Tensor Product Networks

Aims

- to study a network with linear activation, multiple activation modes, and 1-shot learning
- to see tensor product networks in action, modelling human analogical reasoning
- to explore robustness properties using a range of distributed representation strategies

Reference


Other references in body of notes.

Keywords

tensor product network, variable binding problem, rank, one-shot learning, orthonormality, relational memory, teaching/retrieval modes, proportional analogies, lesioning a network, random & sparse random representations, fact recognition scores, representable non-facts

Plan

- activation functions (linear/non-linear)
- variable binding / rank 2 tensor product net
- teaching and retrieving from a TPN
- orthonormal / distributed representations
- TPNs as relational memories
- rank 3 tensor product nets
- modelling proportional analogy-solving with TPNs
- graceful degradation in TPNs
- random representations
- sparse random representations
- fact and non-fact recognition scores

**Tensor Product Networks**

<table>
<thead>
<tr>
<th><img src="image" alt="Tensor Product Networks" /></th>
<th><img src="image" alt="Network Topology and Activation" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>Feedforward Nets</td>
<td>Network Topology and Activation</td>
</tr>
<tr>
<td>• A net is a feedforward net if the the graph consisting of the neurons as nodes and connections as directed edges is a directed acyclic graph.</td>
<td>There are several ways to connect mini-processing units together and several models of computation for the units. A common model of computation is to assume weighted connections $w_{ij}$ from input units with activation $x_j$ to unit $i$, and to define the output (activation function) for unit $i$ to be: $\sigma(\sum_j w_{ij}x_j)$ where $\sigma$ is a ‘squashing’ function such as $\tanh$. This is the model used in feedforward networks which learn by error backpropagation.</td>
</tr>
<tr>
<td>• One can then find nodes with in-degree zero (no connections coming into them) = input nodes; and nodes with out-degree zero (no connections leaving them) = output nodes; anything else is termed a hidden node or unit.</td>
<td></td>
</tr>
<tr>
<td>• Nets like this are often drawn as follows:</td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Feedforward Nets" /></td>
<td><img src="image" alt="Network Topology and Activation" /></td>
</tr>
<tr>
<td>where each rectangle represents a group of neurons/nodes: the input nodes are on the left, output nodes on the right, and hidden units in the middle.</td>
<td></td>
</tr>
<tr>
<td>• The arrows with $\sigma$ signify that there are connections, with trainable weights, between all neurons in one layer and all those in the next.</td>
<td></td>
</tr>
</tbody>
</table>

**Fully Recurrent Nets**

Another possibility for connectivity (also called topology) of networks is total interconnection - every neuron has a weighted connected to every other neuron. The connections may or may not be trainable. Such nets are called fully recurrent.

![Fully Recurrent Nets](image)

Totally connected nets are rather undisciplined. Backpropagation won't work on them, as the problem of finding the error derivatives is too non-linear. (But see also backpropagation-through-time.)
Tensor Product Nets

- Other possibilities for the activation function include linear networks (where $\sigma$ is the identity function, i.e., the output of the nodes is $\sum_j w_{ij}x_j$).
- One particular kind of network with a linear activation function and a special topology is the rank 2 tensor product network.
- Tensor product networks were applied early to the task of dealing with symbolic structures in connectionist systems, by Paul Smolensky.
- Smolensky (1990) deals with the variable binding problem - how to represent concepts such as $x = 3$ in a connectionist system. This can be achieved with a rank 2 tensor product network.

Rank 2 Tensor Product Nets

- Tensor product nets come with different numbers of dimensions, or rank. The first interesting case is rank 2, where the topology is that of a matrix.
- The network below is shown in teaching mode. There is also a retrieval mode, where you feed the net (the representation of) a variable, and it outputs (the representation of) the symbol (the filler).
- From here on, we shall mostly omit the phrase “the representation of” and so you will need to figure out from context if we are talking about the the thing represented, or the representation.

Paul Smolensky

Paul Smolensky developed tensor product networks in about 1989, to counter the claim that neural network approaches to AI couldn’t deal with symbol processing. He has also worked on theoretical approaches to linguistics, notably a novel formal characterization of the complex rule systems of grammar called Optimality Theory. Smolensky was awarded the Rumelhart Prize in 2005. He is at John Hopkins University, Baltimore, Maryland.
Teaching Mode

• In teaching mode, when (vectors representing) a variable and a filler are presented to the two sides of the network, the fact that the variable has that filler is learned by the network.

• The teaching is one-shot, as opposed to the iterative learning used by backpropagation networks, and the settling schemes used by other classes of neural network. Nothing is annealed or repeatedly adjusted, and no stopping criterion applies.

• Teaching is accomplished by adjusting the value of the binding unit memory. Specifically, if the \( i \)-th component of the filler vector is \( f_i \) and the \( j \)-th component of the variable vector is \( v_j \), then \( f_i v_j \) is added to \( b_{ij} \), the \((i,j)\)-th binding unit memory, for each \( i \) and \( j \).

• Another way to look at this is to regard the binding units as a matrix \( B \), and the filler and variable as column vectors \( f \) and \( v \). Then what we are doing is forming the outer product \( f v^T \) and adding it to \( B \):

\[
B' = B + fv^T
\]

Retrieval Mode

• For exact retrieval, the vectors used to represent variables must be orthogonal to each other (i.e. any two of them should have the dot product equal to zero) and the same must be true for the vectors used to represent the fillers. Each representation vector should also be of length 1 (i.e. the dot product of each vector with itself should be 1).

• It is common to refer to a set of vectors with these properties (orthogonality and length 1) as an orthonormal set.

• Orthonormality entails that the representation vectors are linearly independent, and in particular, if the matrix/tensor has \( m \) rows and \( n \) columns, then it can represent at most \( m \) fillers and \( n \) variables.

Orthonormal Bases

It is easy to find an orthonormal set of vectors: the standard basis of \( \mathbb{R}^n \) is orthonormal:

\[
\begin{pmatrix}
1 & 0 & 0 & \ldots & 0 \\
0 & 1 & 0 & \ldots & 0 \\
0 & 0 & 1 & \ldots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \ldots & 1
\end{pmatrix}
\]

This basis has an advantage and a disadvantage. The advantage is that, using it, it is easy to see what is going on in the tensor. If there is a 1 in row \( i \) and column \( j \), then variable \( j \) has value equal to filler \( i \). There can be at most one 1 in each column (a variable has at most one filler) but there can be several 1s in a row (in which case several variables have the same filler).

The disadvantage is that it corresponds to a localist representation - where the responsibility for knowing that variable \( j \) has filler \( i \) resides in the \((i,j)\)th cell of the tensor alone. If the \((i,j)\)th cell were damaged, the fact would be lost from memory.

Distributed Representations in TP Nets

• We shall call a basis a distributed representation if its vectors have no zeroes or few zeroes in them.

• If a distributed representation is used, then the memory for the fact that variable \( j \) has filler \( i \) is distributed all over the tensor.

• Here is a small, simple example of a distributed representation whose rows form an orthonormal set:

\[
\begin{pmatrix}
0.5 & 0.5 & 0.5 & 0.5 \\
0.5 & -0.5 & 0.5 & -0.5 \\
0.5 & 0.5 & -0.5 & -0.5 \\
0.5 & -0.5 & -0.5 & 0.5
\end{pmatrix}
\]
Retrieval from a TP Net

• Retrieval is accomplished by computing dot products. To retrieve the value/filler for a variable $v = (v_j)$ from a rank 2 tensor with binding unit values $b_{ij}$, compute $f_i = \sum_j b_{ij} v_j$, for each $i$. The resulting vector $(f_i)$ represents the filler.

• Notice that the activation function for $f_i$ is a linear activation function.

• To decide whether variable $v$ has filler $f$, compute $D = \sum_i f_i$, $D$ will be either 1 or 0. If it is 1, then variable $v$ has filler $f$, otherwise not.

Hadamard Matrices and Orthonormal Bases

• Where can we get a supply of orthonormal sets of vectors with no non-zero entries?

• Hadamard matrices can help out here.

• A Hadamard matrix is an $n \times n$ matrix $H$ such that $HH^T = nI_n$, where $I_n$ is the $n \times n$ identity matrix, and all of whose entries are ±1.

• The simplest interesting such matrix is:

\[
\begin{bmatrix}
1 & 1 \\
1 & -1
\end{bmatrix}
\]

Another example can be found by doubling the vectors on the last page but 1 and using them as the rows of a matrix.

• Hadamard matrices have the property that their rows are orthogonal to each other, and of length $\sqrt{n}$.

• Hence if you take the rows of a Hadamard matrix and divide them by $\sqrt{n}$, you have an orthonormal set of vectors with all non-zero entries.

Hadamard Matrices and Kronecker Products

• The Kronecker product of two matrices, of dimensions $m \times n$ and $p \times q$ is a matrix of dimensions $mp \times nq$.

• The Kronecker product of two Hadamard matrices is a Hadamard matrix.

• Since there is a $2 \times 2$ Hadamard matrix, Kronecker products can be used to construct Hadamard matrices of side $2^s$, for any positive integer $s$.

• The Kronecker product of $A = (a_{ij})$ with $B$ is

\[
\begin{bmatrix}
a_{11}B & a_{12}B & a_{13}B \\
a_{21}B & a_{22}B & a_{23}B \\
a_{31}B & a_{32}B & a_{33}B
\end{bmatrix}
\]

• For example,

\[
\begin{bmatrix}
1 & 1 \\
1 & -1
\end{bmatrix} \times \begin{bmatrix}
1 & 1 \\
1 & -1
\end{bmatrix} = \begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & -1 & 1 & -1 \\
1 & -1 & -1 & -1 \\
1 & -1 & 1 & -1
\end{bmatrix}
\]

• Hadamard matrices do not exist for some sides that are not powers of 2.

• Notice that the outer product of two vectors is actually the Kronecker product of those vectors regarded as an $n \times 1$ matrix (column vector) and a $1 \times n$ matrix (row vector).
Tensor Product Representations

- Suppose we are using representations as follows:
  - \((+0.5, +0.5, +0.5)\) to represent \(rabb\)
  - \((-0.5, -0.5, 0.5)\) to represent \(mo\)
  - \((+0.5, -0.5, -0.5)\) to represent \(car\)
  - \((-0.5, -0.5, 0.5)\) to represent \(ca\)

  and we want to build a tensor to represent the pair \((rabbit, carrot)\) and \((cat, mouse)\)

\[
\text{carrot} \cdot 1 = \left[ \begin{array}{c} 1 \\ \frac{1}{2} \\ -1 \\ 1 \end{array} \right] \times \left[ \begin{array}{c} 1 \\ 1 \\ 1 \\ 1 \end{array} \right] = \left[ \begin{array}{c} 1 \\ 1 \\ 1 \\ 1 \end{array} \right]
\]

and we check that we can recover carrot from this by unbinding with rabbit: we must compute \(f = \sum_j b_{ij} y_j\) where \(b_{ij}\) is the matrix, and \((v_j)\) is the rabbit vector:

\[
f_1 = b_{11}v_1 + b_{12}v_2 + b_{13}v_3 + b_{14}v_4 = \frac{1}{2} \times 1 = 0.5
\]

and similarly, \(f_2 = 0.5, f_3 = -0.5, \) and \(f_4 = -0.5\), so that \(f\) represents carrot.

Learning with Hadamard Representations

- For \((ca, mou)\), we compute:

\[
mou = \frac{1}{\sqrt{2}} \left[ \begin{array}{cccc} 1 & -1 & -1 & 1 \\ -1 & 1 & -1 & -1 \end{array} \right]
\]

and we check that we can still recover carrot from this by unbinding with rabbit: we must compute \(f = \sum_j b_{ijj} y_j\) where \(b_{ij}\) is the (new) matrix, and \((v_j)\) is the rabbit vector:

\[
f_1 = b_{11}v_1 + b_{12}v_2 + b_{13}v_3 + b_{14}v_4 = \frac{1}{2} \times 1 \times (1 + 1 + 1 + 1) = 0.5
\]

and similarly, \(f_2 = 0.5, f_3 = -0.5, \) and \(f_4 = -0.5\), so that \(f\) represents carrot as before.

TP Nets as Relational Memories

So far we have used tensor product networks to store a particular kind of relational information: variable binding. In variable binding, each variable has a unique filler (at any one time). This restriction on the kind of information stored in the tensor is unnecessary. A rank 2 tensor can store an arbitrary binary relation. For example, take a set of facts about what, say, animals eat:

<table>
<thead>
<tr>
<th>Animal</th>
<th>Food</th>
</tr>
</thead>
<tbody>
<tr>
<td>rabbit</td>
<td>carrot</td>
</tr>
<tr>
<td>mouse</td>
<td>cheese</td>
</tr>
<tr>
<td>crocodile</td>
<td>student</td>
</tr>
<tr>
<td>rabbit</td>
<td>lettuce</td>
</tr>
<tr>
<td>guineapig</td>
<td>lettuce</td>
</tr>
<tr>
<td>crocodile</td>
<td>lecturer</td>
</tr>
</tbody>
</table>

This information can be stored in the tensor in the usual way, (putting the \(animal\) in the side we have been calling \(variable\), and the \(food\) in the \(filler\) side and then we can try retrieving it. For example, we can present the vector representing \(rabbit\) to the \(variable/animal\) side of the tensor. What we get out of the \(filler/food\) side of the tensor will be the sum of the vectors representing the foods that the tensor has been taught that \(rabbit\) eats: in this case \(carrot + lettuce\).

Checking a particular fact, e.g. that \((mouse, cheese)\) is in the relation, is done just as before: we compute \(D = \sum_j b_{ijj} f_j\) where \(v\) is for varmint and \(f\) for food, and if \(D = 1\) then the varmint eats the food.

TP Nets as Relational Memories 2

So far we have used tensor product networks to store a particular kind of relational information: variable binding. In variable binding, each variable has a unique filler (at any one time). This restriction on the kind of information stored in the tensor is unnecessary. A rank 2 tensor can store an arbitrary binary relation. For example, take a set of facts about what, say, animals eat:

<table>
<thead>
<tr>
<th>Animal</th>
<th>Food</th>
</tr>
</thead>
<tbody>
<tr>
<td>rabbit</td>
<td>carrot</td>
</tr>
<tr>
<td>mouse</td>
<td>cheese</td>
</tr>
<tr>
<td>crocodile</td>
<td>student</td>
</tr>
<tr>
<td>rabbit</td>
<td>lettuce</td>
</tr>
<tr>
<td>guineapig</td>
<td>lettuce</td>
</tr>
<tr>
<td>crocodile</td>
<td>lecturer</td>
</tr>
<tr>
<td>tourist</td>
<td></td>
</tr>
</tbody>
</table>

This information can be stored in the tensor in the usual way, (putting the \(animal\) in the side we have been calling \(variable\), and the \(food\) in the \(filler\) side) and then we can try retrieving it. For example, we can present the vector representing \(rabbit\) to the \(variable/animal\) side of the tensor. What we get out of the \(filler/food\) side of the tensor will be the sum of the vectors representing the foods that the tensor has been taught that \(rabbit\) eats: in this case \(carrot + lettuce\).

Checking a particular fact, e.g. that \((mouse, cheese)\) is in the relation, is done just as before: we compute \(D = \sum_j b_{ij} f_j\) where \(v\) is for varmint and \(f\) for food, and if \(D = 1\) then the varmint eats the food.
From what we’ve seen so far, we could better have called these nets ‘matrix nets’. The tensor aspect of things comes in when we generalise to enable us to store ternary (or higher rank) relations.

Suppose we want to store either a ternary relation (like give, where we record who gives, to whom, and what), or a group of binary relations, where we need to record which relation, and the first and second arguments of each relational instance, like:

- `kiss(frank, betty)`
- `hit(max, frank)`

Now we need a tensor net with three sides: say a REL side, an ARG1 side and an ARG2 side, or more generally a u side, a v side and a w side.

This shows the binding units, with shading to show values in their activation memories, and the neurons in the vectors of the u, v, and w sides, but not the interconnections. The net is shown ready for retrieval from the u side, given values for the v and w sides.

The diagram contains 27 binding units, as each side of the tensor has vectors with 3 components. If the u, v, and w representation vectors all have q components, there will be $q^3$ binding units.

Fine Structure of a Rank 3 TP Net

The binding units in the connection diagram below are labelled $t_{ijk}$ - a tensor. Each component of a side, for example $v_1$, is connected to the hyperplane[^1] of binding units $t_{1jk}$, for all i and k.

[^1]: A hyperplane in a vector space or similar structure is a subspace of dimension one less than the whole space. In a space of dimension 3, a hyperplane is just an ordinary plane.

Teaching and Retrieval in a Rank 3 Tensor

**Retrieval:** If we have concepts (or their representations) for any two sides of the tensor, then we can retrieve something from the third side. For example, if we have $u = (u_i)$ and $v = (v_j)$, then we can compute $w_k = \sum t_{ijk} u_i v_j$, for each value of $k$, and the result will be the sum of the vectors representing concepts $w$ such that $u(v,w)$ is stored in the tensor.

This time the activation function for $w_k$ is not linear but multilinear. It roughly corresponds to the activation function for higher-order neural networks, where (biologically speaking) two (or more) axons synapse together (axon-axon-dendritic synapse).

As usual, one can check facts, too. $D = \sum t_{ijk} u_i v_j w_k$ is 1 exactly when $u(v, w)$ is stored in the tensor, and zero otherwise.
Binding Unit Layout
The internal layout of a rank 3 binding unit is more complex, because of the extra input. It is shown below, configured for teaching, and configured for retrieval with w as output:

Higher Rank Tensor Product Networks
Tensor product networks of ranks 4, 5, 6, etc. can be constructed in a manner analogous to the way we went from rank 2 to rank 3.

For a rank r tensor product network:

- the binding units would have r subscripts: $t_{i_1i_2\ldots i_r}$;
- there would be $r$ sides;
- there would be $r$ input/output vectors, say $u_1, u_2, \ldots, u_r$;
- to teach the tensor the fact $u_1(u_2, \ldots, u_r)$, you would add $u_1u_2\ldots u_r$ to each binding unit $t_{i_1i_2\ldots i_r}$;
- to retrieve, say, the $r$-th component given the first $r-1$, you would compute $u_{i_r} = \sum_{i_1, i_2, \ldots, i_{r-1}} t_{i_1i_2\ldots i_{r-1}}u_1u_2\ldots u_{i_1}u_{i_2}\ldots u_{i_{r-1}}$;

This rapidly becomes impractical, as the size of the network (number of binding units) grows as $n^r$, and it is desirable to have $n$ fairly large in practice, since $n$ is the largest number of concepts that can be represented (per side of the tensor). For example, with a rank 6 tensor, with 64 concepts per side, we would need $64^6 = 2^{36} = 64$ billion binding units.

Gross Topology of a rank 4 tensor product network
This one has 3 components for each of the 4 ‘directions’, so has a total of $3^4 = 81$ binding units. If you want a challenge exercise, you could try drawing all the connections in a manner analogous to figure some pages back labelled “Network Connectivity for a 3x3x3 Tensor Product Network”.

Gross Topology of a rank 4 tensor product network 2
This one has 3 components for each of the 4 ‘directions’, so has a total of $3^4 = 81$ binding units. If you want a challenge exercise, you could try drawing all the connections in a manner analogous to figure some pages back labelled “Network Connectivity for a 3x3x3 Tensor Product Network”.

To teach the network the fact $u(v, w)$, present $u$, $v$ and $w$ to the net. In teaching mode, this causes the content of each binding unit memory $t_{ij}$ to be altered by adding $u_iw_j$ to it.
Applications of Tensor Product Networks

So far the applications have been in theory building for connectionist models, (as in the connectionist production systems and the solution to the connectionist symbol-binding problem, mentioned above) and in construction of theories of cognition (see reference below). We will be discussing this model in a little more detail. First some terminology: diagrams of tensor product networks rapidly become complicated and confusing if you try to draw the details of topology. From an application point of view, what is important is the fact that there are 3 sides (or 2 or 4, or whatever), and to know which sides are being used for input and which for output. So we’ll use the following symbol for a rank 3 tensor product network:

This one is shown with \(v\) and \(w\) as inputs and \(u\) as output, but obviously we could make any side the output.

Solving Proportional Analogy Problems using Tensor Product Networks

This project was done in a university Psychology department. The aim was to simulate simple human analogical reasoning. The tensor product network was used to store facts relevant to the analogical reasoning problem to be solved.

What is a proportional analogy problem?

A proportional analogy problem is a kind of problem sometimes used in psychological testing. The problems are fairly easy for a human over a certain age, but it is not particularly clear how to solve them on a machine. A typical example is

\[
\text{dog : kennel :: rabbit : ?}
\]

To solve the problem one has to ‘fill in’ the ‘what?’ in the most appropriate way - here the answer is burrow (or maybe hutch, if we are talking about pet rabbits).

A human usually responds to this problem by saying something like: “The dog lives-in the kennel - what does the rabbit live in? - a burrow.” In other words, the human names a relationship between dog and kennel, and then proceeds from there.

However, the human does not pick just any relation between dog and kennel (like smaller-than(dog, kennel)): they pick the most salient relation. How? And how could we do this with a machine?

The tensor product network approach actually finesses this question.

A set of “facts”

| woman | loves | baby |
| woman | mother-of | baby |
| woman | larger | baby |
| woman | feeds | baby |
| mare | feeds | foal |
| mare | mother-of | foal |
| mare | larger | foal |
| mare | larger | rabbit |
| woman | larger | rabbit |
| woman | larger | foal |
| woman | lives-in | house |
| baby | lives-in | house |
| mare | lives-in | barn |
| foal | lives-in | barn |
| rabbit | lives-in | burrow |
| barn | larger | woman |
| barn | larger | baby |
| barn | larger | mare |
| barn | larger | foal |
| barn | larger | rabbit |
Steps in the Simple Analogical Reasoning Algorithm

• present **MOTHER** and **BABY** to the *arg1* and *arg2* sides of the net;
• from the *rel*(ation) side of the network we get a “predicate bundle” - the sum of the vectors representing predicates or relation symbols P such that the net has been taught that P(**MOTHER**, **BABY**) holds;
• present this predicate bundle to the *rel* side of the same network and present **MARE** to the *arg1* side of the net;
• from the *arg2* side of the net we get a “weighted argument bundle” - the sum of the vectors representing second arguments y such that the net has been taught that P(**MARE**, y) holds for some P in the predicate bundle.

The weight associated with each y is the *number* of predicates P in the predicate bundle for which P(**MARE**, y) holds. For the given set of facts, the arg2 bundle is $3\#\text{FOAL} + 1\#\text{RABBIT}$.

• pick the concept (arg2 item) which has the largest weight - **FOAL**;

---

Running the net (a Lisp implementation):

The simulation output below demonstrates four things:

1) A normal network solving **woman : baby :: mare : what?**
2) A network that has had 50% of its binding units ‘killed’ tackling the same problem. The binding units are ‘killed’ by altering them so that they always produce zero output.
3) A network that has had 90% of its binding units ‘killed’ solving the same problem.
4) A network that has had 20% of its binding units ‘randomized’ tackling the same problem.

The point of 2-4 is to demonstrate the *robustness* of tensor product networks. In all three cases, the coefficients of **foal** and **rabbit** are reduced, and other concepts may show up with non-zero coefficients, but **foal** always has the largest coefficient, so we know it’s the right answer.

```
? (solve-simple-analogy 'woman 'baby 'mare :verbose)
Tensor product net analogy solver
Network size: 320 nodes (5\times 8\times 8).
Dead so far: 0 nodes (0.0%). Randomized so far: 0 nodes (0.0%)
Total damaged nodes: 0 (0.0%) ...
Problem: WOMAN : BABY :: MARE : ?
Recognizable patterns in solution vector ... 
  FOAL = 3.000
  RABBIT = 1.000
```
? (reset-net) ; restore net to original state - all binding units alive
? (randomize 64)

Neurons random: 20.00%

? (solve-simple-analogy 'woman 'baby 'mare :verbose)
Network size: 320 nodes (5x8x8).
Dead so far: 0 nodes (0.0%). Randomized so far: 64 nodes (20.0%)
Total damaged nodes: 64 (20.0%)

Problem: WOMAN : BABY :: MARE : ?
Volume setting for randomized nodes: 0.10

Recognizable patterns in solution vector ...
WOMAN = 0.511
MARE = 0.058
BABY = 0.456
POAL = 2.200
RABBIT = 0.899
HOUSE = 0.312
BURROW = 0.067

Accessibility

In solving the proportional analogy problem, the TPN was accessed in two different ways:
(1) ARG1 and ARG2 in, REL out
(2) ARG1 and REL in, ARG2 out

This would be impossible with a backprop net - their input/output structure is "hard-wired. In the TPN, the same information in the tensor supports both these modes of operation.
This is rather like when kids learn addition/subtraction - you learn that 9 + 7 = 16, say, and from this you also know that 16 - 7 = 9. We learn addition tables, but not subtraction tables.
An obvious third access mode: ARG2 and REL in, ARG1 out, is possible. And of course, you can have and ARG1, ARG2, and REL in, YES/NO out access mode.
Less obviously, you can have access modes like:
REL in, ARG1®ARG2 out

In fact there are a total of 7 access modes to a rank 3 tensor.
This generalises to higher rank tensors: there are $2^k-1$ access modes for a rank k tensor. This property is referred to as accessibility (or sometimes as omnidirectional access).

Example: Given a rank 5 tensor, with axes REL, ARG1, ARG2, ARG3, and ARG4, one possible access mode would be
ARG1, ARG2 in, REL® ARG3® ARG4 out

For more details...


This work led to a relational view of cognition which is described in:

What else can tensor product nets do?

Smolensky (AI 46 (1990) full citation above) reviews this. In particular, he presents a lot of theoretical analysis, and inter alia shows that stacks can be implemented in tensor product networks, and indeed that general lists can be. He does this by showing how to implement the basic Lisp primitives car, cdr, and cons (head, tail and ; for Haskell hackers).

If you haven’t met a functional programming language, then
- head(L) is a function that returns the first item on the list L
- tail(L) is a function that returns all of the list L except the first item
- Item : Rest is an operator that returns a list whose first item is Item and the rest of which is the list Rest
Approximate unbinding and random representations

We have seen that we can kill 90% of binding units and still solve problems with the tensor product networks. When we kill a binding unit, this affects the computation of inner products in the retrieval process - it is if the representation vectors are not exactly orthogonal anymore.

So perhaps exact orthogonality is not critical - perhaps approximate orthogonality would be enough. It is also implausible that biological neural networks have orthonormal sets of vectors representing concepts.

What sort of vectors would be ‘approximately orthogonal’? Answer: vectors with uniform random components chosen from the interval \([-r, r]\) have inner products with mean 0 and variance which depends on r.

Experiments with ‘dense’ random representations

The researchers also tried using as representation vectors, normalised random vectors generated by choosing each component to be a uniform random number from the range \([-1,+1]\), and normalising the resulting vector.

Normalisation has the effect that vectors with more components will (on average) have smaller components.

For this reason, they investigated the effect of increasing the number of components on network performance.

Experiments were done with the analogy problem: "woman is to baby as mare is to what?" and a particular set of facts.

We classified runs as good or poor or bad.

- **good**: \(\text{score(faul)} > 1.2 \times \text{score(rabbit)}\) and \(\text{score(rabbit)} > 1.2 \times \text{next highest score}\)
- **fair**: \(\text{score(faul)} > \text{score(rabbit)} > \text{next highest score}\)
- **bad**: not (good or fair)

The factor 1.2 was an arbitrarily chosen cutoff. [It was not "tuned"].

Sparse Random Representations

A distributed representation does not require that all or even most nodes have non-zero activations, just that all concept are represented by a significant number of active nodes.

Indeed, neurophysiological data indicates that percent activity of neurons in the different regions of the human hippocampal system ranges from about 0.5% to about 9%.

Accordingly, the researchers constructed a version of their model which uses sparse random representations: that is representations are generated with most components zero, and the remainder chosen from a uniform random distribution.

Again, the inner products of such vectors have mean zero, and so can be considered quasi-orthogonal.

With small numbers of non-zero components, one would expect freaky behaviour - these cases do not correspond to distributed representations.

In the experiments, they used vectors of length 500 (and so 125 million binding units in the tensor).

Each experimental run used a fixed number of non-zero components, ranging from 1 to 50, and chose the components initially as uniform random numbers in \([-1,+1]\), but then normalized the resulting vectors.

One hundred runs were done with each number of non-zero components.

Dense random representation results

<table>
<thead>
<tr>
<th>components</th>
<th>% good</th>
<th>% fair</th>
<th>% bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>40.9</td>
<td>10.3</td>
<td>48.8</td>
</tr>
<tr>
<td>40</td>
<td>55.5</td>
<td>7.5</td>
<td>37.0</td>
</tr>
<tr>
<td>70</td>
<td>73.5</td>
<td>9.3</td>
<td>18.2</td>
</tr>
<tr>
<td>100</td>
<td>84.8</td>
<td>4.3</td>
<td>10.9</td>
</tr>
<tr>
<td>130</td>
<td>92.6</td>
<td>3.5</td>
<td>3.9</td>
</tr>
<tr>
<td>160</td>
<td>94.5</td>
<td>2.5</td>
<td>3.0</td>
</tr>
<tr>
<td>190</td>
<td>96.7</td>
<td>1.8</td>
<td>1.5</td>
</tr>
<tr>
<td>220</td>
<td>97.4</td>
<td>1.4</td>
<td>1.2</td>
</tr>
<tr>
<td>250</td>
<td>98.7</td>
<td>1.2</td>
<td>0.1</td>
</tr>
</tbody>
</table>
Non-zero components

Figure 1: Results of sparse random vector experiments. In all cases with more than 25 non-zero components, all solutions were classified as “good”.

Fact and Non-Fact Recognition Scores

Further experiments on random representations investigated the ability of the networks to recognize individual facts (as opposed to solving analogy problems, which is a more complex task which involves accessing information relating to a large number of facts).

These experiments confirmed that random representations, with enough components in the vectors, permit one to decide whether a proposition is a fact or a non-fact (where we define a fact to be something that the network has been taught.)

We can only work with representable non-facts – that is, ones that use concepts for which there are representations available. For example, if mother-of, barn, and rabbit are all concepts with representations in our tensor product network, then we can represent the proposition mother-of(rabbit, barn), but we probably don’t teach it to the tensor product net, so it would be a representable non-fact.

What about higher ranks?

The experiments reported here were done with rank 3 networks. More recent experiments on fact and non-fact retrieval with networks of ranks 4, 5, 6, and 7 with Hadamard bases and large numbers of neurons “killed” (rather than with random representations) have shown that one can kill around 90% of the neurons and still be able to distinguish facts from non-facts.

Above rank 7, it begins to become difficult to complete simulations with reasonable amounts of computing resources.
Summary

- Tensor product networks offer precise retrieval
- Tensor product networks use one-shot learning
- TPNs can be used to support variable-value binding
- Different ranks for different relational "arities"
- TPNs can be used in cognitive modelling
- TPNs can be used to build data structures
- TPNs have the omnidirectional access property
- TPNs using distributed representations are robust to damage
- Random representations can be substituted for orthonormal ones
- Sparse random representations work, too