[22] One of the most striking features of connectionist architecture is that there is no clear distinction between memory and central processing. The only representations literally present in a system at any given time are the active ones. Memories, and all representations, are literally (re-) created on the spot.

Thus, the soft generalizations we envision will describe the *representational dispositions* of the system to construct new representations from currently active ones. For each of a vast range of *representational* states that might occur, the system's subrepresentational structure would dispose it to generate a new total representational state that is highly sensitive to the structure and content of the initial state (but, as we said in response to the syntactic argument, not necessarily determined by the preceding representational state).

In some cases, the basis for these dispositions might be fairly direct, as in simple inductive reasoning, in which associative/statistical relations could be encoded in weights. But in general, we suppose that the relation between the subrepresentational basis and the representational level will be both complex and remote, so that, for example, knowledge will be in weights only of the system as a whole. There will be no single weight, or small set of weights, that encodes any particular item of knowledge.

[23] These components need not be *spatially* distinct; there are useful non-spatial ways of defining *component* connectionistically. See Note 11.

[24] Simple rules of logic are sometimes cited as examples of hard rules in human cognition. But if you believe P and you believe if P, then Q, and you put them together, it does not follow that you will believe Q. You might be unwilling to accept Q, and hence give up P (or the conditional) instead, or simply not know what to believe. Modus ponens is not a rule of mental *processing*.

[25] See our "Representations without Rules," *Philosophical Topics* 17 (1989), 147-174; "Soft Laws," *Midwest Studies in Philosophy* 15 (1990), 256-279; and *Connectionism and the Philosophy of Psychology*, Cambridge, MA: MIT Press, forthcoming.

Department of Philosophy
Memphis State University
Memphis, TN 38152

Kirsh, D. Putting a Price on Cognition, Southern Journal of Philosophy, Supp. volume XXVI, 1987. pp. 119-135, Reprinted in Connectionism and Philosophy of Mind. T. Horgan, J. Tienson, (ed), Kluwer. 1990

DAVID KIRSH

PUTTING A PRICE ON COGNITION

INTRODUCTION

In this essay I shall consider a certain methodological claim gaining currency in connectionist quarters: The claim that variables are costly to implement in PDP systems and hence are not likely to be as important in cognitive processing as orthodox theories of cognition assume.

It is widely argued, as a consequence of connectionist accounts, that an adequate theory of human activity must explain how we can act intelligently without much formal reasoning, even at an unconscious level. Formal inference, whether mathematical or purely logical, is something ordinary people do poorly in their heads. To prove theorems, we generally need the help of pencil and paper to keep track of our progress. The same holds for generating proofs in grade school algebra, or in searching a network of possibilities. Humans do not find it easy to recall dozens of freshly generated premises. It is easier to write down intermediate results and then act responsively to those scribblings.

What applies at a conscious level may also apply at an unconscious level due to the nature of the human processor. From a computational viewpoint, an agent places the same demands on memory and processing power whether he reasons consciously or unconsciously. Inference is inference. Whatever makes it difficult for us to reason at a conscious level in a logical manner may also make it difficult for us to reason logically at an unconscious level. In each case, our memory is required to store the values of a large number of variables. If our memories are not appropriately designed for this task, this will be difficult.

According to the classical theory of Artificial Intelligence, (hereafter AI), most intelligent actions presuppose a sizable amount of internal search, which in turn presupposes a sizeable amount of variable binding. Unconscious inference is rampant. For instance, AI dogma holds that rational agents decide what to do next by running through in their heads the consequences of performing chains of actions. They plan. They weigh

alternative courses of action, simulating the consequences of selecting one over another, and choose the best.

Variables enter this picture because actions are defined in AI theories in a sufficiently general way to be applied in many situations. Thus an action, such as **MOVE X**, will standardly be defined by a set of pre and postconditions, which state that IF X satisfies properties $P_1, P_2 \ldots$ such as *X is-liftable, X is-graspable, X is-freely-movable* THEN X will satisfy certain other properties such as *X occupy-newposition, X is-freely-movable...* Accordingly, to decide whether it is wise to move a certain book, an AI system will first bind X to the particular object in question, say **book-9107**, then determine whether **book-9107** satisfies MOVE's preconditions, then update **book-9107**'s properties and relations in accordance with MOVE's postconditions in order to determine whether the changes in the world as a whole bring the agent closer to its goals.

Variables, and the process of binding variables to specific values are an essential part of the standard AI model of reasoning. Indeed, the standard model does not bind variables every now and then; variables are continually being bound. To return to the planning example, a plan itself is represented in an AI planner as a variable that takes a sequence of actions as a value. As a plan is built up, more actions are added to the sequence. The plan is lengthened, hence the variable **plan** is given new values. But old bindings are not dropped away. In many planning contexts we will have to *backtrack:* we will have to delete members in our action sequence, because we discover that certain sequences lead to a dead end, or paint us into a corner, or cause looping. In those cases, we want to be able to unwind the action sequence, and pick up our simulation of the effects of the plan at an earlier choice point where we believe we first chose unwisely. If actions are not totally reversible, the only way we can pick up our simulation at an earlier point without having to start all over again from world state zero, is if we have stored a trace of the world state at each choice point in our plan.

The upshot is that intelligent action, on the received view, involves reasoning with hundreds, perhaps thousands of variables and names. To cope with this complicated assignment of values to variables, standard computers maintain a large address space where they can keep track of the connections between variables and their values, and between variables and other variables. As changes occur the memory cells assigned to store the current value of a variable change. Similarly, as new connections are made between variables, such as that K=X, or that G belongs to the same family as H, additional pointers are introduced to ensure that the relevant inferences about inherited properties etc. can be made.

Now, if it is true that intelligent action requires much unconscious inference, and if it is true that variables are involved in inference, then nature must have found a method of implementing variables neurally. One of the more interesting methodological premises of connectionism is that if certain psychological capacities are expensive to implement in connectionist hardware, then they are expensive to implement in neural hardware, and it is reasonable to suppose that we exercise those psychological capacities far less regularly than assumed in standard computational modes. Accordingly, if we wish to have psychologically realistic models of cognition, we should construct computational models that are less profligate in their use of these 'expensive' capacities. In particular, we should construct computational models of, say, intelligent action, or skillful activity, which do not require much reasoning in a rule governed, or algebraic style. We are advised, that is, to rethink cognitive problems with an eye to finding computational solutions that do not rely on manipulation many explicit variables or names.

If this methodological directive can be made to work, it will be of great import. At present, we conceive of cognitive tasks in algebraic terms, wherever possible. Much of cognition is modelled on problem solving, and our understanding of problem solving is biased by the few explicit algebraic models we have. Who is to say that we are not being narrow minded in our conceptualization of these problems? According to the way we understand cognition now, most higher forms of cognition make essential use of variables, quantifiers, and search control knowledge. Might this not be a bald mistake? Might it not be possible to rethink cognition in a way that does not require those mechanisms?

In what follows I shall make a first pass at appraising this argument. How believable is it? Why couldn't evolution build brains that paid the full cortical price of variables? Do the benefits of having variables as basic psychological capacities outweigh their costs? How should we measure cost? And what exactly is the problem of building PDP networks with variables?

The paper itself is divided in three parts. In the first part, I discuss what a variable is and what powers it confers upon its users. I then consider why standard PDP systems have trouble learning variables and why specially designed PDP systems that support variable binding must pay a high price for that ability. I conclude with a brief discussion of alternative theories of cognition that try to accommodate constraints on variable binding.

What Do Variables Buy Us?

In algebra, variables serve two basic functions: as placeholders for unknowns, and as simplifying descriptions for more complex structures or expressions.

First, as placeholders for unknowns, variables serve to mark a subject that we have yet to identify, as in x is whatever satisfies the following conditions:

$x > 3$
x is odd
$x < 7$

That is, x serves the same function as the impersonal pronoun 'it' in

It transports about 7 people spacebound.
It is covered with heat resistant tiles.
It weighs more than 10 tons.

To decide what *it* refers to, or to determine the value of x, we treat each condition as imposing a constraint on the set of entities that might be possible objects of reference. We then *analyze* the interaction of these conditions to single out the referent that satisfies all the constraints. That is, if a given system of equations determines a unique answer, we can often discover that answer analytically, rather than by plugging in separate values for the equations in order to decide empirically by trial and error what the solution is.

Moreover, once we have variables around, we can often exploit them for higher forms of analysis too. For instance, if we can describe the behavior of a system by a set of n linear equations in n variables, we can determine without calculation that there is a unique answer. Similarly, we can determine whether the behavior of a system is under-determined, and hence indeterminate with respect to our knowledge. The power of analysis should not be underestimated.

Related to analysis, but worth distinguishing, is the power of generalization or abstraction. This is the second major function of variables. It sometimes happens that we know the referent of a variable, or we know a description that uniquely specifies it, but we need a way of referring to

that referent compendiously. In these cases, variables let us simplify descriptions; they let us abstract from complexity.

For instance, to state that when we add two numbers together it does not matter which number we take first (the commutative property of addition), we write down the simple expression $a + b = b + a$.

Variables, here, provide us with a way of *explicitly specifying* constraints or invariant relations that hold over a class of entities. There is no doubt about what the variables refer to: numbers. But variables provide us with the needed mechanism for stating what the generalization is. Without variables to range over all numbers we would have to specify the relation in extension, showing for each number pair that it sums identically whatever its order.

The full power of this descriptive feature of algebra only becomes evident if we consider just how complex the regularities are that we can describe simply. This is the celebrated virtue of notation.

In the case of the set $[(1,1)(2,4)(3,9) ..., (x,x^2)]$ it is easy to notice that $y + x^2$ expresses the invariant relation that holds among the members. As with the commutative property, variables here let us state a general relation compendiously, though it is less clear that we need variables themselves to find the invariant relation in the first place.

But consider recursion. A sequence such as [1, 2, 6, 24, 120, 720 ...] may have no obvious structure to it until we think to compare each member with the member preceding it. Even then, the structure is not transparent, for we must note that if we divide every number by the number preceding it, we get a factor that is equal to n, the position in the sequence. Clearly, the invariant here is much harder to discover. And for a good reason. The position a number has in a sequence is not explicitly represented as an element in the sequence itself. The invariant here is not a relation between members, but a relation between members and their position in the sequence. How can we be expected to notice that?

We need a perspicuous notation. Owing to the creative nature of algebra, we can create whatever notation we need. In this case, a notation using variabilized names, as in R_n and R_{n-1}, where the variable n explicitly designates the position in the sequence and R_n designates the number at that position, nicely exposes both position and number. With this notation we can state the relevant invariance. Without such notation it is extremely hard to describe and recognize such regularities. We cannot keep track of their structure.

It is one of the singular virtues of algebra, then, that it makes it easy to generate increasingly abstract functions, abstract names for relations. This power, too, should not be underestimated.

To distinguish the two powers discussed, let us call the first, *the power of analysis*, and the second, *the power of abstraction.*

In order to be able to achieve these two powers a variable must satisfy at least three conditions:

1. Variables must be logically distinct from their values. Thus tokens of x must be logically distinct from tokens of the value of x. This means we can create tokens of a variable on the fly, erase and duplicate those tokens, (i.e., physically pass them around), while not physically touching or creating tokens of its value.

2. Variables and their values must be semantically connected in the interesting sense that values can be explicitly assigned to variables, as in the statement *Therefore* $x = 3$ *and* $y = 1$; or in the statement *Let* $x = 5$ where x is assigned a value in order to test certain conjectures by substituting 5 for x. In physical systems this semantic relation must be enforced by a process which can *access x's* assigned values and use that value in place of x. Thus, characteristically, x will be physically connected to its values. And x will hold that value stably until some process alters it.

3. Variables, unlike names, must range over a set with more than one element. Thus a variable is something with a suppressed quantifier. This is the feature which allows them to encode generalizations so simply, as in '$a + b = c$', where a, b, and c are all universally quantified; and also to serve as logical subjects of predicates, as in x weighs more than 10 tons, where the logical form displays an implicit existential quantifier: 'There exists an x, such that ....'

Variables behave in a related manner in programming languages. In LISP, for instance, great power is achieved by relying on simple names and variables to refer to the results of performing complex computations. We can call on a variable, or pass it around, without triggering the process that would determine its value. The two entities: variable and value are logically distinct. Thus we can use a variable to refer to the result of performing

arbitrarily complex operations. In this way computational objects of increasing complexity can be built up step by step. For once we have a name for a function, we can use it as an argument in a higher order function, and so incrementally increase the complexity of the functions we can define.

For example, we may define (square x) to be the result of multiplying x by itself, as in:

```
(define (square x) (* x x))
```

Once defined, however, we can use **square** as a building block in defining other procedures. For example, $x^2 + y^2$ can be expressed as:

```
(+ (square x) (square y))
```

and we can define a new procedure **sum-of-squares** that, given any two numbers as arguments, produces the sum of their squares:

```
(define (sum-of-squares x y) (+ (square x)(square y)))
```

And so on.

Moreover, analytic power is exploited in programming languages because the value sought after need not be known. We may wish to introduce a variable to hold the place of the value we are trying to discover. As more is known, constraints are added to the description of x. At some point it may be possible to show that a unique answer exists, or that no answer exists; or it may be possible to simply solve for x.

The power of a variable, then, whether in algebra or in programming, comes from having something *explicit* in a system which can serve as a proxy for a value (or a procedure if the value of the variable is a procedure). Because the two entities, variable and value, are logically distinct we can perform operations on variables without actually performing operations on the values they designate.

We must now consider why it is hard for PDP systems to learn and implement variables.

Systems Which Use Variables Do More Than Just Pass Values

The simplest explanation of why standard PDP systems have trouble learning and implementing variables is that PDP systems do not have separate tokens for variables and their values. Hence, unless they are specially designed, they cannot distinguish variables and values. Input enters a system as activations on input nodes, it gets transformed by weights and nodal functions, and finally it emerges as activations on output nodes. The whole system works simply by modifying and passing activations from input to output. The system simply propagates values. Hence standard PDP systems crunch values, not variables.

Now of course at one level of analysis familiar variable manipulating computers simply work by modifying and passing activations from input to output as well. All physical systems crunch values. But at another level of analysis it is essential to the performance of digital computers that they explicitly represent variables and rules. This difference in explicitness marks a real difference in possible performance.

To see this difference, imagine that we have an electrical circuit which obeys the constraints $x + y = 4$ and $2x - y = 5$ without explicitly representing those constraints. See Figure 1a. This circuit performs as an activity passer. Input enters the system as activations on certain wires, it gets transformed at certain modules in accordance with certain functions, and finally it emerges as activations on output wires. It is immaterial to our general point that the modules are labelled and that it is easy for us to interpret the 'meaning' of the signals flowing across wires. For whether *we* can interpret what is happening in the system is irrelevant to whether the system itself is merely an activity propagator. If at a certain point in its processing the system passes a signal with the value of 6, we may interpret that signal, in light of its function in the whole network, to mean something like 'the value of $2x$ is 6 right now'. But *our* ability to so interpret the signal should not be confused with that signal being treated by the system as a token of the variable $2x$. The system has no way of referring to $2x$ in general. It behaves exactly like a localist connectionist device, passing values around without having a means of independently labelling those values.

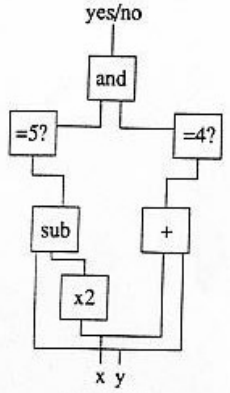

Figure 1a

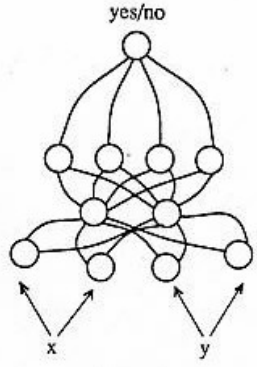

Figure 1b

Figure 1. A non-connectionist circuit which acts *in accordance* with the rules $x + y = 4$ and $2x - y = 5$. Because wires flow between labelled modules, it is easy to interpret the meanings of the signals that flow through the system. Nonetheless, this system does not have variables represented explicitly. At each point in the process there are only values. There is no logical distinction between a token of a value and a token of a variable. The system is a value propagator rather than a variable manipulator. Figure 1b is a connectionist version that implements the same constraint system. Exactly how the constraints are embedded in the network constraints is more complex.

Such a circuit was designed to behave in a predictable matter. If we put in any values for (x, y) other than (3,1) the system outputs (0). For (3,1), the system outputs (1). What ensures that this system is a value propagator, and nothing more, is that the only way it has of deciding whether a given pair is acceptable--that is, of deciding whether the pair satisfies the constraints--is to have the pair presented as input. It can decide only by trial and error. It cannot determine analytically that (3,1) is the only answer because analysis requires explicit representation of the constraints.

Thus the system acts as a recognizer; it decides only by being presented with input. It can never make any global judgments about possible inputs. For systems that can pass only values of variables can reason only with single values.

By contrast, systems that use variables as entities distinct from their values can achieve the effect of reasoning over sets of values in single transformations, as when x is assumed to refer to the odd integers. There

is no way this extra power can be recouped by a system lacking explicit variables. Even if a system used some clever way of coding lists of numbers instead of singletons in order to operate over lists, this clever coding could never encode infinite lists. In any event, the type of circuits that would operate over lists of numbers would be different in design than those designed to act in accordance with the rules $x + y = 4$ and $2x - y = 5$ as defined earlier. The list processing circuit would, in effect, be processing the rules $x + y = 4$ and $2x - y + 5$, where x, y, 4 and 5 refer to lists or matrices of the same cardinality, as in $[x_1,x_2,...,x_n + y_1,y_2,...,y_n = 4, 4,...,4]$. These are different constraints entirely than the simple $x + y = 4$ and $2x - y = 5$ constraints where x and y refer to single values.

The upshot is that systems that act in accordance with constraints (rules) are not as powerful as systems which explicitly represent the constraints they use. Explicit representation buys analytic power.

The Cost Of Implementing Genuine Variables

Suppose, now, that a connectionist wishes to instill his system with the power to use explicit variables. This means that whatever implements the variable must in principle be able to be bound to any value in the variable's range, and to be usable in more ways than just as the object of a 'retrieve value' call--i.e., to be manipulable. How might this be done?

In ordinary computers, to achieve this flexibility, and to permit the variable to be bound as many times as is needed for reasoning, the standard digital design is to append a binary pointer to the variable to indicate the address of the memory cell containing its current value. If the same variable is multiply instantiated, as when Px is instantiated by Pa, and Pb, and Pc, and all these bindings are held in memory, there is some contextual marker used to distinguish when x is bound to a, b, or c. In both cases, though, each memory cell is free to accept any value, including a pointer to other memory cells, for it often happens that the value bound to one variable is itself a variable which in turn has a value bound to it.

Such a system satisfies the three conditions of genuine variablehood mentioned above. It logically (and physically) separates variable tokens from value tokens; it permits explicit assignment of values; and because of the freedom, in principle, to write any value in a memory cell it treats variables as ranging over many possibilities.

Pointers are an elegant, even if obvious, solution to the variable binding problem. But they have their cost. They work only if a system can read addresses, write values and new addresses in memory cells, and follow the address named in a pointer to the relevant memory location. Connectionist systems do not operate like that. They are not designed to create and exploit pointers on the fly.

How Do Connectionists Bind Variables?

One obvious way connectionists bind variables is shown in Figure 2. Here the fact that A can range over the values 1 to 4 means that we must be able to distinguish A with value 1, from A with value 2, and so on. As A's value changes, its effect on the computation performed by the network must change. Thus A with the value 2 must effect the network as a whole differently than A with the value 3.

To do this, a network is constructed with enough nodes to represent every variable value pair. Thus, if we have n variables and m values, we will have n times m nodes. The greater the number of values each variable can range over, the greater the number of nodes that will be required. This model is a purely localist representation, for each node has a unique representational function in the system. Nodes are dedicated; the same node can never participate in the representation of other variable value pairs.

As Smolensky[1] has pointed out, purely local representations of variable value pairs suffer from three problems:

(1) n times m units are required, most of which are inactive and do no work at any given time;
(2) the number of units and hence the number of possible pairings has a fixed, rigid upper limit;
(3) the fact that different variables may take the same or similar values is not exploited; thus every pairing is distinct, displaying no common structure.

Each of these problems, in effect, emphasizes that a localist representation of pairings is not space efficient.

Smolensky's own solution diverges from the localist answer in using wholly distributed representations. Accordingly, the same units may be used

to represent many different variable value pairs. This has the effect of answering his three objections because 1) almost all units will be involved in representing variable value pairs, hence units will rarely be inactive; 2) the number of units will not set a rigid upper limit on possible pairings because a set of units can always be trained to accept another distributed representation defined over it, as long as all representations remain linearly separable; and 3) similar pairings will be represented by similar representations because that is a natural feature of distributed representations.[2]

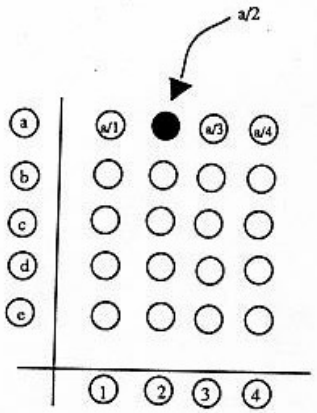

Figure 2. A localist method of binding variables. Neither the variables along the left edge of the matrix, or the values along the bottom, are actually present. Rather the network consists of the central units, which represent variable value pairs. Thus when A has the value 2 the unit in the second column at the top comes on. All the other values for A are inhibited. The system has the capacity to simultaneously represent one value for every variable; for each variable is independent. But it is impossible in this design to assign more than one value to a variable.

Yet this is still not good enough. A tensor product approach also suffers from inefficiency when we consider large value sets. Although we can bind more variables to their values by superposition, we cannot bind more values to a variable without increasing the vector field.

Furthermore, a tensor product approach does not distinguish a token of a variable from a token of a value. This is not an intrinsic limitation, for

it is likely that if we added real nodes for the left column and bottom row, the matrix would now have distinct representations of variables and distinct representations of values, but the cost will again be greater nodal size.

Other connectionists, such as, Touretzky and Hinton,[3] Touretzky,[4] have developed elaborately structured models which explicitly represent tokens of variables and tokens of values. These support variable binding, and even some measure of recursion. But, as Pollack has argued[5]

(1) A large amount of human effort was involved in the design of these systems; and
(2) They require expensive and complex access mechanisms, such as pullout networks or clause spaces.

In short, the various elaborate solutions seem both *ad hoc*, and more importantly, inefficient. A high price is paid to be able to bind a small number of variables to a small number of possible values. Once again, as we increase the number of values the variables can be bound to, the size of the network must be expanded by a multiple. This might not be a problem if we never use variables that range over large sets, but *prima facie* that is just false. It is natural and easy to entertain thoughts about arbitrary people we have met, even though that number easily exceeds a thousand. The same applies to numbers, books, houses, friends, countrymen and Romans. *Prima facie,* we often think thoughts whose logical form quantifies over variables with large ranges. There seems no ready way of escaping the fact that a static network has to pay a high price for duplicating the powers of a dynamically reconfigurable network.

Static vs. Reconfigurable Networks

To see what, at bottom, is the real problem of variable binding let us consider in the simplest manner how both dynamically reconfigurable and static networks might bind variables.

To make the problem concrete, let us suppose we are a telephone company hired to build a network which will link salesmen in Building A with possible buyers in Building B. Each salesman wants to be able to talk to each potential buyer. We may think of this association as follows: A_1 wishes to bind with $B_1, B_2, ..., B_m$, A_2 wishes the same thing, and so on up to A_n.[6]

Now, the simplest case is where each salesman has a direct line to each buyer. In Figure 3, we see what this looks like for the retrograde case where there is only one salesman, and then in the complete case where there are A_n salesmen. Networks like 3b are called totally connected. The localist network described earlier formally resembles this case. Perceptron models are also examples of totally connected networks, as are the layers in PDP networks, though characteristically these other systems use non-localist representations.

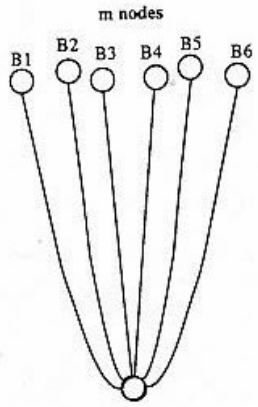

Figure 3a

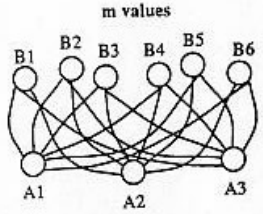

Figure 3b

Figure 3. In 3a and 3b we see examples of totally connected networks. 3a shows the retrograde case where there is only one node (one variable) connected by dedicated lines to m nodes (values). 3b shows the standard graph, where n variables are connected with m values. As the number of values rises by j the number of connections increases by nj.

A reconfigurable network, by contrast, will be more like a telephone system, where any salesman can reach any buyer by dialing his location. It is a further characteristic of such systems that buyers too can reach any other buyer.

Reconfigurable networks have switches which translate addresses into paths that lead to the physical unit residing at that address. In Figure 4 we see a simple switching network. It takes addresses as input, and locks open the relevant path from origin to address.

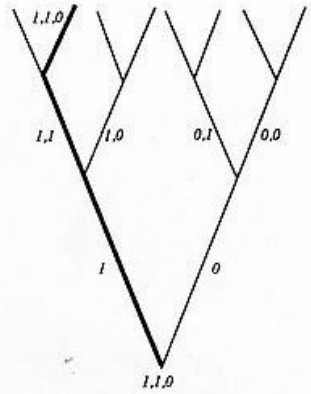

Figure 4. A switching network, such as a telephone system, is designed to open paths to named destinations. In a simple system, each numeral in the destination address corresponds to a position of a switch. As the address courses through the system, local switches mechanically check the relevant digit and respond appropriately. In this figure we can see the system opening the path (1,1,0).

Now, let us consider how the two networks fair on two cases: 1) where x may designate a list of arbitrary length; and 2) where we stipulate that $x \quad x +$ 1, as part of an iteration, or a recursive call.

To cope with a *list* of arbitrary length, where x may be$[n_1]$ or$[n_1, n_2]$ or $[n_1, n_2, \ldots, n_m]$, a static network of the sort just shown must be large enough to have direct lines not only to the n possible values that can occupy every position, but n^m nodes, for each list structure is a distinct possible value.

Put in terms of buyers and sellers, if a certain seller wishes to canvass the opinions of buyer 1, and buyer 2, first separately, then sequentially, where the sequence could matter, he will have to have set up his lines specially for each possibility. He cannot alter the set up once the lines are laid down, so he cannot alter, on the fly, whom he speaks with. If he wishes to speak with buyer 1, the system will be set up so that he speaks to buyer 1 and no one else. If he wishes to speak to several buyers simultaneously, the same network can be rewired to permit that too. But if he wishes to speak to several buyers simultaneously where sequence matters--a type of conference call where he hears from each buyer in predetermined order--the network will have to be extended, for his lines cannot mark order.

By contrast, in reconfigurable networks we can handle sequences and changes in values simply by binding the new values with pointers. Thus, to handle extensions to a list, we can begin by pointing to [1] as the value of x, and then if we discover that x now is [1,2], we cancel the left parenthesis marker and add a pointer to a new memory cell which shows that the value of the second position in the list is 2. Unlike static networks, longer lists do not require larger memory capacities, for we can introduce pointers that pick out existing memory cells, providing they contain the appropriate values. Moreover, we are not obliged to destroy our first entry; we simply add to it by creating pointers as we need it. So the process is conservative, monotonic. Whereas in static systems if the new values range beyond the set of values the network was designed for, a new network would have to be built by adding more units to the old one and then be retrained. Often the new training is more like training from scratch, for the old learned connections do not carry over.[7]

Smolensky maintains that totally distributed representations allow him to reuse some of the same units over again when representing extensions of a set such as the extension from the set [1] to the set [1,2]. This is accomplished by seeing [1,2] as a superposition of [1] and [2].

This is a significant improvement as far as it goes. But as Smolensky emphasizes, this technique applies only if the two values are linearly separable. That means that a network will have trouble distinguishing the sequence [1,2] from the sequence [2,1] unless it has further nodes to represent position.

However, even with nodes for position the system falls short of a pointer system. Suppose we wish to use the same network to represent $x = [n_1, n_2, \ldots n_m]$ and $y = [j_1, j_2, \ldots, j_m]$. It is reasonable to want to make use of large networks to store bindings of more than one variable if they have the same or overlapping ranges. Yet, how can we do this in a static distributed network? The values of y must be superimposed on the network in the same way that [1,--] as a value for x is superimposed on [--,2] as a value for x to yield [1,2] as the composite value for x. But if we superimpose yet another [1,--], this time as a value for y, we will have altered the vector field that is supposed to represent x's value as well. Thus we cannot make use of large networks to store bindings of more than one variable if they have the same or overlapping ranges.

The second task exposes the same difficulty. It is one of the great virtues of a reconfigurable network that portions can be used again and again. Thus, if we have designed a network to perform the procedure $x \rightarrow x + 1$, we can reuse the same network as many times as we like just by resetting the value of x and starting over again. Naturally, the process which makes this possible involves pointers. For on each iteration we must quickly set the new value of x before calling up x's value at the new start of the procedure.

In a static feed-forward network, though, we have to know in advance how many times the procedure in question will be reapplied. We then string out in space as many duplicates of the procedure as will be needed.

To be sure this conversion of iteration to space can be surmounted by having a static network feedback on itself, mediated by a counter which checks that the system does not cycle past a certain value. But this style of network violates the spirit of connectionism. It presupposes that part of the network--the counter--is *insulated* from the other processes--the procedure. A mechanism had to be introduced to let each iteration rapidly set a new value for the variable *number-of-times-iterated-so-far.* Yet this rapid setting of a value is precisely one of the issues that makes variable binding so different in connectionism. Arbitrary iteration then, is another problem for static networks.

Some Consequences

I have been at pains to show that simple connectionist models do not make use of variables in the full and proper sense. More complex models can be constructed which do use variables explicitly, but they pay a high price in size and complexity. If these complex models teach us anything about neural design, it seems to be that nature does not encourage the type of reasoning associated with explicit rules.

This is not a strong argument against rule-governed thought, however. For a sequence of information states may be rule-governed even if there is no rule explicitly represented in a system to structure the process. Such a sequence is correctly called rule-governed, as opposed to rule-obeying, if it supports the appropriate counterfactuals. For instance, if a certain bit of information had not entered the system then the resulting information trajectory would have to change exactly as predicted by the rule set. Similarly, if the process were to be interrupted, or pushed to an extreme, the state of the system at the halt point or at the point of system deviation would make sense with respect to the rule set.

Nor does the argument establish that the adaptive benefits of reasoning do not outweigh the adaptive costs of using up much of our cortex for explicit variable binding. From an evolutionary standpoint, the one time set-up costs involved in building networks able to manipulate variables may have been well worth the price. Only neurophysiologists can tell us whether the brain has a design that accommodates variable binding to a considerable degree.

Nonetheless, I think connectionism justly forces us to reconsider whether there are alternative formulations of the problems the cognitive system solves, formulations that lend themselves to solution without much variable value pairing either explicitly or implicitly.

It is patent that static networks run into trouble as soon as they must represent large numbers of variables or small numbers of variables with large ranges of values. But static networks may be quite adequate in accommodating a reasonable number of variables with small ranges. The trick of rethinking cognition, then, becomes that of finding a way to solve problems with variables that have a restricted value set.

One suggestion[8] is to use indexicals in place of constants. This may allow us to cope with complex situations, by using and reusing a small number of binding slots. The fact that in many situations we do not care whether the object we are dealing with is object-25 or object-26 as long as it 'does the job,' means that we may only have to keep in our minds a small set of variable value pairings.

For example, in setting the table, I am usually indifferent to the token identity of cutlery. I care whether I have laid a fork and knife, but not whether I laid salad-fork-10. This generalization of instances may extend in all directions. Accordingly, on this view the hard part of acquiring a skill is to discover the general properties of the environment that matter in performing the task. These general properties may well be non-standard, and quite unintelligible outside the context of the task. Hence they must be discovered through practice. Once they are learned, however, they have the consequence of simplifying the complexity of the task with respect to the number of variables and values that may be active at any one time.

To take another example, when driving a car, we seldom worry about the token identity of other cars as they approach us. What matters is their relative position, velocity, and so on. It is reasonable to expect a connectionist system to learn the appropriate relations between these n-tuples. And it is reasonable to expect the system to be able to make

appropriate responses. Whether these responses are readily codified by a set of rules is irrelevant to the basic issue. For the real problem was to learn which properties of the world partition the task environment in a tractable manner. That is, what properties simplify the task of driving? These may be arbitrarily task specific. Once we have discovered these specialized properties, it may not be necessary to bind them to many different values. Thus, from the vantage point of driving, it may not be important to be able to identify and reidentify objects in any more satisfying way than as 'the-car-now-coming-at-me-from-the-right.'[9] At times there may be more than one car filling the description. On those occasions the system will have to be able to bind the predicate to separate entities, separate values. Even under such conditions, though, the number of such bindings can be expected to remain small.

It is an empirical matter how far the indexicalization of our knowledge can be pushed. Some tasks lend themselves to this reduction, others resist it. It is one of the virtues of the connectionist approach that it encourages us to explore this problem.

ACKNOWLEDGEMENTS. I would like to thank Phil Agre, David Chapman and Eric Saund for many hours of helpful discussions.

NOTES

[1] "A Method for Connectionist Variable Binding," Proceedings of the American Association of the Artificial Intelligence, 1987.

[2] *Ibid.*

[3] "Symbols Among the Neurons: Details of Connectionist Inference Architecture," in Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, CA.

[4] "BoltzCONS: Reconciling Connectionism with the Recursive Nature of Stacks and Trees," in Proceedings of the Eighth Annual Conference of the Cognitive Science Society, Amherst, MA, 522-530.

[5] "Recursive Auto-Associative Memory: Devising Compositional Distributed Representations." Computing Research Laboratory, New Mexico State University, Las Cruces, MCCS-88-124.

[6] This does not quite duplicate the binding problem in connectionism because the point of binding in PDP is to turn on nodes which have the right overall effect on the system. This may be achieved without actually representing the variable and its value in distinct nodes, as the localist and Smolensky do. Nonetheless, it seems reasonable to assume that if two nodes are on at the same time their joint effect on a network will be different than their separate effects. If this joint effect does not duplicate the overall effect of a variable value binding, we can introduce more nodes, which are appropriately related to the rest of the network and which co-vary with a variable value pairing.

[7] Although it is an open empirical question just how much new training is required to teach an old network to respond appropriately with its new nodes, it is clear that as more new nodes are added the amount of new training required rises non-linearly.

[8] For an interesting first try at explaining a skill as a controlled response to 'indexical-functional' properties, see P. Agre, and D. Chapman, Pengi: "Implementing a Theory of Activity," in Proceedings of the American Association for Artificial Intelligence, 1987.

[9] Cf. ibid.

Cognitive Science
Room D-015
University of California
La Jolla, CA 92093