Improving the Associative Rule Chaining Architecture

Nathan Burles, Simon O'Keefe, and James Austin

Advanced Computer Architectures Group,
Department of Computer Science,
University of York,
York, YO10 5GH, UK
`{nburles,sok,austin}@cs.york.ac.uk`
`http://www.cs.york.ac.uk`

Abstract. This paper describes improvements to the rule chaining architecture presented in [1]. The architecture uses distributed associative memories to allow the system to utilise memory efficiently, and superimposed distributed representations in order to reduce the time complexity of a tree search to $O(d)$, where d is the depth of the tree. This new work reduces the memory required by the architecture, and can also further reduce the time complexity.

Keywords: rule chaining, correlation matrix memory, associative memory, distributed representation, parallel distributed computation

1 Introduction

Rule chaining is a common problem in the field of artificial intelligence; searching a tree of rules to determine if there is a path from the starting state to the goal state. The Associative Rule Chaining Architecture (ARCA) [1] uses correlation matrix memories (CMMs)—a simple associative neural network [2]—to perform rule chaining. We present an improvement to the original ARCA architecture that reduces the memory required, and can also reduce the time complexity.

Rule chaining includes both forward and backward chaining. In this work we describe the use of forward chaining, working from the starting state towards the goal state, although there is no reason that backward chaining could not be used with this architecture.

In forward chaining, the search begins with an initial set of conditions that are known to be true. Each of the rules is then checked in turn, to find one for which the antecedents match these conditions. The consequents of that rule are then added to the current state, which is checked to decide if the goal has been found. If it has not, then the search continues by iterating—if no further rules are found to match then the search results in failure.

This is essentially a tree search, and so classical algorithms such as depth-first search are commonly used. The time complexity of such an algorithm is $O(b^d)$, where b is the branching factor and d is the depth of the tree. Reducing this to $O(d)$ therefore represents a potentially significant improvement.

1.1 Correlation Matrix Memories (CMMs)

The CMM is a simple, fully connected, associative neural network consisting of a single layer of weights. Despite their simplicity, associative networks are still an active area of research (e.g. [3,4]). In this work we use a sub-class of CMMs, where these weights are restricted to binary values, known as binary CMMs [5].

Binary CMMs use simple Hebbian learning [6]. Learning to associate pairs of binary vectors is thus an efficient operation, requiring only local updates to the CMM. This learning 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, n is the number of training pairs, and $\bigvee$ indicates the logical OR of binary vectors or matrices.

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

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

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

It is possible to greatly optimise this recall operation, using the fact that the input vector contains only binary components. For the j^{th} bit in the output, the result of a matrix multiplication is the vector dot product of the transposed vector $\mathbf{x}^T$ and the j^{th} column of matrix $\mathbf{M}$. In turn the 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 $\mathbf{M}$. Given the binary nature of $\mathbf{x}$ it is clear that this dot product is equal to the sum of all values $\mathbf{M}_{j,i}$ where $\mathbf{x}_i = 1$, formalised in Equation 3.

$$\mathbf{y}_j = f(\textstyle\sum_{i(\mathbf{x}_i=1)} \mathbf{M}_{j,i}) \tag{3}$$

There are various options as to which threshold function, f, may be applied during recall. The choice of function depends on the application, and on the data representation used. ARCA uses superposition of vectors, so the selection is limited to Willshaw thresholding, where any output bit with a value at least equal to the (fixed) trained input weight is set to one [5].

1.2 Associative Rule Chaining

The Associative Rule Chaining Architecture stores multiple states superimposed in a single vector using a distributed representation [7], which also helps to provide more efficient memory use and a greater tolerance to faults than a local representation [8]. For example, the superposition of two vectors {`0 0 1 0 0`} and {`1 0 0 0 0`} is the vector {`1 0 1 0 0`}.

ARCA performs rule chaining using superimposed representations, hence reducing the time complexity of a tree search. The main challenge overcome by the architecture is to maintain the separation of each superimposed state throughout the search, without needing to separate out the distributed patterns or revert to a local representation.

To solve this challenge, each rule is assigned a unique “rule vector” which exists in a separate vector space to those used for the antecedent and consequent tokens. ARCA stores the antecedents and consequents of rules in two separate CMMs, connected by the rule vector [1], described further in Section 2.

2 Improving the ARCA Architecture

In the original architecture, two CMMs are used to separate the antecedents and consequents of rules. When storing a rule, for example $a \rightarrow b$, a unique “rule vector” must be generated. This is essentially a label for the rule, but can be considered as our example rule becoming $a \rightarrow r_0 \rightarrow b$.

The first CMM is used to store the associations between the superimposed antecedents of a rule (a) and the assigned rule vector (r_0). This results in the rule firing if the tokens in the head of each rule are contained within a presented input, i.e. $a \rightarrow r_0$.

When training the second CMM, a slightly more complex method is required. Initially, a tensor product (TP) is formed between the superimposed consequents of a rule (b) and the rule vector (r_0); this TP is “flattened” in row-major order to form a vector ($b : r_0$). The associations between the rule vector and this TP are then stored in the second CMM. This means that when a rule fires from the antecedent CMM, the consequent CMM will produce a TP containing the output tokens bound to the rule that caused them to fire, i.e. $r_0 \rightarrow (b : r_0)$. These tokens can then be added to the current state in order to iterate.

2.1 Using a Single CMM

In ARCA the separation of superimposed states during the recall process is actually performed by the rule vector, rather than through the use of two CMMs. We propose that this is unnecessary and we can use a single CMM mapping directly from the antecedents to the consequents, reducing both the memory required to store the rules and the time required to perform a recall operation.

To train this reduced ARCA requires a similar operation as originally used when training the second CMM. Every rule is still assigned a unique rule vector, which is used to form a TP with the superimposed consequents of the rule ($b : r_0$). The single CMM is now trained using the superimposed antecedents of the rule as an input and this TP as an output, i.e. $a \rightarrow (b : r_0)$. Upon presentation of an input containing the tokens in the head of a rule, the CMM will produce a TP containing the output tokens bound to the rule that caused them to fire.

2.2 Recall

Fig. 1 shows a recall process performed on the reduced ARCA. To initialise the recall, an input state TP_{in} is created by forming the TP of any initial tokens

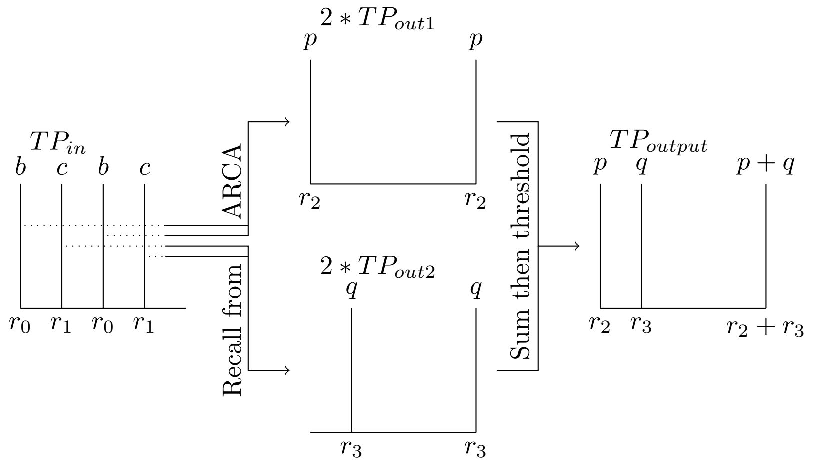


Fig. 1. A visualisation of the recall process within the reduced ARCA. The tensor products (TPs) contain different token vectors bound to rule vectors. Each column is labelled at the top with the tokens contained within the column. The position of each column is defined by the positions of the bits set to one in the rule vector to which the token is bound, labelled at the base of the column. The remainder of the TP consists solely of zeros. The vector weight is 2, and hence each column appears twice.

with a rule vector. In the diagram we assume that the vector weight is 2, and we initialise the search with two separate, but superimposed, inputs—$b : r_0$ and $c : r_1$.

The recall process can now begin, by recalling each column of TP_{in} from the CMM in turn. The result of each column recall is an entire TP of equal dimensions to the original input (TP_{outx}) containing the consequents of a rule, bound to the rule that fired them. In our diagram each of these output TPs is recalled twice, once for each column in TP_{in}.

We need to combine these to form a single TP to allow the system to iterate. As can be seen in the diagram, any antecedents will appear in TP_{in} a number of times equal to the weight of a rule vector. Thus, the consequents will appear in the same number of TP_{out}s, bound to the rule that caused them to fire. When these are summed, the weight of a rule vector can therefore be used as a threshold to obtain the final output—a single binary TP, TP_{output}.

Before the system iterates, we need to check whether the search has completed. Firstly we can check whether the search should continue. If TP_{output} consists solely of zeros, then no rules have been matched and hence the search is completed without finding a goal state.

If TP_{output} is not empty, then we must check whether a goal state has been reached. This is achieved by treating TP_{output} as a CMM. The superposition of the goal tokens is used as an input to this CMM, and the threshold set to the combined weight of these superimposed goal tokens. If the resulting binary vector contains a rule vector, then this indicates that this rule vector was bound to the goal token and we can conclude that the goal state has been reached.

2.3 Time Complexity of the reduced ARCA

We have previously shown [1] that ARCA is able to search multiple branches of a tree in parallel, while maintaining separation between them. This reduces the time complexity of a search to $O(d)$, where d is the depth of the tree. Contrasted with a depth-first approach with a time complexity of $O(b^d)$, where b is the branching factor, this is a notable improvement.

In order to compare the worst case time complexity of the original and the reduced ARCA, we now perform a more detailed analysis. When binary matrices and vectors are stored sparsely, the time complexity of a simple CMM recall operation depends on the weight of the input vector $w_{\mathbf{x}}$ and the number of bits set in each row of the matrix $w_{\mathbf{M}[i]}$. In the worst case, $w_{\mathbf{M}[i]}$ will be equal to the length of the output vector $l_{\mathbf{y}}$.

Using Equation 3, applied to all columns, the time complexity is found as the time to sum each row $\mathbf{M}[i]$ where $\mathbf{x}_i = 1$, plus a linear scan through the output vector to apply the final threshold. This is given in Equation 4.

$$\mathrm{TC}_{\mathrm{recall}} = w_{\mathbf{x}} l_{\mathbf{y}} + l_{\mathbf{y}} \tag{4}$$

In applying Equation 4 to the ARCA, we must consider the lengths and weights of vectors used to represent tokens and rules as potentially different, resulting in four terms: $l_{\mathbf{t}}$, $w_{\mathbf{t}}$, $l_{\mathbf{r}}$, and $w_{\mathbf{r}}$—representing the lengths and weights of tokens and rules respectively.

It is also important to remember that this equation calculates the time complexity of the recall of a single vector, where in ARCA every column of an input TP is recalled in turn. As both the original and the reduced ARCA operate in the same fashion, this multiplier can be ignored for the purpose of this comparison.

The worst case time complexity of recalling a single vector from the original ARCA and the reduced ARCA can now be derived to find Equations 5 and 6 respectively.

$$\begin{aligned}\mathrm{TC}_{\mathrm{original}} &= \mathrm{CMM1} + \mathrm{CMM2} \\ &= (w_{\mathbf{t}} l_{\mathbf{r}} + l_{\mathbf{r}}) + (w_{\mathbf{r}} l_{\mathbf{t}} l_{\mathbf{r}} + l_{\mathbf{t}} l_{\mathbf{r}}) \\ &= l_{\mathbf{r}}(w_{\mathbf{t}} + 1 + l_{\mathbf{t}}(w_{\mathbf{r}} + 1))\end{aligned} \tag{5}$$

$$\begin{aligned}\mathrm{TC}_{\mathrm{reduced}} &= w_{\mathbf{t}} l_{\mathbf{t}} l_{\mathbf{r}} + l_{\mathbf{t}} l_{\mathbf{r}} \\ &= l_{\mathbf{r}} l_{\mathbf{t}} (w_{\mathbf{t}} + 1)\end{aligned} \tag{6}$$

In order to find the relative parameters for which the reduced ARCA becomes more efficient than the original ARCA we equate 5 and 6 and simplify as far as possible, as in Equation 7.

$$\begin{aligned}l_{\mathbf{r}}(w_{\mathbf{t}} + 1 + l_{\mathbf{t}}(w_{\mathbf{r}} + 1)) &= l_{\mathbf{r}} l_{\mathbf{t}} (w_{\mathbf{t}} + 1) \\ 1 + l_{\mathbf{t}} w_{\mathbf{r}} &= w_{\mathbf{t}}(l_{\mathbf{t}} - 1)\end{aligned} \tag{7}$$

This equation shows that in the worst case, if the weight of both rule and token vectors is equal, that the original and the reduced ARCA will perform essentially equally. If the weight of rule vectors is greater than that of token vectors, then the reduced ARCA will outperform the original.

In reality, the worst case is not possible—if all of the rows of the matrix were completely full, then it would be impossible to successfully recall any vector that was originally stored. Rather than attempting to time a comparison, as this is unlikely to provide accurate or fair results, we instead compare the memory requirements experimentally.

2.4 Comparison of Memory Requirements

The time complexity of a recall operation is dependent on the number of bits set in each row of the matrix. As such, a comparison of the memory required by the original and the reduced ARCA also provides a good indication of any improvement in the time complexity.

In order to compare the memory required, both variants of ARCA have been applied to the same randomly generated problems. For each experiment a tree of rules was generated with a given depth d and maximum branching factor b, using the same procedure as detailed in previous work [1]. These rules were then learned by both systems, and rule chaining was performed on them.

In all experiments, we fixed the vector weight for both rules and tokens to be 4. This value results in sparse vectors over the range of vector lengths investigated, and should provide good performance in the CMMs [9].

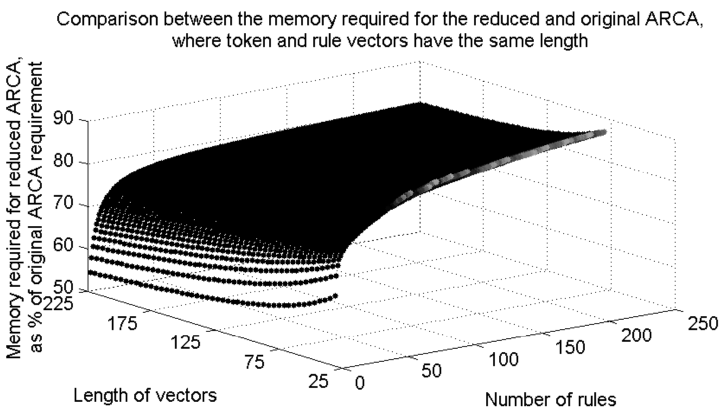
Given our analysis of the time complexity, using the same weight in both types of vector would also be expected to result in very similar memory requirements in both systems, and so any deviation from this will indicate a difference between the worst case and expected times.

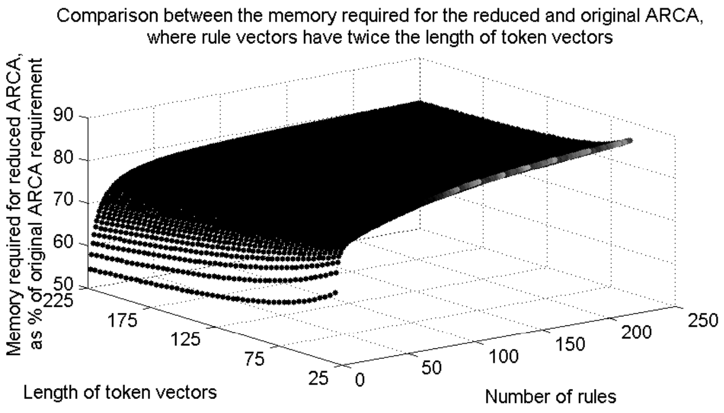
The experiments have been performed over a range of values for d, b, and the vector length. In order to further test our time complexity analysis we also varied the token vector and rule vector lengths independently, on the expectation that the comparative memory requirement would not vary.

The graphs in Fig. 2 are 3D scatter plots showing the memory requirement of the reduced ARCA as a percentage of the memory required of the original ARCA. The results clearly demonstrate that varying the relative token and rule lengths has very little effect on the comparative memory requirement.

It is also clear that the memory required by the reduced ARCA tends towards around 80% of that required by the original ARCA, which implies that the time complexity in the expected case is likely to be similarly improved.

Where the colour of a point is black, then the recall success rate for both architectures was at least 99%. As the colour of a point tends towards white (for the shortest vector lengths), it indicates that the reduced ARCA continued to achieve at least 99% recall success, but the original ARCA had a lower success rate. In no cases was the recall success rate of the original ARCA higher than that of the reduced ARCA, and so it can be concluded that the reduced ARCA improves the storage capacity of the network, even while reducing the memory





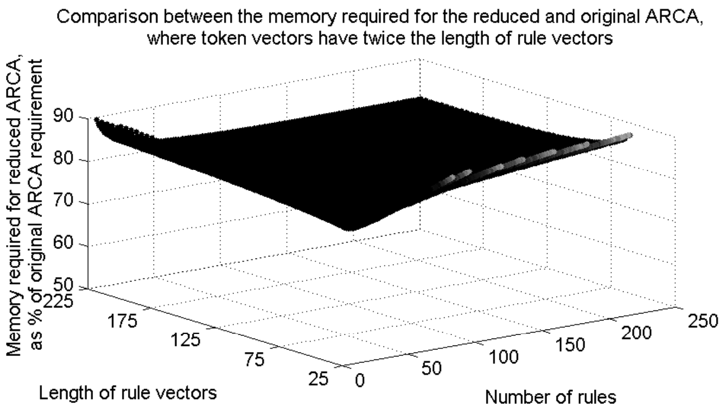


Fig. 2. 3D scatter plots showing the memory requirement of the reduced ARCA as a percentage of the memory requirement of the original ARCA. In the top plot, the length of token and rule vectors are equal. In the middle plot, the length of token vectors is double that of rule vectors. In the bottom plot, the length of rule vectors is double that of token vectors. For all points, the recall success rate for the reduced ARCA was at least 99%. As the colour tends from black to white, the recall success rate for the original ARCA moves from $\geq$99% towards 0%.

requirement. We chose to show the points at which recall success was at least 99%, as the nature of neural networks is that they may introduce a small amount of uncertainty. This cut-off, however, could be selected at any point—including 100%—with the trade-off that higher accuracy results in lower network capacity.

3 Conclusions and Further Work

This paper has described further improvements to the ARCA, with a simplified architecture which will aid in understanding, as well as reducing the memory requirements, the execution time, and improving the capacity.

Our analysis indicated that the performance of the original and the reduced architectures should be similar, if the token and rule vector weights are chosen to be equal. Our experimentation, on the other hand, clearly demonstrated that the reduced ARCA offers a 20% improvement over the original. This is due to the limitations of the worst case complexity, as previously explained. In order to improve the analysis, further work is required to create a better model of the interactions between vector pairs stored in a CMM. With such a model, the expected case could be analysed, based additionally on the number of rules stored in the system. This analysis will improve the theoretical basis for the application of ARCA to real rule-chaining problems.

References

1. Austin, J., Hobson, S., Burles, N., O'Keefe, S.: A Rule Chaining Architecture Using a Correlation Matrix Memory. Artificial Neural Networks and Machine Learning—ICANN 2012, 49–56 (2012)
2. Kohonen, T.: Correlation Matrix Memories. In: IEEE Transactions on Computers, pp. 353–359. IEEE Computer Society, Los Alamitos (1972)
3. Gorodnichy, D.O.: Associative Neural Networks as Means for Low-Resolution Video-Based Recognition. IJCNN 2005, 3093–3098 (2005)
4. Ju, Q., O'Keefe, S., Austin, J.: Binary Neural Network Based 3D Facial Feature Localization. IJCNN 2009, 1462–1469 (2009)
5. Willshaw, D.J., Buneman, O.P., Longuet-Higgins, H.C.: Non-holographic Associative Memory. Nature 222, 960-962 (1969)
6. Ritter, H., Martinetz, T., Schulten, K., Barsky, D., Tesch, M., Kates, R.: Neural Computation and Self-Organizing Maps: An Introduction. Addison Wesley, Redwood City (1992)
7. Austin, J.: Parallel Distributed Computation in Vision. In: IEE Colloquium on Neural Networks for Image Processing Applications, pp. 3/1–3/3. (1992)
8. Baum, E.B., Moody, J., Wilczek, F.: Internal Representations for Associative Memory. Biol. Cybernetics 59, 217–228 (1988)
9. Palm, G.: Neural Assemblies, an Alternative Approach to Artificial Intelligence. Chapter: On the Storage Capacity of Associative Memories, pp. 192–199. Springer, New York (1982)