Extended Sparse Distributed Memory

Javier SNAIDER[1], Stan FRANKLIN
Computer Science Department & Institute for Intelligent Systems, The University of Memphis

Abstract. Sparse distributed memory is an auto-associative memory system that stores high dimensional Boolean vectors. Here we present an extension of the original SDM that uses word vectors of larger size than address vectors. This extension preserves many of the desirable properties of the original SDM: auto-associability, content addressability, distributed storage, robustness over noisy inputs. In addition, it adds new functionality, enabling an efficient auto-associative storage of sequences of vectors, as well as of other data structures such as trees.

Keywords. Sparse Distributed Memory, Episodic Memory, Sequence Representation, Cognitive Modeling

Introduction

First proposed by Kanerva [1], sparse distributed memory (SDM) is based on large binary vectors, and has several desirable properties. It is distributed, auto associative, content addressable, and noise robust. Moreover, this memory exhibits interesting psychological characteristics as well (interference, knowing when it doesn't know, the tip of the tongue effect), that make it an attractive option with which to model episodic memory [2][3]. SDM is still being implemented for various applications (e.g., [4][5][6]). Several improvements have been proposed for SDM; for example Ramamurthy and colleagues introduced forgetting as part of an unsupervised learning mechanism [7][8]. The same authors also proposed the use of ternary vectors, introducing a "don't care" symbol as a third possible value for the dimensions of the vectors [9]. Kanerva, in his original work, described the use the SDM to store sequences. His procedure has the disadvantage of losing most of the auto-associativeness and noise robustness of the memory. Later he proposed hyperdimentional arithmetic as a new mechanism for storing sequences and other data structures such as sets and records [10]. Even though this new mechanism is an improvement over the original SDM mechanism, it is still limited in its noise robustness, and it is very sensitive to interference (see below).

Here we propose an extension to the original SDM that is especially suitable for storing sequences and other data structures such as trees. This extension can also improve the hyperdimensional arithmetic introduced by Kanerva. In the following section we briefly describe SDM. Then we introduce Extended SDM, discussing several uses of this extension and its results. Finally we propose some future directions.

[1] Corresponding Author: FedEx Institute of Technology #403h, 365 Innovation Dr., Memphis, TN 38152; E-mail: jsnaider@memphis.edu

1. Sparse Distributed Memory

Here we present a brief introduction to SDM concepts. Both leisurely descriptions [11] and highly detailed descriptions [1] are available. Readers already familiar with SDM can skip this section.

SDM implements a content addressable random access memory. Its address space is of the order of 2^{1000} or even more. Both addresses and words are binary vectors whose length equals the number of dimensions of the space. In this example, we will think of bit vectors of 1000 dimensions. To calculate distances between two vectors in this space, the Hamming distance is used. Surprisingly, and of importance to SDM, the distances from a point of the space to any other point in the space are highly concentrated around half of the maximum distance. In our example, more than 99.9999% of the vectors are at a distance between 422 and 578 from a given vector of the space [1].

To construct the memory, a sparse uniformly distributed sample of addresses, on the order of 2^{20} of them, is chosen. These addresses are called hard locations. Only hard locations can store data. Several hard locations participate in the storing and retrieving of any single word of data. Each hard location has a fixed address, and contains one counter for each dimension. In our example, each hard location has 1000 counters. A counter is just an integer counter with a range of -40 to 40. Counters can be incremented or decremented in steps of size one.

To write a word vector in a hard location, for each dimension, if the bit of this dimension in the word is 1, the corresponding counter is incremented. If it is 0, the counter is decremented. To read a word vector from a hard location, we compute a vector such that, for each dimension, if the corresponding counter in the hard location is positive, 1 is assigned to this dimension in the vector being read, otherwise 0 is assigned.

When a vector is written in an address in the SDM, it is stored in several hard locations. In the same way, to read from an address in the SDM, the read vector is a composition of the readings of several hard locations. To determine which hard locations are used to read or write, the access sphere is defined. The access sphere for an address vector is a sphere with center at this address that on average encloses 0.1% of all the hard locations of the memory. The radius of the access sphere depends on the number of dimensions of the space. For example, for a SDM with 1000 dimensions, the radius of the access sphere is 451. In the example, the access sphere will contain any hard location whose address is less than 451 away from the address vector.

To write a word vector in any address of the memory, the word is written to all hard locations inside the access sphere of the address. To read from any address, all hard locations in the access sphere of the address vector are read and a majority rule for each dimension is applied.

In general, the SDM is used as an auto-associative memory, so the address vector is the same as the word vector. In this case, after writing a word in the memory, the vector can be retrieved using partial or noisy data. If the partial vector is inside a critical distance from the original one, and it is used as address with which to cue the memory, the read vector will be close to the original one. This critical distance depends on the number of vectors already stored in the memory. If the process is repeated, using the first recovered vector as address, the new reading will be even closer to the original. After a few iterations, typically less than ten, the readings converge to the original vector. If the partial or noisy vector is farther away than the critical distance, the

successive readings from the iterations will diverge. If the partial vector is about at the critical distance form the original one, the iterations yields vectors that are typically at the same critical distance from the original vector. This behavior mimics the "tip of the tongue" effect.

When storing sequences of vectors in this SDM, the address cannot be the same as the word, as it is in the auto-associative use. The vector that represents the first element of the sequence is used as address to read the memory. The read vector is the second element in the sequence. This second vector is used as address to read the memory again to retrieve the third element. This procedure is repeated until the whole sequence is retrieved. The problem with this way of storing sequences is that it is not possible to use iterations to retrieve elements of the sequence from noisy input cues. So, the memory is far less robust.

Kanerva [10] introduced hyperdimensional computing, based on large binary vectors, as an appropriate tool for cognitive modeling, including holistic representation of sets, sequences and mappings. Among the various vector operations proposed, multiplication of binary vectors as bitwise xor, permutation, and sum with normalization are relevant to the present work, and will be discussed here.

When two binary vectors are combined using bitwise xor, the result of this operation is a new vector of the same dimensionality as the original ones. This operation has several interesting properties. First, the resulting vector is dissimilar, i.e. farther than the critical distance, to the two original ones. Second, the xor operation is reversible.

$$\text{If } A \times B = C \text{ then } C \times B = A \text{ and } C \times A = B$$

Third, this operation preserves Hamming distances.

Permutation is an operation that shuffles the positions (dimensions) of one vector. Mathematically, this corresponds to multiplying the vector by a square matrix M with one 1 in each row and column while the other positions contain 0.

$$\text{Permutation } (A) = (AM)^T$$

This operation is also reversible, multiplying by M^T, and it preserves Hamming distances as well.

Finally, the sum operation is the arithmetic (integer) sum of the values of each dimension of two or more vectors. For this operation the bipolar representation of the vectors, i.e., the value 0 is replaced by -1, is used. The resulting vector is an integer vector. To transform this vector into a binary vector, a normalization operation is required. If one dimension has a positive value, the normalized binary vector has a 1 in this dimension. If the value is negative, the normalized vector has a 0 in this dimension. Ties are resolved at random. The sum with normalization has interesting properties: the resulting vector is similar to each of the vectors summed up; i.e. the distance between them is less than the expected distance between any two vectors in the space. Also, xor multiplication distributes over the sum.

Based in these properties, it is sometimes possible to retrieve the individual added vectors from the sum vector. This is feasible only if the number of vectors added is small, i.e. three or fewer vectors. Even with this small number, the interference among the vectors in the sum makes the retrieval of the original vectors from the sum not very reliable.

Kanerva describes how to store sequences of vectors using hyperdimensional arithmetic [10]. We will briefly describe this procedure and compare it with our implementation in section 3. The main problem of this procedure is that it uses the sum operation, and so shares the same problems with the sum mentioned above while reconstructing the sequence. Also it uses permutation, and as we discuss before, this operation requires matrices that are outside of the binary vector domain.

2. Extended SDM

Here we present a novel structure, built upon SDM, that we call extended sparse distributed memory (ESDM). The main idea of this new memory structure is the use of vectors with different lengths for the addresses and the words. A word has a longer length than the address in which it is stored. Each address has n dimensions while each word has m dimensions with n<m. Moreover, the address vector is included in the word vector (See Figure 1). Formally, a word of length m and an address with length n, the first n bits of the word compose the address.

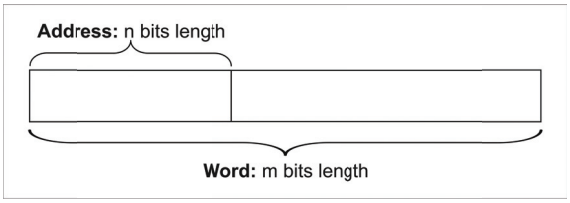

Figure 1 A word vector with its address section.

The structure of this new memory system is similar to the original SDM. It is composed of hard locations, each of which has an address and counters. The address is a fixed vector of length n. But each hard location has m counters, where m is greater than n. To store a word vector in the memory, the procedure is the same as described for SDM, except that now the first n bits of the word are used as address. To read from an address in the memory, again the procedure is similar to the one used for SDM. During each iteration, a word is read from the memory and its first n bits are used to read in the next iteration.

Formally, the address vector is $A = (WM)^T$, where A is an address vector of size n, W is the word vector of size m and M is a n x m rectangular diagonal matrix with all 1s in the diagonal.

It is important to notice that the whole word vector, including the address, comprises the useful data. Conceptually, this memory is a mix of auto-associative and hetero-associative memories. The address part of the word is auto-associative whereas the rest of the word is hetero-associative. This allows us to preserve, and even to improve, the desirable characteristics of the SDM. First, with an initial vector as address to cue the memory, it is possible to retrieve the corresponding word, even if the initial vector is a noisy version of the stored one. This means that ESDM maintains the noise robustness characteristic of SDM. Second, the data of each vector is stored in a number of hard locations in a distributed way. So, it is also robust in the case that some hard locations are corrupted or lost. Third, the previously discussed psychological

characteristics in SDM are also present in ESDM. Finally, the hetero-associative part of the words in ESDM allows storing other data related with the address data but without interfering with it. This is a notable improvement over the original SDM that relies on the flawed sum operation to achieve the same goal but with far less effectiveness.

3. Storing sequences and other data structures

Sequences are important representations for cognitive agents. Agents act over time and cognitive agents adapt *and* act over time. Simple events can be combined into more complex ones forming sequences, or even trees, of simpler events [12][13].

In section 1 we mentioned two approaches suggested by Kanerva [1][10] for storing sequences in SDM. We also mentioned that both approaches have important disadvantages that weaken the auto-associatively, content addressability and noise robustness properties of the memory. The implementation of sequence storing in ESDM is straightforward and eliminates these disadvantages. The most basic implementation uses addresses of length n and words of length 2n, as shown in figure 2. The sequence is composed of vectors of length n. To store the sequence, the first two vectors E_1 and E_2 are concatenated forming a word of length 2n. We will say that the word has two *sections* of n bits each. This word is stored in address E_1. Then E_2 and E_3 are concatenated and stored in address E_2. The process continues until the full sequence is stored. A special vector can be used to indicate the end of the sequence.

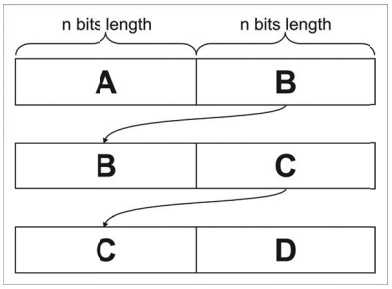

Figure 2 Basic sequence representation using 2n word vectors

To retrieve the sequence, the initial vector of the sequence is used to read a word from the memory. This word is divided in two halves. The second half is the second vector in the sequence. Repeating this procedure, the whole sequence is retrieved. Notice that in each reading during the retrieval of the sequence, the vector used as address can have some noise, but the iterating reading from the memory cleans it up, as explained previously. One problem with this implementation occurs when two sequences that share a common vector are stored in the memory. For example:

AB**CD**E and FG**CH**I

In the example, the word CD is stored in address C but the word CH is stored in C also. This produces the undesirable interference between D and H that prevents the

correct retrieval of one or even both of the sequences. One plausible solution is to use the same procedure proposed by Kanerva using hyperdimensional operations. The first reading from the memory again uses the initial vector of the sequence. But the following addresses are calculated using the previously read vectors of the sequence. An elegant combination is achieved using permutation and sum operations [10]. For example if P() denotes a random permutation, then:

$$A_3 = [P(E_1)+E_2]$$

With this address we read the memory and from the read word the next vector of the sequence, i.e. E_3, is retrieved. The following addresses are calculated in the same way.

$$A_{i+1} = [P(A_i)+E_i]$$

An interesting option is to preserve the sum of the vectors in each reading and multiply it by a scalar *k* between 0 and 1, for example 0.8. This produces an effect of *fading away* of the old vectors of the sequence in the calculation of the next address.

$$A'_{i+1} = k*P(A'_i)+E_i$$

$$A_{i+1} = [A'_{i+1}]$$ where A' is the real vector with the sum before normalization.

All these equations can be used in the original SDM, as pointed out by Kanerva. In both situations, operations with sums are used but the advantage of this implementation is that the retrieval of the succeeding vector in the sequence does not depend on operations that extract the vector from the sum. Here the sum is used only to compute the next address, but the vector is extracted directly from the second part of the read word.

In a similar way, other data structures can be stored in ESDM. For example, to store binary trees, addresses of length n and words of length 3n are used. With the address of the root of the tree the first word is retrieved. The word is divided into three sections, left, center and right. The left section holds the content of the node in the tree; the center section is used as an address with which to read the left child node of the tree; the right section holds the address of the right child node. This procedure is repeated until the whole tree is retrieved. Notice that here again noisy vectors can be used, and ESDM takes care of cleaning them up. Also, a similar mechanism to the one described for sequences can be used to avoid problems related to repeated vectors in several structures.

Other data structures can be easily derived from sequences and trees. A *double linked sequence* can be constructed adding another section of n bits to the word. The address of the previous element in the sequence is stored there. This allows navigating the sequence in reverse order. Something similar can be used to store the parent of a node in a tree. This allows navigating the tree from the bottom up. Finally, more sections of n bits can be added to each word in the tree so that trees with greater degrees can be stored. Interestingly, a tree can represent a more meaningful data structure, like a record, where each child node represents a field of the record, and the root the record itself. An even simpler representation for record is a word with several sections where each section represents a field of the record.

4. Conclusions

Here we have presented an extension of the original SDM that addresses several of its difficulties with storing compound data structures like sequences, trees and records. Our ESDM preserves the desirable, biologically inspired, properties of the original. It is also noise robust, auto-associative and distributed. These, combined with the possibility of storing sequences and other compound data structures, make ESDM an even more attractive option with which to model episodic memories.

The current ESDM implementation uses a data base for the main storage of the hard locations, and a ram cache to speed up the store and retrieve operations. This allows us to create large ESDMs, with millions of hard locations and word dimensions of the order of 10,000 bits even with modest computers. Simulations of storing and retrieving sequences and trees, together with our evaluation of the advantages of ESDM over the standard SDM, are forthcoming.

ESDM is compatible with other improvements already studied, such as the forgetting mechanism [7][8]. Including this forgetting mechanism is a natural following step for this architecture.

ESDM has the potential for further extensions. Representation of other data structures and combining it with hyperdimensional vector arithmetic are possible paths for further development.

References

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

[2] Baddeley, A., Conway, M., & Aggleton, J. (2001). *Episodic Memory*. Oxford: Oxford University Press.

[3] Franklin, S., Baars, B. J., Ramamurthy, U., & Ventura, M. (2005). The Role of Consciousness in Memory. *Brains, Minds and Media, 1*, (pp.1 - 38).

[4] Furber, S. B., Bainbridge, W.J., Cumpstey J.M. & Temple, S. (2004). A Sparse Distributed Memory based upon N-of-M Codes, *Neural Networks Vol: 17/10* (pp 1437 - 1451).

[5] Bose, J., S.B. Furber, J.L. Shapiro, A.(2005). Spiking neural sparse distributed memory implementation for learning and predicting temporal sequences. *Lecture Notes in Computer Science, Volume 3696*, (pp.115 - 120), Springer-Verlag GmbH. ISSN: 0302-9743

[6] Meng, H., Appiah, K., Hunter, A., Yue, S., Hobden, M., Priestley, N., Hobden, P. & Pettit, C. (2009). A modified sparse distributed memory model for extracting clean patterns from noisy inputs. In: *(Proceedings) International Joint Conference on Neural Networks (IJCNN)*. (pp. 2084 - 2089).

[7] Ramamurthy, U., D'Mello, S. K., & Franklin, S. (2006). Realizing Forgetting in a Modified Sparse Distributed Memory System. *Proceedings of the 28th Annual Conference of the Cognitive Science Society*, (pp. 1992 - 1997).

[8] Ramamurthy, U. & Franklin, S. (2011). Memory Systems for Cognitive Agents. *Proceedings of Human Memory for Artificial Agents Symposium at the Artificial Intelligence and Simulation of Behavior Convention (AISB'11)*, University of York, UK,, (pp. 35 - 40).

[9] D'Mello, S. K., Ramamurthy, U., & Franklin, S. (2005). Encoding and Retrieval Efficiency of Episodic Data in a Modified Sparse Distributed Memory System *Proceedings of the 27th Annual Meeting of the Cognitive Science Society.* Stresa, Italy.

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

[11] Franklin, S. (1995). *Artificial Minds*. Cambridge MA: MIT Press.

[12] Snaider, J., McCall, R., & Franklin, S. (in press). Time Production and Representation in a Conceptual and Computational Cognitive Model. *Cognitive Systems Research*.

[13] Kurby, C. A., & Zacks, J. M. (2008). Segmentation in the perception and memory of events. *Trends in Cognitive Science, 12*(2), (pp. 72 – 79).