs may be presented to an

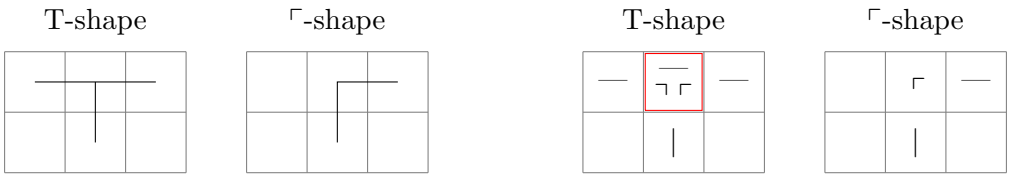


Figure 5.3: (a) Two shapes which demonstrate the limitations imposed by the CANN's use of arity networks and (b) these shapes converted to primitives. Note that the central cell of the T-shape (highlighted) contains three primitives, as all of these basic shapes are contained within the cell in the original image.

alternative CMM or the threshold may be reduced to allow partial matching to occur.

This limitation that an input may only be presented to the appropriate CMM means that each module does not truly contain an arity network, but simply a number of independent CMMs storing different rules. It is possible, however, to remove this requirement entirely—simplifying each module to contain only a single CMM—without losing any ability for relaxation.

5.3.1 Use of a `NULL` Vector

In the CANN, each module has a fixed number of inputs—for example the "combiner" module has 5 inputs. Multiple CMMs were used in each module of the CANN because although the number of inputs is fixed, one or more of those inputs may be empty. Rather than storing these rules in separate CMMs, a `NULL` vector can instead be used to represent an empty input. If this `NULL` vector has the same weight as other vectors used in the CANN, then all rules will have the same input weight and so may be stored in a single CMM.

The `NULL` vector is only ever used in the input of a rule—unrecognised inputs will still result in an empty output. If a cell receives an empty input, either from a neighbour or as its own state, then it simply uses the `NULL` vector in its place. During the training phase, a single check is required: if all of the inputs to a module are empty, then a new rule should not be generated.

5.3.2 Relaxation

The final requirement that must be satisfied is the ability for the CANN to perform relaxation equivalent to that used in the original design. Brewer [16] defined various relaxation options, and grouped them into classes of priority. A recall was first attempted

Number of errors	CMM	Threshold	Grouping	Threshold without arity
No relaxation	a	aw	0	nw
1 incorrect symbol	a	$(a-1)w$	1	$(n-1)w$
1 extra symbol	$a-1$	$(a-1)w$	1	$(n-1)w$
1 missing symbol	$a+1$	aw	1	$(n-1)w$
2 incorrect symbols	a	$(a-2)w$	9	$(n-2)w$
1 extra & 1 incorrect	$a-1$	$(a-2)w$	9	$(n-2)w$
2 extra symbols	$a-2$	$(a-2)w$	9	$(n-2)w$
1 missing & 1 incorrect	$a+1$	$(a-1)w$	9	$(n-2)w$
2 missing symbols	$a+2$	aw	9	$(n-2)w$

Table 5.1: Relaxation options with the CMM arity and threshold that should be used for recall, where a is the rule arity, and w is the fixed vector weight. The grouping numbers were used as Brewer's "Default Global Tolerance Setting" [16]. The final column shows the threshold required to achieve each relaxation option without the use of an arity network, where n is the number of inputs to a module.

using each of the relaxation options in group 0. If a recall was unsuccessful, then each of the options in the next group was attempted. A grouping number of 9 indicated that relaxation option would not be used. Table 5.1 is derived from [16] and shows each of the defined relaxation options with the CMM arity and threshold that will be used for recall, where a is the rule arity, and w is the fixed vector weight. The grouping numbers were used as Brewer's "Default Global Tolerance Setting". The final column shows the threshold required to achieve each relaxation option without the use of an arity network.

To illustrate how these relaxation thresholds have been determined, we will use a simple example with a combiner module where individual vectors have a weight of w. Our original rule in all cases has the input {**a**, **b**, **c**, `NULL`, `NULL`}, which should normally be recalled with a threshold of $5w$. Firstly we will consider the case where one symbol is incorrect. If we present an incorrect input of {**a**, **b**, **p**, `NULL`, `NULL`}, then we must use a threshold of $4w$ ($1 \times w$ for each correct vector)—or $(n-1)w$. The vector **c** has been replaced by **p** and so the input neurons usually activated by **c** will not be activated, causing the output values to be w lower than during a perfect recall.

Next we consider the case where there is an extra symbol present in our input: {**a**, **b**, **c**, **q**, `NULL`}. As `NULL` vectors are treated the same as any other symbol vector, we can see that this is actually an alternative case where one symbol is incorrect. In this case a `NULL` vector has been replaced by **q**, and so the threshold must be $(n-1)w$ for the same reason as above.

Finally, the case where there is a symbol missing from our input, for example {**a**, **b**, `NULL`, `NULL`, `NULL`}. Again we can see that this is simply an alternative case where one

symbol is incorrect. On this occasion the vector **c** has been replaced by `NULL`, and so the threshold must yet again be $(n-1)w$.

The small example above shows that, as well as simplifying the architecture, using a `NULL` vector simplifies relaxation in the CANN. Any error is now equivalent to an incorrect symbol, and each grouping of equivalent errors is replaced with a single threshold value to use, meaning that a recall operation may be faster. For example, if relaxation up to group 1 is required by a cell in the original architecture then a total of 3 recalls and 4 thresholds must be performed. When using the `NULL` vector, only 1 recall and 2 thresholds are required to have the same effect.

5.4 Incorporating Scale Invariance

Apart from the most basic brute force technique—trying to match a pattern at numerous different scales—there are various ways in which scale invariance has been achieved in pattern recognition, using the detection of edges and interest points. These include the Generalised Hough Transform [12], graph matching [67], geometric hashing [114], or curvature scale space [72]. These use a range of analytical techniques, such as statistical and probabilistic models and Gaussian filtering. None of these methods are suitable for the distributed network of the CANN, however, as they use a global view of an image rather than the local view provided to each cell.

There are two obvious but impractical methods which may be used in order to incorporate scale invariance into the CANN in a neural and distributed manner, both a variant of the brute force method. The first requires training the CANN on multiple versions of the same pattern, presented at numerous different scales (within a predetermined range). This will increase the time required to initially train the network, but allow a recall to be performed quickly. Notably, however, it will significantly increase the number of rules generated—and hence the memory required to store those rules.

The second method trains only a single version of a pattern, presented at its original scale. A pattern must now be presented for recall at numerous different scales (within a predetermined range), in order that the CANN may find a match with the originally trained pattern. This minimises the number of rules generated and the memory required, however potentially imposes a great penalty on every recall performed.

A novel third method requires only a single version of each pattern to be trained, while minimising the performance penalty imposed by individually recalling a pattern

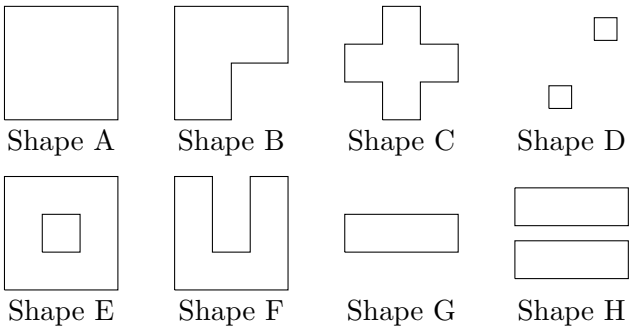


Figure 5.4: The 8 patterns trained into the scale invariant CANN [16]

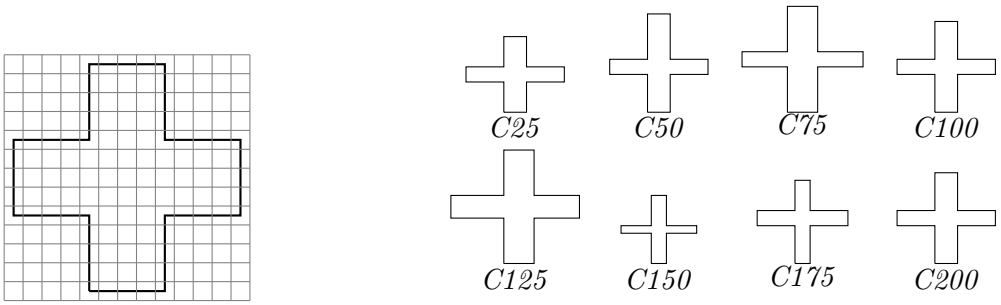
at numerous different scales [20]. As we saw in Section 2.3.3, Smolensky introduced the concept of a tensor product as a structure which stores bindings between variables and their values [98]. We have seen, in Chapter 4, that tensor products formed between input data and unique, randomly-generated, binary vectors may be superimposed and successfully recalled from a CMM. Using this technique, we can improve upon the second method by presenting a pattern for recall at numerous different scales simultaneously.

5.4.1 Recall

When recalling a pattern, the whole image is first scaled to each of the desired sizes—each of these images is assigned a unique binding vector. Primitives are extracted from each of the images in turn, and a tensor product is formed for each cell by binding this primitive to the image's binding vector. All of the tensor products for a given cell are then superimposed, and recall continues in the original fashion—in this case recalling each column of the tensor product in turn. If a pattern is recognised, it is possible to determine the scale at which it was found. Vectors remain in a tensor product throughout the operation of the system, which means that any assigned object labels are also in a tensor product. If this final tensor product is treated as a CMM, and the object label is presented as an input, then the output vector is the binding vector that was originally assigned to the scaled input pattern.

5.4.2 Initial Experimentation

In order to test the recall success of the scale invariant CANN, the 8 patterns used in Brewer's previous work [16] (shown in Figure 5.4) were trained into a CANN using the original method. Each pattern was symbolically encoded by overlaying a grid, as shown for Shape C in Figure 5.5a, and extracting the primitive features to be used as the input



(a) (b)

Figure 5.5: (a) Cellular grid used when extracting primitives from Shape C and (b) the input image closest to 100% of the original size of Shape C, for each resized version from *C25* to *C200*.

for each cell.

Each of the symbolically encoded shapes was then presented for recall at a range of different sizes—every 25% between 25% and 200% of the original size. We next selected a range of scales to use when recalling, such that the scale invariant mechanism would not simply return the images to the original size and result in a perfect recall. Each input shape (e.g. *C25*) was scaled to a range of sizes—every 50% between 50% and 400% of its new size (e.g. *C75* would have been rescaled such that the superimposed recall input ranged from 37.5% to 300% of the original size of Shape C, in steps of 37.5%). As the shapes were already in symbolic form before resizing, the primitive features are immediately available to be bound to their respective binding vectors.

Figure 5.5b shows the input image closest to 100% of the original size of Shape C, for each resized version from *C25* to *C200*. Resizing the shapes when in symbolic form, rather than as images, has introduced significant variation and distortion. In the next section, in order to achieve better results, images are scaled before the primitive features are extracted. For this work, however, the variation serves as an important test of the CANN's ability to recognise distorted shapes.

Table 5.2 shows the results obtained when presenting the eight shapes for recall at each of the eight scales. Each result shows firstly the label or labels applied to the shape after recall, and then the percentage of the input shape which was incorrectly labelled. In the majority of cases, the shape was correctly labelled, however there are a number of errors that warrant further examination.

A number of recalled shapes, namely various scales of B, F, and H, were labelled as both A and the correct label. Similarly, three of the scales of Shape G were incorrectly labelled

Shape	Scale of image presented for recall							
	25%	50%	75%	100%	125%	150%	175%	200%
A	A 0.00	A 0.00	A 0.00	A 0.00	A 0.00	A 0.00	A 0.00	A 0.00
B	B 6.38	B 0.00	AB 44.83	B 0.00	AB 43.55	AB 27.78	B 0.00	B 0.00
C	C 0.00	C 7.69	C 43.33	C 0.00	– 100.00	C 10.53	C 0.00	C 0.00
D	D 0.00	D 0.00	D 0.00	D 0.00	D 0.00	D 0.00	D 0.00	D 0.00
E	E 3.45	E 0.00	E 61.11	E 0.00	E 43.59	E 22.73	E 3.57	E 0.00
F	F 0.00	F 0.00	AF 62.86	F 0.00	AF 60.81	F 13.64	F 0.00	F 0.00
G	G 0.00	A 80.00	A 80.95	G 0.00	A 82.61	G 0.00	G 12.50	G 0.00
H	H 0.00	H 0.00	AH 52.38	H 0.00	AH 56.52	H 0.00	H 6.25	H 0.00

Table 5.2: Error rates of the scale invariant CANN, when recalling Shapes A–H presented at scales ranging from 25% to 200%. Each result consists of the label(s) applied to the shape after recall, as well as the percentage of incorrectly labelled symbols.

as Shape A. Given the similarities between these shapes, and the relaxation ability of the CANN, this is to be expected. This relaxation allows the CANN to recognise distorted and similar shapes, but can lead to incorrect recognition if two similar shapes are both initially trained into the CANN.

Presenting Shape C at a scale of 125% failed to result in any labels being applied. As can be seen in Figure 5.5b, the *C125* shape is the most distorted—being larger than at any other scale, as well as having different length arms. As mentioned earlier, this distortion could be reduced by scaling images before extracting the primitives.

5.4.3 Further Experimentation

In order to extend the results from Section 5.4.2, and to demonstrate the capabilities of the CANN without the additional hindrance caused by resizing symbolic shapes, the experiments have been repeated—this time scaling the images prior to extracting features. The images used are still artificially created line drawings, and so edge detection is not necessary in this case. Each image was scaled and OpenCV's facility for template matching was used to identify any line and corner segments within the input to each cell, creating the symbolic shapes ready to present for recall.

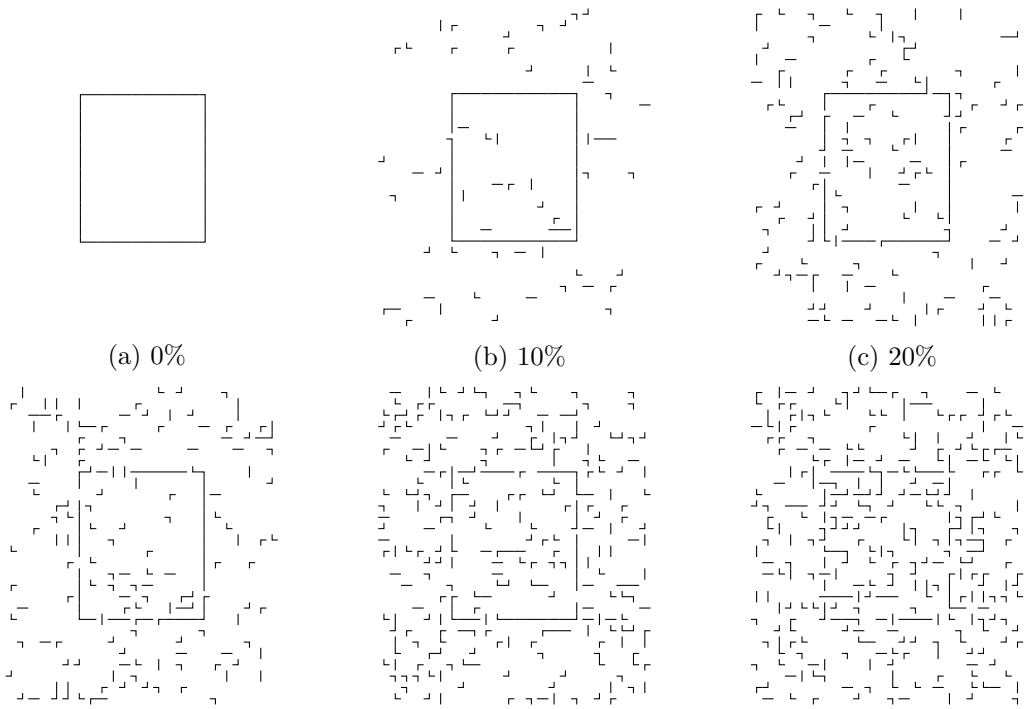
Table 5.3 shows the results obtained when presenting the eight shapes for recall at each of the eight scales. As with the previous results, each result shows firstly the label or labels applied to the shape after recall, and then the percentage of the input shape which was incorrectly labelled. In the majority of cases, the results are an improvement upon those presented in Table 5.2, however there are a number of errors that warrant further examination.

Shape	Scale of image presented for recall							
	25%	50%	75%	100%	125%	150%	175%	200%
A	A 0.00	A 0.00	A 0.00	A 0.00	A 0.00	A 0.00	A 0.00	A 0.00
B	B 0.00	B 0.00	B 4.00	B 0.00	B 5.77	B 2.63	B 26.19	B 0.00
C	C 0.00	C 0.00	C 13.79	C 0.00	C 3.85	C 0.00	C 0.00	C 0.00
D	D 0.00	D 0.00	D 0.00	D 0.00	D 0.00	D 0.00	D 0.00	D 0.00
E	E 0.00	E 0.00	E 29.41	E 0.00	E 42.11	E 22.73	E 12.50	E 0.00
F	F 0.00	F 0.00	F 51.52	F 0.00	AF 60.81	F 18.18	F 0.00	F 0.00
G	G 0.00	G 0.00	AG 52.38	G 0.00	AG 56.52	G 14.29	G 0.00	G 0.00
H	H 0.00	H 0.00	H 47.62	H 0.00	AH 56.52	H 0.00	H 12.50	H 0.00

Table 5.3: Error rates of the scale invariant CANN, using feature extraction, when recalling Shapes A–H presented at scales ranging from 25% to 200%. Each result consists of the label(s) applied to the shape after recall, as well as the percentage of incorrectly labelled symbols.

In these results all of the images were correctly labelled, however at a scale of 75% or 125% there were still a number of errors resulting in the shapes being labelled as both A and the correct label. The cause of these additional labels is once again the distortion introduced by scaling images, as well as the CANN's ability to relax and recognise inputs similar to those originally trained. Scaling the images prior to performing feature extraction has reduced the amount of distortion introduced, which in turn has reduced the number of additional labels.

The percentage of incorrectly labelled symbols has similarly decreased for almost all of the presented shapes, with a few notable exceptions. In these cases, for example 175% E and H, the shapes continued to be labelled correctly but with an increased error. This is actually caused by the cellular nature of the feature extraction process used—each cell of the CANN receives a predefined section of an input image with a fixed size, features are extracted from this section, and these are used to initiate a recall. At certain scales, features may in fact be separated across cell boundaries—for example a corner segment ⌟ may be divided into two cells, a horizontal bar adjacent to a vertical bar. In addition to this, features may be duplicated by the position of cell boundaries—a vertical bar that sits on the boundary between two cells may be recognised by each of the cells as a vertical bar. The CANN is clearly able to relax and cope with these errors, however the results could be improved even further by reducing them—possibly by using more loosely-defined cell boundaries, or by searching for a "best" positioning of the cells over an input image.



(a) 0% (b) 10% (c) 20%

(d) 30% (e) 40% (f) 50%

Figure 5.6: An example of the noisy inputs presented to the CANN for recall, where the subcaption denotes the probability of noise.

5.5 Noisy Inputs

The results presented thus far clearly show that the CANN is effective at recalling images at scale, even with the distortion introduced by this scaling. As such, the final test required is its performance in the presence of noisy inputs—similar to those experiments performed by Brewer [16], but with the added dimension of scale.

Given the intentional similarity between the shapes A–H used in previous experiments, it would be unreasonable to test noisy recalls using a CANN trained with all of the shapes. As such, for this experiment a CANN has been trained solely with shape A. The range of scales of shapes to be recognised remains the same as in previous experimentation—from 25% to 200% in increments of 25%. Similarly, the range of scales that recall inputs are presented at also remains the same—between 50% and 400% in increments of 50%. A new parameter, N, represents the level of noise that is applied to each cell's input—the probability of replacing a cell's input with a random feature or empty input.

The first stage of the recall follows the same procedure as used in Section 5.4.3, that is:

1. The input shape A is scaled to each of the range 25% to 200% in increments of 25%.

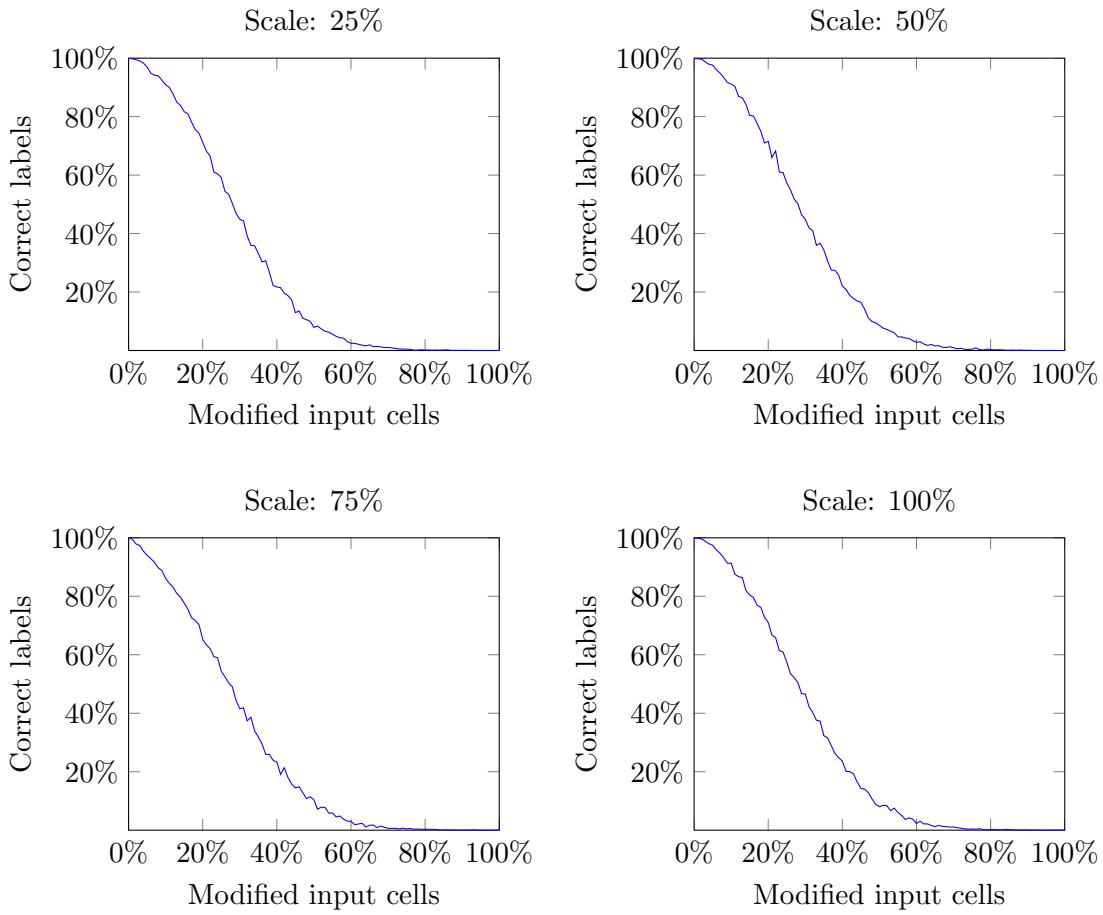


Figure 5.7: Plots showing the recall performance of the CANN with noisy inputs presented at scales 25%–100%.

2. Each scaled input is then resized by the scale invariant mechanism to each of the range 50% to 400% of their scaled size in increments of 50%.

3. Feature extraction is used to create symbolic images, ready for superposition and recall.

At this point, distortion is applied to the symbolic images for a range of N between 0% and 100%. For every cell, the input provided is a random feature (empty, |, —, ⌜, ⌝, ⌞, or ⌟) with probability N% or the extracted feature with probability $(100 - N)$%. Figure 5.6 shows an example of these noisy inputs at a single scale. Recall then proceeds as before, repeatedly iterating until a shape is recognised or no output is generated. As the noise is added probabilistically, each of these experiments has been run 100 times and the results averaged across all runs.

Figures 5.7 and 5.8 show the results of this experiment, for each different scale of input

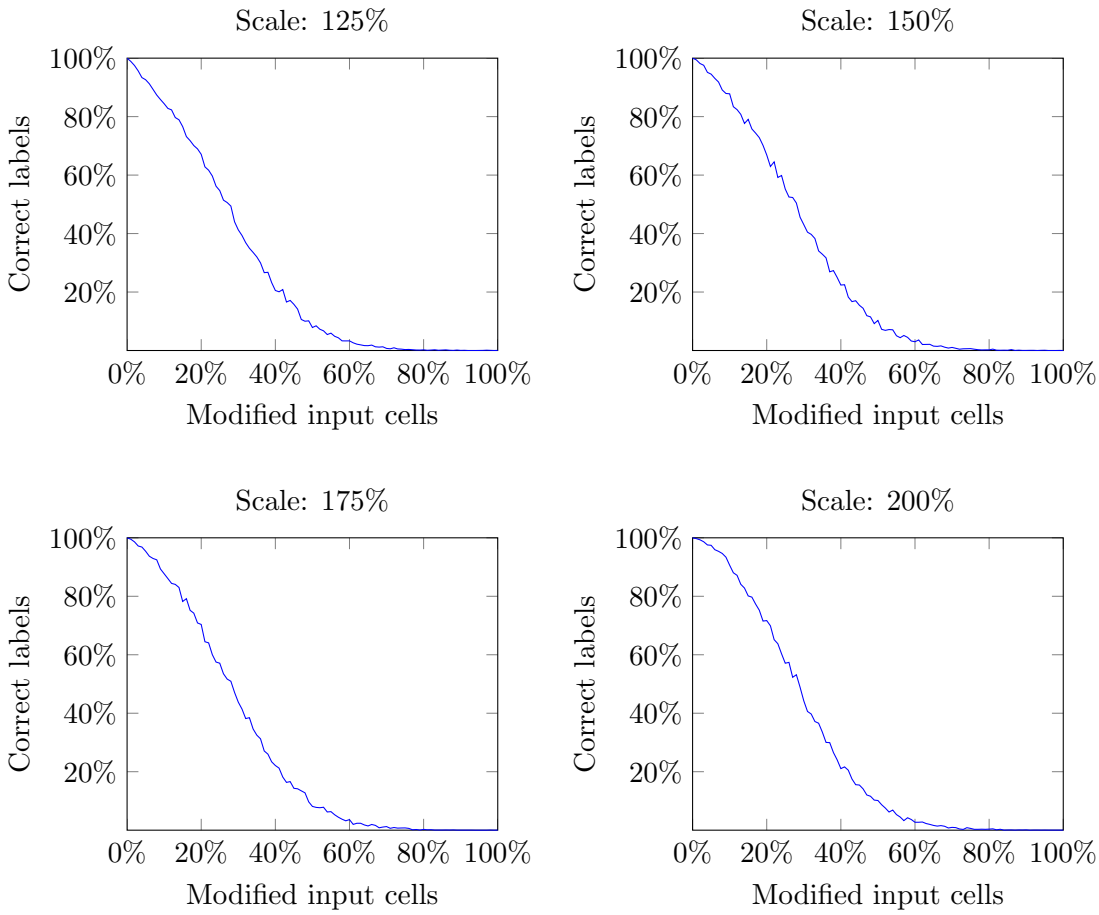


Figure 5.8: Plots showing the recall performance of the CANN with noisy inputs presented at scales 125%–200%.

for shape A. The graphs plot the percentage of cells which were correctly labelled after the iterative recall, against the probability of noise being applied to a cell's input. The plots are very similar between the different scales, showing that the scale invariant mechanism is working as expected even in the presence of noise. The only discernable difference between the 8 plots is the gradient of the curve at the top-left—the percentage of correctly applied labels decreases faster, as the probability of noise increases, on those scales which are most distorted from the original: 75%, 125%, and to a lesser extent 150% and 175%. This is to be expected, as the distortion and noise combine to make the problem harder.

After this initial curve there is no significant difference in the percentage of correctly applied labels in any of the plots, indicating that the level of noise has a far larger effect than the distortion caused by scaling. It should be noted that these results do not contain enough information to thoroughly evaluate the performance of the CANN. Only a single shape was trained into the network, and so if a label is applied at a particular cell then

it will inevitably be the correct one. Similarly no inputs were presented that would not be expected to recall successfully, and so there is no indication of a potential rate of "false positives". Finally, it is important to remember that a low percentage of cells being correctly labelled is not a direct indicator of a failed recall—even when the network was trained with 8 similar shapes, as in Table 5.3, it was able to correctly assign the final label with as few as 48.48% of the cells bearing the correct label.

5.6 Photographs

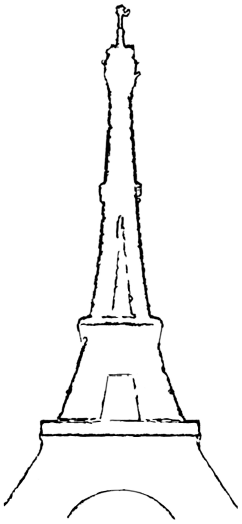
The majority of experimentation with the CANN has used synthetic images, with the exception of a brief demonstration by Brewer that the system is capable of recognising shapes within a photograph [16]. In order to show that further investigation of the network is warranted, this section applies the scale invariant CANN to photographic input using generic methods of feature extraction.

The CANN has been trained on an image of the Eiffel Tower, shown in Figure 5.9a. The edges were first extracted using the Canny Edge Detector [22] and extraneous edges caused by the trees removed manually, resulting in the image shown in Figure 5.9b. OpenCV's template matching was again used to extract a symbolic image, with each cell being given a 50×50 pixel section of the Canny Edge Detector output. Finally this symbolic image was trained into the CANN using the normal method described in Section 5.2.1.

The process for recall was similar, with the additional step required for the scale invariant mechanism. The tower in the recall image is approximately 21% the size of that in the trained image, and so the scales used during recall were chosen as the range from 50% to 1000% in increments of 50%. The recall image was scaled to each of these scales, creating inputs *tour-50*, *tour-100*, ..., *tour-1000*, and the features extracted from each using the Canny Edge Detector followed by OpenCV's template matching (again using 50×50 pixel sections as the input to each cell). A unique vector was generated for each of the 20 scales, and used to create a tensor product for each scaled input to each cell—finally these were superimposed to obtain the recall input. This input was recalled using the method described in Section 5.4.1.

Figure 5.9c shows the recall image, with the results of this recall superimposed. Each green square denotes that the cell at that position labelled its output correctly—in this case at a scale of 500%, as was expected. It is clear that, on the whole, the recall was successful—with a large proportion of expected labels being correctly applied, and only

(a) Original image to train [100]



(b) Training image after application of the Canny Edge Detector [22]

(c) Recalled image [105], with cells that recognised the shape marked in green

Figure 5.9: Application of the scale invariant CANN to photographs of the Eiffel Tower. The tower in the trained image is approximately 4000 pixels tall, whereas in the recalled image it is approximately 850 pixels tall.

a few cells incorrectly applying a label. Although this does help to justify continued development of the CANN, there are a number of factors of this experiment that must be emphasised.

Firstly, the choice of scales was deliberate—knowing that the target shape was scaled down to 21% of the trained image, a range was selected such that it would be returned to a similar size as the original. Secondly, the CANN was trained with only a single image, and similarly a single image was recalled—this provides no information about the ability of the CANN to distinguish between shapes, nor does it test for false positives. Finally, the use of the Canny Edge Detector and OpenCV's feature matching provides the opportunity to tune these algorithms to provide a comparatively clean input to the CANN. A neural-inspired edge or feature detector may not provide such an ideal output, which would inevitably affect the CANN's capabilities.

5.7 Further Work

Although the CANN has been shown to be able to perform scale and translation invariant pattern recognition, there are a number of areas which require further work or warrant further investigation.

5.7.1 Image Scaling

Experimental results have shown that the CANN is capable of performing scale invariant pattern matching effectively, but that the sometimes extreme distortion introduced by image scaling can reduce its accuracy. It has been shown that this distortion can be reduced by scaling images prior to extracting primitives, rather than scaling the symbolically encoded images. Further work should also experiment with the number and range of scales used during a recall, and attempt to determine whether there is an optimal set of scales to use. In addition to this, using "fuzzy" cell boundaries or searching for an improved positioning of the cells over an input image may be able to reduce the distortion and improve the results further.

5.7.2 Rotation Invariance

Ideally, a pattern recognition system should be able to recognise objects invariant to translation, scale, and rotation. The CANN provides translation invariance due to its cellular

nature, and has been extended to provide scale invariance through the superposition of scaled recall images. Rotation invariance has not been attempted, however the tensor product mechanism used for scale invariance is a general purpose solution and so should be able to be used for any transformation—including rotation, skew, and shearing.

5.7.3 Primitive Features Used

Work with the CANN has so far always used the same 6 primitives: |, —, ⌜, ⌝, ⌞, and ⌟. Further investigation is warranted into the effect that changing these primitives may have on the CANN. The use of a single primitive—that denotes a cell as containing an edge or not—may reduce the number of rules generated, and possibly increase the ability to generalise, however it may also affect the ability of the CANN to distinguish similar classes of patterns.

Alternatively, a number of additional primitives may be introduced. The lack of diagonal line primitives, / and \, means that angled edges will most likely be represented by the various corner shapes. Additionally, the architecture of the CANN currently allows only a single primitive to be present in each cell, which limits the potential shapes which may be stored. For example, there is no "T"-shape primitive, meaning that this shape is likely to be represented by a corner shape—or possibly by a horizontal bar in one cell, and a somewhat disconnected vertical bar in the cell below. Incorporation of further primitives may help the CANN to distinguish similar shapes, however it will increase the difficulty of initial primitive extraction.

5.7.4 Superposition of Primitives

Rather than increasing the number of primitives, modifying the architecture of the CANN in order to allow more than one primitive to be present in a cell may be an alternative solution. Figure 5.3b shows the primitives which may be extracted from a "T"-shape. Currently the CANN must use only one of these, which means information must be thrown away. If an arity network similar to that described in Section 4.5 was used in the CANN, then these primitives could all be used.

Only a single transition symbol is created for each rule by the CANN's learning process, and so it is not possible for module inputs after the first iteration to contain superimposed information. Thus, every iteration except the first can be recalled from a single CMM. This helps to minimise the impact that introducing an arity network may have on the

speed of a recall—usually an input must be recalled from every CMM of the arity network in turn.

5.8 Summary

In summary, two significant improvements to the CANN have been presented. The architecture has been streamlined by replacing the "arity network" within each module by a single CMM. Using a `NULL` vector to represent an empty input provides equivalent pattern recall ability as the original architecture, as well as providing simplified and faster relaxation.

A mechanism to provide scale invariant pattern matching has also been incorporated, using tensor products and superposition of scaled recall images. Results have been presented that demonstrate that it is capable of performing this task effectively, but that further work is required in order to experiment with the number and range of scales used during a recall in order to reduce the distortion introduced by scaling of images. Finally an experiment has been performed to test the CANN's ability to successfully recall from photographs, using generic feature extraction.

Chapter 6

Conclusions

6.1 Introduction

This chapter brings the work presented in this thesis to a conclusion. Chapter 1 described the problems with pattern recognition using correlation matrix memories—firstly a lack of scalability due to memory usage, followed by significant limitations in the capabilities of the Associative Rule Chaining Architecture (ARCA) and the Cellular Associative Neural Network (CANN). Finally the aims of the project were presented, based on the identified limitations.

Chapter 3 presented a domain specific language for use with binary correlation matrix memories (CMMs), and most importantly developed an interpreter for this language that uses an efficient storage mechanism to allow the memory requirements of correlation matrix memories to scale linearly with the number of associations stored.

Chapter 4 examined ARCA in detail, firstly describing the original rule chaining architecture, before showing that the work in Chapter 3 successfully resolved the exponential increase in ARCA's memory requirement that was seen previously as the number of rules increased. A modification to the architecture was proposed, which reduced the memory requirements yet further, before presenting a solution to the problem of multiple arity rules—allowing ARCA to successfully store and operate on any directed acyclic graph or forest of rules. Finally a method was introduced which allows the path between two nodes to be found, greatly increasing the potential domains to which ARCA may be applied.

Chapter 5 focused on the CANN, detailing the existing pattern recognition architecture and describing a limitation created by its use of arity networks. The architecture was adapted to allow these networks to be removed, before showing that the CANN retains its

ability to perform relaxation—an important part of its recall process. Lastly a mechanism to allow the CANN to perform invariant pattern recognition was introduced, and applied to scale invariance with symbolically encoded patterns.

The following sections bring together the results and conclusions of this thesis, evaluating the degree to which the work completed meets the motivations and aims of the project before finally summarising any further work which has been identified.

6.2 The Extended Neural Associative Memory Language

The Extended Neural Associative Memory Language (ENAMeL) domain specific language, described in Chapter 3, was created with an intent to simplify future development of architectures based on binary CMMs. More importantly, the interpreter was developed to use an efficient storage mechanism in order that the memory requirement of CMMs may be reduced and larger architectures may scale more effectively. Simplification of development is subjective and unquantifiable, and so this section will concentrate on the storage mechanism.

Although CMMs offer fast learning and recall, a perception that they are memory-intensive has helped to limit their use. Compression algorithms such as run-length encoding, Huffman coding [56], or LZ77 [117] are not well suited for the storage of CMMs because of a lack of fast access to rows and cells of the matrix—instead the matrix would need to be decompressed before use, and recompressed if any changes were made.

As a result, the work focused on the use of sparse storage—a binary version of the Yale format [30] and a hybrid storage which can use either non-sparse storage or the binary Yale format for each individual row. Experimentation showed definitively that a hybrid approach provides the most efficient memory usage of those tested—allowing the memory to increase linearly with the number of vector pairs associated, while providing an upper bound on the usage.

Experimentation also showed that the ideal weight of vectors associated in a CMM, in order to achieve maximum capacity, is not always $\log_2 l$ (where l is the vector length). This is widely asserted, but the results show that this is not always the case. When using Baum vectors, and where the weight of input and output vectors may vary, the maximum capacity is achieved by using the minimum output vector weight (2) and an input vector weight higher than $\log_2 l$ (between 13 and 15 with a vector length of 1024). This deserves further investigation, as well as experimentation on CMMs storing randomly-generated

vectors, to determine if it is possible to calculate the ideal input weight for a CMM in advance.

In summary, the work performed on ENAMeL—and particularly on the hybrid storage format—provided a foundation that allowed the remaining work of this thesis to be undertaken. As was seen in Chapter 4, the sparse storage format can help the memory requirement of architectures that use CMMs to grow linearly, rather than exponentially, with problem size.

6.3 The Associative Rule Chaining Architecture

Chapter 4 focused on the Associative Rule Chaining Architecture (ARCA), a system which performs forward chaining and can examine all the branches of a tree of rules simultaneously using superimposed tensor products. Although tensor products have been shown to be limited in their direct application to certain cognitive tasks, when incorporated into a larger architecture they are more than capable of performing this activity. Only a very restricted form of productivity is required in ARCA, and the matrix is fixed at two dimensions—additional vectors are superimposed rather than bound to create extra dimensions. Some of the difficulties identified in other uses of tensor products are therefore avoided, allowing ARCA to remain feasible. The aim of this work was to resolve both the problem of exponential memory growth and the lack of support for multiple arity rules identified in previous work [9]. These improvements are intended to allow ARCA to scale effectively, and to be applied to a wide range of problems—demonstrated by applying ARCA to Solitaire in Section 4.7. Each of the improvements is now addressed individually, firstly summarising the work before discussing the results and any conclusions that can be drawn from them.

6.3.1 Addressing the Memory Requirement of the Architecture

Before the memory requirement could be addressed, it first needed to be understood more clearly. Hobson [51] plotted the memory required against the depth of the tree of rules, which provided very little insight into ARCA's performance with a branching factor greater than 1. This stems from two causes. Firstly, the number of rules stored for a given depth and branching factor can vary very widely. For example with a depth of 5 and branching factor of 2, the number of rules can range from 5 (if all nodes have 1 child) to 31 (if all

nodes have 2 children). More importantly, for any branching factor greater than 1, the number of rules grows exponentially with the depth of the tree and so it is impossible to determine whether the memory growth is largely as a result of this or is due to other causes.

Improved results were presented in Section 4.3.4 to resolve this difficulty, plotting the memory required against the number of rules. These results clearly showed that the architecture was causing an exponential increase in memory as the number of rules increased linearly, and that the memory requirement needed to be addressed.

The first modification was thus to use the efficient storage mechanism provided by ENAMeL. These results were presented in Section 4.3.5, and clearly show that the growth in the memory requirement has been reduced to linear with respect to the number of rules. In addition to resolving ARCA's scaling problem, this validates the work on storage mechanisms within ENAMeL in Chapter 3 and shows that the use of an efficient storage mechanism for CMMs can greatly improve architectures using them.

ARCA's high memory requirement was not solely caused by inefficient storage. Section 4.4 presented a modification to the architectural design, intended to help further by reducing the number of CMMs used—mapping directly from antecedents to consequents in a single CMM. This was expected to provide a small improvement to the memory usage; moving from two CMMs with dimensions $l_t \times l_r$ and $l_r \times l_t l_r$ to a single CMM with dimensions $l_t \times l_t l_r$, where l_t and l_r are the token and rule vector lengths respectively. The results obtained were therefore surprising, showing that even when using non-sparse storage the growth in memory is reduced to almost linear.

The memory reduction displayed shows that the exponential growth was compounded by the original design of the architecture. The antecedents and consequents of a rule were separated into two CMMs, connected by the rule vector. The design required the output of the second CMM to be a tensor product, in order that superimposed states would remain distinguishable, however this causes a large difference in the dimensions and capacities of the two CMMs. The antecedent CMM reaches saturation far more quickly than the consequent CMM, meaning that the vector lengths must be increased—affecting the size of both CMMs. By mapping directly from antecedents to consequents in a single CMM, this problem is avoided and the CMM's capacity can be exploited more fully.

In summary, ARCA originally displayed exponential memory growth as the problem size increased. The modifications demonstrate that this limitation of ARCA can be re-

solved, allowing the memory requirement to increase linearly with the number of rules. This is an important step, as it allows ARCA to remain applicable to larger scale problems with a high branching factor or containing millions of rules.

6.3.2 Incorporating Multiple Arity

Arity networks were proposed as a solution to the problem of multiple arity rules by Austin et al. [10], and their suitability has been reiterated in later work [9, 51]. In Section 4.5 I demonstrated a clear problem with the use of arity networks—although they seem to resolve the problem of assigning a threshold value during a recall, there is still the possibility of an overlap between superimposed vectors and so the difficulty remains.

Weight networks are an effective alternative solution, separating rules by the combined weight of their superimposed antecedents rather than simply by the number of antecedents. These allow ARCA to perform rule chaining correctly. The results also showed that the linear relationship remains between the memory usage and the number of rules stored. The required memory is higher than that needed by an equivalent number of single arity rules, however, even when adjusted to take account of the additional bits which must be stored due to the higher input and output weights.

Due to the success in reducing the memory requirement found with the single CMM variant of ARCA, the multiple arity work focused on incorporating weight networks into this variant. It is possible, however, that the original architectural design may be better suited to this application. In the two CMM variant of ARCA, antecedents and consequents are separated into two CMMs and connected using the rule vector. This means that in order to support multiple arity, the weight networks solution need only be applied to the antecedent CMM. After recalling an input from each of the CMMs forming this "antecedent weight network", the results can be superimposed and then recalled from the single consequent CMM.

The original problem identified with ARCA's design is that the antecedent CMM reaches saturation while the consequent CMM is still very sparse—meaning that the available storage is utilised inefficiently. By separating the antecedents of rules with differing input weights into multiple antecedent CMMs, this limitation may be alleviated—allowing vector lengths to remain shorter, and the consequent CMM to be better utilised. There is, however, a caveat which must be considered—if the weight of the superimposed antecedents of a set of rules is clustered around only a few of the possible weights, then

saturation is still likely to be a problem.

To summarise, this work identified a problem with arity networks and proposed a solution to this difficulty. The incorporation of weight networks has been shown to successfully allow ARCA to perform rule chaining with multiple arity rules, importantly maintaining a linear relationship between the number of rules stored and the memory required. The work focused on the single CMM ARCA, however a comparison between weight networks implemented in both of the variants may be interesting.

6.3.3 Demonstrating Pathfinding

When rule chaining is used in an expert system the task is to infer knowledge, given a number of facts that are asserted as being true. The path taken during inference is unimportant, only the result is relevant. This is not the case for all applications of rule chaining; in some situations finding the path is the desired outcome. This usage has not been addressed in previous work on ARCA, and so a demonstration that pathfinding is possible shows that ARCA is suitable for further applications.

Section 4.6 introduced a mechanism through which pathfinding can be implemented using ARCA, with a simple demonstration of its operation. An experimental simulation should be performed in future work, in order to determine if the mechanism employed has any effect on the capacity of ARCA—although the architecture is not modified, successful recall from the transposed CMM will be affected by its level of saturation.

6.3.4 Summary

Prior to this work it was demonstrated that ARCA was capable of performing rule chaining [51], however significant limitations were identified that meant the architecture was unsuitable for application to all but the smallest and simplest of problems. Chapter 4 presented modifications designed to resolve these difficulties, specifically reducing the growth in memory usage from exponential to linear as the problem size increases and providing a mechanism to allow ARCA to store and recall multiple arity rules. These changes allow the architecture to be applied to any directed acyclic graph, and to successfully scale to larger problem sizes. The work demonstrated that ARCA can be adapted to pathfinding, in order to allow further applications such as the control of autonomous agents, before finally demonstrating that the architecture can successfully scale to the problem of Solitaire—involving 185.9M rules—reducing the time complexity of finding a solution to a given state

by up to 4 orders of magnitude compared to a DFS. A number of areas for further work have been identified, and are summarised in Section 6.6.

6.4 The Cellular Associative Neural Network

Chapter 5 focused on the Cellular Associative Neural Network (CANN), an architecture designed to perform syntactic pattern recognition. The aim was to develop the architecture to allow it to perform pattern recognition in a generally invariant manner, so that it becomes suitable for application to a large range of pattern recognition problems. The improvements made are now addressed individually, firstly summarising the work before discussing the results and any conclusions that can be drawn from them.

6.4.1 Simplifying the Architecture

Before the CANN could be adapted to support scale and rotation invariance, an important limitation of the architecture needed to be resolved to allow the integration of tensor products for the purpose of superposition. The original design used a form of arity network within each cell in order to avoid unwanted partial matching of rules. In order for the arity network to be successful in this aim, an input can only be recalled from the "correct" CMM—the CMM which stored rules with the input's arity. This works as expected if a single input is used, however if inputs are superimposed then it is impossible to know which is the correct CMM from which to recall—and in fact the arities of superimposed inputs may differ.

Section 5.3.1 introduced a solution to this problem, allowing all of the rules to be stored in a single CMM. Rather than storing rules with differing arities in separate CMMs, any "missing" antecedents are substituted with a `NULL` vector—meaning that the number of antecedents of all rules is identical, and the threshold value is simply the number of antecedents multiplied by the weight of an antecedent vector.

One of the CANN's most important features is its ability to perform relaxation on a local scale, providing tolerance for distortion and occlusion in input images. Showing that the architecture is still able to operate in this fashion is therefore imperative for the `NULL` vector to be a viable alternative to arity networks. Section 5.3.2 showed that there is a direct mapping between the relaxation provided by the original CANN, and that of the simplified architecture. As an additional benefit, each "class" of relaxation can be satisfied

with the application of only a single additional threshold—for comparison, in the original architecture class 1 relaxation (a single erroneous antecedent) required an additional two recalls and three threshold applications. This has the potential to greatly increase the speed of a recall operation, especially in the case of a noisy or distorted input where relaxation may be required often.

6.4.2 Addition of Tensor Products

Many common statistical techniques for scale or rotation invariance are not suitable for integration into the CANN, as they use a global view of an image rather than the local view provided to each cell. As a result, the remaining options are variations of the brute force method—either training or recalling a pattern at multiple scales or rotations. The former method (training) can significantly increase both the time required to train the network and the number of rules generated—and hence the memory required to store them. Most importantly, this method requires knowing in advance at which scales and rotations a pattern should be stored—for example if a pattern is stored at a range of scales from 1 to 10 but recalled at a scale of 20, then it will not be recognised. The latter method (recall), on the other hand, can significantly slow the process of a recall operation as it requires repeated presentation of an input until a result is found.

With all the rules stored in a single CMM it becomes possible to perform superimposed recall, as it is not required to know the arity of an individual input. Superposition in the CANN would, however, experience the same difficulties as were discussed in Section 2.3.5.1—namely an inability to distinguish between different superimposed outputs. Tensor products are used in Section 5.4 in order to resolve this problem, allowing the CANN to perform superimposed recall.

The results showed that the technique works, specifically when applied to scale invariance, but there are some limitations. In particular, when a pattern was presented for recall at 75% or 125% of the original size, a number of shapes were incorrectly labelled or received multiple labels. Given the similarity between a number of the shapes used, this is not surprising—the errors are almost entirely the mistaken identification of a shape very similar to the target. An improved design for future experiments would store distinct patterns, or possibly a single pattern.

Another problem identified is the distortion caused to an input pattern due to the scaling mechanism that was initially used. Although some distortion when enlarging or

shrinking an image is inevitable, performing the scaling prior to extracting the pattern's basic features causes fewer problems than scaling after the feature extraction. On the other hand, the often extreme distortion in the initial experimentation provides an excellent example of the CANN's ability to use relaxation at a local level in order to recognise inputs similar to patterns which have been previously trained. The later experiments used a generic method to perform the feature extraction of scaled images, and demonstrated improved results due to the reduced distortion.

The final point which may be perceived as a limitation of the experimentation is that the results are not compared with alternative scale-invariant pattern recognition techniques. At its current stage of development, however, a comparison with a well-developed statistical technique such as the Generalised Hough Transform would not be informative. With the addition of a neural-inspired preprocessor able to reliably extract basic features for input to the CANN, future work might practically perform a comparison with other neural-inspired techniques such as the Neocognitron.

6.4.3 Summary

The CANN has previously been shown to be effective at syntactic pattern recognition on simple line drawings [77], and successful in limited experimentation with photographic inputs [16]. The architecture was restricted in its applicability to general pattern recognition problems, however, due to a lack of support for scale or rotation invariance. Chapter 5 presented improvements to the architecture intended to provide this support, firstly simplifying the design to allow for superposition and then integrating tensor products to maintain a separation between superimposed recall states. These modifications allow the CANN to be generally invariant during pattern recognition—for example invariant to scale, rotation, skew, or stretch, in addition to the built-in translation invariance. There are a number of areas deserving further attention, these are presented in Section 6.6.

6.5 General Conclusions

The hypothesis of this project was that CMM-based architectures can be effective and proficient for use with rule-based systems, demonstrating efficient memory usage and potential to scale. The work on ENAMeL presented in Chapter 3 showed that CMMs can be stored in a memory efficient manner, although there remain a number of questions relating

to the best selection of parameters for vector generation. The results gathered when using the hybrid format with ARCA in Chapter 4 further supported the hypothesis—the growth in the memory requirement of this architecture was reduced to a linear relationship with regard to the number of rules. Further investigation into integrating compression algorithms into the hybrid format is warranted, as this may also help to reduce the memory usage further without having a debilitating effect on the speed of operations.

The other aim of this work, in support of the main hypothesis, was to resolve the problems previously identified in the ARCA and CANN rule-based systems. To this end, Chapter 4 presented a resolution for a number of ARCA's limitations—showing that the architecture could be modified in order to allow the effective storage of any directed acyclic graph or forest of rules in a scalable and memory-efficient manner. Further work has been identified, however, such as investigating the effect of changing the vector lengths and weights or integrating loop detection to allow ARCA to operate correctly on cyclic graphs.

Finally, Chapter 5 revised the architecture of the CANN to allow generally invariant pattern recognition to be performed. Previously the architecture was significantly limited in this regard, as its ability stretched only to translation-invariant pattern recognition. The experimental results showed that the architecture can be suitable for application to general pattern recognition problems, however further investigation into the pattern primitives is warranted and may improve the network's ability to distinguish similar patterns. Development of a suitable preprocessor able to extract primitives from an image would allow a thorough evaluation of the architecture's capabilities.

In summary, the work presented in this project demonstrates that the hypothesis was correct—CMM-based architectures can demonstrate efficient memory usage and scale effectively. By making a number of important modifications, the ARCA and CANN systems can be made suitable for their intended applications: rule chaining and pattern recognition.

6.6 Further Work

A number of areas have been identified throughout the thesis as deserving further investigation. These are summarised in the following sections.

6.6.1 The Extended Neural Associative Memory Language

Although ENAMeL is complete, and the hybrid storage format has been shown to provide reasonable memory efficiency while still allowing fast access to individual rows and bits of a CMM, there are a number of unanswered questions that have developed during experimentation.

Firstly, an in-depth analysis of the time complexity of operations given different storage mechanisms would allow the choice of storage to be guided by both memory and time constraints. As an extension to this, compression algorithms may be able to be partially integrated into the hybrid format. Considering run-length encoding (RLE) as an example: the use of RLE on an entire CMM imposes the unreasonable requirement that the CMM must be decompressed from the beginning in order to access an individual bit. Using RLE within the hybrid format, however, would reduce the performance penalty as the CMM would only need to be decompressed or recompressed on a row-by-row basis. Depending on the level of compression achieved, providing an option to trade some execution speed for memory usage may become appropriate.

The choice of vector weight is very important in determining the capacity of a CMM; affecting both the number of correlations stored within the CMM for every associated vector pair and the threshold used during a recall. Section 3.3.3 showed that using $\log_2 l$ as the vector weight, where l is the vector length, does not provide the highest possible capacity when input and output vectors can have different lengths. Experimentation has shown that a minimal output vector weight results in the highest CMM capacity, however further work is required to fully understand the relationship between the input vector weight and the capacity of a CMM when vectors are generated using a method such as Baum's algorithm.

In Chapter 3 CMMs were trained with synthetic data, in an attempt to determine their capacity or memory requirement. When CMMs are applied to a real problem, the required capacity is likely to be determined in advance and the memory requirement may be a constraint. Determining the vector parameters to use in order to achieve this capacity is a difficult problem. An approximation of a CMM's capacity can be obtained using equations in [11] or [104], however these calculate the capacity based on randomly generated vectors rather than those created using an algorithm that attempts to increase the orthogonality between vectors—and hence the capacity of a CMM—such as Baum's algorithm. A complete mathematical analysis of the capacity of a CMM storing Baum

vectors could be of great use, creating a model that estimates the capacity of a CMM with given input and output vector lengths and weights. In order to be most useful this should then be modified to work from the opposite angle: a user would give the target capacity as an input, as well as any constraints—for example input and output vectors must be the same length—and the model would suggest vector parameters to use in order to achieve this capacity.

As an extension to the mathematical analysis of CMM capacity, modifying the model in order to probabilistically determine the interactions and overlaps between vector pairs associated within a CMM would allow it to be used to estimate the memory requirement of a CMM using different storage mechanisms.

6.6.2 The Associative Rule Chaining Architecture

The work in Chapter 4 showed that ARCA is capable of performing rule chaining effectively, with a linear relationship between the number of rules stored and the memory requirement. Further work is still required, however, particularly an investigation into the relationship between vector lengths and weights and ARCA's capacity. The lengths and weights of vectors used to represent tokens and rules in ARCA are very important, as these determine the capacity and memory requirement. The weight chosen in this work may not be optimal, and so this needs further investigation. Similarly, the relationship between the token vectors and rule vectors is not well understood—further experimentation of varying their lengths and weights independently may yield performance or capacity improvements.

ARCA is able to perform rule chaining with any directed acyclic graph, tree, or forest of rules, whether the rules are single or multiple arity. It is not ideal for use with cyclic graphs, however, due to the stopping condition employed. In an unsuccessful recall, the stopping condition is that the output of an iteration is all zeros. If a loop is encountered, for example with two rules $\mathbf{a} \rightarrow \mathbf{b}$ and $\mathbf{b} \rightarrow \mathbf{a}$, then an unsuccessful recall will never end—recall operations will only complete if the goal state is reached. Loop detection may therefore need to be incorporated if ARCA is to be used with cyclic graphs.

The pathfinding presented in Section 4.6 is limited to unweighted edges—essentially finding the smallest number of "hops" between two nodes, where traversing any edge carries the same cost. It may be possible to find an encoding to be used for the rule vectors, in order to incorporate weighted edges into ARCA. If the system is also extended to fully support graphs then ARCA would be able to find the shortest path between two

nodes, providing a fast, neural-inspired alternative to Dijkstra's Algorithm or A*.

6.6.3 The Cellular Associative Neural Network

In Chapter 5 the CANN was modified to allow it to perform scale and translation invariant pattern recognition, however there are still a number of areas which require further work or warrant further investigation.

Experimental results showed that the CANN is capable of performing scale invariant pattern matching effectively, but that the sometimes extreme distortion introduced by image scaling can reduce its accuracy. It has been shown that this distortion can be reduced by scaling images prior to extracting primitives, rather than scaling the symbolically encoded images. Further work should also experiment with the number and range of scales used during a recall, and attempt to determine whether there is an optimal set of scales to use. In addition to this, using "fuzzy" cell boundaries or searching for an improved positioning of the cells over an input image may be able to reduce the distortion and improve the results further.

Ideally, an object recognition system should be able to recognise objects invariant to translation, scale, and rotation. The CANN provides translation invariance due to its cellular nature, and has been extended to provide scale invariance through the superposition of scaled recall images. Rotation invariance has not been attempted, however the tensor product mechanism used for scale invariance is a general purpose solution and so should be able to be used for any transformation—including rotation, skew, stretch, and shearing. An investigation into the effectiveness of the tensor product technique would be an important step towards demonstrating the suitability of the CANN for pattern recognition.

Work with the CANN has so far always used the same 6 primitives: |, —, ⌜, ⌝, ⌞, and ⌟. Further investigation is warranted into the effect that changing these primitives may have on the CANN. The use of a single primitive—that denotes a cell as containing an edge or not—may reduce the number of rules generated, and possibly increase the ability to generalise, however it may also affect the ability of the CANN to distinguish similar classes of patterns. Alternatively, a number of additional primitives may be introduced. The lack of diagonal line primitives, / and \, means that angled edges will most likely be represented by the various corner shapes. Additionally, the architecture of the CANN currently allows only a single primitive to be present in each cell, which limits the potential

shapes which may be stored. For example, there is no "T"-shape primitive, meaning that this shape is likely to be represented by a corner shape—or possibly by a horizontal bar in one cell, and a somewhat disconnected vertical bar in the cell below. Incorporation of further primitives may help the CANN to distinguish similar shapes, however it will increase the difficulty of initial primitive extraction.

Rather than increasing the number of primitives, modifying the architecture of the CANN in order to allow more than one primitive to be present in a cell may be an alternative solution to this problem. Instead of creating a "T"-shape primitive, the shape could be represented by the superposition of both a horizontal bar and a vertical bar. This would require the introduction of a "weight network" into the CANN, in order that rules with differing numbers of primitives could be successfully stored and recalled. Only a single transition symbol is created for each rule by the CANN's learning process, and so it is not possible for module inputs after the first iteration to contain superimposed information. Thus, every iteration except the first can be recalled from a single CMM. This helps to minimise the impact that introducing an arity network may have on the speed of a recall—usually an input must be recalled from every CMM of the arity network in turn.

Finally, with the inclusion of a preprocessor to extract primitives from an image, an experimental evaluation of the CANN's effectiveness should be performed. This may be executed as a comparison against alternative neural-based pattern recognition architectures, such as the Neocognitron, or using a benchmark set of images for a specific application such as handwritten digit recognition.

Appendix A

ENAMeL Code

This appendix presents working examples of ENAMeL code, along with equivalent MATLAB versions to aid understanding. Firstly a minimal example of ENAMeL is given, followed by a sample of code used when experimenting with ARCA.

A.1 Minimal Example of ENAMeL Code

This is a minimal example of ENAMeL code, creating two vectors and associating them in a matrix using the two different methods.

enamel.txt

```
# A Minimal Example of ENAMeL Code

# Create an arbitrary vector, length 512-bits and weight 2
#   (bits 3 and 258 set to 1)
a = Pattern 512 3 258

# Create a Baum vector generator, for vectors of length 512 and weight 2
Generator 255 257
# Create a Baum vector, length 512-bits and weight 2
#   (bits 0 and 255 set to 1, as this is the first vector in the series)
b = Vector 2 512

# Create a 512x512 matrix, containing a:b
M1 = a:b

# Create an empty 512x512 matrix
M2 = Matrix 512 512
```

```
# Store b:a in M2
M2 = M2v(b:a)

# Recall a from M1
r = a.M1
# Print r to stdout (prints 'r: 0(2) 255(2)')
Print r
# Apply Willshaw's threshold with a value of 2
r = r|2
# Print r to stdout (prints 'r: 0 255')
Print r

# Recall b from M2, applying Willshaw's threshold
r = (b.M2)|2
# Print r to stdout (prints 'r: 3 258')
Print r

# Exit from ENAMeL
Exit
```

A.1.1 MATLAB Equivalent

createVector.m

```
% Create an arbitrary vector
function v = createVector(l, bits)
    v = zeros(l, 1);
    v(bits) = 1;
end
```

createBaumVector.m

```
% Create a Baum vector
function v = createBaumVector(l, coprimes, num)
    v = zeros(l, 1);
    bits = ones(1, length(coprimes)) + [0 cumsum(coprimes(1:end-1))] ...
           + mod(num, coprimes);
    v(bits) = 1;
end
```

createMatrix.m

```
% Create a matrix
function m = createMatrix(inlength, outlength)
    m = zeros(inlength, outlength);
end
```

trainMatrix.m

```
% Train a matrix
function M = trainMatrix(M, input, output)
    M(:, output == 1) = M(:, output == 1) | repmat(input, 1, sum(output));
end
```

recallMatrix.m

```
% Recall from a matrix
function v = recallMatrix(input, M)
    v = sum(M(input == 1, :))';
end
```

willshawThreshold.m

```
% Apply Willshaw's threshold
function v = willshawThreshold(v, value)
    v(v < value) = 0;
    v(v > 0) = 1;
end
```

enamel.m

```
% A Minimal Example of ENAMeL Code, converted to MATLAB

% Create an arbitrary vector, length 512-bits and weight 2
%   (bits 4 and 259 set to 1 [1-based indexing])
a = createVector(512, [4 259]);

% Create the 1st Baum vector, length 512-bits and weight 2
%   (bits 1 and 256 set to 1, as this is the first vector in the series)
b = createBaumVector(512, [255 257], 0);

% Create a 512x512 matrix, containing a:b
M1 = trainMatrix(createMatrix(512, 512), a, b);

```

```
% Create an empty 512x512 matrix
M2 = createMatrix(512, 512);
% Store b:a in M2
M2 = trainMatrix(M2, b, a);

% Recall a from M1
r = recallMatrix(a, M1);
% Print r to stdout (prints a 512-length vector containing twos at
%   positions 1 and 256 [1-based indexing] and zeros otherwise)
r
% Apply Willshaw's threshold with a value of 2
r = willshawThreshold(r, 2);
% Print r to stdout (prints a 512-length vector containing ones at
%   positions 1 and 256 [1-based indexing] and zeros otherwise)
r

% Recall b from M2, applying Willshaw's threshold
r = willshawThreshold(recallMatrix(b, M2), 2);
% Print r to stdout (prints a 512-length vector containing ones at
%   positions 4 and 259 [1-based indexing] and zeros otherwise)
r
```

A.2 ENAMeL Code Used for ARCA

This ENAMeL code runs an experiment on the single-CMM ARCA, with a branching factor of 1 and tree depth of 5. The experiment stores 5 rules, and then attempts to iteratively recall these. After each iteration the target token **t5** is recalled from the output tensor product **O**, in order to determine if it is present within the output. The `stdout` stream is captured by an external program, allowing the analysis of these results—a recall has failed if at any intermediate point the rule **r** is not empty, or if at the end it does not equal the target rule **r4**.

arca.txt

```
# ENAMeL code used to run the single-CMM ARCA with branching factor 1,
#   depth 5, token length 128-bits, rule length 64-bits, vector weight 4

# Create a Baum vector generator, for vectors of length 128 and weight 4
Generator 29 31 33 35
# Create a Baum vector generator, for vectors of length 64 and weight 4
```

```
7  Generator 13 15 17 19
8
9  # Generate 6 token vectors (initial input and 5 outputs,
10 #   1 for each iteration)
11 t5 = Vector 4 128
12 t3 = Vector 4 128
13 t1 = Vector 4 128
14 t0 = Vector 4 128
15 t2 = Vector 4 128
16 t4 = Vector 4 128
17 # Generate 6 rule vectors (1 for each iteration,
18 #   and 1 used during recall initialisation)
19 r2 = Vector 4 64
20 r4 = Vector 4 64
21 r1 = Vector 4 64
22 r3 = Vector 4 64
23 r0 = Vector 4 64
24 rextra = Vector 4 64
25
26 # Create an empty 128x8192 matrix
27 M = Matrix 128 8192
28
29 # Store the 5 rules in the matrix, each is of the form
30 #   'rule(x): token(x) -> token(x+1)'
31 M = Mv(t0:(t1:r0))
32 M = Mv(t1:(t2:r1))
33 M = Mv(t2:(t3:r2))
34 M = Mv(t3:(t4:r3))
35 M = Mv(t4:(t5:r4))
36
37 # Print the target rule to stdout (prints 'r4: 1 14 29 46')
38 Print r4
39
40 # Initialise the recall by creating an output tensor product,
41 #   used first as the input
42 O = t0:rextra
43
44 # Iteration 1
45 # Recall each column of O from M, apply Willshaw's threshold with a value
46 #   of 4, and sum these together.  Finally, apply Willshaw's threshold
47 #   again with a value of 4
```

```
48 O = (O,M,4)|4
49 # Recall t5 (the target token) from O, applying Willshaw's threshold
50 #   with a value of 4
51 r = (t5.O)|4
52 # Print the output rule to stdout, for comparison to the target
53 #   (prints 'r: ')
54 Print r
55
56 # Iteration 2
57 O = (O,M,4)|4
58 r = (t5.O)|4
59 # (prints 'r: ')
60 Print r
61
62 # Iteration 3
63 O = (O,M,4)|4
64 r = (t5.O)|4
65 # (prints 'r: ')
66 Print r
67
68 # Iteration 4
69 O = (O,M,4)|4
70 r = (t5.O)|4
71 # (prints 'r: ')
72 Print r
73
74 # Iteration 5
75 O = (O,M,4)|4
76 r = (t5.O)|4
77 # (prints 'r: 1 14 29 46')
78 Print r
79
80 # Exit from ENAMeL
81 Exit
```

A.2.1 MATLAB equivalent

trainARCAMatrix.m

```
% Train a matrix, input -> output:rule
function M = trainARCAMatrix(M, input, output, rule)
    % Create output:rule tensor product
    tempM = trainMatrix(createMatrix(length(output), length(rule)), ...
                        output, rule);
    % Train the actual matrix
    M = trainMatrix(M, input, reshape(tempM, numel(tempM), 1));
end
```

recallARCAMatrix.m

```
% Recall from a matrix, recalling each column in turn, applying a
%   threshold, and then summing them together
function v = recallARCAMatrix(INPUT, M, value)
    v = zeros(size(M, 2), size(INPUT, 2));
    for col = 1:size(INPUT, 2)
        v(:, col) = recallMatrix(INPUT(:, col), M);
    end
    v = willshawThreshold(v, value);
    v = sum(v, 2);
    v = reshape(v, 128, 64);
end
```

arca.m

```
% ENAMeL code used to run the single-CMM ARCA with branching factor 1,
%   depth 5, token length 128-bits, rule length 64-bits, vector weight 4,
%   converted to MATLAB

% Generate 6 token vectors (initial input and 5 outputs,
%   1 for each iteration)
t5 = createBaumVector(128, [29 31 33 35], 0);
t3 = createBaumVector(128, [29 31 33 35], 1);
t1 = createBaumVector(128, [29 31 33 35], 2);
t0 = createBaumVector(128, [29 31 33 35], 3);
t2 = createBaumVector(128, [29 31 33 35], 4);
t4 = createBaumVector(128, [29 31 33 35], 5);
% Generate 6 rule vectors (1 for each iteration,
%   and 1 used during recall initialisation)
```

```
15 r2 = createBaumVector(64, [13 15 17 19], 0);
16 r4 = createBaumVector(64, [13 15 17 19], 1);
17 r1 = createBaumVector(64, [13 15 17 19], 2);
18 r3 = createBaumVector(64, [13 15 17 19], 3);
19 r0 = createBaumVector(64, [13 15 17 19], 4);
20 rextra = createBaumVector(64, [13 15 17 19], 5);
21
22 % Create an empty 128x8192 matrix
23 M = createMatrix(128, 8192);
24
25 % Store the 5 rules in the matrix, each is of the form
26 %   'rule(x): token(x) -> token(x+1)'
27 M = trainARCAMatrix(M, t0, t1, r0);
28 M = trainARCAMatrix(M, t1, t2, r1);
29 M = trainARCAMatrix(M, t2, t3, r2);
30 M = trainARCAMatrix(M, t3, t4, r3);
31 M = trainARCAMatrix(M, t4, t5, r4);
32
33 % Print the target rule to stdout (prints a 64-length vector containing ones
34 %   at positions 2, 15, 30, and 47 [1-based indexing] and zeros otherwise)
35 r4
36
37 % Initialise the recall by creating an output tensor product,
38 %   used first as the input
39 O = trainMatrix(createMatrix(128, 64), t0, rextra);
40
41 % Iteration 1
42 % Recall each column of O from M, apply Willshaw's threshold with a value
43 %   of 4, and sum these together.  Finally, apply Willshaw's threshold
44 %   again with a value of 4
45 O = willshawThreshold(recallARCAMatrix(O, M, 4), 4);
46 % Recall t5 (the target token) from O, applying Willshaw's threshold
47 %   with a value of 4
48 r = recallMatrix(t5, O);
49 r = willshawThreshold(r, 4);
50 % Print the output rule to stdout, for comparison to the target
51 %   (prints a 64-length vector containing zeros)
52 r
53
54 % Iteration 2
55 O = willshawThreshold(recallARCAMatrix(O, M, 4), 4);
```

```
r = willshawThreshold(recallMatrix(t5, O), 4);
% (prints a 64-length vector containing zeros)
r

% Iteration 3
O = willshawThreshold(recallARCAMatrix(O, M, 4), 4);
r = willshawThreshold(recallMatrix(t5, O), 4);
% (prints a 64-length vector containing zeros)
r

% Iteration 4
O = willshawThreshold(recallARCAMatrix(O, M, 4), 4);
r = willshawThreshold(recallMatrix(t5, O), 4);
% (prints a 64-length vector containing zeros)
r

% Iteration 5
O = willshawThreshold(recallARCAMatrix(O, M, 4), 4);
r = willshawThreshold(recallMatrix(t5, O), 4);
% (prints a 64-length vector containing ones at positions 2, 15, 30, and 47
%   [1-based indexing] and zeros otherwise)
r
```

Abbreviations and Nomenclature

$\wedge$	Logical AND; denotes conjunctions and bitwise AND operations
$\vee$	Logical OR; denotes the superposition of binary vectors or matrices
:	Represents the binding of two vectors to form a tensor product using the outer product operation, e.g. $\mathbf{a} : \mathbf{b}$ *(in some other work this is represented by* $\mathbf{a} \otimes \mathbf{b}$*)*
M	A matrix is represented as an emboldened uppercase character
v	A vector is represented as an emboldened lowercase character
l_x	The length of a vector x
w_x	The weight of a vector x
ADAM	Advanced Distributed Associative Memory
ANN	Artificial Neural Network
ARCA	Associative Rule Chaining Architecture
Arity	The number of antecedents of a rule
BFS	Breadth-First Search
Binary	Unless otherwise stated, binary denotes {0, 1}
CAM	Content Addressable Memory
CANN	Cellular Associative Neural Network
CMM	Correlation Matrix Memory *(in this work specifically a binary CMM)*

C-NNAP Cellular Neural Network Associative Processor

DFS Depth-First Search

DSL Domain Specific Language

ENAMeL Extended Neural Associative Memory Language

Ghost An incorrectly set bit within a recall output

Hamming weight The Hamming weight of a binary vector is the number of bits it contains that are set to `1` *(in this work referred to simply as the vector's weight)*

HRR Holographic Reduced Representation

MLP Multi-Layer Perceptron

Polar Polar denotes {`-1`, `1`}

Rule vector A fixed-weight binary vector representing a rule

RLE Run-length encoding

Token vector A fixed-weight binary vector representing an antecedent or consequent token

Weight See Hamming weight

References

[1] R.D. Adams. Content addressable memories. *High Performance Memory Testing: Design Principles, Fault Modeling and Self-Test*, pages 67–75, 2003.

[2] J.A. Anderson. *Automata theory with modern applications.* Cambridge University Press, 2006.

[3] J. Austin. The design and application of associative memories for scene analysis. *PhD, Department of Electrical Engineering, Brunel University*, 1986.

[4] J. Austin. Grey scale n tuple processing. *Pattern Recognition*, pages 110–119, 1988.

[5] J. Austin. Parallel distributed computation. In *Artificial Neural Networks and Machine Learning–ICANN 1992*, 1992.

[6] J. Austin. Distributed associative memories for high-speed symbolic reasoning. *Fuzzy Sets and Systems*, 82(2):223–233, 1996.

[7] J. Austin. A production system architecture using a neural associative memory. Unpublished manuscript, 2007.

[8] J. Austin, M. Brown, I. Kelly, and S. Buckle. ADAM neural networks for parallel vision. In *JFIT Technical Conference*, pages 173–180, 1993.

[9] J. Austin, S. Hobson, N. Burles, and S. O'Keefe. A rule chaining architecture using a correlation matrix memory. In *Artificial Neural Networks and Machine Learning–ICANN 2012*, pages 49–56. Springer, 2012.

[10] J. Austin, J. Kennedy, and K. Lees. The advanced uncertain reasoning architecture, AURA. In J. Austin, editor, *RAM-Based Neural Networks*, pages 43–50. World Scientific Publishing, 1998.

[11] J. Austin and T.J. Stonham. Distributed associative memory for use in scene analysis. *Image and vision computing*, 5(4):251–260, 1987.

[12] D.H. Ballard. Generalizing the hough transform to detect arbitrary shapes. *Pattern recognition*, 13(2):111–122, 1981.

[13] E. B. Baum, J. Moody, and F. Wilczek. Internal representations for associative memory. *Biological Cybernetics*, 59(4-5):217–228, 1988.

[14] C.M. Bishop. *Pattern recognition and machine learning*. Springer, New York, 2006.

[15] W.W. Bledsoe and I. Browning. Pattern recognition and reading by machine. In *Proceedings of the Eastern Joint Computer Conference*, pages 225–232. IEEE Computer Society, 1959.

[16] G. Brewer. *Spiking cellular associative neural networks for pattern recognition*. PhD thesis, University of York, 2008.

[17] N. Burles. 'Quantum' parallel computation with neural networks. Master's thesis, University of York, 2010.

[18] N. Burles, J. Austin, and S. O'Keefe. Extending the associative rule chaining architecture for multiple arity rules. In *Neural-Symbolic Learning and Reasoning Workshop*, pages 47–51, Beijing, 5 August 2013.

[19] N. Burles, S. O'Keefe, and J. Austin. Improving the associative rule chaining architecture. In *Artificial Neural Networks and Machine Learning–ICANN 2013*, pages 98–105. Springer, 2013.

[20] N. Burles, S. O'Keefe, and J. Austin. Incorporating scale invariance into the cellular associative neural network. In *Artificial Neural Networks and Machine Learning–ICANN 2014*, pages 435–442. Springer, 2014.

[21] N. Burles, S. O'Keefe, J. Austin, and S. Hobson. ENAMeL: A language for binary correlation matrix memories. *Neural Processing Letters*, 40(1):1–23, 2014.

[22] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (6):679–698, 1986.

[23] D. Casasent and B. Telfer. High capacity pattern recognition associative processors. *Neural Networks*, 5(4):687–698, 1992.

[24] L.O. Chua and L. Yang. Cellular neural networks: Applications. *IEEE Transactions on Circuits and Systems*, 35(10):1257–1272, 1988.

[25] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[26] M. de Kamps, V. Baier, J. Drever, M. Dietz, L. Mösenlechner, and F. van der Velde. The state of MIIND. *Neural Networks*, 21(8):1164–1181, 2008.

[27] C. De La Higuera. A bibliographical study of grammatical inference. *Pattern recognition*, 38(9):1332–1348, 2005.

[28] T. de Saint Pierre and M. Milgram. New and efficient cellular algorithms for image processing. *CVGIP: Image Understanding*, 55(3):261–274, 1992.

[29] C.P. Dolan and P. Smolensky. Tensor product production system: a modular architecture and representation. *Connection Science*, 1(1):53–68, 1989.

[30] S.C. Eisenstat, M.C. Gursky, M.H. Schultz, and A.H. Sherman. Yale sparse matrix package I: The symmetric codes. *International Journal for Numerical Methods in Engineering*, 18(8):1145–1151, 1982.

[31] M. Friedman and A. Kandel. *Introduction to Pattern Recognition: Statistical, Structural, Neural and Fuzzy Logic Approaches*. World Scientific, 1999.

[32] K.S. Fu. *Handbook of Pattern Recognition and Image Processing*, chapter Syntactic Pattern Recognition. In Young and Fu [115], 1986.

[33] K. Fukunaga. *Handbook of Pattern Recognition and Image Processing*, chapter Statistical Pattern Classification. In Young and Fu [115], 1986.

[34] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, 1980.

[35] K. Fukushima. Neocognitron for handwritten digit recognition. *Neurocomputing*, 51:161–180, 2003.

[36] K. Fukushima. Restoring partly occluded patterns: a neural network model. *Neural Networks*, 18(1):33–43, 2005.

[37] K. Fukushima. Interpolating vectors for robust pattern recognition. *Neural Networks*, 20(8):904–916, 2007.

[38] K Fukushima. Neocognitron. *Scholarpedia*, 2(1):1717, 2007.

[39] K. Fukushima. Training multi-layered neural network neocognitron. *Neural Networks*, 40:18–31, 2013.

[40] K. Fukushima. One-shot learning with feedback for multi-layered convolutional network. In *Artificial Neural Networks and Machine Learning–ICANN 2014*, pages 291–298. Springer, 2014.

[41] R.W. Gayler. Multiplicative binding, representation operators and analogy. In *Advances in analogy research: Integration of theory and data from the cognitive, computational, and neural sciences*, 1998.

[42] R.W. Gayler. Vector symbolic architectures answer Jackendoff's challenges for cognitive neuroscience. In P.P. Slezak, editor, *Proceedings of the Joint International Conference on Cognitive Science*, 2003.

[43] R.W. Gayler. Vector symbolic architectures are a viable alternative for Jackendoff's challenges. In *peer commentary on [107]*. Cambridge University Press, 2006.

[44] M. Gori and A. Tesi. On the problem of local minima in backpropagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(1):76–86, 1992.

[45] D. Graupe. *Principles of artificial neural networks*. World Scientific, 2nd edition, 2007.

[46] A.K. Gupta and Y.P. Singh. Analysis of neocognitron of neural network method in the string recognition. *ACEEE International Journal on Network Security*, 2(3):18–22, 2011.

[47] S. Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall, 2nd edition, 1997.

[48] Donald O. Hebb. *The Organization of Behavior*. John Wiley and Sons, 1949.

[49] G.E. Hinton. Mapping part-whole hierarchies into connectionist networks. *Artificial Intelligence*, 46(1):47–75, 1990.

[50] G.E. Hinton, J.L. McClelland, and D.E. Rumelhart. *Distributed representations*. Carnegie Mellon University, Computer Science Dept., 1984.

[51] S. Hobson. *Correlation Matrix Memories: Improving Performance for Capacity and Generalisation*. PhD thesis, University of York, 2011.

[52] S. Hobson and J. Austin. Improved storage capacity in correlation matrix memories storing fixed weight codes. In *Artificial Neural Networks and Machine Learning–ICANN 2009*, pages 728–736. Springer, 2009.

[53] V. Hodge, T. Jackson, and J. Austin. Intelligent decision support using pattern matching. In *International Workshop on Future Internet Applications for Traffic Surveillance and Management—FIATS-M 2011*, pages 44–54, 2011.

[54] 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, 1982.

[55] D.H. Hubel and T.N. Wiesel. Receptive fields and functional architecture in two nonstriate visual areas (18 and 19) of the cat. *Journal of Neurophysiology*, 1965.

[56] D.A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[57] A. Ishai, L.G. Ungerleider, A. Martin, J.L. Schouten, and J.V. Haxby. Distributed representation of objects in the human ventral visual pathway. *Proceedings of the National Academy of Sciences of the United States of America*, 96(16):9379, 1999.

[58] R. Jackendoff. *Foundations of language: Brain, meaning, grammar, evolution*. Oxford University Press, 2002.

[59] A.K. Jain, J. Mao, and K.M. Mohiuddin. Artificial neural networks: A tutorial. *Computer*, 29(3):31–44, 1996.

[60] P. Kanerva. The spatter code for encoding concepts at many levels. In *Artificial Neural Networks and Machine Learning–ICANN 1994*, pages 226–229. Springer, 1994.

[61] J.V. Kennedy, J. Austin, R. Pack, and B. Cass. C-NNAP-a parallel processing architecture for binary neural networks. In *IEEE International Conference on Neural Networks*, volume 2, pages 1037–1041. IEEE, 1995.

[62] S. Klinger. *Chemical Similarity Searching with neural graph matching methods.* PhD thesis, University of York, 2006.

[63] D.E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and searching.* Addison-Wesley, 1973.

[64] T. Kohonen. Correlation matrix memories. *IEEE Transactions on Computers*, 100(4):353–359, 1972.

[65] B. Kröse and P. van der Smagt. An introduction to neural networks. 1996.

[66] D. Kustrin and J. Austin. Connectionist propositional logic a simple correlation matrix memory based reasoning system. *Emergent neural computational architectures based on neuroscience*, pages 534–546, 2001.

[67] T.K. Leung, M.C. Burl, and P. Perona. Finding faces in cluttered scenes using random labeled graph matching. In *Fifth International Conference on Computer Vision*, pages 637–644. IEEE, 1995.

[68] W.S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[69] R. McEliece, E. Posner, E. Rodemich, and S. Venkatesh. The capacity of the Hopfield associative memory. *IEEE Transactions on Information Theory*, 33(4):461–482, 1987.

[70] M.L. Minsky and S.A. Papert. *Perceptrons - Expanded Edition: An Introduction to Computational Geometry.* MIT Press, Boston, 1987.

[71] M. Mishkin and L.G. Ungerleider. Contribution of striate inputs to the visuospatial functions of parieto-preoccipital cortex in monkeys. *Behavioural Brain Research*, 6(1):57–77, 1982.

[72] F. Mokhtarian and A.K. Mackworth. A theory of multiscale, curvature-based shape representation for planar curves. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(8):789–805, 1992.

[73] R. Mounce, G. Hollier, M. Smith, V.J. Hodge, T. Jackson, and J. Austin. A metric for pattern-matching applications to traffic management. *Transportation Research Part C: Emerging Technologies*, 29:148–155, 2013.

[74] B. Müller, J. Reinhardt, and M.T. Strickland. *Neural networks: an introduction*. Springer Verlag, 1995.

[75] N.M. Nasrabadi and W. Li. Object recognition by a Hopfield neural network. *IEEE Transactions on Systems, Man and Cybernetics*, 21(6):1523–1535, 1991.

[76] C. Orovas. Cellular associative neural networks for pattern recognition. Technical Report YCST-99-12, University of York Department of Computer Science, 1999.

[77] C. Orovas. *Cellular associative neural networks for pattern recognition*. PhD thesis, University of York, 2000.

[78] C. Orovas and J. Austin. Cellular associative neural networks for image interpretation. In *Sixth International Conference on Image Processing and Its Applications*, volume 2, pages 665–669. IET, 1997.

[79] K. Pagiamtzis and A. Sheikholeslami. Content-addressable memory (CAM) circuits and architectures: A tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, 2006.

[80] G. Palm. On associative memory. *Biological Cybernetics*, 36(1):19–31, 1980.

[81] G. Palm. On the storage capacity of associative memories. In *Neural assemblies, an alternative approach to artificial intelligence*, pages 192–199. Springer-Verlag, New York, 1982.

[82] G. Palm, F. Schwenker, F.T. Sommer, and A. Strey. Neural associative memory. *Associative Processing and Processors*, pages 307–326, 1997.

[83] R.S. Petersen, S. Panzeri, and M. Maravall. Neural coding and contextual influences in the whisker system. *Biological cybernetics*, 100(6):427–446, 2009.

[84] TA Plate. Holographic reduced representations: Convolution algebra for compositional distributed representations. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 30–35. Morgan Kaufmann, 1991.

[85] T.A. Plate. Estimating analogical similarity by dot-products of holographic reduced representations. *Advances In Neural Information Processing Systems*, pages 1109–1116, 1994.

[86] K. Preston and M.J.B. Duff. *Modern cellular automata: theory and applications*, volume 198. Plenum Press, New York, 1984.

[87] D.A. Rachkovskij and E.M. Kussul. Binding and normalization of binary sparse distributed representations by context-dependent thinning. *Neural Computation*, 13(2):411–452, 2001.

[88] R. Rojas. *Neural networks: a systematic introduction*. Springer, 1996.

[89] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning representations by back-propagating errors. *Nature*, 323(9):533–536, 1986.

[90] D.E. Rumelhart and J.L. MacClelland. *Parallel distributed processing*. MIT Press, 1986.

[91] S. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Prentice Hall, 3rd edition, 2009.

[92] R. Schalkoff. *Pattern Recognition: Statistical, Structural and Neural Approaches*. John Wiley & Sons, Inc., 1992.

[93] M.N. Shadlen and W.T. Newsome. Noise, neural codes and cortical organization. *Current Opinion in Neurobiology*, 4(4):569–579, 1994.

[94] H. Shouno. Recent studies around the neocognitron. In *Neural Information Processing*, pages 1061–1070. Springer, 2008.

[95] P.K. Simpson. *Artificial Neural Systems: Foundations, Paradigms, Applications, and Implementations*. Pergamon Press, Inc., 1990.

[96] P. Smolensky. Neural and conceptual interpretations of parallel distributed processing models. Technical Report CU-CS-322-86, Department of Computer Science, University of Colorado, Boulder, 1986.

[97] P. Smolensky. On variable binding and the representation of symbolic structures in connectionist systems. Technical Report CU-CS-355-87, Department of Computer Science, University of Colorado, Boulder, 1987.

[98] P. Smolensky. Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial intelligence*, 46(1):159–216, 1990.

[99] F.T. Sommer and P. Kanerva. Can neural models of cognition benefit from the advantages of connectionism? In *peer commentary on [107]*. Cambridge University Press, 2006.

[100] B.L. Song. Tour Eiffel Wikimedia Commons. `http://commons.wikimedia.org/wiki/File:Tour_Eiffel_Wikimedia_Commons.jpg`, June 2009.

[101] K. Tanaka. Inferotemporal cortex and object vision. *Annual review of neuroscience*, 19(1):109–139, 1996.

[102] G. Tesauro and B. Janssens. Scaling relationships in back-propagation learning. *Complex Systems*, 2(1):39–44, 1988.

[103] S. Theodoridis and K. Koutroumbas. *Pattern Recognition*. Academic Press, London, 4th edition, 2009.

[104] M. Turner and J. Austin. Matching performance of binary correlation matrix memories. *Neural Networks*, 10(9):1637–1648, 1997.

[105] E. Underwood and B.E. Underwood. Eiffel Tower. `http://news.nationalgeographic.com/news/2014/04/140401-paris-france-125-anniversary-eiffel-tower-engineering`, September 1914.

[106] F. van der Velde and M. de Kamps. From knowing what to knowing where: Modeling object-based attention with feedback disinhibition of activation. *Journal of Cognitive Neuroscience*, 13(4):479–491, 2001.

[107] F. van der Velde and M. de Kamps. Neural blackboard architectures of combinatorial structures in cognition. *Behavioral and Brain Sciences*, 29(01):37–70, 2006.

[108] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.

[109] J. von Neumann and A.W. Burks. Theory of self-reproducing automata. 1966.

[110] H. Wechsler and G.L. Zimmerman. 2-D invariant object recognition using distributed associative memory. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(6):811–821, 1988.

[111] R.W. Williams and K. Herrup. The control of neuron number. *Annual Review of Neuroscience*, 11(1):423–453, 1988.

[112] D.J. Willshaw, O.P. Buneman, and H.C. Longuet-Higgins. Non-holographic associative memory. *Nature*, 1969.

[113] S. Wolfram. *Cellular automata and complexity: collected papers*. Addison-Wesley, Reading, 1994.

[114] H.J. Wolfson and I. Rigoutsos. Geometric hashing: An overview. *Computational Science & Engineering*, 4(4):10–21, 1997.

[115] T.Y. Young and K.S. Fu, editors. *Handbook of Pattern Recognition and Image Processing*. Academic Press, Inc., 1986.

[116] L.A. Zadeh. Fuzzy logic, neural networks, and soft computing. *Communications of the ACM*, 37(3):77–84, 1994.

[117] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[118] J. Zou and G. Nagy. Human-computer interaction for complex pattern recognition problems. In *Data complexity in pattern recognition*, pages 271–286. Springer, 2006.