Hey Pentti, We Did It!: A Fully Vector-Symbolic Lisp

Eilene Tomkins-Flanagan (eilenetomkinsflanaga@cmail.carleton.ca)
Mary Alexandria Kelly (mary.kelly4@carleton.ca)
Department of Cognitive Science, Carleton University
1125 Colonel By Drive, Ottawa, ON, K1S 5B6, Canada

Abstract

Kanerva (2014) suggested that it would be possible to construct a complete Lisp out of a vector-symbolic architecture. We present the general form of a vector-symbolic representation of the five Lisp elementary functions, lambda expressions, and other auxiliary functions, found in the Lisp 1.5 specification (McCarthy, 1960), which is near minimal and sufficient for Turing-completeness. Our specific implementation uses holographic reduced representations (Plate, 1995), with a lookup table cleanup memory. Lisp, as all Turing-complete languages, is a Cartesian closed category (nLab authors, 2024), unusual in its proximity to the mathematical abstraction. We discuss the mathematics, the purpose, and the significance of demonstrating vector-symbolic architectures' Cartesian-closedness, as well as the importance of explicitly including cleanup memories in the specification of the architecture.

Keywords: vector-symbolic architecture; Lisp; holographic reduced representations; cartesian closed category; modern hopfield network

At Clojure/Conj 2023, the conference of the Clojure programming language, Meier (2023) introduced vector-symbolic architectures to the Clojure community. Her presentation echoed a motif oft heard listening to programmers who discover vector-symbolic architectures (VSAs) for the first time; namely, VSAs' unusual properties and computational niceties are objects of fascination, but it is not immediately obvious what good a vector-symbolic architecture does for the programmer. We present an existence proof that VSAs are completely general computational tools. In technical terms, VSAs can do anything one wants. In pragmatic terms, "technically anything" does not answer questions of naturalness and ease of representation. To answer practical questions, VSAs have been used most frequently in representing human cognition, fruitfully in simultaneous localization and mapping (SLAM), and, pertinent to our analysis, promisingly in the syntactic manipulation of neural network states.

In an remark at the end of her talk, Meier mentioned a "challenge", issued by Kanerva, in "one of his papers", to implement Lisp using exclusively a vector-symbolic architecture representing all the language's expressions. However, the exact words read in the talk as a challenge, "One could create a 'High dimensional computing-Lisp"', do not seem to have been written by Kanerva. This apparent mistake is not Meier's, as those exact words have been published in Neubert, Schubert, and Protzel (2019), who attribute the enclosed quote to Kanerva (2014). While the quote does not appear in Kanerva's paper, the mistake is plausibly a case of miscitation of something said during the associated conference talk, and in any case it is not serious. Kanerva's paper, disappointingly, does not include any challenge, but rather a discussion of how a vector-symbolic Lisp might be implemented, coupled with a loose specification of some of the tools that might be required to do so. We are going to pretend that counts as a challenge, and fully specify a Lisp language in terms of a generic VSA[1].

Hold Up, What's a VSA?

A vector-symbolic architecture is an algebra (i.e., a vector space with a bilinear product),

1. that is closed under the product $\otimes : V \times V \to V$ (i.e., if $u \otimes v = w$, then $u, v, w \in V$),
2. whose product has an "approximate inverse" $\overline{\otimes}$ that, given a product w and one of its operands u or v, yields a vector correlated with the other operand,
3. for which there is a dogma for selecting vectors from the space to be treated as atomic "symbols",
4. that is paired with a memory system $\mathcal{M}$ that stores an inventory of known symbols for retrieval after lossy operations (e.g., inversion), that can be recalled from $\mathcal{M}(p)$, and which is appendable $\mathcal{M} \leftarrow t$, and
5. possesses a measure of the correlation (a.k.a., similarity) of two vectors, $\mathbf{sim}(u, v) \in [-1, 1]$, where 1 and -1 imply that u, v are colinear, 0 that they are linearly independent.

The $+$ and $\otimes$ operators behave analogously to disjunction and conjunction, or set-theoretic union and intersection. Additionally, VSAs may have an analogue for negation, often the vector rejection on Euclidean space $\mathbf{rej}_v(u)$ (Widdows & Cohen, 2014), and permutations Π, which are typically used to introduce asymmetry to the product operator, by applying different permutations to the operands. For a detailed review of vector symbolic architectures, see Kleyko, Rachkovskij, Osipov, and Rahimi (2022, 2023). Heddes et al. (2023) develop a software library for applying VSAs based on Torch.

In our implementation, we use holographic reduced representations (Plate, 1995). They are defined over Euclidean space $\mathbb{R}^n$, and have circular convolution as their product, co-

[1] Our implementation may be found at `https://github.com/eilene-ftf/holis`

sine as their similarity, and atomic symbols sampled from a Gaussian distribution. Our memory system is a lookup table.

Some VSAs (e.g., Kanerva, 1996) are not defined over vector spaces per se, or otherwise relax some of the above properties, but behave sufficiently similarly to be used in a similar way. The programmer's choice of VSA comes down to preference and different computational conveniences. For the most part, all VSAs are as good as all others.

Vector-symbolic architectures are an answer to an old tension in cognitive science between the actual machinery of the brain and properties cognition is believed to necessarily possess. Brain states, to the one side, are described in terms of the activity of multiple cell populations, and they exist over a fixed number of cells. Information is typically assumed to be distributed over measured populations, degrading gracefully and uniformly when cells are disabled at random. To the other, Fodor and Pylyshyn (1988) made a compelling case that central cognition must have states that behave like discrete symbols, that can be strung together in a combinatorial syntax. But, the traditional tool for representing such syntaxes, computer memory, uses strings of bits for individual symbols, composed by concatenation into ever-longer strings.

With disjunction (sum), conjunction (product), inverse, and similarity operations, plus a cleanup memory, a VSA is sufficient to describe any syntax one could want, to a finite precision. Thus, VSAs appear to satisfy the cogntive scientist's parallel demands for syntax and biomimicry.

Cartejian Clojure and Lisp

A Cartesian closed category (nLab authors, 2024) is the mathematical generalization of what it means to compute. It generalizes the equivalence of universal Turing machines (Li & Vitányi, 2008, ch. 1) with other definitions of computation, set theory, first-order logic, and, of interest to us, the recursively enumerable languages **RE** (Chomsky, 1955). All instances of a Cartesian closed category have the preceding equivalences; to say that C is Cartesian closed is also to say that it is Turing-complete. It follows that C can define **RE**, and, therefore, any syntax, as the language generated by any syntax rules, or grammar, is a subset of that generated by **RE**.

Categories have two contents: *objects* and *morphisms*[2]. For example, while we normally treat vector spaces as sets of vectors augmented with some functions, they are equally categories that *include* both vectors (their objects) and functions (their morphisms). Cartesian closed categories in particular are useful because they are very simple, and so it is usually easy to demonstrate that a formal system is Cartesian closed.

In a Cartesian closed category C, there is (1) one object, called a terminal object **1** (so-named because there is a morphism from every object in the category to it). There is also (2) a product that can compose any two objects, under which C is closed, i.e., if A, B are objects in C, the product of any objects $A \times B$ in C is also an object in C. (3) The functions $A \to B$ on objects in C are together an object in C, written B^A. (4) A morphism that evaluates functions in C, parameterized by objects in C, $\mathbf{eval}_C : B^A \times A \to B$, is in C.

These four properties give us four tests for whether some formal system S is Cartesian closed. S must have at least one base data object, and we should be able to transform any expression into it (1). S must have some means to compose arbitrary expressions from its objects, that are data objects still usable by S (2). S must be able to express functions that may map any objects to any others, and those functions must be representable as data objects (3). Finally, we should be able to describe a complete interpreter for S, in terms of S(4).

Keen readers will have noticed why the above defines computation. We have some base symbols; we may construct sequences of symbols, any length; we can specify any function that transforms sequences to other sequences; and, we can evaluate those functions. That is pretty much a description of a universal Turing machine (see Li and Vitányi, ch. 1).

McCarthy (1960) described Lisp for the 1.5th time, giving us the mother document of all subsequent Lisp dialects. Its simplicity will enable us to complete our "challenge" without taking up a whole book. To demonstrate that VSAs can compute, we need only implement the five "elementary" functions of Lisp 1.5, plus some other functions that can be recursively defined in terms of the others. The elementary functions are: `CONS`, `CAR`, `CDR`, `EQ`, and `ATOM`. Additionally, there are `LAMBDA`, `COND`, and `LABEL`. `LAMBDA` is the most important, as it allows us to define lambda expressions, i.e., arbitrary functions.

In Lisp 1.5, a tuple is represented in the form `(a . b)`, where `a` and `b` are either atomic symbols (written as alphanumeric sequences) or other tuples. A list `(a b c)` is equivalent to the tuple `(a . (b . (c . NIL)))`, where `NIL` is an atomic symbol that represents the end of a list. Naturally, the singleton `(a)` is the tuple `(a . NIL)` and the empty list `()` is just `NIL`. The atomic symbols `NIL`, `T`, and `F` are always defined. We'll define the elementary functions, where lowercase letters are variables that may be any valid Lisp expression:

```
(CONS a b) = (a . b)
(CAR (a . b)) = a
(CDR (a . b)) = b
(EQ a a) = T
(EQ a b) = F
(ATOM (a . b)) = F
(ATOM a) = T
```

The preceding definitions use a pattern-matching format, such that the earlier definition takes precedence. Where the same letter is used for two variables, the variables must be identical. In plain English, `CONS` takes any two expressions, and constructs a tuple containing them. `CAR` takes a constructed pair, and yields the left element, while `CDR` does so with the right. `EQ` tests whether two atomic symbols are identical, and is undefined for non-atomic symbols. `ATOM` tests whether an expression is atomic. This already seems like very

[2]Generally, a morphism is any way objects can be related such that, if you have two morphisms f, g, you can construct $h = f \circ g$ such that $h(a) = f(g(a))$ for some appropriate notion of equivalence.

little, but armed with an understanding of Cartesian closure, it can be understood that we don't even need all of Lisp to have a Turing-complete language. We just need `CONS` (our product), `ATOM` (in case it is not immediately clear, there is a morphism from any expression `e` to `T` given by `(ATOM (ATOM e))`), and `LAMBDA` (for functions; the Lisp interpreter evaluates expressions and can be described as a Lisp expression). What makes Lisp remarkable as a point of reference is that there is almost no fat on top of the basic building blocks of a bicartesian closed category; we can describe anything we would like recursively in terms of the basic functions. To wit:

```
((LAMBDA NIL e)) = e
((LAMBDA x NIL) a) = NIL
((LAMBDA x ((CAR x) . e)) a) = (LAMBDA (CDR x)
    (a . ((LAMBDA x e) a)))
((LAMBDA x ((c . d) . e)) a) = (LAMBDA (CDR x) (
    ((LAMBDA x (c . d)) a) . ((LAMBDA x e) a)))
((LAMBDA x e) a) = (LAMBDA (CDR x) ((CAR e) .
    ((LAMBDA x (CDR e)) a)))
```

The preceding recursively defines lambda expressions entirely in terms of the Lisp elementary functions, provided that arguments are always curried[3]. The above recursive definition has five cases, where any time `LAMBDA` is called, the earliest definition that fits the arguments takes precedence. A lambda expression is a three element list, containing `LAMBDA`, a list of parameters `x`, and an enclosed expression `e`. At base, `(LAMBDA x e)` does nothing, but it can be called on one argument `a`, which may be `NIL`, `((LAMBDA x e) a)`, and then it is evaluated over `a`, returning either a lambda expression where `a` is substituted for all instances of the first parameter, or, if there are no arguments left, the resultant body expression with all substitutions made. In our definition, all lambda expressions are always curried, so a function on three arguments `a`, `b`, `c` is called as `((((LAMBDA x e) a) b) c))`, with the final call being implicitly on the single argument `NIL`, as `NIL` terminates all lists. The parameters, `x`, are a list that is assumed to consist of atomic symbols. `LAMBDA` is undefined where elements of `x` are nonatomic or duplicate.

`COND` implements conditional expressions:

```
(COND ((T . q) e)) = q
(COND e) = (COND (CDR e))
```

The way conditionals work is pretty straightforward. We write some implications, and when evaluated, we take the first branch whose condition is satisfied after evaluation.

It is worth noting that this form of recursive definition is useful for its terseness, but it is not proper to LISP 1.5, which would require the use of a `DEFINE` pseudo-function to instantiate a function definition. `DEFINE` is not one of the elementary functions because it just maps a name to an expression in system memory. Recursive expressions are possible *without* using `DEFINE`, so the above effects can be achieved (if not persistently named) by using the `LABEL` function. `LABEL` is the fundamental tool by which recursion is achieved in the Lisp 1.5 specification, but we choose to omit it due to redundancy.

What we notice in the Lisp 1.5 specification is that there is remarkable inclarity as to what is core to the language and how things are formally defined. Understanding Cartesian closed categories, however, helps to clear up some details. We have chosen to define the parts of Lisp that are elementary, plus lambda expressions, and functions that make programming minimally less painful: `QUOTE`, `COND`, and `DEFINE`. We are now prepared to describe the Lisp VSA.

The Lisp VSA

The logic of a LISP VSA is straightforward. We are going to map all the elementary functions of Lisp to operations in a vector-symbolic architecture. This proves remarkably straightforward. An interpreter for the Lisp VSA reads a Lisp program and, instead of executing, e.g., `CONS`, over two bytes in order to make a two-byte array in memory, it will apply the vector-symbolic `CONS` over two vectors in order to create a joint representation of the pair, as a single vector.

One detail that has not been addressed is how atomic symbols are to be constructed. As the dimension of a space grows, fixing one vector and choosing another vector arbitrarily, the expected value of their similarity goes to 0, and vectors with nonnegligible similarity are exceedingly rare (Kainen & Kůrková, 1993). In holographic reduced representations (Plate, 1995), we sample vectors on $\mathbb{R}^n$ from a normal distribution, with $\mu = 0$ and $\sigma = \frac{1}{\sqrt{n}}$, producing vectors v with $\mathbb{E}[||v||] = 1$. Thus, we can have many more base symbols than dimensions in the space, all *nearly* linearly independent[4], which would not be the case if they were truly orthogonal. For Euclidean vectors, it typical to see $n \in \{2^k, k \in [6,12]\}$.

Kanerva (2014) suggested representing lists by permuting one operand then adding the two operands together. By keeping a fixed permutation in memory, the united representation is most similar to its unpermuted operand by default, and then, by applying the inverse permutation, winds up most similar to its permuted operand, with fixed permutation Π.

$$\mathbf{cons}(a,b) = a + \Pi(b)$$
$$\mathbf{car}(c) = \mathcal{M}(c)$$
$$\mathbf{cdr}(c) = \mathcal{M}(\Pi^{-1}(c))$$

This method has some flaws if we care about retrieval, however. Taking advantage of the property just described, either operand can be retrieved from a tuple very simply. But, problematically, if one wishes to make a list of arbitrarily many elements, one needs to store sublists in memory. Once

[3] A more Lisp-appropriate definition might have been written such that arguments do not have to be curried, but this version was chosen for ease of presentation.

[4] We can calculate the expected variance of pairwise $\mathbf{sim}(a,b)$ for an arbitrary overcomplete basis B (i.e., a finite sample of $\mathbb{R}^n$ where $|B| > n$) with $a \neq b \in B \subset \mathbb{R}^n$ exactly, but that calculation is outside the scope of this paper. A commonly used "margin of safety" expects $\mathbf{sim}(a,b) \in (-0.2, 0.2)$, but for $n \geq 512$ the actual expected variance is much smaller, even for large $|B|$.

one stores a list in memory, however, the vector to which that list is most similar is itself. It becomes necessary to do something to tag *both* operands, such that each operand and their disjunction may be reliably distinguished in memory.

Permutation is also a redundant operation if we are implementing a Lisp. Although it is often used to make the conjunction operator $\otimes$ asymmetric, this behaviour is not necessary if we are implementing a Cartesian closed category, as the role of taking an operation that builds a joint representation of two operands, and making it asymmetrical, is already satisfied in the VSA algebra. In the Lisp VSA, the product's job is making a combined representation of two objects that is dissimilar to either, but both remain retrievable, using their pair and a cleanup memory, which is exactly what permutation is doing in Kanerva. As such, we will leave out the permutation operator and work just with $+, \otimes, \overline{\otimes}, \mathbf{sim}, \mathcal{M}$,

We define additional operators for convenience. $\mathcal{F}$ is another cleanup memory, that separates global function bindings from the inventory of retrievable expressions. $\nu(v) = \frac{v}{||v||}$ divides v by its magnitude, such that $\mathbf{sim}(\nu(v), v) = 1$, $||\nu(v)|| = 1$. $\oplus$ is a variant addition operator that saturates on both an upper and lower threshold, respectively $\theta_\uparrow$, $\theta_\downarrow$:

$$\begin{aligned} a \oplus b = &(\mathbb{E}[||a||] > \theta_\uparrow)a \\ &+ (\mathbb{E}[||a||] < \theta_\downarrow)b \\ &+ (\theta_\downarrow \leq \mathbb{E}[||a||] \leq \theta_\uparrow)\,\nu(a+b) \end{aligned} \tag{1}$$

Because $\oplus$ is defined using three mutually exclusive cases, the operands a and b can be lazily evaluated, such that the values are only computed if they are needed. Defined assuming lazy evaluation, $\oplus$ is a useful operator for writing recursive definitions. Setting $\theta_\uparrow$ and $\theta_\downarrow$ respectively a little under 1 and a little over 0, and the expected magnitude of all vectors is 1, expressions written with $\oplus$ are meant to be read as evaluating the left operand if a test multiplying it succeeds (the test yields a scalar value $\alpha = \mathbb{E}[||a||] > \theta_\uparrow$), and evaluating the right operand if it fails ($\alpha < \theta_\downarrow$). Several additional atomic expressions are used in the preceding definitions, notably L and R, which mark the left hand and right hand sides of a tuple, as well as φ, which marks that a vector is nonatomic.

$\mathbf{f}(a)$ marks a call to programmer-defined function $\mathbf{f}$, and requires some special treatment, as $\mathbf{cons}(\mathbf{f}, a)$ is equivalent in our notation to $\mathbf{f}(a)$. What the notation means is that the interpreter should leave the list including $\mathbf{f}$ as an unevaluated expression if $\mathbf{f}$ is not in the function namespace.

Below are the definitions of the Lisp VSA. Single unbolded lowercase letters refer to variables that may contain arbitrary Lisp expressions, but typically they are expected to be of a certain form and lead to undefined behaviour when not of that form. Bolded words and letters refer to function names, and are expected to always be atomic. Functions in general are called by simply using their name, and so all function calls are marked by atomic symbols at the head of a list of arguments, except in the case of lambda expressions, which are three-element lists. Lisp expressions of the form `(F a b ...)` are translated to our notation as $\mathbf{f}(a, b, ..., t)$, where the last element t is always the tail of the list of arguments. Recalling that lists are recursively nested tuples, `(F a b ...)` is equivalent to `(F . (a . (b ...)))`, and likewise $\mathbf{f}(a)(b)... = \mathbf{f}(\mathbf{cons}(a, \mathbf{cons}(b, ...))) = \mathbf{cons}(\mathbf{f}, (\mathbf{cons}(a, \mathbf{cons}(b, ...)))$ in our notation. Programmer-defined functions are always fully curried. The special case of `(F)` is, following the definition of lists, equivalent to $\mathbf{f}(\text{NIL})$. Therefore, we never technically have functions on no arguments. Here are the definitions:

$$\mathbf{cons}(a, b, \_) := \nu(L \otimes a + R \otimes b + \varphi) \mid \mathcal{M} \leftarrow\!\!+ a, b \tag{2}$$

$$\mathbf{car}(a) := \mathcal{M}(L \overline{\otimes} a) \tag{3}$$

$$\mathbf{cdr}(a) := \mathcal{M}(R \overline{\otimes} a) \tag{4}$$

$$\mathbf{eq}(a, b, \_) := \mathbf{sim}(a, b)T + (1 - \mathbf{sim}(a, b))F \tag{5}$$

$$\begin{aligned} \mathbf{atom}(a, n) := &\ \mathbf{sim}(n, \text{NIL})\mathcal{M}(\mathbf{sim}(a, \varphi)F \\ &+ (2\theta_\downarrow - \mathbf{sim}(a, \varphi))^+ T) \\ &+ (2\theta_\downarrow - \mathbf{sim}(n, \text{NIL}))^+ F \end{aligned} \tag{6}$$

$$\mathbf{define}(a, e, \_) := * \mid \mathcal{F} \leftarrow\!\!+ \mathbf{cons}(a, e) \tag{7}$$

$$\begin{aligned} \mathbf{cond}(r) := &\ \mathbf{sim}(\mathbf{car}(\mathbf{car}(r)), T)\mathbf{cdr}(\mathbf{car}(r)) \\ &\oplus \mathbf{cond}(\mathbf{cdr}(r)) \end{aligned} \tag{8}$$

$$\begin{aligned} (\lambda(x, e))(a) := &\ \mathbf{sim}(x, \text{NIL})e \\ &\oplus \mathbf{sim}(e, \text{NIL})\text{NIL} \\ &\oplus \lambda(\mathbf{cdr}(x), \lambda\mathbf{s}(x, e)(a)) \end{aligned} \tag{9}$$

$$\begin{aligned} (\lambda\mathbf{s}(x, e))(a) := &\ \mathbf{sim}(x, \text{NIL})e \\ &\oplus \mathbf{sim}(e, \text{NIL})\text{NIL} \\ &\oplus \mathbf{sim}(\mathbf{car}(x), e)\mathbf{car}(a) \\ &\oplus \mathbf{sim}(\mathbf{atom}(e), T)e \\ &\oplus \mathbf{sim}(\mathbf{car}(x), \mathbf{car}(e)) \\ &\quad \mathbf{cons}(\mathbf{car}(a), \lambda\mathbf{s}(x, \mathbf{cdr}(e))(a)) \\ &\oplus \mathbf{sim}(\mathbf{atom}(\mathbf{car}(e)), F) \\ &\quad \mathbf{cons}(\\ &\quad\ (\lambda\mathbf{s}(x, \mathbf{car}(e)))(a), \\ &\quad\ (\lambda\mathbf{s}(x, \mathbf{cdr}(e)))(a) \\ &\quad) \\ &\oplus \mathbf{cons}(\mathbf{car}(e), (\lambda\mathbf{s}(x, \mathbf{cdr}(e)))(a)) \end{aligned} \tag{10}$$

$$\begin{aligned} \mathbf{f}(a) := &\ \mathbf{sim}(\mathbf{f}, \mathbf{car}(\mathcal{F}(L \otimes \mathbf{f}))) \\ &\quad \mathbf{cons}(\mathbf{cdr}(\mathcal{F}(L \otimes \mathbf{f})), a) \\ &\oplus \mathbf{cons}(\mathbf{f}, a) \end{aligned} \tag{11}$$

With some minor modifications due to simplifications of the specification, the above definitions can be used to implement the Lisp 1.5 interpreter (McCarthy, Appendix B).

What is particularly notable in the above definitions is the frequency and fundamentalness of the use of cleanup memories. Every VSA has a cleanup memory, but usually, the cleanup memory relies on a big matrix M that

stores every known symbol as an approximately unit length row vector. Thus, on $\mathbb{R}^n$, the cleanup memory is $\mathcal{M}(p) = \textbf{argmax}(pM^T)M$, where p is a probe vector to be "cleaned up", by retrieving its nearest neighbour from memory. Because of the historic difficulty of implementing both time- and space-efficient cleanup memories, and a low appraisal of the biological significance of "memory is a big lookup table and you test every entry in order to retrieve the one you want", the choice of cleanup memory being used by any given VSA is an embarrassment one typically glosses over (e.g. while Kanerva, 2014 discusses cleanup memories, they are not explicit in his algebraic notation, and are left as a black box in his diagrams). We emphasize the explicit notation of cleanup memories, because they are essential for achieving features of VSAs in frequent day-to-day use, because different cleanup memories have distinctive computational properties that may fit some applications better than others, and because memories with certain computational properties are essential to achieving Turing-completeness in our Lisp.

Kanerva (2014), following Eliasmith et al. (2012), refers to the vectors of a VSA as "pointers". That is because, partly, a probe in the memory can be taken to "reference" its nearest neighbour; a trace can be looked up using any of the points in space near it. Traditional memory pointers similarly "probe" memory, though, in general, what is at the probed memory location need not have high mutual information with the probe. Variant cleanup memories can also be defined that are similarly *heteroassociative*, making probes much more like traditional pointers. One glaring flaw appears in the pointer analogy, however: Where d is the number of stored traces, retrieval from computer memory is $O(\log(d))$, if d is close to the total available space ($O(1)$ if significantly less). Probing a traditional cleanup memory is at least $O(d)$. In fact, because probing memory *requires* traversing all stored traces to test for similarity, the lookup is also at least $\Omega(d)$. It is not an issue of principle versus practice either; because VSAs use vectors of extremely high dimensionality, comparisons take a long time, and because one is often storing thousands or millions of vectors in memory for practical applications, one is really getting one's n's worth of comparisons in.

So What is to be Done?

Neubert et al. make a second apparent misattribution to Kanerva (2014), which is also fruitful to pretend was written as attributed. They suggest the possibility of another type of cleanup memory: an attractor neural network. In such networks, information is often (though not always) distributed over the network's weights, which makes them robust to noise or damage, as the vector representations of VSAs are robust. Attractor networks feature interacting cells converging to stable patterns over time, a tantalizingly brain-like property. However, most attractor networks in use are no more time or memory efficient than a big matrix. The Hopfield network (Hopfield, 1982, made continuous in Hopfield, 1984) has a storage capacity of $O(n)$ with respect to its input dimensions. Hopfield networks work almost *exactly like* the big matrix format, with a different *activation function* (above, our activation function was **argmax**) and the proviso that the network's outputs may be fed back into it, until it converges[5].

Another appealing option is to use the match networks of Grossberg (2021), as they've seen some success in modelling human brains, and also claim to solve retroactive interference. Unfortunately, they also look like the big matrix approach of before[6], and they eliminate retroactive interference by "gating" gradient descent, with a function that updates only on one row at a time, prohibiting the storage of more than $O(n)$ traces or retrieving them in less than $\Omega(d)$ time, if traces are assumed to have low mutual information.

If we relax the requirement that traces be near-orthogonal, better results may be obtained. Ororbia and Kelly (2023) use a continuous variant of MINERVA2 (Hintzman, 1984) as the memory system of a reinforcement learning agent. MINERVA2 also resembles the "big matrix" memory: $\mathcal{M}(p) = (pM^T)^\rho M$ where ρ is an odd integer power. ρ can be allowed to be a real number using the variant equation $\mathcal{M}(p) = \textbf{sgn}(\xi)(\textbf{sgn}(\xi)\xi)^\rho M$ where $\xi = pM^T$. Traces are still inserted row-wise, but Ororbia and Kelly do not expect to retrieve traces exactly as-stored, and rather interpolate between stored traces using probes similar to several of them. They also employ a forgetting mechanism: when information is unused, it fades out of memory. Thus, the size of memory is capped, without running out of space. Their system is not strictly vector-symbolic; there is no syntactic manipulation. But, if atomic symbols are allowed to be correlated and we permit forgetting, similarly advantageous properties may be usable. One reason to specify the exact cleanup memory used in one's VSA is that its space, timing, and information loss characteristics are very relevant topics for study. Different tradeoffs of characteristics might significantly affect the behaviour of the VSA in a specific use-case.

For our vector-symbolic Lisp, MINERVA2 is inadequate, at least without significantly modifying the specification. Let us reflect on the general form of the cleanup memories we have looked at: $\mathcal{M}(p) = \tau(\sigma(\beta pM^T)M + q)$ with activation function σ, normalization function τ, scalar constant β, and some added factor q. In most of the preceding cases, τ has been the identity, $\beta = 1$ and $q = 0$. M is an $m \times n$ matrix, where m is typically $O(d)$, each row storing one n-dimensional trace. Ideally, we want M to distribute information about retrievable traces, as in the case of Hopfield networks and MINERVA2; we want to retrieve traces exactly as-stored, as in the big matrix case; we want to store a number of traces that is superlinear relative to the input dimension n, both for the sake of having a cleanup memory with a great capacity, and for improving the memory's timing characteristics. Capacity is important, because our Lisp relies so heavily

[5]Hopfield called for updating neuron activations at random, but both bulk and random updates converge to the same outcomes.

[6]Converging to $\mathcal{M}(p) = \tau(\sigma(pM^T)M + p)$, where σ behaves similarly to **argmax**, and $\tau(v) = \varsigma(\frac{v}{||v||_1})$ with logistic function ς.

on memory to allow for a program to be written with arbitrary functions, and because the set of functions on a S is the powerset $\mathcal{P}(S)$; if arbitrary programs are allowed, we need to be able to store and retrieve arbitrary sequence-to-sequence maps, requiring a memory system that is at least exponential in storage capacity with respect to the input dimension. Just one candidate cleanup memory fits the bill: the modern Hopfield Network (MHN) (Ramsauer et al., 2020).

Mathematically demonstrating an exponential storage capacity and therefore $O(\log(d))$ lookup time, Ramsauer et al. describe a neural network with a familiar form:

$$\mathcal{M}(p) = \textbf{softmax}(\beta\ pM^T)M$$

Information is distributed because stored traces are encoded in M using gradient descent (no gating). Special cases are the big matrix variant (equal to the limit of the MHN equation as $\beta \to \infty$) and a linear memory system (with $\beta = 0$). If β is allowed to be a function of ξ, then MINERVA2 is also a special case (after normalization) with $\beta = \frac{\rho \log(x)}{x}$ (demonstrable algebraically), but this is not a very useful example. The special cases serve to indicate that the capacity of the network is sensitive to the choice of β. Given an arbitrary program, then, some amount of tuning is necessary to make the MHN store what needs to be stored.

However, MHNs obtain an exponential capacity by way of encoding by gradient descent, so, while capacity is large and retrievals are $O(1)$, encoding is $O(n)$, and requires a backup of all stored traces, if retroactive interference is to be avoided. As it is possible for many traces to be stored at runtime in the VSA Lisp, MHNs present significant drawbacks for us, so long as they depend on a gradient descent learning rule.

And Whatfor This Lisp?

The behaviour of vector-symbolic architectures is very sensitive to the choice of cleanup memory. While memory characteristics are not typically used to demonstrate Turing-completeness, Turing-completeness makes sufficiently great demands of memory that reasonable performance requires specific memory characteristics. As the computing applications of VSAs expand, studying these characteristics and making good tradeoffs will be very important.

One application that stands out is suggested by Tamkin, Taufeeque, and Goodman (2023), who trained a transformer network to exhibit states that were decomposible into "monosemantic" vectors S. The semantic content of the network's output was manipulated by adding or subtracting features drawn from S. As such, transformer states may be made to behave as additive compositions of atomic vector symbols, of the sort syntactically composable by VSAs.

Creating a vector-symbolic Lisp has been alluded to a few times, in particular by Kanerva (2014), Smolensky (1990), and Legendre, Miyata, and Smolensky (1990). The appeal is obvious to cognitive scientists and explicit in Smolensky: We think that brain states have syntax, and we know information is distributed over them. VSAs are a means to express syntax in terms that may describe brain states, and Lisp instantiates the most general class of syntaxes. Respecting actual neural networks, Chen et al. (2020) and Smolensky, McCoy, Fernandez, Goldrick, and Gao (2022) have put syntactic manipulation of network states into practice, with promising results.

Traditional cognitive architectures, such as ACT-R, describe memory states syntactically (Stewart & West, 2006), and take actions according to rules that are sensitive to syntactic features. Such states have already been described in terms of a VSA (Kelly, Arora, West, & Reitter, 2020), and, taking into account their rules and memory systems, these cognitive architectures are already Turing-complete. However, it is uncommon for these cognitive architectures to treat memory states as arbitrary programs, and attempt to evaluate them. It is notable that these architectures directly descend from an attempt to describe artificial general intelligence (Newell, 1980), that the only formal description of artificial general intelligence (Hutter, 2000) expects states of memory to be arbitrary programs, that recent research obliquely referencing the latter suggests it is at least worth considering that states of human memory are such arbitrary programs (Dehaene, Al Roumi, Lakretz, Planton, & Sablé-Meyer, 2022), and that none of the preceding should be at all surprising, since Turing's universal machine was originally a description of the sorts of things people do in their head, manipulating either their memory or a piece of paper (Turing, 1950). Therefore, it may be necessary to increase the expressivity of memory states in order for existing paradigms to capture some complex human behaviour. To that end, it is only necessary to describe rules that treat certain states of memory as lambda expressions and evaluate them (as we did above). Then, memory may encode arbitrary programs, although, based on the behaviour of our own interpreter (see footnote 1), our simple design is likely not the most efficient that can be achieved.

But what is most striking is that, in several leading functional theories of consciousness, a necessary feature is one's ability to pursue one's goals by reading and manipulating of one's own internal state (Butlin et al., 2023, p. 5, Table 1, properties RPT-1, 2, HOT-2, 3, AST-1, and AE-1). By identifying atomic states, and training a network to represent compositions of states, an arbitrary syntax can be defined over the network's state space, though not all networks can be trained to make all syntaxes useful. If any useful syntaxes are possible, which it seems they are, then the states of some networks can be decomposed into sequences, so sequence-to-sequence models may become capable of reading and writing the very states that control their behaviour. Because VSAs are provably Turing-complete, there are no limits to how they can subject network states to syntactic manipulation, if those syntaxes can be encoded and learned over. If any of the noted theories surveyed in Butlin et al. are correct in requiring such auto-manipulation, *vector-symbolic architectures might even be the gateway to machine consciousness*.

References

Butlin, P., Long, R., Elmoznino, E., Bengio, Y., Birch, J., Constant, A., ... VanRullen, R. (2023). *Consciousness in artificial intelligence: Insights from the science of consciousness.* doi: 10.48550/arXiv.2308.08708

Chen, K., Huang, Q., Palangi, H., Smolensky, P., Forbus, K., & Gao, J. (2020, 13–18 Jul). Mapping natural-language problems to formal-language solutions using structured neural representations. In H. D. III & A. Singh (Eds.), *Proceedings of the 37th international conference on machine learning* (Vol. 119, pp. 1566–1575). PMLR. Retrieved from https://proceedings.mlr.press/v119/chen20g.html

Chomsky, N. (1955). *The logical structure of linguistic theory*. Unpublished doctoral dissertation, University of Pennsylvania. (Published as a monograph by Plenum Press, New York, in 1975)

Dehaene, S., Al Roumi, F., Lakretz, Y., Planton, S., & Sablé-Meyer, M. (2022). Symbols and mental programs: a hypothesis about human singularity. *Trends in Cognitive Sciences*, *26*(9), 751-766. doi: 10.1016/j.tics.2022.06.010

Eliasmith, C., Stewart, T. C., Choo, X., Bekolay, T., DeWolf, T., Tang, Y., & Rasmussen, D. (2012). A large-scale model of the functioning brain. *Science*, *338*(6111), 1202-1205. doi: 10.1126/science.1225266

Fodor, J. A., & Pylyshyn, Z. W. (1988). Connectionism and cognitive architecture: A critical analysis. *Cognition*, *28*(1), 3-71. doi: 10.1016/0010-0277(88)90031-5

Grossberg, S. (2021). *Conscious mind, resonant brain: How each brain makes a mind.* Oxford University Press.

Heddes, M., Nunes, I., Vergés, P., Kleyko, D., Abraham, D., Givargis, T., ... Veidenbaum, A. (2023). Torchhd: An open source python library to support research on hyperdimensional computing and vector symbolic architectures. *Journal of Machine Learning Research*, *24*(255), 1–10. Retrieved from http://jmlr.org/papers/v24/23-0300.html

Hintzman, D. L. (1984, March). MINERVA 2: A simulation model of human memory. *Behavior Research Methods, Instruments, & Computers*, *16*(2), 96–101. doi: 10.3758/BF03202365

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, *79*(8), 2554-2558. doi: 10.1073/pnas.79.8.2554

Hopfield, J. J. (1984). Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the National Academy of Sciences*, *81*(10), 3088-3092. doi: 10.1073/pnas.81.10.3088

Hutter, M. (2000). *Towards a universal theory of artificial intelligence based on algorithmic probability and sequential decision theory.*

Kainen, P. C., & Kůrková, V. (1993). Quasiorthogonal dimension of euclidean spaces. *Applied Mathematics Letters*, *6*(3), 7-10. doi: 10.1016/0893-9659(93)90023-G

Kanerva, P. (1996). Binary spatter-coding of ordered K-tuples. In C. von der Malsburg, W. von Seelen, J. C. Vorbrüggen, & B. Sendhoff (Eds.), *Artificial neural networks — ICANN 96* (pp. 869–873). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/3-540-61510-5_146

Kanerva, P. (2014). Computing with 10,000-bit words. In *2014 52nd annual Allerton conference on communication, control, and computing (Allerton)* (p. 304-310). doi: 10.1109/ALLERTON.2014.7028470

Kelly, M. A., Arora, N., West, R. L., & Reitter, D. (2020). Holographic declarative memory: Distributional semantics as the architecture of memory. *Cognitive Science*, *44*(11), e12904. doi: 10.1111/cogs.12904

Kleyko, D., Rachkovskij, D., Osipov, E., & Rahimi, A. (2023, jan). A survey on hyperdimensional computing aka vector symbolic architectures, part ii: Applications, cognitive models, and challenges. *ACM Comput. Surv.*, *55*(9). doi: 10.1145/3558000

Kleyko, D., Rachkovskij, D. A., Osipov, E., & Rahimi, A. (2022, dec). A survey on hyperdimensional computing aka vector symbolic architectures, part i: Models and data transformations. *ACM Comput. Surv.*, *55*(6). doi: 10.1145/3538531

Legendre, G., Miyata, Y., & Smolensky, P. (1990). Distributed recursive structure processing. In R. Lippmann, J. Moody, & D. Touretzky (Eds.), *Advances in neural information processing systems* (Vol. 3). Morgan-Kaufmann. Retrieved from https://proceedings.neurips.cc/paper_files/paper/1990/file/8cb22bdd0b7ba1ab13d742e22eed8da2-Paper.pdf

Li, M., & Vitányi, P. (2008). *An introduction to Kolmogorov complexity and its applications* (Vol. 3). Springer.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM*, *3*(4), 184–195. doi: 10.1145/367177.367199

Meier, C. (2023). *Vector symbolic architectures in Clojure.* Retrieved from https://youtu.be/j7ygjfbBJD0 (Clojure/Conj 2023)

Neubert, P., Schubert, S., & Protzel, P. (2019). An introduction to hyperdimensional computing for robotics. *KI - Künstliche Intelligenz*, *33*(4), 319–330. doi: 10.1007/s13218-019-00623-z

Newell, A. (1980). Physical symbol systems. *Cognitive Science*, *4*(2), 135-183. doi: 10.1016/S0364-0213(80)80015-2

nLab authors. (2024, February). *cartesian closed category.* https://ncatlab.org/nlab/show/cartesian+closed+category. (Revision 42)

Ororbia, A. G., & Kelly, M. A. (2023). Maze learning using a hyperdimensional predictive processing cognitive architecture. In B. Goertzel, M. Iklé, A. Potapov, & D. Ponomaryov (Eds.), *Artificial general intelligence* (pp. 321–331). Cham: Springer International Publishing. doi: 10.1007/978-3-031-19907-3_31

Plate, T. (1995). Holographic reduced representations. *IEEE Transactions on Neural Networks*, *6*(3), 623-641. doi: 10.1109/72.377968

Ramsauer, H., Schäfl, B., Lehner, J., Seidl, P., Widrich, M., Adler, T., ... Hochreiter, S. (2020). *Hopfield networks is all you need.* arXiv. doi: 10.48550/ARXIV.2008.02217

Smolensky, P. (1990). Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial Intelligence*, *46*(1), 159-216. doi: 10.1016/0004-3702(90)90007-M

Smolensky, P., McCoy, R., Fernandez, R., Goldrick, M., & Gao, J. (2022). Neurocompositional computing: From the central paradox of cognition to a new generation of ai systems. *AI Magazine*, *43*(3), 308-322. doi: 10.1002/aaai.12065

Stewart, T. C., & West, R. L. (2006). Deconstructing act-r. In *Proceedings of the seventh international conference on cognitive modeling* (Vol. 1, pp. 298–303).

Tamkin, A., Taufeeque, M., & Goodman, N. D. (2023). *Codebook features: Sparse and discrete interpretability for neural networks.* arXiv. doi: 10.48550/arXiv.2310.17230

Turing, A. M. (1950). Computing machinery and intelligence. , *59*, 433–460. Retrieved from `http://cogprints.org/499/`

Widdows, D., & Cohen, T. (2014, 11). Reasoning with vectors: A continuous model for fast robust inference. *Logic Journal of the IGPL*, *23*(2), 141-173. doi: 10.1093/jigpal/jzu028