Extending the Associative Rule Chaining Architecture for Multiple Arity Rules

Nathan Burles, James Austin, and Simon O'Keefe
Advanced Computer Architectures Group
Department of Computer Science
University of York
York, YO10 5GH, UK
{nburles,austin,sok}@cs.york.ac.uk

Abstract

The Associative Rule Chaining Architecture uses distributed associative memories and superimposed distributed representations in order to perform rule chaining efficiently [Austin *et al.*, 2012]. Previous work has focused on rules with only a single antecedent, in this work we extend the architecture to work with multiple-arity rules and show that it continues to operate effectively.

1 Introduction

Rule chaining is a common problem in the field of artificial intelligence; searching a set of rules to determine if there is a path from the starting state to the goal state. The Associative Rule Chaining Architecture (ARCA) [Austin *et al.*, 2012] performs rule chaining using correlation matrix memories (CMMs)—a simple type of associative neural network [Kohonen, 1972].

1.1 Rule Chaining

Rule chaining can be used to describe both forward and backward chaining, the choice of which to use is application specific. In this work we use forward chaining, working from the starting state towards a goal state, although there is no reason that the architecture could not be used to perform backward chaining.

In forward chaining, a search begins with an initial set of conditions that are known to be true. All of the rules are searched to find one for which the antecedents match these conditions, and the consequents of that rule are added to the current state. This state is checked to decide if the goal has been reached, and if it has not then the system will iterate. If no further matching rules can be found, then the search will result in failure.

1.2 Correlation Matrix Memories (CMMs)

The CMM is a simple neural network containing a single layer of weights. This means that the input and output neurons are fully connected, and simple Hebbian learning can be used [Ritter *et al.*, 1992].

In this work we use binary CMMs [Willshaw *et al.*, 1969], a sub-class of CMMs where the weights are binary. Learning to associate pairs of binary vectors is thus an efficient operation, and requires only local updates to the CMM. This is formalised in Equation 1, where $\mathbf{M}$ is the resulting CMM (matrix of binary weights), $\mathbf{x}$ is the set of input vectors, $\mathbf{y}$ is the set of output vectors, and n is the number of training pairs. The CMM is formed by taking the logical OR, $\bigvee$, of the outer products formed between each of the training pairs.

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

A recall operation can be performed as a matrix multiplication between a transposed input vector and the CMM. This results in a non-binary output vector, to which a threshold function f must be applied in order to produce the final binary output vector, as shown in Equation 2.

$$\mathbf{y} = f(\mathbf{x}^T \mathbf{M}) \tag{2}$$

Using the fact that the input vector contains only binary components, this recall operation may be optimised. To calculate the j^{th} bit of an output vector when performing a matrix multiplication, one finds the vector dot product of the input vector $\mathbf{x}$ and the j^{th} column of the matrix $\mathbf{M}$. This vector dot product is defined as $\sum_{i=1}^{n} \mathbf{x}_i \mathbf{M}_{j,i}$, where $\mathbf{M}_{j,i}$ is the value stored in the j^{th} column of the i^{th} row of the CMM $\mathbf{M}$. As $\mathbf{x}$ is known to be a binary vector, it is clear that the result of this dot product is equal to the sum of all values $\mathbf{M}_{j,i}$ where $\mathbf{x}_i = 1$, as shown in Equation 3.

$$y_j = f\left(\sum_{i(\mathbf{x}_i=1)} \mathbf{M}_{j,i}\right) \tag{3}$$

The choice of which threshold function (f) to apply depends on the application and the data representation used. When storing fixed-weight vectors, the L-max threshold has been shown to be effective [Austin, 1996]. Alternatively, when using Baum's algorithm to generate vectors, the L-wta threshold has been shown to increase a CMM's storage capacity [Hobson and Austin, 2009]. When superimposed vectors are used, as they are in ARCA, the selection of threshold function is limited to Willshaw thresholding, where any output bit with a value at least equal to the trained input weight is set to one [Willshaw *et al.*, 1969].

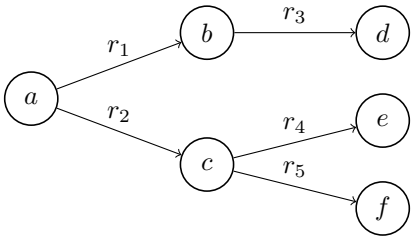

Figure 1: An example tree of rules with a maximum branching factor of 2. Each input token causes at least 1 and at most 2 rules to fire.

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

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

Table 1: An example set of tokens with a binary vector allocated to each, and the set of rules used in examples.

1.3 Associative Rule Chaining

The Associative Rule Chaining Architecture uses distributed representations [Austin, 1992] to allow superposition of multiple states in a single vector, while also providing more efficient memory use and a greater tolerance to faults than a local representation [Baum *et al.*, 1988]. As an example, the superposition of two distributed vectors $\{10100\}$ and $\{10010\}$ is the vector $\{10110\}$.

ARCA performs rule chaining using superimposed representations, and hence reduces the time complexity of a tree search by searching an entire level of the tree in a single operation. A traditional method such as depth-first search would search each rule in turn, resulting in a time complexity of $O(b^d)$, where b is the branching factor, and d is the depth of the tree. By allowing superposition of states, ARCA reduces this to $O(d)$—considering the tree of rules in Figure 1 as an example, if presented with an input a, both rules r_1 and r_2 are considered simultaneously. When the system iterates the next level of rules are then searched concurrently—r_3, r_4, and r_5.

Training

ARCA separates the antecedents and consequents of a rule into two CMMs using a unique "rule vector" that exists in a different vector space to the tokens.

The antecedent CMM associates the antecedents of each rule with its rule vector—for example $a : r_1$. The rule will then fire if its antecedents are present during a later recall.

The consequent CMM then associates each rule with the consequents of that rule. In order to maintain separation between multiple, superimposed branches during a recall this requires a slightly more complex method of training. Firstly we create a tensor, or outer, product between the rule vector and the consequents of a rule—represented as $r_1 : b$. We proceed by "flattening" this tensor product, in a row-major order, with the result being a single vector with a length equal to the product of the token and rule vector lengths. The rule vector can now be associated with this "flattened" tensor product, and stored in the consequent CMM—essentially this stores $r_1 : (b : r_1)$. This means that recalling a rule from the consequent CMM will produce a tensor product that contains the output tokens bound to the rule that caused them to fire.

It should be noted that, depending on the application, it may be possible and appropriate to prune the rule tree prior to training the ARCA network. Reducing the number of rules in this fashion will help to reduce the memory requirement of the system, however will have little effect on its operation.

Recall

Figure 2 shows a single pass through the ARCA system—searching one level of the tree. An input state is initialised by forming TP_{in}—the tensor product of any input tokens (in this case a) with a rule vector (r_0).

To determine which of the rules are matched by our input tokens, we recall each column of this tensor product in turn from the antecedent CMM. Each resulting vector is used as a column in a new tensor product—TP_{rule}. These columns may contain a number of superimposed vectors, representing any rules which fired.

In order to finish this pass through ARCA, we must now find the consequents of any rules which were matched. To do this, we recall each column of TP_{rule} from the consequent CMM. Due to the way this second CMM is trained, the result of each recall is a "flattened" tensor product containing rules bound to output tokens—these can be simply reformed to recover a number of tensor products, TP_{out}. To find our final result, TP_{sum}, we can sum these tensor products and apply a threshold at a value equal to the weight of a rule vector.

2 Multiple Arity Rules

In some cases, allowing only rules with a single antecedent may be sufficient. In complex rule chaining systems or applications such as feature matching, however, rules with more than one antecedent—multiple arity—may be necessary.

Rules with different arities cannot be stored in the same CMM, due to the operation of Willshaw's threshold function and the relaxation ability of CMMs. Table 1 assigns a binary vector to a number of tokens used in our continuing example, with a vector length of 7 and weight of 2.

We can demonstrate this issue by considering a system trained with the example rules r_1 to r_6, shown in Table 1. Upon presentation of a vector containing both a and b $\{1101100\}$ we require the system to match every rule that contains a or b in the antecedents: r_1, r_2, r_3, and r_6. To match the single-arity rules correctly the threshold value used must be equal to the weight of a single input vector, a value of 2.

Using a threshold value of only 2, however, means that presentation of a vector containing only a will match every rule that contains a in the antecedents: r_1, r_2, and r_6. This occurs because of the associative memory's ability to relax and

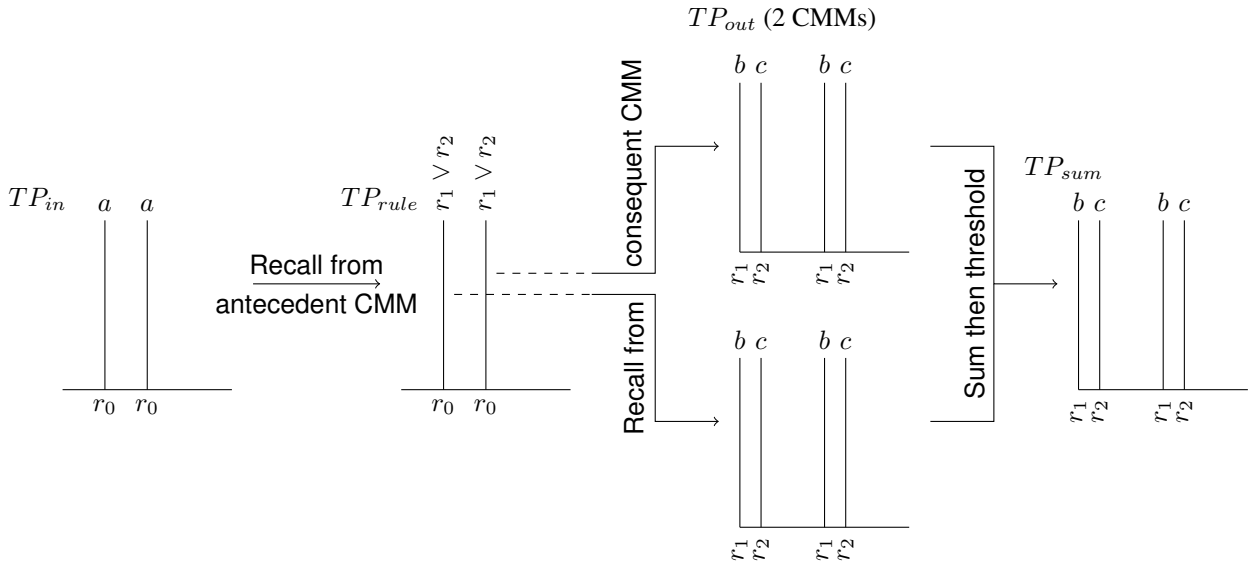

Figure 2: A visualisation of one iteration of the rule chaining process within ARCA. The process is initialised by creating a tensor product, TP_{in}, binding the input tokens (in this case a) to a rule vector (r_0). Each column of this tensor product is then recalled in turn from the antecedent CMM, to form a new tensor product, TP_{rule}, containing the rules which have fired. Each column of TP_{rule} can then be recalled in turn from the consequent CMM, resulting in a number of output tensor products (TP_{out}—one tensor product for every non-zero column of TP_{rule}). These output tensor products can be summed, to form a non-binary CMM, before a threshold is applied using a value equal to the weight of a rule vector to form TP_{sum}. The recall can continue in the same fashion, using TP_{sum} as the new input tensor product.

recognise partial inputs. In this case, however, it is undesirable behaviour as we only wish to match those rules for which all antecedents are present. Setting a threshold to resolve this case is impossible, as any value allowing the single-arity rules to match will also allow the 2-arity rule to match.

2.1 Training

One possible solution for this problem is to use arity networks, introduced in [Austin, 1996] and shown in Figure 3. Under this scheme, multiple distinct ARCA sub-networks are created—one for each arity of rules used in the system. Each rule is then trained into the correct n-arity ARCA sub-network for the number of tokens in its antecedent.

Although this scheme will help in many cases, it still does not fully solve the problem. Consider the 3-arity example rules given in Table 1. When recalling rule r_7 the superimposed tokens will form a vector $\{1001011\}$ with a weight of only 4, thus the threshold for the 3-arity network must be set as 4.

For rule r_8, the superposition of input tokens forms a vector $\{1011011\}$. It can clearly be seen that this vector is very similar to that of rule r_7, with the addition of only a single bit. Unfortunately, we have already determined that the threshold used with the 3-arity network must be at most 4. We can see, therefore, that presentation of the rule r_7 inputs ($a \wedge d \wedge g$) will cause rule r_8 to match incorrectly.

In order to resolve this we propose to modify the networks such that instead of separating rules by arity, they separate the rules by the combined weight of their superimposed antecedent tokens. This will operate in the same way as shown in Figure 3, but each ARCA network will be identified by the total weight of the superimposed antecedent tokens rather than by the number of antecedent tokens.

When training a rule its antecedents are first superimposed, and the weight of this vector determines in which of the ARCA networks the rule should be trained. This means that the threshold value for each ARCA network is well defined, while still allowing for relaxation if this is required by the application (by reducing this threshold value).

2.2 Recall

A block level diagram of the recall operation is shown in Figure 3. To initialise the recall any input tokens are superimposed, for example a, d, and g. The superimposed input vector is then recalled from each of the individual ARCA networks. As each ARCA network is distinct, this recall may happen in parallel where this is supported by the infrastructure. Given this particular input, the rules r_1 and r_2 will match in the weight-2 ARCA network, and r_7 will match in the weight-4 ARCA network. Rule r_8 will not be matched, as it is stored in the weight-5 ARCA network and so requires all 5 set bits to be present in the input.

After recall from each of the ARCA networks, the result is a number of vectors containing the consequent tokens for each of the matched rules. In order to be able to iterate we can simply superimpose these outputs.

Testing for search completion can operate in the same fashion as the single-arity ARCA [Austin *et al.*, 2012]. If the su-

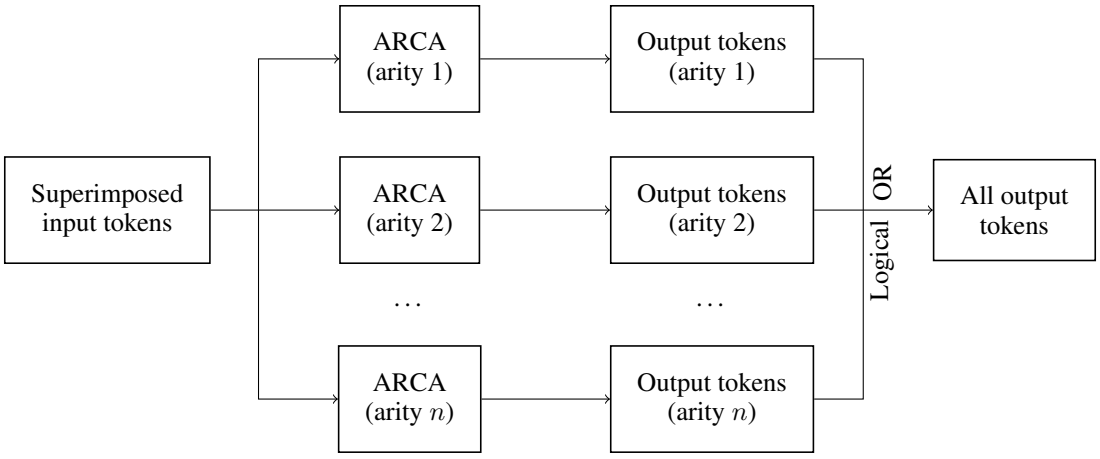

Figure 3: Using arity networks with ARCA. The input tokens are recalled from each arity network in turn, and the output tokens from each network are all superimposed to form the final result.

perimposed output consists solely of zeros then no rules have been matched and hence the search is completed without finding a goal state, if it is not empty then we must check whether all of the goal state's consequent tokens are present in the output—if they are, then the search has been successful.

3 Experiments

In order to show that our proposed solution is effective we implemented it using the Extended Neural Associative Memory Language (ENAMeL), a domain specific language created to simplify the development of applications using binary vectors and CMMs.

We generated a tree of rules, with a depth of 8 and a branching factor of 3 (where the number of children of a given node was uniformly randomly sampled from the range [1, 3]). The number of tokens in the antecedent of each rule, n, was randomly sampled from the range [1, 5], allowing a rule to share at most $\lceil n/2 \rceil$ antecedent tokens with any others. An example of this is shown in Figure 4. We selected these parameters to ensure that this experimentation would remain computationally feasible.

In the worst case these parameters will generate 3^8 rules. To ensure that any potential recall errors were not caused by undesired overlap between vectors, we decided to use a vector length of 5120 and weight of 10—extrapolating from results obtained during previous work [Austin *et al.*, 2012], this should comfortably allow over 10000 rules to be stored.

We then trained the multiple ARCA systems with each of the rules, generating a code for each unique token using Baum's algorithm [Baum *et al.*, 1988]. As we wished to test the correct operation and reliability of the multiple ARCA system, and not ARCA itself, we proceeded as follows:

1. Form the superposition of all tokens in the root of the rule tree to use as the starting state.
2. Recall the current state from each of the ARCA networks in turn, comparing the result to that which was trained—if the result differs from the expected value, then record the error.
3. Superimpose the results from each of the ARCA networks to form a new current state, and repeat from the previous step until all 8 levels of the tree have been searched.

This experimental design meant that if a recall error did occur, we would be able to easily determine the point of failure. We ran this experiment 100 times, using a different random seed each time, to ensure that different rule trees were used.

Our results showed that segregating rules by the weight of their superimposed antecedent tokens is effective, and allows correct operation of the rule chaining system in the presence of multiple-arity rules. In all of the experimental runs the recall was successful, with each ARCA system's output containing the correct tokens and no others.

In one of the runs, however, the output of the weight-40 ARCA network after the third iteration contained 8 set bits in addition to those which were expected. Upon iterating, this did not cause the erroneous recall of further incorrect bits, however with more iterations this could potentially occur again and cause an incorrect recall. The weight-40 ARCA network stored rules with 4 antecedent tokens that have no overlap when superimposed. Upon further analysis of the randomly generated rule tree, it seems that a disproportionately high number of rules were generated with this configuration. This caused the CMMs to become saturated, which means that too many bits in the matrix were set. This can result in erroneous bits being present after a recall operation, due to interference within the matrix.

4 Conclusions and Further Work

This paper has described an important extension to the ARCA, allowing it to perform forward chaining using rules that are not limited to a single antecedent token. We have shown that our proposed solution operates correctly, and is able to successfully store and recall large sets of rules.

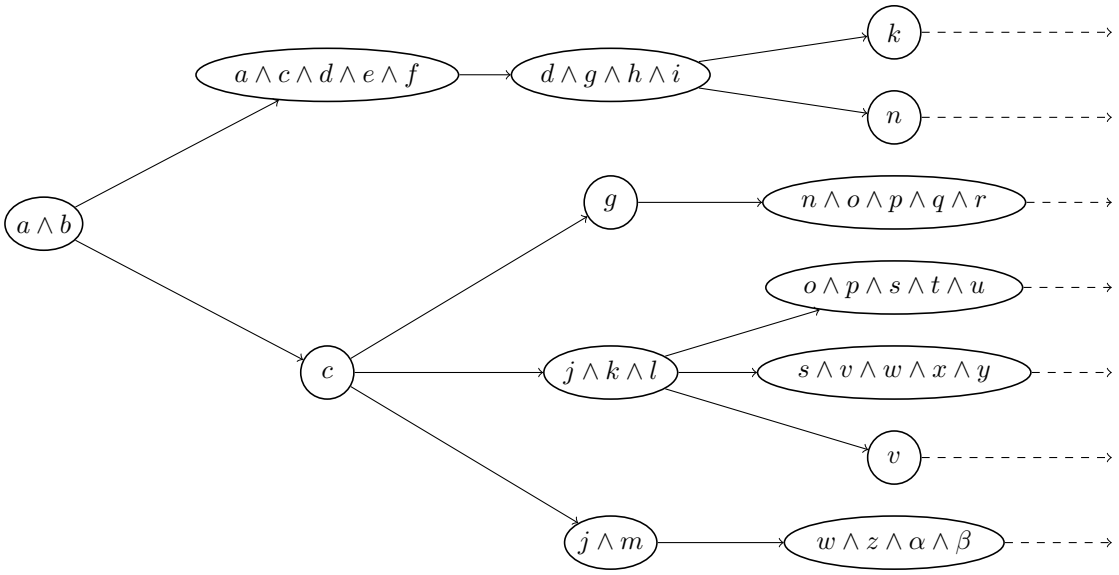

Figure 4: An example tree of rules with a maximum branching factor of 3. Each set of input tokens causes at least 1 and at most 3 rules to fire. Each rule has between 1 and 5 antecedents and consequents and shares at most $\lceil n/2 \rceil$ antecedent tokens with other rules, where n is the number of antecedent tokens of a given rule.

Our initial experimentation was limited to testing only trees of rules with a depth of 8 and branching factor of 3. Having shown that the system is able to operate successfully, a full investigation of these parameters can be undertaken in order to understand the effect that they have on network operation and capacity. Further to this, experimenting with these parameters will also allow a sensitivity analysis to be performed. This is important given the random nature of our experimental inputs, and the potentially large variance between sets of vectors that can occur for a number of reasons—such as the vector generation algorithm used.

In the experimentation we chose to use a vector length of 5120 and weight of 10, as we extrapolated from previous results to determine that this should provide ample capacity. All previous results used rules containing a single token in the antecedent and consequent, so they cannot necessarily be used to predict the capacity when storing rules with greater arities. As the number of tokens involved increases, so do the number of bits set when training a rule—leading to quicker CMM saturation, as we found in one of the experimental runs.

Further investigation into the effect of changing parameters such as the vector length and weight is required, in order to understand how the capacity of each of the ARCA sub-networks changes as well as the memory requirement for the complete system.

Finally, further work is required to understand how the network reacts when reaching saturation; in the experiments performed so far a number of the ARCA sub-networks remained unused, as the weight of the superimposed antecedent tokens was never equal to their value. It may be possible to allocate binary vectors using a different algorithm, in order to either spread the rules more evenly through the sub-networks, or possibly to guarantee that some of them are unnecessary.

References

[Austin *et al.*, 2012] James Austin, Stephen Hobson, Nathan Burles, and Simon OKeefe. A rule chaining architecture using a correlation matrix memory. *Artificial Neural Networks and Machine Learning–ICANN 2012*, pages 49–56, 2012.

[Austin, 1992] James Austin. Parallel distributed computation in vision. In *Neural Networks for Image Processing Applications, IEE Colloquium on*. IET, 1992.

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

[Baum *et al.*, 1988] Eric B Baum, John Moody, and Frank Wilczek. Internal representations for associative memory. *Biological Cybernetics*, 59(4):217–228, 1988.

[Hobson and Austin, 2009] Stephen Hobson and Jim Austin. Improved storage capacity in correlation matrix memories storing fixed weight codes. *Artificial Neural Networks–ICANN 2009*, pages 728–736, 2009.

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

[Ritter *et al.*, 1992] Helge Ritter, Thomas Martinetz, Klaus Schulten, Daniel Barsky, Marcus Tesch, and Ronald Kates. *Neural computation and self-organizing maps: an introduction*. Addison Wesley Longman Publishing Co., Inc., 1992.

[Willshaw *et al.*, 1969] David J Willshaw, O Peter Buneman, and Hugh Christopher Longuet-Higgins. Non-holographic associative memory. *Nature*, 1969.