

Hyperdimensional Computing: An Introduction to Computing in Distributed Representation with High-Dimensional Random Vectors

Pentti Kanerva

Published online: 28 January 2009
© Springer Science+Business Media, LLC 2009

Abstract The 1990s saw the emergence of cognitive models that depend on very high dimensionality and randomness. They include Holographic Reduced Representations, Spatter Code, Semantic Vectors, Latent Semantic Analysis, Context-Dependent Thinning, and Vector-Symbolic Architecture. They represent things in high-dimensional vectors that are manipulated by operations that produce new high-dimensional vectors in the style of traditional computing, in what is called here hyperdimensional computing on account of the very high dimensionality. The paper presents the main ideas behind these models, written as a tutorial essay in hopes of making the ideas accessible and even provocative. A sketch of how we have arrived at these models, with references and pointers to further reading, is given at the end. The thesis of the paper is that hyperdimensional representation has much to offer to students of cognitive science, theoretical neuroscience, computer science and engineering, and mathematics.

Keywords Holographic reduced representation · Holistic record · Holistic mapping · Random indexing · Cognitive code · von Neumann architecture

Introduction: The Brain as a Computer

In this tutorial essay we address the possibility of understanding brainlike computing in terms familiar to us from

conventional computing. To think of brains as computers responsible for human and animal behavior represents a major challenge. No two brains are identical yet they can produce the same behavior—they can be functionally equivalent. For example, we learn to make sense of the world, we learn language, and we can have a meaningful conversation about the world. Even animals without a full-fledged language can learn by observing each other, and they can communicate and function in groups and assume roles as the situation demands.

This means that brains with different “hardware” and internal code accomplish the same computing. Furthermore, the details of the code are established over time through interaction with the world. This is very different from how computers work, where the operations and code are prescribed in detail from the outside by computer-design engineers and programmers.

The disparity in architecture between brains and computers is matched by disparity in performance. Notably, computers excel in routine tasks that we—our brains—accomplish with effort, such as calculation, whereas they are yet to be programmed for universal human traits such as flexible learning, language use, and understanding.

Although the disparity in performance need not be due to architecture, brainlike performance very likely requires brainlike architecture. The opposite is not necessarily true, however: brainlike architecture does not guarantee brainlike “intelligent” behavior, as evidenced by many kinds of mental illness. Thus, we can look at the brain’s architecture for clues on how to organize computing. However, to build computers that work at all like brains, we must do more than copy the architecture. We must understand the principles of computing that the architecture serves.

P. Kanerva (✉)
Center for the Study of Language and Information, Stanford
University, Stanford, CA 94305, USA
e-mail: pkanerva@csli.stanford.edu

An Overview

We will discuss a set of ideas for a computing architecture for cognitive modeling—we can think of it as a possible infrastructure for cognitive modeling. The section “[The von Neumann architecture](#)” establishes the conventional von Neumann computer architecture as a baseline for the discussion that follows. The section “[An engineering view of computing](#)” highlights the central role of representation in computing. The section “[Properties of neural representation](#)” looks at the mathematical properties of neural representation suggested by the brain’s circuits, and in the section “[Hyperdimensional computer](#)” we envisage a computer for dealing with the hypothesized hyperdimensional representation and discuss its arithmetic, which is that of vectors, matrices, permutations, and probability. The section “[Constructing a cognitive code](#)” is concerned with representing composite entities, such as sets, sequences, and mappings, in terms of their components—that is, using the arithmetic operations to make new representations from existing ones. The section “[Three examples with cognitive connotations](#)” suggests ways to use the machinery of the previous two sections in modeling cognition. The first example (context vectors as examples of sets; random indexing) constructs meaningful vectors from random ones by assimilating massive data. The second example (learning to infer by holistic mapping; learning from example) demonstrates the learning of a rule from examples—the rule takes the form of a high-dimensional vector, and its application is by vector arithmetic. The third example “What is the dollar of Mexico?” models figurative meaning and analogy with high-dimensional arithmetic and suggests the blending of formal cognitive structure and prototypes in our minds. The last two sections are an overview of the past and a possible future of this kind of computing and of cognitive modeling based on it, with references to representative literature. Rather than attempting a scholarly review, they include pointers and terms that are useful for searching the subject on the Internet.

The von Neumann Architecture

Modern computer architecture, known as the von Neumann architecture, is a mere 60 years old. It is based on the simple idea that data and the instructions for manipulating the data are entities of the same kind. Both can be processed and stored as data in a single uniform memory. The phenomenal success of this architecture has made computers an ubiquitous part of our lives.

Our limited success in programming computers for the kind of flexible intelligence that characterizes human and

animal behavior has led many of us to wonder whether a totally different computing architecture is needed. What kind? Nobody knows. Here we argue, however, that an architecture very similar to von Neumann’s could work, and that the burden falls on representation—on the kinds of entities that the computer computes with.

The basic idea is to compute with large random patterns—that is, with very high-dimensional random vectors. Such vectors have subtle mathematical properties that can be used for computing. Even the simplest of high-dimensional vectors, namely binary (i.e., long bit strings), possess these properties, and so we will demonstrate the properties with them whenever possible.

The von Neumann architecture consists of a memory (random access memory, RAM), a processor (central processing unit, CPU), and channels for accepting data (input) and presenting results (output). The CPU is further divided into a sequencing unit for running the program, instruction by instruction, and an arithmetic–logic unit (ALU) for doing basic math operations such as addition. This paper deals with the memory and the ALU for large random patterns. More specifically, it is about the “arithmetic” operations that would form the basis of computing with large random patterns. The presentation is necessarily mathematical and, although slow-paced to mathematicians, it may inspire exploration and discovery also by them.

An Engineering View of Computing

In thinking of computing as something that is carried out by physical devices, be they computers or brains, it is helpful to look at computing in engineering terms. Computing is the transformation of representations by algorithms that can be described by rules. A representation is a *pattern* in some physical medium, for example, the configuration of ONs and OFFs on a set of switches. The algorithm then tells us how to change these patterns—how to set the switches from one moment to the next based on their previous settings.

This characterization of computing is abstract and may even seem pointless. However, the very nature of computing is abstract and becomes meaningful only when the patterns correspond in a systematic way to things in the world, or to abstract entities such as names and numbers—which is to say that they represent—or when they drive actuators. Thus, from an engineering perspective, computing is the *systematized and mechanized manipulation of patterns*.

For transforming patterns, computers have circuits. The adder circuit is an example: given the patterns representing two numbers, it produces the pattern for their sum. The

algorithm for computing the sum is embodied in the design of the circuit.

The details of circuit design—its “logic”—depend crucially on how numbers are represented. Computers usually represent numbers in the binary system, that is, in strings of bits, with each bit being either a 0 or a 1. The logical design of the adder circuit then specifies how each bit of the sum is formed from the bits of the two numbers being added together. When computer engineers speak of logic, it is in this restricted sense of how patterns are transformed by circuits.

The materials that a circuit is made of are incidental—we could say, immaterial. Computer-logic design is therefore abstract and mathematical, and finding suitable materials for implementing a design is a separate field in its own right. This holds a lesson for us who want to understand neural circuits: the logical design can be separated from neural realization. We need all the insight into the brain’s circuits and representations that neuroscience can provide, but we must then abstract away from neurotransmitters, ion channels, membrane potentials, and spike trains and face the challenge as a mathematical puzzle driven by the behavior to be reproduced. Such abstraction is essential to the understanding of the underlying principles and to the building of computers based on them.

An Engineering View of Representation

Representation is crucial to traditional computing as illustrated by the following example, and apparently it is equally important to the brain’s computing. Computers use binary representation almost exclusively, which means that an individual circuit component has two possible states, usually denoted by 0 and 1. The reason for restricting to only two states has to do with physics: electronic components are most reliable when they are bistable. Richer and more meaningful representations are then gotten by using a set of such binary components. Thus, a representation is the pattern of 0s and 1s on a set of components and it can be thought of as a string of bits or as a binary vector. In terms of computing theory, the binary-based system is fully general.

Representation must satisfy at least one condition, it must *discriminate*: the bit patterns for different things must differ from one another. Beyond that, how the patterns relate to each other determines their possible use for computing. For example, the representation of numbers must be suited for arithmetic—that is, for computing in the traditional sense of the word. This is accomplished with positional representation: by treating a string of bits as a number in the binary number system. The rules for addition, subtraction, multiplication, and division of binary

numbers are relatively simple and are readily expressed as computer circuits.

The choice of representation is often a compromise. The following example illustrates a bias in favor of some operations at the expense of others. Any number (a positive integer) can be represented as the product of its prime factors. That makes multiplication easy—less work than multiplication in base-2 arithmetic—but other operations such as addition become unduly complicated. For overall efficiency in arithmetic, the base-2 system is an excellent compromise. The brain’s representations must be subject to similar tradeoffs and compromises.

The brain’s representations of number and magnitude are subject to all sorts of context effects, as seen in the kinds of errors we make, and obviously are not optimized for fast and reliable arithmetic. Rather than being a design flaw, however, the context effects very likely reflect a compromise in favor of more vital functions that brains must perform.

The brain’s representations are carried on components that by and large are nonbinary. Yet many brainlike context effects can be demonstrated with binary patterns and operations, and there is a good reason to do so in our modeling, namely, the important properties of the representation follow from high dimensionality rather than from the precise nature of the dimensions. When binary is sufficient for demonstrating these properties, we should use it because it is the simplest possible and is an excellent way to show that the system works by general principles rather than by specialized tailoring of individual components.

Since the dimensionality of representation is a major concern in this paper, we need to touch upon *dimensionality reduction*, which is a standard practice in the processing of high-dimensional data. However, it is also possible that very high dimensionality actually facilitates processing: instead of being a curse, high dimensionality can be a blessing. For example, numbers (i.e., scalars), by definition, are one-dimensional, but in a computer they are represented by strings of bits, that is, by high-dimensional vectors: a 32-bit integer is a 32-dimensional binary vector. The high-dimensional representation makes simple algorithms and circuits for high-precision arithmetic possible. We can contrast this with one-dimensional representation of numbers. The slide rule represents them one-dimensionally and makes calculating awkward and imprecise. Thus, the dimensionality of an entity (a number) and the dimensionality of its representation for computing purposes (a bit vector) are separate issues. One has to do with existence in the world, the other with the suitability for manipulation by algorithms—that is, suitability for computing. The algorithms discussed in this paper work by virtue of their high (hyper)dimensionality.

Properties of Neural Representation

Hyperdimensionality

The brain's circuits are massive in terms of numbers of neurons and synapses, suggesting that large circuits are fundamental to the brain's computing. To explore this idea, we look at computing with ultrawide words—that is, with very high-dimensional vectors. How would we compute with 10,000-bit words? How like and unlike is it from computing with 8-to-64-bit words? What is special about 10,000-bit words compared to 8-to-64-bit words?

Computing with 10,000-bit words takes us into the realm of very high-dimensional spaces and vectors; we will call them *hyperdimensional* when the dimensionality is in the thousands, and we will use *hyperspace* as shorthand for hyperdimensional space, and similarly *hypervector*. In mathematics, “hyperspace” usually means a space with more than three dimensions; in this paper it means a lot more.

The theme of this paper is that hyperspaces have subtle properties on which to base a new kind of computing. This “new” computing could in reality be the older kind that made the human mind possible, which in turn invented computers and computing that now serve as our standard!

High-dimensional modeling of neural circuits goes back several decades under the rubric of artificial neural networks, parallel distributed processing (PDP), and connectionism. The models derive their power from the properties of high-dimensional spaces and they have been successful in tasks such as classification and discrimination of patterns. However, much more can be accomplished by further exploiting the properties of hyperspaces. Here we draw attention to some of those properties.

Robustness

The neural architecture is amazingly tolerant of component failure. The robustness comes from redundant representation, in which many patterns are considered equivalent: they mean the same thing. It is very unlike the standard binary representation of, say, numbers in a computer where a single-bit difference means that the numbers are different—where every bit “counts.”

Error-correcting codes of data communications are robust in the sense that they tolerate some number of errors. A remarkable property of hyperdimensional representation is that the number of places at which equivalent patterns may differ can become quite large: the *proportion* of allowable “errors” increases with dimensionality.

Replication is a simple way to achieve redundancy. Each of the bits in a nonredundant representation, such as a binary number, can be replaced by three bits, all with the

same value, and letting the majority rule when the three disagree. However, there are much better ways to achieve redundancy and robustness.

Independence from Position: Holistic Representation

Electrical recording from neurons shows that even seemingly simple mental events involve the simultaneous activity of widely dispersed neurons. Finding out directly how the activity is organized is extremely difficult but we can try to picture it by appealing to general principles. For maximum robustness—that is, for the most efficient use of redundancy—the information encoded into a representation should be distributed “equally” over all the components, that is, over the entire 10,000-bit vector. When bits fail, the information degrades in relation to the number of failing bits irrespective of their position. This kind of representation is referred to as *holographic* or *holistic*. It is very different from the encoding of data in computers and databases where the bits are grouped into fields for different pieces of information, or from binary numbers where the position of a bit determines its arithmetic value.

Of course, some information in the nervous system is tied to physical location and hence to position within the representation. The closer we are to the periphery—to the sense organs and to muscles and glands—the more clearly the position of an individual component—a neuron—corresponds to a specific part of a sense organ, muscle, or gland. Thus, the position-independence applies to representations at higher, more abstract levels of cognition where information from different senses has been integrated and where some of the more general computing mechanisms come into play.

Randomness

We know from neuroanatomy that brains are highly structured but many details are determined by learning or are left to chance. In other words, the wiring does not follow a minute plan, and so no two brains are identical. They are incompatible at the level of hardware and internal patterns—a mind cannot be “downloaded” from one brain to another.

To deal with the incompatibility of “hardware” and the seeming arbitrariness of the neural code, our models use randomness. The system builds its model of the world from random patterns—that is, by starting with vectors drawn randomly from the hyperspace.

The rationale for this is as follows. If random origins can lead to compatible systems, the incompatibility of hardware ceases to be an issue. The compatibility of systems—and the equivalence of brains—is sought not in the actual patterns of the internal code but in the relation of the patterns to one

another within each system. Language is a prime example of a system like that at a higher level: we can say the same thing in different languages in spite of their different grammars and vocabularies. Likewise at the level of the internal code, the patterns for girl and boy, for example, should be more similar than the patterns for girl and locomotive in the same system, whereas the patterns for girl in different systems need not bear any similarity to each other. Examples of such model building will be given below.

Randomness has been a part of artificial neural systems from the start. Self-organizing feature maps and the Boltzmann machine are good examples. We can think of randomness as the path of least assumptions. A system that works in spite of randomness is easy to design and does not necessarily require randomness. The randomness assumption is also used as a means to simplify the analysis of a system's performance.

Hyperdimensional Computer

Notation Mathematics will be displayed as follows: lowercase for scalars, variables, relations, and functions (a , x , f), Latin uppercase for vectors (A , X), and Greek uppercase for (permutation) matrices (Π , Γ). Letters are chosen to be mnemonic when possible (A for address, G for grandmother). The order of operations when not shown by parentheses is the following: multiplication by matrix first (ΠA), then multiplication by vector (XOR , $*$), and finally addition ($+$).

Hyperdimensional Representation

The units with which a computer computes make up its *space of representations*. In ordinary computers, the space is that of relatively low-dimensional binary vectors. The memory is commonly addressed in units of eight-bit bytes, and the arithmetic operations are commonly done in units of 32-bit words. A computer with a 32-bit ALU and up to 4 GB of memory can be thought of as having 32-bit binary vectors as its representational space, denoted mathematically by $\{0, 1\}^{32}$. These are the building blocks from which further representations are made.

Hyperdimensional representational spaces can be of many kinds: the vector components can be binary, ternary, real, or complex. They can be further specified as to sparseness, range of values, and probability distribution. For example, the space of n -dimensional vectors with i.i.d. components drawn from the normal distribution with mean 0 and variance $1/n$ was originally used. A cognitive system can include several representational spaces. One kind may be appropriate for modeling a sensory system and another for modeling language.

Important properties of hyperdimensional representation are demonstrated beautifully with 10,000-bit patterns, that is, with 10,000-dimensional binary vectors. The representational space then consists of all 2^{10000} such patterns—also called *points* of the space. That is truly an enormous number of possible patterns; any conceivable system would ever need but an infinitesimal fraction of them as representations of meaningful entities.

Our experience with three-dimensional space does not prepare us to intuit the shape of this hyperspace and so we must tease it out with analysis, example, and analogy. Like the corner points of an ordinary cube, the space looks identical from any of its points. That is to say, if we start with any point and measure the distances to all the other points, we always get the same distribution of distances. In fact, the space is nothing other than the corners of a 10,000-dimensional unit (hyper)cube.

We can measure distances between points in Euclidean or Hamming metric. For binary spaces the Hamming distance is the simplest: it is the number of places at which two binary vectors differ, and it is also the length of the shortest path from one corner point to the other along the edges of the hypercube. In fact, there are 2^k such shortest paths between two points that are k bits apart. Naturally, the maximum Hamming distance is 10,000 bits, from any point to its opposite point. The distance is often expressed relative to the number of dimensions, so that here 10,000 bits equals 1.

Although the points are not concentrated or clustered anywhere in the space—because every point is just like every other point—the *distances* are highly concentrated half-way into the space, or around the distance of 5,000 bits, or 0.5. It is easy to see that half the space is closer to a point than 0.5 and the other half is further away, but it is somewhat surprising that less than a millionth of the space is closer than 0.476 and less than a thousand-millionth is closer than 0.47; similarly, less than a millionth is further than 0.524 away and less than a thousand-millionth is further than 0.53. These figures are based on the binomial distribution with mean 5,000 and standard deviation (STD) 50, and on its approximation with the normal distribution—the distance from any point of the space to a randomly drawn point follows the binomial distribution. These distance ranges give the impression that a 600-bit wide “bulge” around the mean distance of 5,000 bits contains nearly all of the space! In other words, if we take two vectors at random and use them to represent meaningful entities, they differ in approximately 5,000 bits, and if we then take a third vector at random, it differs from each of the first two in approximately 5,000 bits. We can go on taking vectors at random without needing to worry about running out of vectors—we run out of time before we run out of vectors. We say that such vectors are *unrelated*.

Measured in standard deviations, the bulk of the space, and the unrelated vectors, are 100 STDs away from any given vector.

This peculiar distribution of the space makes hyperdimensional representation robust. When meaningful entities are represented by 10,000-bit vectors, many of the bits can be changed—more than a third—by natural variation in stimulus and by random errors and noise, and the resulting vector can still be identified with the correct one, in that it is closer to the original “error-free” vector than to any unrelated vector chosen so far, with near certainty.

The robustness is illustrated further by the following example. Let us assume that two meaningful vectors A and B are only 2,500 bits apart—when only 1/4 of their bits differ. The probability of this happening by chance is about zero, but a system can create such vectors when their meanings are related; more on such relations will be said later. So let us assume that 1/3 of the bits of A are changed at random; will the resulting “noisy” A vector be closer to B than to A —would it be falsely identified with B ? It is possible but most unlikely because the noisy vector would be 4,166 bits away from B , on the average, and only 3,333 bits from A ; the difference is 17 STDs. The (relative) distance from the noisy A vector to B is given by $d + e - 2de$ with $d = 1/4$ and $e = 1/3$. Thus, adding e amount of noise to the first vector *increases* the distance to the second vector by $(1 - 2d)e$ on the average. Intuitively, most directions that are away from A in hyperspace are also away from B .

The *similarity* of patterns is the flip-side of distance. We say that two patterns, vectors, points are similar to each other when the distance between them is considerably smaller than 0.5. We can now describe points of the space and their neighborhoods as follows. Each point has a large “private” neighborhood in terms of distance: the volume of space within, say, 1/3 or 3,333 bits is insignificant compared to the total space. The rest of the space—all the unrelated “stuff”—becomes significant only when the distance approaches 0.5. In a certain probabilistic sense, then, two points even as far as 0.45 apart are very close to each other. Furthermore, the “private” neighborhoods of any two unrelated points have points in common—there are patterns that are closely related to any two unrelated patterns. For example, a point C half-way between unrelated points A and B is very closely related to both, and another half-way point D can be unrelated to the first, C . This can be shown with as few as four dimensions: $A = 0000$, $B = 0011$, $C = 0001$, and $D = 0010$. However, the “unusual” probabilities implied by these relative distances require high dimensionality. This is significant when representing objects and concepts with points of the hyperspace, and significantly different from what we are accustomed to in ordinary three-dimensional space.

In addition to being related by similarity, patterns can relate to each another by transformation—that is, by how one is transformed into another or how several patterns are combined to form a new pattern, in a kind of pattern arithmetic. This is analogous to what ordinary computers do: new patterns are created from existing ones by arithmetic operations that are built into the computer’s circuits. This way of interpreting the neural code is mostly unexplored. We have much to say about it below.

Hyperdimensional Memory

Memory is a vital part of an ordinary computer, and we would expect that something like it would also be a part of any computer for emulating cognition. An ordinary computer memory is an array of addressable registers, also called memory locations. Each location holds a string of bits of a fixed length; the length is called the word size. The contents of a location are made available for processing by probing the memory with the location’s address, which likewise is a string of bits. An n -bit address can access a memory with 2^n locations, with memories of 2^{30} or a thousand million eight-bit wide locations becoming more and more common.

It is possible to build a memory for storing 10,000-bit vectors that is also addressed with 10,000-bit vectors, although 2^{10000} locations is far too many ever to be built or needed. In artificial neural-net research they are called associative memories. An associative memory can work somewhat like an ordinary computer memory in that when the pattern X is stored using the pattern A as the address, X can later be retrieved by addressing the memory with A . Furthermore, X can be retrieved by addressing the memory with a pattern A' that is similar to A .

This mode of storage is called *heteroassociative*, to be contrasted with *autoassociative*. Both are based on the same mechanism, the difference being that autoassociative storage is achieved by storing each pattern X using X itself as the address. This may appear silly but in fact is useful because it allows the original stored X to be recovered from an approximate or noisy version of it, X' , thus making the memory robust. Such recovery typically takes several iterations (fewer than ten) where the address X' is used to retrieve X'' is used to retrieve X''' ... as the process converges to X . However, if the amount of noise is too great—if X' is too far from X —the original X will not be recovered. The pattern X is called a point attractor, the region of space surrounding it is called the basin of attraction, and the memory is referred to as *content addressable*.

The same kind of iteration to a noise-free X is not possible in heteroassociative storage. If the memory is probed with a noisy address A' , the retrieved pattern X' will usually have some noise relative to X . If the memory is

then addressed with X' , there is no guarantee that anything useful will be retrieved. We therefore envisage a cognitive computing architecture that relies primarily on autoassociative memory. It will serve as an item memory or clean-up memory, which is discussed below.

Hyperdimensional Arithmetic

The ALU is an essential part of a computer. It has the circuits for the computer's built-in operations—its inherent capabilities. For example, it has the adder circuit that produces the sum—a binary string for the sum—of two numbers given to it as arguments. The ALU is a transformer of bit patterns.

The idea that also brains compute with a set of built-in operations is sound, although trying to locate the equivalent of an ALU seems foolish, and so we will merely look for operations on hyperdimensional patterns that could be used for computing. We will view the patterns as vectors because we can then tap into the vast body of knowledge about vectors, matrices, linear algebra, and beyond. This indeed has been the tradition in artificial neural-net research, yet rich areas of high-dimensional representation remain to be explored. By being thoroughly mathematical, such exploration may seem peripheral to neuroscience, but the shared goal of understanding the brain's computing can actually make it quite central. Time will tell.

We will start with some operations on real vectors (vectors with real-number components), which are commonly used in artificial neural-net research.

Weighting with a constant is a very basic operation that is often combined with other, more complex operations, such as addition. The math is most simple: each component of the vector is multiplied with the same number, and the result is a *vector*.

The *comparison* of two vectors (e.g., with the cosine) is another basic operation, and the resulting measure of similarity, a *number*, is often used as a weighting factor in further computations.

A set of vectors can be combined by componentwise *addition*, resulting in a *vector* of the same dimensionality. To conform to the distributional assumptions about the representation, the arithmetic-sum-vector is *normalized*, yielding a *mean* vector. It is this mean-vector that is usually meant when we speak of the sum of a set of vectors. The simplest kind of normalization is achieved with weighting. Other kinds are achieved with other transformations of the vector components, for example by applying a threshold to get a binary vector.

The sum (and the mean) of random vectors has the following important property: it is *similar* to each of the vectors being added together. The similarity is very pronounced when only a few vectors are added and it plays a

major role in artificial neural-net models. The sum-vector is a possible representation for the set that makes up the sum.

Subtracting one vector from another is accomplished by adding the vector's complement. The complement of a real vector is gotten by multiplying each component with -1 , and of a binary vector by flipping its bit (turning 0s into 1s and 1s into 0s).

Multiplication comes in several forms, the simplest being weighting, when a vector is multiplied with a number as described above. Two vectors can be multiplied to form a *number*, called the inner product, that can be used as a measure of similarity between the vectors. The cosine of two vectors is a special case of their inner product. Another way of multiplying two vectors yields a *matrix* called the outer product. It is used extensively for adjusting the weights of a network and thus plays an important role in many learning algorithms. Multiplication of a vector with a matrix, resulting in a *vector*, is yet another kind, ubiquitous in artificial neural nets. Usually the result from a matrix multiplication needs to be normalized; normalizing was mentioned above. *Permutation* is the shuffling of the vector components and it can be represented mathematically by multiplication with a special kind of matrix, called the permutation matrix, that is filled with 0s except for exactly one 1 in every row and every column.

The above-mentioned examples of multiplication differ from addition in one important respect, they are heterogeneous: besides vectors they involve numbers and matrices. In contrast, addition is homogeneous, as all participants are vectors of the same kind: we start with vectors and end up with a vector of the same dimensionality.

A much more powerful representational system becomes possible when the operations also include multiplication that is homogeneous—in mathematical terms when the system is *closed* under both addition and multiplication. Further desiderata include that the

- multiplication is *invertible*, i.e., no information is lost,
- multiplication *distributes* over addition,
- multiplication *preserves distance* and, as a rule,
- product is *dissimilar* to the vectors being multiplied.

The product's being dissimilar is in contrast with the sum that is similar to the vectors that are added together. These desired properties of multiplication make it possible to encode compositional structure into a hypervector and to analyze the contents of composed hypervectors, as will be seen below. We now merely state that multiplication operations of that kind exist for binary, real, and complex vectors, and will discuss them later.

The above-mentioned examples of vector arithmetic suggest that computing in hyperdimensional representation—with large random patterns—can be much like

conventional computing with numbers. We will next look at how the various operations can be used to build a system of internal representations—what can be called a *cognitive code*. One example has already been mentioned, namely, that a sum-vector can represent a set. The cognitive equivalence of brains should then be sought in part in how representations are computed from one another rather than what the specific activity patterns, the exact vectors, are. Thus we can think of hyperdimensional random vectors as the medium that makes certain kinds of computing possible.

Constructing a Cognitive Code

Conventional computing uses a uniform system for representation that allows different kinds of entities to be represented in the same way. This is accomplished with pointers, which are addresses into memory; they are also numbers that can take part in arithmetic calculations. Pointers are the basis of symbolic computing.

Corresponding to traditional pointers we have hypervectors, corresponding to traditional memory we have content-addressable memory for hypervectors, and corresponding to the ALU operations we have hyperdimensional arithmetic. How might we use them for building a representational system for entities of various kinds?

Item Memory

When a pattern—a hypervector—is chosen to represent an entity it becomes meaningful. It is recorded in an *item memory* for later reference, and so the item memory becomes a catalog of meaningful patterns. The item memory is an autoassociative memory that recognizes patterns even when they contain some noise. When probed with a noisy pattern, the memory puts out the noise-free stored pattern; its job is nearest-neighbor search among the set of stored (meaningful) patterns. The item memory is therefore also called a *clean-up memory*. Many “arithmetic” operations on patterns produce approximate or noisy results that require cleaning up in order to recover the original pattern. We will see examples of this below.

Some operations produce meaningful patterns that are very similar to each other and can thereby interfere with each other’s retrieval from the item memory. For example, the sum pattern is similar to the patterns that make up the sum. In such cases it is possible to transform a pattern—map it into a different part of the space—before storing it, so long as the mapping can later be reversed when needed. We will see examples of this as well.

Representing Basic Entities with Random Vectors

Classical formal systems start with a set of primitives, that is, with “individuals” or “atoms” and predicates, and build up a universe of discourse by using functions, relations, first-order logic, quantification, and other such means. We will borrow from this tradition and assume a world with basic atomic entities. This assumption, however, is for convenience—it is to get our representation story underway—rather than a commitment to a world with basic atomic entities for cognitive systems to discover and deal with.

The smallest meaningful unit of the cognitive code is a large pattern, a hypervector, a point in hyperspace. The atomic entities or individuals are then represented by random points of the space. In fact, when we need to represent anything new that is not composed of things already represented in the system, we simply draw a vector at random from the space. When a vector is chosen to represent an entity in the system, it is stored in the item memory for later reference.

Because of hyperdimensionality, the new random vector will be unrelated to all the vectors that already have meaning; its distance from all of them is very close to 5,000 bits. In mathematical terms, it is approximately orthogonal to the vectors that are already in use. A 10,000-dimensional space has 10,000 orthogonal vectors but it has a huge number of nearly orthogonal vectors. The ease of making nearly orthogonal vectors is a major reason for using hyperdimensional representation.

Representing Sets with Sums

The mathematical notion of a *set* implies an unordered collection of elements. We want to represent both the set and its elements with hypervectors. The operation for combining the elements needs therefore to be commutative so that the order does not matter. The simplest such operation is vector addition, and the sum-vector (or the mean-vector) has the property of being similar to the vectors added together. Thus, the elements are “visible” in the representation of the set, and sets that share elements give rise to similar sums.

If we want the vector for the set to look different from the vectors for the set’s elements, we must map it into a different part of space before storing it in the item memory. The mapping should be invertible so that the original sum-vector can be recovered exactly, and it should preserve distance so that the memory can be probed with partial and noisy sums. Mapping with multiplication has these properties and is discussed below.

Elements are recovered from a stored sum-vector by first restoring the sum (with the inverse mapping) and then probing the item memory with it for the best match. The

element that is found will then be subtracted off the sum-vector and the difference-vector is used to probe the item memory, to recover another of the set's elements. The process is repeated to recover more and more of the sets's elements. However, only small sets can be analyzed into their elements in this way, and slightly larger sets can, by accumulating a (partial) sum from the vectors recovered so far, and by subtracting it from the original (total) sum before probing for the next element. However, if the unmapped sum has been stored in the item memory, this method fails because probing the (autoassociative) memory with the sum will always retrieve the sum rather than any of its elements.

It is also possible to find previously stored sets (i.e., sums) that contain a specific element by probing the memory with that element (with its vector). Before probing, the element must be mapped into the same part of space—with the same mapping—as sums are before they are stored. As mentioned above, after one vector has been recovered, it can be subtracted off the probe and the memory can be reprobed for another set that would contain that particular element.

Besides being unordered, the strict notion of a set implies that no element is duplicated, and thus a set is an enumeration of the kinds of elements that went into it. A slightly more general notion is *multiset*, also called a *bag*. It, too, is unordered, but any specific kind of element can occur multiple times. We might then say that a set is a collection of types whereas a multiset is a collection of tokens.

A multiset can be represented in the same way as a set, by the sum of the multiset's elements, and elements can be extracted from the sum also in the same way. In this case, the frequent elements would be the first ones to be recovered, but reconstructing the entire multiset from this representation would be difficult because there is no reliable way to recover the frequencies of occurrence. For example, the normalized sum is not affected by doubling the counts of all the elements in the multiset.

Two Kinds of Multiplication, Two Ways to Map

Existing patterns can give rise to new patterns by mappings of various kinds, also called functions. One example of a function has already been discussed at length: the (componentwise) addition of two or more vectors that produces a sum-vector or a mean-vector. The following discussion about multiplication is in terms of binary vectors, although the ideas apply much more generally.

Multiplication by Vector

A very basic and simple multiplication of binary vectors is by componentwise Exclusive-Or (XOR). The XOR of two

vectors has 0s where the two agree and it has 1s where they disagree. For example, $0011\dots10 \text{ XOR } 0101\dots00 = 0110\dots10$. Mathematically, the XOR is the arithmetic sum modulo 2. The $(1, -1)$ -binary system, also called *bipolar*, is equivalent to the $(0, 1)$ -binary system when the XOR is replaced by ordinary multiplication. We will use the notation $A * B$ for the multiplication of the vectors A and B —for their product-vector. Here $*$ is the XOR unless otherwise noted.

The XOR commutes, $A * B = B * A$, and is its own inverse so that $A * A = O$, where O is the vector of all 0s (in algebra terms O is the unit vector because $A * O = A$). Since the XOR-vector has 1s where the two vectors disagree, the number of 1s in it is the Hamming distance between the two vectors. By denoting the number of 1s in a binary vector X with $|X|$ we can write the Hamming distance d between A and B as $d(A, B) = |A * B|$.

Multiplication can be thought of as a *mapping* of points in the space. Multiplying the vector X with A maps it to the vector $X_A = A * X$ which is as far from X as there are 1s in A (i.e., $d(X_A, X) = |X_A * X| = |(A * X) * X| = |A * X * X| = |A|$). If A is a typical (random) vector of the space, about half of its bits are 1s, and so X_A is in the part of the space that is *unrelated* to X in terms of the distance criterion. Thus we can say that multiplication *randomizes*.

Mapping with multiplication *preserves distance*. This is seen readily by considering $X_A = A * X$ and $Y_A = A * Y$, taking their XOR, and noting that the two A s cancel out thus:

$$X_A * Y_A = (A * X) * (A * Y) = A * X * A * Y = X * Y$$

Since the XOR-vector is the same, the Hamming distance is the same: $|X_A * Y_A| = |X * Y|$. Consequently, when a set of points is mapped by multiplying with the same vector, the distances are maintained—it is like moving a constellation of points bodily into a different (and indifferent) part of the space while maintaining the relations (distances) between them. Such mappings could play a role in high-level cognitive functions such as analogy and the grammatical use of language where the relations between objects are more important than the objects themselves.

In the above-mentioned example, we think of the vector A as a mapping applied to vectors X and Y . The same math applies if we take two mappings A and B and look at their effect on the same vector X : X will be mapped onto two vectors that are exactly as far from each other as mapping A is from mapping B . Thus, when vectors represent mappings, we can say that the mappings are similar when the vectors are similar; similar mappings map any vector to two similar vectors. Notice that any of the 2^{10000} vectors of the representational space is potentially a mapping, so that what was said above about the similarity of vectors in the space holds equally to similarity of mappings.

Because multiplication preserves distance it also preserves noise: if a vector contains a certain amount of noise, the result of mapping it contains exactly the same noise. If each of the multiplied vectors contains independent random noise, the amount of noise in the product—its distance to the noise-free product-vector—is given by $e = f + g - 2fg$, where f and g are the relative amounts of noise in the two vectors being multiplied.

A very useful property of multiplication is that it distributes over addition. That means, for example, that

$$A * [X + Y + Z] = [A * X + A * Y + A * Z]$$

The brackets [...] stand for normalization. Distributivity is invaluable in analyzing these representations and in understanding how they work and fail.

Distributivity for binary vectors is most easily shown when they are bipolar. The vector components then are 1s and -1s, the vectors are added together into an ordinary arithmetic-sum-vector, and the (normalized) bipolar-sum-vector is gotten by considering the sign of each component (the signum function). The XOR becomes now ordinary multiplication (with 1s and -1s), and since it distributes over ordinary addition, it does so also in this bipolar case. If the number of vectors added together is even, we end up with a ternary system unless we break the ties, for example, by adding a random vector.

Permutation as Multiplication

Permutations reorder the vector components and thus are very simple; they are also very useful in constructing a cognitive code. We will denote the permutation of a vector with a multiplication by a matrix (the permutation matrix Π), thus $X_{\Pi} = \Pi X$. We can also describe the permutation of n elements as the list of the integers 1, 2, 3, ..., n in the permuted order. A random permutation is then one where the order of the list is random—it is a permutation chosen randomly from the $n!$ possible permutations.

As a mapping operation, permutation resembles vector multiplication: (1) it is invertible, (2) it distributes over addition—in fact, it distributes over any componentwise operation including multiplication with the XOR—and as a rule (3) the result is dissimilar to the vector being permuted. Because permutation merely reorders the coordinates, (4) the distances between points are maintained just as they are in multiplication with a vector, thus $\Pi X * \Pi Y = \Pi(X * Y)$ and $d(\Pi X, \Pi Y) = |\Pi X * \Pi Y| = |\Pi(X * Y)| = |X * Y| = d(X, Y)$.

Although permutations are not elements of the space of representations (they are not n -dimensional hypervectors), they have their own rules of composition—permutations are a rich mathematical topic in themselves—and they can be assessed for similarity by how they map vectors. As

mentioned above, we can map the same vector with two different permutations and ask how similar the resulting vectors are: by permuting X with Π and Γ , what is the distance between ΠX and ΓX , what can we say of the vector $Z = \Pi X * \Gamma X$? Unlike above with multiplication by a vector, this depends on the vector X (e.g., the 0-vector is unaffected by permutation), so we will consider the effect on a typical X of random 0s and 1s, half of each. Wherever the two permutations (represented as lists of integers) agree, they move a component of X to the same place making that bit of Z a 0; let us denote the number of such places with a . In the $n - a$ remaining places where the two permutations disagree, the bits of ΠX and ΓX come from different places in X and thus their XOR is a 1 with probability 1/2. We then have that the probability of 1s in Z equals $(n - a)/2$. If the permutations Π and Γ are chosen at random, they agree in only one position ($a = 1$) on the average, and so the distance between ΠX and ΓX is approximately 0.5; random permutations map a given point to (in)different parts of the space. In fact, pairs of permutations (of 10,000 elements) that agree in an appreciable number of places are extremely rare among all possible pairs of permutations. Thus we can say that, by being dissimilar from one another, random permutations randomize, just as does multiplying with random vectors as seen above.

Representing Sequences with Pointer Chains

Sequences are all-important for representing things that occur in time. We can even think of the life of a system as one long sequence—the system's individual history—where many subsequences repeat approximately. For a cognitive system to learn from experience it must be able to store and recall sequences.

One possible representation of sequences is with pointer chains or linked lists in an associative memory. The sequence of patterns $ABCDE\dots$ is stored by storing the pattern B using A as the address, by storing C using B as the address, by storing D using C as the address, and so forth; this is a special case of heteroassociative storage. Probing the memory with A will then recall B , probing it with B will recall C , and so forth. Furthermore, the recall can start from anywhere in the sequence, proceeding from there on, and the sequence can be retrieved even if the initial probe is noisy, as subsequent retrievals will converge to the noise-free stored sequence in a manner resembling convergence to a fixed point in an autoassociative memory.

Although straightforward and simple, this way of representing sequences has its problems. If two sequences contain the same pattern, progressing past it is left to chance. For example, if $ABCDE\dots$ and $XYCDZ\dots$ have been stored in memory and we start the recall with A , we

would recall BCD reliably but could thereafter branch off to Z because D would point somewhere between E and Z . Clearly, more of the history is needed for deciding where to go from D . Longer histories can be included by storing links that skip over elements of the sequence (e.g., by storing E using B as the address) and by delaying their retrieval according to the number of elements skipped. The element evoked by the more distant past would then bias the retrieval toward the original sequence.

Representing Sequences by Permuting Sums

As with sets, several elements of a sequence can be represented in a single hypervector. This is called *flattening* or *leveling* the sequence. However, sequences cannot be flattened with the sum alone because the order of the elements would be lost. Before computing the vector sum, the elements must be “labeled” according to their position in the sequence so that X one time step ago appears different from the present X , and that the vectors for AAB and ABA will be different. Such labeling can be done with permutations.

Let us first look at one step of a sequence, for example, that D is followed by E . This corresponds to one step of heteroassociative storage, which was discussed above. The order of the elements can be captured by permuting one of them before computing their sum. We will permute the first and represent the pair with the sum

$$S = \Pi D + E$$

and we will store S in the item memory. The entire sequence can then be stored by storing each of its elements and each two-element sum such as S above in the item memory. If we later encounter D we can predict the next element by probing the memory not with D itself but with a permuted version of it, ΠD . It will retrieve S by being similar to it. We can then retrieve E by subtracting ΠD from S and by probing the memory with the resulting vector.

Here we have encoded the sequence step DE so that the previous element, D , can be used to retrieve the next, E . However, we can also encode the sequence so that the two previous elements C and D are used for retrieving E . In storing the sequence we merely substitute the encoding of CD for D , that is to say, we replace D with $\Pi C + D$. After the substitution, the S of the preceding paragraph becomes $S = \Pi(\Pi C + D) + E = \Pi\Pi C + \Pi D + E$, which is stored in memory. When CD is subsequently encountered, it allows us to make the probe $\Pi\Pi C + \Pi D$ which will retrieve S as above, which in turn is used to retrieve E as above.

We can go on like this, including more and more elements of the sequence in each stored pattern and thereby including more and more of the history in them and in their retrieval. Thus, with one more element included in the

history, the vector that is stored in the item memory encodes the sequence $BCDE$ with $S = \Pi\Pi\Pi B + \Pi\Pi C + \Pi D + E$, and later when encountered with BCD we would start the retrieval of E by probing the item memory with $\Pi\Pi\Pi B + \Pi\Pi C + \Pi D$. By now the stored vectors contain enough information to discriminate between $ABCDE$ and $XYCDZ$ so that E will be retrieved rather than Z .

Even if it is possible to encode ever longer histories into a single vector, the prediction of the next element does not necessarily keep on improving. For example, if the sequence is k th order Markov, encoding more than $k + 1$ elements into a single vector weakens the prediction. Furthermore, the capacity of a single binary vector sets a limit on the length of history that it can represent reliably. How best to encode the history for the purposes of prediction depends of course on the statistical nature of the sequence.

A simple recurrent network can be used to produce flattened histories of this kind if the history at one moment is permuted and then fed back and added to the vector for the next moment. By normalizing the vector after each addition we actually get a flattened history that most strongly reflects the most recent past and is unaffected by the distant past. If we indicate normalization with brackets, the sequence $ABCDE$ will give rise to the sum

$$\begin{aligned} S &= \Pi[\Pi[\Pi[\Pi[A + B] + C] + D] + E \\ &= [[[\Pi\Pi\Pi\Pi A + \Pi\Pi\Pi B] + \Pi\Pi C] + \Pi D] + E \end{aligned}$$

The last element E has equal weight to the history up to it, irrespective of the length of the history—the distant past simply fades away. Some kind of weighting may be needed to keep it from fading too fast, the proper rate depending on the nature of the sequence. As mentioned before, the various permutations keep track of how far back in the sequence each specific element occurs without affecting the relative contribution of that element.

Several remarks about permutations are in order. An iterated permutation, such as $\Pi\Pi\Pi$ above, is just another permutation, and if Π is chosen randomly, iterated versions of it appear random to each other with high probability. However, all permutations are made of loops in which bits return to their original places after some number of iterations (every bit returns at least once in n iterations), and so some care is needed to guarantee permutations with good loops.

Pseudorandom-number generators are one-dimensional analogs. The simpler ones get the next number by multiplying the previous number with a constant and truncating the product to fit the computer’s word—they lop off the most significant bits. Such generators necessarily run in loops, however long. Incidentally, the random permutations of our computer simulations are made with random-number generators.

A feedback circuit for a permutation is particularly simple: one wire goes out of each component of the vector and one wire comes back in, the pairing is random, and the outgoing signal is fed back after one time-step delay. The inverse permutation has the same connections taken in the opposite direction.

Representing Pairs with Vector Multiplication

A pair is a basic unit of association, when two elements A and B correspond to each other. Pairs can be represented with multiplication: in $C = A * B$ the vector C represents the pair. If we know the product C and one of its elements, say A , we can find the other by multiplying C with the inverse of A .

The XOR as the multiplication operation can “overperform” because it both commutes ($A \text{ XOR } B = B \text{ XOR } A$) and is its own inverse ($A \text{ XOR } A = O$). For example, any pair of two identical vectors will be represented by the 0-vector. This can be avoided with a slightly different multiplication that neither commutes nor is a self-inverse. As with sequences, we can encode the order of the operands by permuting one of them before combining them. By permuting the first we get

$$C = A * B = \Pi A \text{ XOR } B$$

This kind of multiplication has all the desired properties: (1) it is invertible although the right and the left-inverse operations are different, (2) it distributes over addition, (3) it preserves distance, and (4) the product is dissimilar to both A and B . We can extract the first element from C by canceling out the second and permuting back (right-inverse of $*$),

$$\Pi^{-1}(C \text{ XOR } B) = \Pi^{-1}((\Pi A \text{ XOR } B) \text{ XOR } B) = \Pi^{-1} \Pi A = A$$

where Π^{-1} is the inverse permutation of Π , and we can extract the second element by canceling out the permuted version of the first (left-inverse of $*$),

$$\Pi A \text{ XOR } C = \Pi A \text{ XOR } (\Pi A \text{ XOR } B) = B$$

Because of the permutation, however, this multiplication is not associative: $(A * B) * C \neq A * (B * C)$. For simplicity in the examples that follow, the multiplication operator $*$ will be the XOR.

Representing Bindings with Pairs

In traditional computing, memory locations—their addresses—represent variables and their contents represent values. The values are set by assignment, and we say that it *binds* a variable to a value. A number stored in one location

can mean age and the same number—the same bit pattern—stored in another location can mean distance. Thus the meaningful entity is the address–value pair. The value can be many other things besides a number. In particular, it can be a memory address. Data structures are built from such pairs and they are the basis of symbolic representation and processing.

In holistic representation, the variable, the value, and the bound pair are all hypervectors of the same dimensionality. If X is the vector for the variable x and A is the vector for the value a , then the bound pair $x = a$ can be represented by the product-vector $X * A$. It is dissimilar to both X and A but either one can be recovered from it given the other. *Unbinding* means that we take the vector for the bound pair and find one of its elements, say A , by multiplying with the other, as seen above. In cognitive modeling, variables are often called roles and values are called fillers.

Representing Data Records with Sets of Bound Pairs

Complex objects of ordinary computing are represented by data records composed of fields, and by pointers to such data records. Each field in the record represents a variable (a role). The roles are implicit—they are implied by the location of the field in the record. Holistic representation makes the roles explicit by representing them with vectors. Vectors for unrelated roles, such as name and age, can be chosen at random. The role x with the filler a , i.e., $x = a$, will then be represented by $X * A$ as shown above.

A data record combines several role–filler pairs into a single entity. For example, a record for a person might include name, sex, and the year of birth, and the record for Mary could contain the values ‘Mary Myrtle’, female, and 1966. Its vector representation combines vectors for the variables and their values—name (X), sex (Y), year of birth (Z), ‘Mary Myrtle’ (A), female (B), and 1966 (C)—by binding each variable to its value and by adding the three resulting vectors into the holistic sum-vector H :

$$H = X * A + Y * B + Z * C$$

The vector H is self-contained in that it is made of the bit patterns for the variables and their values, with nothing left implicit. Being a sum, H is similar to each of the three pairs, but the pairs by being products “hide” the identity of their elements so that H is dissimilar to each of A, B, C, X, Y, Z . However, the information about them is contained in H and can be recovered by unbinding. For example, to find the value of x in H we multiply H with (the inverse of) X and probe the item memory with the result, retrieving A . The math works out as follows:

$$\begin{aligned}
X * H &= X * (X * A + Y * B + Z * C) \\
&= X * X * A + X * Y * B + X * Z * C \\
&= A + R_1 + R_2
\end{aligned}$$

where multiplication by X is distributed over the three vectors that make up the sum, and where the X s in $X * X * A$ cancel out each other. The result of the unbinding (of multiplying with X) can thus be expressed as the sum of three vectors, A , R_1 , and R_2 . Of these, A has been stored in the item memory, whereas nothing similar to R_1 or R_2 has been and so they act as random noise. For example, by being a product, R_1 is dissimilar to each of X , Y , and B . Therefore, $X * H$ retrieves A from the item memory. Notice that unbinding a pair such as $X * A$ requires no clean-up.

Holistic representation accommodates the adding of “fields” into a data record. Supplementing H with two more variables u and v with values d and e gives us a sum-vector $X * A + Y * B + Z * C + U * D + V * E$ that is very similar to H because it shares three pairs with H . In other words, we need not worry about aligning the fields of a record because each “field” spans the entire vector.

Representing Substitution with a Computed Mapping

The power of the human mind comes largely from our ability to understand things by analogy. In computing terms it means *substituting* one set of particulars for another within a framework. The frameworks of traditional computing are record structures—that is, the variables or roles—and the particulars are the values or fillers.

In holistic representation, substitution can be done with multiplication. We have seen that $X * A$ can represent the binding of the variable x to the value a . If we want to substitute d for a —to bind d to the variable that a is bound to—we can recover the variable first by unbinding with A , $(X * A) * A = X$, and then bind the result to D to get $X * D$. This can be written as $((X * A) * A) * D$ based on how we arrived at X , and it equals $(X * A) * (A * D)$. Thus the substitution of d for a is affected by multiplication with $(A * D)$ —that is, by mapping the previously bound pair with a pair that represents the substitution.

This is a simple case of computing a mapping. The product-vector $A * D$ can be used as a mapping that performs a substitution. The ability to compute such mappings is a key feature of hyperdimensional representation and it is due to the absence of implicit information. Moreover, these formulas are a way of saying that $x = a$ and $a = d$ equals $x = d$ while blurring the distinction between variables and values. Such blurring actually seems cognitively more realistic than the strict distinction between variables and values that is usually made in formal systems.

Multiple Substitutions with a Holistic Mapping

The idea of a computed mapping can be taken further. Here we consider two records with identical roles but different fillers—such as in two records of a database. One is the H above and the other fills the same roles x , y , z with d , e , f and thus is encoded by

$$K = X * D + Y * E + Z * F$$

When the fillers A , B , C , D , E , F are dissimilar to one another, the vectors H and K will be dissimilar. However, we can compute a mapping M that transforms one to the other:

$$M = H * K$$

(i.e., $H * M = K$ and $K * M = H$). The analysis of M shows that it contains the three matched pairs of fillers—namely, it contains $A * D + B * E + C * F$, we will denote it with M' —plus other terms that act as noise. Therefore M' is similar to M . From the three pairs of substitutions we can thus compute the mapping M' that maps one holistic record approximately to the other: from $H * M = H * (M' + \text{noise}) = K$ we get that $H * M' + H * \text{noise} = K$ and hence

$$H * M' = H * (A * D + B * E + C * F) = K' \approx K$$

The exact K would then be recovered by probing the item memory with K' . Again we will emphasize that such mapping is possible because all the information is encoded into the holistic record—no information is implicit, making the mapping a simple matter of hyperdimensional vector arithmetic.

So as not to paint too rosy a picture of substituting within a record, we need to point out cases where it works less well if at all. Consider the mapping between two records that agree in two roles and disagree in one, where the respective values are a , b , c and a , b , f . Only one substitution is needed, and so the mapping vector M' becomes $C * F$. When applied to H we get that

$$\begin{aligned}
H * M' &= H * (C * F) \\
&= X * A * C * F + Y * B * C * F + Z * C * C * F \\
&= \text{noise} + \text{noise} + Z * F
\end{aligned}$$

which still resembles $X * A + Y * B + Z * F$ but the two bound pairs at which the records agree have been lost.

Three Examples with Cognitive Connotations

Modeling the brain’s representations with holistic hyper-vectors has been justified on several grounds: the size of neural circuits, the brain’s tolerance for variation and noise in the input signal, robustness against component failure,

and the match between our subjective judgments of similarity of concepts and the distribution of distances in hyperspace. Here we see that the modeling is further justified by hyperdimensional arithmetic—by its producing effects that suggest cognitive functions.

We are still some way from a fully worked-out architecture for cognitive computing. The examples below are meant to serve not as a recipe but as a source of ideas for future modelers. Worth pointing out is the likeness of hyperdimensional computing to conventional computing: things are represented with vectors, and new representations are computed from existing ones with (arithmetic) operations on the vectors. This idea is central and should be taken to future models.

Context Vectors as Examples of Sets; Random Indexing

Context vectors are a statistical means for studying relations between words of a language. They are high-dimensional representations of words based on their contexts. They provide us with an excellent example of random initial vectors giving rise to compatible systems.

The idea is that words with similar or related meanings appear in the same and similar contexts and therefore should give rise to similar vectors. For example, the vectors for synonyms such as ‘happy’ and ‘glad’ should be similar, as should be the vectors for related words such as ‘sugar’ and ‘salt’, whereas the vectors for unrelated words such as ‘glad’ and ‘salt’ should be dissimilar. This indeed is achieved with all context vectors described below, including the ones that are built from random vectors.

The context vector for a word is computed from the contexts in which the word occurs in a large body of text. For any given instance of the word, its context is the surrounding text, which is usually considered in one of two ways: (1) as all the other words within a short distance from where the word occurs, referred to as a *context window*, or (2) as a lump, referred to as a *document*. A context window is usually narrow, limited to half a dozen or so nearby words. A document is usually several hundred words of text on a single topic, a news article being a good example. Each occurrence of a word in a text corpus thus adds to the word’s context so that massive amounts of text, such as available on the Internet, can provide a large amount of context information for a large number of words. When a word’s context information is represented as a vector, it is called that word’s context vector. One way to characterize the two kinds of context vectors is that one represents the *multiset of words* (a bag of words) in all the context windows for a given word, and the other kind represents the *multiset of documents* in which a given word appears.

The context information is typically collected into a large matrix of frequencies where each word in the vocabulary has its own row in the matrix. The columns refer either to words of the vocabulary (one column per word) or to documents (one column per document). The rows are perfectly valid context vectors as such, but they are usually transformed into better context vectors, in the sense that the distances between vectors correspond more closely to similarity of meanings. The transformations include logarithms, inverses, and frequency cut-offs, as well as principal components of the (transformed) frequency matrix. Perhaps the best known method is latent semantic analysis (LSA), which uses singular-value decomposition and reduces the dimensionality of the data by discarding a large number of the least significant principal components.

Random-vector methods are singularly suited for making context vectors, and they even overcome some drawbacks of the more “exact” methods. The idea will be demonstrated when documents are used as the contexts in which words occur. The standard practice of LSA is to collect the word frequencies into a matrix that has a row for each word of the vocabulary (for each “term”) and a column for each document of the corpus. Thus for each document there is a column that shows the number of times that the different words occur in that document. The resulting matrix is very sparse because most words do not occur in most documents. For example, if the vocabulary consists of 100,000 words, then a 500-word document—a page of text—will have a column with at most 500 non-0s (out of 100,000). A fairly large corpus could have 200,000 documents. The resulting matrix of frequencies would then have 100,000 rows and 200,000 columns, and the “raw” context vectors for words would be 200,000-dimensional. LSA reconstructs the frequency matrix from several hundred of the most significant principal components arrived at by singular-value decomposition of the 100,000-by-200,000 matrix. One drawback is the computational cost of extracting principal components of a matrix of that size. Another, more serious, is encountered when data are added, when the documents grow into the millions. Updating the context vectors—computing the singular-value decomposition of an ever larger matrix—becomes impractical.

Random-vector methods can prevent the growth of the matrix as documents are added. In a method called Random Indexing, instead of collecting the data into a 100,000-by-200,000 matrix, we collect it into a 100,000-by-10,000 matrix. Each word in the vocabulary still has its own row in the matrix, but each document no longer has its own column. Instead, each document is assigned a small number of columns at random, say, 20 columns out of 10,000, and we say that the document *activates* those columns. A 10,000-dimensional vector mostly of 0s except for the twenty 1s

where the activated columns are located is called that document's random index vector.

When the frequencies are collected into a matrix in standard LSA, each word in a document adds a 1 in the column for that document, whereas in random indexing each word adds a 1 in all 20 columns that the document activates. Another way of saying it is that each time a word occurs in the document, the document's random index vector is added to the row corresponding to that word. So this method is very much like the standard method of accumulating the frequency matrix, and it produces a matrix whose rows are valid context vector for words, akin to the "raw" context vectors described above.

The context vectors—the rows—of this matrix can be transformed by extracting dominant principal components, as in LSA, but such further computing may not be necessary. Context vectors nearly as good as the ones from LSA have been obtained with a variant of random indexing that assigns each document a small number (e.g., 10) of "positive" columns and the same number of "negative" columns, at random. In the positive columns, 1s are added as above, whereas in the negative columns 1s are subtracted. The random index vectors for documents are now ternary with a small number of 1s and -1 s placed randomly among a large number of 0s, and the resulting context vectors have a mean of 0—their components add up to 0.

Several things about random indexing are worth noting. (1) Information about documents is distributed randomly among the columns. In LSA, information starts out localized and is distributed according to the dominant principal components. (2) Adding documents—i.e., including new data—is very simple: all we need to do is to select a new set of columns at random. This can go on into millions of documents without needing to increase the number of columns in the matrix. In LSA, columns need to be added for new documents, and singular-value decomposition needs to be updated. (3) Random indexing can be applied equally to the vocabulary so that the matrix will have fewer rows than there are words in the vocabulary, and that new words will not require adding rows into the matrix. In that case, individual rows no longer serve as context vectors for words, but the context vectors are readily computed by adding together the rows that the word activates. (4) Semantic vectors for documents can be computed by adding together the columns that the documents activate. (5) Random indexing can be used also when words in a sliding context window are used as the context. (6) And, of course, all the context vectors discussed in this section capture meaning, in that words with similar meaning have similar context vectors and unrelated words have dissimilar context vectors.

Two further comments of a technical nature are in order, one mathematical, the other linguistic. We have seen above

that the sum-vector of high-dimensional random vectors is similar to the vectors that make up the sum and it is therefore a good representation of a set. When the context of a word is defined as a set of documents, as above, it is naturally represented by the sum of the vectors for those documents. That is exactly what random indexing does: a context vector is the sum of the random index vectors for the documents in which the word occurs. Thus two words that share contexts share many documents, and so their context vectors share many index vectors in their respective sums, making the sums—i.e., the context vectors—similar.

The other comment concerns the linguistic adequacy of context vectors. The contexts of words contain much richer linguistic information than is captured by the context vectors in the examples above. In fact, these context vectors are linguistically impoverished and crude—with language we can tell a story, with a bag of words we might be able to tell what the story is about. The technical reason is that only one operation is used for making the context vectors, namely, vector addition, and so only sets can be represented adequately. However, other operations on vectors besides addition have already been mentioned, and they can be used for encoding relational information about words. The making of linguistically richer context vectors is possible but mostly unexplored.

To sum up, high-dimensional random vectors—that is, large random patterns—can serve as the basis of a cognitive code that captures regularities in data. The simplicity and flexibility of random-vector methods can surpass those of more exact methods, and the principles apply to a wide range of tasks—beyond the computing of context vectors. They are particularly apt for situations where data keep on accumulating. Thus random-vector-based methods are good candidates for use in incremental on-line learning and in building a cognitive code.

Learning to Infer by Holistic Mapping; Learning from Example

Logic deals with inference. It lets us write down general statements—call them rules—which, when applied to specific cases, yield specific statements that are true. Here we look at such rules in terms of hyperdimensional arithmetic.

Let us look at the rule 'If x is the mother of y and y is the father of z then x is the grandmother of z .' If we substitute the names of a specific mother, son, and baby for x , y , and z , we get a true statement about a specific grandmother. How might the rule be encoded in distributed representation, and how might it be learned from specific examples of it?

Here we have three relations, 'mother of', 'father of', and 'grandmother of'; let us denote them with the letters M ,

F , and G . Each relation has two constituents or arguments; we will label them with subscripts 1 and 2. That x is the mother of y can then be represented by $M_{xy} = M_1 * X + M_2 * Y$. Binding X and Y to two different vectors M_1 and M_2 keeps track of which variable, x or y , goes with which of the two arguments, and the sum combines the two bound pairs into a vector representing the relation ‘mother of’. Similarly, $F_{yz} = F_1 * Y + F_2 * Z$ for ‘father of’ and $G_{xz} = G_1 * X + G_2 * Z$ for ‘grandmother of’.

Next, how to represent the implication? The left side—the antecedent—has two parts combined with an ‘and’; we can represent it with addition: $M_{xy} + F_{yz}$. The right side—the consequent G_{xz} —is implied by the left; we need an expression that maps the antecedent to the consequent. With XOR as the multiplication operator, the mapping is effected by the product-vector

$$R_{xyz} = G_{xz} * (M_{xy} + F_{yz})$$

So the mapping R_{xyz} represents our rule and it can be applied to specific cases of mother, son, baby.

Now let us apply the rule. We will encode ‘Anna is the mother of Bill’ with $M_{ab} = M_1 * A + M_2 * B$ and ‘Bill is the father of Cid’ with $F_{bc} = F_1 * B + F_2 * C$, combine them into the antecedent $M_{ab} + F_{bc}$, and map it with the rule R_{xyz} :

$$R_{xyz} * (M_{ab} + F_{bc}) = G_{xz} * (M_{xy} + F_{yz}) * (M_{ab} + F_{bc})$$

The resulting vector, we will call it G'_{ac} , is more similar to G_{ac} (i.e., more similar to $G_1 * A + G_2 * C$) than to any other vector representing a relation of these same elements, thus letting us infer that Anna is the grandmother of Cid.

The above example of inference can also be interpreted as learning from example. It uses a traditional formal framework with variables and values to represent relations, merely encoding them in distributed representation. The traditional framework relies on two-place relations and on the variables x , y , and z to identify individuals across the relations that make up the rule. However, because variables in distributed representation are represented explicitly by vectors, just as individuals are, the encoding of the rule ‘mother–son–baby implies grandmother’, and of an instance of it involving Anna, Bill, and Cid, are identical in form. We can therefore regard the rule itself as a specific instance of it(elf); we can regard it as an *example*. Thus we can interpret the above description as computing from one example or instance of mother–son–baby implying grandmother another instance of grandmother. It is remarkable that learning from a single example would lead to the correct inference.

We can go further and learn from several examples. If one example gives us the mapping (rule) R_{xyz} and we have another example involving u , v , and w —think of them as a second set of specific individuals—we can recompute the

same “rule” to get $R_{uvw} = G_{uv} * (M_{uv} + F_{vw})$. If we combine these two rules simply by adding them together, we get an improved rule based on two examples: $R = R_{xyz} + R_{uvw}$. The new rule is better in the sense that if applied to—multiplied by—the antecedent involving Anna, Bill, and Cid, as above, we get a vector G''_{ac} that improves upon G'_{ac} by being closer to G_{ac} . We can go on adding examples, further improving the result somewhat. This can also be thought of as learning by analogy. The thing to note is that everything is done with simple arithmetic on random hypervectors.

So as not to give the impression that all kinds of inference will work out as simply as this, we need to point out when they don’t. Things work out here because the relations in the antecedent and the consequent are different. However, some of them could be the same. Examples of such include (1) ‘if x is the mother of y and y is a brother of z (and not half-brother) then x is the mother of z ,’ and the transitive relation (2) ‘if x is a brother of y and y is a brother of z (different from x) then x is a brother of z .’ When these are encoded in the same way as the mother–son–baby example above and the resulting rule applied to a , b , and c , the computed inference correlates positively with the correct inference but a relation that is a part of the antecedent—a tautology—correlates more highly; in both cases ‘ b is a brother of c ’ wins over the intended conclusion about a ’s relation to c . An analysis shows the reason for the failure. It shows that the mapping rule includes the identity vector, which then takes the antecedent into the computed inference. The analysis is not complicated but it is lengthy and is not presented here.

A major advantage of distributed representation of this kind is that it lends itself to analysis. We can find out why something works or fails, and what could be done to work around a failure.

What is the Dollar of Mexico?

Much of language use, rather than being literal, is indirect or figurative. For example, we might refer to the peso as the Mexican dollar because the two have the same role in their respective countries. For the figurative expression to work, we must be able to infer the literal meaning from it. That implies the need to compute the literal meaning from the figurative.

The following example suggests that the inference can be achieved with holistic mapping. We will encode the country (x) and its monetary unit (y) with a two-field “record.” The holistic record for the United States then is $A = X * U + Y * D$ and for Mexico it is $B = X * M + Y * P$, where U , M , D , P are random 10,000-bit vectors representing United States, Mexico, dollar, and peso, respectively.

From the record for United States A we can find its monetary unit by unbinding (multiplying it) with the variable Y . We can also find what role dollar plays in A by multiplying it with the dollar D : $D * A = D * X * U + D * Y * D = D * X * U + Y \approx Y$. If we take the literal approach and ask what role dollar plays in the record for Mexico B we get nonsense: $D * B = D * X * M + D * Y * P$ is unrecognizable. But we have already found out above the role that dollar plays in another context, namely the role Y , and so we can use it to unbind B and get P' that is similar to P for peso. The interesting thing is that we can find the Mexican dollar without ever explicitly recovering the variable Y ; we simply ask what in Mexico corresponds to the dollar in the United States? This question is encoded with $(D * A) * B$, and the result approximately equals P . The math is an exercise in distributivity, with vectors occasionally canceling out each other, and is given here in detail:

$$\begin{aligned}
 (D * A) * B &= (D * (X * U + Y * D)) * (X * M + Y * P) \\
 &= (D * (X * U) + D * (Y * D)) * (X * M + Y * P) \\
 &= (D * X * U + D * Y * D) * (X * M + Y * P) \\
 &= (D * X * U + Y) * (X * M + Y * P) \\
 &= (D * X * U + Y) * (X * M) + (D * X * U + Y) \\
 &\quad * (Y * P) \\
 &= ((D * X * U) * (X * M) + Y * (X * M)) \\
 &\quad + ((D * X * U) * (Y * P) + Y * (Y * P)) \\
 &= (D * X * U * X * M + Y * X * M) \\
 &\quad + (D * X * U * Y * P + Y * Y * P) \\
 &= (D * U * M + Y * X * M) \\
 &\quad + (D * X * U * Y * P + P)
 \end{aligned}$$

The only meaningful term in the result is P . The other three terms act as random noise.

Cognitive Structure Based on Prototypes

The last two examples let us question the primacy of variables in cognitive representation. We have learned to think in abstract terms such as country and monetary unit and to represent more concrete objects in terms of them, as above, but we can also think in terms of prototypes and base computing on them, accepting expressions such as ‘the dollar of Mexico’ and ‘the dollar of France’ as perfectly normal. In fact, this is more like how children start out talking. Mom and Dad are specific persons to them, and somebody else’s mother and father become understood in terms of my relation to Mom and Dad. The instances encountered early in life become the prototypes, and later instances are understood in terms of them. This kind of prototyping is very apparent to us when as adults we are

learning a second language. To make sense of what we hear or read, we translate into our native tongue. Even after becoming fluent in the new language, idioms of the mother tongue can creep into our use of the other tongue.

To reflect this view, we can leave out X and Y from the representations above and encode United States as a prototype, namely, $A_0 = U + D$. The holistic record for Mexico is then encoded in terms of it, giving $B_0 = U * M + D * P$. The dollar of Mexico now becomes simply $D * B_0 = D * (U * M + D * P) = D * U * M + D * D * P = D * U * M + P = P' \approx P$, with U and D taking the place of the variables X and Y . Using U and D as variables, we can in turn interpret ‘the peso of France’ exactly as ‘the dollar of Mexico’ is interpreted in the original example.

Looking Back

Artificial neural-net associative memories were the first cognitive models to embrace truly high dimensionality and to see it as a possible asset. The early models were the linear correlation-matrix memories of Anderson [1] and Kohonen [2] that equate stored patterns with the eigenvectors of the memory (weight) matrix. Later models were made nonlinear with the application of a squashing function to the memory output vector, making stored patterns into point attractors. The best-known of these models is the Hopfield net [3]. They have one matrix of weights, which limits the memory storage capacity—the number of patterns that can be stored—to a fraction of the dimensionality of the stored vectors. By adding a fixed layer (a matrix) of random weights, the Sparse Distributed Memory [4] allows the building of associative memories of arbitrarily large capacity. The computationally most efficient implementation of it, by Karlsson [5], is equivalent to the RAM-based WISARD of Aleksander et al. [6]. Representative early work on associative memories appears in a 1981 book edited by Hinton and Anderson [7], more recent by Hassoun [8], and more detailed analyses of these memories have been given by Kohonen [9] and Palm [10].

The next major development is marked by the 1990 special issue of *Artificial Intelligence* (vol. 46) on connectionist symbol processing edited by Geoffrey Hinton. In it Hinton [11] argues for the necessity of a reduced representation if structured information such as hierarchies were to be handled by neural nets. Smolensky [12] introduced tensor-product variable binding, which allows the (neural-net-like) distributed representation of traditional symbolic structures. However, the tensor product carries all low-level information to each higher level at the expense of increasing the size of the representation—it fails to reduce. This problem was solved by Plate in the holographic reduced representation (HRR) [13]. The solution

compresses the $n \times n$ outer product of two real vectors of dimensionality n into a single n -dimensional vector with *circular convolution*, it being the multiplication operator. The method requires a clean-up memory to recover information that is lost when the representation is reduced. The problem of clean-up had already been solved, in theory at least, by autoassociative memory. We now have a system of n -dimensional distributed representation with operators for addition and multiplication, that is *closed* under these operations and sufficient for encoding and decoding of compositional structure, as discussed above.

Plate also discusses HRR with complex vectors [14]. The addition operator for them is componentwise addition, as above, and the multiplication operator is componentwise multiplication. HRR for binary vectors is called the Spatter Code [15] for which componentwise XOR is an appropriate multiplication operator; for the equivalent bipolar spatter code it is componentwise multiplication, making the spatter code equivalent to the complex HRR when the “complex” vector components are restricted to the values 1 and -1 .

The circular convolution includes all $n \times n$ elements of the outer-product matrix. However, Plate points out that multiplication can also be accomplished with a subset of the elements. The simplest such has been used successfully by Gayler [16] by taking only the n diagonal elements of the outer-product matrix, making that system a generalization of the bipolar spatter code.

Permutation is a very versatile multiplication operator for hyperdimensional vectors, as discussed above. Rachkovskij and Kussul use it to label the variables of a relation [17], and Kussul and Baidyk [18] mark positions of a sequence with permutations. Gayler [16] uses permutations for “hiding” information in holographic representation. Rachkovskij and Kussul [17] use them for Context-Dependent Thinning, which is a method of normalizing binary vectors—that is, of achieving a desired sparseness in vectors that are produced by operations such as addition.

When a variable that is represented with a permutation is bound to a value that is represented with a hypervector, the inverse permutation will recover the value vector. Similarly, when a holistic record of several variables is constructed as a sum of permuted values—each variable having its own random permutation—the inverse permutations will recover approximate value vectors. However, there is no practical way to compute the permutations—to find the variables—from the holistic record and to determine what variable is associated with a given value. In that sense binding with vector multiplication and with permutation are very different.

Another thread in the development of these models leads to LSA, which is described in detail by Landauer and Dumais [19]. LSA takes a large sparse matrix of word

frequencies in documents and represents it with several hundred dominant principal components of the (transformed) frequency matrix. The desire to avoid the computational task of extracting principal components inspired Random Indexing by Kanerva et al. [20], the idea of which is discussed above. Random indexing is a special case of Random Projections by Papadimitriou et al. [21] and Random Mappings by Kaski [22]. All are examples of low-distortion geometric embedding, which has been reviewed by Indyk [23].

Language is a prime motivator and a rich source of ideas and challenges for hyperdimensional models. The original word-space model of Schütze [24] and the hyperspace analogue to language (HAL) model of Lund et al. [25], as well as LSA, are here called “exact” because they do not distribute the frequency information with random vectors. Sahlgren’s [26] results at capturing word meaning with random indexing are comparable. However, context vectors that are based solely on the co-occurrence of words ignore a major source of linguistic information, namely, grammar. First attempts at including grammar have been made by encoding word order into the context vectors. Jones and Mewhort [27] do it with circular convolution applied to real-valued HRR-vectors, Sahlgren et al. [28] do it with permutations applied to ternary random-index vectors. Notice that both use multiplication—both circular convolution and permutation are multiplication operators. Widdows [29] covers numerous studies that represent word meaning with points of a high-dimensional space.

We can conclude from all of the above that we are dealing with very general properties of high-dimensional spaces. There is a whole family of mathematical systems that can be used as the basis of computing, referred to here as hyperdimensional computing and broadly covered under HRR, the definitive work on which is Plate’s book [14] based on his 1994 PhD thesis.

Looking Forth; Discussion

In trying to understand brains, the most fundamental questions are philosophical: How does the human mind arise from the matter we are made of? What makes us so special, at least in our own eyes? Can we build robots with the intelligence of, say, a crow or a bear? Can we build robots that will listen, understand, and learn to talk?

According to one view, such questions will be answered in the positive once we understand how brains compute. The seeming paradox of the brain’s understanding its own understanding is avoided by modeling. If our theories allow us to build a system whose behavior is indistinguishable from the behavior of the intended “target” system, we have

understood that system—the theory embodies our understanding of it. This view places the burden on modeling.

This paper describes a set of ideas for cognitive modeling, the key ones being very high dimensionality and randomness. They are a mathematical abstraction of certain apparent properties of real neural systems, and they are amenable to building into cognitive models. It is equally important that cognition, and behavior in general, is described well at the phenomenal level with all their subtleties, for example, how we actually think—or fail to—how we remember, forget, and confuse, how we learn, how we use language, what are the concepts we use, their relation to perception. With all of it being somehow produced by our brains, the modeler's task is to find a plausible explanation in underlying mechanisms. That calls for a deep understanding of both the phenomenon and the proposed mechanisms.

Experimental psychologists have a host of ways of testing and measuring behavior. Examples include reaction time, memory recognition and recall rates, confusions and errors introduced by priming and distractions, thresholds of perception, judgments of quantity, eye-tracking, and now also imaging brain activity. We can foresee the testing of hyperdimensional cognitive codes in a multitude of psychological experiments.

If you have never doubted your perceptions, visit a psychophysicist—or a magician. It is amazing how our senses are fooled. All the effects are produced by our nervous systems and so tell of its workings. They seriously challenge our cognitive modeling, and serve as a useful guide. Hyperdimensional representation may explain at least some illusions, and possibly our bistable perception of the Necker cube.

Language has been cited above as a test-bed for ideas on representation, for which it is particularly suited on several accounts. The information has already been filtered by our brains and encoded into letters, words, sentences, passages, and stories. It is therefore strongly influenced by the brain's mechanisms, thus reflecting them. Linguists can tell us about language structure, tolerance of apparent ambiguity, stages of learning, literal and figurative uses, slips of the tongue, and much more, presenting us with a host of issues to challenge our modeling. Data are available in ever-increasing amounts on the Internet, in many languages, easily manipulated by computers. If we were to limit the development and testing of ideas about the brain's representations and processing to a single area of study, language would be an excellent choice. Our present models barely scratch the surface.

Neuroscience can benefit from mathematical ideas about representation and processing. Work at the level of individual neurons cannot tell us much about higher mental functions, but theoretical—i.e., mathematical—

considerations can suggest how an individual component or a circuit needs to work to achieve a certain function. The mathematical modeler, in turn, can follow some leads and dismiss others by looking at the neural data.

It has been pointed out above that no two brains are identical yet they can be equivalent. The flip side is individual differences, which can be explained by randomness. An individual's internal code can be especially suited or unsuited for some functions simply by chance. This is particularly evident in the savant's feats of mental arithmetic, which to a computer engineer is clearly a matter of the internal code. The blending of sensory modalities in synesthesia is another sign of random variation in the internal code. The specifics of encoding that would result in these and other anomalies of behavior and perception are yet to be discovered—as are the specifics that lead to normal behavior! The thesis of this paper is that discovering the code is a deeply mathematical problem.

The mathematics of hyperdimensional representation as discussed above is basic to mathematicians, and the models based on it will surely fall short of explaining the brain's computing. Yet, they show promise and could pave the way to more comprehensive models based on deeper mathematics. The problem is in identifying mathematical systems that mirror ever more closely the behavior of cognitive systems we want to understand. We can hope that some mathematicians become immersed in the problem and will show us the way.

Of the ideas discussed in this paper, random indexing is ready for practical application. The example here is of language, but the method can be used in any task that involves a large and ever increasing sparse matrix of frequencies. The analysis of dynamic networks of many sorts—social networks, communications networks—comes readily to mind, but there are many others. The benefit is in being able to accommodate unpredictable growth in data within broad limits, in a fixed amount of computer memory by distributing the data randomly and by reconstructing it statistically when needed.

The ideas have been presented here in terms familiar to us from computers. They suggest a new breed of computers that, contrasted to present-day computers, work more like brains and, by implication, can produce behavior more like that produced by brains. This kind of neural-net computing emphasizes computer-like operations on vectors—directly computing representations for composite entities from those of the components—and deemphasizes iterative searching of high-dimensional “energy landscapes,” which is at the core of many present-day neural-net algorithms. The forming of an efficient energy landscape in a neural net would still have a role in making efficient item memories.

Very large word size—i.e., hyperdimensionality—means that the new computers will be very large in terms of

numbers of components. In light of the phenomenal progress in electronics technology, the required size will be achieved in less than a lifetime. In fact, computer engineers will soon be looking for appropriate architectures for the massive circuits they are able to manufacture. The computing discussed here can use circuits that are not produced in identical duplicates, and so the manufacturing of circuits for the new computers could resemble the growing of neural circuits in the brain. It falls upon those of us who work on the theory of computing to work out the architecture. In that spirit, we are encouraged to explore the possibilities hidden in very high dimensionality and randomness.

A major challenge for cognitive modeling is to identify mathematical systems of representation with operations that mirror cognitive phenomena of interest. This alone would satisfy the engineering objective of building computers with new capabilities. The mathematical systems should ultimately be realizable in neural substratum. Computing with hyperdimensional vectors is meant to take us in that direction.

Acknowledgements Real World Computing Project funding by Japan's Ministry of International Trade and Industry to the Swedish Institute of Computer Science in 1994–2001 made it possible for us to develop the ideas for high-dimensional binary representation. The support of Dr. Nobuyuki Otsu throughout the project was most valuable. Dr. Dmitri Rachkovskij provided information on early use of permutations to encode sequences by researchers in Ukraine. Dikran Karagueuzian of CSLI Publications accepted for publication Plate's book on Holographic Reduced Representation after a publishing agreement elsewhere fell through. Discussions with Tony Plate and Ross Gayler have helped shape the ideas and their presentation here. Sincere thanks to you all, as well as to my coauthors on papers on representation and to three anonymous reviewers of the manuscript.

References

1. Anderson JA. A simple neural network generating an interactive memory. *Math Biosci.* 1972;14:197–220.
2. Kohonen T. Correlation matrix memories. *IEEE Trans Comput.* 1984;C21(4):353–9.
3. Hopfield JJ. Neural networks and physical systems with emergent collective computational abilities. *Proc Natl Acad Sci USA.* 1982;79(8):2554–8.
4. Kanerva P. *Sparse distributed memory.* Cambridge, MA: MIT Press; 1988.
5. Karlsson R. A fast activation mechanism for the Kanerva SDM memory. In: Uesaka Y, Kanerva P, Asoh H, editors. *Foundations of real-world computing.* Stanford: CSLI; 2001. p. 289–93.
6. Aleksander I, Stonham TJ, Wilkie BA. Computer vision systems for industry: WISARD and the like. *Digit Syst Ind Autom.* 1982;1:305–23.
7. Hinton GH, Anderson JA, editors. *Parallel models of associative memory.* Hillsdale, NJ: Erlbaum; 1981.
8. Hassoun MH, editor. *Associative neural memories: theory and implementation.* New York, Oxford: Oxford University Press; 1993.
9. Kohonen T. *Self-organization and associative memory.* 3rd ed. Berlin: Springer; 1989.
10. Palm G. *Neural assemblies: an alternative approach to artificial intelligence.* Heidelberg: Springer; 1982.
11. Hinton GE. Mapping part-whole hierarchies into connectionist networks. *Artif Intell.* 1990;46(1–2):47–75.
12. Smolensky P. Tensor product variable binding and the representation of symbolic structures in connectionist networks. *Artif Intell.* 1990;46(1–2):159–216.
13. Plate T. Holographic Reduced Representations: convolution algebra for compositional distributed representations. In: Mylopoulos J, Reiter R, editors. *Proc. 12th int'l joint conference on artificial intelligence (IJCAI).* San Mateo, CA: Kaufmann; 1991. p. 30–35.
14. Plate TA. *Holographic reduced representation: distributed representation of cognitive structure.* Stanford: CSLI; 2003.
15. Kanerva P. Binary spatter-coding of ordered K -tuples. In: von der Malsburg C, von Seelen W, Vorbruggen JC, Sendhoff B, editors. *Artificial neural networks – ICANN 96 proceedings (Lecture notes in computer science, vol. 1112).* Berlin: Springer; 1996. p. 869–73.
16. Gayler RW. Multiplicative binding, representation operators, and analogy. Poster abstract. In: Holyoak K, Gentner D, Kokinov B, editors. *Advances in analogy research.* Sofia: New Bulgarian University; 1998. p. 405. Full poster <http://cogprints.org/502/>. Accessed 15 Nov 2008.
17. Rachkovskij DA, Kussul EM. Binding and normalization of binary sparse distributed representations by context-dependent thinning. *Neural Comput.* 2001;13(2):411–52.
18. Kussul EM, Baidyk TN. On Information encoding in associative-projective neural networks. Report 93-3. Kiev, Ukraine: V.M. Glushkov Inst. of Cybernetics; 1993 (in Russian).
19. Landauer T, Dumais S. A solution to Plato's problem: the Latent Semantic Analysis theory of acquisition, induction and representation of knowledge. *Psychol Rev.* 1997;104(2):211–40.
20. Kanerva P, Kristoferson J, Holst A. Random Indexing of text samples for latent semantic analysis. Poster abstract. In: Gleitman LR, Josh AK, editors. *Proc. 22nd annual conference of the Cognitive Science Society.* Mahwah, NJ: Erlbaum; 2000. p. 1036. Full poster <http://www.rni.org/kanerva/cogsci2k-poster.txt>. Accessed 23 Nov 2008.
21. Papadimitriou C, Raghavan P, Tamaki H, Vempala S. Latent semantic indexing: a probabilistic analysis. *Proc. 17th ACM symposium on the principles of database systems.* New York: ACM Press; 1998. p. 159–68.
22. Kaski S. Dimensionality reduction by random mapping: fast similarity computation for clustering. *Proc. int'l joint conference on neural networks, IJCNN'98.* Piscataway, NJ: IEEE Service Center; 1999. p. 413–8.
23. Indyk P. Algorithmic aspects of low-distortion geometric embeddings. Annual symposium on foundations of computer science (FOCS) 2001 tutorial. <http://people.csail.mit.edu/indyk/tut.ps>. Accessed 15 Nov 2008.
24. Schütze H. Word space. In: Hanson SJ, Cowan JD, Giles CL, editors. *Advances in neural information processing systems 5.* San Mateo, CA: Kaufmann; 1993. p. 895–902.
25. Lund K, Burgess C, Atchley R. Semantic and associative priming in high-dimensional semantic space. *Proc. 17th annual conference of the Cognitive Science Society.* Mahwah, NJ: Erlbaum; 1995. p. 660–5.
26. Sahlgrén M. *The word-space model.* Doctoral dissertation. Department of Linguistics, Stockholm University; 2006. <http://www.sics.se/~mange/TheWordSpaceModel.pdf>. Accessed 23 Nov 2008.

27. Jones MN, Mewhort DJK. Representing word meaning and order information in a composite holographic lexicon. *Psychol Rev.* 2007;114(1):1–37.
28. Sahlgren M, Holst A, Kanerva P. Permutations as a means to encode order in word space. *Proc. 30th annual conference of the Cognitive Science Society.* Austin, TX: Cognitive Science Society. p. 1300–5.
29. Widdows D. *Geometry and meaning.* Stanford: CSLI; 2004.