Distributed Associative Memories for High Speed Symbolic Reasoning.

Dr. J Austin Advanced Computer Architecture Group Department of Computer Science
University of York, York YO1 5DD UK
austin@minster.york.ac.uk

ABSTRACT

This paper briefly introduces a novel symbolic reasoning system based upon distributed associative memories which are constructed from correlation matrix memories (CMM). The system is aimed at high speed rule based symbolic operations. It has the advantage of very fast rule matching without the long training times normally associated with neural network based symbolic manipulation systems In particular the network is able to perform partial matching on symbolic information at high speed. As such, the system is aimed at the practical use of neural networks in high speed reasoning systems. The paper describes the advantages and disadvantages of using CMM and shows how the approach overcomes those disadvantages. It then briefly describes a system incorporating CMM.

Key Words: Neural Networks, Associative Memory, superimposed coding, rule based systems.

1 Introduction.

There have been a number of neural models of reasoning (Sun 1994) which have mainly been aimed at explaining human cognitive processes. This paper briefly introduces a connectionist system for implementing reasoning that is aimed at the practical use of such systems. Thus, it considers the aspect of implementation efficiency on currently available computing platforms. One of the essential features of most forms of reasoning is the ability to match a piece of input information with a potentially very large amount of stored knowledge. For practical reasoning systems neural networks offer a very efficient matching process. If we consider a conventional multi-layer network, this can be trained to take a piece of input data (expressed as a pattern) and perform a match with the data (or knowledge) stored in the network. The result of the matching process will be an output label, expressed in the firing of the output neurons. As such, the neural network operates as a distributed associative memory (AM). In this mode, the neural network is able to recall an association using one pass through the network. The speed of this operation is much faster than found in conventional listing approaches to matching, where a piece of input data is compared to a list of potentially matching data. Through training, the neural network AM is able to form a very efficient and compact representation of the associations. In addition to their ability to perform fast match, the network is able to recall with inputs that are not exactly like the input but are similar to associations stored (in a pattern sense). These two features (partial match and speed) are exploited in the system described here to produce flexible and fast reasoning.

Although a neural network can be used in this way a major problem is the time required to train the network, especially if the input patterns are large. The approach taken here overcomes the long training times through the use of correlation matrix memories (CMM).

Although the approach described here could be used to implement other types of reasoning (i.e. for a frame based approach see Jackson 1994), we concentrate on rule based reasoning as this is the most common methodology used in industry. The main aim of the paper is to describe how rule matching is achieved. The process of rule chaining, backtracking etc. is not considered.

2 Capabilities of the method

The system is able to match logical rules, in the form;

A and B -> C

i.e. if A is true and B is true then this implies that C is true.

The system supports binary variables (i.e. true or false), variable names, pre-condition matching and instantiation processes. Both AND and OR logical connectives are supported in the preconditions of rules. Furthermore the system supports binding between A (the variable name) and 4 (the value) (expressed as A:4) The system supports rule conflicts, such as occurs when two (or more) rules match a set of pre-conditions. One of the most novel

features of the system is its capability of matching partial input data to pre-conditions of rules at high speed.

The next section describes the type of neural network used. This is followed by a description of how the rule and input data is presented to the system. Sections 4-8 describe how the data is coded to allow efficient rule matching. Section 9 describes how rule conflicts are dealt with. The rule consequences are handled is described in section 10. The partial match capability is reviewed in section 11. Finally, some brief notes are presented on the implementation of the system.

3 The neural network architecture used.

A multi-layer neural network can support the fast match of one pattern to elicit the recall of another. In the current system the input to the neural network consists of rule pre-conditions and the output of the network rule post-conditions. To train the neural network we cannot use conventional error back propagation methods as this takes a long time to train a set of patterns and it does not permit new patterns to be trained without training all other patterns at the same time (on-line learning). This is a problem for practical production systems as they will typically need new rules to be added without a significant training time. Furthermore, storage of all rules for re-training the network may not be possible.

The approach taken here is based on a two layer network architecture with (1) binary weights (2) Hebbian learning (3) L-max hidden layer encoding. In this form the system can be seen as two correlation matrix memories (CMM) and is an extension of the Advanced Distributed Associative Memory (ADAM, Austin (1987)). The network consists of two layers of fully connected neurons with binary (1 or 0) weights. The conventional sum of products activation function is used, but the system uses an L-max non-local output thresholding. The hidden layer pattern is not generated through error descent learning, but is chosen by the system. This can be done as no generalization is needed *between* the pattern associations stored; only between one pattern and the input. All the patterns used by the system (including the hidden layer patterns) are made up of a fixed number, L, of bits set to one in an array. The benefit of this is to ensure that the input to hidden pattern association can be reliably recalled using the L-max threshold method. L-max operates by selecting the L highest responding neurons in the hidden layer and sets their outputs to one, all other neurons are set to zero. Each neuron is trained using simple Hebbian learning, i.e. if the particular neuron input is set to one and output of the neuron is set to 1 then the weight connecting the input to the neuron is set to one. This is used for both the first and second layer neurons. A detailed description of ADAM can be found in Beale and Jackson (1990). The basic network architecture is given in Equations 1 to 5. In this, $\overline{I}$ is the

$$O_j = \sum_{alli} W_{ji}^{in} I_i \qquad 1$$

$$\overline{C} = f_1(\overline{O}) \qquad 2$$

$$P_k = \sum_{allj} W_j^{out} C_j \qquad 3$$

$$\overline{T} = f_2(\overline{P}) \qquad 4$$

$$f_2(x) = \begin{cases} \text{if } (x_i > T) & then \quad f(x_i) = 1 \\ else & f(x_i) = 0 \end{cases} \qquad 5$$

$$f_1 = \text{L-max } (\bar{y}) \qquad 6$$

array containing the pre-conditions, $\overline{C}$ is the array containing the separator and $\overline{T}$ is the array containing the post-conditions. The way these data items are represented in these arrays is explained in section 5. Equation 1 gives the activation function for the first layer of the network, where W is the array of binary weights, I is the input array of binary digits and O is the activation of the neurons. Equation 2 shows that the output is then thresholded using the function f_1, which applies the function shown in Equation 6, the L-max threshold discussed above. Equations 3 and 4 are for the second layer, which operates in the same way, but taking a different set of weights. The threshold function for this layer is given in equation 5. This is typically known as the Willshaw threshold (Willshaw and Longuit-Higgins 1969).

The storage capacity of this type of network has been discussed in detail in Austin (1987), Casasent and Telfer (1992), Willshaw, Buneman and Longuet-Higgins (1969) and Nadal, Toulouse (1990). The overall result is that if the bits set to one in the patterns used by the system are equal to Log_2 of the pattern array size, maximum storage is found. The result of this is that although the size of the network can be large, and thus the number of weights stored is large, the number of weights accessed in any one recall is small. This results in a very practical

implementation of the system.

Because the process of storage is distributed, there is always a small probability of failure to recall the correct association. This is the cost paid for the fast recall times achieved. However, the recall failure will always consist of extra bits being set to one in the output of the network, thus the output will be the correct result + noise. This type of error can be corrected through further processing.

The benefits of using L-max encoding are described in detail in Austin (1987), Casasent and Telfer (1992). The network allows on-line learning and training of each association by a single pass through the network. The use of binary weights allows very simple implementation in dedicated hardware (Kennedy and Austin 1994), although very good processing rates can be achieved using software simulations on conventional computers. In addition, the network training time scales well with the number of training patterns, thus the network can be used for real world sized problems.

All these features make the use of this type of network attractive. The next section describes how the rule pre-conditions are encoded into the network.

4 Rule presentation and basic system architecture.

The pre-conditions of a rule consist of a set of tokens joined with AND and OR terms. The present system deals with rules consisting of sum-of-products notation, i.e.

$A.B + \overline{B} . C + D \rightarrow ?$

Each 'AND' (product) term is dealt with individually, as though the 3 product terms were 3 rules. .e.

$A.B \rightarrow ?$ $\overline{B}.C \rightarrow ?$ $D \rightarrow ?$

Thus each rule will consist of a single product term in the pre-condition and a set logical consequences.

Pre-conditions -> post-conditions.

As described in section 2 the system uses a two layer network architecture. Training the network starts by placing the pre-conditions on the input to the first layer and the post-conditions on the output layer. The system then generates a hidden layer pattern which is used as the input values for the inputs to the second layer neurons and the output values of the input layer neurons. Thus we have;

Pre-conditions -> separator -> post-conditions.

The hidden layer pattern is called the separator because its action is to ensure that all rules are correctly separated from each other. Once this is done the system uses the Hebbian learning method to train the two layers. This only takes one pass through the data.

The success of the training process is dependent on the correct selection of the hidden layer pattern. To be sure that all rules are kept apart from each other all the hidden layer patterns must be orthogonal with each other. These pattern codes can be generated off-line. However, for optimal storage (size of the network in relation to the number of rules trained) the separator patterns need to be chosen with reference to the rules trained in the network. If this is required, more optimal methods can be used (Brown, Austin 1992). However, all these methods require iterative training, which can take time to store new rules. The simplest is the test-and-train method described in Austin and Turner (1995). This can be shown to generate a near optimal set of orthogonal codes (i.e. it is a good heuristic method).

5 Input representation.

Each input value and variable name in the queries presented to the system is converted to a token, which is a binary pattern with a fixed number of bits set to one. The system that maps the input symbols to tokens is a simple look-up table in a pre-processor, termed the lexical token converter.

The system deals with binary variables by assigning a special boolean truth value to the variable, TRUE or FALSE. Thus $\overline{A}$ is A:FALSE. The system reserves binary patterns to represent TRUE and FALSE. The notation ':' is used to represent a binding, as assignment '=' implies different semantics. The system deals with variables in the same way, by binding the variable to the variable name. For example, AGE=4 is written as the binding

AGE:4, as age has the associated value 4. If we used the notation '=' then we are saying that "AGE is replaced by 4" which has the incorrect meaning.

The issue of binding one token with another, as in AGE:4, has been difficult in neural network systems. The discussion surrounding this will not be described here, but the approach taken appears to overcome a number of the problems. We use a method that is similar to Smolenski (1989) but instead of using continuous tensor products, we use binary tensors, an approach discounted by Smolensky for no obvious reason.

In the present system the aim is to bind two tokens together, so that they do not get confused with the representation used when the conjuction of tokens is implemented (see later). A binding of A:4 is represented as a rank-two tensor of the tokens representing A and 4. This is shown below, which shows 3 bindings for the tokens in the precondition of a rule A:4 . B:TRUE . C:FALSE, where (a) represents the binding A:4, (b) for B:TRUE and

$$A = \begin{bmatrix}1&1&0&0&0\end{bmatrix} \qquad B = \begin{bmatrix}1&0&0&0&1\end{bmatrix} \qquad C = \begin{bmatrix}0&1&0&1&0\end{bmatrix}$$

$$TRUE = \begin{bmatrix}0&1&0&0&1\end{bmatrix} \qquad FALSE = \begin{bmatrix}1&0&1&0&0\end{bmatrix} \qquad 4 = \begin{bmatrix}0&0&1&1&0\end{bmatrix}$$

$$A{:}4 = \begin{bmatrix}1\\1\\0\\0\\0\end{bmatrix}\begin{bmatrix}0&0&1&1&0\end{bmatrix} = \begin{bmatrix}0&0&1&1&0\\0&0&1&1&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\end{bmatrix} \qquad a$$

$$B{:}\text{TRUE} = \begin{bmatrix}0&1&0&0&1\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&1&0&0&1\end{bmatrix} \qquad b$$

$$C{:}\text{FALSE} = \begin{bmatrix}0&0&0&0&0\\1&0&1&0&0\\0&0&0&0&0\\1&0&1&0&0\\0&0&0&0&0\end{bmatrix} \qquad c$$

(c) for C:FALSE.

The tensors can be presented to the first layer of the network as the first stage of the matching process (as *I* in equation 1) as a 1D vector formed from the concatenation of the tensors a, b and c. The 2D representation of the concatenated tensors is given below.

$$Input = \begin{matrix}0&0&1&1&0&\;&0&1&0&0&1&\;&0&0&0&0&0\\0&0&1&1&0&\;&0&0&0&0&0&\;&1&0&1&0&0\\0&0&0&0&0&\;&0&0&0&0&0&\;&0&0&0&0&0\\0&0&0&0&0&\;&0&0&0&0&0&\;&1&0&1&0&0\\0&0&0&0&0&\;&0&1&0&0&1&\;&0&0&0&0&0\end{matrix}$$

In practice, all variables that occur un-bound in the pre-condition of a rule are converted to a bound variable. This is needed because of the fixed input size to the network. The approach taken here is to bind a NULL token to the un-bound variables.

6 Training the network.

The network is trained by placing the input (I) onto the network and generating a separator (C) for the input. The weights are altered by a simple and fast learning rule which is based on the Hebbian learning method. This is given in equation 7, where $\boldsymbol{W}$ is the weight array in the first layer, and $\cup$ represents a logical OR.

$$\boldsymbol{W}^{in}_{new} = (\bar{C}^T \times \bar{I}) \cup \boldsymbol{W}^{in}_{old} \qquad 7$$

7 Rule pre-condition evaluation

The AND logical connective between the tensors is achieved through the action of the network and not through explicit manipulation of the input bindings. Smolensky used higher order tensors to form the AND logical connectives between the bindings and, although effective, these scaled badly with problem size.

In our approach the AND logical connective is achieved by thresholding the output of the network at a fixed level. In the example given above each token consists of N bits set to one (where N = 2 in the examples). This results in each tensor (e.g. A:4) having N^2 bits set, and a total of approximately $3 * N^2$ set bits input to the net (this number can be smaller due to tokens sharing bit positions in the input). When this input is presented to the network, each neuron in the hidden layer that should be set to one will have an activation of $3 * N^2$ because all the input bits that match the stored pattern will respond. A threshold set to $3 * N^2$ will ensure that the output from the first layer of neurons is the separator pattern for the matched rule. All other neurons will fire at a level based on the random correlation of the input pattern to the stored patterns.

This approach works well for rules with a fixed number of bindings, or a fixed arity (where arity is the number of bindings plus the number of unbound items present in the pre-condition of the rule, after the rule has been separated in to a set of rules containing only a single product term). However, if the arity of rules that are trained into the network vary then the network can generate false matches. To illustrate this problem consider the following rules trained into a network:

(i) A:TRUE . B:TRUE -> S_1

(ii) A:TRUE -> S_2

Where S_N is the Nth separator token. The rule (i) has an arity of 2; rule (ii) has an arity of 1.

Then the network is presented with;

A:TRUE -> ?

which asks if there is a binding that matches A:TRUE. The network will respond with an accumulation in the register that stores the $\overline{C}$ vector, this is then thresholded using the function $f()$ (see equations 4 and 5). It is not clear what threshold should chosen to allow rule (i) to match but not rule (ii). If it were chosen equal the number of bits set to one in the binding A:TRUE, then both rules would fire. This would be exhibited by both S_1 and S_2 simultaneously appearing in the hidden layer, which is incorrect, and the network is unable to separate the two rules. To overcome this, the network is extended by adding a new set of inputs and associated set of weights to the network for each arity of rule. So that the equations that define the first layer are now:

$$O^n_j = \sum_{all i} W^{in(n)}_{ji} I^n_i \qquad 7$$

$$\bar{P} = \sum_{all\ n} \bar{O}^n \qquad 8$$

where n specifies the input to arity 'network' n. The outputs from each arity network are summed separately as shown in Equation 8, then the threshold function in Equation 5 can then be applied to the outputs.

In the implementation, a pre-processor determines the arity of the input rule and feeds it to the correct arity network. The threshold used at the output determined according to the rule arity. i.e for rules of arity A, the threshold, T, is set to N * A, where N is the number of bits set in each binding.

8 Allowing commutativity of input bindings

Although sufficient to perform rule matching, the representation cannot cope with commutativity of the input arguments (i.e bindings). A network trained on the pre-conditions given above would not recall the correct rule if

the order of presentation of arguments was changed.

This is a severe limitation of this and other connectionist implementations. A typical suggestion is that the tokens should be ordered prior to their application to the network. Unfortunately, in the case where variables are missing (due to insufficient information) this is not possible. An alternative is to use unary encoding, where each input value has its a single input. While this approach works, it does not scale well.

The approach taken here is to use a superimposed coding. This has been popularly used in database systems for forming compact keys (Knuth 1973, Sacks-Davis, K Ramamohanaro 1983). The approach is simple and consists of superimposing all the bindings together as shown in the example below. for A:4 . B:TRUE . C:FALSE, using the outputs given in equations a, b and c. This superimposed binding (SIB) removes all order from the input

$$Input_{SIB} = \bigcup_{all\ n} I_n$$

$$Input_{SIB} = \begin{bmatrix} 0&0&1&1&0 \\ 0&0&1&1&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} \cup \begin{bmatrix} 0&1&0&0&1 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&1&0&0&1 \end{bmatrix} \cup \begin{bmatrix} 0&0&0&0&0 \\ 1&0&1&0&0 \\ 0&0&0&0&0 \\ 1&0&1&0&0 \\ 0&0&0&0&0 \end{bmatrix} = \begin{bmatrix} 0&1&1&1&1 \\ 1&0&1&1&0 \\ 0&0&0&0&0 \\ 1&0&1&0&0 \\ 0&1&0&0&1 \end{bmatrix}$$

bindings, thus allowing commutativity of the matching process. This input is presented to the network. It is important to note that (1) the input to the network is now smaller and more dense. To preserve the storage capacity of the network the binding arrays must be larger; (2) Because the bindings are superimposed (logically OR'ed together) there is a small probability of error due to interference between the bindings; (3) It allows the size of the input query to the network to be independent of the number of variable names that the input contains.

The use of the SIB coding method explains why the tensor product method of encoding the bindings has been used. By forming tensors, the binding between two tokens does not get lost when SIB coding is used. If the two tokens were not bound together, when superimposed, individual pairs of tokens would be confused.

The following shows the logical stages in the complete input system.

Rule input:

A:TRUE . B:TRUE + C:4 . D:TRUE $\longrightarrow$ X

Separate up into two component rules:

A:TRUE . B:TRUE $\longrightarrow$ X
C:4 . D:TRUE $\longrightarrow$ X

Convert lexical items to binary patterns (tokenize) shown by lower case:

a:true . b:true $\longrightarrow x$
c:4 . d:true $\longrightarrow x$

Bind tokens to form tensors:

a:true . b:true $\longrightarrow x$
c:4 . d:true $\longrightarrow$ x

Assign separator tokens:

a:true . b:true $\longrightarrow S_1 \longrightarrow x$
c:4 . d:true $\longrightarrow S_2 \longrightarrow x$

Superimpose to form SIBs:

(a:true $\cup$ b:true) $\longrightarrow S_1 \longrightarrow x$
(c:4 $\cup$ d:true) $\longrightarrow S_2 \longrightarrow x$

Train into arity 2 network:

(a:true $\cup$ b:true) x i + $W_{in}{}^2 = W_{in}{}^2$
(c:4 $\cup$ d:true) x j + $\overline{W}_{in}{}^2 = W_{in}{}^2$

$$i \times W_{out} = W_{out}$$
$$j \times W_{out} = W_{out}$$

When the input data is presented, the rules are decomposed to produce one product in each pre-condition and the arity of the rule noted. The variable/value bindings are then formed and the superimposed binding is created. This is then sent to the correct arity network inputs for training or testing. In training, the separator code generated and fed to the hidden layer, where the separator code is trained against the post-conditions. During testing, the stages used to prepare the input to the network are identical. The appropriate threshold (see section 7) applied for the recovery of the separator.

9 Separator recovery in the case of rule conflicts.

If an input matches a number of rules, G, stored in the neural network then the network will output a number of separators, G, all superimposed on each other. These binary patterns must be individually identified prior to their presentation to the next layer of the network. To achieve this a list of valid separators is held and used to identify the individual separator codes. The efficient implementation of this list is vital for the rapid operation of the system. Because no partial match is required in this stage of the memory, the separator list and match engine can be implemented in standard content addressable memory.

10 Output processing.

The output from the system is generated through the second layer of the neural network. Rule consequences supported by the system include both single and multiple bindings. An example of a multiple binding output is;

pre-condition -> A:4 . B:3 . C:TRUE

This indicates that the pre-condition implies A:4 and B:3 and C:TRUE.

The output of the second layer of the net is in the form of concatenated pairs of tokens. Neither the tensor product or SIB representation is needed for the output as commutativity in the output tokens is meaningless.

To support a set of R bindings in the post-condition of a rule, it is necessary to allow R binding fields, or groups of outputs from the second layer of the network. This sets a strict limit on the number of bindings that can be represented for any given implementation. However, it is possible to support more than R output bindings by using a chaining method. In this method the Rth binding output from the network is either a separator token that can be used to re-input to the network for another set of consequences, or a blank output (indicating no more bindings need to be recalled).

The threshold for the second layer output from the network is set to the number of bits present in the separator token. Once the outputs have been obtained, they are passed to the LTC to convert them back to labels used by the external interface. Alternatively they can be fed back to the input to support rule chaining (not considered here).

11 Partial match capability.

The basic system described here allows a set of preconditions expressed as a set of bindings to be applied to the network so that all the rules and associated post-conditions can be recalled. One of the main aims of the work has been to allow a partial match capability. In this work, partial match relates to the ability to recall rules when only given a subset of the correct bindings. To illustrate the operation, consider a system with the following rules stored;

(i) A:1 . B:3 . C:4 -> S_1 -> X

(ii) A:1 . B:5 . D:2 -> S_2 -> Y

(iii) A:1 . B:6 -> S_3 -> Z

A partial match capability would allow rule (i) to fire when the network was only given the pre-conditions (A:1 . B:3). This can be achieved in the network by (1) applying the pre-conditions to the arity 3 network input and (2) thresholding the output (the arity 3 network output) at a level equal to the number of bits in the input SIB. This is like asking “are there any 3 arity rules that match 2 or more bindings”. In this case there is and the separator (S_1)

for rule (i) would be output from the network.

This capability can be used to find any sub-set of bindings that match the rules stored. The is a very fast and efficient operation, which only requires one pass through the network.

The partial match capability can also cope with more complex problems. Consider the input;

A:1 . B:3 . F:4 . B:6

The operation is now "find all the rules that match any 2 bindings in the set of 4 supplied bindings". Typically, this is a combinatorial problem, which requires a time-consuming search process. If this input were used to search the list of rules given above, both rule (i) and (ii) should match.

12 Speed of the approach.

The speed of operation of the method can be estimated in general. The analysis makes no account of the size of the network being used, apart from the fact that it takes one pass through the network to match a rule. A simple comparison shows that the great benefit of the approach is the ability to perform partial rule matches very efficiently.

Consider the problem of finding the exact matching rule to the one input. We denote Q as the number of bindings in the input. The process of finding an exact match is to input the Q tokens to the network with an arity Q. The separator output is then thresholded at a level J, where $J = N^2 \times Q$, where N is the number of bits set to one in a token. This operation takes approximately O(Q) time to compute for a given network.

A more complex operation is to ask the question, "give me all the rules that match T *or more* of any of the input bindings". To do this would involve inputting the Q tokens in the input to all networks with an arity of T or more, and using the threshold level $J = N^2 \times T$ for each separator output. The computational complexity for the network is approximately O(Q x F), where F is the number of arity inputs consulted.

It is often necessary to find the set of rules which "best" match the input set of bindings. In the example here, this would ask "give me the rule that *best* matches this set of the input bindings". Again the Q tokens would be presented to the network, but all arity inputs would be used in this case. The L-max threshold is used to recover the separator on each arity network. This threshold selects the L highest responding neurons and sets them to output 1, where L equals the number of bits set in one separator code. If more than L neurons are activated with the maximum value, then all those neurons are set to output 1. The complexity is O(Q x G), where G is the number of arity networks in use.

To all these timings, the time to recall the output (post-conditions), is assumed to be constant. Furthermore, the amount of time to access one arity network with one token is assumed to be a constant multiplier to each time calculation.

It is difficult to compare this performance to a conventional method. There a many ways of implementing these types of matches. For our initial work on this see Filer and Austin (1995). The most efficient alternative method we have found is the Superimposed Bit Slice method (SBS) (Sacks-Davis and Ramamohanaro 1983) which is similar to our approach. However, SBS cannot do partial matches of the type given here, it can only perform exact matches. Their scheme uses the same superimposed tokens (but without binding) which allows a commutative exact match in the same time as our method. Furthermore, they do not implement N bit encoding of the separators for each rule (records in their case). To do a partial match using the SBS method on (for example) A.B.C.D where *any* 2 tokens must match involves presenting all possible pairs to the system (i.e. A B, A C, A D, B C, B D and C D). Thus, partial matching for an input of Q tokens, where T tokens must match would take $O(C_T^Q x Q)$, where C_T^Q is the combination of T items from a set of Q items. For problems of any size, this is a major bottleneck.

For problems requiring an answer to the question "give me all the rules that match T *or more* of any of the input bindings" requires even more processing using SBS because a range of values of T must be used. The complexity for this task is

$$O\left(\sum_{i=T}^{Q}(C_i^Q \times Q)\right)$$

The time complexity for quires of the form “give me the rule that best matches this set of the input bindings”, is the same.

From this, the advantages of using the neural networks can clearly be seen. As an example, the time taken to match an input of 10 tokens, where any 5 tokens are permitted to match for a rule to be selected would take about 50 times longer using the SBS method compare to the approach described in this paper. If any 10 tokens are to be matched against a set of 20 input tokens would take 18475 times longer.

13 Implementation.

The implementation of the neural network in software is particularly simple as only binary weights are used. We are currently developing a special purpose chip that supports the binary multiply and accumulate process necessary in each layer of the neural network. The work is based on our implementation of the ADAM network (Kennedy, Austin 1994).

14 Conclusion.

This paper has described the basic rule matching engine. The system has applications in many areas of automated reasoning as well as being of interest to cognitive scientists. It has shown how a simple two layer neural network can be used to perform complex and fast rule matching. In particular it allows the fast matching of rules without worrying about the order of presentation of the pre-conditions.

15 References.

R Sun (1994_, Integrating Rules and Conenctionism for Robust Commonsense Reasoning, John Wiley and Sons.

T J Jackson (1994), Associative Neural Networks for Frame Based Reasoning, PhD Thesis, University of York.

J Austin, T J Stonham (1987), An Associative Memory for use in Image Recognition and Occlusion Analysis, Image and Vision Computing, Vol. 5, No. 4, P 251-261.

R Beale, T Jackson (1990), Neural Computing: an introduction, Adam Hilger.

D P Casasent and B A Telfer (1992), High Capacity Pattern Recognition Associative Processors, Neural Network, Vol. 5 No. 4. p687 - 698.

D J Willshaw, O P Buneman and H C Longuet-Higgins (1969), Non-Holographic Associative Memory, Nature 222, June 7, p960-962.

J P Nadal and G Toulouse (1990), Information Storage in Sparsely Coded Memory Nets, Networks, Vol 1.

J Kennedy and J Austin (1994), A Hardware Implementation of a Binary Neural Network, p178-185, Fourth International Conf. on Microelectronics for Neural Networks and Fuzzy Systems, IEEE Computer Press.

C P Dolan, P Smolensky (1989), Tensor product production system: a modular architecture and representation, Connection Science, No. 1, p 53-68.

D E Knuth (1973), The Art of Computer Programming, Vol 3, Addison Wesley.

R Sacks-Davis and K Ramamohanaro (1983), A Two Level Superimposed Coding Scheme for Partial Match Retrieval, Information Systems, p273-280.

M Brown and J Austin (1992), Optimum Selection of Class Vectors for ADAM, Proceedings of International Conference on Artificial Neural Networks, Brighton, UK.

J Austin and A Turner (1995), A simple method for selecting near orthogonal class patterns, In preparation, University of York.

R J H Filer and J Austin (1995), The suitability of Correlation matrix memory as an inference engine for rule-based reasoning, ASIB-93: Hybrid Problems, Hybrid Solutions, 1995.