VSA demonstration: The shifter problem

Ross Gayler

r.gayler@gmail.com

Motivation

- Vector Symbolic Architectures (VSAs) are a little known family of connectionist representation schemes (Plate; Kanerva; Rachkovskij; Gayler) that are able to address symbolic issues (e.g. compositional structures).
- Although VSAs are connectionist in form they are quite unlike “traditional” connectionist systems in behaviour.
- Researchers familiar with traditional connectionist systems may inappropriately generalise from them and fail to notice the unique capabilities of VSAs.
- The aim of this demonstration is to provide a simple example of the unique power of VSAs.
- This demonstration is novel, the VSA concepts are not.

VSA architectural commitments

- Network of connectionist units
  - Single scalar output, multiple scalar inputs, fixed local numerical calculation
  - Small set of functional forms (VSA primitive operators)
- Sets of units are construed as vectors
  - High-dimensional vectors of unit outputs (1,000s to 10,000s of units)
- All vectors have the same dimensionality
  - Allows closed representation of composite structure (e.g. trees and graphs)
- Distributed representations on vectors
  - Representations appear random

Abstract VSA primitives

- Mathematical primitives able to be implemented in connectionist hardware
  - The details are not critical (Plate 2003, pp 227-230)
- Operands are fixed dimension vectors
  - High dimensional (typically > 1,000)
  - Low resolution elements (1 bit is enough)
- Operators

	Mathematical	Hardware
Multiplication-like	$\times$	$\otimes$
Addition-like	$+$	$\oplus$
Permutation-like	$P_i(\ )$	$\circlearrowright_i$

Specific VSA primitives

	Real HRR (Plate)	Complex HRR (Plate)	Spatter code (Kanerva)	MAP code (Gayler)	Context-Dependent Thinning (Rachkovskij)
Vector elements	Real	Complex	Bits	Real (signs)	Bits
$\times$	Circular convolution	Element-wise product	Element-wise XOR	Element-wise product	Element-wise OR and AND
$+$	Element-wise sum	Element-wise sum	Element-wise majority	Element-wise sum	

Levels of description

	VSA	Computer
Vector Representation	**High-dimensional vectors & Vector operators ($\otimes$, $\oplus$, P( ))**	**Binary words & Logical operators (AND, XOR, etc)**
Representational Architecture	**Fixed network of operators**	**CPU**
Cognitive Architecture	**Mapping of task onto lower levels** (representational idioms, population of memory, etc.)	**Program** (data structures, function definitions, etc.)

The importance of VSA lies in what is made easy

Similarity

- Connectionist approaches depend on vector similarity
  - Similar input vectors (generally) lead to similar outputs
- Define similarity as the normalised dot product
  - Correlation of the vectors
  - Cosine of angle between the vectors
- E[corr(**a**,**b**)] = 0   (distribution over random **a** and **b**)
- sd[corr(**a**,**b**)] = *small*   (scales as $n^{-0.5}$)
- Random pairs of vectors generally maximally dissimilar
  - Symbol-like behaviour (vectors generally same or different)
  - Graded similarity can be constructed as a special case
  - E[corr(**a**,**a+b**)] >> sd[corr(**a**,**a+b**)]

Distribution of similarity

Plate, T.A. (1994) *Distributed representations and nested compositional structure.* Ph.D. thesis, Department of Computer Science, University of Toronto

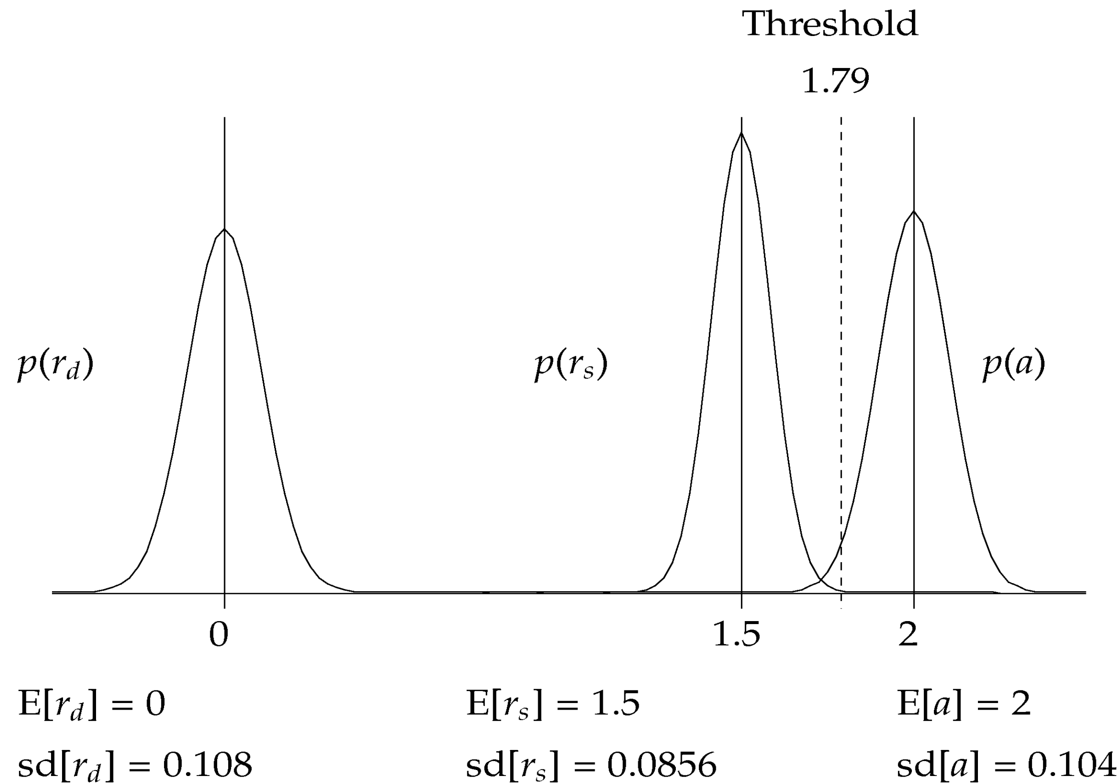


Figure B.4: Probability density functions for signals in an superposition memory with similarity among vectors, with $n = 512$, $m = 1000$, $|\mathbb{E}_p| = 100$, $\alpha^2 = 0.5$, and $k = 3$.

Contribution of sign & magnitude

Plate, T.A. (1994) *Distributed representations and nested compositional structure.* Ph.D. thesis, Department of Computer Science, University of Toronto

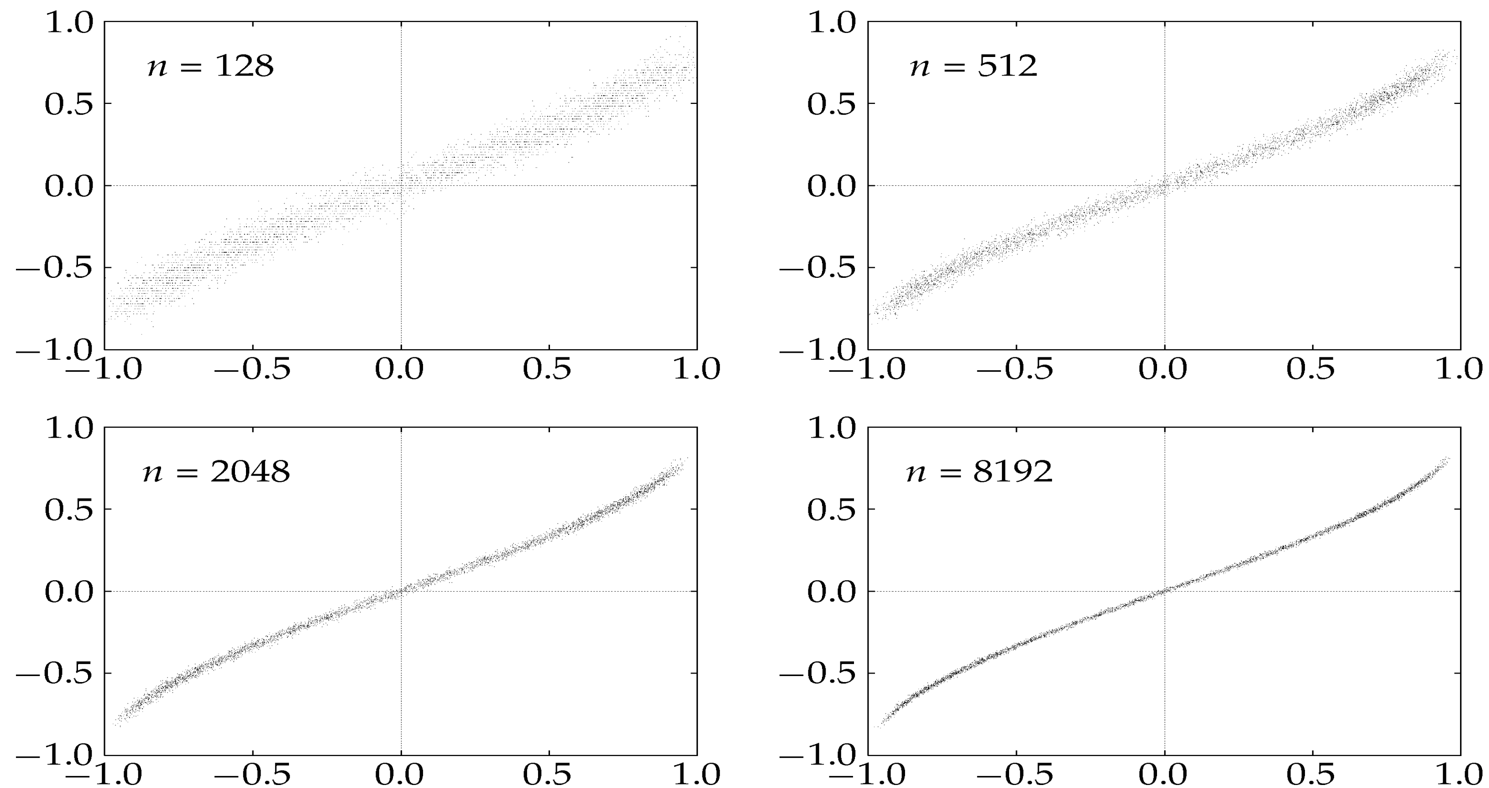


Figure 3.8: Bitwise dot-products (vertical axis) versus floating-point dot-products (horizontal axis) for various vector dimensions.

Bundling (+)

- Used to create weighted sets (intuitive interpretation)
- The bundle is the superposition of its constituents
- The bundle operator preserves similarity
  - The resulting vector is **similar** to all its constituent vectors.
- Example: given 10,000-dimensional random vectors **x**, **y**, **z**, and **r**, we can calculate **m** = **x + y + z**
  - The correlation coefficients of the resulting vector with its constituents are around 0.5:
    corr(**x**,**m**) = 0.50, corr(**y**,**m**) = 0.48, corr(**z**,**m**) = 0.51
  - While for another random vector **r** not a constituent of **m**:
    corr(**r**,**m**) = 0.01
- Bundles can be probed and can be used holistically

Binding (×)

- Used to create structural associations (intuitive)
- The binding is the association of its components
- The components may be retrieved from the binding

  Generically, given bind(**a**,**b**):

  unbind(bind(**a**,**b**),**a**) ≈ **b**

  unbind(bind(**a**,**b**),**b**) ≈ **a**

  unbind(bind(**a**,**b**),**c**) ≈ **0**

- Details of bind(.,.) and unbind(.,.) vary between VSAs
- The bind operator partially destroys similarity

  corr(bind(**a**,**b**),**a**) ≈ 0

  corr(bind(**a**,**b**),bind(**a’**,**b**)) = corr(**a**,**a’**)

- Bindings may be bundled and used holistically

Implementations of binding

- Details of bind(.,.) and unbind(.,.) vary between VSAs
  - HRR: bind ≡ circular convolution, unbind ≡ circular correlation
  - BSC & MAP: bind ≡ unbind ≡ element-wise product (XOR,×)
- In BSC & MAP every vector is self-inverse w.r.t. bind(.,.):

  $(+1)\times(+1) = +1;$ $(-1)\times(-1) = +1;$ $\mathbf{a}\times\mathbf{a} = \mathbf{1}$
- Whether binding or unbinding occurs depends on the relation between the operands:

  $\mathbf{a}\times\mathbf{b} = (\mathbf{a}\times\mathbf{b})$ (binding)

  $\mathbf{a}\times(\mathbf{a}\times\mathbf{b}) = (\mathbf{a}\times\mathbf{a})\times\mathbf{b} = \mathbf{1}\times\mathbf{b} = \mathbf{b}$ (unbinding)
- This makes design more difficult
- But allows substitution as a primitive operation
  - Substitution traditionally seen as a symbolic operation

Substitution

- The association of two vectors can be used as a substitution operator:

  $(\mathbf{a}\times\mathbf{b})$ (substitutes **a** for **b** and vice versa)

  $(\mathbf{a}\times\mathbf{b})\times(\mathbf{a}\times\mathbf{x}) = \mathbf{a}\times\mathbf{a}\times\mathbf{b}\times\mathbf{x} = (\mathbf{a}\times\mathbf{a})\times\mathbf{b}\times\mathbf{x} = (\mathbf{b}\times\mathbf{x})$

  $(\mathbf{a}\times\mathbf{b})\times(\mathbf{b}\times\mathbf{x}) = \mathbf{b}\times\mathbf{b}\times\mathbf{a}\times\mathbf{x} = (\mathbf{b}\times\mathbf{b})\times\mathbf{a}\times\mathbf{x} = (\mathbf{a}\times\mathbf{x})$

- **a**,**b**,**x** can represent complex structures
  - The hardware sees them only as vectors (blind to complexity)
- Substitution can be applied holistically to a bundle

  $(\mathbf{a}\times\mathbf{b})\times(\mathbf{a}\times\mathbf{x} + \mathbf{a}\times\mathbf{y} + \mathbf{a}\times\mathbf{z}) = (\mathbf{b}\times\mathbf{x} + \mathbf{b}\times\mathbf{y} + \mathbf{b}\times\mathbf{z})$

- Substitution is necessarily systematic

Cleanup memory

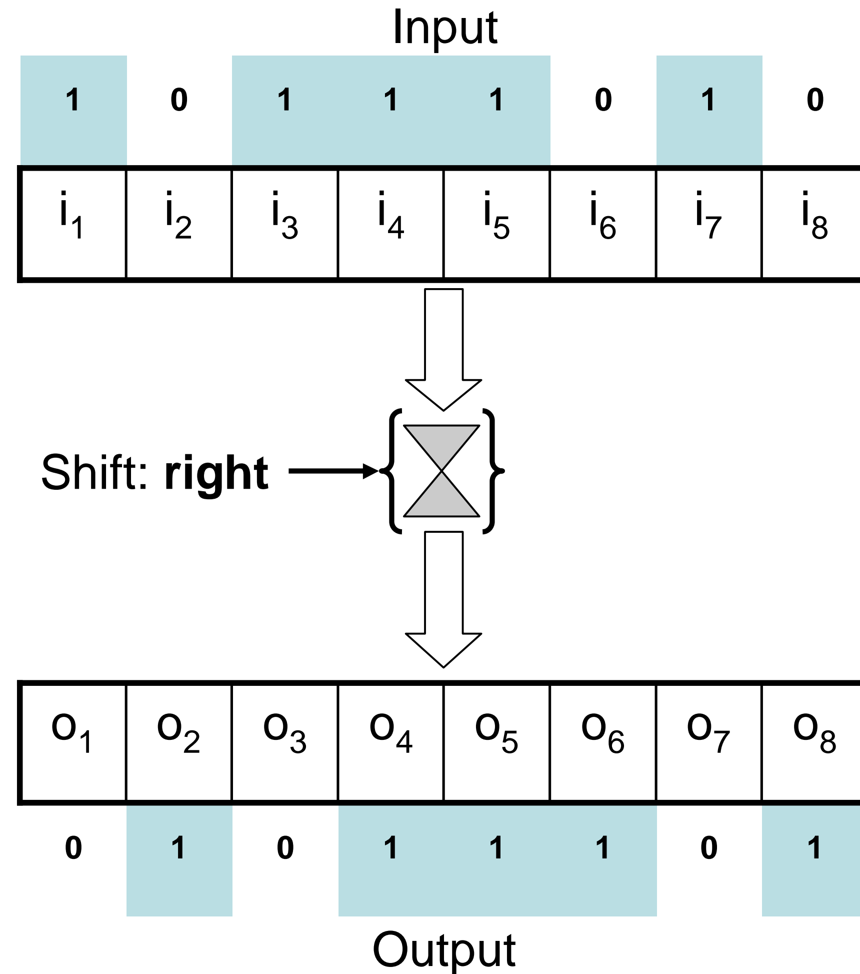
- Results of operations on bundles usually approximate
  - $(\mathbf{a})\times(\mathbf{a}\times\mathbf{b} + \mathbf{c}\times\mathbf{d} + \mathbf{e}\times\mathbf{f} + \ldots)$
  - $= \mathbf{b} + \mathbf{a}\times\mathbf{c}\times\mathbf{d} + \mathbf{a}\times\mathbf{e}\times\mathbf{f} + \ldots$
  - $= (\mathbf{b} + \boldsymbol{\varepsilon})$
  - Where $\mathrm{corr}((\mathbf{b} + \boldsymbol{\varepsilon}),\mathbf{b})$ = *large*
- We can tell the result contains **b** but need an extra step if we want to extract **b** as a signal
- There are no inherently distinguished vectors (directions)
- Cleanup memory distinguishes vectors we choose

  cleanup(($\mathbf{b} + \boldsymbol{\varepsilon}$),item-set) = **b**

  e.g. **result** = (**input** ʘ **item**) **item**

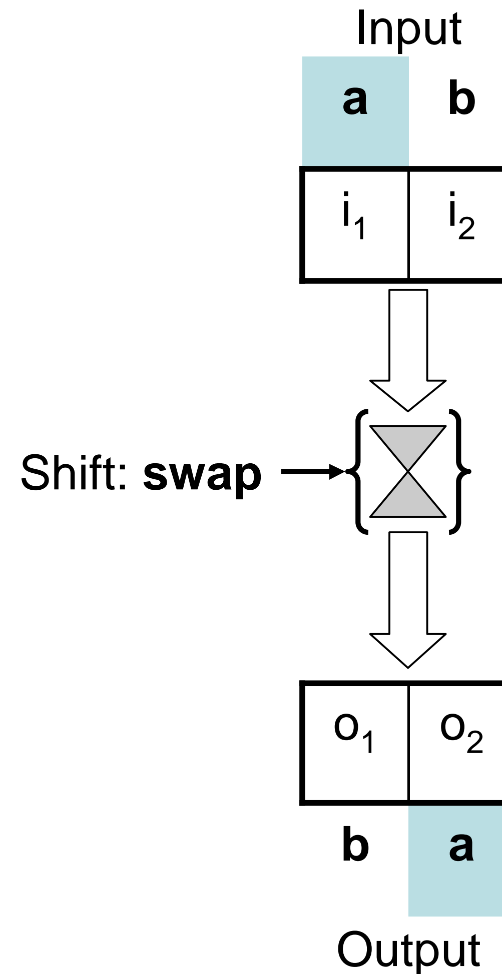
The shifter problem

- PDP (1986) V1 pp299-303
- Registers
  - 8 bits in
  - 8 bits out
  - shift (left, right, none)
- Boltzmann net
- Train: Show 360,000 <in,shift,out> triples
- Test: Recognise the shift from the input & output
  - 50% to 89% correct (fn. of number of active bits)



The demo problem (modified)

- Registers
  - 2 arbitrary patterns in
  - 2 arbitrary patterns out
  - shift (swap, none)
- Patterns (not bits)
  - More general than bits
  - Shifts are unambiguous if unique patterns used
- Shorter registers
  - Fewer possible shifts
  - Less information load
  - More ambiguous if patterns are not unique

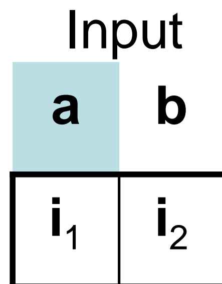


Encoding the shifter problem

- Represent the input and output registers as “frames”
  - A frame is a set of <role,value> pairs
- Each position in a register is a “role”
- Bind the input and output register frames with the shift
  - <input,shift,output> triples
- This choice of representation is not unique, but it is simple, natural, and works
- Store <i,s,o> triples as examples in the memory trace
- Query the memory trace with novel <i,s,o> fragments to test retrieval and generalisation

Encoding the input register

Input Register



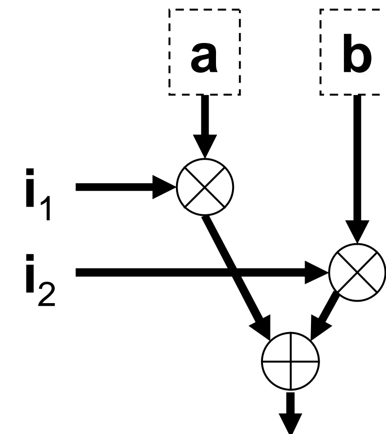
- **a** and **b** are input patterns
- $\mathbf{i}_1$ and $\mathbf{i}_2$ identify the positions of the input register

Mathematical

$$(\mathbf{i}_1 \times \mathbf{a} + \mathbf{i}_2 \times \mathbf{b})$$

- **a** and **b** are inputs
- $\mathbf{i}_1$ & $\mathbf{i}_2$ constants
- values are vectors
- named vectors are approx. orthogonal

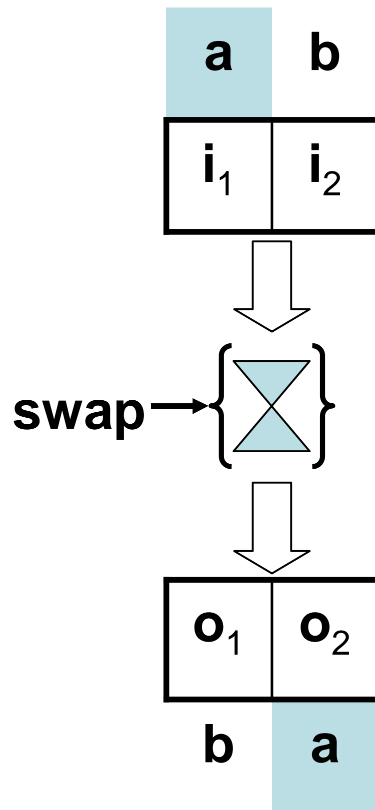
Hardware



- arrows carry vector values
- **a** and **b** are vector inputs
- $\mathbf{i}_1$ and $\mathbf{i}_2$ are vector constants

Encoding a complete example

Shift Register

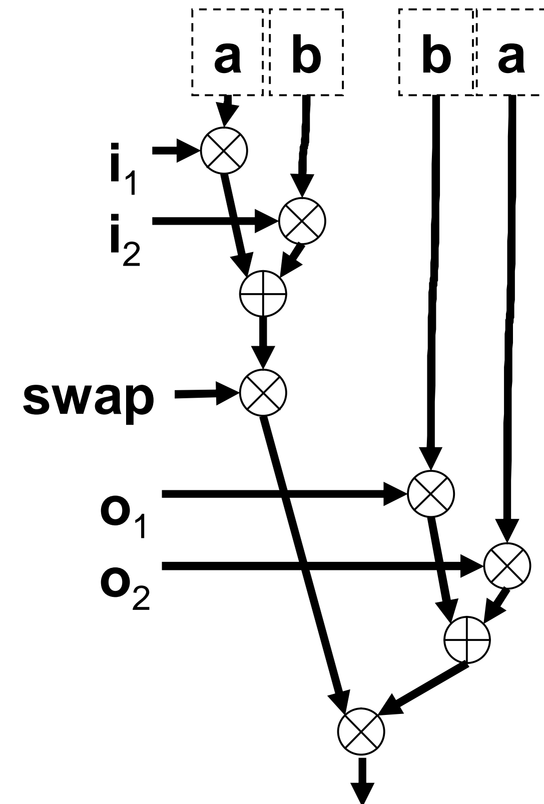


Mathematical

$(i_1 \times a + i_2 \times b)$
$\times$ **swap** $\times$
$(o_1 \times b + o_2 \times a)$

- **swap** and **none** are vector constants

Hardware



Every vector is just a vector

- The encoding of an example looks complex

  $(\mathbf{i_1 \times a} + \mathbf{i_2 \times b}) \times \mathbf{swap} \times (\mathbf{o_1 \times b} + \mathbf{o_2 \times a})$

  $= (\mathbf{i_1 \times a \times swap} + \mathbf{i_2 \times b \times swap}) \times (\mathbf{o_1 \times b} + \mathbf{o_2 \times a})$

  $= \mathbf{i_1 \times a \times swap \times o_1 \times b} + \mathbf{i_2 \times b \times swap \times o_1 \times b} + \mathbf{i_1 \times a \times swap \times o_2 \times a} + \mathbf{i_2 \times b \times swap \times o_2 \times a}$

  $= \mathbf{swap \times i_1 \times o_1 \times a \times b} + \mathbf{swap \times i_2 \times o_1 \times (b \times b)} + \mathbf{swap \times i_1 \times o_2 \times (a \times a)} + \mathbf{swap \times i_2 \times o_2 \times b \times a}$

  $= \mathbf{swap \times i_1 \times o_1 \times a \times b} + \mathbf{swap \times i_2 \times o_1 \times 1} + \mathbf{swap \times i_1 \times o_2 \times 1} + \mathbf{swap \times i_2 \times o_2 \times b \times a}$

  $= \mathbf{swap \times i_1 \times o_1 \times a \times b} + \mathbf{swap \times i_2 \times o_1} + \mathbf{swap \times i_1 \times o_2} + \mathbf{swap \times i_2 \times o_2 \times b \times a}$

  $= \mathbf{swap \times i_2 \times o_1} + \mathbf{swap \times i_1 \times o_2} + \textbf{(input-output value dependent stuff)}$

- But as far as the hardware is concerned it is just a vector like any other. The decomposition into terms is in the mind of the analyst and not visible to the hardware.

Memory traces in MAP coding

- A memory trace is a vector ($\mathbf{T}_t$) from which values can be retrieved.

- The memory trace is updated as items are added
- Add an association to the memory trace
  $\mathbf{T}_{t+1} = \mathbf{T}_t + \mathbf{a}\times\mathbf{b}$

- Retrieve from the memory trace using **a** as the cue
  $\text{retrieve}(\mathbf{T}_{t+1}, \mathbf{a}) = \mathbf{T}_{t+1} \times \mathbf{a}$
  $= (\mathbf{a}\times\mathbf{b} + \mathbf{T}_t) \times \mathbf{a}$
  $= \mathbf{a}\times\mathbf{a}\times\mathbf{b} + \mathbf{T}_t\times\mathbf{a}$
  $= \mathbf{a}\times\mathbf{a}\times\mathbf{b} + \sim\mathbf{0}$
  $\approx \mathbf{a}\times\mathbf{a}\times\mathbf{b}$
  $\approx (\mathbf{a}\times\mathbf{a})\times\mathbf{b}$
  $\approx \mathbf{1}\times\mathbf{b}$
  $\approx \mathbf{b}$

Adding To and Querying the Memory Trace

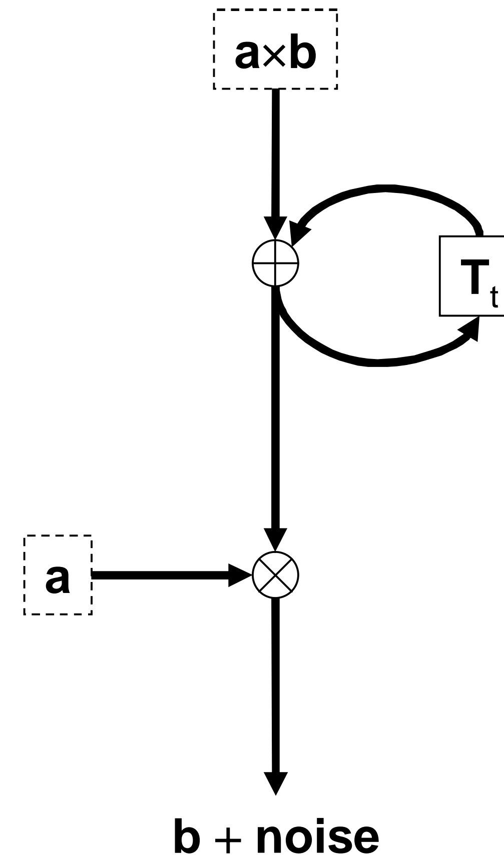
Mathematical

$\mathbf{T}_{t+1} = \mathbf{T}_t + \mathbf{a} \times \mathbf{b}$

retrieve($\mathbf{T}_{t+1}$, $\mathbf{a}$)

$= \mathbf{T}_{t+1} \times \mathbf{a}$

Hardware



Value independent substitution

- The memory trace of a single example contained terms like: $\mathbf{swap} \times \mathbf{i}_1 \times \mathbf{o}_2$
- These terms can be applied as substitution operators
- They are independent of the actual input/output values
- When we query the memory of examples with a partial cue (e.g. $(\mathbf{i}_1 \times \mathbf{p} + \mathbf{i}_2 \times \mathbf{q}) \times \mathbf{swap})$) the role vectors are substituted to yield the expected answer

$$(\mathbf{swap} \times \mathbf{i}_2 \times \mathbf{o}_1 + \mathbf{swap} \times \mathbf{i}_1 \times \mathbf{o}_2 + \ldots) \times ((\mathbf{i}_1 \times \mathbf{p} + \mathbf{i}_2 \times \mathbf{q}) \times \mathbf{swap})$$

$$\approx (\mathbf{o}_1 \times \mathbf{q} + \mathbf{o}_2 \times \mathbf{p})$$

- Generalises to novel values (if in cleanup memory)
- Can generalise from one example