Computing with 10,000-Bit Words*

Pentti Kanerva[1]

Abstract— **Today's computers compute with numbers and memory pointers that hardly ever exceed 64 bits. With nanotechnology we will soon be able to build computers with 10,000-bit words. How would such a computer work and what would it be good for? The paper describes a 10,000-bit architecture that resembles von Neumann's. It has a random-access memory (RAM) for 10,000-bit words, and an arithmetic–logic unit (ALU) for "adding" and "multiplying" 10,000-bit words, in abstract algebra sense. Sets, sequences, lists, and other data structures are encoded "holographically" from their elements using addition and multiplication, and they end up as vectors of the same 10,000-dimensional space, which makes recursive composition possible. The theory of computing with high-dimensional vectors (e.g. with 10,000-bit words) has grown out of attempts to understand the brain's powers of perception and learning in computing terms and is based on the geometry and algebra of high-dimensional spaces, dynamical systems, and the statistical law of large numbers. The architecture is suited for statistical learning from data and is used in cognitive modeling and natural-language processing where it is referred to by names such as Holographic Reduce Representation, Vector Symbolic Architecture, Random Indexing, Semantic Indexing, Semantic Pointer Architecture, and Hyperdimensional Computing.**

I. Introduction

The conventional von Neumann computer architecture has been phenomenally successful and affects nearly all aspects of our lives. The idea in itself is powerful but its success is equally due to phenomenal advances in semiconductors—it would not have happened with vacuum tubes—and to the development of programming languages. However, computing today is easily traced back to computing in the vacuum-tube era. We compute with numbers and pointers that seldom exceed 64 bits, and store them in a random-access memory, which also stores the program code. When we need something beyond numbers, such as vectors, matrices, trees, lists, general data structures, and the program code, we make them from numbers and memory pointers. The computer's circuits are designed to work with these numbers and pointers and to run the program code. In terms of computing fundamentals, very little has changed in over half a century, and programming languages from over 50 years ago are still used today.

With nanotechnology we are entering an era that can equal in significance the advent of the transistor. It is due to the possibility of building ever larger circuits that consume minuscule power compared to present-day semiconductor circuits. The technology will no doubt benefit conventional computing, for example in the form of very large nonvolatile memories, but it will also make possible new computing architectures that require very large circuits. The true significance of nanotechnology is likely to be realized through these architectures. Here we describe one such, the possibility of computing with high-dimensional vectors, or "hyperdimensional" to stress the truly high dimensionality that is at its core.

Ideas for high-dimensional computing have grown out of attempts to understand nervous systems and brains in computing terms, fostered in cognitive-science and artificial-neural-networks research. The vectors can be binary or real or complex so long as their dimensionality is in the thousands. The idea is explained conveniently with reference to computing with 10,000-bit words. It differs from conventional computing mainly by having "arithmetic" operations and memory for 10,000-bit words. The paper will show how to compute with them, what the underlying mathematics is, and what the potential benefits are vis-a-vis conventional computing.

II. Artificial Intelligence as the Backdrop

Computers were invented for calculating with numbers. Yet the idea that the brain is a kind of computer and that computers could be intelligent is as old as the stored-program computer (McCulloch & Pitts 1943; Turing 1950; McCarthy et al. 1955; von Neumann 1958; Rosenblatt 1958; Feigenbaum & Feldman 1963), and Artificial Intelligence or Machine Intelligence became a part of computer science taught in colleges and universities. The traditional AI approach is symbolic, based on explicit rules and logical reasoning and realized in programming languages such as Lisp. Lisp is a prime example of computing with pointers, which are addresses to memory (they are also numbers). A single pointer of no more than 32 bits can represent arbitrarily large and complex data that is encoded with further pointers and numbers held in the computer's memory. A 10,000-bit word can similarly be thought of as a memory pointer, but it is more than a traditional pointer, as we will see below. AI has been successful in areas where the rules and logic can be clearly defined, as in expert systems of many kinds.

Artificial Neural Nets represent the second major approach to artificial intelligence—neural nets are also referred to by "Parallel Distributed Processing" (Rumelhart & McClelland 1986) and "Connectionism." Learning is statistical, from data, and is captured in the coefficients or connection weights

*This work was supported by the Systems On Nanoscale Information fabriCs (SONIC) program organized by the University of Illinois, Urbana–Champaign, and funded by the Department of Defense and U.S. semiconductor and supplier companies. The author is grateful for the opportunity to participate in the program.

[1]Redwood Center for Theoretical Neuroscience, UC Berkeley pkanerva@berkeley.edu

of large matrices. Early models dealt with associative memory (e.g., Kohonen 1977; Hinton & Anderson 1981; Hopfield 1982; Kanerva 1988) and pattern classification (e.g., Cortes & Vapnik 1995), and the most actively researched area today is deep learning (Schmidhuber 2014). While neural nets are good for learning from data, they are poor for logical reasoning and manipulating structured information, which are the forte of traditional symbolic AI. The two are complementary, but neither one alone is sufficient for producing the artificial intelligence that the early pioneers envisaged.

High-dimensional computing is an attempt at combining the strengths of the two. Hinton (1990) introduced the idea of a reduced representation, as a prerequisite for dealing with compositional structure (i.e., with lists, trees, etc.—the staple of symbol processing), and Smolensky (1990) described the tensor-produce binding as a means to associate variables with values when distributed representation is used. Plate (1991, 2003) combined the ideas in Holographic Reduced Representation (HRR) by reducing the tensor product with circular convolution, thus yielding a model of computing with fixed-width vectors. Computing with 10,000-bit words is a special case, namely, when the vector dimensions are binary (in Plate's HRR they are real or complex). Binary representation is the simples for explaining the idea.

III. Computing with 10,000-Bit Words vs. 64-Bit Words

10,000-bit computing is essentially similar to conventional computing, so it is important to understand why the differences would make a difference. In purely abstract terms a 10,000-bit computer is no more powerful than a 64-bit computer, or the Universal Turing Machine for that matter. The question is, what can be computed efficiently?

A. Memory

A random-access memory (RAM) is an essential part a computer. It is an array of registers that are referred to and addressed by their position in the array. An address can refer to an 8-bit byte or to a wider (16–64-bit) word. A megabyte of memory means 8 million bits of storage. To address a million bytes we need 20 bits, to address a billion bytes we need 30 bits. It is rather obvious that a 64-bit word can address more memory than a computer would ever have.

The memory for the 10,000-bit computer stores 10,000-bit words. Moreover, it is addressed by 10,000-bit words, defying the notion that there be a register—a physical memory location—for each possible address. However, it matters only that there be enough physical memory for the storage needs of a system's lifetime, and that the information be available "directly" in RAM-like fashion rather than serially as on a tape. A high-dimensional RAM could have millions of locations, with each location capable of storing 10,000 bits or small integers.

Memories of that kind have been studied as neural nets and are called associative or content-addressable (Baum, Moody & Wilczek 1988; Kanerva 1988; Stewart, Tang & Eliasmith 2011). Addressing the memory with a 10,000-bit word selects multiple locations, although but a tiny fraction of all the locations there are (addressing an ordinary RAM selects exactly one lactation). A word is stored in memory by adding it into the selected locations and is retrieved by averaging over the contents of the selected locations.

This kind of memory has three important properties: (1) it allows random access to an address space that is much larger than the physical memory, (2) addressing can be approximate, and (3) it affords recognition. The data retrieved from memory includes an estimate of its reliability—whether the exact or a similar address has previously been used for storing data. The memory can be used for finding the best match to a retrieval cue in the set of stored patterns.

B. Arithmetic

When we think of computing, we naturally think of the addition and multiplication of numbers. They are used to calculate values of functions from the values of their arguments, which was the root purpose of computers. They are also used to calculate addresses to memory when data are stored in and retrieved from composite structures such as vectors and matrices; in other words, addition and multiplication are used to make pointers.

Addition and multiplication so dominate computing that we build specialized circuits for them. The size of the circuit—its width in bits—is built to match the computer's word size, 32 and 64-bit architectures being common today. By analogy, the 10,000-bit architecture needs operations for 10,000-bit words, such as addition and multiplication but taken in a more general, algebraic, sense (note: we are not talking about addition and multiplication of numbers with 10,000-bit precision). References to addition and multiplication below are in this more general sense.

The rules of arithmetic with numbers are a consequence of addition and multiplication forming an algebraic field. The operations on 10,000-bit words (on high-dimensional vectors at large) can likewise be used for computing when they form a field over the vector space, or *approximate a field*. At least the following should be approximately true: (1) addition is commutative, (2) multiplication is invertible, (3) multiplication distributes over addition, and (4) vectors can be compared for similarity (the space has a metric). A representational system that satisfies such criteria can serve as the basis of computing whether its elements be numbers or vectors or mathematical objects of some other kind. Here we use binary vectors to demonstrate addition and multiplication and to discuss further properties of the space and its operations.

1) Distance: Hamming distance is a natural way to compare binary vectors for *similarity*. In 10,000 dimensions nearly all vectors are dissimilar to, or approximately 5,000 bits away from, any given one: only a billionth of the space—of all 10,000-bit vectors—is within 4,700 bits away (and a billionth further than 5,300 bits away). A randomly drawn vector is dissimilar—nearly orthogonal—to any already selected vector with very high probability even if millions of

vectors have been selected already. These statistics follow from the binomial distribution with n = 10,000 and $p = 0.5$ and its approximation by the normal. The mean is 5,000 and the standard deviation is 50, and so most of the space is 100 standard deviations away from any given point. This is often referred to as "concentration of measure."

2) Addition: For addition of two or more binary vectors we use componentwise majority rule: their arithmetic sum vector is thresholded at half the number of vectors. Adding an even number of binary vectors can produce ties that need braking, to end up binary. Ties can be broken at random or with a pseudorandom function of the argument vectors. The sum vector can represent a set of its arguments and it is *similar*—close in Hamming distance—to each of the argument vectors. The similarity is strong for sums of fewer than a dozen vectors and statistically significant for over a thousand.

3) Multiplication: For multiplication we use componentwise Exclusive-Or (XOR, addition modulo 2, $*$). Contrary to addition that produces vectors similar to its arguments, multiplication produces *dissimilar* vectors—the product vector ends up "somewhere else" in the space. This makes multiplication useful for variable binding, which is a key to computing with structured data (with lists, trees, sequences, etc.—things Lisp excels at). XOR is its *own inverse* and it *distributes* over the thresholded sum; the distribution is exact when the sum is of an odd number of binary vectors, approximate when their number is even. Furthermore, XOR *preserves distance*: the Hamming distance between $A * X$ and $A * Y$ is exactly the same as between X and Y for any binary vectors A, X, and Y. All these properties play a role in encoding and decoding structure.

C. Representing Structured Data

The following example shows the encoding and decoding of composite structure with addition and multiplication. Consider a data record of three variables or "fields" x, y, z with values a, b, c for representing '$(x = a)$ and $(y = b)$ and $(z = c)$'. The values could be names or numbers or pointers to further structure, for example. Traditional representation reserves a separate memory location for each of the variables. Their identity is implicit—it is hidden in the program code.

The corresponding 10,000-bit representation is depicted in Figure1. The variables and values are represented by 10,000-bit vectors X, Y, Z, A, B, C. They could be the result of some previous processing or they could be chosen at random. We will assume them to be random. The variable x is bound to the value a with multiplication, thus $X * A$ represents the fact that $x = a$. The other two fields are encoded similarly with $Y * B$ and $Z * C$. Because binding is done with multiplication, the three new vectors are dissimilar from any of the six for the variables and the values, and also from each other. Next the three are combined into a singe 10,000-bit vector with addition, yielding

$$H = [(X * A) + (Y * B) + (Z * C)]$$

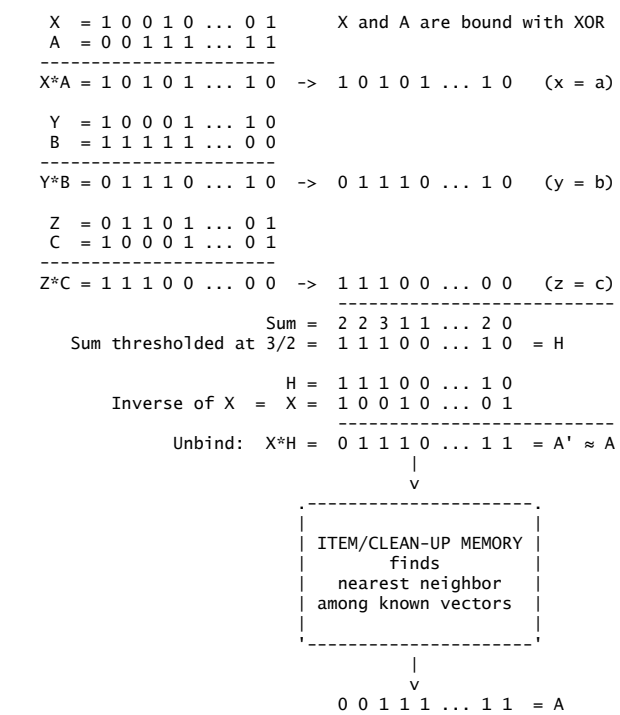

Fig. 1. Encoding and decoding in distributed representation. '$(x = a)$ and $(y = b)$ and $(z = c)$' is encoded into the vector H, and the value of X is extracted from H by "unbinding" followed by "clean-up." X, Y, Z, A, B and C are random 10,000-bit vectors chosen to represent x, y, z, a, b and c, and $*$ is componentwise Exclusive-Or.

where the bracket around the sum denote thresholding. The representation is called *holographic* or *holistic* because the fields are superposed over each other—there are no spatially identifiable fields.

Decoding makes use of inverse multiplication. From the bound pair $X * A$ we can find the vale of X by multiplying with the inverse of X, which for XOR is X itself: $X * (X * A) = (X * X) * A = A$ (the two Xs cancel out). We can also find the value of X in the composite vector H by multiplying it with (the inverse of) X. This follows from the facts that a thresholded sum is similar to each of its arguments and that XOR preserves distance. Since H is similar to $X * A$, $X * H$ is (equally) similar to $X * X * A$, which equals A. Because the vectors are high dimensional, the vector most similar to $X * H$ among all the vectors stored in a system's memory is A with very high probability. It could be retrieved from a content-addressable memory as the best match.

D. Permutation

Permutation of the coordinates is an extremely simple, extremely useful operation and is an essential part of the architecture (Gayler, 1998). It can be used for encoding and decoding sequences, for example. A permutation can be represented formally as a multiplication of a vector with a matrix—the permutation matrix has exactly one 1 in each row and in each column. Permutations in themselves embed rich algebraic structure but since they are represented by matrices they are not part of the field structure described

by the addition and multiplication of binary vectors, and so we do not compute with them in the same sense. Of cause it is possible to define systems of computing where the basic elements are matrices, including permutations, and the operations would be between matrices rather than between vectors. Here we use specific permutations merely as constant operators on vectors.

A permutation resembles multiplication (with XOR) in the sense that it distributes over vector addition—it distributes over any coordinatewise operation including multiplication with XOR—it preserves distance, and it is invertible. However, it is not its own inverse: permutation of a permutation is another, different permutation, which makes it suitable for encoding sequences.

The circuit for a permutation and its inverse is very simple: it is realized by wires that connect each coordinate to some other coordinate in a one-to-one fashion. In a 10,000-bit architecture, that means 10,000 wires. A random permutation is one where the one-to-one connections are made at random. Much like with multiplication with XOR, a randomly permuted vector is dissimilar to the original vector.

E. Representing Sequences

The representation of sequences involves encoding and memory storage. A key idea in symbolic computing is that a fixed-width vector—a single word—can represent a single entity as well as one composed of many such, which is accomplished with pointers. A sequence can then be encoded with records or cells consisting of two variables, one for an element of the sequence and the other for a pointer to the next cell, which represents the rest of the sequence. The variables are accessed by "car" and "cdr" in Lisp. It is possible to define a 10,000-bit lisp on the same principle, by encoding each cell holographically as described above and by choosing vectors at random to serve as "pointers" to the next cell. It is unclear, however, whether a lisp of that kind would have any advantage over the Lisp we have now. Here it serves as a point of reference for further discussion.

A sequence should be stored in such a way that it can be found by a short subsequence taken from anywhere, resembling how we can lock onto a familiar tune by hearing any part of it. This can be done by having each element of a sequence serve as the address for the next element, and so the data serve also as the pointer. In this case, 10,000 bits of information would fetch the next 10,000 bits, and so forth, in the manner of a linked list. This allows a sequence to be found by any of its elements, but it breaks down when different sequences contain the same or very similar elements. The problem and a possible solution are discussed at length in the book on sparse distributed memory (Kanerva 1988). The solution involves the addressing of memory with several different delays, and the idea is that a more distant past steers retrieval past points where sequences cross.

A more versatile representation of sequences uses permutation. The idea is to add (i.e., superpose) to each element of the sequence a permuted version of its history, and to store the sum vector in memory with itself as the address—storing in this way is called *autoassociative*. Encoding the sequence of vectors $(A, B, C, D, \ldots)$ with Π as the permutation proceeds as follows:

$$\begin{aligned} S_1 &= A \\ S_2 &= \Pi S_1 + B &&= \Pi A + B \\ S_3 &= \Pi S_2 + C &&= \Pi(\Pi A + B) + C \\ S_4 &= \Pi S_3 + D &&= \Pi(\Pi(\Pi A + B) + C) + D \end{aligned}$$

Distributing the permutations over the sums in S_4 gives

$$\begin{aligned} S_4 &= \Pi\Pi\Pi\Pi A + \Pi\Pi\Pi B + \Pi C + D \\ &= \Pi^3 A + \Pi^2 B + \Pi C + D \end{aligned}$$

which shows that the different permutations keep count of how far back an element appears in the sequence. We can also see, for example, that with S_3 we can find the next element D: by permuting S_3 we get a vector that is similar to S_4, which allows us to retrieve S_4 from memory, and based on it find D; or possibly find D starting with $\Pi B + C$ or with C alone. In other words, a sequence can be found by a short subsequence (note: this is not true with sequences encoded in Lisp; elements of the sequence do not provide random access into the sequence).

Several comments are due regarding sequences encoded with permutation. (1) Decoding relies on the sum being similar and the product dissimilar to the argument vectors. (2) When a 10,000-dimensional permutation Π is chosen at random, it and its powers work reliably with very high probability. They "randomize" their argument vector making it dissimilar (nearly orthogonal) to the vectors recognized by the system and thus act as random noise in a sum vector. (3) We have overlooked normalization to binary in the example above. When and how to normalize, whether and how to weigh the history vector (S_3 is the history vector of D in the example), and whether and how to add elements from the history into S without first permuting them can affect the finding and following sequences and are ready topics for experimentation.

IV. Mining for Meaning with Random Indexing

Random Indexing is a "big-data" algorithm that is ideally suited for high-dimensional computing. It is based on random projections (Johnson & Lindenstrauss 1984; Kaski 1998; Papadimitriou et al. 1998; Kanerva, Kristoferson & Holst 2000; Sahlgren 2005) and is easily explained with reference to language although it is applicable much more widely. In general, it works well with sparse data, such as generated in social networks where the number of individuals is large and keeps changing but any one individual communicates with only a few others. To drive this point home, how many people are there in the world, and how many of them do you know? Yet everyone is connected to everyone else via but a few intermediaries in what is called a "small-world network." The 10,000-bit architecture and random indexing are for dealing with situations like that.

The example from language consists of computing *semantic vectors* for words. A semantic vector represents a word's

meaning so that words with similar meaning (e.g., train and bus) have similar vectors, and words with dissimilar meaning (e.g., train and salami) have dissimilar vectors. The words of a language are like a small-world network where most pairs of words have dissimilar meaning but can be bridged by a small number of similar-meaning words (e.g., train–engine–fuel–food–salami). The similarity measure is based on vector dot-product, and the cosine between vectors is often used.

Semantic vectors are made by reading millions of words of ordinary English and by keeping count of which words occur with which other words, and by encoding the statistics into vectors: a semantic vector for a word combines into a single vector all the contexts in which the word occurs and is therefore also called a *context vector*. This idea has been around a long time, there are different ways of collecting and processing the statistics, and semantic vectors of one kind or another are at the heart of search engines. Semantic vectors are an important tool of computational linguistics and natural-language processing.

Our example is of 10,000-dimensional semantic vectors made with random indexing, but instead of binary we let the vectors be of small integers. The text could be news article on the Internet, for example, or Wikipedia pages. By *vocabulary* we mean all the different words in the text—all unique ASCII strings. We allow for the fact that the vocabulary keeps on growing as more and more text is read, and so one part of the program needs to maintain an up-to-date vocabulary, which is a routine task for traditional programming.

Each word in the vocabulary is represented by *two* 10,000-dimensional vectors, one constant and the other variable. The constant vector is called the word's index vector or *label* and is chosen at random when the word is first encountered (or it can be a pseudorandom function of the word in ASCII). Sparse ternary labels have worked well: a small number of 1s and the same number of -1s (e.g., 15 each) randomly placed among all the 0s. The variable vector is the word's semantic vector. It starts out as the zero-vector and is updated every time the word occurs in the text.

The text is read one (*focus*) word at a time, and a few words before and after are its *context window*. The semantic vector for the focus word is then updated by the random labels of the context words. For example, if the text includes the phrase "the quick BROWN FOX JUMPED over THE LAZY DOG'S back" and the reading has progressed to the word "over," and if the context window is 3 + 3 words wide (shown in capital letters), then the semantic vector for "over" is updated by adding to it the random labels for "brown," "fox," "jumped," "the," "lazy" and "dog's." In practice, words are sometimes changed to their basic form and very frequent words such as "the" are ignored.

Several facets of random indexing deserve pointing out:

1. Random indexing is incremental—no further processing of the semantic vectors is necessary. This is in contrast to semantic-vector algorithms that rely on principal components of the frequency statistics (e.g., Latent Semantic Analysis uses Singular-Value Decomposition).

2. The memory can be reduced further. Here we have fixed the dimensionality of the semantic vectors (at 10,000) but need a new such vector for every new word in the vocabulary. A vocabulary of v words then requires a $v\times$10,000 memory matrix (in which one byte per matrix element is ample). However, the growth of storage can be contained further by random-indexing the memory matrix in both directions. A million-word vocabulary can then be handled with a 10,000 $\times$10,000-byte matrix, and it could accommodate several languages at once. If each random label has 30 non-0s, accessing a word's semantic vector would require the accessing of 30 rows of the memory matrix.

3. Random labels that are sparse and ternary and have an equal number of 1s and -1s are particularly convenient because the expected value of every memory-matrix element will be 0. The matrix elements can have a small dynamic range without major information loss from overflow or underflow.

4. The semantic vectors of the example above are made with the addition operator alone. If also multiplication is used, logical dependencies and structural information can be included in the vectors. This was shown by Jones & Mewhort (2007) by encoding word order with circular convolution, which is a multiplication operator for vectors of reals, and by Sahlgren, Holst & Kanerva (2008) by encoding word order with permutation. Cohen et al. (2011, 2012; Widdows & Cohen, in press) use both addition and multiplication to make semantic vectors from medical abstracts for discovering dependencies between drugs, genes, and diseases. These kinds of vectors seem appropriate for encoding the information that languages convey with grammar, which would be a landmark in computational linguistics.

6. Random indexing is practical on today's computers. A software package for semantic vectors is freely available on the Internet (Widdows & Cohen 2010).

V. Will the Difference Make a Difference?

We have seen so far that computing in 10,000 dimensions has much in common with computing in one dimension—i.e., with numbers—and of course the 10,000-dimensional can be simulated to any desired precision by computing with numbers. The question is of efficiency, of what is practical under constraints on time, material, and energy. Conventional computing puts a premium on high precision, reliability, and speed and pays for it in energy, yet for a multitude of tasks much of that is unnecessary. A prime example is the kind of computing that brains are good at and that can be described as *qualitative* rather than quantitative. The human mind works with concepts, which we cannot really measure but can relate to one another, and the mind arrives at the meaning of things by relating them to some larger context. That kind of computing can be accomplished with a sufficient number of components that are neither fast nor precise, as attested to by the number and variability of neurons in the brain. Computing with 10,000-bit words tries to capture some of the essence of such computing by exploiting new computational concepts derived from high dimensionality (Kanerva 2009). Below are two that have not yet been discussed.

A. 10,000-Bit Words as Memory Pointers

Traditional symbolic AI can be thought of as computing with pointers, which are addresses to memory. Pointers allow us to encode relations among objects and concepts, but a pointer in itself has very little meaning. We can deduce that two identical pointers have the same meanings, but pointers that differ by only one bit usually lead to totally different meanings. In practice, we hardly ever think of the pointers as bit patterns.

The situation is very different in high dimensions. A 10,000-bit vector that encodes data, can also serve as an address to memory, and so we are addressing the memory with an amount of information that is roughly equivalent to a page of text. Furthermore, if two such vectors are similar, their meanings are similar and the parts of memory they address will overlap. The idea is demonstrated in the *Semantic Pointer Architecture* of Eliasmith et al. (2012; it is explained in the Supplementary Materials). What kind of programming is eventually possible with semantic pointers remains to be seen. It is clear, however, that we are dealing with a computational concept for which there is no counterpart in traditional computing.

B. 10,000-Bit Words as Mappings

It is useful to view multiplication as a mapping in the vector space. When multiplication is used for variable binding, as when X is bound to A in the example above (see Figure 1), the vector for the variable effectively maps the vector for the value into a part space unrelated to the two. To recover the value of X from the combined vector H, multiplying it with the inverse of X maps H back into the neighborhood of A.

Mapping-vectors can be learned from examples, giving us a machine-learning algorithm for relational learning and reasoning by analogy (Plate 2003; Kanerva 2000; Emruli & Sandin 2014). In fact, holographic representation blurs the distinction between the formal notion of a variable and the informal idea of a prototype, allowing us to deal with questions such as "what is the dollar of Mexico?" when the literal answer is that no such thing exists. A vector that maps Dollar to Peso can be computed from other pairs of correspondences between the two countries (Kanerva 2010). Vectors that are computed from examples and are used for mapping by simple multiplication are another computational concept for which there is no counterpart in traditional computing.

VI. Conclusion

Computing in high dimensions is based on the rich—and often subtle—geometry and algebra of high-dimensional spaces (Widdows 2004). High dimensionality (10,000-D) is more important than the nature of the dimensions, as addition and multiplication operators of the right kind exist also for real and complex vectors (Plate 2003). Underlying is the notion that all things are represented in the same mathematical space, subject to the same mathematical operations, capable of being combined and composed with things of different kinds, and serving different functions in different contexts, as when Dollar in one instance is United States currency and in another a "variable" that refers to the monetary unit of a country. This hints of the flexibility with which we humans use concepts and analogy in dealing with everything from the mundane to the sublime (Hofstadter & Sander 2013). Unified representation in high dimensions also allows new algorithms to be developed for old problems such as graph-matching (Levy & Gayler 2009).

Holographic/holistic representation makes high-dimensional computing robust and brainlike. To build computers like that in today's technology may be impractical but it can be made practical by nanotechnology, which is poised to produce massive circuits that occupy very little space and consume very little energy. However, nanocircuits are not guaranteed to be flawless. For that, holographic representation can be the answer, because no individual component or a small number of them is vital—information degrades gradually with a growing number of failing components.

What would this kind of computer be good for? Many properties of high-dimensional spaces are a good match to human memory and to the mind's dealing with concepts. We might therefore expect that finding regularities and structure in large heterogeneous data is a natural application. Extending semantic vectors to include grammar is an obvious target, and language-understanding a long-term goal. If we have to live with automated telephone robots, at least we should be able to get reasonable answers to fully articulated questions! High-dimensional representation and computing seem also appropriate for the integration of signals from a multitude of sensors, and so autonomous and semiautonomous robots would be another application. A robot vehicle could train here on earth and then be sent to explore the moons of Jupiter and Saturn, for example.

How would the computer be programmed? It would be an add-on to a regular computer, which would control it, at least to start with because we know how to program regular computers. New ways of programming the new systems, or having them learn, will be discovered over time. That would mimic our experience with the stored-program computer. It replaced computing that was programmed by plugging cords to patch panels that looked like old telephone exchanges. The first stored programs were virtual cords in virtual patch panels, expressed in numbers. That all changed with the invention of symbolic programming, and we no longer think of our programs as rat's nests of patch cords. Some development like that is quite possible as we become familiar with the ways of high-dimensional computing. It is worth noting that we can start now because many high-dimensional algorithms can be used productively on today's computers, random indexing being one such.

Precise calculation will always be needed and we are not going to replace what works now but will extend computing to areas that are handled poorly if at all with today's computers. The extensions are likely into areas that artificial intelligence has tried to master for decades but has lacked the computing architecture for.

References

Baum, E. B., Moody, J. and Wilczek, F. (1988). Internal representations for associative memory. *Biological Cybernetics* 59:217–228.

Cohen, T., Widdows, D., Schvaneveldt, R. and Rindflesch, T. (2011). Finding schizophrenias prozac: Emergent relational similarity in predication space. *Proc. 5th International Symposium on Quantum Interactions (QI'11).* Berlin, Heidelberg: Springer–Verlag

Cohen, T., Widdows, D., Schvaneveldt, R. W., Davies, P. and Rindflesch, T. C. (2012). Discovering discovery patterns with predication-based semantic indexing. *Journal of Biomedical Informatics* 45(6):1049–1065.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine Learning* 20(3):273–297.

Eliasmith, C., Stewart, T. C., Choo, X., Bekolay, T., DeWolf, T., Tang, C. and Rasmussen, D. (2012). A large-scale model of the functioning brain. *Science* 337:1202–1205, 30 November 2012. Supplementary Materials: www.sciencemag.org/cgi/content/full/338/6111/1202/DC1

Emruli, B. and Sandin, F. (2014). Analogical mapping with sparse distributed memory: A simple model that learns to generalize from examples. *Cognitive Computation* 6(1):74–88.

Feigenbaum, E., and Feldman, J., eds. (1963). *Computers and Thought.* New York: McGraw–Hill.

Gayler, R. (1998). Multiplicative binding, representation operators, and analogy. In K. Holyoak, D. Gentner and B. Kokinov (eds.), *Advances in Analogy Research: Integration of Theory and Data from the Cognitive, Computational, and Neural Sciences*, p. 405. Sofia, Bulgaria: New Bulgarian University.

Hinton, G. E. (1990). Mapping part–whole hierarchies into connectionist networks. *Artificial Intelligence* 46(1–2):47–75.

Hinton, G. H. and Anderson, J. A., eds. (1981). *Parallel Models of Associative Memory.* Hillsdale, NJ: Erlbaum.

Hofstadter, D. and Sander, E. (2013). *Surfaces and Essences: Analogy as the Fuel and Fire of Thinking.* New York: Basic Books.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proc. National Academy of Sciences USA* 79(8):2554–2558.

Johnson, W. B. and Lindenstrauss, J. (1984). Extensions of Lipschitz mapping into Hilbert space. *Contemporary Mathematics* 26:189–206.

Jones, M. N., and Mewhort, D. J. K. (2007). Representing word meaning and order information in a composite holographic lexicon. *Psychological Review* 114(1):1–37.

Kanerva, P. (1988). *Sparse Distributed Memory.* Cambridge, MA: MIT Press.

Kanerva, P. (2000). Large patterns make great symbols: An example of learning from example. In S. Wermter and R. Sun (eds.), *Hybrid Neural Systems*, pp. 194–203. Heidelberg: Springer.

Kanerva, P. (2009). Hyperdimensional Computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive Computation* 1(2):139–159.

Kanerva, P. (2010). What we mean when we say "What's the Dollar of Mexico?": Prototypes and mapping in concept space. Report FS-10-08-006, AAAI Fall Symposium on Quantum Informatics for Cognitive, Social, and Semantic Processes.

Kanerva, P., Kristoferson, J. and Holst, A. (2000). Random Indexing of text samples for Latent Semantic Analysis. In Gleitman, L. R. and Josh, A. K. (eds.), *Proc. 22nd Annual Conference of the Cognitive Science Society*, p. 1036. Mahwah, NJ: Erlbaum.

Kaski, S. (1998). Dimensionality reduction by random mapping: Fast similarity computation for clustering. *Proc. IJCNN'98, International Joint Conference on Neural Networks,* vol. 1, pp. 413–418. Piscataway, NJ: IEEE Service Center.

Kohonen, T. (1977). *Associative Memory: A System Theoretic Approach.* New York: Springer–Verlag.

Levy, S. D. and Gayler, R. W. (2009). "Lateral inhibition" in a fully distributed connectionist architecture. In A. Howes, D. Peebles & R. Cooper (eds.), *Proc. 9th International Conference on Cognitive Modeling,* pp. 318–323. Mahwah, NJ: Erlbaum.

McCarthy, J., Minsky, M. L., Rochester, N. and Shannon, C. E. (1955). A proposal for the Dartmouth summer conference on artificial intelligence. 31 Aug. 1955.

McCulloch, W. S., and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics* (4):115–133.

Papadimitriou, C. H., Raghavan, P., Tamaki, H. and Vempala, S. (1998). Latent semantic indexing: A probabilistic analysis. *Proc. 17th ACM Symp. on the Principles of Database Systems,* pp. 159–168. New York: ACM Press.

Plate T. A. (1991). Holographic Reduced Representations: Convolution algebra for compositional distributed representations. In J. Mylopoulos and R. Reiter (eds.), *Proc. 12th International Joint Conference on Artificial Intelligence (IJCAI),* pp. 30–35. San Mateo, CA: Morgan Kaufmann.

Plate, T. A. (2003). *Holographic Reduced Representation: Distributed Representation of Cognitive Structure.* Stanford, CA: CSLI Publications.

Rumelhart, D. E., and McClelland, J. L., eds. (1986). *Parallel Distributed Processing.* Cambridge, MA: MIT Press.

Sahlgren, M. (2005). An introduction to Random Indexing. *Proc. Methods and Applications of Semantic Indexing Workshop at the 7th International Conference on Terminology and Knowledge Engineering, TKE 2005*, August 16, Copenhagen, Denmark.

Sahlgren, M., Holst, A. and Kanerva, P. (2008). Permutations as a means to encode order in word space. *Proc. 30th Annual Conference of the Cognitive Science Society,* pp. 1300–1305. Austin, TX: Cognitive Science Society.

Schmidhuber, J. (2014). Deep learning in neural networks: An overview. Report IDSIA-03-14, The Swiss AI Lab IDSIA, Istituto Dalle Molle di Studi sull'Intelligenza Artificiale, University of Lugano & SUPSI, Manno-Lugano, Switzerland.

Smolensky, P. (1990). Tensor product variable binding and the representation of symbolic structures in connectionist networks. *Artificial Intelligence* 46(1–2):159–216.

Stewart, T. C., Tang, Y. and Eliasmith, C. (2011). A biologically realistic cleanup memory: Autoassociation in spiking neurons. *Cognitive Systems Research* 12:84–92.

Rosenblatt, F. (1958). The Perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review* 65(6):386–408.

Turing, A. (1950). Computing machinery and intelligence. *Mind* 59(236):433–460.

von Neumann, J. (1958). *The Computer and the Brain.* New Haven/ London: Yale University Press.

Widdows, D. (2004). *Geometry and Meaning.* Stanford, CA: CSLI Publications.

Widdows, D. and Cohen, T. (2010). The semantic vectors package: New algorithms and public tools for distributional semantics. *Proc. Fourth IEEE International Conference on Semantic Computing (ICSC).* http://code.google.com/p/semanticvectors/

Widdows, D. and Cohen, T. (in press). Reasoning with vectors: A continuous model for fast robust inference. *Logic Journal of the IGPL.*

Invited paper at the 52nd Annual Allerton Conference on Communication, Control, and Computing
1–3 October, 2014
To appear in the conference proceedings

Tue Oct 7 18:23:46 PDT 2014