Department of Computer Science, University of York

‘Quantum’ Parallel computation with neural networks

Submitted in part fulfilment for the degree of MEng

Nathan Burles
10 May 2010

Word count: 23,989 on 70 pages as counted by the Microsoft Word *word count* command.
This includes the glossary, but excludes all appendices.

Abstract

Correlation matrix memories have been successfully applied to many domains. This work implements a production system put forward in [Austin, 2003], to demonstrate its viability as an efficient rule-chaining process. Background information on rule-chaining and CMMs is given, followed by a review of the proposed production system.

Throughout the iterative development process, experimentation is performed in order to investigate the effects of changing the properties of vectors used in this system. The results show that generating vectors using the algorithm proposed in [Baum, 1988] with a weight close to $\log_2$ of the vector length provides the highest storage capacity.

The simple system implemented in this work performs rule-chaining effectively. This leads to the conclusion that the proposed production system is viable, and that this area warrants further work.

Acknowledgements

Thanks to Jim Austin for supervising this work and providing assistance and advice throughout
Thanks to Raymond Birch for the use of his computer to run experiments

Table of contents

List of figures

List of equations

List of tables

1 Introduction

Artificial neural networks were designed as an attempt to mimic the higher functions of the animal brain, particularly for applications such as pattern recognition. Correlation matrix memories (CMMs) are a special class of neural network that have the benefit of a fast, online training algorithm.

CMMs have been successfully applied to many and varied domains, including pattern recognition [35], chemical similarity searching [26], and spell checking algorithms [22]. This work is concerned with the application of rule-chaining, or inference.

1.1 Aims

The main aim of this work is to show whether the rule-chaining system proposed in [1] is a feasible application of CMMs. A simple CMM is required in order to demonstrate this, as well as a classical system in which to store rules for comparison.

Experimentation will be performed at each stage of the development of the final system. This is in order to fulfil the secondary aims of this work, namely to investigate the properties of CMMs, vectors, and tensor products.

1.2 Report structure

Chapter 2 introduces the background information to this work, progressing gradually through the field of knowledge. This chapter initially gives a basic description of finite state architectures, before introducing rule-chaining. A very limited introduction to quantum computation follows, in order to demonstrate the benefits that may be achieved through the use of superposition. Next is a description of neural networks, with CMMs in the following section. Finally, a detailed explanation of the proposed method of rule-chaining is given.

The following six chapters describe the development and experimentation performed within this work. Chapter 3 contains the initial planning that is imperative when performing an iterative development. Each of the remaining development chapters contain a series of sections that match the stages of the iterative development process: overview, design, implementation, testing, results, and evaluation. Chapter 4 describes the first iteration, which is the development of a simple CMM. Chapter 5 moves on to investigate the use of tensor products that can later allow the superposition of tokens, and Chapter 6 extends this by performing rule-chaining – taking the output of a CMM directly back to be an input. Chapter 7 explores the superposition of tensor products, and Chapter 8 examines the final system of this work.

Finally, Chapter 9 gives the conclusions made within this work, as well as making some further conclusions. Some possible future work is also described, such as the implementation of arity networks, and the mathematical proof of improvements that the CMM system can offer in comparison to a classical system with regards to time or space complexity.

1.3 Statement of ethics

The work presented here does not carry any ethical implications, as it only serves to be a proof of concept with the provision of a prototype system. If the proposed production system were to be used for control of a safety critical system, then a formal proof of safety would be required for that system.

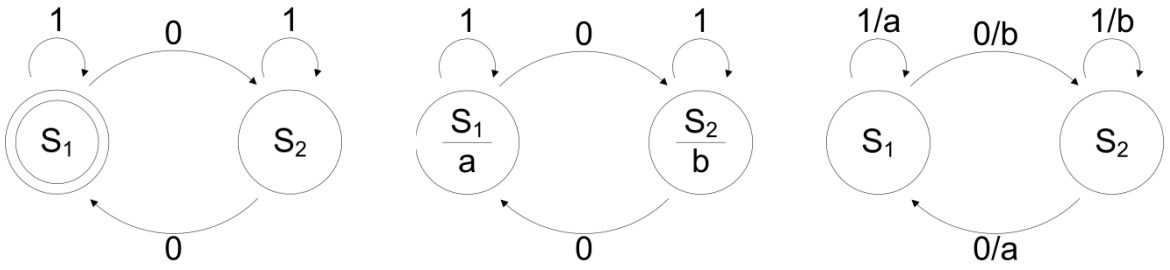
2 Literature review

This chapter presents the background information to this work, gradually focusing on the techniques used to perform rule-chaining within a CMM network. Section 2.1 gives a brief introduction to finite state architectures, and Section 2.2 builds upon this with a description of rule-chaining. Section 2.3 is a simple description of quantum computation and the benefits that superposition can offer. Section 2.4 introduces neural networks, while Section 2.5 describes a particular class of neural networks known as a CMM. Finally, Section 2.6 details the techniques that have been proposed to perform rule-chaining using CMMs.

2.1 Finite state architectures

Finite state architectures can be used to model the behaviour of a system, or object within a system [11]. They can be represented as a 5-tuple M = (Q, Σ, q_0, δ, A) [31], where:

- Q is a finite set of states
- Σ is an alphabet of input symbols
- $q_0 \in Q$ is the initial state
- $\delta: Q \times \Sigma \rightarrow Q$ is the transition function
- $A \subseteq Q$ is the set of accepting states



a: Finite state acceptor b: Moore machine c: Mealy machine

Figure 1: Finite state machines

More simply, they can be thought of as a directed graph (Figure 1), with three main elements:

- A finite set of states, represented as nodes
- State transitions, represented as edges
- Input symbols, governing which state transition should be taken given the current state

Finite state machines (FSM) are often categorised into two classes – acceptors and transducers [30]. A finite state acceptor (FSA) is an automaton that will simply accept or reject an input string, and as such they can be used to recognise a language. This class of automaton (Figure 1a) is most widely described in literature as it is simple, yet powerful, and can be used for purposes such as lexical analysis.

On the other hand, finite state transducers (FST) often receive only a cursory mention within a chapter on FSM [30]. An FST is a more general automaton whose output is a string, rather than merely an accept or reject, and a form of this is implemented in this work. There are two sub-classes of FST – Moore machines (Figure 1b) and Mealy machines (Figure 1c).

A Moore machine produces output in each state [32]. In contrast to this, Mealy machines produce output in each transition [33]. Although the two types of FST differ in how they represent and produce an output, they actually have the same expressive power as "it is possible to convert any Mealy type machine into an equivalent Moore type machine, and vice-versa" [18].

2.2 Rule-chaining

Logic is the study of reasoning – forming conclusions from a set of premises [17]. These arguments are commonly given in the form:

$$\frac{Premise\ 1}{\underline{Premise\ 2}}$$
$$Conclusion$$

Alternatively, it can be more efficient to use an inline form of propositional logic, for example:

$$Premise\ 1 \wedge Premise\ 2 \Rightarrow Conclusion$$

In this report, I will use the inline representation and a subset of the syntax used in [38], specifically:

$$\begin{array}{rl} \neg\, p & Negation - not\ p \\ p \wedge q & Conjunction - p\ and\ q \\ p \vee q & Disjunction - p\ or\ q \\ p \rightarrow q & Implication - p\ (the\ antecedent)\ implies\ q\ (the\ consequent) \end{array}$$

There are two methods of inference commonly used for rule-chaining – forward chaining and backward chaining [40]. In propositional logic, both of these types of reasoning have the same goal. They approach the problem of forming a conclusion from a series of premises from opposing angles.

2.2.1 Inference

Forward chaining, also known as data-driven rule-chaining [13], searches the available rules in order to find one for which the antecedent is known to be true. This allows the current state to be updated to include the consequent of that rule. By repeated application of this process, the goal may be achieved. This process effectively implements a finite state machine, although commonly with few or no cycles.

The alternative, backward chaining, can be known as goal-driven rule-chaining [13]. In this method, all rules with a consequent matching the desired goal and with antecedents that are not known to be false are added to the current state. This is repeated, with new rules added such that their consequents match the antecedents already in the current state, until a rule is found that has an antecedent that is known to be true.

To better explain the mechanisms used by forward and backward chaining, I have adapted an example from [13]. The goal is to determine the mood of a university professor, given that they are currently lecturing, and the knowledge held by the system is:

$$\begin{array}{rl} 1 & lecturing \rightarrow overworked \\ 2 & \neg\, lecturing \rightarrow researching \\ 3 & overworked \rightarrow grumpy \\ 4 & researching \rightarrow happy \end{array}$$

Using forward chaining, the antecedents of all the rules are checked against the current state. Rule 1 matches, and so the consequent $overworked$ is added to the state. The knowledge base is searched again, and this time rule 3 matches, resulting in $grumpy$ being added to the state. At this point the system may perform one of two actions:

1. Halt, if it has further information that enables it to recognise $grumpy$ as a possible mood, and hence conclude that it has reached the goal.
2. Perform a further search of the knowledge base to determine if there are any further rules whose antecedents match. If there are not, then it may conclude that the final state reached was the goal state.

In backward chaining, the consequents of all the rules are checked against the goal state – and so the system is required to hold further information as to what constitutes a "mood". Rules 3 and 4 both match the goal, and so their antecedents are compared to the current state to check for consistency. Neither $overworked$ nor $researching$ are currently known to be either true or false, and so both of these antecedents are added to the goal list. The knowledge base is searched again, and this time rules 1 and 2 match the goals. Upon checking their antecedents, the system is able to eliminate rule 2, and confirm rule 1. This confirmation leads to the proof of rule 3, which in turn gives the desired goal.

It is claimed that "forward checking is another ... technique which offers an improvement over the inefficient behaviour of chronological backtracking" [40]. On the other hand, [13] argues that "whether you use forward or backwards reasoning to solve a problem depends on the properties of your rule set and initial facts". In general, forward chaining has been found to be more efficient than backward chaining for applications such as agent control in the context of artificial intelligence [38], or a tree-based database search [13]. Complementary to this backward chaining is well suited to applications such as proving a fact, given a particular knowledge base – for example Prolog uses this technique when performing Selective Linear Definite (SLD) clause resolution [17].

2.3 Quantum computation

It is stated that:

> Quantum computation and quantum information is the study of the information processing tasks that can be accomplished using quantum mechanical systems. [34]

The motivations and mechanics behind quantum computation are outside the scope of this work; however some of the concepts are important to understand. The most important of these concepts is superposition, and the reduction in computational complexity that can be achieved as a result of its use.

Conventional computation is performed using transistors to manipulate bits. Quantum computation is analogous to this, manipulating 'quantum bits', or qubits [10]. Although qubits are physical objects, I will consider them to be abstract entities for the purposes of description.

Bra-ket, or Dirac, notation is commonly used to represent the state of a qubit $|\psi\rangle$ and whereas a classical bit may be in one of two states – often represented as 0 or 1 – a qubit may be in a superposition of states [34]:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

Figure 2: Qubit in superposition of states

where α and β are complex numbers.

2.3.1 Basis sets

A basis set can be described as a minimal set of orthonormal states such that any other state may be described as a linear combination of those states [10]. The simplest set of basis states for a single qubit is known as the computational basis set. This comprises of:

$$|0\rangle = 1|0\rangle + 0|1\rangle$$
$$|1\rangle = 0|0\rangle + 1|1\rangle$$

Figure 3: Computational basis set

The computational basis set is useful for representation of states, however in order to exploit the potential parallelism that quantum computation offers, a basis set comprising of an equal superposition of states is commonly used. Represented in the computational basis, for a single qubit system these states are:

$$|+\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$
$$|-\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$$

Figure 4: Equal superposition basis set

2.3.2 Measurement collapses an unknown state

Although a qubit may be in one of infinitely many combinations of states, it is not possible to determine what that state is through measurement [34]. That is to say, for a qubit in state:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

Figure 5: Qubit in unknown state

the coefficients α and β denote the probability of finding a particular state upon measurement. The result will either be a 0 with a probability of $|\alpha|^2$ or a 1 with a probability of $|\beta|^2$, where the sum of the probabilities is 1.

In addition to this limit, the action of performing a measurement on a qubit changes the state – causing it to collapse from an unknown superposition of $|0\rangle$ and $|1\rangle$ to one of the computational basis states, equal to the result of the measurement [34].

2.3.3 Grover's algorithm

In classical computation, a set B of logic gates is universal if for any Boolean function F, a circuit can be built comprising only of gates from B that computes F [10]. It can be demonstrated that there exists at least one universal set of quantum gates, which would allow any Boolean circuit to be synthesised as a quantum computer. This is not, however, the main objective in quantum computing as it does not exploit the benefits offered by quantum mechanics.

Instead of this, the advantage of quantum computing is the ability to solve problems that are infeasible using a classical computer [34], and the challenge is in designing algorithms and quantum circuits to do so. Examples of this, that offer very impressive results, include Shor's algorithm [41] and Deutsch's algorithm [15].

Grover's algorithm [19] is most relevant to this work, however, as it is an algorithm capable of improving the computational complexity of a database search.

Explained by Nielsen:

> Given a search space of size N, and no prior knowledge about the structure of the information in it, we want to find an element of that search space satisfying a known property. [34]

In a classical system, and in the worst case, this search will require N operations in order to obtain a result. Grover's algorithm is capable of obtaining a result in only $\sqrt{N}$ operations; a quadratic improvement. This is a smaller improvement than found in the previously mentioned algorithms; however it is still a significant gain, especially when the database is very large.

This section on quantum computation is only written as a simple introduction to the field, as there are many sources from which a more detailed explanation can be obtained (such as [10], [25], and [34]). It is, however, intended to highlight the potential improvement that can be achieved by computing using a superposition of states as opposed to the brute-force approach which must be applied to many of these problems when using conventional computation.

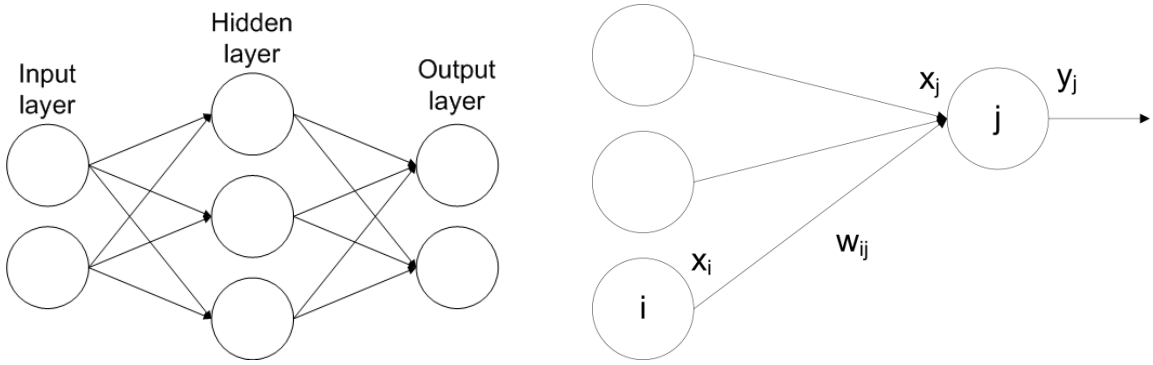
2.4 Neural networks

Traditional – or hard – computing is concerned with the precision and certainty of a solution, and rigour in the approach to obtaining that solution [49]. Soft computing, on the other hand, is "tolerant of imprecision, uncertainty, partial truth, and approximation" [24]. Within soft computing are three main fields: fuzzy logic, neural networks, and probabilistic reasoning [49].

Neural networks are "a biologically inspired approach to the pattern recognition problem" [48]. The animal brain consists of a great number of neurons, each connected to many other neurons by synapses. Artificial neural networks were designed to simulate this behaviour, as an attempt to reproduce our natural ability at pattern recognition. As each neuron within a network is fully self-contained, the system can be designed to be massively parallel [48]. It is important to emphasise that although neural networks are inspired by biological processes, they need not be constrained by them: the accurate modelling of biological neural networks, and the processes within them, is a field in its own right [47].

2.4.1 Feed-forward neural networks

Figure 6a shows a basic three-layer feed-forward artificial neural network. Each of the connections between two neurons is directional, which means that the network is effectively a directed graph. In addition to this, as it is a feed-forward network, the outputs of neurons in one layer are always connected as inputs to a neuron that is closer to the final output. This ensures that the network is a directed acyclic graph, and helps to reduce the complexity of the system by eliminating feedback [47].



a: Three-layer feed-forward neural network b: Inputs to a single neuron

Figure 6: Feed-forward artificial neural network

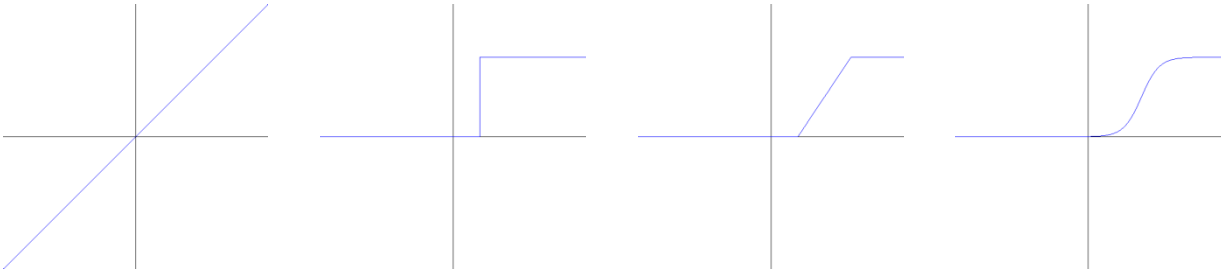
The output of a neuron is generated by applying a particular function to the weighted sum of its inputs. Figure 6b shows the inputs to a single neuron j, and the total input is given by [48]:

$$x_j = \alpha_j + \sum_{i \to j} w_{ij} x_i$$

Equation 1: Input to a neuron

where x_i is the output of 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 y_j of a neuron can then be calculated by applying a function to the value of the input x_j. This function may be different for each neuron, as they are self-contained processing units, however it is common for the same function to be used for all neurons in a layer [47]. Although any function may be used, there are four that are most commonly found in artificial neural networks [39]:



a: Linear function b: Threshold function c: Linear threshold function d: Sigmoidal function

Figure 7: Common neuron functions

The threshold function (Figure 7b) effectively outputs a Boolean value, and as such is useful in the output layer of a neural network in order to generate a binary result [47]. The sigmoidal function (Figure 7d), on the other hand, most closely approximates the function used by biological neurons [39]. It is often used in the hidden layer in order that the neural network is able to represent non-linear relationships. The equation of this function is:

$$y = \frac{1}{1 + e^{-\alpha x}}$$

Equation 2: Sigmoidal function

where the value of α controls the shape of the curve. In the limit $\alpha \to \infty$, the sigmoidal function converges to become a threshold function [37].

In order to perform a specific task, a general neural network must be trained in some way. This training could involve alterations to any of the parameters in the network, such as the function used

by each neuron, the weights of each connection, or the number and position of neurons and connections. Generally, within the literature, most emphasis has been placed on the consequences of modifying the weights of connections between individual neurons as this is most likely to mimic the approach used in nature [36].

2.4.2 Back-propagation

There are various algorithms used to train neural networks, classified into one of two categories: supervised learning and unsupervised learning [42]. Supervised learning requires pairs of known inputs and outputs, a training set. The inputs are presented to the neural network, and the output generated in each case is compared to the desired output in order to generate information on the errors. The neural network is then able to apply an algorithm to update the connection weights in order to minimise these errors. This is in contrast to unsupervised learning, where only the inputs are provided and the neural network must update the connection weights in order to minimise a given function [42].

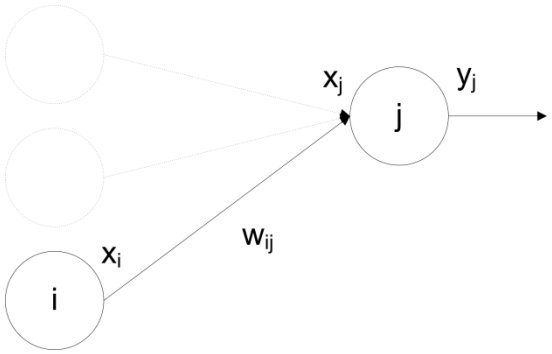


Figure 8: Neural network for back propagation

Due to its simplicity, one of the most common algorithms used within supervised learning is back-propagation [42]. A very limited description of this algorithm, restricted to only those neurons shown in Figure 8 can be described in 5 steps:

1. Present the neural network with an input taken from the training set.
2. Compare the output of neuron j with that position of the desired output from the training set, and calculate the error value for this neuron.
3. Adjust the weights of each input connection to neuron j in order to reduce the error.
4. Assign an error value to each of the neurons that input to neuron j. This is calculated as a portion of the error value of neuron j, and allocates more error to those connections with the highest weight as it considers those to be most to blame for the error.
5. Repeat from step 3 for neuron i, using the sum of all error values assigned to it from the neurons in the output layer.

Ripley [36] presents an algorithm that may be used in order that a feed-forward neural network may be trained using back-propagation whilst online, meaning that the training of the neural network occurs while it is in use. Despite the potential benefits that this offers, such as the ability to learn from all experiences, and the possibility of a faster convergence to a fully trained network [36], this is not commonly used; training is generally performed offline and before the neural network is used.

Much of the literature regarding neural networks is also concerned with pattern recognition. This may lead to a miscomprehension that pattern recognition is the only task to which neural networks

are applied. They are particularly well suited to the problem of pattern recognition as that was one of the driving factors behind their development, and they are modelled loosely on the structure of the brain which is very effective at pattern recognition [39]. However, neural networks have been used for a wide range of applications including control, database retrieval, and fault-tolerant computing [42].

2.5 Correlation matrix memories

A particular class of neural networks is often known as a RAM-based neural network. These include multi-layered weighted networks, such as the Probabilistic Logic Node (PLN) amongst others. In addition to this, however, are also single layer neural networks using only binary weights, such as the Multi-RAM Discriminator (MRD), or Advanced Distributed Associative Memory (ADAM) [2].

One of the subclasses of RAM based neural networks is called a CMM, of which ADAM is a particular implementation [5]. A CMM is "a simple binary associative neural network" [1] that utilises Hebbian learning to allow for high speed, online training.

Hebb wrote:

> 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. [20]

This refers to the work that Hebb had carried out to investigate how neurons within the human brain contribute to higher functions such as learning. Although 'The Organization of Behavior' was first published in 1949, this is still considered to be a valid theory for the human learning process, and was important in the initial development of artificial neural networks.

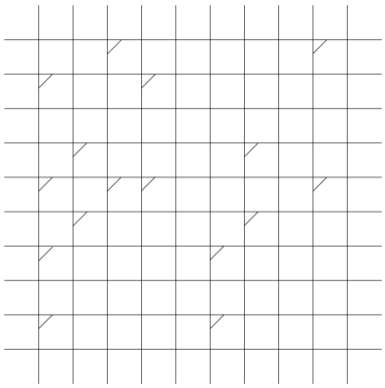
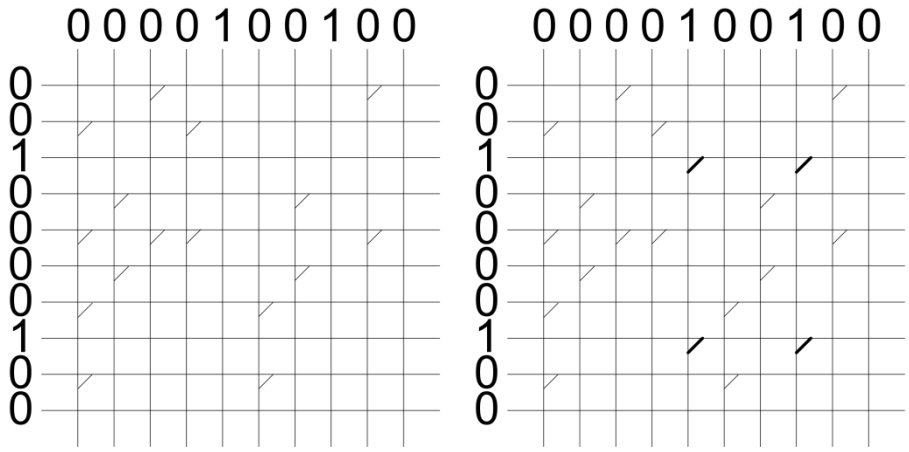


Figure 9: A binary correlation matrix memory

Figure 9 shows a binary CMM, where each of the intersections between vertical and horizontal wires represents a memory cell. As it is a binary CMM, each neuron (or memory cell) is capable of storing only a logical 0 or 1, and so a marked intersection represents a logical 1 where an unmarked intersection represents a logical 0. Effectively, the neuronal function for every neuron is simply the threshold function (Figure 7b). Inputs and outputs of a binary CMM are generally referred to as vectors or codes, and the weight of such a vector is the number of bits that are set to logical 1. A correlation within the matrix, or a vector bit set to logical 1, will be referred to as being 'set'.

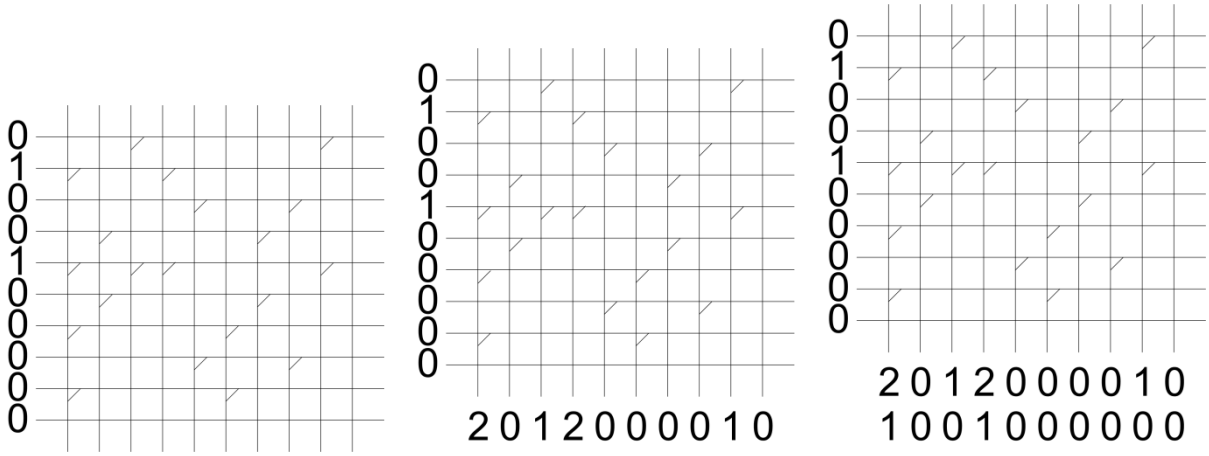


a: Present input and output vectors b: Store correlations

Figure 10: Training a binary correlation matrix memory

2.5.1 Training

Training of a binary CMM requires only a single step, as shown in Figure 10. When the neural network is presented with an input and output vector (Figure 10a), any correlation between the two vectors is set in the matrix. That is to say, a neuron at the intersection of any horizontal wire that is set on the input with a vertical wire set on the output is itself set (Figure 10b).



a: Present input vector b: Recall output pattern c: Threshold to obtain output vector

Figure 11: Recalling a pattern from a binary correlation matrix memory

2.5.2 Recall

Recall of a vector from a binary CMM is a similar process, shown in Figure 11. When the neural network is presented with only an input vector (Figure 11a), any neurons that are set and presented with a set input give an output value of logical 1. To obtain the output pattern, the outputs of all neurons on a vertical wire are simply summed (Figure 11b). Thresholding this pattern with a suitable threshold is then able to recall the original output vector (Figure 11c).

2.5.3 Thresholding

Deciding on a suitable threshold is the primary issue with regards to vector recall: there are various methods that may be used – Willshaw [7], L-max [3], and L-wta (Winner Takes All) [21] – each of which is suited to different applications [16].

3	1	2	1	0	a: Output pattern
1	1	1	1	0	b: Threshold at 1
1	0	1	0	0	c: Threshold at 2
1	0	0	0	0	d: Threshold at 3

Table 1: Willshaw thresholding

Using Willshaw thresholding, an absolute threshold value is selected: any integer in the output pattern that is greater than or equal to this threshold value is set in the output vector [46]. Selection of the threshold value is an important consideration, and will affect the retrieved vector as shown in Table 1. Various output vectors are shown, alongside the threshold value used to obtain them from the output pattern (Table 1a). In [45], Willshaw uses the weight of the input vector as the threshold value.

3	1	2	1	0	a: Output pattern
1	0	0	0	0	b: L=1
1	0	1	0	0	c: L=2
1	1	1	1	0	d: L=3

Table 2: L-max thresholding

L-max approaches the problem of thresholding from a very different angle. In this method, the L integers with the highest value in the output pattern are set in the output vector [12]. This is shown in Table 2, with various values for L; in circumstances when the output vector has a fixed weight, L can be selected to match this weight [21]. Table 2d is an example of a situation where the weight of the output vector differs from L – the algorithm is unable to distinguish the two integers with value 1 and so both are set in the output.

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

Table 3: Vectors generated using Baum's algorithm

3	1	2	1	0	a: Output pattern
1	0	0	1	0	b: Sections {3,2}
1	0	1	0	0	c: Sections {2,3}

Table 4: L-wta thresholding

In order to use L-wta thresholding, the output vectors must be generated using an algorithm described by Baum et al. [8]. This will deterministically generate fixed weight vectors by dividing the vector into L co-prime sections and moving the position of the set bit forward by one for every vector generated. An example of vectors generated using this algorithm with a length of 5, weight of 2, and with sections of lengths 3 and 2 is given in Table 3.

L-wta thresholding is essentially an extension of L-max that uses prior knowledge about the format of an output vector in order to improve recall accuracy. Each of the sections of an output pattern are thresholded individually, using L-max thresholding where L=1. Naturally, within each section, this suffers from the same problem as L-max thresholding, although the chance of this is reduced due to the segmentation of the vector. An example of L-wta thresholding is given in Table 4, using differing values for the lengths of each section in order to retrieve different output vectors.

It can be seen that all three of the thresholding methods has the potential to generate the same output vector, depending on the parameters used (Table 1c, Table 2c, and Table 4c). As such, the decision as to which method to use falls down to the application for which the CMM is to be used. Willshaw thresholding has the potential to be the fastest method available, due to the availability of an absolute threshold that need not be calculated in relation to the output pattern. On the other hand, for pattern recognition, L-max has been shown to be better at recalling vectors given an input that does not exactly match a trained pattern [7]. Finally, if the CMM is able to be trained using Baum generated vectors, then it was shown in [21] that there is a potential for up to 15% improvement in the capacity of the CMM by the use of L-wta.

2.5.4 Capacity

The allocation of vectors is a domain-specific problem, yet it greatly affects the potential capacity of a CMM. The number of fixed weight vectors that can be created by setting k bits given a vector length n is given by the equation for the binomial coefficient:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

Equation 3: n choose k

In the case of non-distributed vectors, where a vector has a weight of 1, this simply results in n possible vectors. Using non-distributed vectors would mean that each bit within a vector represents an entire input or output pattern. This is infeasible for two reasons; due to inefficiency with regard to the storage capacity, but more importantly the lack of ability to generalise given a noisy input.

A distributed representation is therefore typically used [4]. In order to maximise the number of possible vectors, k can be chosen such that:

$$k \cong \frac{n}{2}$$

Equation 4: Maximising n choose k

However, although this will allow the largest number of patterns to be represented by a fixed weight and fixed length vector, it does not necessarily follow that this will result in the highest capacity for a given CMM. As discussed by Austin and Stonham:

> [An] associative memory has a finite storage ability, but unlike a conventional listing memory, recall and storage do not suddenly fail as soon as its rated capacity is reached, but instead they gradually fail as its rated capacity is reached. [7]

This is a probabilistic process, such that if the capacity of a CMM is surpassed, there is still a probability of correctly recalling each of the associated vectors. Given in terms of the probability of failure to perform a perfect recall:

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

Equation 5: Probability of recall failure

where P is the probability of failure, R and I are the length and weight of the input vector respectively, H and N are the length and weight of the output vector respectively, and T is the number of associated vectors. Equation 5 is derived fully in Appendix 1 of [7]. This equation is highly conservative, and a deeper analysis of the recall abilities within a CMM is given in [44] that returns a lower probability of error. This is unnecessary for the purposes of this review, however, where a pessimistic value will be sufficient.

Vector weight	Number of vectors trained				
	1	2	3	4	5
2	0.016	0.060	0.125	0.205	0.293
3	0.007	0.050	0.140	0.270	0.421
4	0.007	0.073	0.243	0.482	0.704
5	0.010	0.149	0.487	0.801	0.949
6	0.022	0.351	0.828	0.982	0.999

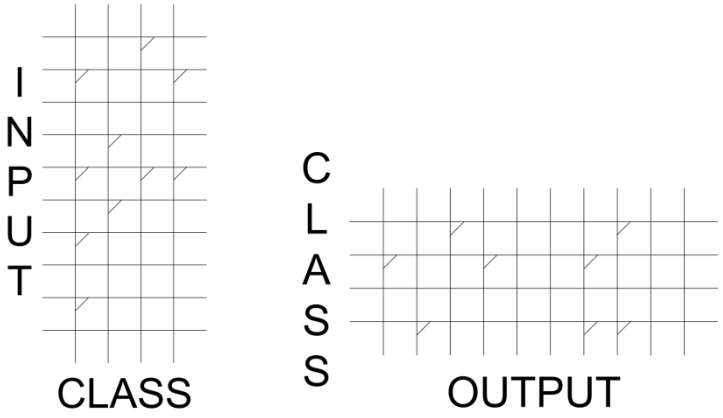
Table 5: Probability of recall error given a vector length of 10

It can be seen in Equation 5 that the probability of recall failure, and hence the potential capacity of a CMM, is directly affected by the ratio of length to weight for both input (I/R) and output (N/H) vectors. Table 5 contains values calculated for P, given that the input and output vectors have the same weight and a constant length of 10. With very few vectors trained, the probability of failure is smallest with a vector weight that is close to half of the vector length. It is clear, however, that as the vector weight increases, so does the rate of increase in P, given further trained vectors.

This is also the intuitive result: as the weight of the input or output increases, so more correlations will be stored in the CMM for every pair of input and output vectors that is trained. As the number of correlations stored in the CMM increases, so too does the probability that the presentation of a particular input will cause extra neurons to give an output. Initially this should not pose a problem, as the thresholding will be able to obtain the correct output vector with unexpected values in the output pattern. As the number of extra neurons that fire increases, however, so does the possibility that one of the values in the output pattern matches or even surpasses that of the correct output which will then threshold to give an incorrect output.

2.5.5 Pattern recognition using a two-layer correlation matrix memory

One of the important benefits of pattern recognition using a neural network is the ability for a network to generalise. It is common for a neural network to be trained to recognise a particular feature through the use of similar training samples [36]; the neural network is then expected to be able to recognise this feature within an input, even when the input that is presented differs from any of the trained examples.



a: 1st correlation matrix memory b: 2nd correlation matrix memory

Figure 12: Two-layer correlation matrix memory

It is shown in [7] that CMMs, as a specific class of neural network, also share this ability – in particular with regards to pattern recognition. Instead of simply associating two vectors directly in a single CMM, a two-layer associative memory is used (Figure 12). In the process that is described, the input vector is associated with an internally generated fixed-weight 'class' vector in the first CMM (Figure 12a). This class vector is then associated with the output vector within the second CMM (Figure 12b). A recall is performed in the same way as with a single-layer network, except that the thresholded output of the first CMM is then presented as an input to the second CMM.

In addition to the ability to recognise partially occluded patterns on an input, using a two-layer network has the potential to reduce the memory required in order to save the associations between large patterns [7]. In software implementations, such as that of the Advanced Uncertain Reasoning Architecture (AURA), a sparse matrix representation can be used in order to save only those values in the CMM that have been set [14]. In a hardware implementation, however, this is a lot more difficult and processing speed can be greatly reduced [43].

Assuming matrices are stored in their entirety, the number of bits required for a single-layer CMM to associate an input vector of length R with an output vector of length H would be:

$$Number\ of\ bits\ required = RH$$

Equation 6: Bits required in a one-layer CMM

In contrast, the number of bits required when using a two-layer network, where the internally generated vector has a length of C is given by:

$$Number\ of\ bits\ required = RC + CH$$

Equation 7: Bits required in a two-layer CMM

Using a relatively small value for C, Equation 7 can give a far smaller memory requirement than Equation 6. If the value chosen is too small, however, it will limit the storage capacity of the neural network, and so this is a further consideration when designing a two-layer CMM for a particular task.

2.5.6 Applications and alternatives

A large amount of literature exists with regard to neural networks and CMMs that describe various methods to perform pattern recognition with visual data (for example [7], [35], and [36]), or improve the performance – accuracy, speed, or capacity – of such a neural network.

This may serve to mask the potential applicability of these systems and techniques to a wide range of pattern matching problems. A few examples of the domains to which CMMs have been applied include chemical similarity searching [26], spell checking algorithms [22], and inference in expert systems [6]. These are essentially variations on a more general application of data mining within a knowledge base, with a domain-specific data representation and pre-processing and post-processing requirements.

A common alternative to using a CMM for these purposes would be to use a multi-layer neural network. This would provide the requisite pattern matching, and also generalise well to unknown or noisy inputs, however the major problem with this approach would be long training times [3]. Although [36] presents an online training algorithm for back-propagation, this only solves the requirement that all training of the neural network occurs prior to recall – training a new pattern is still a time-consuming process.

Using a CMM to search a knowledge base is also a lot more efficient than a conventional approach using a list, in terms of both time and space complexity [6]. CMMs are particularly well suited to searching unstructured data – a listing approach may be able to provide $\mathrm{O}(\log n)$ time complexity on structured data, but this degenerates to $\mathrm{O}(n)$ with unstructured data. CMMs, on the other hand, are able to search any knowledge base with only a single pass through the network.

2.6 Rule-chaining in correlation matrix memories

Systems using propositional logic within neural networks have been designed and implemented with various underlying technologies (for example [9] and [28]). In addition, the theory regarding the use of CMMs in particular for propositional logic and inference has been developed in [6] and [29].

2.6.1 Rule-chaining

As described in Section 2.2, rule-chaining is a method of inference also known as forward chaining. A system that performs forward chaining is often referred to as a production system. Using the pattern matching abilities of a CMM discussed in Section 2.5.5, rule-chaining in CMMs should be reasonably simple to implement. There are, however, a number of issues that must be resolved for the approach to be successful [1].

Rule-chaining is effectively a tree search, finding a path from an initial state to a goal. As such, anything described within this section may actually be adapted and applied to other problems involving a tree search.

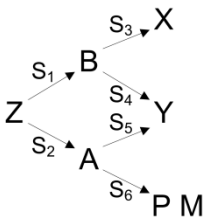
As an illustration of rule-chaining, consider the rules in Figure 13 (adapted from [2]):

1. $Z \rightarrow B$
2. $Z \rightarrow A$
3. $B \rightarrow X$
4. $B \vee A \rightarrow Y$
5. $A \rightarrow P \wedge M$

Figure 13: Example rule set

Any rules that contain a disjunction in the antecedent are separated into multiple rules, as shown in Figure 14:

1. $Z \rightarrow B$
2. $Z \rightarrow A$
3. $B \rightarrow X$
4. $B \rightarrow Y$
5. $A \rightarrow Y$
6. $A \rightarrow P \wedge M$



a: Set of rules b: Search tree, given an initial state of Z

Figure 14: Example rule set without disjunctions

Each branch of the search tree is labelled (S_1, S_2, etc.) with what will be referred to as a state token. These represent the separation of paths within the tree – for instance there are two routes from Z to Y, but they must remain independent during the search, especially if the aim of the search is to find the path to a goal rather than the goal itself.

2.6.2 Correlation matrix memories

When performing a tree search, there may be a large number of intermediate states to investigate between the initial state and the goal. In the most basic system, a search could be performed in various ways – the most common being depth-first and breadth-first. Both of these search methods have the disadvantage of a potential $O(n)$ time complexity.

In a conventional system, parallelism of a depth-first search could be used to improve this. At every branch, for instance, a new process could be created in order to continue the search along every possible path simultaneously. This would reduce the time complexity to $O(d)$, where d is the maximum depth of the tree. Other problems would be posed, however, such as the requirement to maintain a large number of separate search processes. In addition to this, if the problem posed is one of finding the shortest route from the initial state to a particular goal state, then it cannot be guaranteed that the first route that is found is indeed the correct route – due to potential processor scheduling constraints.

Using a CMM to perform rule-chaining allows for the proposed parallelism of the depth-first search, whilst solving the issues that a conventional system would face. Operation of the CMM for this purpose is no different to a generic CMM as described in Section 2.5. Each token is assigned a unique vector, as shown in Table 6 (in this example using a vector length of 8, vector weight of 2, and generated using Baum's algorithm):

Data token	Vector
Z	10010000
B	01001000
A	00100100
X	10000010
Y	01000001
P	00110000
M	10001000

Table 6: Rule set with assigned vectors

In the case that a rule contains a conjunction, either in the antecedent or the consequent, each individual token is assigned a unique vector and the conjunction is simply the logical OR of these. For example, rule 6 within Figure 14a contains the conjunction $P \wedge M$. Using the vectors assigned in Table 6, the conjunction would be represented as 10111000.

Training of the CMM is performed as in Section 2.5.1, with the only difference being the possibility that input or output vectors may not all have the same weight due to the existence of conjunctions. A CMM trained with the example rule set is shown in Figure 15.

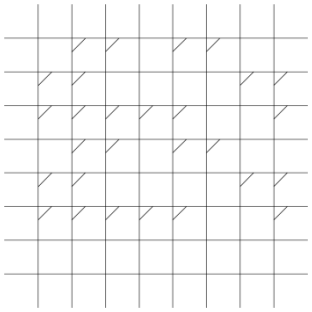


Figure 15: Correlation matrix memory trained with example rule set

Input vector	Output vector
10010000	01101100
01101100	11111011

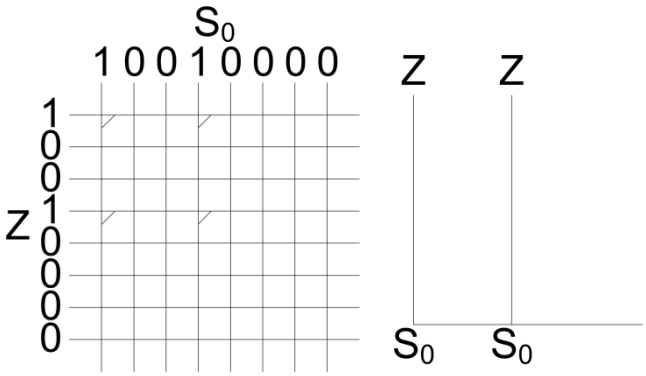
Table 7: Tree search performed on the example

It can be seen in Table 7 that the tree search was indeed conducted rapidly and in parallel, as only two passes through the CMM were required. It is also clear, however, that although the final result is technically correct (the logical OR of tokens X, Y, P and M is in fact 11111011) it is not useful. It is impossible to distinguish which of the tokens comprise the final result, the only certainty is that the vector for A is not included.

This then is one of the problems posed by performing rule-chaining within a CMM – to maintain the separation of multiple tokens within a state to allow them to be individually recognised [1]. The only viable solution to this particular issue is the use of tensor products, as described in the following section.

2.6.3 Tensor products

A tensor product is the result of binding two vectors together with an outer product operation [1]. The result is a matrix containing the correlations between those two vectors – effectively it is a CMM, as shown in Figure 16a. A simplified block diagram of a tensor product is shown in Figure 16b, and the shorthand $Z{:}\,S_0$ represents the tensor product of vectors Z and S_0.



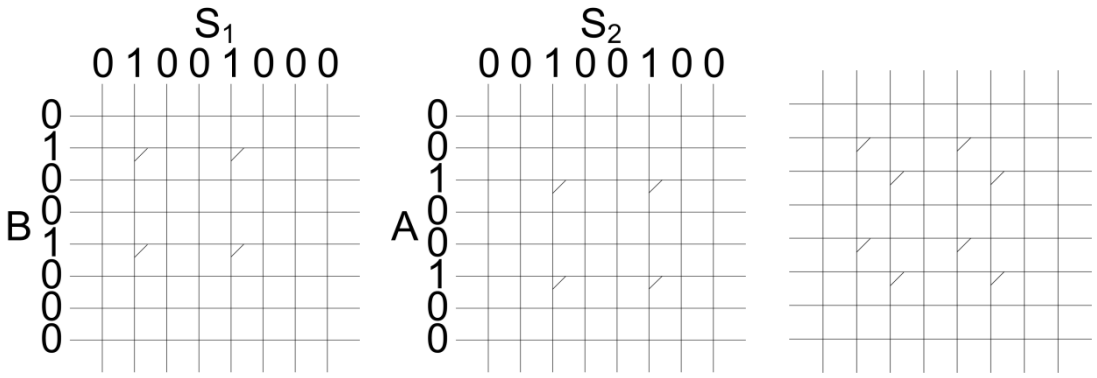
a: CMM representation b: Block representation

Figure 16: Tensor product of token Z with state S_0

An ideal vector with which to tensor a data token within this system is the state token – these are the labelled branches taken from Figure 14b. Each state token can be assigned a unique vector, however as the data tokens and state tokens are entirely separate it is acceptable for a state token to be assigned the same vector as a data token. The example set of states with assigned vectors is given in Table 8.

State token	Vector
S_0	10010000
S_1	01001000
S_2	00100100
S_3	10000010
S_4	01000001
S_5	00110000
S_6	10001000

Table 8: State tokens with assigned vectors



a: Tensor product of token B with state S_1 b: Tensor product of token A with state S_2 c: Superposition of tensor products

Figure 17: Superposition of tensor products

A superposition of multiple tensor products remains as simply the logical OR, however as can be seen in Figure 17c, it is now possible to visually distinguish each of the data tokens. More importantly, it is also possible for the system to distinguish each of the data tokens whenever this is required. This requires a two-stage process, as follows:

1. Determine the state tokens that are present in the tensor product
2. Use these state tokens to perform what is effectively a recall operation on the tensor product to determine the data tokens

The use of tensor products resolves the original issue – to allow for the superposition of multiple tokens, whilst maintaining the ability for each of the tokens to be individually recognised. Unfortunately it also exacerbates the issue of matrix size, originally discussed in Section 2.5.5.

Length	Weight	Section lengths	Maximum possible vectors
21	2	{10, 11}	110
64	2	{31, 33}	1023
201	2	{100, 101}	10100

Table 9: Lengths and weights to allocate a given number of vectors

Consider, for example, a system designed to store 100 rules, containing a possible 1000 tokens. Table 9 shows the vector lengths required in the best possible case, and using Baum's algorithm to allocate vectors to each of the state and data tokens. Each input and output vector would have a length of $21 \times 64 = 1344$ bits, meaning the CMM would be $1344^2 = 1806336$ bits, or around 221 kilobytes.

Although 221 kilobytes is not particularly large, consider now a system designed to store 1000 rules, containing a possible 10000 tokens. Each input and output vector would have a length of $64 \times 201 = 12864$ bits, meaning the CMM would be $12864^2 = 165482496$ bits, or around 20 megabytes. This is two orders of magnitude larger, and the trend continues in a virtually linear fashion as the number of rules and tokens increases, to the extent that such a system is infeasible for use with large datasets.

There has not been a large amount of research into this area, and as such only one viable solution has been proposed – to separate the storage of antecedents and consequents into different matrices [1].

2.6.4 Two-layer correlation matrix memory

Figure 18 shows a simple block diagram representation of the operation of a two-layer implementation, where the states labelling each of the branches in Figure 14b are used as a link between the first and second CMMs. This does not have to be the case, but is a simple method of assigning a unique label to each rule.

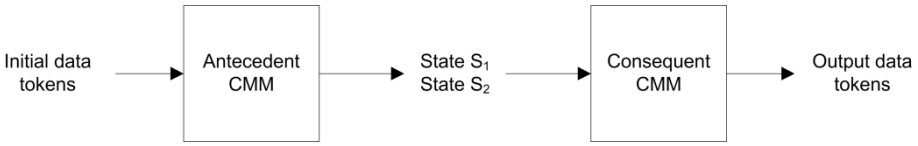


Figure 18: Simple two-layer CMM

The output of this system is designed to be in a format ready to be passed immediately back into the antecedent CMM in order that rule-chaining occurs with only a very simple control system.

1. $Z \rightarrow S_1 \rightarrow B$
2. $Z \rightarrow S_2 \rightarrow A$
3. $B \rightarrow S_3 \rightarrow X$
4. $B \rightarrow S_4 \rightarrow Y$
5. $A \rightarrow S_5 \rightarrow Y$
6. $A \rightarrow S_6 \rightarrow P \wedge M$

Figure 19: Two-layer example rule set

In order to train the two-layer network, the rule set must be modified in order that each rule contains an antecedent, a consequent, and additionally an intermediary state. The previously used example rule set with this adaptation is given in Figure 19.

Antecedent correlation matrix memory

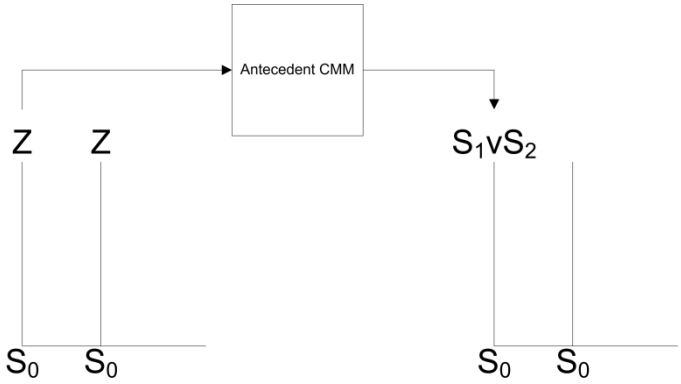


Figure 20: Recall of a tensor product

Training of the first CMM proceeds in the same way as it would with any other single-layer CMM system described in Section 2.5.1, using the antecedent as an input and the state as an output. Recall from this CMM is also performed using a standard method, as described in Section 2.5.2, except that the input in this system is a tensor product rather than just a vector. It is proposed in [1] that each column of the tensor product is individually passed as an input to the CMM. This is shown in Figure 20, where the CMM has been trained with the example rule set. The output is labelled as $S_1 \vee S_2$ to indicate that the vector contains the logical OR of S_1 and S_2.

It is suggested that Willshaw's method of thresholding be used, with a value equal to the weight of the input vector during training. This will ensure that all rules that have been trained will be recalled, although if the CMM is heavily populated it is possible that extra rules will also fire [6].

Consequent correlation matrix memory

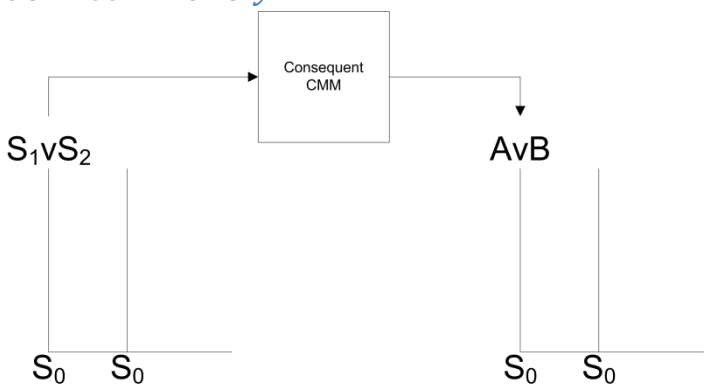


Figure 21: Undistinguishable recall of superimposed states

Training the second CMM is not quite as simple a process. As can be seen in Figure 20, the output of the first CMM will contain superimposed state vectors when two separate rules have the same antecedent. If each rule were to be trained into the CMM using the state as an input and the consequent as an output, then a recall operation on superimposed states would result in undistinguishable consequents (Figure 21).

1. $Z \rightarrow S_1 \rightarrow B : S_1$
2. $Z \rightarrow S_2 \rightarrow A : S_2$
3. $B \rightarrow S_3 \rightarrow X : S_3$
4. $B \rightarrow S_4 \rightarrow Y : S_4$
5. $A \rightarrow S_5 \rightarrow Y : S_5$
6. $A \rightarrow S_6 \rightarrow (P \wedge M) : S_6$

Figure 22: Two-layer example rule set with tensor products

The solution to this problem suggested in [1] is that the second CMM should instead store correlations between a state and the tensor product of the consequent with that state. To demonstrate this, the example rule set is adapted once more in Figure 22. Training the CMM is then a matter of presenting the state as an input and the tensor product of the state and the consequent as an output.

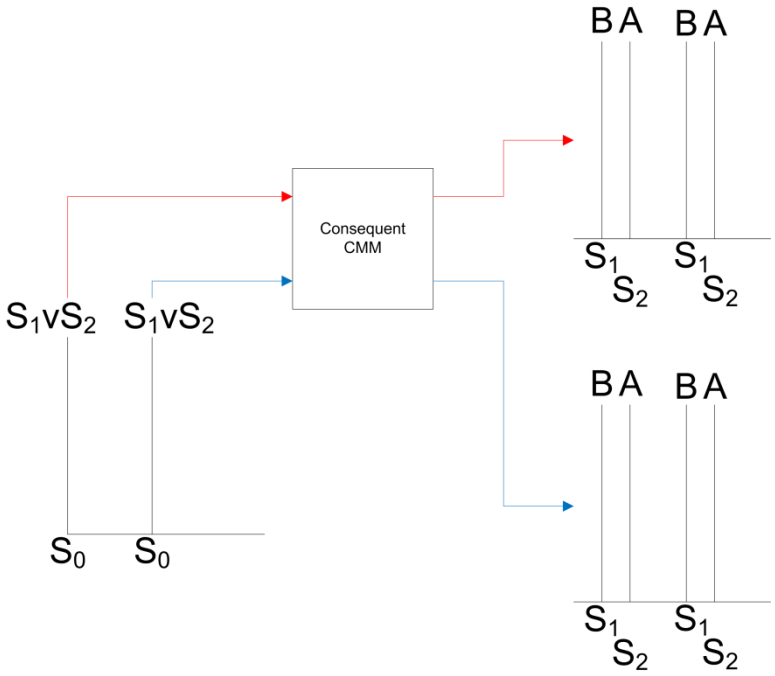


Figure 23: Correct recall of superimposed states

The recall operation for the second CMM is similar to that for the first CMM; each column of the state tensor product is passed as an input in turn. The output for each column, however, is an entire tensor product as shown in Figure 23. A threshold is applied to each of these output tensor products, and then they are summed – taking a logical 1 to be the value 1 [1]. A threshold can then be applied to the result of this sum, using Willshaw's method with a value equal to the weight of a state vector. This final tensor product can then be directly applied as an input to the antecedent CMM, in order to continue the rule chaining.

Complete rule-chaining system

Combining the antecedent and consequent CMMs gives the system shown in Figure 24. The state and data tokens used are taken from the example rule set, using an initial input of $B : S_1$ superimposed with $A : S_2$ in order to demonstrate concurrent recall of multiple tokens.

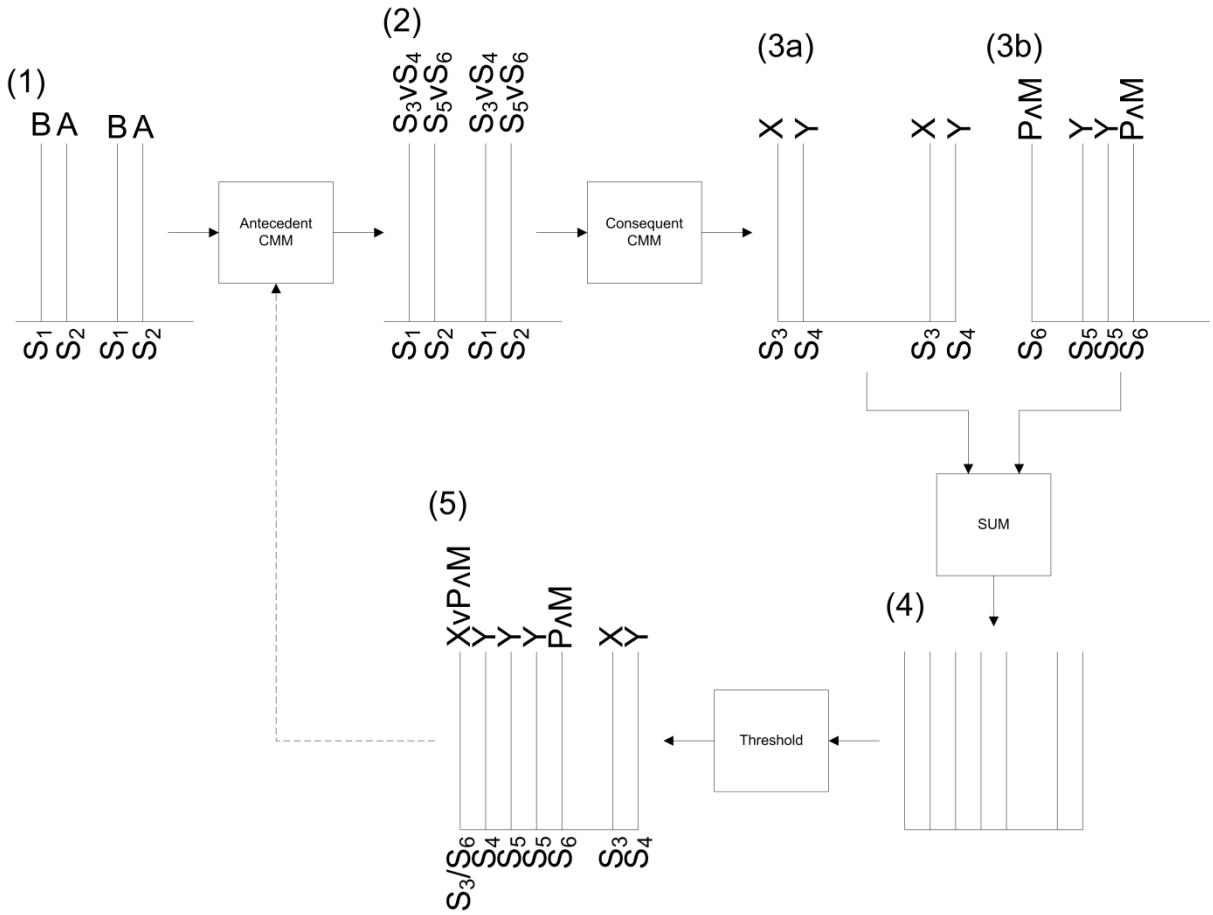


Figure 24: Complete two-layer CMM rule-chaining system (adapted from [1])

The operation of the system is as follows, where numbers refer to Figure 24:

1. A tensor product (1) is presented as an input to the antecedent CMM
2. A threshold is applied to the output of this CMM (2), which is then presented as an input to the consequent CMM
3. Each column in (2) will generate a complete tensor product, for example $S_3 \vee S_4$ will generate (3a) and $S_5 \vee S_6$ will generate (3b); a threshold is applied to each of these individual tensor products
   As there are two copies of $S_3 \vee S_4$, there will be two copies of (3a) generated, and likewise for (3b) – this duplication is not shown in the figure
4. All of the output tensor products are summed, with a logical 1 within a tensor product equating to the value 1 for the purposes of addition (4)
5. A threshold is applied to this tensor product, to retrieve the final output (5)
6. In order to continue with rule-chaining, (5) can be presented as an input to the antecedent CMM

Arity networks

A two-layer CMM is able to generalise and recognise partial inputs, as discussed in Section 2.5.5. When applied to pattern matching, this can be a useful feature; for the application of rule-chaining, however, this is undesirable.

To illustrate the problem, consider a system containing the rule set in Figure 25. If the system is presented with an input of A, then rules 5 and 6 would be expected to match. In order for this to be the case, the threshold value would need to be set to the weight of a single input token. This would also allow rule 7 to partially match, and generate an output containing D [1].

1. $Z \rightarrow B$
2. $Z \rightarrow A$
3. $B \rightarrow X$
4. $B \rightarrow Y$
5. $A \rightarrow Y$
6. $A \rightarrow P \wedge M$
7. $A \wedge C \rightarrow D$

Figure 25: Example rule set for arity networks

The solution proposed by [1] is to use a system containing multiple arity networks (Figure 26) Prior to training a rule into the system, the arity is checked and the rule is trained into the appropriate CMM.

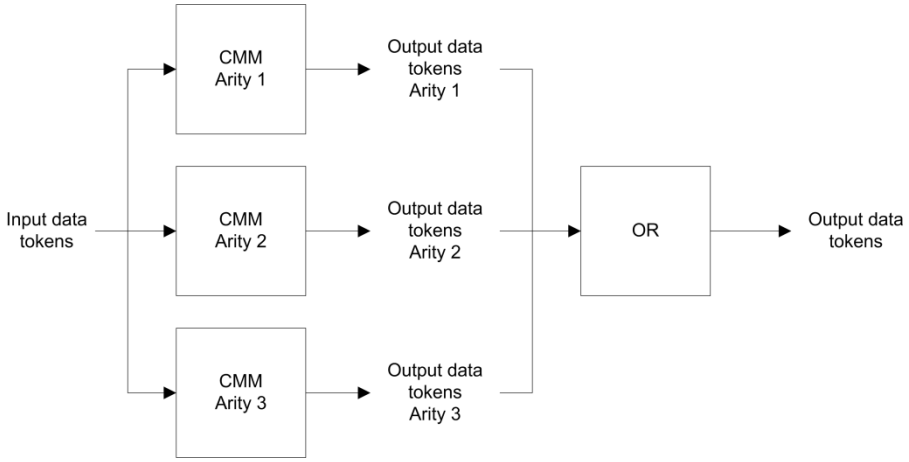


Figure 26: Arity networks

Recall from all of the CMMs is able to be executed in parallel, and each of the different CMMs can threshold their outputs with a different value appropriate for their arity. Superimposing the outputs of the various CMMs gives the final output, containing no partially matched rules. This output can then be presented as an input to the system if required.

2.7 Final considerations

In terms of processing power, computer hardware is becoming increasingly more powerful. However, some applications are always pushing the limits of the available technology – such as the artificial intelligence that provides control of machine-controlled characters within video games [23].

In order to make the best use of available resources, novel architectures have been developed for various purposes. The rule-chaining system proposed in [1] is an example of such an architecture, offering a potential improvement that could be applied to many domains that use production and expert systems. The following chapters implement and evaluate this proposed architecture, and discuss further work necessary to prove or disprove the validity of the claim that the production system provides an improvement over classical alternatives.

3 Initial planning

It was decided that an iterative development process should be used for this work, as it allowed the addition of functionality in an incremental fashion. Careful planning was required in order to successfully apply the iterative development process to this work. This planning stage was used to decide on the experimentation to be performed during each iteration, and hence the functionality required. Performing an iterative development also aided the testing process, by simplifying the implementation at each stage

Although the AURA library was available for use within this work, it was decided that fully developing a simple implementation without the use of an external CMM library would result in a deeper understanding of the CMM rule-chaining architecture and the techniques used to improve recall reliability.

To this end, five stages were identified that would give a logical progression from the development of a simple CMM to the final demonstration system capable of rule-chaining and online training. The functionality required, and experimentation performed, at each of these stages is described in detail within the next five chapters:

4. Vector recall
5. Tensor product recall
6. Rule-chaining with tensor product recall
7. Superposition of tensor products
8. Use of a two-layer network

This series of stages initially develops a basic CMM capable of rule firing, and gradually incorporates the required features of superposition, rule-chaining, and multi-layer networks. The full source code developed in this work is provided on the attached CD, with header files of classes and functions given in Appendix 1.

4 Vector recall

4.1 Overview

This first iteration was intended to demonstrate the operation of a simple CMM, with the ability to train and recall vector correlations. As such, this work experimented with the storage capacity of a CMM that has input and output vectors with a fixed length and weight. As well as verifying the correct operation of the CMM, this work investigates the effect of changing the threshold method used and different means of generating vectors, to determine the effects of using Baum's algorithm rather than a random generation.

4.2 Design

As described in Section 2.5 and [1], a binary CMM is required to store the correlations between an input and an output vector. A CMM is therefore a two-dimensional $m \times n$ matrix which may be stored with a Boolean data type, as a location within the matrix may be either set (at logical 1) or unset (at logical 0).

Creating a class for the CMM allows the main operations of training and recall to be properly encapsulated. In preparation for later iterations of the development process it also allows the use of two separate CMMs, as training or recall operations on one matrix will be independent of any other matrix.

Neural networks and CMMs are known to be applicable to uncertain and fuzzy reasoning, including applications such as pattern recognition [44]. For the purposes of inference, it may be desirable for a system that is capable of reasoning under uncertainty. Alternatively, it may be preferred that the system is able to be certain that any derived consequences are logically entailed by the provided antecedents. As such, the experimentation performed used the premise that an incorrect recall was defined as an output vector that did not exactly match the trained vector.

This work consisted of two experiments, firstly to determine the probability of recall error for a CMM with a fixed input and output vector length and weight, and then to investigate the capacity of a CMM with varying vector weights. As a corollary, this also investigates the effect of changing the methods used for generating vectors and applying a threshold after a recall operation.

Both experiments in this iteration were run 20 times, with the mean results used for evaluation.

4.2.1 Probability of recall error

Using a random vector generation gives no guarantee as to the Hamming distance between vectors. With an improved Hamming distance, as provided by using Baum's algorithm [8] (described in Section 2.5.3), the capacity of a CMM is expected to increase as the interference between vectors is decreased.

It is important that throughout this work all data is represented by a distributed pattern, in order to achieve an efficient use of memory [1] and greater fault tolerance [8]. To this end, the weight used for the input and output vectors is set at 5, as this is the vector weight used within the literature [21]. A vector length of 500 has been selected, in order that the number of vector pairs that may be successfully trained and recalled from a matrix is sufficiently large to allow results generated with different parameters to be distinguished.

As Baum's algorithm is deterministic, different seed values are used for each execution of the experiment. It was decided that although it has been shown that using L-wta to threshold can improve the storage capacity of a CMM [21], it would not be appropriate for this application. When performing rule-chaining, it is likely that there will be multiple rules with the same antecedent. Where this is the case, the output should contain both consequents superimposed. Using L-wta could potentially favour one of the rules over another, resulting in an undesired pruning of the search space. For this reason, only L-max and Willshaw's thresholding will be compared.

In this experiment, a recall error is defined as an output vector that does not exactly match that which was trained. The operation of this experiment is as follows:

1. Create two identical matrices – one to store randomly generated vectors, and one to store vectors generated using Baum's algorithm
2. Randomly generate a new input and output vector pair, and ensure that they are unique within the randomly generated system
3. Train the first CMM with this vector pair
4. Generate a new input and output vector pair, using Baum's algorithm
5. Train the second CMM with this vector pair
6. Recall every previously trained input vector of the first CMM in turn, comparing the output vector to that which was trained
    a. Threshold the output pattern using Willshaw's method
    b. Threshold the output pattern using L-max
7. Recall every previously trained input vector of the second CMM in turn, comparing the output vector to that which was trained
    a. Threshold the output pattern using Willshaw's method
    b. Threshold the output pattern using L-max
8. Record the rates of recall error and continue from 2 if any of the results has a rate of recall error less than 100%

4.2.2 Changing vector weight

Using input and output vectors with a greater weight will increase the number of correlations to be stored within the CMM for every vector pair that is trained. It is possible, however, that this increased number of correlations required to meet a threshold may serve to counteract the interference within the matrix.

As such, this experiment is designed to investigate the effect of increasing the weight of vectors stored within a CMM. The results of this experiment may then influence the decision as to the vector weight used for future development iterations. The operation of this experiment is as follows:

1 Create an empty matrix
2 Randomly generate a new input and output vector pair, ensure that they are unique within the system, and train the matrix with this pair
3 Recall every previously trained input vector in turn, using Willshaw's method to threshold and comparing the output vector to that which was trained
4 Continue from 2 if none of the recalled vectors were erroneous, otherwise increase the vector weight and continue from 1

4.3 Implementation

The CMM is implemented as a class, containing various fields and methods. Using a class in C++ also allows for a destructor to be defined, which is used to free any memory that has been dynamically assigned for the CMM.

Vectors and output patterns (before a threshold is applied) are defined as structures, in order to incorporate important parameters such as their length as well as an array. They are not required to be full classes, as they have no methods that require encapsulation.

A class named vectorGenerator was created in order to create vectors to be stored in a CMM. Although this could have been implemented using functions rather than a class, it was decided that the use of a class would simplify any experimentation. Once the class is instantiated, generating a new vector is a matter of calling a method within that object.

Finally, a rule structure and rules class were created to store any rules that are trained into a particular CMM. This is to allow the system to compare a vector that is recalled from a CMM with the vector that was originally trained; effectively this is using the classical rule storage method of a list in order to test the correct operation of a CMM.

With regards to the implementation of experiments, only a simple control structure is required – firstly defining any variables to be used, and then using loops to train and recall vectors in the CMM.

4.4 Testing

Unit testing was undertaken to ensure that all classes and functions were correctly implemented. Each unit was tested with various test cases, designed to confirm the correct operation of the unit, with the expected results being manually calculated before each test was executed. The complete set of test cases for this development iteration are given in Appendix 2.

Having ensured that all units operated as expected, testing the implementation of the experiments required using the debug facilities of Visual Studio to ensure that the control loops were executed as expected and that variables were declared within the correct scope.

4.5 Results

4.5.1 Probability of recall error

When displaying the probability of recall error from a CMM, it is common to give the number of vector pairs that may be stored and recalled with various probabilities of error [21]. This is due to the possibility that a given probability of error is acceptable within a system.

Table 10 shows the number of vector pairs that may be stored to achieve the probability of error given as a column heading, using different methods of vector generation and threshold. The full results are represented graphically in Figure 27.

	0% error	0.1% error	1% error	5% error	10% error
Random generation, Willshaw threshold	424	650	1081	1605	1902
Random generation, L-max threshold	424	650	1081	1605	1902
Baum generation, Willshaw threshold	2668	2669	2681	2736	2808
Baum generation, L-max threshold	2668	2669	2681	2736	2808

Table 10: Probability of recall error

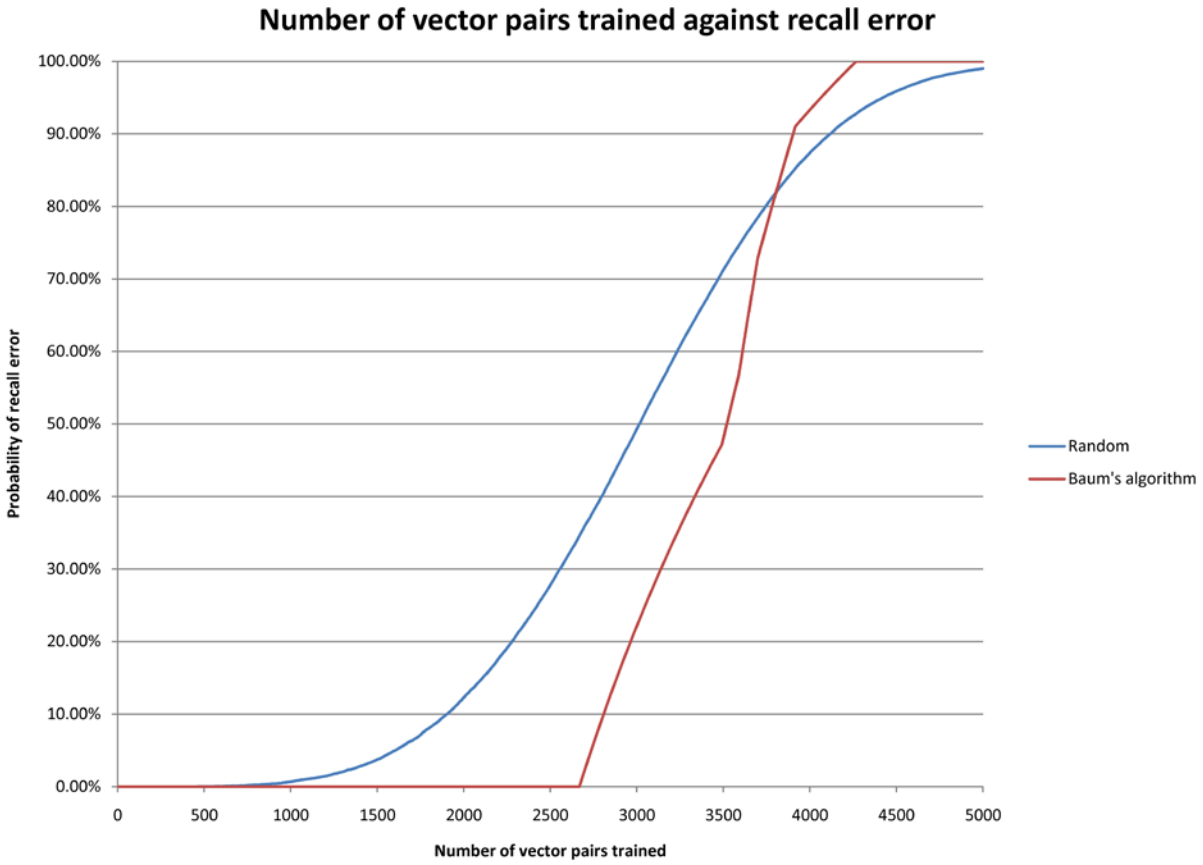


Figure 27: Probability of recall error

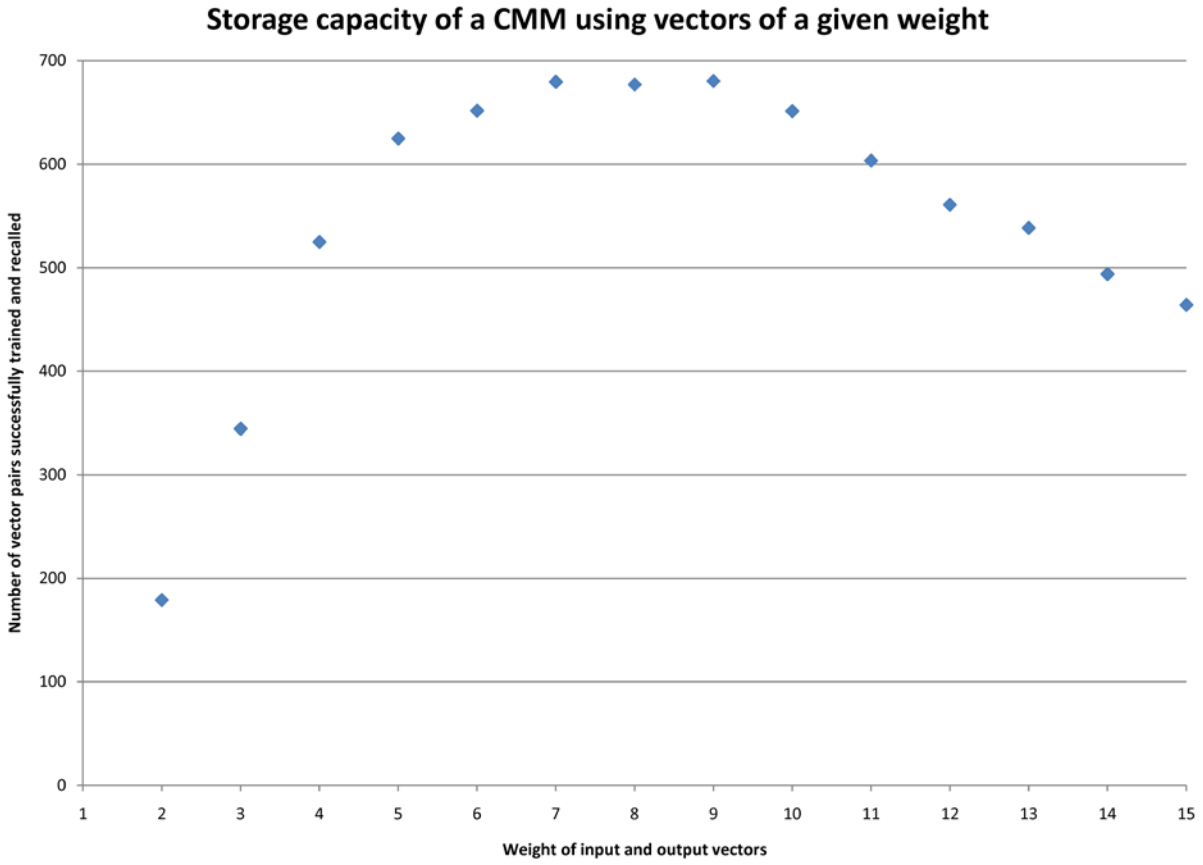


Figure 28: Number of vector pairs recalled against vector weight

4.5.2 Changing vector weight

This experiment was designed only to investigate the capacity at which a CMM failed to achieve perfect recall reliability, and as such Figure 28 shows the number of vector pairs that may be stored before any recall fails. A table of these results is given in Appendix 3.

4.6 Evaluation

It was expected that generating vectors using Baum's algorithm would increase the storage capacity of a CMM, due to the increase in Hamming distance between vectors leading to a reduction of interference. As can be seen in Table 10, and more clearly in Figure 27, this is indeed the case.

It can also be observed from the graph that although a CMM storing randomly generated vectors fails earlier than one storing vectors generated using Baum's algorithm, it also fails at a slower rate. Due to the determinism of Baum's algorithm, this is to be expected; when the CMM reaches capacity, it is guaranteed that every further vector pair trained into the CMM will cause interference with at least one vector pair that has previously been trained.

Somewhat unexpectedly, the threshold method applied did not affect the recall reliability of a CMM, as both L-max and Willshaw's methods achieved exactly the same probabilities of error. This inability to distinguish the threshold methods is the reason that Figure 27 shows only two lines; one line for randomly generated vectors, and one for vectors generated using Baum's algorithm. Threshold methods are discussed in [7], where it was found that when presented with a noisy input, using L-max gave a higher recall reliability than using Willshaw's method. In this experiment all inputs are exactly as they were initially trained, and so L-max does not provide any benefit.

The results of the second experiment show that the vector weight used affects the storage capacity of a CMM in the way that was expected. Although using a weight of 2 creates the least correlations to be stored in the CMM, it is far more likely for interference from other trained vector pairs to cause an additional position with a weight of 2 in the recalled output pattern thereby causing an extra bit to be set in the output vector after a threshold is applied. The capacity of the CMM therefore increases with the vector weight until the point that the number of correlations stored in the CMM, and the interference caused by this, outweighs the ability of a threshold to remove any incorrectly firing neurons. The results obtained agree with previous work on this area [3], which found that using vectors with a weight equal to $\log_2$ of their length maximises the storage capacity of a CMM.

5 Tensor product recall

5.1 Overview

A tensor product is formed by taking the outer product of two vectors, and the use of tensor products allows superposition of data tokens within the input or output of a CMM [1]. Having demonstrated the correct operation of a basic CMM, this iteration was developed to extend the recall functionality to include tensor products.

In [3] it is suggested that a CMM be trained using stacked tensor products, i.e. each column of the tensor product is stacked on top of the others, resulting in a single vector. A recall simply requires presenting the stacked input tensor product. On the other hand, [1] proposes a CMM that is trained using only the original input and output vectors. In this process, the recall operation requires iterating through the tensor product, presenting each column as an input to the CMM.

Without the use of a sparse matrix representation, the first method is infeasible due to the very large amounts of memory that would be required. Later iterations of the development process will require a second CMM that is recalled by iterating through a tensor product. This means that although the use of the second proposed method requires more recall operations, any benefits in terms of speed provided by using the first method would be negligible because of the requirement for the second CMM to use the second method. As such, this work investigates the operation of only the second alternative.

5.2 Design

This iteration adds the functionality of recall from a CMM using a tensor product. This is an important aspect of the final system, as it is used to allow superposition of vectors within an input or output while maintaining the ability to distinguish them.

This work consisted of a single experiment, to compare the probability of recall error for a CMM using tensor product recall with one that used basic vector recall. As the recall operation using a tensor product is merely a vector recall applied multiple times, it was expected that the probability of error would also be the same.

5.2.1 Probability of recall error

From the results found in the previous iteration, it was clear that a vector length of 500, using Baum's algorithm to generate vectors, was sufficient to store a large number of vector pairs. The second experiment that was undertaken showed that with this vector length, a vector weight of around 7 to 9 would give the highest capacity. It was decided, however, that the capacity offered by using a vector weight of 5 would be more than sufficient for this experiment. This has the benefit of allowing the results from the first experiment to be used in comparison to the results of recall using a tensor product.

In the previous iteration it was found that L-max and Willshaw's methods of applying a threshold were able to provide equal CMM capacities with a perfect input. Given superimposed vectors, as is included within the development process at a later stage, their recall properties will differ. L-max is designed to be suitable for a recall operation with an output of a specific weight. For a simple vector recall this is ideal, however if multiple vectors are superimposed, the expected weight of the output is unknown. In this case, L-max has the potential to fail to recall all of the correct bits of the output

vector. On the other hand, with a suitable threshold value such as the weight of the input vector, Willshaw's method will never fail to recall all of the correct bits; instead extra bits known as ghost bits may be set in the output. For this reason, Willshaw's method was selected for use within this and later iterations of the development cycle.

A recall error is defined as an output vector that does not exactly match that which was trained. This experiment was run 20 times, and the operation of this experiment is as follows:

1. Create an empty matrix
2. Generate a new input and output vector pair, using Baum's algorithm, and train the matrix with this pair
3. Recall every previously trained input vector in turn, comparing the output vector to that which was trained
    a. Generate a new tensor vector, using Baum's algorithm
    b. Create the tensor product of the input and tensor vectors
    c. Recall the output tensor product, using Willshaw's method to threshold the output pattern
    d. Perform a recall operation on the output tensor product, using the tensor vector as an input, in order to calculate the output vector
    e. If the output vector is the same as that which was trained, then the output was correctly recalled
4. Record the rates of recall error and continue from 2 if the rate of recall error is less than 100%

5.3 Implementation

The classes, structures, and functions used in the first iteration were designed in order that they would be suitable for use throughout the development process. As such, the only modification required was an alteration to the recall method within the CMM class to allow a tensor product to be recalled.

Various additional functions were also required, specifically one to perform the tensor product operation on two vectors, and another to perform a recall operation on a tensor product in order to retrieve the output vector.

Very little modification was required to the code used in the first experiment of Chapter 4 in order for it to be suitable for this experiment, as the control structure remains the same. Aside from the removal of multiple methods of vector generation and threshold, the tensor product generation and recall was inserted.

5.4 Testing

Testing for all iterations will be largely the same, using unit testing to ensure correct implementation of any new code. The test cases used for this development iteration are given in Appendix 2.

The implementation of the experiment was also tested, again using the debug facilities within Visual Studio. Finally, the experiments from Chapter 4 were executed once in order that the results could be compared to those obtained previously. This was to confirm that the modification made to the

recall function did not have a detrimental effect on the operation of any previously developed functionality.

5.5 Results

5.5.1 Probability of recall error

Table 11 shows the number of vector pairs that may be stored to achieve the probability of error given as a column heading, using basic vector recall and tensor product recall. The results for basic vector recall are taken from Section 4.5.1.

	0% error	0.1% error	1% error	5% error	10% error
Basic vector recall	2668	2669	2681	2736	2808
Tensor product recall	2668	2669	2681	2736	2808

Table 11: Probability of recall error using tensor product recall

5.5.2 Experiments from Chapter 4

Table 12 and Figure 29 show the outcome of executing the experiments from the previous iteration, given the updated code. These experiments were run only once, as this is sufficient to allow a comparison to the previously obtained results.

	0% error	0.1% error	1% error	5% error	10% error
Random generation, Willshaw threshold	440	440	1017	1588	1893
Random generation, Willshaw threshold (previous)	424	650	1081	1605	1902
Random generation, L-max threshold	440	440	1017	1588	1893
Random generation, L-max threshold (previous)	424	650	1081	1605	1902
Baum generation, Willshaw threshold	2668	2669	2681	2736	2808
Baum generation, Willshaw threshold (previous)	2668	2669	2681	2736	2808
Baum generation, L-max threshold	2668	2669	2681	2736	2808
Baum generation, L-max threshold (previous)	2668	2669	2681	2736	2808

Table 12: Probability of recall error, with updated code

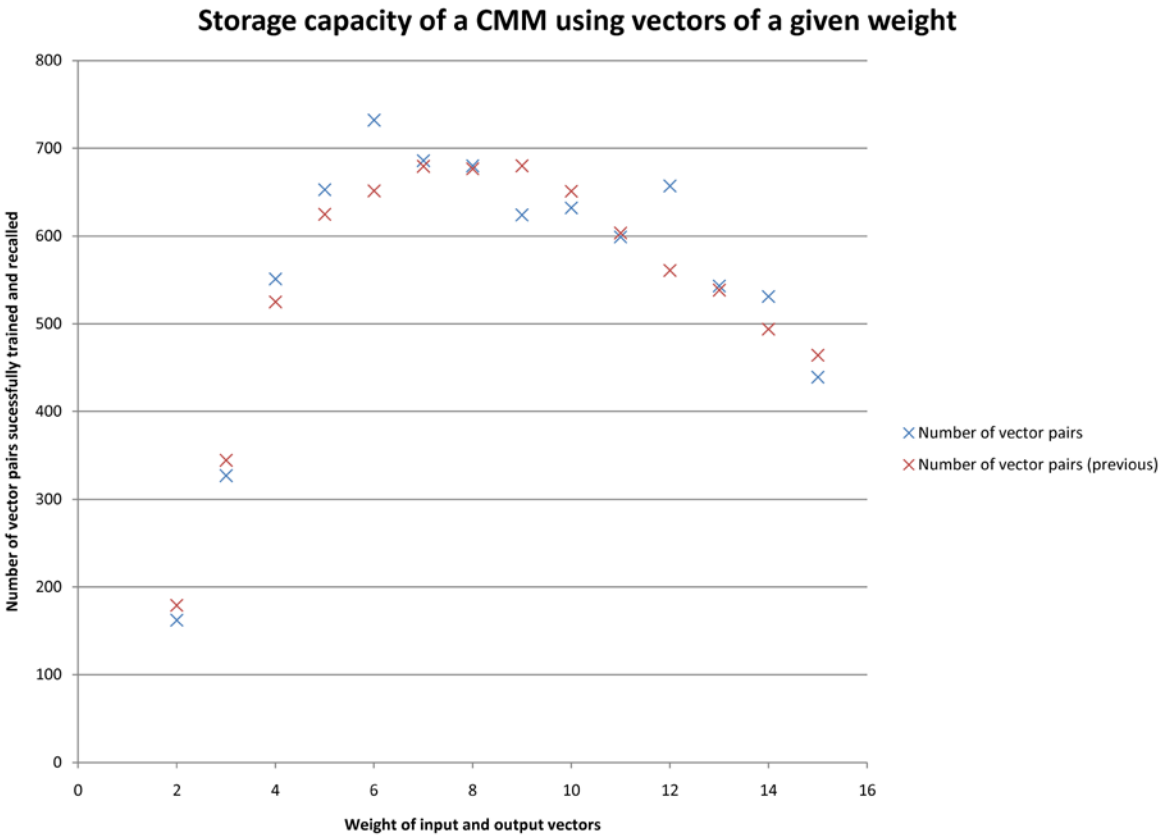


Figure 29: Number of vector pairs recalled against vector weight, with updated code

5.6 Evaluation

As expected, the results in Table 11 show that the storage capacity of a CMM is not affected by this method of tensor product recall. Due to the nature of this method, if a basic vector recall is successful then a tensor product recall will also be successful; if a basic vector recall is unsuccessful then a tensor product recall will be likewise.

The reason for this can be most easily seen in Figure 20, a diagram of the operation of the tensor product recall. As the tensor product is recalled, each input presented to the CMM is either a copy of the original input vector or an empty vector. Thus, each column in the output tensor product is either a copy of the same recalled output vector or an empty vector. Due to each output vector being exactly the same, this duplication does not aid in the accuracy of recall.

The execution of this experiment was also significantly slower than its counterpart from Chapter 4, due to each tensor product recall operation requiring multiple basic vector recalls. As previously discussed, however, this would not be an issue in the final system as the second CMM will require tensor product recall to operate in this fashion. An optimisation could therefore operate the two CMMs in parallel; rather than waiting for an entire tensor product to be recalled from the first CMM, each output vector of the first CMM could be passed immediately as an input vector to the second CMM.

The results of testing the experiments from the previous iteration (Table 12 and Figure 29) show that all previously developed functionality still operates correctly. The results are not an exact match; however they display the same characteristics as the previously obtained mean results. This is to be expected as each experiment was run only once and the vector generation is entirely stochastic.

6 Rule-chaining with tensor product recall

6.1 Overview

Rule-chaining is the term given to two methods of inference, forward chaining and backward chaining [13]. In this work we are interested only in forward chaining, or data-driven rule-chaining. As described in Section 2.2.1, this involves searching a set of rules in order to move from a set of known facts to a goal.

There is no extra functionality required in order to perform rule-chaining with a CMM. As such, this iteration is dedicated to the development of functionality that aids the process of rule-chaining. This includes the ability to associate a human-readable token with a particular vector, a file parser that can read rules from a text file, and a system capable of simple dialogue with a user.

6.2 Design

To associate a human-readable token with a particular vector requires an addition to the vector structure. Storing the token in a string allows the use of C++ string comparisons to test if two tokens are equal. In addition to this, the vector generation class requires a new method to generate a vector, accepting the token as a parameter and including this in the returned vector.

The ability to associate a token with a vector is only useful if the same token is always associated with a particular vector. As such, functions to search a set of rules for a particular token are essential. Having recalled a vector it is useful to be able to provide the human-readable token as an output, and so functions to search a set of rules for a particular vector are also important.

Finally, creating a function able to parse an input text file allows rules to be generated externally to this system. These rules may be generated either automatically or manually, and separating the rule generation from the main system helps to simplify testing.

This iteration does not contain any experimentation, rather just simple testing to ensure the rule-chaining operation works correctly in preparation for the next stage of development. As such, a known set of rules are trained into the CMM, and various antecedents are recalled.

6.3 Implementation

The addition to the vector data structure of a human-readable token, in the form of a string, does not have any effect on the functionality from any previous iteration. C++ supports method overloading, meaning it is simple to add an additional vector generation method that accepts the human-readable token as a parameter without needing to alter the original implementation.

Sets of rules are already stored within a class and so incorporating additional methods, rather than using external functions, is the most suitable implementation. The simplest methods return a vector data structure, as they are searching for only a single human-readable token or machine-readable vector. The remaining search method is intended for use during training of the CMM, in order to correctly assign the machine-readable vectors for a newly created rule.

A very simple structure was devised for the text file, consisting of a series of rules on separate lines, in the format:

```
a->b
```

Reading the text file is separated from parsing each rule, in order that future development iterations may use the rule parsing function from within a more advanced control structure.

The control loop required to operate this testing is only marginally more complex than that developed for previous iterations. Initially the text file containing any rules to be trained is parsed, and the rules that were contained within are trained into a CMM. The system then enters an infinite loop, requesting an input from the user of either "quit" or a token with which to perform a recall on the CMM. The output of a recall is interpreted and displayed to the user, and if the output vector is recognised, it is passed directly back into the CMM as an input to recall. Finally, there is an arbitrary limit on the number of recall operations that a single input may trigger, in order that a cyclic rule set will not cause an infinite loop.

6.4 Testing

As with previous stages in development, unit testing was performed to ensure the correct implementation of each new function before it was used in the main control loop. Appendix 2 gives the test cases used in this development iteration.

Although the control loop used in this iteration has more complexity than that used in previous iterations, there are no formal experiments for this development. As such, having separately verified each function, in order to test this it was sufficient to inspect the output and compare it to that which was expected.

The rule set used in testing was simple, consisting of every letter in the alphabet as an antecedent and the next letter as a consequent. Explicitly this is:

```
1.  a->b        14. n->o
2.  b->c        15. o->p
3.  c->d        16. p->q
4.  d->e        17. q->r
5.  e->f        18. r->s
6.  f->g        19. s->t
7.  g->h        20. t->u
8.  h->i        21. u->v
9.  i->j        22. v->w
10. j->k        23. w->x
11. k->l        24. x->y
12. l->m        25. y->z
13. m->n
```

Figure 30: Rule set used in testing rule-chaining

6.5 Results

A series of antecedents was used to test the rule set; these results show a small selection. The loop limit used was 30, to allow the recall of the entire rule set.

Antecedent	a	q
Recalled rules	a->b ->c ->d ->e ->f ->g ->h ->i ->j ->k ->l ->m ->n ->o ->p ->q ->r ->s ->t ->u ->v ->w ->x ->y ->z ->Output vector not recognised	q->r ->s ->t ->u ->v ->w ->x ->y ->z ->Output vector not recognised

Table 13: Results of testing rule-chaining

With an untrained input token of `input`, the result is:

```
Input vector not recognised
```

6.6 Evaluation

It is clear from the results displayed in Table 13 that the rule-chaining operates correctly; the assertion of a particular antecedent causes a looped recall until the output is an invalid vector. Although this is a somewhat contrived example, it does show that the approach used in this work is effective.

For a production system, an alteration to the operation of the control loop would be made for efficiency. Interpretation of the output tensor product is not required in order for the rule-chaining to continue, as the output tensor product is already in the correct form to be input directly to the CMM. Instead, the execution of this interpretation could be in parallel with the operation of the next recall loop.

7 Superposition of tensor products

7.1 Overview

As discussed in Section 2.6.3, and shown in Figure 16a, a tensor product is effectively a CMM. Forming the superposition of tensor products is similarly analogous to training a CMM with a new input and output. It is therefore to be expected that the capacity of a tensor product, in terms of the number of superimposed vectors that it may hold, will be similar to that of a CMM.

If a tensor product is too heavily populated it should not affect the ability of a CMM to recall correctly, as each of the input bits will still trigger a response from the correct neurons within the matrix. The issue will instead arise when attempting to detect a goal state, using either of the methods discussed in Section 2.2.1.

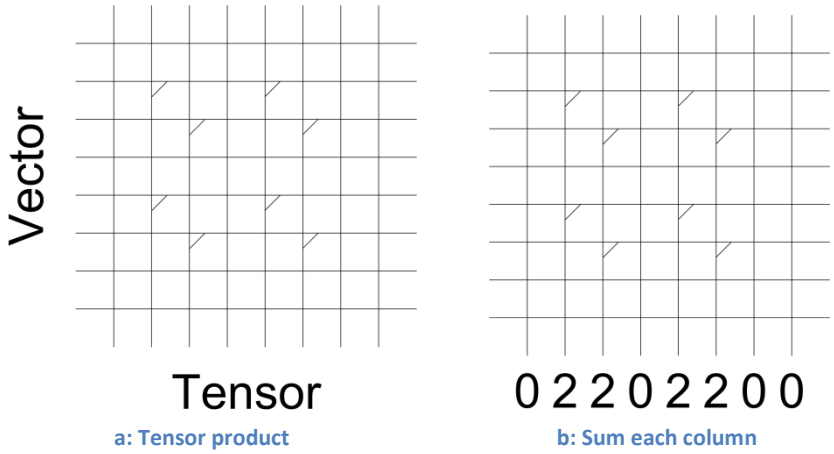
Performing a recall with a tensor product that is too heavily populated is likely to result in a noisy output. If the goal is to continue a tree search until no new rules fire, then it is plausible that this goal could be missed due to spurious data within the output. On the other hand, if the goal is defined as a state containing particular vectors, then it may be incorrectly recognised due to the overlap of other vectors causing an incorrect response.

This work investigates the capacity of a tensor product, with regard to the length and weight of the tensor used in creating the tensor product. In addition to this, it also explores the use of different methods for tensor generation in order to attempt to achieve the most efficient memory usage.

7.2 Design

This work is largely concerned with recalling superimposed tensor products from a CMM. For the purposes of this development iteration, however, performing a recall is unnecessary in order to explore the capacity of a tensor product. Instead, the reliability of recalling a vector from the superposition of multiple tensor products may be tested directly.

Previous development has already implemented the functionality of recalling a vector from a tensor product. As such, the only additional functionality that is required is the ability to recognise individual tensor products within a superimposed tensor product. Without this, a classical list of tensors used would need to be maintained, which would defeat the objective of a CMM rule-chaining system. The literature does not describe an algorithm to perform this operation, and so one solution is presented below.



a: Tensor product b: Sum each column

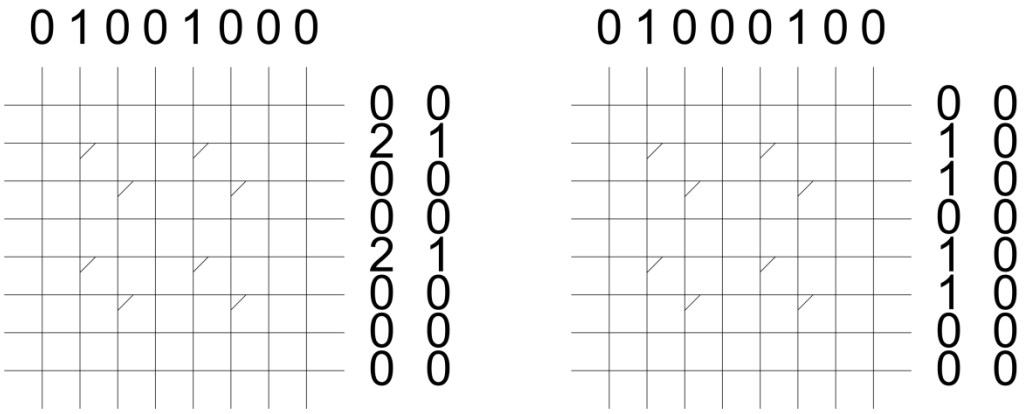
Figure 31: Identifying tensors within a tensor product

The first step is to identify which tensors may exist within the tensor product. In order to do so, the tensor product is considered as a matrix (Figure 31a) and each column is summed (Figure 31b). The application of a threshold, using a value equal to the weight of a single vector, is then able to retrieve the superposition of all tensors within the tensor product. For the example given, this threshold value is 2 and the superposition of tensors after applying this threshold is "01101100".

Baum's algorithm (lengths {3, 5})	Random generation
01001000	01100000
01000100	01001000
00101000	01000100
00100100	00101000
	00100100
	00001100

Table 14: Possible tensors within a tensor product

As shown in Table 14, all possible tensors within a tensor product may then be enumerated. If the tensors were generated using Baum's algorithm, then the additional information about the structure of a tensor may be used to reduce the possible tensors, by removing those that are invalid.



a: Correct output vector b: Empty output vector

Figure 32: Identifying vectors within a tensor product

It appears that the problem now is to determine exactly which of the possible tensors is actually present within the tensor product. Knowing which tensors are present is actually unnecessary, as a simple recall of each tensor in turn from the tensor product will result in the correct output vectors. Figure 32 shows the possible cases, with the threshold value set to the weight of a tensor; in Figure 32a the use of a correct tensor results in the correct output vector, and in Figure 32b the use of an incorrect tensor results in an empty output vector.

If the capacity of a particular tensor product is exceeded, errors may result from two sources. As with a recall performed on a CMM, an output vector may contain extra set bits due to interference from other superimposed tensor products. In addition to this, however, an incorrect tensor may unexpectedly cause the recall of an output vector.

This work consisted of four experiments, initially to determine the effects of changing the generation method, length, and weight of tensors, and finally to test the previously proposed algorithm to recognise individual tensor products. Unless otherwise specified, for the purposes of these

experiments, a recall error is defined as a recalled vector that differs in any way from the original vector. Also, each experiment was run 20 times, with the average results being used for evaluation.

7.2.1 Tensor generation

Due to the similarities between a tensor product and a CMM, it is to be expected that the use of tensors generated using Baum's algorithm [8] will result in a larger capacity than the use of tensors generated randomly. In addition to random generation and Baum's algorithm, this iteration introduces a further method of vector generation – lexicographic generation [27].

This method of generation enumerates all possible vectors for a given length and weight in lexicographic order. As an example, consider the comparison of lexicographic and Baum generation of vectors with a length of 5 and weight of 2 given in Table 15.

Lexicographic generation	Baum's algorithm (lengths of {2, 3})
11000	10100
10100	01010
10010	10001
10001	01100
01100	10010
01010	01001
01001	
00110	
00101	
00011	

Table 15: Generation of lexicographic and Baum vectors

It is clear in the given example that Baum's algorithm is able to generate fewer vectors than lexicographic generation. This is due to the enforced minimum Hamming distance provided by Baum's algorithm. Random generation is actually capable of generating the same vectors as lexicographic generation, the sole difference being that with lexicographic generation the order in which they are generated is deterministic. As such, it would be expected that vectors generated lexicographically and randomly should display similar properties.

As in previous experiments, the vector length is set at 500, with a weight of 5. In order to reduce the computation and memory requirements of recalling a tensor product from a CMM, it is ideal if a tensor can be significantly smaller than a vector. To this end, the tensor length used is 20, with a weight of 2. As Baum's algorithm and lexicographic generation are deterministic, different seed values are used for each execution of the experiment. The operation of this experiment is as follows:

1. Generate a new input vector, using Baum's algorithm
2. Generate three new tensors, using Baum's algorithm, randomly, and lexicographically
3. Create three tensor products, using the single input vector with each of the tensors
4. Superimpose these tensor products with any previously generated tensor products for the respective method of tensor generation
5. Recall every previously superimposed tensor in turn, for each of the methods of tensor generation, comparing the recalled vector to that which was originally superimposed
6. Record the rates of recall error, and continue from 1 if any of the results has a rate of recall error less than 100%

7.2.2 Tensor weight

Although it might be expected that the capacity of a tensor product is similar to that of a CMM, all previous experiments have used the same parameters for generating input and output vectors. In this iteration, a tensor is equivalent to an output vector and tensors and input vectors are generated with different lengths and weights.

The capacity of a tensor product is related to the length and weight of both the input vector and the tensor, as discussed in Section 2.5.4. This experiment is designed to investigate the effect of increasing the weight of tensors, while maintaining constant parameters for the input vector; a length of 500, weight of 5, and generation using Baum's algorithm. Lexicographic generation, rather than random, is used for creating the tensors as the determinism of this algorithm should aid comparability between different tensor weights.

The operation of this experiment is as follows:

1. Generate a new input vector, using Baum's algorithm, and a new tensor lexicographically
2. Create the tensor product of this input vector and tensor, and superimpose this with any previously generated tensor products for this tensor weight
3. Recall every previously superimposed tensor in turn, comparing the output vector to that which was originally superimposed
4. Continue from 1 if none of the recalled vectors were erroneous, otherwise increase the tensor weight, clear the tensor product, and continue from 1

7.2.3 Tensor length

This experiment is designed for the same purpose as the previous experiment, but increasing the tensor length given a fixed weight. As such, lexicographic generation is used to create the tensors, while vectors are generated using Baum's algorithm, with a length of 500 and weight of 5.

The operation of this experiment is as follows:

1. Generate a new input vector, using Baum's algorithm, and a new tensor lexicographically
2. Create the tensor product of this input vector and tensor, and superimpose this with any previously generated tensor products for this tensor length
3. Recall every previously superimposed tensor in turn, comparing the output vector to that which was originally superimposed
4. Continue from 1 if none of the recalled vectors were erroneous, otherwise increase the tensor length, clear the tensor product, and continue from 1

7.2.4 Recognising individual tensor products

Previous experiments within this iteration have used an external list of tensors in order to recall vectors from the superimposed tensor product. This experiment differs by using the algorithm proposed above in order to calculate the tensors, and then comparing vectors recalled with these tensors to those that were originally superimposed. As such, this serves to verify the ability of this algorithm to correctly determine individual vectors within a superimposed tensor product.

Due to the difference in the design of this experiment, the definition of a recall error has been slightly modified. A recalled vector that differs in any way from the original vector is still classified as a recall error, however any additionally recalled vectors are also classified as a recall error.

The operation of this experiment is as follows:

1. Generate a new input vector, using Baum's algorithm
2. Generate three new tensors, using Baum's algorithm, randomly, and lexicographically
3. Create three tensor products, using the single input vector with each of the tensors
4. Superimpose these tensor products with any previously generated tensor products for the respective method of tensor generation
5. Use the proposed algorithm to determine possible tensors within the tensor product
6. Recall every possible tensor in turn, for each of the methods of tensor generation; if the recall of a tensor results in a vector, then this is compared to the input vector that was originally superimposed
7. Record the rates of recall error, and continue from 1 if any of the results has a rate of recall error less than 100% and while the number of superimposed tensors is fewer than the maximum unique tensors that may be generated using Baum's algorithm

7.3 Implementation

Aside from the experimental control system, very little implementation was required for this iteration. The only additional functionality needed is the algorithm used in the fourth experiment in order to determine possible tensors within a tensor product. As well as this, the comparison between tensors calculated using this algorithm and those that were actually superimposed is implemented as a function as this may be useful in the next iteration.

The control structure for each of the experiments is simple, remaining similar to previous iterations but superimposing tensor products instead of training vector pairs into a CMM.

7.4 Testing

Unit testing was undertaken to ensure the correct implementation of the new functions. The complete set of test cases for this development iteration are given in Appendix 2. Experiments were also tested, to ensure their correct operation, using a combination of the debugging facilities available within Visual Studio and console output.

7.5 Results

7.5.1 Tensor generation

Figure 33 shows the probability of recall error from a tensor product, given the number of tensor products that are superimposed, for each of the three methods of vector generation.

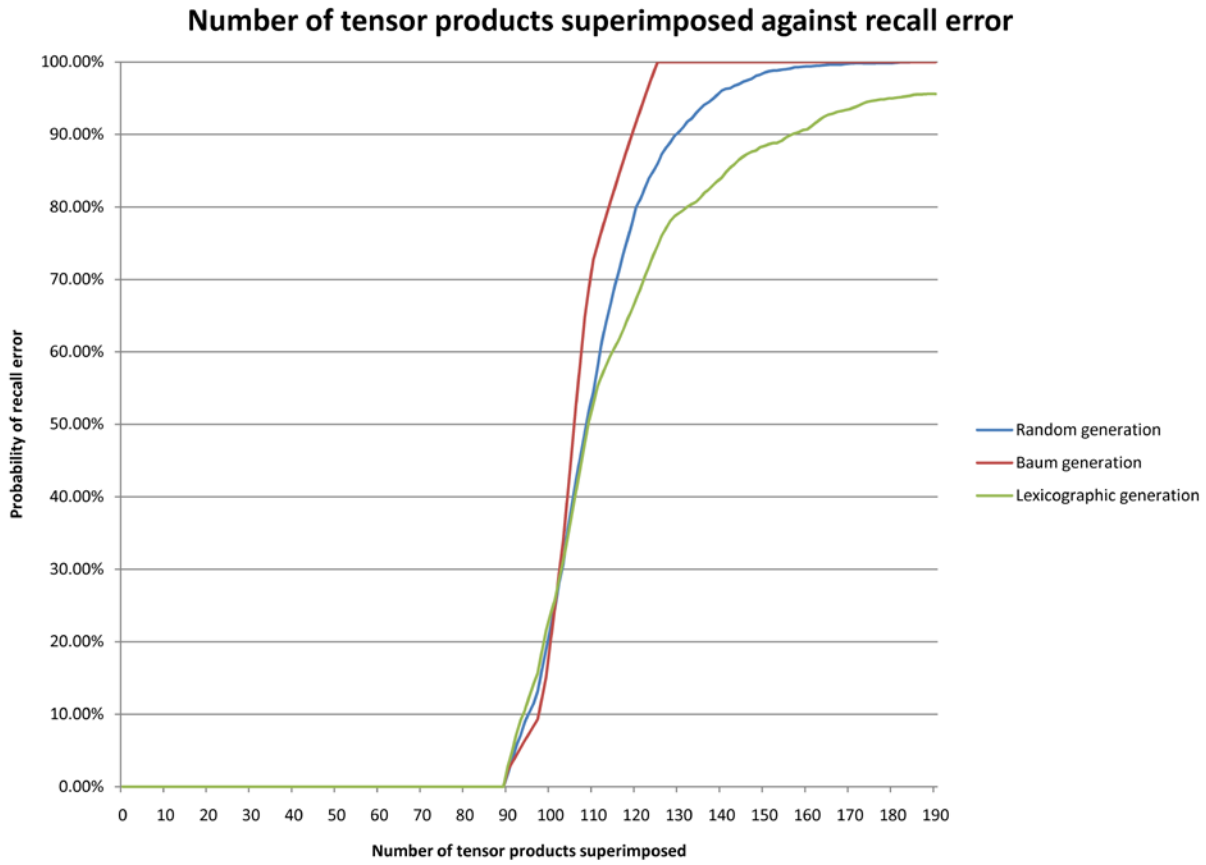


Figure 33: Probability of recall error from a tensor product

7.5.2 Tensor weight

The experiment was designed only to investigate the capacity at which a tensor product failed to achieve perfect recall reliability. As such, the table below shows the number of tensor products that may be superimposed for a given tensor weight before any recall fails.

Tensor weight	Tensor products superimposed
2	89
3	89
4	89
5	89
6	89
7	89
8	89
9	89
10	89
11	89
12	89
13	89
14	89
15	89
16	89
17	89
18	89

Table 16: Number of tensor products successfully recalled for a given tensor weight

7.5.3 Tensor length

Similarly to the previous experiment, this was designed to explore the capacity at which a tensor product failed to achieve perfect recall reliability. Figure 34 shows these results graphically, with the full results table given in Appendix 3.

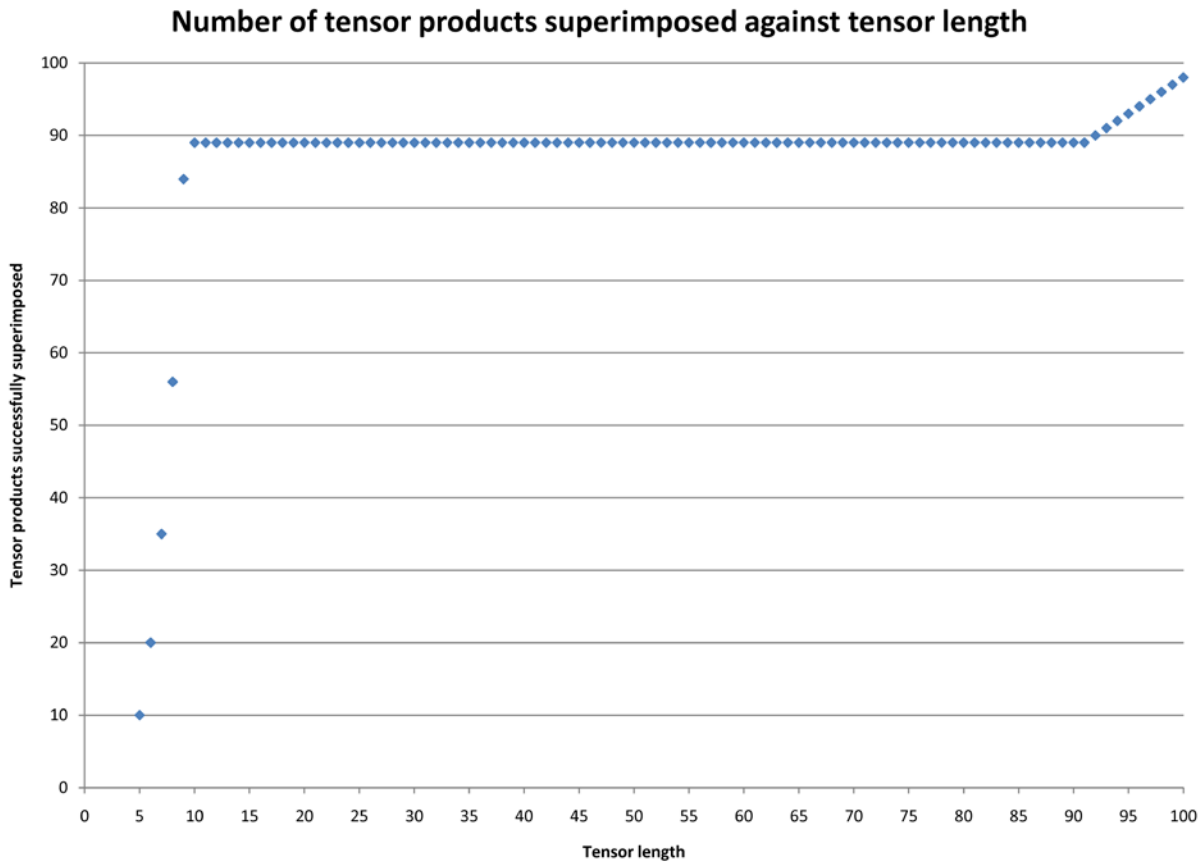


Figure 34: Number of tensor products successfully recalled for a given tensor length

7.5.4 Recognising individual tensor products

The final experiment of this development iteration is similar to the first experiment, but this time using an algorithm to determine which tensors are present within a tensor product rather than a simple list. The results are given in Figure 35, in the same format as in the first experiment in order to aid comparison.

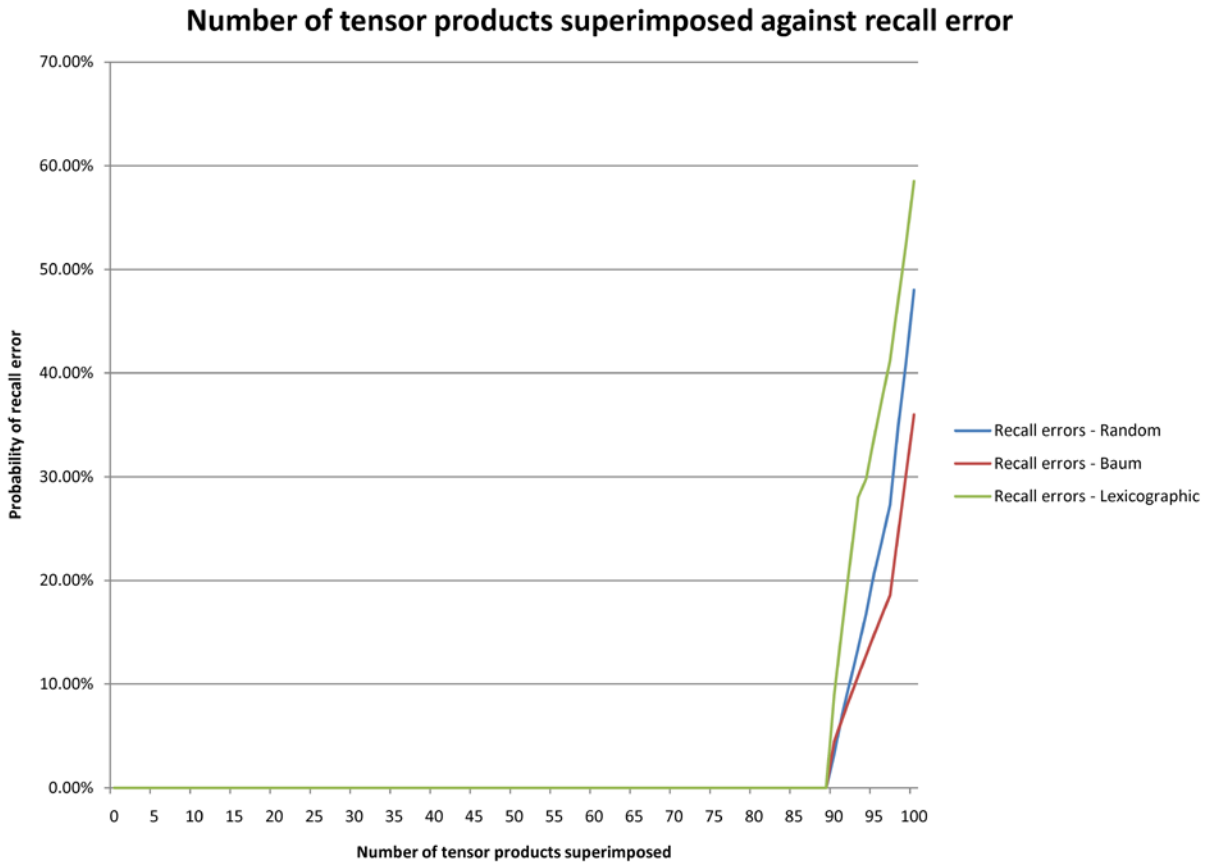


Figure 35: Probability of recall error from a tensor product using tensor detection algorithm

7.6 Evaluation

It was expected that with regard to the storage capacity, a tensor product would display similar properties to a CMM. From the results of the first three experiments performed in this iteration, this would appear not to be the case.

The first experiment was expected to show that a tensor product storing tensors generated using Baum's algorithm would have a higher capacity than one storing tensors generated randomly or lexicographically. In actual fact, the results in Figure 33 show that using the parameters selected (a tensor length of 20 and weight of 2, in combination with a vector length of 500 and weight of 5) all three algorithms result in virtually equal storage capacities.

Likewise, the results of the second experiment were expected to show similar results to those given in Section 4.5.2 – as the tensor weight increased, the storage capacity decreased. The results in Table 16 appear to show that changing the weight of a tensor does not in fact affect the storage capacity of a tensor product.

At first sight the results of the third experiment in Figure 34 appear to display the same unexpected features. Using Equation 3 it is possible to generate the maximum unique tensors that can be generated for a given length and weight, a selection of which are shown in Table 17. From this it is clear that the limiting factor for tensors with a length shorter than 14 is the number of unique vectors that exist for a given length. However, this does not serve to explain the constant tensor product capacity found with a tensor length between 14 and 90.

Tensor length (tensor weight of 2)	Maximum unique tensors
12	66
13	78
14	91
15	105
16	120

Table 17: Unique tensors that may be generated for a given length and weight

The results obtained for a tensor length greater than 90 offer the only insight available as to why these experiments did not behave as originally expected. For these tensor lengths, the number of tensor products that may be superimposed increases in a linear manner. One would have expected this to have been the case for all tensor lengths, given the lexicographic method of tensor generation.

The only plausible explanation for these results thus relates to Equation 5. The probability of recall failure from a CMM or tensor product, as given by this equation, is dependent on the length and weight of both the input and output vectors. The disparity between the vector and tensor lengths, however, is evidently an important factor in determining the capacity of a CMM or tensor product.

The results of the final experiment (Figure 35) show that the algorithm to detect individual tensors within a tensor product does work as expected. The same number of tensor products may be successfully superimposed and perfectly recalled before any error occurs. At this point, the probability of error (Figure 33) increases more sharply than when the recall is performed from a basic list. This is caused by the updated definition of a recall error, to include tensors that were not originally used in the superposition but provide a vector output when recalled from the tensor product.

8 Use of a two-layer network

8.1 Overview

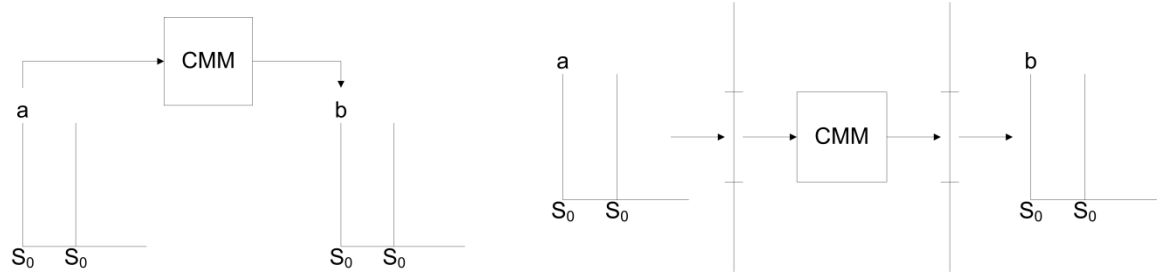
Previous iterations of this work have shown that a CMM is an effective tool for storing and recalling rules. It has then been demonstrated that rule-chaining is possible, by simply passing the output of a CMM directly back as an input. Finally, the use of tensor products, especially the superposition of multiple tensor products, has been investigated as a method to increase the throughput of a CMM by searching multiple rules simultaneously.

This final iteration is intended to resolve one of the main problems when performing rule-chaining with a CMM, that of branching rules. For example, consider a very simple rule set that contains only two rules:

1. $a \rightarrow b$
2. $a \rightarrow c$

Using the system as it currently stands, if an a was presented as an input, the output would be the superposition of b and c. It would not be possible to identify particular vectors within this superposition, and so branching rules may not be used. There are two possible solutions to this issue, as described below.

8.1.1 Training the correlation matrix memory with tensor products



a: Iterate through each column of tensor product b: Recall entire tensor product

Figure 36: Two methods to recall a tensor product

Figure 36 shows two methods for performing a recall operation using a tensor product. The first method (Figure 36a), proposed in [1], is to iterate through the tensor product and present each column as an input to the CMM. This gradually builds the output tensor product, and results in a CMM with a size equal to

$$v^2$$

where v is the vector length.

The second method (Figure 36b) considers the input tensor product to be a single vector, by stacking each column of the tensor product. It is then possible to recall this vector in a single operation, resulting in another stacked tensor product. If this recall method is utilised, then it is required to train the CMM with stacked tensor products; the CMM therefore has a size equal to

$$(v \times t)^2$$

where v is the vector length and t is the tensor length.

This method of recall requires fewer operations than the first method, however it does have the potential to generate extremely large matrices as the vector and tensor lengths increase. It would therefore be infeasible to store a large rule set, without the use of a sparse matrix representation such as that provided by AURA [14].

With regard to branching rules, this method of tensor product recall should be used as follows to resolve the issue:

1. Each unique data token within the rule set is assigned a unique tensor
2. When training a rule into the CMM, the input and output vectors are tensored with their respective tensor and the entire tensor product is trained
3. When recalling a rule from the CMM, the input vector is tensored with its respective tensor and the correct output tensor products will be recalled

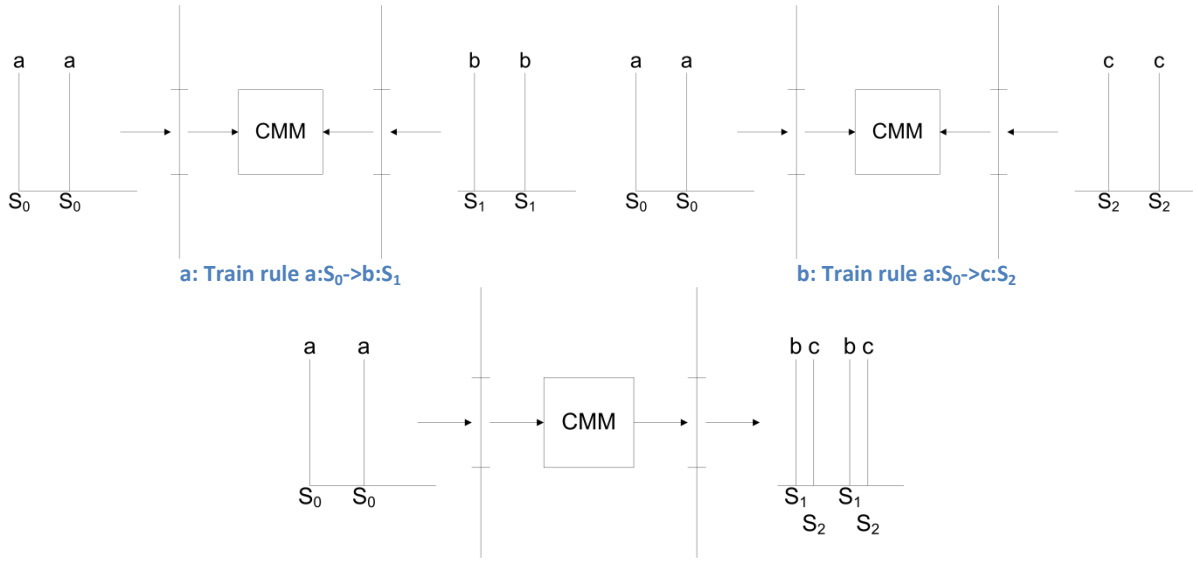


Figure 37: Tensor product training and recall

This is shown in Figure 37, using the very simple rule set given earlier. The two rules are trained as tensor products (Figure 37a and Figure 37b) and when the input $a{:}S_0$ is presented, the superposition of both rules is recalled. As was demonstrated in the previous iteration, it is possible to identify particular vectors within a tensor product.

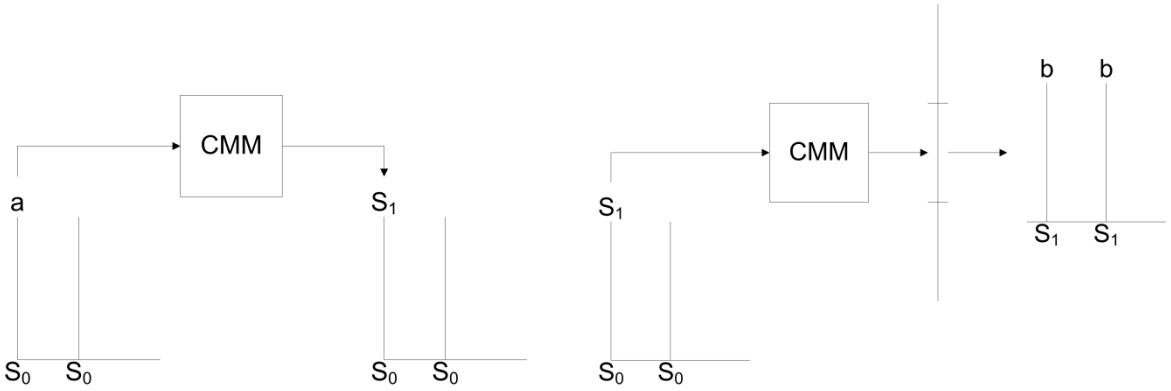
8.1.2 Using two correlation matrix memories

The second method, as proposed in [1] and described in detail in Section 2.6.4, is to use two separate CMMs; a two-layer CMM network.

Each rule within the rule set is assigned a unique tensor, and the first CMM is trained using the input vector and this tensor as an output. Using the tensor as an input, the second CMM is then trained with the stacked tensor product of the tensor and the output vector as an output. The total size of these CMMs is therefore

$$(v \times t) + (v \times t^2)$$

where v is the vector length and t is the tensor length.



a: Recall from the first CMM b: Recall from the second CMM

Figure 38: Recall using a two-layer CMM network

Figure 38 shows a recall operation. A tensor product is created using the input vector and a unique tensor. Each column of this is recalled from the first CMM in turn, resulting in an intermediate tensor product that can be recalled from the second CMM in the same manner. Each vector recalled from the second CMM, however, results in an entire tensor product output. These must be superimposed using one of two methods:

- Applying a threshold before performing the logical OR
- Applying a threshold, before summing and applying a second threshold, as suggested in [1]

This work firstly compares the memory requirements of a one-layer and two-layer implementation, both with and without the use of sparse matrices. It then investigates the differences between the two methods of superimposing tensor products output from the two-layer CMM network. Finally, as a demonstration of the technology, it develops a simple user dialogue that can be used to control a two-layer implementation.

8.2 Design

The number of set bits within a CMM determines the memory required to store it as a sparse matrix. The simple matrix used within this work is not stored sparsely, and so a modification is required to the CMM class in order to count the number of bits set.

As an alternative to iterating through the matrix and counting the number of bits set, a modification can be made to the method used to store correlations in the matrix. Previously, this method would set a particular bit to logical 1, without first testing its current state. This optimisation reduced the average number of operations required to store a correlation. In order to maintain a count of the number of bits set to logical 1 within the matrix, this optimisation is removed. The matrix location needs to be tested; if it is currently logical 0 then it will be set to logical 1 and the count of set bits incremented, otherwise no action is required.

Using a two-layer network introduces the notion of a state, a hidden layer between the first and second CMM. The rule structure is thus amended to store this state in addition to the input and output vectors.

This work consisted of two experiments, the first to determine the memory requirements of the one-layer and two-layer implementations given a fixed vector and tensor length and weight. The second experiment then explores the reliability of a recall operation from a two-layer network,

comparing the two methods of tensor product superposition. This investigates the effects of both increasing the number of correlations stored in the matrix as well as the number of tensor products that are superimposed for the input.

8.2.1 Memory requirements

This experiment has been designed to compare the memory requirements of a one-layer and two-layer solution to the problem of branching rules. This comparison effectively counts the number of matrix locations that are set within the system. The vector length of 500 and weight of 5 have been chosen to maintain consistency with previous experiments. The tensor length of 50 and weight of 3 were selected to allow for a large number of tensors to be generated.

Both the vector and tensor generation are performed using Baum's algorithm, and as such this experiment is deterministic. This experiment was run 20 times, with different seed values, operated as follows:

1. Create three empty CMMs, with appropriate dimensions as described in Sections 8.1.1 and 8.1.2
2. Generate a new input and output vector pair and a new input and output tensor pair, using Baum's algorithm
3. Generate two tensor products – one using the input vector and tensor, and the second with the output vector and tensor
4. Train the CMMs with the appropriate inputs and outputs
    a. The first CMM with the input tensor product and output tensor product
    b. The second CMM with the input vector and output tensor
    c. The third CMM with the output tensor and output tensor product
5. Record the size of each system, in terms of the number of bits set multiplied by the size of an integer
6. Continue from 2 if the training limit has not been reached

8.2.2 Methods of output tensor product superposition

Using a two-layer network, the final output is a set of tensor products that must be combined in some way to form only a single tensor product. Having applied a threshold to each of the individual tensor products, two methods are proposed:

1. Sum them, taking a logical 1 to be the value 1, and threshold the result
2. Superimpose them using a logical OR

The second method requires slightly fewer operations, which would be beneficial to the operational speed of the system. On the other hand, it is asserted in [1] that the first method will result in a more accurate recall, helping the system to recover from spurious tokens that appear in the output recalled from the first CMM.

Previous iterations have shown that the number of correlations within a CMM, as well as the number of tensor products superimposed before input, has an effect on the recall reliability. For this reason, this experiment is designed to investigate the recall reliability provided by a two-layer network while adjusting both of these parameters.

Applying a threshold using Willshaw's method ensures that an output contains all of the bits that are expected to be set [6]. In addition to this, an output may contain extra, spurious data. This experiment uses the number of additional set bits as the distance metric when comparing a recalled output to the perfect, expected output. In addition to this, the number of recall errors are also counted, using the definition that includes both output vectors that differ from the original vector and any additionally recalled vectors.

The operation of this experiment is as follows:

1. Create two empty CMMs, with the dimensions as described in Section 8.1.2
2. Generate a new input and output vector pair, and a new tensor, using Baum's algorithm
3. Generate a tensor product using the tensor and the output vector
4. Train the CMMs with the appropriate inputs and outputs
    a. The first CMM with the input vector and tensor
    b. The second CMM with the tensor and tensor product
5. Continue from 2 until 50 rules have been trained
6. Recall each of the previously trained input vectors in turn
    a. Generate a new tensor, and create the tensor product of an input vector with this tensor
    b. Superimpose this tensor product on the input that was previously used, or with nothing if this is the first input vector
7. Compare the output tensor product to that which was expected, and record the number of additional set bits and erroneous vectors that occur when using both of the proposed methods for combining output tensor products
8. Continue from 2, until the recall of a single input tensor product causes erroneous output

8.2.3 Simple demonstration system

This demonstration system is an extension of the rule-chaining system developed in Chapter 6. It brings together the use of Baum's algorithm to increase CMM capacity, tensor products for superimposed recall, and a two-layer network to allow for branching rules. Additionally, it has been designed to allow online training of additional rules – one of the benefits that CMMs provide over standard neural networks.

8.3 Implementation

The CMM class has been updated, to contain an extra field that stores the number of locations within the matrix that contain a logical 1. Additionally, the methods that are used to change a location within the matrix between logical 0 and logical 1 have been modified to maintain the correct count within this field.

In order to store the state vector within a rule, another field has been added to the rule structure. This does not affect the operation of any previously developed functionality, as it will simply remain unreferenced if not required.

The control structure used in these experiments is very similar to that developed in previous iterations, using loops to train and recall vectors and tensor products in the CMM. The control structure for the demonstration system is included on the attached CD, in 'demoSystem.cpp'.

8.4 Testing

As in previous iterations, unit testing was performed to ensure that any new functions, as well as any that have been modified, were correctly implemented. The complete test cases for this development iteration are given in Appendix 2, with previous test cases relating to the CMM being re-run.

Having completed unit testing of all functions developed in this and previous iterations, the implementation of experiments was tested by simply ensuring that the control loops executed as expected, with each variable declared in the correct scope.

8.5 Results

8.5.1 Memory requirements

The memory required for a matrix that is stored in its entirety is independent of the number of correlations stored within that matrix. As such, there are only one set of results for this:

	One-layer solution	Two-layer solution
Memory requirement	74.51 Megabytes	155.64 Kilobytes

Table 18: Memory requirements without a sparse matrix representation

When a sparse representation is used, however, the memory usage is affected by the number of correlations stored. Figure 39 shows the memory required by a one-layer and two-layer implementation, given the number of vector pairs that have been trained.

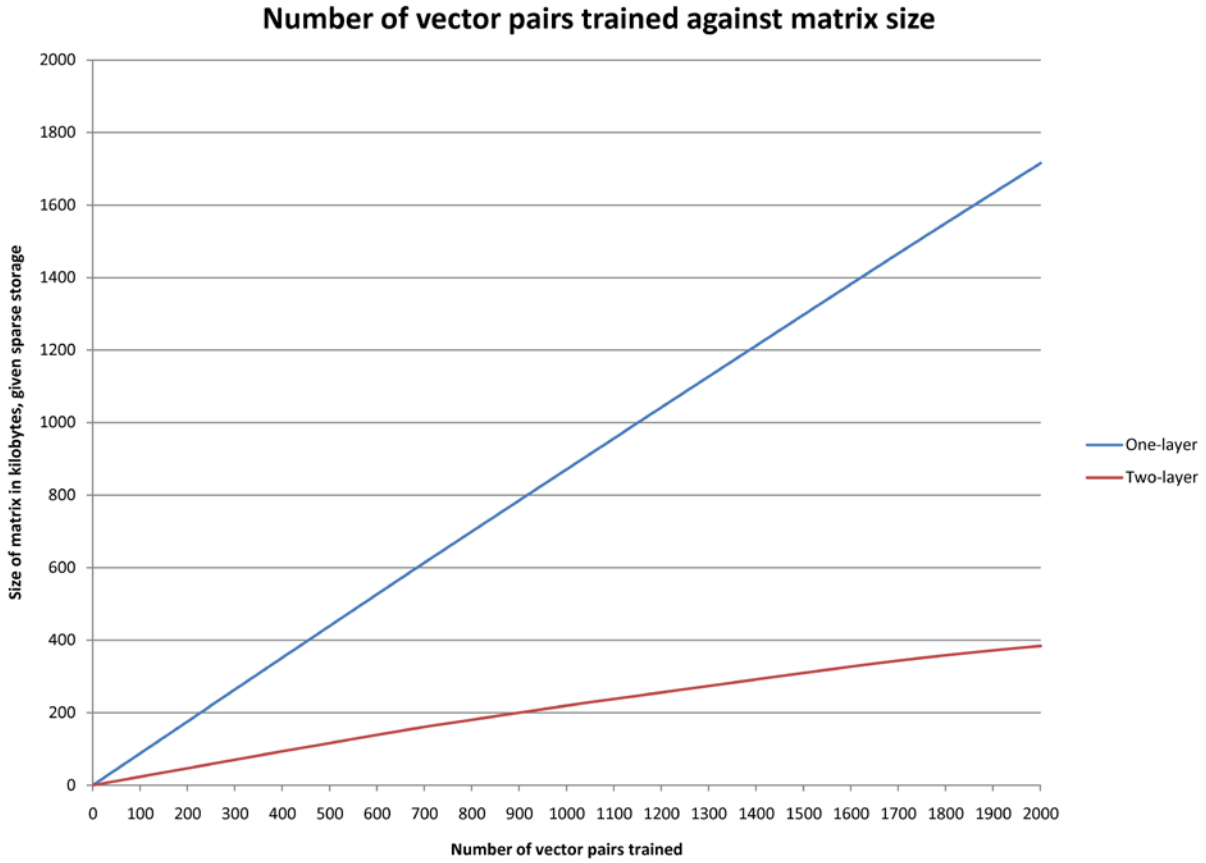


Figure 39: Memory requirements of a CMM using a sparse matrix

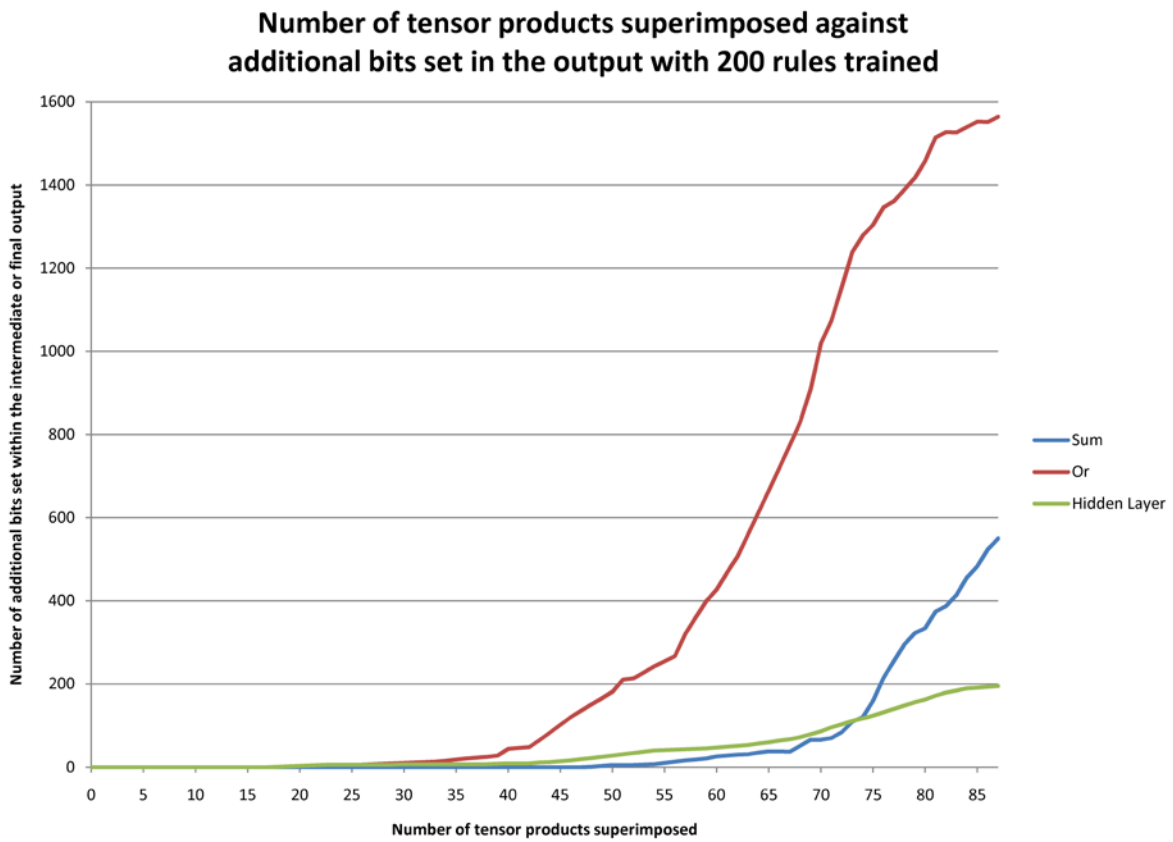


Figure 40: Number of additional bits set in the output of a system trained with 200 rules

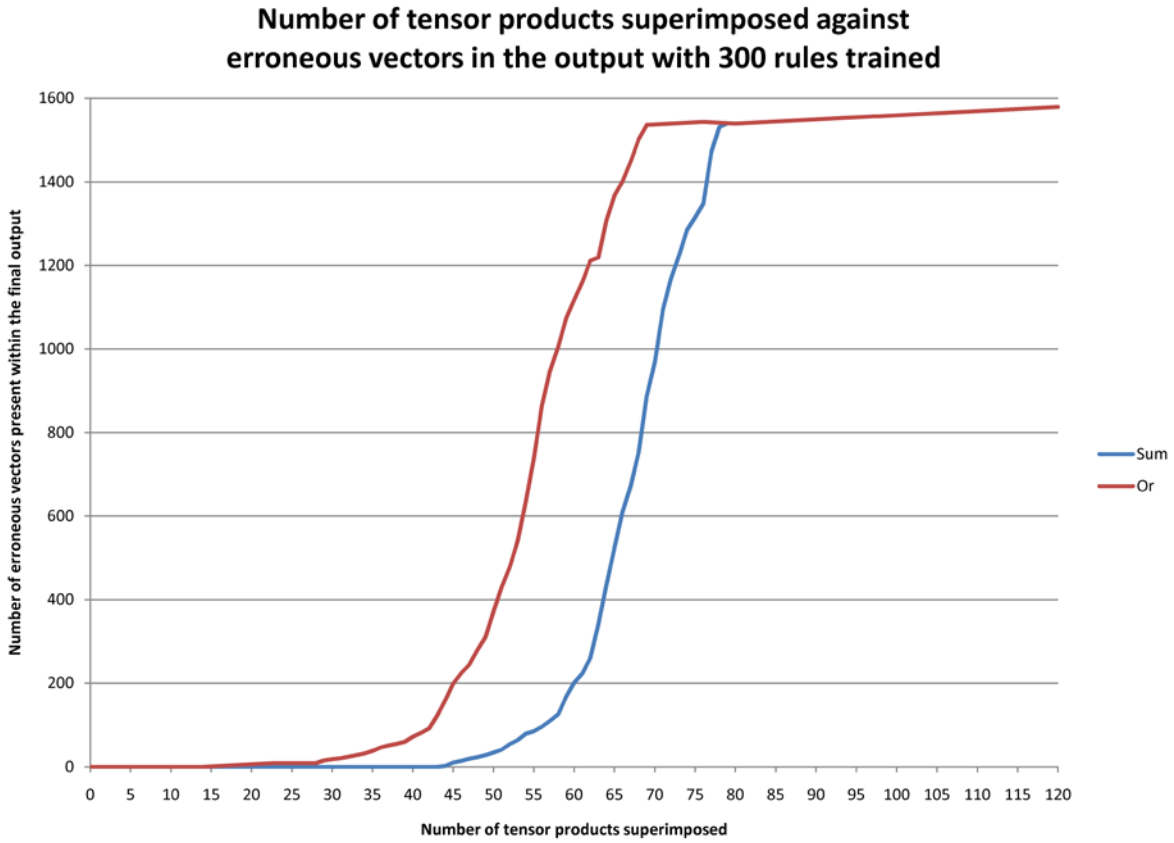


Figure 41: Number of erroneous vectors in the output of a system trained with 300 rules

8.5.2 Methods of output tensor product superposition

The results obtained at different levels of CMM saturation show largely similar properties. As such, only a small selection of graphs are given above, with further results in Appendix 3. Figure 40 shows the number of additional bits set within the final output of a system that has been trained with 200 rules, using both the logical OR and the sum-then-threshold methods. In addition to this, it shows the number of additional bits set in the output of the first CMM – the hidden layer. Figure 41 shows the number of erroneous vectors present within the final output of a system that has been trained with 300 rules.

Table 19 shows the number of input tensor products that may be superimposed for a given number of rules trained, while still achieving a recalled output that exactly matches the expected output.

Number of rules trained	Number of input tensor products superimposed with perfect recall	
	Logical OR	Sum-then-threshold
50	(maximum) 50	(maximum) 50
100	42	(maximum) 100
150	34	51
200	17	47
250	14	33
300	14	29
350	0	0

Table 19: Number of tensor products superimposed against number of rules trained

8.6 Evaluation

It is clear from the results that when a sparse matrix is not used, a one-layer solution will require a larger amount of memory than a two-layer solution. Using the vector and tensor lengths that were selected, the result is in fact two orders of magnitude larger. This is as expected, in fact given that the matrix sizes are known this result can be calculated without the need for any implementation.

The results obtained using a sparse matrix representation are therefore more interesting. It is possible to calculate a very conservative upper bound for the number of correlations stored in a matrix simply with:

$$I \times N \times T$$

where I is the input vector weight, N is the output vector weight, and T is the number of rules trained. Using this basic equation, the upper bound of a one-layer solution with a vector weight of 5 and tensor weight of 3, storing 1000 rules is

$$(5 \times 3) \times (5 \times 3) \times 1000 = 225000\ correlations$$

Similarly, a two-layer solution has an upper bound of

$$5 \times 3 \times 1000 + 3 \times (5 \times 3) \times 1000 = 60000\ correlations$$

These upper limits assume that there is zero overlap between vector pairs stored in the CMM. Although the use of Baum's algorithm does reduce vector overlap by increasing the Hamming distance, it does not entirely remove it. As such, the experimental results are slightly below these theoretical limits, with the one-layer solution storing 223050 correlations and the two-layer solution storing 56100 correlations.

Figure 39 shows the memory requirements for both a one-layer and two-layer solution graphically. With the number of rules that were trained, both systems display a reasonably linear result. This would be caused by the low level of saturation within each CMM reducing the number of collisions between correlations. Exactly as expected, the one-layer solution imposes a far greater memory requirement than the two-layer solution.

The results of the comparison between methods of tensor product superposition show that the method proposed in [1] is significantly better at reducing spurious data in a recalled output than the more naive approach of performing a logical OR.

Figure 40 shows the number of additional bits set in the recalled output of a CMM network trained with 200 rules. It shows that up to a particular limit, in this case 17 superimposed input tensor products, a perfect recall is obtained. After this point, additional bits begin to be set in the state layer – the output from the first CMM. Whereas creating the final output using the logical OR results in additional bits also being set, the sum-then-threshold approach is able to give a perfect output with up to 47 superimposed input tensor products.

Figure 41 shows similar results, this time in terms of the number of erroneous vectors present within the final output of a network trained with 300 rules. The graphs have a very similar shape, but using the sum-then-threshold method is able to give a perfect output with a far greater number of superimposed input tensor products than the simple logical OR. The number of erroneous vectors is defined as the number of vectors that are not a perfect match for the original vector that was trained plus the number of additional vectors that appear within a tensor product.

The results given in Table 19 show that both the number of input tensor products superimposed and the number of rules trained affect the recall reliability of this system. As the number of correlations stored within the matrix increases, spurious bits set within the output from the first matrix are more likely to occur. As the number of superimposed input tensor products increases, the ability for the sum-then-threshold method to recover from these spurious bits decreases.

The demonstration system is designed only to show that the rule-chaining operation can be successfully performed using a two-layer network. The test cases given in Appendix 2 and Table 20 are therefore sufficient to show that a rule set containing branching rules, i.e. multiple rules with the same antecedent, can be searched successfully and in parallel using the techniques developed within this work.

Rules			
a->s0->b	b->s2->d	c->s4->f	d->s6->h
a->s1->c	b->s3->e	c->s5->g	
Input			
a, {enter}, recall, {enter}			
Expected output			
Loop 1: First matrix: State: State vector not recognised // Superposition of s0, s1 Second matrix: Output: b Output: c Loop 2: First matrix: State: State vector not recognised // Superposition of s2, s3 State: State vector not recognised // Superposition of s4, s5 Second matrix: Output: d Output: e Output: f Output: g Loop 3: First matrix: State: s6 Second matrix: Output: h Loop 4: First matrix: No states recognised Second matrix: No output vectors recognised			
Pass, comments (//) added for the purposes of clarity			

Table 20: Test case for the demonstration production system

9 Conclusions and future work

This work involved the development of a production system using CMMs, as proposed in [1]. During this process, various components of the system have been explored in order to both test the system and provide results with which to compare to a classical rule-chaining system.

9.1 Experimental conclusions

9.1.1 Vector recall

The experimentation performed in the first iteration showed that training a CMM with vectors generated using Baum's algorithm [8] resulted in a significantly higher capacity than that achieved by using randomly generated vectors. Baum's algorithm provides a guaranteed minimum Hamming distance between vectors, and so this result was as expected due to reduced interference in the matrix. The results also showed that when a CMM is presented with a perfect input, applying a threshold using either Willshaw's method or L-max give the same result.

The experimentation used a CMM that stored input and output vectors generated with the same length and weight parameters. The results obtained show that the vector weight does affect the storage capacity in the way that was expected. As the vector weight increases, so too does the capacity of the matrix. This is due to the increased number of neurons that would need to fire as a result of interference in order to surpass the threshold value. After a certain limit, the capacity begins to drop as the vector weight continues to increase. This is because the increased number of correlations, and the resulting interference, outweighs the ability of a threshold to remove any incorrectly firing neurons.

9.1.2 Tensor product recall

The results obtained in this iteration showed that performing a recall operation using a tensor product has no effect on the probability of recall error. The method of tensor product recall proposed consisted of multiple basic vector recall operations, and so this result was to be expected. The requirement for multiple recall operations to be performed also resulted in significantly slower recall. As discussed, however, this would not pose a problem as the second matrix of the final system requires recall in this manner and so simple pipelining could be used.

9.1.3 Rule-chaining with tensor product recall

The third iteration did not involve any formal experimentation, and as a result there was little to conclude. The testing performed did, however, serve to show that rule-chaining operated correctly, and that the approach used is effective.

9.1.4 Superposition of tensor products

This iteration contained experimentation that investigated the capacity of a tensor product. A tensor product is formed by performing an outer product operation between two vectors. As such, it is similar to training a CMM with those two vectors, and it was expected to display similar properties as those found in the first iteration.

The experimentation varied the parameters used in generating only the second vector, known as the tensor. As such, there was a large disparity between the length used for the vector and the tensor. The results showed that under these conditions, the length and weight used for the tensor have very

little effect on the capacity of a tensor product. Only when the tensor length approached 20% of the vector length was the capacity affected in any way.

Additionally, this iteration introduced an algorithm to determine individual vectors within a superimposed tensor product. The results obtained regarding tensor product capacity using this algorithm were on a par with those using a classical system for comparison, and thus showed that the algorithm is effective.

9.1.5 Use of a two-layer network

The production system proposed in [1] uses a two-layer network in order to allow the recall of multiple rules with the same antecedent but different consequents to be successful. An alternative method would be to use a one-layer network, but to train it using stacked tensor products. The results obtained in the first experiment of this iteration show that even when using a sparse matrix representation, a one-layer network has a greater memory requirement than a two-layer network.

This does not investigate the capacity of the two solutions; such investigation may find that a one-layer network trained in this way results in a higher capacity than the two-layer alternative. This result would not be relevant, however, as the capacity of either solution may be increased by simply increasing the vector and tensor lengths. When using a sparse matrix, the length of the vector does not directly affect the memory required; thus a two-layer solution would still be more memory efficient than the one-layer solution.

The second experiment performed in this iteration investigated two methods of superimposing the multiple tensor products that are the output of the second CMM. The first method was suggested in [1] and consists of summing the tensor products (where a logical 1 is taken to be the value 1) and applying a threshold to the result. The more naive alternative is simply to perform the logical OR of the tensor products.

The results obtained from this experiment uphold the claim made in [1], that summing before applying a threshold helps to reduce spurious data in the final output when compared to simply performing the logical OR. The reason for this is due to the nature of the method used to perform tensor product recall.

Each column within the tensor product may contain multiple superimposed vectors. When a column is recalled from the CMM, the combination of superimposed vectors may cause additional bits to be incorrectly set within the output. Each tensor product is generated using a different tensor, and so the probability that multiple columns within the superimposed tensor product will contain the same superimposed vectors is low. As such, the probability that multiple columns will result in the same incorrectly set bits will be reduced.

Using the logical OR, each incorrectly set output bit is contained within the final output. Using the sum-then-threshold approach, on the other hand, removes any spurious data that was not recalled a sufficient number of times to result in a sum higher than the threshold value. As the capacity of either the tensor product or the CMM is attained, the probability of spurious data appearing in the output increases – even when using the sum-then-threshold method.

The final task within this work was to build a simple demonstration system, capable of showing that the rule-chaining operation can be successfully performed using the system proposed in [1]. This

was completed, and the test cases given in Appendix 2 show an effective and parallel search of a rule set containing branching rules can be performed using this simple production system.

9.2 Future work

There are various areas within this project which warrant further consideration; the main three are described below.

9.2.1 Arity networks

The next iteration within this work would be to implement and experiment with arity networks. As described in Section 2.6.4, the complete system proposed in [1] uses arity networks in order to be able to store rules with varying numbers of tokens in the antecedent.

The need for arity networks arises due to the selection of a threshold value. For example, consider the very simple set of rules below:

1. $a \rightarrow c$
2. $a \wedge b \rightarrow d$

In order for the first rule to correctly fire, the threshold value must be equal to the weight of a single vector. If this is the case, however, presentation of an input a will also cause the second rule to fire. As such, a separate network is required for each desired arity, to allow the use of different threshold values [1].

9.2.2 Proof of time and space complexity improvements

Using a classical system to perform rule-chaining requires the search of a list of rules for any that match. When a rule set contains branching rules, each of the branches has to be maintained separately. The system proposed in [1] is able to search an entire set of rules with only a single operation, and in parallel using superposition.

For these reasons, it is believed that an efficient implementation of the production system used within this work may reduce both the time and space required to perform a search, when compared to a classical system. This will require a formal proof, comparing the time and space complexity of both rule-chaining solutions.

9.2.3 Fuzzy computation

Throughout this work, there has been an emphasis on the ability of a CMM to provide a perfect recall. As described in the literature review, however, neural networks and CMMs are particularly well suited to fuzzy computation.

The definition of a successful recall operation can thus be amended to include those output vectors that contain up to a certain number of extra bits set. When used with a basic recall operation, this change would be expected to increase the capacity of a CMM. When performing rule-chaining, however, it is unknown as to whether the number of extra bits will increase in line with the number of loops that have been performed or whether the sum-then-threshold method will be able to keep the spurious data at an acceptable level.

9.3 Evaluation of the project

The main aim of this work was to show whether the production system proposed in [1] is a feasible application of CMMs. The simple prototype system that has been developed serves to demonstrate that the production system is viable and operates correctly; in this regard the work has been successful.

The use of thorough unit testing meant that any bugs in the implementation were simple to diagnose and resolve. This is shown in the test cases – if one of the tests failed, then it was clear which function was incorrect. After fixing any bugs that were discovered, any tests relating to that unit were re-run, to ensure that the unit now executed correctly.

The decision to implement a simple CMM system, rather than to use the available AURA library, was made in order that I would develop a better understanding of CMMs and their operation. As such, I believe that this was the correct decision at the time. With the experience and knowledge I have gained, I would now choose to use the AURA library in order to reduce development time and improve efficiency.

The main failing within this work was the decision to write the literature review relatively late within the schedule. Writing the literature review was a challenging part of this project, however while undertaking this I learnt much that has since helped with the understanding of this work. With this deeper understanding, I believe that the development time may have been reduced and the scope of the work could possibly have been extended.

References

[1] Jim Austin, "A production system architecture using a Neural Associative Memory," University of York, York, Unpublished 2003.

[2] Jim Austin, "A Review of RAM based Neural Networks," in *Fourth International Conference on Microelectronics*, Turin, 1994, pp. 58-66.

[3] Jim Austin, "Distributed Associative Memories for High Speed Symbolic Reasoning," *Fuzzy Sets and Systems*, vol. 82, no. 2, pp. 223-233, September 1996.

[4] Jim Austin, "Parallel Distributed Computation," in *International Conference on Artificial Neural Networks*, Brighton, 1992.

[5] Jim Austin, "Uncertain Reasoning with RAM based Neural Networks," *Journal of Intelligent Systems*, vol. 2, no. 4, pp. 121-154, November 1990.

[6] Jim Austin and Richard Filer, "Using Neural Networks for Inferencing in Expert Systems," in *Neural Networks and Their Applications*, John G Taylor, Ed. New York, United States of America: John Wiley & Sons, 1996, ch. 16, pp. 243-.

[7] Jim Austin and Thomas J Stonham, "The ADAM Associative Memory," University of York, York, Yellow Report YCS 94, 1986.

[8] Eric B Baum, J Moody, and F Wilczek, "Internal representations for associative memory," *Biological Cybernetics*, vol. 59, no. 4-5, pp. 217-228, September 1988.

[9] Rafal Bogacz and Christophe Giraud-Carrier, "A Novel Modular Neural Architecture for Rule-Based and Similarity-Based Reasoning," in *Hybrid Neural Systems*, Stefan Wermter and Ron Sun, Eds. Heidelberg, Germany: Springer-Verlag, 2000, pp. 63-77.

[10] Samuel Braunstein. (2009, October) Quantum Information Processing Course website. [Online]. http://www-course.cs.york.ac.uk/qip

[11] Jason Brownlee, "Finite State Machines," *AI Depot - AI Article Writing Contest*, June 2002.

[12] David Casasent and Brian Telfer, "High capacity pattern recognition associative processors," *Neural Networks*, vol. 5, no. 4, pp. 687-698, July-August 1992.

[13] Alison Cawsey. (1994, August) School of Mathematics and Computer Science, Heriot Watt University. [Online]. http://www.macs.hw.ac.uk/~alison/ai3notes/section2_4_4.html

[14] Rob Davis and Aaron Turner. (2007) AURA library documentation.

[15] David Deutsch, "Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer," *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, vol. 400, no. 1818, pp. 97-117, 1985.

[16] Michael Freeman, Michael Weeks, and Jim Austin, "AICP: Aura Intelligent Co-Processor for Binary Neural Networks," in *IP Based SOC Design Forum and Exhibition*, Grenoble, 2004.

[17] Alan Frisch. (2010, January) Logic Programming and Artificial Intelligence Course website. [Online]. http://www-course.cs.york.ac.uk/lpa

[18] Paul Gillard. (2007) Department of Computer Science, Memorial University. [Online]. http://web.cs.mun.ca/~paul/cs3724/material/web/notes/node25.html

[19] Lov Grover, "A fast quantum mechanical algorithm for database search," in *Annual ACM Symposium on Theory of Computing*, Philadelphia, 1996, pp. 212-219.

[20] Donald O Hebb, *The Organization of Behavior*. New York, United States of America: John Wiley and Sons, Inc., 1949, p. 62.

[21] Stephen Hobson and Jim Austin, "Improved Storage Capacity in Correlation Matrix Memories Storing Fixed Weight Codes," in *19th International Conference on Artificial Neural Networks: Part I*, Limassol, 2009, pp. 728-736.

[22] Victoria J Hodge and Jim Austin, "A Comparison of Standard Spell Checking Algorithms and a Novel Binary Neural Approach," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 5, pp. 1073-1081, September-October 2003.

[23] Damian Isla, "Handling Complexity in the Halo 2 AI," in *Game Developers Conference*, San Francisco, March 2005.

[24] Yaochu Jin. (2010, March) Soft Computing. [Online]. http://www.soft-computing.de

[25] Phillip Kaye, Raymond Laflamme, and Michele Mosca, *An Introduction to Quantum Computing*. Oxford, United Kingdom: OUP Oxford, 2007.

[26] Stefan Klinger and Jim Austin, "Chemical Similarity Searching Using a Neural Graph Matcher," in *European Symposium on Artificial Neural Networks*, Bruges, 2005.

[27] Donald Knuth, *The Art of Computer Programming, Volume 4, Fascicle 2*. Reading, United States of America: Addison-Wesley Professional, 2005.

[28] Teuvo Kohonen, Erkki Oja, and Pekka Lehtiö, "Storage and Processing of Information in Distributed Associative Memory Systems," in *Parallel Models of Associative Memory, Updated Edition*, Geoffrey E Hinton and James A Anderson, Eds. Hillsdale, United States of America: Lawrence Erlbaum Associates, Inc., 1989, ch. 4, pp. 129-170.

[29] Daniel Kustrin and Jim Austin, "Connectionist Propositional Logic: A Simple Correlation Matrix Memory Based Reasoning System," in *Emergent neural computational architectures based on neuroscience: towards neuroscience-inspired computing*, Stefan Wermter, Jim Austin, and David Willshaw, Eds. New York, United States of America: Springer-Verlag, 2001, pp. 534-546.

[30] Peter Linz, *An Introduction to Formal Languages and Automata*, 4th ed. Sudbury, United States of America: Jones and Bartlett Publishers, 2006.

[31] John C Martin, *Introduction to Languages and the Theory of Computation*, 3rd ed. Boston, United States of America: McGraw-Hill, 2003.

[32] Edward F Moore, "Gedanken Experiments on Sequential Machines," in *Automata Studies*, J. McCarthy C. E. Shannon, Ed., 1956, pp. 129-153.

[33] Bernard M Moret, *The Theory of Computation*. Reading, United States of America: Addison-Wesley, 1998.

[34] Michael A Nielsen and Isaac L Chuang, *Quantum Computation and Quantum Information*. Cambridge, United Kingdom: Cambridge University Press, 2000.

[35] Christos Orovas and Jim Austin, "Cellular Associative Symbolic Processing for Pattern Recognition," in *MFCS '98 workshop on Grammar Learning*, Opava, 1998, pp. 269-280.

[36] Brian D Ripley, *Pattern Recognition and Neural Networks*. Cambridge, United Kingdom: Cambridge University Press, 1996.

[37] Raul Rojas, *Neural Networks - A Systematic Introduction*. New York, United States of America: Springer, 1996.

[38] Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Upper Saddle River, United States of America: Prentice Hall, 2003.

[39] Robert Schalkoff, *Pattern Recognition: Statistical, Structural and Neural Approaches*. New York, United States of America: John Wiley & Sons, Inc., 1992.

[40] Murray Shanahan and Richard Southwick, *Search, Inference and Dependencies in Artificial Intelligence*. Chichester, United Kingdom: Ellis Horwood Limited, 1989.

[41] Peter Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Foundations of Computer Science*, Santa Fe, 1994, pp. 124-134.

[42] Patrick K. Simpson, *Artificial Neural Systems: Foundations, Paradigms, Applications, and Implementations*. New York, United States of America: Pergamon Press, Inc., 1990.

[43] F S Smailbegovic, G N Gaydadjiev, and S Vassiliadis, "Sparse Matrix Storage Format," in *16th Annual Workshop on Circuits, Systems and Signal Processing, ProRisc*, Veldhoven, 2005, pp. 445-448.

[44] Michael Turner and Jim Austin, "Matching performance of binary correlation matrix memories," *Transactions of the Society for Computer Simulation International*, vol. 14, no. 4, pp. 1637-1648, December 1997.

[45] David J Willshaw, *Parallel models of associative memories*, G E Hinton and J A Anderson, Eds. Hillsdale, United States of America: Erlbaum, 1981.

[46] David J Willshaw, O P Buneman, and H C Longuet-Higgins, "Non-Holographic Associative Memory," *Nature*, vol. 222, no. 5197, pp. 960-962, June 1969.

[47] Richard Wilson, "Neural Networks," in *Unpublished, available from http://www-course.cs.york.ac.uk/pat/book*. United Kingdom, 2010, ch. 7.

[48] Richard Wilson. (2009, November) Pattern Recognition and Neural Networks Course website. [Online]. http://www-course.cs.york.ac.uk/pat

[49] Lotfi A Zadeh, "Fuzzy Logic, Neural Networks, and Soft Computing," *Communications of the ACM*, vol. 37, no. 3, pp. 77-84, March 1994.

Glossary

This glossary is not intended as an exhaustive list, it serves merely to clarify the meanings of terms used in this work that may otherwise be ambiguous or not entirely clear.

Arity network – the arity of a production is the number of tokens in its antecedent; an arity network is a series of CMMs that stores rules with a given arity

Advanced Uncertain Reasoning Architecture (AURA) – an implementation of a CMM and supporting architecture developed for the manipulation of symbolic knowledge

Correlation Matrix Memory (CMM) – a memory that stores the associations between an input and output vector

Hebbian learning – the weight of any neurons that represent a correlation between an input and output vector is set to logical 1; specifically the tensor product of the input and output vectors is generated, and any correlations are set to a logical 1 in the CMM

Lexicographic generation – generation of vectors in a deterministic manner, using a language such as pure binary or gray codes

Production system – a production is a rule that moves from a set of antecedents to a set of consequents; a production system uses a set of productions to move from an initial state to a goal

Sparse matrix – an efficient storage method for a matrix that only stores the positions of any logical 1s

Stacked tensor product – a tensor product that is represented as a vector; each column of the tensor product is 'stacked' on top of the others

State – a token / vector that is assigned to every production, and used in forming tensor products, in order to maintain separation of rules

Tensor – a vector used in forming tensor products

Tensor product – the result of an outer product operation between two vectors

Threshold – the output of a recall operation is a pattern containing a series of integers; applying a suitable threshold to this pattern results in a vector containing logical 0s and 1s selected as those positions that had an integer greater than or equal to the threshold value

Token – the string that is associated with a particular vector, in order to aid human-computer interaction

Vector – a series of logical 0s and 1s that represent a particular piece of data; the weight of a vector is the number of logical 1s that the vector contains

Appendices

Appendix 1. Source Code

This appendix provides only header files for the classes and functions developed in this work. A full source listing, including the control structures used for experimentation, is contained on the attached CD.

1.1 matrix.h

```
// Define the CMM class
class cmmType {
    int inLength;
    int outLength;
    int inWeight;
    int outWeight;
    vector<bool> matrix;
    bool read(int inputPosition, int outputPosition);
    void set(int inputPosition, int outputPosition);
    void unset(int inputPosition, int outputPosition);
    int sparseSize;
  public:
    cmmType(int inputLength, int outputLength, int inputWeight, int
outputWeight);
    ~cmmType();
    void print(void);
    int size(bool sparse = true);
    bool train(vectorType inputVector, vectorType outputVector);
    patternType recallRaw(vectorType inputVector);
};
```

1.2 rules.h

```
// Define the rule structure
struct ruleType {
  vectorType input;
  vectorType state;
  vectorType output;
};

// Define the rules storage class
class rulesType {
    vector<ruleType> trainedRules;
  public:
    rulesType();
    ~rulesType();
    int storeRule(ruleType newRule);
    int ruleCount(void);
    void deleteRule(int position = -1);
    ruleType getRule(int position = -1);
    ruleType findMachine(ruleType inputRule);
    vectorType findMachine(string token);
    vectorType findHuman(vectorType vector, bool state = false);
    int findInput(vectorType output);
};
```

1.3 functions.h

```
// Calculate the possible vectors for a given length and weight, without
using factorial
int fnPossibleVectors(int vectorLength, int vectorWeight);

// Calculate the possible Baum vectors for a given set of lengths and
weight
int fnPossibleBaum(int *tensorBaumLengths, int tensorWeight);

// Generate an output vector using Willshaw thresholding given an input of
ints
vectorType fnWillshawOutput(patternType *outputPattern, int willshawValue =
WILLSHAWLEVEL);

// Generate an output vector using L-max thresholding given an input of
ints
vectorType fnLmaxOutput(patternType *outputPattern);

// Write output to a file
bool fnWriteOutput(string filename, int mode, string output);

// Convert a vector machine representation to a string
string fnVectorToString(vectorType *printVector, bool shortForm = false);

// Convert an output pattern to a string
string fnPatternToString(patternType *printPattern);

// Performs a tensor operation on two vectors
vectorType fnPerformTensor(vectorType vector, vectorType tensor);

// Recalls the vector associated with a tensor within a tensor product
patternType fnRecallVectorRaw(vectorType tensorProduct, vectorType tensor);

// Recalls the tensor associated with a vector within a tensor product
patternType fnRecallTensorRaw(vectorType tensorProduct, vectorType vector);

// Reads rules from a file, and returns them in a rulesType
rulesType fnParseFile(string filename = "trainingData.txt");

// Parses an input rule
ruleType fnParseRule(string strRule);

// Parses an input recall
ruleType fnParseRecall(string strRecall);

// Superimposes two vectors
vectorType fnSuperimposeTensors(vectorType first, vectorType second);

// Sums two patterns
patternType fnSumPatterns(patternType first, patternType second);

// Sums a vector and a pattern
patternType fnSumPatterns(vectorType first, patternType second);

// Compares the rules found within a tensor product to those stored in a
ruleset
int fnCompareRules(vectorType tensorProduct, rulesType ruleSet, int
tensorLength, int tensorWeight, int vectorLength, int vectorWeight, bool
baum, int *tensorLengths = NULL, bool matrix2 = false);
```

```
// Finds the vectors within an intermediate output (i.e. states after first
matrix recall)
vector<vectorType> fnFindVectorsInIntermediate(vectorType tensorProduct,
rulesType ruleSet, int tensorLength, int tensorWeight, bool baum, int
*tensorLengths = NULL);

// Finds the vectors within a final output (i.e. output after second matrix
recall)
vector<vectorType> fnFindVectorsInFinal(vectorType tensorProduct, rulesType
ruleSet, int tensorLength, int tensorWeight, int vectorLength, int
vectorWeight, bool baum, int *tensorLengths = NULL);

// Calculates all possible tensors within a tensor product
vector<vectorType> fnCollapseToTensors(vectorType tensorProduct, int
tensorLength, int tensorWeight, int vectorLength, int vectorWeight, bool
baum, int *tensorLengths = NULL);

// Recalls the tensor within a tensor product
patternType fnCollapseTensorRaw(vectorType tensorProduct, int tensorLength,
int tensorWeight, int vectorLength);
```

1.4 vector.h

```
// Define the pattern structure
struct patternType {
  vector<int> patternInts;
  int weight;
  int length;
};

// Define the vector structure
struct vectorType {
  string humanString;
  vector<bool> machineString;
  int weight;
  int length;
};

// Define the vector generator class
class vectorGeneratorType {
    int length;
    int weight;
    int type;
    vector<int> lengths;
    vector<int> positions;
    vectorType createRandom(vectorType output);
    vectorType createBaum(vectorType output);
    vectorType createComplete(vectorType output);
  public:
    vectorGeneratorType(int vectorLength, int vectorWeight, int
generationType = BAUM, int *vectorLengths = NULL);
    ~vectorGeneratorType();
    vectorType create(void);
    vectorType create(string humanString);
};
```

Appendix 2. Test cases

All test cases are given in the format:

Test case number	Function to be tested
Test execution	
Result	

2.1 Iteration 1, Vector recall

1 fnVectorToString

1. Initialise two vectorTypes using the vectors 100010 and 010001
2. Print the output of this function to the console and compare with the original vectors
3. Print the short form output (i.e. the positions of each set bit) to the console and compare with the original vectors

Pass

2 fnPatternToString

1. Initialise two patternTypes using the sequences {1 0 2 2 1 0} and {0 2 1 2 0 1}
2. Print the output of this function to the console and compare with the original integers

Pass

3 vectorGeneratorType::createRandom

1. Generate random vectors with length 5 and weight 2, printing them to the console with fnVectorToString
2. Continue until all 10 possible vectors have been created

Fail, in certain situations the (int) casting resulted in rand() % 1

Retest after fix: *Pass*

4 vectorGeneratorType::createBaum

1. Generate Baum vectors with weight 2 and lengths {2, 3}, printing them to the console with fnVectorToString
2. Continue until all 6 possible vectors have been created deterministically

Pass

5 vectorGeneratorType::create

1. Initialise two vectorGeneratorTypes – one to create random vectors and one for Baum generated vectors
2. Create a vector using each of these objects

Pass

6 cmmType::train (and cmmType::set)

1. Declare and instantiate a variable of type cmmType
2. Train a vector pair into the CMM
3. Use Visual Studio debugging facilities to ensure it has been trained correctly, comparing to manually calculated positions

Pass

7 cmmType::print (and cmmType::read)

1. Declare and instantiate a variable of type cmmType
2. Train 3 vector pairs into the CMM
3. Manually calculate the matrix, and compare this with the printed output

Fail, the matrix is rotated by 90^{o}

Retest after fix: *Pass*

8	cmmType::recallRaw

1. Train 3 vector pairs into the CMM, ensuring that interference will occur between them
2. Manually calculate the raw output of a recall operation performed using the first input vector trained, and compare this with the recalled output using fnPatternToString

Pass

9	fnWillshawOutput

1. Initialise two patternTypes using the sequences {1 0 2 2 1 0} and {0 2 1 2 0 1}
2. Apply a threshold, using a threshold value of 1 (101110 / 011101)
3. Apply a threshold, using a threshold value of 2 (001100 / 010100)
4. Apply a threshold, using a threshold value of 3 (000000 / 000000)

Pass

10	fnLmaxOutput

1. Initialise two patternTypes using the sequences {1 0 2 2 1 0} and {0 2 1 2 0 1}
2. Apply a threshold, using a weight of 1 (001100 / 010100)
3. Apply a threshold, using a weight of 2 (001100 / 010100)
4. Apply a threshold, using a weight of 3 (101110 / 011101)
5. Apply a threshold, using a weight of 4 (101110 / 011101)
6. Apply a threshold, using a weight of 5 (101110 / 011101)

Fail, using a weight of 5 results in zeros in the pattern being set to 1 in the output vector
Retest after fix: *Pass*

11	fnWriteOutput

1. Call the function with a valid filename, ios::out, and a string to be written

Pass

12	rulesType::storeRule

1. Create two new rules, with random vectors
2. Store these in an object of rulesType, and ensure they are correctly stored using Visual Studio's debugging facilities

Pass

13	rulesType::ruleCount

1. Store a number of rules within a rulesType
2. Print the output of a call to the ruleCount

Pass

14	rulesType::getRule

1. Store a number of rules within a rulesType, keeping track of exactly what they are
2. Retrieve each rule in turn, comparing the printed output with what was stored

Pass

2.2 Iteration 2, Tensor product recall

15 fnPerformTensor

1. Generate two vectors, and print them to the console
2. Manually calculate the tensor product of these vectors
3. Print the output of this function to the console, for comparison

Pass

16 cmmType::recallRaw

1. Train 3 vector pairs into the CMM
2. Manually calculate the raw output of a recall operation performed first using a standard input vector and then again using a tensor product, and compare this with the recalled output using fnPatternToString

Pass

17 fnRecallVectorRaw

1. Generate a tensor product, using a known vector and tensor
2. Manually calculate the expected output pattern, and compare with the output of this function using fnPatternToString

Pass

2.3 Iteration 3, Rule-chaining with tensor product recall

18 fnParseRule

1. Pass the function two strings, “a->b” and “a->b->c”
2. The output should be two rules {input a, output b} and {input a, state b, output c}

Pass

19 fnParseFile

1. Create a text file containing some simple rules
2. Ensure that the output matches the rules

Pass

20 rulesType::findMachine(string)

1. Create a rulesType object and store some known rules
2. Search for a token that exists, and check that the returned vector is correct
3. Search for a token that does not exist, and check that the returned vector is empty

Pass

21 rulesType::findHuman

1. Create a rulesType object and store some known rules
2. Pass this function two vectors – one containing a vector that does exist in the rulesType, and one that does not exist
3. Check that the first returned vector contains the correct token, and that the second contains no token

Pass

22 vectorGeneratorType::create(string)

1. Create a random vector with a token passed as an input
2. Check that the returned vector is valid, and contains the input token

Pass

2.4 Iteration 4, Superposition of tensor products

23 vectorGeneratorType::createComplete

1. Generate lexicographic vectors with length 5 and weight 2, printing them to the console with fnVectorToString
2. Continue until all 10 possible vectors have been created deterministically

Pass

24 fnSuperimposeTensors

1. Call the function with inputs of a tensor product and a standard vector, and ensure that the output is the original tensor product
2. Call the function with inputs of two tensor products, and compare the output to a manually calculated superposition

Pass

25 fnPossibleVectors

1. Call the function with inputs of length: 10, weight: 3, and ensure that the output is 120
2. Call the function with inputs of length: 20, weight: 5, and ensure that the output is 15504

Fail, using the $\frac{n!}{(n-k)!k!}$ algorithm causes an overflow as $20! > 2^{32}$

Retest after fix: *Pass*

26 fnPossibleBaum

1. Call the function with inputs of lengths: {2, 3, 5}, weight: 3, and ensure that the output is 30
2. Call the function with inputs of lengths: {3, 5, 7, 8}, weight: 4, and ensure that the output is 840

Pass

27 fnCollapseTensorRaw

1. Superimpose multiple tensor products
2. Manually sum each of the columns of this tensor product, and compare with the output pattern produced by this function

Pass

28 fnCollapseToTensors

1. Superimpose multiple tensor products
2. Manually sum each of the columns of this tensor product, threshold, and calculate the possible tensors that may exist within the tensor product (using both random and Baum generation)
3. Compare these with the generated output

Pass

29 rulesType::deleteRule

1. Delete a specific rule within a rulesType
2. Delete the latest rule using a number larger than the number of rules

Pass

30 rulesType::findInput

1. Search for a rule that contains an output vector that does exist, and ensure the correct position is returned
2. Search for a rule that contains an output vector that does not exist, and ensure that -1 is returned

Pass

31 fnRecallTensorRaw

1. Generate a tensor product, using a known vector and tensor
2. Manually calculate the expected output pattern, and compare with the output of this function using fnPatternToString

Pass

32 fnCompareRules

1. Ensure that the output is the number of tensor products superimposed when an empty set of rules is passed
2. Ensure that the output is the number of rules when an empty tensor product is passed
3. Ensure that the output is zero when the tensor products superimposed match the set of rules (superimpose only a small number of tensor products to reduce interference)

Pass

2.5 Iteration 5, Use of a two-layer network

33 cmmType::size

1. Create a CMM with input and output lengths of 5, and ensure that the sparse size is 0, while the not sparse size is 25
2. Train 3 vector pairs, manually keeping track of the state of the matrix, and compare the sparse size with the manual state

Pass

34 fnSumPatterns

1. Call the function with inputs of two patterns, and ensure that the output is the sum of these two patterns
2. Call the function with inputs of a vector and an empty pattern, and ensure that the output is a pattern of the same length as the input vector, containing 1s at each relevant position
3. Call the function with inputs of a vector and a non-empty pattern, and ensure that the output is the original pattern with each relevant position from the vector incremented

Pass

35 rulesType::findMachine(rule)

1. Create a rulesType object and store some known rules
2. Create a ruleType structure, with a mixture of tokens that do and do not exist in the rulesType object
3. Check that the returned ruleType contains the correct vectors for each of the tokens

Fail, if the token was not found, the vector length would be undefined
Retest after fix: *Pass*

36 fnFindVectorsInIntermediate

1. Create a tensor product using a series of known rules with tokens
2. Manually calculate the possible states in this tensor product, including their tokens if they exist
3. Compare this with the set of states returned by this function

Pass

37 fnFindVectorsInFinal

1. Create a tensor product using a series of known rules with tokens
2. Manually calculate the possible vectors in this tensor product, including their tokens if they exist
3. Compare this with the set of vectors returned by this function

Pass

38 fnParseRecall

1. Pass the function two strings, “a” and “a:b”
2. The output should be two rules {input a} and {input a, state b}

Pass

2.6 Demonstration system

2.6.1 Rule set 1

Rules

a->b	e->f	i->j	m->n	q->r	u->v	y->z
b->c	f->g	j->k	n->o	r->s	v->w	
c->d	g->h	k->l	o->p	s->t	w->x	
d->e	h->i	l->m	p->q	t->u	x->y	

Input

```
p, {enter}, q, {enter}, r, {enter}, recall, {enter}
```

Expected output

```
  Loop 1:
    First matrix:
      State: newState15
      State: newState16
      State: newState17
    Second matrix:
      Output: q
      Output: r
      Output: s
  Loop 2:
    First matrix:
      State: newState16
      State: newState17
      State: newState18
    Second matrix:
      Output: r
      Output: s
      Output: t
  Loop 3:
    First matrix:
      State: newState17
      State: newState18
      State: newState19
    Second matrix:
      Output: s
      Output: t
      Output: u
  Loop 4:
    First matrix:
      State: newState18
      State: newState19
      State: newState20
    Second matrix:
      Output: t
      Output: u
      Output: v
  Loop 5:
    First matrix:
      State: newState19
      State: newState20
      State: newState21
    Second matrix:
      Output: u
      Output: v
      Output: w
  Loop 6:
    First matrix:
      State: newState20
      State: newState21
      State: newState22
    Second matrix:
      Output: v
      Output: w
      Output: x
  Loop 7:
    First matrix:
      State: newState21
      State: newState22
      State: newState23
    Second matrix:
      Output: w
      Output: x
      Output: y
  Loop 8:
    First matrix:
      State: newState22
      State: newState23
      State: newState24
    Second matrix:
      Output: x
      Output: y
      Output: z
  Loop 9:
    First matrix:
      State: newState23
      State: newState24
    Second matrix:
      Output: y
      Output: z
  Loop 10:
    First matrix:
      State: newState24
    Second matrix:
      Output: z
  Loop 11:
    First matrix:
      No states recognised
    Second matrix:
      No output vectors recognised
```

Pass

2.6.2 Rule set 2

Rules

a->b	c->f	d->j	l->n	m->r	h->v	x->z
a->c	c->g	d->k	n->o	g->s	h->w	
b->d	d->h	k->l	n->p	g->t	w->x	
b->e	d->i	l->m	m->q	h->u	x->y	

Input

```
a, {enter}, recall, {enter}
```

Expected output

```
  Loop 1:
    First matrix:
      State: newState0
      State: newState1
    Second matrix:
      Output: b
      Output: c
  Loop 2:
    First matrix:
      State: newState2
      State: newState3
      State: newState4
      State: newState5
    Second matrix:
      Output: d
      Output: e
      Output: f
      Output: g
  Loop 3:
    First matrix:
      State: newState6
      State: newState7
      State: newState8
      State: newState9
      State: newState17
      State: newState18
    Second matrix:
      Output: h
      Output: i
      Output: j
      Output: k
      Output: s
      Output: t
  Loop 4:
    First matrix:
      State: newState19
      State: newState20
      State: newState21
      State: newState10
    Second matrix:
      Output: u
      Output: v
      Output: w
      Output: l
  Loop 5:
    First matrix:
      State: newState22
      State: newState11
      State: newState12
    Second matrix:
      Output: x
      Output: m
      Output: n
  Loop 6:
    First matrix:
      State: newState23
      State: newState24
      State: newState15
      State: newState16
      State: newState13
      State: newState14
    Second matrix:
      Output: y
      Output: z
      Output: q
      Output: r
      Output: o
      Output: p
  Loop 7:
    First matrix:
      No states recognised
    Second matrix:
      No output vectors recognised
```

Pass, not all states were correctly recognised due to interference, but all output vectors were recalled perfectly

2.6.3 Rule set 3

Rules	
a->b	b->a
Input	
a, {enter}, recall, {enter}	
Expected output	

```
  Loop 1:
    First matrix:
      State: newState0
    Second matrix:
      Output: b
  Loop 2:
    First matrix:
      State: newState1
    Second matrix:
      Output: a
  Loop 3:
    First matrix:
      State: newState0
    Second matrix:
      Output: b
  Loop 4:
    First matrix:
      State: newState1
    Second matrix:
      Output: a
  Loop 5:
    First matrix:
      State: newState0
    Second matrix:
      Output: b
  Loop 6:
    First matrix:
      State: newState1
    Second matrix:
      Output: a
  Loop 7:
    First matrix:
      State: newState0
    Second matrix:
      Output: b
  Loop 8:
    First matrix:
      State: newState1
    Second matrix:
      Output: a
  Loop 9:
    First matrix:
      State: newState0
    Second matrix:
      Output: b
  Loop 10:
    First matrix:
      State: newState1
    Second matrix:
      Output: a
...
```

```
...
  Loop 21:
    First matrix:
      State: newState0
    Second matrix:
      Output: b
  Loop 22:
    First matrix:
      State: newState1
    Second matrix:
      Output: a
  Loop 23:
    First matrix:
      State: newState0
    Second matrix:
      Output: b
  Loop 24:
    First matrix:
      State: newState1
    Second matrix:
      Output: a
  Loop 25:
    First matrix:
      State: newState0
    Second matrix:
      Output: b
  Loop 26:
    First matrix:
      State: newState1
    Second matrix:
      Output: a
  Loop 27:
    First matrix:
      State: newState0
    Second matrix:
      Output: b
  Loop 28:
    First matrix:
      State: newState1
    Second matrix:
      Output: a
  Loop 29:
    First matrix:
      State: newState0
    Second matrix:
      Output: b
  Loop 30:
    First matrix:
      State: newState1
    Second matrix:
      Output: a
```

Pass

Appendix 3. Results

3.1 Iteration 1

Vector weight	Number of vector pairs
2	178.90
3	344.30
4	524.85
5	624.70
6	651.55
7	679.45
8	676.80
9	680.15
10	651.10
11	603.40
12	560.75
13	538.30
14	493.75
15	463.90

Table 1: Number of vector pairs recalled against vector weight

Table 1 shows the mean number of vector pairs that may be stored before any recall fails.

3.2 Iteration 4

Tensor length	TPs super-imposed	Tensor length	TPs super-imposed	Tensor length	TPs super-imposed	Tensor length	TPs super-imposed
5	10	29	89	53	89	77	89
6	15	30	89	54	89	78	89
7	21	31	89	55	89	79	89
8	28	32	89	56	89	80	89
9	36	33	89	57	89	81	89
10	45	34	89	58	89	82	89
11	55	35	89	59	89	83	89
12	66	36	89	60	89	84	89
13	78	37	89	61	89	85	89
14	89	38	89	62	89	86	89
15	89	39	89	63	89	87	89
16	89	40	89	64	89	88	89
17	89	41	89	65	89	89	89
18	89	42	89	66	89	90	89
19	89	43	89	67	89	91	90
20	89	44	89	68	89	92	91
21	89	45	89	69	89	93	92
22	89	46	89	70	89	94	93
23	89	47	89	71	89	95	94
24	89	48	89	72	89	96	95
25	89	49	89	73	89	97	96
26	89	50	89	74	89	98	97
27	89	51	89	75	89	99	98
28	89	52	89	76	89	100	99

Table 2: Number of tensor products (TPs) successfully recalled for a given tensor length

Table 2 shows the number of tensor products that may be superimposed before a tensor product fails to achieve perfect recall reliability.

3.3 Iteration 5

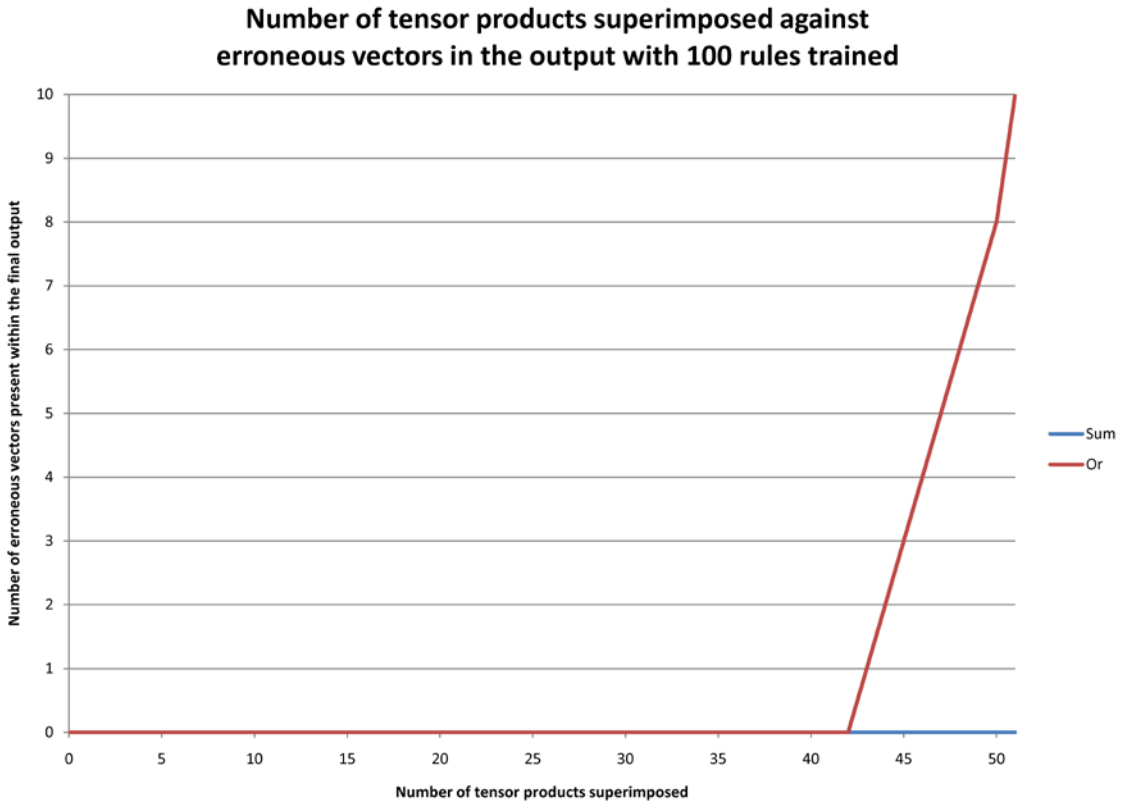


Figure 1: Number of erroneous vectors in the output of a system trained with 100 rules

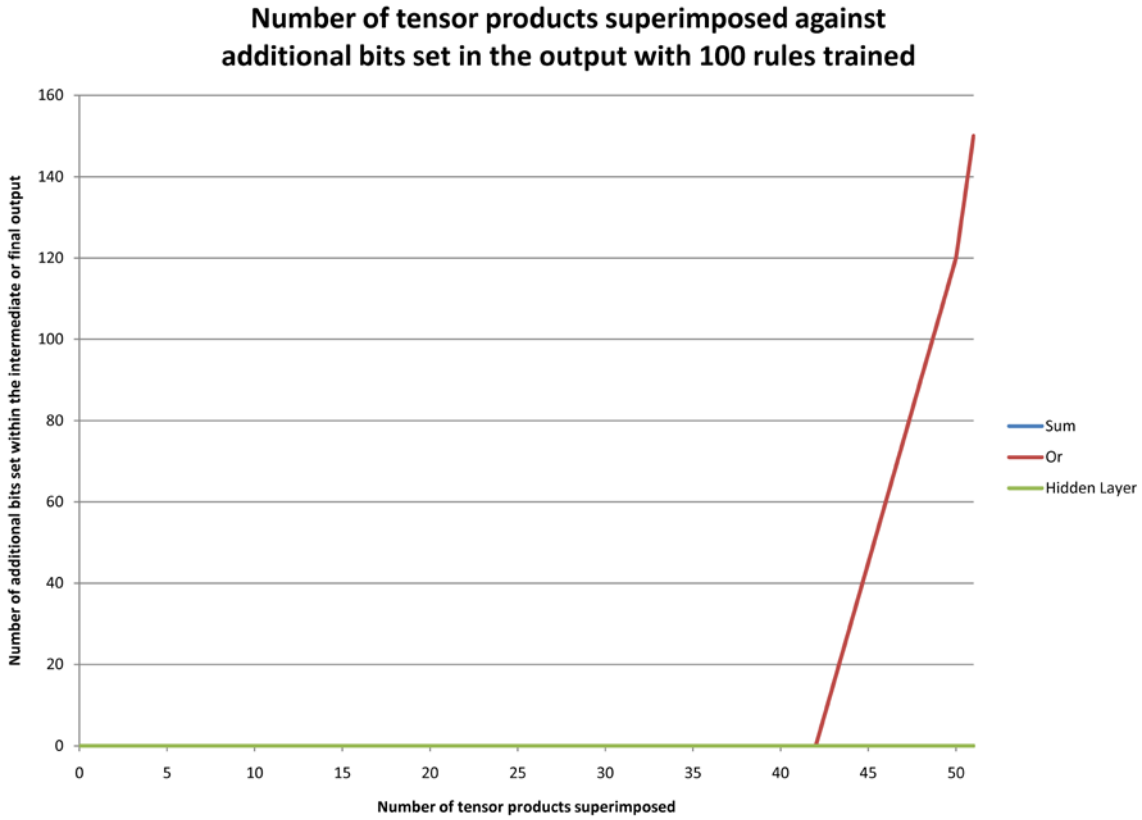


Figure 2: Number of additional bits set in the output of a system trained with 100 rules

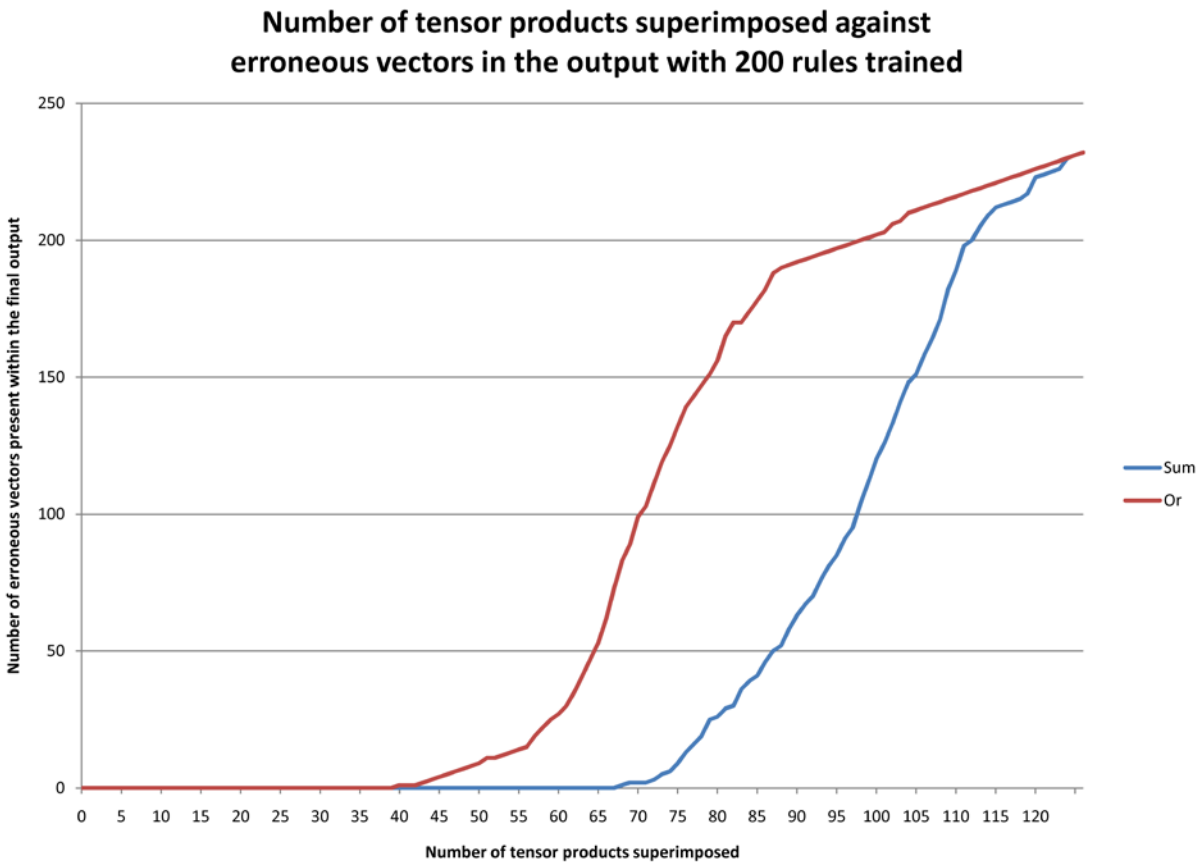


Figure 3: Number of erroneous vectors in the output of a system trained with 200 rules

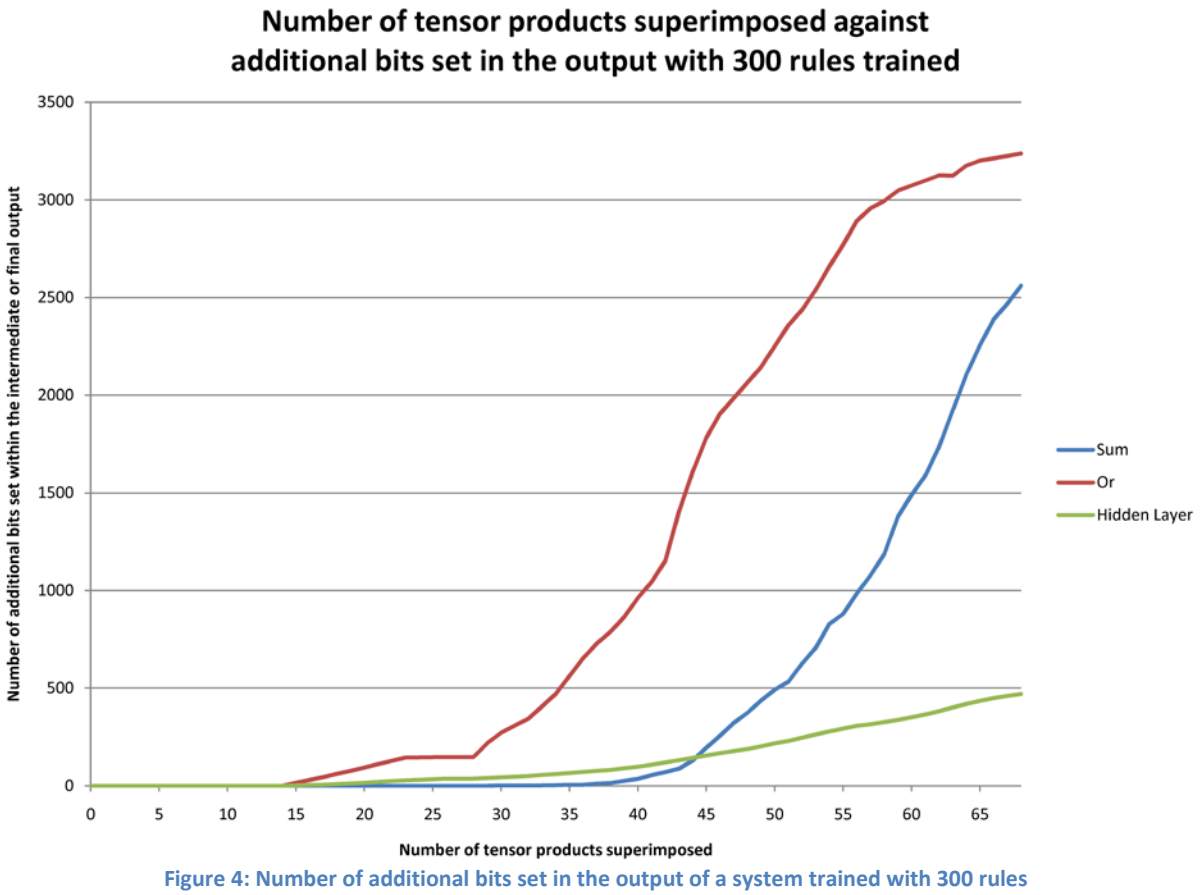


Figure 4: Number of additional bits set in the output of a system trained with 300 rules