A Connectionist Implementation of the ACT-R Production System

Christian Lebière & John R. Anderson

Department of Psychology

Carnegie-Mellon University

Pittsburgh, PA 15213

cl+@cmu.edu, ja0s+@andrew.cmu.edu

Abstract[1]

This paper describes a connectionist implementation of the ACT-R production system. Declarative knowledge is stored as chunks in separate associative memories for each type. Procedural knowledge consists of the pattern of connections between the type memories and a central memory holding the current goal. ACT-R concepts such as adaptive learning and activation-based retrieval and matching naturally map into connectionist concepts. The implementation also provides a more precise interpretation for issues in ACT-R such as time of memory retrieval and production firing, retrieval errors and partial matching. Finally, the implementation suggests limitations on production rule structure.

Introduction

Anderson (1993) describes a production system model of cognition called ACT-R. ACT-R, as its predecessor ACT* (Anderson, 1983) involves a distinction between declarative memory (defined by elements called chunks) and procedural memory (defined by productions), a goal structure for coordinating productions, an activation-based retrieval system, and a scheme for learning new productions. ACT-R differs from ACT* in that both its principles for activation computation and for conflict resolution have been explicitly guided by the rational analysis of cognition (Anderson, 1990). It also differs in that it is a fully implemented computer simulation.

As ACT*, ACT-R claims to be a theory of both the symbolic level of cognition and of a neural-like implementation of that symbolic level. While there are some things with clear neural interpretation, such as the activation-based computation, the claim of neural implementation is more promissory than real. We have now begun what would be involved in developing a detailed connectionist implementation of the ACT-R theory.

The system we describe here can be viewed as one of a number of new hybrid models that have been described which involve both connectionist and symbolic components. These systems have attempted to combine the modularity and generality of symbolic systems with the generalization capabilities and homogeneity of representation and learning of neural networks. There have been a number of attempts to specifically implement production systems in connectionist terms (e.g. Touretzky & Hinton, 1988; Dolan & Smolensky, 1989). These have basically started with connectionist principles and tried to derive symbolic capabilities. Our attempt is different in that we are starting with an existing symbolic system (with a great many correspondences established with behavioral data) and exploring how it might be implemented.

We do not expect that ACT-R will remain totally unchanged in this effort. We have already found some aspects of the ACT-R theory which are both unnecessary and

[1]This research was supported by ONR grant number N00014-90-J-1489.

difficult to translate. In general one might say that the ACT-R theory has too many degrees of freedom, and one consequence of this effort will be a reduced and more constrained version of that theory.

It is also the case that the ACT-R theory lacks an adequate theory of partial matching and so has difficulty in emulating certain aspects of human behavior, particularly those concerned with slips that involve errors of commission. Partial matching is something that connectionist networks do particularly well.

The system which we are building is tentatively called ACT-RN, for ACT-R in a Neural network. We will describe in this paper its current state of development. In summary, the motivations for developing ACT-RN are to increase the plausibility of ACT-R by showing how it might be mapped in detail onto a neural implementation, to explore how connectionist systems can achieve symbolic capabilities, to find further constraints on ACT-R, and to provide ACT-R with a theory of partial matching.

Declarative Memory

The fundamental distinction in ACT-R is between declarative and procedural knowledge. Declarative knowledge is represented by means of structures called chunks. A chunk consists of a unique identifier, together with a number of slots each containing a value, which can be either another chunk, or an external object or a list. Since the power and generality of list structures and Lisp functions might be excessive, and for the sake of simplicity and uniformity, in ACT-RN we will constrain the slot values to be chunks.

Each chunk is of a particular type. A type is defined by its name and list of slots. Whereas in ACT-R all chunks were stored in a common declarative memory, for a number of reasons we decided here to define a separate memory for each type.

The first considerations regard neural network capacity. It is widely believed that the capacity of associative memories grows linearly with their size. (Hopfield, 1982) If full connectivity is used, however, the number of connections grows with the square of the number of units. By breaking up declarative memory into type memories, we preserve capacity while considerably decreasing the number of connections necessary. Separate type memories can also learn better the representational structure of each chunk type without being perturbed by the other types. Finally, since various types have different numbers of slots and therefore different lengths, having separate memories for each type improves memory efficiency.

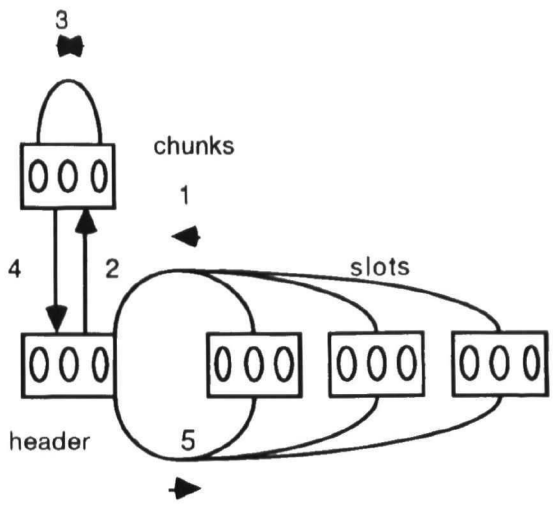

Figure 1: Type Memories

An associative memory for chunks should, given a chunk name or some slot values, retrieve the corresponding chunk. To implement associative memories, we use a simplified version of real-valued Hopfield networks (Hopfield, 1984). Each slot as well as the chunk name is represented by a pool of units. The unit pool for the chunk name is called the header. Instead of having complete connectivity between all pools, the slots are only connected to the header and vice versa. Therefore retrieval works not by energy minimization on a recurrent network but through a forward-backward mapping mechanism (Figure 1). First, the slot values are mapped to the header units to retrieve the chunk identifier which most closely matches these contents (1). Then, the header is mapped back to the slots to fill the remaining values (5). If the header is specified then step (1) is omitted.

To insure optimal retrieval, we found it necessary to "clean" the header. This can be achieved in a number of ways. One would be to

implement the header itself as an associative memory. We have chosen instead to connect the header to a pool of units in which each unit represents a chunk (2). The connections between the header and a particular unit are set to that unit's representation. By assembling these units in a winner-take-all network (3), the chunk with the representation closest to the retrieved header ultimately wins. That chunk's representation is then copied back to the header (4). We also use that mechanism to output to the user the name of the chunk which has been retrieved. A similar mechanism is described in (Dolan & Smolensky, 1989). The initial activation level of the winning chunk is related to the number of iterations needed to find a clear winner. This maps onto retrieval time in ACT-R.

Currently, only external events can modify or create memory chunks. This is a substantial difference from ACT-R where productions can directly change declarative memory. So far, we have not found this restriction to be seriously limitative. Every time a new chunk is created, a new representation appears in the header pool, and a new unit is initialized in the localist network with the proper connections. Hebbian learning is then used to add the correlation between header units and slot units to the connections between header and slots. If chunk representations are orthogonal, such one-time learning should be sufficient. (Hinton & Anderson, 1981) A special case of orthogonal representations which are particularly easy to generate are localist representations. If rather than using orthogonal representations we use random representations we will get interference between representations that decreases with the size of the representation.

The other possibility is to allow representations to be correlated to the extent they are similar. This would both increase interference and promote generalization. To learn this type of representation, an iterative supervised learning algorithm such as the Delta Rule (Rumelhart & McClelland, 1986) is necessary.

For those chunks which we regard as symbolic we use either random or orthogonal representations for their identifiers. For those chunks which we regard as analog we encode their similarity in the patterns of correlations among their identifiers. For instance, in the addition model described below, we have symbolic chunks representing the addition columns but we specify for the integers and the addition facts a representation encoding their magnitude. That allows the addition table to be represented compactly and to generalize well. We will refer to such representations as semantic.

Procedural Memory

ACT-R is a goal-oriented system. To implement this in ACT-RN we have created a central memory, which at all times contains the current goal (Figure 2), with connections to and from each type memory. With this system we can implement productions which retrieve information from a type memory and deposit it in central memory. Such a production might retrieve from an addition table the sum of two digits held in central memory. For example, given the goal of adding 2 and 3, a production would copy to the *addition-fact* type memory the chunks 2 and 3 in the proper slots, let the memory retrieve the sum (5) and then transfer that chunk to the appropriate goal slot.

This central memory system allows all

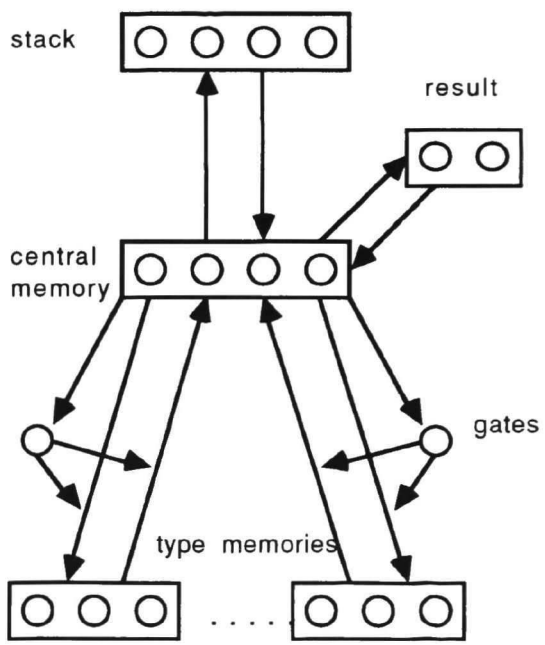

Figure 2: Procedural Memory

memory types to communicate with each other with a cost in connections that is linear in the number of types rather than the square of the number of types. These connections can be set by a production compiler and/or learned with a Hebbian learning method similar to the one for declarative knowledge, to record the patterns of transfer between central memory and type memories.

To provide a degree of control over production firing, we need a way to decide not only what gets transferred where, but also when. That is the role of the production evaluation and selection mechanism in ACT-R. In ACT-RN, that task is achieved by gating units. Each unit stands for a particular production and has incoming connections from central memory which reflect the constraints on the left-hand side of that production. For example, if goal slot S is required to have as value chunk C in production P, then the connections between S and the gating unit for P will be the representation for C, with an appropriate threshold. At each epoch, all the gating units are activated by the current state of central memory, and a winner-take-all competition selects the winner. Again, the relative activation of the gating units determines the number of iterations, which can be used as a measure of production selection time.

Connections between central and type memories describe the patterns of transfer to and from central memory. The winning gating unit is used to select which of these connections are used by that production. Each gating unit is restricted to a particular type memory, which is necessary to limit the number of gating connections for each production to a constant independent of the number of type memories. That leaves two fundamental production types:

I	II
p lookupfact	p loadnewgoal
goal	goal
fact	==>
==>	newgoal
goal'	

Type I looks up some fact and copies some slot value(s) (or header) back into the goal. Type II changes the goal to a new goal. That new goal is a chunk that is retrieved from one of the type memories. These severe limitations over the unlimited complexity of ACT-R productions do not appear to limit the expressive power of the language, but serve to make more explicit the cognitive steps involved.

Goal Stack

An important feature for the usability of ACT-R is its goal stack. Although such a feature could be left to the user to implement using a type memory, its importance and specificity warrants a special mechanism. This matches its special treatment in ACT-R.

The goal stack is implemented in ACT-RN by using a dedicated type memory. Both new goal G and parent goal PG are represented among the slots. The header representation is chosen at random. To push the new subgoal, both G and PG are copied in the appropriate slots, and the correlation is memorized. When G is to be popped, it is copied to the stack memory and PG is retrieved, then restored back to central memory. The association between G and PG is erased from the stack memory through explicit unlearning. Weight decay would be a less exact but perhaps more psychologically plausible technique to accomplish this.

We have made one modification to how ACT-R handles goals: We have added a mechanism to ACT-RN to return a result from a subgoal to its parent goal. We have long felt that the inability to do this led to rather awkward production rules in past ACT theories. In addition, this facility avoids a problem that would otherwise arise in ACT-RN: When the goal has been restored to its previous value, the production which initially pushed the subgoal would fire again and the system is caught in a loop. The appearance of the result value changes the goal and so naturally prevents the old production from firing.

That goal stack mechanism has been added in ACT-RN by introducing two commands. The **push** command specifies a slot of the parent goal in which the result returned by the subgoal is to be copied. The **pop** command in turn specifies a value (chunk) to be returned when the subgoal is popped. That value is copied to a result memory, which is then restored in the proper goal slot after popping. These commands modify the second production type given above to add the **push** command and introduce a third type for the **pop** command:

II'	III
p pushgoal	**p** popgoal
goal	goal
==>	==>
[**push** goalslot]	**pop** value
subgoal	

Application

In Table 1 is the ACT-RN code to solve the multicolumn addition problem. This production system was adapted for ACT-RN from the ACT-R production system published in Anderson (1993). It involves one significant representational change which is that two digit numbers like 12 are represented as a 1 in the tens column and a 2 in the units column rather than as an unanalyzed chunk. This change, which we think is a step in the direction of psychological plausibility, eliminated the need for two productions in the original set that extracted the ones digit from the multidigit number. On the other hand it requires a special production for carry when a nine is present. This model has therefore four productions instead of five.

In addition to the production rules we have given fragments of the code to set the system up to give a little of the flavor of what it is like to write code in ACT-RN. All the commands are, except for a few modifications, identical to ACT-R's. **WMEType** declares a type by specifying its name and list of slots. Types used here are *symbol* (todo), *number-integer*, *addition-fact* and *column*. **AddWM** adds a list of chunks to declarative memory. Each chunk is defined by its name, type, value for (some of) its slots, and possibly the **representation** option to specify which memory representation is to be used for that chunk. To keep the network small and improve generalization, we have used semantic rather than orthogonal or random representations for the integers and addition facts. This representation works because it reflects the semantics of the numbers and addition facts.

The goal is always the column to be processed. The production *add* performs the addition of the two numbers *=top* and *=bottom* by retrieving the relevant addition fact, then

Table 1: Multicolumn Addition Code

```
;; Type declarations
(WMEType column top bottom
         answer carry status next)
...
;; Working Memory definitions
(AddWM
  ...
 (two isa number-integer
         representation (+ + (-)))
  ...
 (twothree isa addition-fact
         addend1 two addend2 three
         sumtens zero sumunits five
         representation (+ + + + + (-)))
  ...
 (column1 isa column
         top two bottom three
         carry zero status todo
         next column2)...)
;; Productions
(p add                          (p cancel
=goal                           =goal
isa column                      isa column
top =top                        top =top
bottom =bottom                  top (- nine)
carry zero                      carry one
status todo                     status todo
=fact                           =fact
isa addition-fact               isa addition-fact
addend1 =top                    addend1 =top
addend2 =bottom                 addend2 one
sumtens =tens                   sumunits =units
sumunits =units                 ==>
==>                             =goal
=goal                           isa column
isa column                      top =units
answer =units                   carry zero)
carry =tens
status (- todo))                (p nine-carry
                                =goal
                                isa column
                                top nine
(p next                         bottom =bottom
=goal                           carry one
isa column                      status todo
status (- todo)                 ==>
carry =carry                    =goal
next =next                      isa column
==>                             top zero
=next                           answer =bottom
isa column                      carry one
carry =carry)                   status (- todo))
```

copying the answer, in the form of a units digit *=units* and a tens digit *=tens* , into the *answer* slot and the *carry* slot respectively, then marking the column as solved. The production *next* focuses on the next column by retrieving it from memory and copying the carry into it.

The production *add* has a condition requiring the carry to be *zero* before performing the addition. Should the carry be *one* , the production *cancel* will first add it to the top number *=top* by retrieving the sum of *=top* and *one* and copying it back and zeroing the carry, therefore enabling the *add* production to fire. The special case when the top number is *nine* and a carry is present has to be handled by the production *nine-carry* .

We also converted into ACT-RN the model for the Tower of Hanoi problem. The initial ACT-R solution (Anderson, 1993) had only three large productions where the processing complexity lay in complex matches involving list matching and negation clauses. In contrast, the ACT-RN model has to decompose these productions into basic cognitive steps, which result in a total of 17 productions. Preliminary results indicate that the model closely reflects the latency patterns of human subjects.

Conclusion

In this paper, we demonstrate how neural networks can be used to implement the ACT-R production system. Declarative knowledge is stored as chunks in a separate associative memory for each type.

To limit the number of connections necessary, we introduce a central memory which holds the current goal. Procedural memory consists of the pattern of connections between central memory and the type memories. Evaluation and selection of productions is done through a winner-take-all network of gating units activated from central memory.

We are encouraged by the fact that we have already been able to successfully convert into ACT-RN several ACT-R models such as the multicolumn addition example and the Tower of Hanoi problem. This is evidence that the system is succeeding in incorporating the symbolic power of a real production system. We plan to continue developing ACT-RN into a general modeling tool and see how well it can account for a wide range of psychological phenomena.

References

Anderson, J.R. 1983. *The Architecture of Cognition*. Cambridge, MA: Harvard University Press.

Anderson, J.R. 1990. *The Adaptive Character of Thought*. Hillsdale, NJ: Erlbaum.

Anderson, J.R. 1993. *The Rules of the Mind*. Hillsdale, NJ: Erlbaum.

Dolan, C.P. & Smolensky, P. 1989. Tensor Product Production System: a Modular Architecture and Representation. *Connection Science* (1): 53-68.

Hinton, G.E. & Anderson J.A. 1981. *Parallel Models of Associative Memory*. Hillsdale, NJ: Erlbaum.

Hopfield, J.J. 1982. Neural Networks and Physical Systems with Emergent Collective Computational Abilities. In *Proc. Natl. Acad. Sci. USA 79*, 2554-2558.

Hopfield, J.J. 1984. Neurons with Graded Response have Collective Computational Properties like those of Two-state Neurons. In *Proc. Natl. Acad. Sci. USA 81*, 3088-3092.

Rumelhart, D.E. & McClelland, J.L. 1986. *Parallel Distributed Processing.: Explorations in the Microstructure of Cognition*, Vol. 1. Cambridge, MA: MIT Press/Bradford Books.

Touretzky, D.S. & Hinton, G.E. 1988. A Distributed Connectionist Production System. *Cognitive Science* (12) 423-466.