BoltzCONS: Reconciling Connectionism with the Recursive Nature of Stacks and Trees

David S. Touretzky

Computer Science Department
Carnegie-Mellon University

Abstract

Stacks and trees are implemented as distributed activity patterns in a simulated neural network called BoltzCONS. The BoltzCONS architecture employs three ideas from connectionist symbol processing -- coarse coded distributed memories, pullout networks, and variable binding spaces, that first appeared together in Touretzky and Hinton's neural net production system interpreter. In BoltzCONS, a distributed memory is used to store triples of symbols that encode cons cells, the building blocks of linked lists. Stacks and trees can then be represented as list structures. A pullout network and several variable binding spaces provide the machinery for associative retrieval of cons cells, which is central to BoltzCONS' operation. Retrieval is performed via the Boltzmann Machine simulated annealing algorithm, with Hopfield's energy measure serving to assess the results. The network's ability to recognize shallow energy minima as failed retrievals makes it possible to traverse binary trees of unbounded depth without maintaining a control stack. The implications of this work for cognitive science and connectionism are discussed.

1. Introduction

This paper begins with an assumption: that recursive symbol structures have a place in connectionist theories of cognition. A recursive structure is one whose components may be structures of the same type. Trees are recursive because their branches are trees; stacks are recursive because their tails (that which remains when the first element is removed) are stacks. The recursiveness of a data structure is independent of any algorithm; a recursive data structure may be manipulated by a nonrecursive algorithm, and recursive algorithms may operate on nonrecursive data structures, such as integers. This paper is not concerned directly with the feasibility or utility of recursive *algorithms*. Instead it will focus on the two most common recursive data structures: stacks and trees.

There are many potential uses of stacks in an intelligent system. At the level of conscious problem solving behavior, stacks could be used to keep track of goals while focusing temporarily on subgoals. In dialog or in narrative reading, stacks might be used to track conversation topics in the presence of digressions, parenthetical remarks, or interruptions. In these two examples stack manipulation occurs at a conscious or mentally accessible level. If there turn out to be recursive mental procedures at a *sub*conscious level of processing, then their implementation may rely on an internal (i.e., inaccessible to introspection) control stack. But even without recursion, a control stack might be useful for saving contextual information when a mental procedure must invoke several levels of subprocedure.

Mental trees may be harder to justify than stacks. In conventional AI programs trees are used to represent many sorts of things, such as syntactic structures (parse trees), decision procedures (discrimination nets), and goal/subgoal hierarchies (AND/OR graphs). Whether trees have any reality in the brain, or any essential role in connectionist theories of mind, is another matter, and for now it is an open question. But if connectionist systems could not represent tree structures, that would place a serious *a priori* constraint on the plausibility of connectionist theories.

To demonstrate that connectionism is capable of accomodating recursive structures, this paper presents BoltzCONS, a Lisp-inspired Boltzmann machine (Fahlman et al., 1983; Ackley et al., 1985). The ability to perform

associative retrievals in BoltzCONS and other connectionist architectures makes them unlike conventional von Neumann computers.

2. Stacks and Trees in a Distributed Memory

We begin by considering how stacks and trees might be represented in a distributed memory. In the following section we go on consider how they can be manipulated. Figure 1 shows a cons cell representation of the stack (A B C D) in what is almost conventional Lisp notation; the difference is that each cons cell in the figure has associated with it a unique symbol, or tag.[1] The figure can thereby be encoded as a set of triples of form (**tag car cdr**), as shown below:

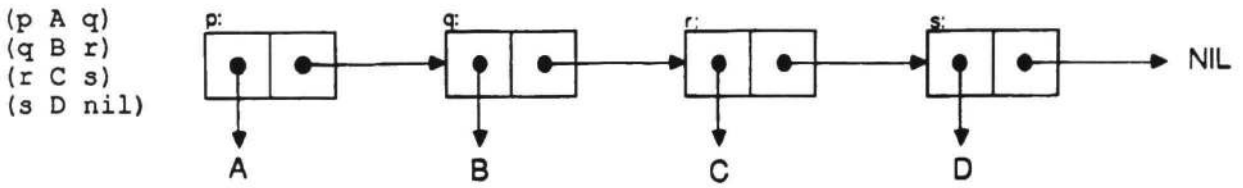

Figure 1: The stack (A B C D) represented as triples and as cons cells.

Trees can be encoded as triples in the same way. Figure 2a shows a binary tree, and figure 2b shows its cons cell representation. The translation from cons cells to triples is straightforward. But how can a connectionist network represent these triples in a computationally plausible way? We reject on efficiency grounds the extreme localist position that there be one unit for every possible triple, since we will only want to store a few triples at a time.

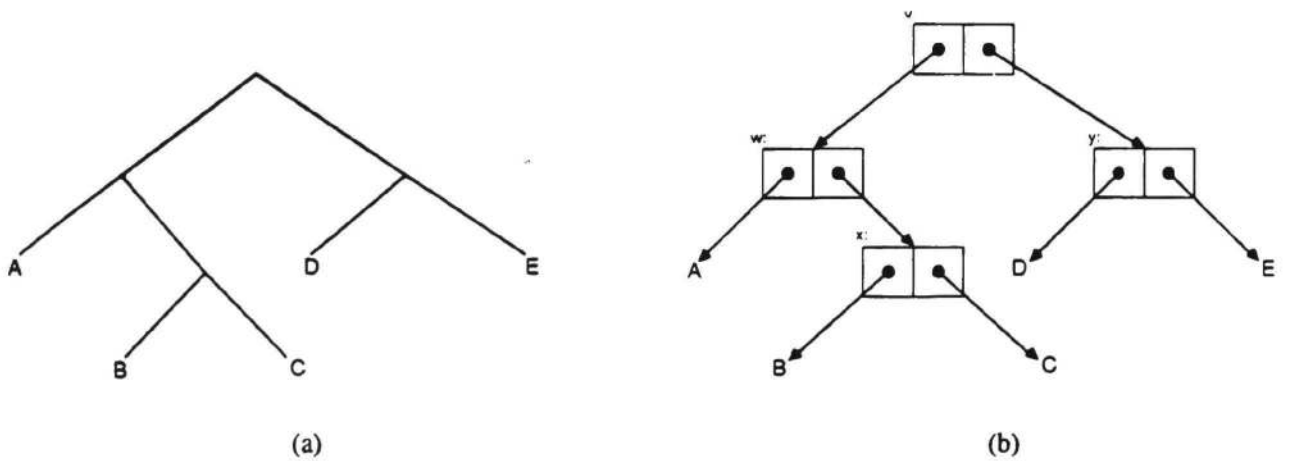

Figure 2: (a) a binary tree; (b) its cons cell representation.

In an earlier paper, Geoffrey Hinton and I described a coarse coded memory for triples of symbols which served as the working memory of a production system interpreter (Touretzky & Hinton, 1985). The same memory design is used in BoltzCONS for encoding cons cells. This scheme is called a distributed representation (Hinton et al., 1986) because each triple is represented by the collective activity of many units, while conversely, each unit contributes to the representation of many possible triples.

[1]Computer scientists might prefer the term "address" to "tag," but a distributed symbol memory such as the one used in BoltzCONS has neither sequential addresses nor discrete memory locations, so "address" would be misleading. Tags are unordered symbols, not integers.

Each memory unit in this representation has a randomly-generated receptive field table such as the one shown in figure 3, with three columns of six symbols each. The receptive field of the unit is defined by the crossproduct of the three columns, e.g., (F A P) would be recognized by the unit described in figure 3 but not (F J T), since J doesn't appear in the second column and T doesn't appear in the third. If the memory contains 2,000 units and there are 25 possible symbols, a triple will fall in the receptive field of $6^3/25^3 \times 2000$ or roughly 28 units, on average. This distributed representation is described as "coarse coded" because each receptive field is a $6 \times 6 \times 6$ slice out of the $25 \times 25 \times 25$ space of possible triples; the receptors can thus be said to be "coarsely tuned" along each of the three dimensions.

C	A	A
F	B	D
H	H	J
P	K	M
S	S	P
W	Z	R

Figure 3: A receptive field table

Starting with a twenty-five symbol alphabet, there are 25^3 or 15,625 possible triples one could potentially store in this memory. The number that can be reliably held there at one time depends on the number of units available. With 2,000 units, one can store roughly a dozen triples at once. As the alphabet size increases, the savings of a distributed representation over an extreme localist one grows rapidly.

To manually store a triple in this memory, we find its 28 or so receptors and turn them on. To delete a triple, we turn its receptors off. To test whether a triple is present, we examine the state of its receptors. If most of them (say, at least 70 percent) are on, we assume that the triple is present; if most are off we assume it is absent. These functions are embedded in the wiring pattern of the network; BoltzCONS does not actually search through receptive field tables when storing or retrieving triples.

The distributed memory has some interesting properties. It is immune to small amounts of noise, i.e., if a few units change state randomly there is no observable effect on the memory's contents. The performance of the memory degrades gradually as the memory fills up. Since each receptor is shared by 6^3 or 216 potential triples, there is some overlap in the representation of triples. As a consequence, if we store a triple in memory and then store and delete many other triples, the original triple will decay and eventually fade away unless it is refreshed by turning on all its receptors again. Finally, the memory exhibits a local blurring phenomenon, whereby if we store many closely related triples such as (F A A), (F A B), (F A C), etc., it becomes increasingly difficult to determine whether a related triple such as (F A L) is present in memory or not. However, it is still possible to tell whether an unrelated triple such as (G V Q) is present. Local blurring, like the gradual decay of triples due to deletions of other triples, is a consequence of the sharing of units in distributed representations.

3. The BoltzCONS Architecture

The architecture of BoltzCONS is shown in figure 4. Cons cells encoded as triples are stored in the space labeled Cons Memory, which contains 2,000 units. There is a one-one mapping of excitatory connections from Cons Memory units to units in the space labeled Cons Pullout; each active Cons Memory unit therefore tries to turn on the corresponding Cons Pullout unit. However, the Cons Pullout units are sufficiently mutually inhibitory so that only about 28 at a time can be on, i.e., just enough to represent one triple. Cons Pullout space is used to "pull out" one triple from the several that are stored in Cons Memory. The term "pullout network," first used by Michael Mozer (1984), is synonymous with the "clause spaces" described by Touretzky & Hinton (1985).

The three spaces labeled TAG, CAR, and CDR are winner-take-all networks (Feldman & Ballard, 1982) which settle into stable states representing one of the 25 symbols in BoltzCONS' alphabet. They are coarse coded versions of the bind spaces described in (Touretzky & Hinton, 1985). Since the spaces are coarse coded, units vote for sets of symbols rather than individual ones. The units all have bidirectional excitatory connections to appropriate units in Cons Pullout space. For example, a unit in CDR space that codes for the symbols **p** and **w** will have connections to a random subset of all the units in Cons Pullout space where either **p** or **w** appears in the third column of the unit's receptive field table.

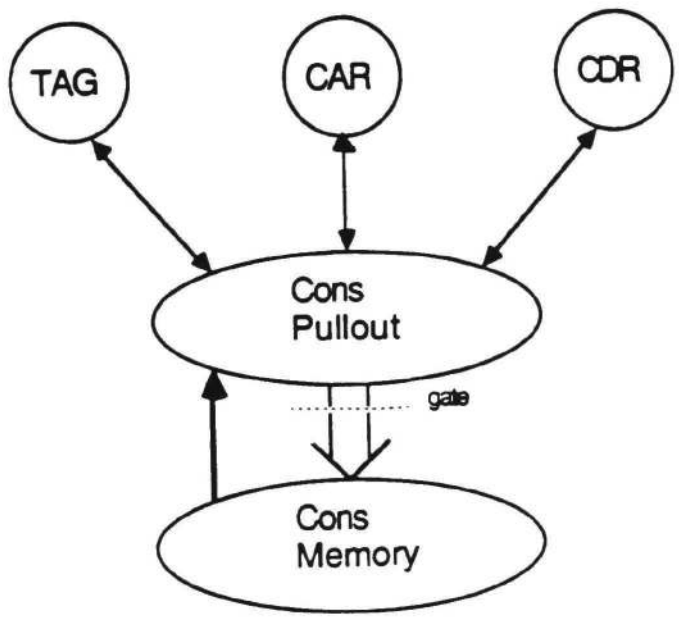

Figure 4: The architecture of BoltzCONS.

4. Manipulating Distributed Stacks

Let us assume the stack of figure 1 has been loaded into BoltzCONS' Cons Memory. The four cons cells in this figure are represented by an activity pattern. The top cell of the stack, encoded as the triple (**p A q**), is also represented by a pattern in Cons Pullout space, and its three component symbols are represented by patterns in TAG, CAR, and CDR spaces, respectively. Now suppose we wish to pop the stack. First, via a gated connection, the Cons Pullout units turn off their corresponding Cons Memory units, thereby deleting the triple (**p A q**) from Cons Memory. Next, the activity pattern in CDR space (denoting the symbol **q**) is transmitted to TAG space, which is then clamped. Finally, while TAG space supplies an excitatory stimulus to Cons Pullout space, a simulated annealing algorithm (Kirkpatrick et al., 1983) is run on the Cons Pullout, CAR and CDR spaces to effect an associative retrieval. The annealing algorithm, which is what makes BoltzCONS a Boltzmann Machine, causes BoltzCONS to search for a minimum energy state subject to the constraints imposed by units in TAG space and

Cons Memory. When it finds this stable, minimum energy state, the Cons Pullout network will have settled into a representation of the triple in Cons Memory whose **tag** component is **q**, namely **(q B r)**, and the CAR and CDR spaces will have settled into states **B** and **r**, respectively, representing the car and cdr components of the new top of the stack.

Pushing an element onto the stack is simpler than popping it, because a stack push does not require annealing. The current top cell of the stack, which after the the first pop would be **(q B r)**, is always represented in both Cons Pullout space and the TAG, CAR, and CDR spaces. To push the symbol **E** onto the stack at this point, we copy the contents of TAG space into CDR space, load the symbol **E** into CAR space, and load a new symbol, say **t**, into TAG space. Then we allow the TAG, CAR, and CDR spaces to independently apply excitation to units in Cons Pullout space. Their combined influence causes the triple **(t E q)** to appear there. (Recall that due to mutual inhibition only about 28 units at a time can be active in the pullout space; the units most likely to be active are those that receive support from the TAG, CAR, and CDR spaces simultaneously.) Finally, by opening a gated connection, Cons Pullout units are allowed to turn on their corresponding Cons Memory units, thus creating a new cons cell.

The use of associative retrieval allows BoltzCONS to perform certain additional stack operations not possible on a convenional machine. One of these is called "associative stack pop." The idea is that rather than popping the stack a fixed number of times, we may want to pop back to a particular state, e.g. the state where C is the top element. This can be accomplished in one step by clamping C into CAR space and running an associative retrieval. Note that in this case the elements "popped" from the stack in order to reach C are still present in Cons Memory; all we have done is move the stack pointer. Because these elements are not deleted, it is possible to restore them by sequentially "un-popping" the stack, again using associative retrieval. To perform one un-pop operation, we copy the symbol presently in TAG space into CDR space and then anneal with CDR space clamped.

5. Unbounded Depth Tree Traversal

One can do many more interesting things with trees than with stacks. Common tree problems include traversal, structural comparison (the Lisp EQUAL function), and terminal node comparison (the "samefringe" problem, often used to motivate coroutines.) Here we consider the problem of traversal. The goal is to find and display in correct order all the terminal nodes of a tree such as the one in figure 2b. The simplest Lisp solution is:

```
(defun traverse (tree)
  (cond ((atom tree) (print tree))
        (t (traverse (car tree))
           (traverse (cdr tree)))))
```

Unfortunately the above algorithm is recursive but not tail-recursive, so it requires a stack. One goal of this paper is to show how recursive structures can be manipulated without resorting to a control stack. (We cannot categorically deny the existence of such a stack in the brain, but our defense of the plausibility of connectionist models would be weakened if it depended on its existence.) The traversal problem cannot be solved on a conventional computer without such a stack.[2] Although we can replace the recursive algorithm with an iterative one, shown below, this merely forces us to build and manage the stack explicitly rather than relying on Lisp's internal control stack.

[2]This claim rests on two assumptions: that trees are represented as *one-way* linked lists as in Lisp, and that destructive operations on the tree are not permitted.

```
(defun iterative-traverse (tree &aux stack)
  (loop
    (cond ((consp tree)
           (push (cdr tree) stack)
           (setq tree (car tree)))
          (t (print tree)
             (if (null stack) (return nil)
                 (setq tree (pop stack)))))))
```

The iterative algorithm below, which uses associative retrieval, does not require a stack to execute. For conciseness, it is expressed using the abstract Lisp terminology of variables, pointers, and cons cells, but its implementation is actually in terms of BoltzCONS operations. We assume that the input tree contains at least one cons cell, and that the variable PTR, which points to the current position in the tree as we traverse it, starts out by pointing to the root.

1. If PTR points to a cons cell, set OLD to PTR, set PTR to its car, and go to step 1. Otherwise PTR must point to a symbol, so print it; then set PTR to cdr of OLD and go to step 2.
2. If PTR now points to a cons cell, go to step 1. Otherwise PTR must point to a symbol, so print it. Then set PTR to OLD and go to step 3.
3. If PTR points to the root, halt. Otherwise, use associative retrieval to search for a cell whose car is PTR. If the retrieval succeeds, make OLD point to that cell, set PTR to the cdr of OLD, and go to step 2.
4. Use associative retrieval to locate the cell whose cdr is PTR. Make PTR point to that cell and go to step 3.

The most important feature of this algorithm is that associative retrieval is used to follow pointers "backward," an operation that is not possible in Lisp. Since a cell may be pointed to by either the car or cdr of its parent cell in the tree, two retrieval attempts may be necessary in order to find the parent. Steps 3 and 4 accomplish this. Table 1 shows a trace of the algorithm during a traversal of the tree in figure 2b. The values shown for PTR and OLD are those immediately before the given step is executed. The places where the algorithm backs up over a pointer are indicated.

Step	PTR	OLD	Action or Comment
1	v		
1	w	v	
1	A	w	print "A"
2	x	w	
1	x	w	
1	B	x	print "B"
2	C	x	print "C"
3	x	x	*Associative retrieval on CAR fails,*
4	x	x	*but retrieval on CDR succeeds.*
3	w	x	*Retrieval on CAR succeeds.*
2	y	v	
1	D	y	print "D"
2	E	y	print "E"
3	y	y	*Associative retrieval on CAR fails,*
4	y	y	*but retrieval on CDR succeeds.*
3	v	y	halt

Table 1: Tree traversal algorithm applied to figure 2b.

Because we can follow pointers backward, there is no need to use a control stack for backtracking. In theory, then, the above iterative algorithm can be used to traverse any binary tree no matter how great its depth. In practice the only depth limitation derives from the size of the trees BoltzCONS can store, which in turn depends on the number of symbols in its alphabet and the number of triples that can be stored in Cons Memory.

6. Use of Hopfield's Energy Measure

Because BoltzCONS behaves as a Boltzmann Machine during simulated annealing[3], it acts to minimize an energy measure as it settles into a stable state. This energy measure was first proposed by Hopfield in an analogy to spin glass models in physics (Hopfield, 1982). If s_i is the state of the *ith* unit (either 0 or 1), θ_i is its threshold, and $w_{i,j}$ is the weight between it and the *jth* unit, then the energy of the network in a given state is:

$$E = \sum_i s_i \theta_i - \sum_{i<j} s_i s_j w_{i,j}$$

The energy measure can be used to detect whether an associative retrieval has succeeded. For example, suppose BoltzCONS is examining the cons cell labeled **x** in figure 2b, and the traversal algorithm now wants to find the parent cons cell. The parent will either have **x** in its car or in its cdr. If we clamp the symbol **x** into CAR space and run an annealing, the network will of course settle into an energy minimum, but in this case the energy will be high. This is because there is no cons cell in Cons Memory with **x** in its car, so there is no way for the network to find a good solution to the combination of constraints imposed by CAR space and Cons Memory space. If the energy of the stable state found by the annealing algorithm is above some empirically determined threshold, we know that the associative retrieval has failed.

When the first retrieval fails, BoltzCONS can try a retrieval with the symbol **x** clamped into CDR space instead. This time the retrieval will succeed, with the Cons Pullout units settling into a representation of the triple (**w A x**). The low energy of this state confirms that a valid cons cell has been retrieved.

Of course, we would prefer that the units in a connectionst model not be forced to measure global properties such as energy in order to function. In BoltzCONS, they don't. What actually happens is this: after the network has settled into a stable state, the thresholds of all the units are raised by a fixed amount[4], thus changing the energy landscape. If the network is in a deep energy minimum, representing a valid retrieval, its state will remain stable. However, if the network has settled into a shallow minimum, then when the thresholds change it will no longer be in a minmum energy state, but in a state from which it can reach a new local minimum by turning all its units off. So, the test for a successful associative retrieval consists of raising the thresholds, running the annealing algorithm for a few more steps at very low temperature (which makes the network act like a Hopfield net), and then checking to see whether any units remain active.

7. Controlling BoltzCONS

One important issue that has not been addressed in this paper is how BoltzCONS is to be controlled. Some external agent must send it commands to clamp a space, copy the state of one space into another, or begin an annealing. A simple sequence of these commands results in a stack push or pop; a more elaborate sequence can result in a tree traversal. For the work described here, the job of controlling BoltzCONS was handled by a piece of Lisp code. In addition to issuing control commands, the Lisp code was responsible for generating new tags for stack

[3]BoltzCONS is not a true Boltzmann Machine because it contains gated and asymmetric connections. However, during an annealing the network is equivalent to one that is a true Boltzmann Machine.

[4]This has the same effect as supplying all the units with some inhibitory bias.

push operations.

Recent work on transforming parse trees in a neural net led to the development of a connectionist control unit for BoltzCONS (Touretzky, 1986). This control unit is a modified version of Touretzky and Hinton's production system interpreter. Production rules describing a multi-step parse tree transformation contain right hand side actions that send commands and data to BoltzCONS. Executing these production rules in the proper sequence causes the parse tree (represented in BoltzCONS' Cons Memory) to be transformed in the desired way. A similar control unit, programmed with a different set of production rules, could be used to automate the steps involved in tree traversal.

8. Discussion

Turing machines, which as far as anyone knows can compute all computable functions, consist of an immutable finite state machine and a mutable external tape of unbounded length. Conventional von Neumann computers, being of fixed size, are merely large finite state machines, but it is usually more helpful to think of them as imperfect Turing machines. Doing so allows us to divide the computer into a finite state machine whose repertoire of states is relatively small, plus a tape (embedded in the computer's memory rather than being external) that is much larger in size, albeit finite. The linear address structure of a von Neumann machine's memory makes it natural to equate this memory with the Turing machine's tape.

Connectionists much prefer distributed memory models, both because they are more physiologically plausible and because they are more amenable to parallel processing. A distributed memory such as the one found in BoltzCONS offers neither discrete locations nor sequential addresses. Therefore the analogy between connectionist hardware and imperfect Turing machines is less straightforward (Pylyshyn, 1984). While there is no evidence that connectionist models can compute things Turing machines can't, to harp on this misses the real point, which is that connectionist models can compute things *in ways in which* Turing machines and von Neumann computers can't. The ability to rapidly solve complex constraint satisfaction problems through massive parallelism, such as when performing associative retrievals, is a crucial part of the connectionist program.

9. Conclusions

This work demonstrates that connectionist models can indeed represent and manipulate recursive symbolic structures. Furthermore, thanks to associative retrieval, these recursive structures can sometimes be manipulated iteratively without the use of stacks, while a conventional computer would require a stack. The associative retrieval capability also makes certain operations such as associative stack pop possible that cannot be done (in constant time) on a conventional computer.

Three techniques used in BoltzCONS -- coarse coded symbol memories, pullout networks, and bind spaces, were first put together in a neural network production system interpreter. They now appear to be generally useful devices for connectionist symbol processing, and are likely to turn up again in other connectionist applications. Coarse coded, distributed representations are useful for building complex symbol structures economically. Without pullout networks and bind spaces, though, we would have no way to manipulate them.

Acknowledgements

This material is based on work supported by a grant from the System Development Foundation, and by National Science Foundation Grant No. IST-8516330. I am grateful to Geoffrey Hinton, Mark Derthick, Jay McClelland, and David Plaut for useful discussions, and to Cindy Wood for help with the illustrations.

References

Ackley, D. H., Hinton, G. E., & Sejnowski, T. J. (1985) A learning algorithm for Boltzmann Machines. *Cognitive Science* 9(1):147-169.

Fahlman, S. E., Hinton, G. E., & Sejnowski, T. J. (1983) Massively parallel architectures for AI: Netl, Thistle, and Boltzmann Machines. *Proceedings of AAAI-83*, Washington, DC, 109-113.

Feldman, J. A., & Ballard, D. H. (1982) Connectionist models and their properties. *Cognitive Science* 6:205-254.

Hinton, G. E., McClelland, J. L., & Rumelhart, D. E. (1986) Distributed representations. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume I.* Bradford Books, Cambridge, MA.

Hopfield, J. J. (1982) Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences USA* 79:2554-2558.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983) Optimization by simulated annealing. *Science* 220:671-680.

Mozer, M. C. (1984) The perception of multiple objects: a parallel, distributed processing approach. Unpublished thesis proposal, Institute of Cognitive Science, University of California at San Diego.

Pylyshyn, Z. W. (1984) Why computation requires symbols. *Proceedings of the Sixth Annual Conference of the Cognitive Science Society*, 71-73.

Touretzky, D. S., & Hinton, G. E. (1985) Symbols among the neurons: details of a connectionist inference architecture. *Proceedings of IJCAI-85*, Los Angeles, CA.

Touretzky, D. S. (1986) Representing and transforming recursive objects in a neural network, or "trees *do* grow on Boltzmann machines." *Proceedings of the 1986 IEEE International Conference on Systems, Man, and Cybernetics*, Atlanta, GA.