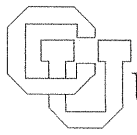
Implementing a Connectionist Production System Using Tensor Products

Charles P. Dolan
Paul Smolensky

CU-CS-411-88



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

To appear in the *Proceedings of the 1988 Connectionist Models Summer School*
Touretzky, Hinton, & Sejnowski (Eds.) Morgan Kaufmann.

Implementing a connectionist production system using tensor products

September, 1988

UCLA-AI-88-15 CU-CS-411-88

Charles P. Dolan
AI Center
Hughes Research Labs
3011 Malibu Canyon Rd.
Malibu, CA 90265
&
UCLA AI Laboratory

Paul Smolensky
Department of Computer Science &
Institute of Cognitive Science
University of Colorado
Boulder, CO 80309–0430

Abstract

In this paper we show that the tensor product technique for constructing variable bindings and for representing symbolic structure—used by Dolan and Dyer (1987) in parts of a connectionist story understanding model, and analyzed in general terms in Smolensky (1987)—can be effectively used to build a simplified version of Touretzky & Hinton's (1988) Distributed Connectionist Production System. The new system is called the Tensor Product Product System (TPPS).

Implementing a connectionist production system using tensor products

Charles P. Dolan
AI Center, Hughes Research Labs
3011 Malibu Canyon Rd
Malibu, CA, 90265,
&
UCLA AI Laboratory

Paul Smolensky
Department of Computer Science &
Institute of Cognitive Science
University of Colorado
Box 430, Boulder, CO 80309-0430

1. Introduction

The ability to represent relationships among entities is a central problem in connectionist models of cognition. The problem has gone under several names including variable binding, role representation, and "thirdness". Smolensky (1987) has shown that many attempts to solve this problem can be analyzed mathematically using tensors, or generalized vector outer products. Dolan and Dyer (1987, in this volume) have applied tensorial representations to demonstrate parts of a connectionist story understanding model.

What has not been shown, however, is whether explicitly adapting representations currently in use to a tensorial scheme actually buys anything in a model. Aspects of a model that one would hope to improve by using a mathematically based approach might be: (1) easier analysis of results, (2) analytical predictions of model performance, and (3) more straightforward construction of a simpler model. The purpose of this paper is to take an existing model that solves some aspects of the variable binding problem, Touretzky & Hinton's (1988) connectionist production system, and show how some of the advantages listed above can accrue from building a closely related model by straightforward application of the tensorial representation. This work demonstrates (3), and previously developed techniques from (Smolensky 1987) could be used to make progress on (1) and (2).

1.1. Tensor products

The tensor product can be straightforwardly understood as a generalization of the outer product of two vectors. Given two column vectors, x and y, the inner product, $x^T y$ is the familiar "dot" product $\mathbf{x}\cdot\mathbf{y}$. The outer product, xy^T, is simply the familiar matrix multiplication which takes a column vector and a row vector and yields a matrix. The *ij* element of this matrix is $x_i y_j$. This outer product operation can also be viewed as a tensor product which is written $\mathbf{x}\otimes\mathbf{y}$. Thus the matrix xy^T can be viewed as a tensor with two indices, or a tensor of *rank two*. A vector is a tensor with one index, or a tensor of rank one; and a simple scalar is a tensor of rank zero. Similarly, there are tensors of rank higher than two, with more than two indices; these can be generated by taking the tensor product of more than two vectors.

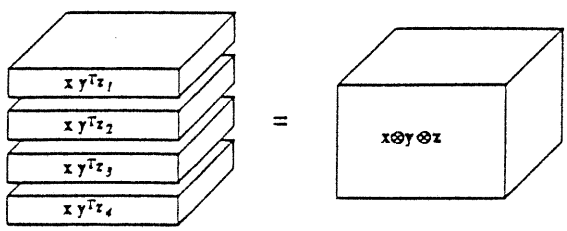


Figure 1: Building a third-rank tensor from three vectors.

If all we ever needed was a 2nd order tensor we could stay with familiar matrix notation. However, the major demonstration of the paper requires 3rd order tensors and so we shall use the more general apparatus of tensor algebra. Figure 1 demonstrates how to view the third order tensor $x \otimes y \otimes z$. Simply take the elements of $x \otimes y$, a familiar matrix, and form planes of them, one for each element of z: in each plane i, the matrix $x \otimes y$ is multiplied by the scalar z_i.

To use tensors to represent structure, we decompose a structure into a set of filler/role pairs and then use a sum of tensor products as the representation of the structure. Formally this can be stated as follows:

Let a set S of structured objects be given a *role decomposition*: a set of fillers, F, a set of roles, R, and for each object s a corresponding set of role/filler bindings:

$$\beta = \bigcup_i \{(r_i \; f_i)\}.$$

Let a connectionist representation of the fillers F be given; each f_i is represented by the activity vector $\mathbf{f}_i$.

Let a connectionist representation of the fillers R be given; each r_j is represented by the activity vector $\mathbf{r}_j$.

Then the corresponding tensor product representation of s is,

$$\mathbf{B} = \sum_i \mathbf{r}_i \otimes \mathbf{f}_i$$

A number of the general properties of tensor product representations are analyzed in (Smolensky 1987). One of them is that they can be recursively imbedded. This leads to overall representations which are tensors of rank higher than two; essentially, each level of imbedding adds a rank to the overall tensor.

To see how a tensor product can be used to represent symbolic information, we only need to realize that most traditional symbolic representations can be broken down into triples of symbols. For example a frame, s with slots, r_i and fillers f_i can be represented with a set of triples, $(s \; r_i \; f_i)$. By establishing a vector representation for frames, slots, and fillers, we can represent a frame by $\Sigma_i \mathbf{s} \otimes \mathbf{r}_i \otimes \mathbf{f}_i$. Examples of decomposing frames into triples can be found in (Dolan and Dyer, in this volume).

1.2. DCPS as a tensor product

One model, which at first glance seems not to be using tensor products, is the distributed connectionist production system (DCPS) (Touretzky & Hinton 1988). DCPS uses an alphabet of 25 symbols, A-Y, and its rules use triples of symbols that are coarse coded. To construct the representation, a pool of 2000 units is used. Each unit has a receptive field table associated with it. An example receptive field is shown in Figure 2.

A unit is part of the representation of a triple if and only if its receptive field table has the 1st, 2nd, and 3rd elements of the triple in its 1st, 2nd and 3rd columns, respectively. For example, the unit with the receptive field in Figure 2 is part of (C A B), (C A D), and (M E D), but it is not part of (C A C) or (G I L). The representation of a triple is the set of all units that have that triple in their receptive field. In the version of DCPS reported in (Touretzky & Hinton 1988), each table had six rows and each triple was represented by the activity of about 28 units.

To see how this representation can be analyzed with the tensor product, we will use a diagonalizing procedure first reported in (Smolensky 1987). We will demonstrate the procedure using pairs of symbols, rather than the triples of DCPS, since that is easier to visualize; the analysis extends immediately to the triple case using third-rank tensors.

We first note that with respect to DCPS the symbols in different columns do not interact and an "A" in the first column really bears no meaningful relation to an "A" in the second column. Therefore a pair can readily be viewed as something like $(A_1 \; B_2)$ or $(G_1 \; A_2)$. Given a set of N receptive field tables (2000 in DCPS) we can form a matrix where the rows and columns are labeled by the columns of the receptive field tables. The matrix is NxN because there are N independent receptive field table, and the tables are Kx2, where K is a "coarseness" parameter.

C	A	B
F	E	D
M	H	J
Q	K	M
S	T	P
W	Y	R

Figure 2: Example Receptive field table

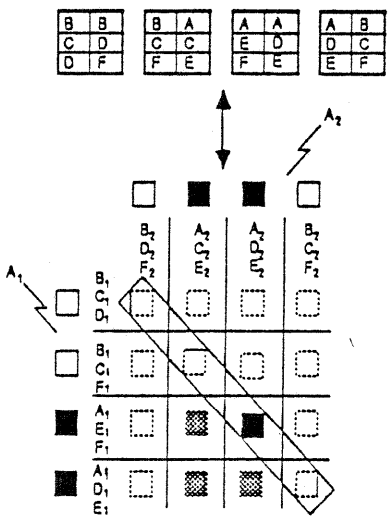


Figure 3: Analyzing DCPS's representation as a tensor product

Now we can represent a symbol ξ_1 by the N bit pattern consisting of active units for each of the tables in which ξ_1 is in column 1, and likewise for any symbol ξ_2. This procedure is demonstrated in Figure 3, where both representations of (A A) are shown with DCPS on top of the tensor product. The grey squares are the active units for the tensor product and the black square is the active unit used by DCPS. In the example shown in the figure, the representations for A_1 = (0 0 1 1) and A_2 = (0 1 1 0) are derived by labeling the rows and columns of the matrix with the first and second columns of the receptive field tables. Using this representation we can view DCPS as using a tensor product representation with 2000 bit symbols. The reason that this does not produce an unreasonable number of units in the working memory representation is that all but the diagonal elements are discarded. In Figure 3, diagonal elements are the four outlined units shown on top of the tensor product representation.

On the average, each symbol would be represented by a 2000 bit vector containing 2000×6/25 = 480 active bits since each receptive field table of DCPS had 6 rows and there are 25 symbols. There are a tremendous number of 480 out of 2000 bit patterns, approximately 2×10^{481}, only 25 of which would be used for each column. This makes DCPS an extremely sparse sampling of that symbol space. In fact those 25 symbols are specially designed so that every three-way conjunction of them has very close to 28 active units. This fact is extremely important to the dynamics of DCPS and finding those special receptive field tables took a considerable amount of computational effort (Touretzky & Hinton 1988).

1.3. The possible advantages of tensors over custom coarse codings

A natural question to ask is whether another approach to the same task might be able to use a dense sampling of the symbol space. The full tensor product has a desirable property that makes it a likely candidate. The property, which is covered in detail in (Smolensky 1987), is the ability to unbind one component of an N^{th} order tensor product given $N-1$ components.

The method of unbinding we will be using here is called the self-addressed technique in (Smolensky 1987). Given a tensor product representation of the triple $(s\ r\ f)$, $s\otimes r\otimes f$, we can unbind the filler f from the frame role it is bound to, $s\otimes r$, by a simple linear computation: $(s\otimes r\otimes f)\cdot(s\otimes r) = \alpha f$ where α is a constant magnitude factor equal to $(s\cdot s)(v\cdot v)$. If we have a superimposed tensor product representations of multiple slot/filler bindings, we can still perform unbinding using self addressing. Now, however, we will get a result that has components in the direction of other fillers. For example, if we try to unbind the filler of $s_k\otimes r_k$ from the superimposed tensor product $\Sigma_i s_i \otimes r_i\otimes f_i$ we will get $\Sigma_i(s_i\cdot s_k)(r_i\cdot r_k)f_i$. If all the different s_i's are orthogonal and likewise for the r_i's, then we are guaranteed to still get αf_k as above.

More generally, if the various s_i's have the same length, and similarly for the various r_i's, then this unbinding procedure will produce a weighted superposition of the f_i's in which f_k has the largest weight. However, in a densely sampled symbol space there may be another filler symbol f_j with a connectionist representation f_j which is closer to the unbound pattern than is f_k.

From the above discussion, one might surmise that using the tensor product representation on densely sampled symbol spaces is not a workable solution. However, it may happen that we are simultaneously trying to satisfy multiple retrieval cues rather than a single one (i.e., if we are looking for something that simultaneously fills multiple roles). This extra constraint can actually *enhance* retrieval. This is exactly the situation in the retrievals needed for the productions used in DCPS, where the conditions require unbinding a filler that simultaneously fills two different roles. We shall see below how these multiple constraints can be exploited.

2. DCPS redone with tensor products: TPPS

We now describe a connectionist production system based on tensor product representations, TPPS, that, like DCPS, operates with productions of the form

$$(s_1\ r_1\ x)\ (s_2\ r_2\ x) \Rightarrow +(s_3\ r_3\ x)\ -(s_4\ r_4\ x)$$

The condition side consists of two triples, each of which contains a common variable in the final position. The action side consists of a triple to add and a triple to delete, and these triples also contain the variable from the condition side in their third position. As in DCPS, there were 25 distinct symbols that could occupy each of the three positions in the triples. Actually both the DCPS and TPPS architectures will work with less restrictive rule formats. Other rules with more than two actions per rule and arbitrary placement of variables in the action portion have been demonstrated with DCPS.

2.1. *Working memory*

The working memory (WM) of TPPS is a network containing a representation of a set of triples $\cup_i\{(s_i\ r_i\ f_i)\}$; the representation is a third-order tensor product $\mathbf{B} = \sum_i \mathbf{s}_i \otimes \mathbf{r}_i \otimes \mathbf{f}_i$. In our simulation, the vectors chosen to represent the elementary symbols s_i were 7-bit vectors consisting of three 1s and four 0s. They were chosen so that no two vectors had more than two 1-bits in common (i.e., the dot-product between any two was at most 2); otherwise the vectors were random. The same procedure was used to assign 7-bit vectors to the r_i, and again to the f_i. There was no reason to be concerned about the relation between vectors representing different types of symbols, (e.g., one of the s_i and one of the f_i) conceptually, they belong to different 7-dimensional vector spaces. Since WM contains tensor products of three 7-bit vectors, it consists of $7^3 = 343$ units.

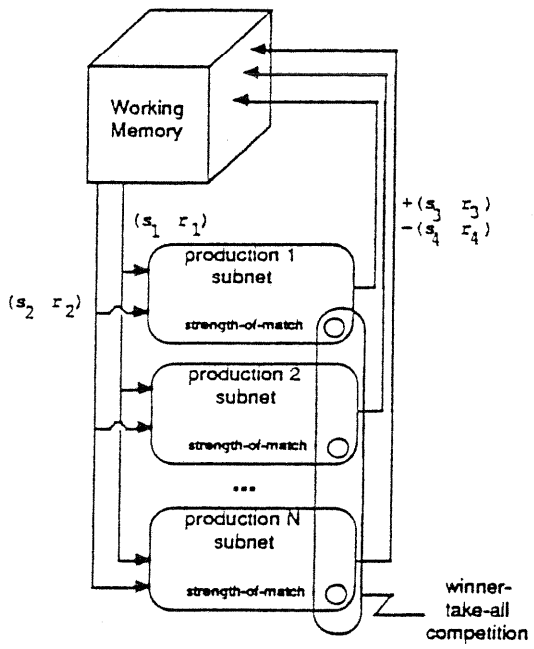


Figure 4: The architecture of TTPS

2.2. *The architecture*

In TPPS, each production corresponds to a separate sub-network. The details of the connections for these sub-networks will be provided in the next section; here we intend only to indicate their qualitative structure. The sub-network for a given production consists of a set of units for holding possible values of x for matching the first triple in the condition, another set of units for holding the corresponding values for the second triple, and a third set of units for building a common value across the two triples. We call these three groups of units the x_1 units, the x_2 units, and the x units, respectively. In our simulation, each group contained 7 units. The connections from WM into the x_1 units encode the pattern $(s_1\ r_1)$ from the first triple; the connections from WM into the x_2 units encode the pattern from the second triple $(s_2\ r_2)$. The connections from the x_1 and x_2 units into the x units are the same for all productions. There is an additional unit in each production sub-network; it registers how strongly that production has matched to WM. These strength-of-match units for all productions are connected in a winner-take-all network, and the production with the strongest match is permitted to send activation back to WM to add and delete the appropriate triples. The connections into WM from the x units in a production sub-network encode the patterns $(s_3\ r_3)$, $(s_4\ r_4)$ from the action side of that production. In order to minimize the propagation of noise into WM, before firing, the production sub-network cleans up its representation of x. The top level organization of the architecture is shown in Figure 4.

The processing in TPPS proceeds as follows. The production sub-networks in parallel perform matching to WM, with feed-forward activation passing in parallel from WM to the x_1 and x_2 units, then to the x units and then to the strength-of-match unit. The winner-take-all competition between the strength-of-match unit achieves best-match conflict resolution, where all the strength-of-match units are driven to zero activity except the most active. While this conflict resolution is going on, each

production sub-network is cleaning up the pattern in its x units, generating a noise-free pattern in another group of units called the x^* units. When the most active strength-of-match unit has been selected, it opens the gated connections between its x^* units and WM. The contents of WM are updated and the cycle begins again.

2.3. *Representation of productions' conditions*

The condition side of each production is encoded in the connections from WM to the x_1 and x_2 units. The connections from WM to the x_1 units perform a linear operation: from B, the state of WM, they set up in the x_1 units the pattern $\mathbf{B} \bullet (\mathbf{s}_1 \otimes \mathbf{r}_1)$. Thus the x_1 units are purely linear, and the connection from WM unit ijk to x_1 unit m is $(\mathbf{s}_1)_i (\mathbf{r}_1)_j \delta_{km}$ (δ_{km} is 1 if $k = m$, otherwise it is 0). The story for the connections from WM to the x_2 units is analogous.

Thus the pattern in the x_1 units indicates those symbols x_1 corresponding to triples $(s_1\ r_1\ x_1)$ present in WM. If there are no such triples, the pattern in the x_1 units will be approximately zero; if there is one such triple, a pattern approximating the corresponding value of $\mathbf{x}_1$ will be found in the x_1 units. If there are several such triples, the superposition of the patterns representing these different possible values of $\mathbf{x}_1$ will be found.

2.4. *Variable binding*

At this point we have done variable binding separately for the two triples in the condition of the production: the separate results are held in the x_1 and x_2 units. We now try to extract from these the representation of a common value in the x units. The feed-forward connections from the x_1 and x_2 units to the x units take the vectors $\mathbf{x}_1$ and $\mathbf{x}_2$ and generate the vector $\mathbf{x}$ by the bilinear computation:

$$\mathbf{x} = \mathbf{x}_1 \blacklozenge \mathbf{x}_2 \ .$$

The $\blacklozenge$ operation is component-wise multiplication: the activity of the m^{th} x unit is the activity of the m^{th} x_1 unit times the activity of the m^{th} x_2 unit. This multiplication can be achieved with Hinton's (1981) triangular multiplicative junctions or with sigma-pi (Rumelhart, Hinton & McClelland, 1986) units.

The $\blacklozenge$ operation can be used to seek a common binding for x because it is a bilinear function with the property that if $\mathbf{x}$ is a Boolean vector (its components are all 0s and 1s) then $\mathbf{x} \blacklozenge \mathbf{x} = \mathbf{x}$. On the other hand, if $\mathbf{x}$ and $\mathbf{y}$ represent two different symbols, $\mathbf{x} \blacklozenge \mathbf{y}$ will be close to zero (its length is $\mathbf{x} \bullet \mathbf{y}$, which is close to zero since the vectors representing distinct symbols are nearly orthogonal). Thus, for example, suppose WM contains two possible bindings, a and b, for x in the first triple of a production's condition, and two possible bindings, a and c, for the second triple. Then we will have (ignoring multiplicative scale factors):

$$\begin{aligned}
\mathbf{x}_1 &\equiv \mathbf{a} + \mathbf{b} \\
\mathbf{x}_2 &\equiv \mathbf{a} + \mathbf{c} \\
\mathbf{x} &= \mathbf{x}_1 \blacklozenge \mathbf{x}_2 \\
&= (\mathbf{a} + \mathbf{b}) \blacklozenge (\mathbf{a} + \mathbf{c}) \\
&= \mathbf{a} \blacklozenge \mathbf{a} + \mathbf{a} \blacklozenge \mathbf{c} + \mathbf{b} \blacklozenge \mathbf{a} + \mathbf{b} \blacklozenge \mathbf{c} \\
&\equiv \mathbf{a} \ .
\end{aligned}$$

Since the vectors representing different symbols are not completely orthogonal, the resulting vector $\mathbf{x}$ will contain noise. TPPS has a clean-up circuit in each production's sub-network that takes the noisy vector $\mathbf{x}$ and replaces it with a vector $\mathbf{x}^*$ that is the symbol vector closest to $\mathbf{x}$. The current version does this with a simple local competition: there is one unit for each possible symbol, and each receives feed-forward activity from $\mathbf{x}$ equal to the dot product of $\mathbf{x}$ with that symbol's vector. A simple winner-take-all feedback system connecting these symbol units selects the most active unit, i.e., the symbol closest to $\mathbf{x}$, and this unit then sends feed-forward activity to the x^* units. The connection from the m^{th} x unit to the unit for a symbol a is the m^{th} element of the vector representing a, and this is also the strength of the connection from the a unit to the m^{th} x^* unit. The symbol units and the x^* units are purely linear.

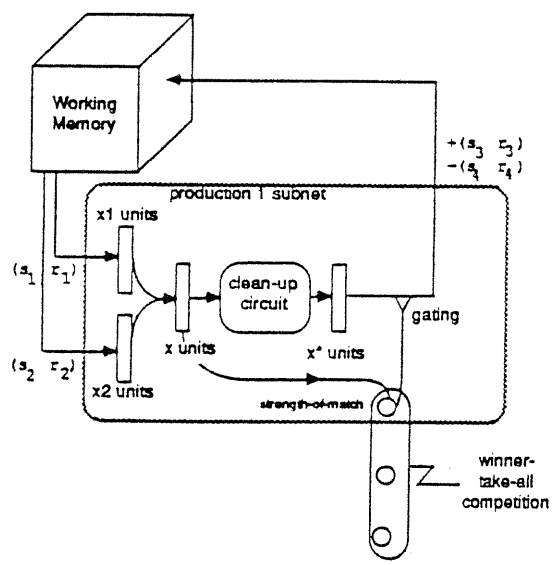


Figure 5: A production sub-network

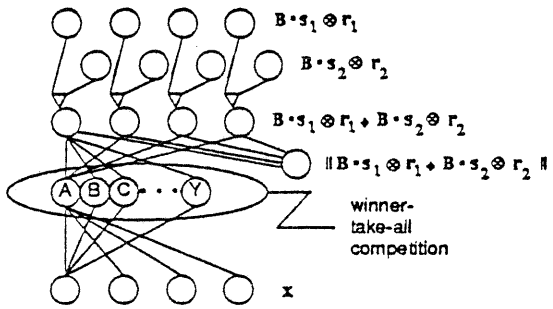


Figure 6: Detail of the clean-up network

Note that the clean-up circuit is in a position not only to eliminate noise, but also to choose between multiple possible of values for x, should WM support more than one simultaneous binding for both triples in a production's condition. To achieve this, the winner-take-all circuit needs some way of breaking a tie when two symbols match x equally well. This can be achieved by adding a small amount of noise to winner-take-all competition.

Thus we see, as mentioned above, that noise and ambiguity problems that may arise in performing a single unbinding can be ameliorated by combining multiple unbindings representing multiple simultaneous constraints on the retrieved item. In fact, the more knowledge we have about what we want to retrieve, the clearer the result will be.

2.5. Best-match conflict resolution

As explained in the preceding section, if there is no way to jointly bind x in the two condition triples, there will be a weak noisy pattern in the x units. Thus the strength of the pattern in the x units can serve as a measure of how well the production's condition matches WM. This part of the architecture is shown in Figure 5.

Each production sub-network contains a unit whose value is the squared length of the vector in the x units. This strength-of-match unit sums the squares of the activities of all the x units. As with the $\blacklozenge$ operation, this can be implemented either by making the strength-of-match unit a sigma-pi unit or by making it a linear unit and using multiplicative triangle junctions. The details of the clean up circuit are shown in Figure 6.

A winner-take-all circuit connecting the strength-of-match units for all productions then chooses the most active unit — the best-matched production, according to TPPS's measure of quality of match.

2.6. Firing a production; representation of productions' actions

The winning strength-of-match unit now gates open connections from its x^* units to WM. These connections use the x^* vector representing the selected value for x to build the patterns representing the two triples on the action side of the production, $(s_3\ r_3\ x)$ and $(s_4\ r_4\ x)$, and add the first and subtract the second from WM. This amounts to changing the vector B in WM by adding $s_3 \otimes r_3 \otimes x^* - s_4 \otimes r_4 \otimes x^*$. Thus the connection from the m^{th} x^* unit to the ijk element of WM is

$$(s_3)_i\,(r_3)_j\,\delta_{km} - (s_4)_i\,(r_4)_j\,\delta_{km} \quad .$$

The units in WM are purely linear.

2.7 Comparison with the DCPS architecture

Figure 7 shows the top level architecture of DCPS form (Hinton and Touretzky 1988). Both architectures use a set of units for working memory. DCPS uses binary working memory units where superimposed representations are inclusively ORed together. TPPS uses linear working memory units where superimposed representations are added together. TPPS used six different activity levels for working memory elements (0.0, 0.2, 0.3, 0.6, 0.8, 1.0). DCPS used clause spaces to extract triples from working memory. TPPS does not explicitly extract triple representations from working memory. In DCPS, the units in the rule space all share the same bind space during the competitive match. In addition, all the effect of variable binding on the rules passed through the clause

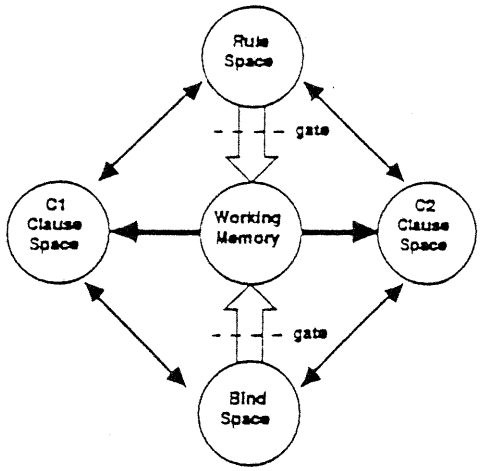


Figure 7: DCPS architecture

spaces. In TPPS the clean-up units perform a function similar to the DCPS bind space. The bind in TPPS space is smaller than in DCPS (equal to the number of symbols) but it is duplicated in every production sub-network. In addition, each rule connects directly to its binding units. In DCPS, the rule space is a set of winner-take-all cliques. In TTPS, the strength-of-match units also engage in winner-take-all competition. In summary the major differences at the architectural are: (1) DCPS and TTPS use different encodings of triples, and (2) because TTPS does not use clause spaces is it almost completely feed forward (except at the end) whereas DCPS uses a competitive matching strategy throughout the course of rule matching.

3. Results and discussion

We tested TPPS on one of the test rule sets from DCPS:

```
(A A ?)  (B B ?)  ⇒  -(A A ?)  +(C C ?)
(B B ?)  (C C ?)  ⇒  -(B B ?)  +(D D ?)
(C C ?)  (D D ?)  ⇒  -(C C ?)  +(E E ?)
(D D ?)  (E E ?)  ⇒  -(D D ?)  +(F F ?)
(E E ?)  (F F ?)  ⇒  -(E E ?)  +(A A ?)
(F F ?)  (A A ?)  ⇒  -(F F ?)  +(B B ?)
```

This rule set was allowed to cycle, starting from the state of having only (A A X) and (B B X) in working memory and is able to run indefinitely without misfiring.

To test for robustness to cross-talk among the symbols, we also ran TPPS with noise; we added one random triple of symbols after each production firing. Note that since our symbols are represented by 7-bit Boolean vectors of length 3, of which there are only 35, we are using a fairly dense sampling of the symbol space, using 25/35 = 71% of the possible symbol vectors (as compared to $1x10^{-480}$% used by DCPS). Thus in adding noise triples of such densely packed symbol vectors, we are introducing quite a high level of cross-talk across symbol vectors.

The average number of productions fired before error was 9 and the standard deviation was 3. An error was defined as either a production firing out of order or getting an incorrect binding on the output. In most cases, even after the first error, the production system was able to pick up the sequence again, either using the correct symbol on the output, X, or a symbol which had a very similar representation. (Note that for a symbolic system, the expected number of error-free production cycles under the same conditions is very large; the probability of randomly generating a triple that will cause the wrong production to fire in collaboration with one of the legitimate triples is $6/25^3 = 4 \times 10^{-4}$ so the expected number of correct firings before such a misfire is about 2500.)

These results suggest that tensor operations can compete with custom coarse codings in their ability to represent a modest number of active elements in working memory chosen out of large number of possible elements. This capability is realized by allowing multiple concurrent unbindings or queries to produce an unambiguous result where a single unbinding might have been impossible to interpret.

Another advantage of this representation is that it allows a much denser sampling of the symbol space than other representations. This is extremely beneficial when we also want to use the bit vector representations of symbols as feature vectors in another part of a model. Less sparse representations of structures can also be used. In DCPS, the fraction of WM cells involved in representing each triple is 28/2000 = 1%, while in TPPS it is $(3/7)^3 = 8\%$ (the ratio of 1:8 is the cube of the ratio of the fraction of active elements in the vectors representing symbols: 6/25 = 24% for DCPS, 3/7 = 43% for TPPS).

The main difference in design of the representation between DCPS and TPPS is that in DCPS the representation was designed at the level of *whole structures* — triples — while in TPPS it was designed at the level of the atomic constituents — symbols — and combined with a simple, general-purpose scheme for building representations of structures from the representation of the constituents. In the tensorial approach, the computational adequacy of the patterns for the structures is a consequence of the adequacy of patterns for the symbols, which is easy to ensure: it suffices that they not be too close in their small vector space. "Mind the symbols and the structures will take care of themselves." In DCPS, on the other hand, the acceptability of the representation of the triples has to be built using considerable computational effort. The representation of the triples did not derive in any general and well-motivated way from the representations of their constituents (indeed, it is only through some kind of rational reconstruction like that presented in the beginning of this paper that we can view the distributed patterns for the triples as deriving in any way from a distributed representation of the constituents).

In addition to its representational advantages, the tensorial approach can also lead to simpler network dynamics. In the TPPS, the network is strictly feed-forward using linear units, with the exception of two places: (1) the winner-

take-all competition among the rules in Figure 4 and (2) the winner-take-all competition among the symbols in each rule in Figure 5. This keeps the feedback within a module (selecting a variable) independent of the feedback between modules (selecting a rule). The two way feedback connections between the rule and clause spaces in DCPS cause it to have more complex dynamics. On the other hand, the settling process of DCPS achieves variable binding and conflict resolution in parallel, while TPPS performs conflict resolution only after all productions have attempted — in parallel — variable binding. In practice, however, TPPS settles in less than 10 steps.

The last point we want to make about tensor representations is that they are easy to manipulate. This makes both the design and the analysis of resulting system much easier than with custom coarse codings. For this reason, tensor products are likely to be a good first choice for any compositional representation problem. In this case, a straight forward application of the tensor product did quite well, but even in other cases where the tensors products model might not work as well as one would like, it is likely to be a good first cut from which custom design can then progress.

Acknowledgements

The core of this work was carried out at the connectionist summer school and we thank the supporters (AFOSR, AAAI, SIGART, and the Sloan Foundation) and organizers of the summer school. We thank the members of the summer school's "tensor product working group", especially Dave Touretzky, for very helpful conversations and many helpful comments on an early draft of this paper.

This work has been partially supported by NSF grants IRI-8609599 and ECE-8617947 to the second author, and by a grant to the second author from the Sloan Foundation's computational neuroscience program.

References

Dolan, C. P. and Dyer, M. G. (1987). Symbolic Schemata, Role Binding, and the Evolution of Structure in Connectionist Memories. *Proceeding of the First International Conference on Neural Networks*, San Diego, CA, Volume II, 287-298.

Dolan, C. P. and Dyer, M. G. (1988). Parallel Retrieval of Conceptual Knowledge, *Proceedings of the 1988 Connectionist Summer School.*

Hinton, G.E. (1981). A parallel Computation that Assigns Canonical Object-based Frames of Reference. *Proceedings of the 7th International Joint Conference on Artificial Intelligence.*

Rumelhart, D. E., Hinton, G. E, & McClelland, J. L. (1986). A Framework for PDP. In D. E. Rumelhart, J. L. McClelland, & the PDP Group, *Parallel Distributed Processing: Exploration in the Microstructure of Cognition.* Volume 1: Foundations.

Smolensky, P. (1987). One Variable Binding and the Representation of Symbolic Structures in Connectionist Systems. Technical Report CU-CS-355-87, Department of Computer Science, University of Colorado Boulder.

Touretzky, D. S. and Hinton, G. E. (1988). A Distributed Connectionist Production System. *Cognitive Science*, 12(3), 423-466.