7-17-2012

Integer Sparse Distributed Memory and Modular Composite Representation

Javier Snaider

INTEGER SPARSE DISTRIBUTED MEMORY AND MODULAR COMPOSITE REPRESENTATION

by

Javier Snaider

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

Major: Computer Science

The University of Memphis

August, 2012

Acknowledgements

I would like to thank my dissertation chair, Dr. Stan Franklin, for his unconditional support and encouragement, and for having given me the opportunity to change my life. He guided my first steps in the academic world, allowing me to work with him and his team on elucidating the profound mystery of how the mind works.

I would also like to thank the members of my committee, Dr. Vinhthuy Phan, Dr. King-Ip Lin, and Dr. Vasile Rus. Their comments and suggestions helped me to improve the content of this dissertation. I am thankful to Pentti Kanerva, who introduced the seminal ideas of my research many years ago, and for his insights and suggestions in the early stages of this work.

I am grateful to all my colleagues at the CCRG group at the University of Memphis, especially to Ryan McCall. Our meetings and discussions opened my mind to new ideas. I am greatly thankful to my friend and colleague Steve Strain for our discussions, and especially for his help editing this manuscript and patiently teaching me to write with clarity. Without his amazing job, this dissertation would hardly be intelligible.

I will always be in debt to Dr. Andrew Olney for his generous support during my years in the University of Memphis, and for being my second advisor, guiding me academically and professionally in my career. My most sincere thanks to you.

To my sons, Gaston and Adam, who have accepted having a *cybernetic* father for the last two years, and for being the source of my commitment to show them that they have to follow their dreams no matter the obstacles in the path, or the effort and energy that they demand. I love you guys.

Finally, to my wife, Ester Bruden, who has been my greatest supporter. She believed in me long before I believed in myself. Without her unconditional love, support, and encouragement I would have never started this life-changing journey. I am so fortunate to share my life with her. Ester, you raise me up to more than I can be[1]. I love you.

[1] From "You Raise Me Up" by Brendan Graham.

Abstract

Javier Snaider. Ph.D. The University of Memphis. Aug/2012. Integer Sparse Distributed Memory and Modular Composite Representation. Major Professor: Stan P. Franklin.

Challenging AI applications, such as cognitive architectures, natural language understanding, and visual object recognition share some basic operations including pattern recognition, sequence learning, clustering, and association of related data. Both the representations used and the structure of a system significantly influence which tasks and problems are most readily supported. A memory model and a representation that facilitate these basic tasks would greatly improve the performance of these challenging AI applications.

Sparse Distributed Memory (SDM), based on large binary vectors, has several desirable properties: auto-associativity, content addressability, distributed storage, robustness over noisy inputs that would facilitate the implementation of challenging AI applications. Here I introduce two variations on the original SDM, the Extended SDM and the Integer SDM, that significantly improve these desirable properties, as well as a new form of reduced description representation named MCR.

Extended SDM, which uses word vectors of larger size than address vectors, enhances its hetero-associativity, improving the storage of sequences of vectors, as well as of other data structures. A novel sequence learning mechanism is introduced, and several experiments demonstrate the capacity and sequence learning capability of this memory.

Integer SDM uses modular integer vectors rather than binary vectors, improving the representation capabilities of the memory and its noise robustness. Several

experiments show its capacity and noise robustness. Theoretical analyses of its capacity and fidelity are also presented.

A reduced description represents a whole hierarchy using a single high-dimensional vector, which can recover individual items and directly be used for complex calculations and procedures, such as making analogies. Furthermore, the hierarchy can be reconstructed from the single vector. Modular Composite Representation (MCR), a new reduced description model for the representation used in challenging AI applications, provides an attractive tradeoff between expressiveness and simplicity of operations. A theoretical analysis of its noise robustness, several experiments, and comparisons with similar models are presented.

My implementations of these memories include an object oriented version using a RAM cache, a version for distributed and multi-threading execution, and a GPU version for fast vector processing.

Table of Contents

List of Tables

Page

List of Figures

Page

Chapter 1: Introduction

Today, computers are ubiquitous. They are not only present in high technology research facilities and complex industrial process control systems, but in everyday places and situations. We have computers in our desks, our cars, and our cell phones. The processor in my cell phone is probably more powerful than the computer onboard Apollo 11, and certainly, it also has more memory capacity. Computers can perform complicated mathematical calculations at amazing speed that was unthinkable just a few years ago. The spectrum of computer applications is equally impressive. Applications cover assorted disciplines such as science, medicine, business, graphics arts, media, industry, education, military science, and so on. Most of these applications exploit the strengths of computers: computer power, memory capacity, communication speed, among others.

Despite the power and success of computers, there are tasks that computers are not yet able to perform well. For many tasks that humans perform almost effortlessly, such as object and face recognition, natural language understanding, and navigation in unknown environments, there are no efficient algorithms that perform at least as well as humans. Interestingly, the kinds of tasks that computers perform efficiently, such as math calculations, frequently challenge people when they are carried out by hand, as shown by the number of errors that people incur performing these operations. On the other hand, computers have trouble with operations that seem simple and unchallenging to humans.

Several authors (Franklin, 1995; Kanerva, 1988, 2009; Winston, 1992) have pointed out the importance of representations to perform tasks efficiently and solve problems. Winston (1992) defined the representation principle in these words: “Once a problem is described using an appropriate representation, the problem is almost solved”

(p. 18). Franklin (1995) discussed the importance of representation for both symbolic AI and connectionist models (p. 365). Kanerva (2009) pointed out how a representation can facilitate certain tasks at the expense of others. A nice example from computer science illustrates this. Usually computers represent signed integers using two's complement format. Addition and subtraction operations can be efficiently performed by the same hardware. On the other hand, Binary Coded Decimal (BCD) format represents each decimal digit of a number with its own bit sequence. BCD excels at fast and accurate translation between machine and human readable formats. However, it requires more complex algorithms and circuits for basic arithmetic operations, and its storage usage is less efficient.

The structure of a system correlates with the representation used. For example, special hardware is needed to support floating point representation efficiently. Without this special hardware, the implementation of mathematical operations will be too slow to be practical. Many Digital Signal Processors (DSPs) have fixed point arithmetic implementations that speedup processing when precision is not an issue.

Both the representation and structure of a system significantly influence which tasks and problems are most readily supported. One key factor underlying representation is the memory mechanism. The characteristics of a system's memory can give clues as to what kind of tasks the system can perform efficiently. Analyzing the features of biological memories helps to define the requirements of some applications, such as

cognitive architectures and robot navigation controllers, that face tasks and problems similar to those of biological entities[1].

Biological memories, and human memory in particular, can be categorized in numerous forms: sensory, procedural, working, declarative, episodic, semantic, long-term memory, and perhaps others (Ramamurthy & Franklin, 2011). Here I discuss properties that may fit in several of these categories.

Human memory is always learning. Although attention is an important component of learning (Kruschke, 2003; Logan, 2002), humans learn effortlessly all the time. Human memory is content addressable; for example, memory of a past event can be cued by a similar event or by partial contents of that memory. This property is called auto-associativity (see below). We can remember a place or a face almost instantly without knowing where it is stored in the memory. Human memory is able to associate related data, such as the name of a person with her face. This property, called hetero-associativity, allows the memory to retrieve some data triggered by related data. Even more important, human memory is particularly good for remembering sequences. Language, motor skills, music, and planning are examples of human activities that require one to learn, recognize and remember sequences.

The human mind handles innumerable kinds of data, including low level sensory information, such as visual or auditory information, past events, motor skills and their relationship with the context in which they are applied, highly abstract concepts, and so on. Several of these types of data, such as visual information, are unlikely to appear twice in exactly the same way. For example, when we observe a landscape or a face, there are

[1] Some features described here may be implemented by functional processes other than memory. Nevertheless, I will assume here that memory is responsible for these functionalities.

myriads of factors that affect the observation: the illumination, the angle of the observer, weather conditions, etc. Human memory is able to handle these factors and recognize the landscape or face anyway. Moreover, the human memory can combine several images into a prototypical view.

Even if there is no certainty about the capacity of the human memory, it seems that data stored in it smoothly degrades or decays. Two main theories about forgetting have been proposed: interference and decay; see for example (Altmann & Gray, 2002) for a discussion on this subject[2]. Interference between similar experiences and the decay of memory affects the recall process. We can often remember a face or a place, even if not perfectly. In contrast, when an item of data is deleted from a computer memory, it is deleted for good.

Computers also have memory modules. However, the usual functionality of computer memory differs from that of humans. First, the computer's main memory comprises an array of registers that generally store data as binary words. Each register has a position in the memory identified by its address. Reading from these memories requires knowing the address of the data that we want to read. Second, in general, there is no relationship between the data and the address where it is stored. Finally, computer memories have a predefined capacity explicitly determined by the number of records or addresses in them.

Several AI applications, such as cognitive architectures (Foundalis, 2006; Ramamurthy & Franklin, 2011), robot controllers (Jockel, 2009; Robertson & Laddaga, 2011), natural language processing, and visual recognition, have in common that they try

[2] However, some authors claim that traces in declarative memory do not decay, but some of them cannot be retrieved (Tulving, 1968).

to solve problems that are generally easy for humans, but even the most advanced algorithms today perform poorly compared with humans. These *challenging* AI applications can benefit from memory modules that share features with human memory. Additionally, the features of these memory modules offer the potential to enhance the power and simplify the implementation of such applications. Moreover, recent innovations in parallel computing (see below) may improve the efficiency of such implementations.

Challenging AI applications, such as the ones described above, must be able to perform a very wide range of tasks: object recognition, planning, action selection, reasoning, and so on. But is there a set of primitive tasks that is common to many of these more high level tasks? It is difficult to give a definitive answer to this question. However, several authors have attempted it. Ramamurthy and Franklin (2011) analyzed the different types and requirements for memories and learning mechanisms for cognitive agents. Jockel (2009) listed the desirable properties of the memory module for the controller system of cognitive, autonomous robots.

In his presentation, Robertson (2011) enumerated several insightful concepts about robot perception and navigation requirements. He defined robust pattern recognition as one of the most important low level tasks for robot navigation controllers. He pointed out that vectors of sensory input data are always noisy, and it is unlikely that exactly the same data will occur twice. Thus, clustering of several similar vectors is critical in order to recognize them as the same information. Kanerva (1988, 2009) and Jockel (2009) discussed similar ideas. Robertson also mentioned sequence learning and integration of similar sequences as important tasks for a robot controller. Other authors

identified sequence learning as a major piece of cognition (Starzyk & He, 2007; Sun & Giles, 2001). Association of related data, or Hebbian learning, is frequently mentioned as a fundamental process for both cognitive agents (Foundalis & Martinez, 2007) and robot controllers (Jockel, 2009; Robertson & Laddaga, 2011). Kanerva (1988, 1993, 2009) also described several important characteristics of representations and memory systems for cognitive agents.

Summing up, a tentative list of some of the basic operations desirable for these kinds of applications includes pattern recognition, including when partial and noisy cues are used, sequence learning, generalization, also known as clustering, and association of related data (i.e., Hebbian learning). A description of the requirements for memories and data representation that facilitate these basic operations follows[3].

Content Addressability

Biological memories are able to retrieve memories using partial or related data. For example, the smell of a baking cake might remind us of our grandmother's kitchen. This is very different than how computer memories store and retrieve data: namely, the content's address or location is required to retrieve the information. Content addressable memories, also called associative memories, come in two types: auto-associative and hetero-associative.

Auto-associativity and Hetero-associativity

Auto-associative memory associates a data item with itself. This allows recovery the data using a noisy or partial version as a cue. For example, a partial image of a person's face

[3] Some of these requirements are also described in Jockel, 2009.

suffices to recall the complete image. Auto-associativity plays a particularly important role in the processing of sensory data, where inputs are often noisy or incomplete.

In hetero-associative memory, a set of data triggers the retrieval of a related set. For example, a person's name enables recall of his face. In a more practical scenario, a robot controller application can relate an action and its context with its probable result and use this information for planning.

Robustness to Noise

Robustness to noise, intimately related with auto-associativity, allows the memory to recall stored information using noisy inputs. Sometimes memories with this property are called *cleanup* memories, because they can eliminate the noise of noisy inputs.

Applications that work with real world data, such as robot controllers, are exposed to noisy input data from sensors and proprioception from sensors monitoring actuators. Robustness to noise is a critical feature for such applications.

Generalization, Clustering, and Pattern Recognition

Clustering, which is essentially a classification problem, consists of grouping elements into a set according to a specific criterion. Individual experiences or patterns are grouped into categories based on common features. Generalization, closely related to clustering, can be defined as a distillation of the common features of the elements in a cluster. Sometimes, this process also creates a new element that represents this generalization.

Several authors consider the recognition and classification of patterns as one of the most fundamental properties of cognition (Foundalis, 2006; Hofstadter, 1995). There are many algorithms for clustering data. However, several of them are not biologically

plausible. First, they are not incremental: adding new data requires the algorithm to reexamine all previous elements. Moreover, many of them must predefine the number of clusters or groups into which to divide the data and an *oracle* that labels the training data set. Human memory seems to be able to recognize patterns, cluster them, and generalize new inputs without requiring the reprocessing of all previous inputs.

Sequence Learning

Several authors, including (Starzyk & He, 2007; Sun & Giles, 2001), consider spatial and temporal sequence learning to be one of the most important forms of learning for humans and animals: sequences are present in procedural learning, to learn new skills, high level planning, and problem solving.

For autonomous agents, time perception and representation are critical (Snaider, McCall, & Franklin, 2010, 2012). Autonomous agents able to plan and foresee the result of an action or group of actions are more likely to succeed in complex environments. The ability to estimate the duration of these actions, or to perform time related logical inferences, is also valuable. Sequence learning is a key component of these processes.

Robust sequence learning requires memory models with both auto-associative and hetero-associative characteristics. The auto-associativity allows cueing the memory with partial or noisy inputs, whereas the hetero-associativity connects one element to the next in the sequence (Lawrence, Trappenberg, & Fine, 2006).

Resilience to Memory Damage

A memory system capable of recalling information even if it suffered minor damage could be a useful feature for robots and other applications. This feature is often related to the distribution and redundancy of the data in the memory.

Autonomous robots implemented with memories possessing this feature may still work even if part of their memory is damaged. This is a critical feature for robots in distant locations, such as space exploration robots.

One of the limitations in the size of integrated circuits is the number of defects per unit of area. A memory model that is able to work even with these defects may be a good candidate for future memory hardware implementations.

One-shot Learning

The ability to learn a particular piece of information with one or few examples is called one-shot learning (Fei-Fei, Fergus, & Perona, 2006). Many connectionist models require large training data sets to learn patterns. For example, feed-forward neural networks trained with backpropagation sometimes require data sets with thousands of examples for training. On the other hand, a young child learns several categories a day using just a few examples (Tenenbaum, Kemp, Griffiths, & Goodman, 2011). Systems with one-shot learning memories tend to be more adaptive and resilient to environmental changes.

Incremental Learning

Incremental learning is the ability to learn and cluster new information without the necessity of reprocessing previously stored or classified data. Storing all the previous

data just to reprocess them when new input data appear is inefficient and most of the time infeasible. See for example (Polikar, Udpa, Udpa, & Honavar, 2001).

Forgetting, Interference, and Graceful Degradation

Forgetting would seem to be a negative feature of memories. However, it possesses significant value related to learning. Forgetting allows retaining only the most relevant or frequent elements in the memory. The two primary theories and possible mechanisms of forgetting are decay (Brown, 1958; Ebbinghaus, 1885; Peterson & Peterson, 1959) and interference (Keppel & Underwood, 1962; McGeoch, 1932). Similar events interfere with one other, affecting their retrieval. Alternatively, decay causes memory loss as a function of time (Ramamurthy, D'Mello, & Franklin, 2006; Sims & Gray, 2004). Altmann and Gray (2002) claim that decay and interference are functionally related and that the decay mechanism prevents old traces from interfering with new ones.

In unsupervised learning, a forgetting mechanism helps to eliminate incorrect data and wrong associations from the memory. For example, a wrong association is unlikely to be frequently repeated, and the forgetting mechanism will eventually discard it from the memory.

High Dimensionality

The input from sensors and the possible state of actuators of robots and cognitive agents may be represented with a high-dimensional feature or state vector. Memories and representations that directly handle these large vectors may be an advantage. However, this is not the main reason for this requirement. High dimensional spaces have properties

that help implement many of the requirements listed previously. Since high dimensionality is a critical issue for this work, the next section discusses it in more detail.

High Dimensional Vector Spaces

The neural system of humans and of some other animals has on the order of 10^{10} neurons. When the activity of neurons is recorded, even for simple mental events or tasks, a wide number of neurons are active across several regions of the brain. Even if it is not yet clear what exactly these patterns of activation represent, we can argue that these representations are distributed across a large number of neurons. On the other hand, in unary representations, each unit, or neuron, represents something by itself.

High dimensional representations have useful properties that would help in achieving the desiderata described above. In the connectionist and machine learning literature, the problem related with high dimensional spaces is known as the *curse of dimensionality*. Such spaces often involve exponential growth in the execution time of algorithms. Because the space increases so quickly, data samples become sparsely distributed, and methods based on statistical significance require an enormous amount of data to be reliable. On the other hand, Kanerva (2009) refers to high dimensionality as a blessing. The inherent noise robustness of high dimensional representations and their potential for holistic processing (see below) can actually facilitate the implementation of the desired processes and features of the system.

Kanerva used binary vectors with thousands of dimensions for his binary Spatter Code representations (1994) and Sparse Distributed Memory (1988, 2009). These vectors have a rich representation capability and are also noise robust. Plate (1995, 2003) created the Holographic Reduced Representation (HRR), a representation based on large vectors

of real numbers that also exploits the properties of high dimensional spaces. Vectors of any of these high dimensional spaces can be used to represent a complex structure, where each vector denotes an element in the structure. However, a single vector can also represent the same structure by implementing a *reduced description*, a mechanism to encode complex hierarchical structures in vectors or connectionist models (Hinton, 1990). These reduced description vectors can be expanded to obtain the whole structure, but may also be used as is for certain operations. This enables a *holistic* processing of the structure. Kanerva's Spatter Code and Plate's HRR are implementations of reduced description models.

Kanerva (2009) introduced a possible new paradigm of computing based on distributed representations named hyperdimensional computing. He described operations that can be performed using Spatter Code vectors, such as analogy-making and inference reasoning. Although he discussed hyperdimensional computing using binary vectors, the same paradigm can be extended to other reduced description models such as HRR or Modular Composite Representation, the one that will be introduced in this dissertation. Plate (2003) also demonstrated the power of HRR vectors to solve several tasks, including sequence learning and logic operations, which complement the hyperdimensional ideas. The features of these models make them good candidates for representation in cognitive architectures and other AI applications.

Several other models are based on large vectors. Developed over the last two decades, semantic space models exhibit success in many fields. Some of the more prominent models are Latent Sematic Analysis (LSA) (Deerwester, Dumais, Furnas, Landauer, & Harshman, 1990), based on statistical analysis; Random Indexing (Sahlgren,

2005), based on random sparse vectors and random permutations; and BEAGLE (Jones & Mewhort, 2007), based on HRR. For recent surveys of semantic space models see (Cohen & Widdows, 2009; Turney & Pantel, 2010). Although most of these models are based on the similarities or distances between words, some of them were extended to support other kinds of data (Jones & Mewhort, 2007; Sahlgren, 2005) .

Parallel Computing Becoming Cheap

Modern computers are based on the Von Neumann model, which dates to the 1940's. This architecture divides the computer's structure into the central processing unit, the memory, and the input-output unit. Computers are designed to perform logic and mathematics based on binary representations of numbers.

Biological brains are composed of neurons. The activation of these neurons and their interconnection play an important role in cognitive processing and memory. The highly parallel and interconnected structure of brains seems very different than the architecture of a computer. However, since its incipience, the latter has undergone innumerable improvements. Nowadays, it is common to have multi-core CPUs executing instructions in parallel. Furthermore, Graphic Processors Units (GPUs), which can perform billions of parallel vector operations per second, are often found even in mid-range computers.

Although these tendencies do not radically change the structure of computers, parallel computing and connectionist models inspired by biological brains are now more easily and more frequently implemented with highly parallel algorithms using such technologies as GPUs. Applications that could run efficiently only on high-end supercomputers a few years ago can now be executed on desktops or laptops. For

example, Leveille and colleagues (2011) have been developing MoNETA (MOdular Neural Exploring Traveling Agent), a highly parallel cognitive architecture implemented to run on GPU based systems or on future memristor technologies (Versace & Chandler, 2011).

The memristor is not the only new hardware technology that is promising for parallel implemetations. Likharev (2009) developed CMOL, a hybrid CMOS-nanoelectronic circuit, and demonstrated several neural networks implementations using this technology. Furthermore, some authors experimented with FPGA (Field-Programmable Gate Array) for hardware implementations of simple cognitive architectures (Lopez, Sanz, Moreno, Salvador, & Alarcon, 2007).

Contributions of this Work

First proposed by Kanerva (1988), sparse distributed memory (SDM) is a mathematical model of human long term memory based on large binary vectors. The previous sections have described this memory's desirable properties. It is distributed, auto-associative, content addressable, and noise robust. Moreover, it exhibits one-shot learning, is resilient to damage, and its contents degrade gracefully. It also possesses interesting psychological characteristics as well, including interference, knowing when it does not know, and the tip of the tongue effect. Furthermore, SDM's structure is ideal for parallel processing or hardware implementation.

SDM's features make it an attractive option for modeling memory modules in cognitive architectures and other challenging AI applications. The proposed variations on SDM, Extended SDM and Integer SDM, further improve its features.

Extended SDM increases the hetero-associativity feature of the memory. Data to be described herein will show that a novel mechanism using this extension is particularly effective for sequence learning.

Integer SDM extends the domain of the memory to accept integer vectors, with a range of possible values for each dimension. The benefits of this model are retained when merged with Extended SDM into a combined SDM model that uses integer vectors, has better hetero-associativity support, and improves sequence learning. These models can be further expanded, for instance with the forgetting mechanism (Ramamurthy, D'Mello et al., 2006), which would presumably improve the unsupervised learning capabilities of the memory.

Finally, a new reduced description representation, the Modular Composite Representation (MCR) is introduced in this work. Spatter Code uses binary vectors and simple operations such as bitwise XOR and arithmetic sums, but has some limitations in its representation capabilities. Data from the real world are not always Boolean, and representations using more than two values are desirable. Moreover, the sum with normalization operation required in Spatter Code may introduce excessive noise into the representation, making it brittle. Holographic Reduced Representation uses real-valued vectors, endowing it with a rich expressiveness, but it requires complex operations such as circular convolution to combine vectors. Modular Composite Representation provides a good tradeoff between representation expressiveness and simplicity of operations.

Each of these representational models requires a cleanup memory for retrieving the components of a composite vector. Integer SDM is a good option for this function in MCR.

This research aims to achieve several specific goals. In particular it produces the following contributions to computer science:

- Design and implementation of a new variation of SDM, Extended SDM, that improves the hetero-associativity and sequence learning capabilities of the memory. (Chapter 4: Extended Sparse Distributed Memory; Chapter 7: Implementations.)
- A new mechanism that allows the application of Extended SDM to the important and widely studied field of sequence storage and retrieval. I compared the sequence storage and retrieval performance of Extended SDM to the original SDM. (Chapter 4: Extended Sparse Distributed Memory.)
- Design and implementation of a second variation of SDM, Integer SDM, that expands the representation capability of the memory. Integration of Integer SDM and Extended SDM into a dual-feature model. (Chapter 5: Integer Sparse Distributed Memory; Chapter 7: Implementation.)
- Definition and empirical test of Modular Composite Representation (MCR), a new reduced description model that balances representational expressiveness and implementational simplicity. I also demonstrated the use of Integer SDM as cleanup memory for MCR. (Chapter 6: Modular Composite Representation.)
- Demonstration of the implementation feasibility of these memory models in state-of-the-art parallel and distributed technologies. (Chapter 7: Implementations.)

Structure of this Dissertation

This dissertation has the following organization. Chapter 2 introduces SDM and the required mathematical background. Chapter 3 reviews the main concepts and models of

vector representations. Chapter 4 introduces Extended SDM and several experiments. Sequence learning using Extended SDM is also covered in this chapter. Chapter 5 develops Integer SDM and its applications. Chapter 6 introduces Modular Composite Representation and several examples of its use, as well as its integration with Integer SDM. Chapter 7 describes several implementations of the technologies introduced herein. Finally, Chapter 8 suggests directions for future research, and discusses the conclusions and contributions.

Chapter 2: Sparse Distributed Memory

Many challenging AI applications including cognitive architectures, robot controllers, image and speech recognition, and several others have memory requirements that are not well fulfilled by conventional memory models. Not surprisingly these same characteristics are also found in biological memories. All these applications require recollection of previous memories from current data, percepts, or information. This is not different from many other applications in computer science and software engineering, but what make these applications special is that the current data are not exactly the same as the stored data in the memory. A useful way to see this situation is considering the new data as a noisy version of the old data. The memory has to be able to retrieve the stored data using noisy cues. Along the same lines, it would be desirable if the memory were associative and content addressable. That is, it should be capable of retrieving stored data based on the same information, or part of it. This is different from conventional memories, where the data are retrieved by knowing their address in the memory. Another very important feature of the memory is the capability of recalling sequences based on a few of its elements. For example, humans can remember a melody using a few notes as a cue. Moreover, notice that the cue for the sequence may correspond to an inner part of it, and even then the memory should be capable of retrieving the sequence from that point to the end[1]. It is not surprising that humans and other animals have memories that exhibit these same properties. In summary, a desirable model of memory for challenging AI applications should be auto-associative, content addressable, noise robust, and able to

[1] It is also possible that cueing with an inner part of the sequence might retrieve the sequence from the beginning, as in the melody example, but this is a different mechanism that I am not going to discuss here.

store and recall sequences. For a complete analysis of the desirable properties of these memories, see Chapter 1.

Sparse distributed memory (SDM) is a mathematical model of human long-term memory based on large binary vectors (Kanerva, 1988, 1993). This memory has several desirable properties. It is distributed, auto-associative, content addressable, and noise robust. Furthermore, it presents interesting psychological characteristics (e.g., interference, knowing when it does not know, and the tip of the tongue effect), that make it an attractive option with which to model episodic memory (Baddeley, Conway, & Aggleton, 2001; Franklin, Baars, Ramamurthy, & Ventura, 2005). SDM can also store sequences of vectors as described by Kanerva (1988, 1993); moreover, the extension explained in Chapter 4 is particularly well suited to store sequences and produces even better results in this task than the original SDM.

The main idea behind SDM is based on the correspondence of the *distance* between concepts in the human mind and the distance between vectors in a high-dimensional space, that is, vectors with hundreds or thousands of dimensions. The idea of distance between concepts is not new; actually several sematic spaces use this same idea, such as Latent Sematic Analysis (LSA) (Deerwester et al., 1990), based on statistical analysis, Random Indexing (Sahlgren, 2005), and BEAGLE (Jones & Mewhort, 2007). Here we use the distance between concepts in a slightly different way, but conceptually, it is the same idea. Kanerva defines *point of interest* as a general term for concepts, percepts, events and other similar entities of the mind. The distance between concepts can be extended and applied in a more general way to any kind of point of interest. Thus, distances between events, or percepts are also possible.

There are diverse ways to represent points of interest, for instance, using nodes and links in graphs, or data structures such as records. However, particularly for this work, points of interest may be represented by vectors in a high-dimensional space. An interesting property of vectors (also known as points) in a high-dimensional space is that each point is far away from almost any other point in the space. This implies that two randomly chosen points of the space are likely to be far away from each other. Points of interest that are unrelated will be represented by distant vectors in the space; any vector in the space that represents a point of interest is far away from other points of interest. Moreover if we slightly alter the vector, it will still be closer to the original vector than to any other point of interest. Thus, the representation of a point of interest does not need to be an exact vector or point in the space. Noisy versions of this vector can represent the same point of interest and they still will be far away from other points of interest. This makes the representation noise robust, one of the most important qualities of SDM. This representation can also be interpreted as a *halo* that surrounds each point of interest. Any vector in this halo is also a representation of the point of interest. For example, if the memory is used to recall a previous event or concept stored in the memory, the new stimulus or cue does not need to be exactly the same as the original one, which is a common scenario in robotics or visual recognition.

The original SDM developed by Kanerva uses high-dimensional binary vectors with 1,000 or more dimensions. This space exhibits the important properties of high-dimensional spaces described here. These vectors are used both as addresses of the memory and also as words, the data stored in the memory. Normally, SDM is used as an auto-associative memory, thus the address vector is the same as the word vector (but see

Chapter 4). In this case, after writing a word in the memory, the vector can be retrieved using partial or noisy data.

The rest of this chapter describes SDM in detail. First some required mathematical background is explained. Then the structure and functionality of the memory is delineated. The following section analyzes the fidelity and capacity of the memory. The final two sections compare SDM with other memory models and describe several applications that use SDM.

Mathematical Background

This section describes the fundamental mathematical structure behind Sparse Distributed Memory: the binary space $\mathbb{Z}_2^n = \{0,1\}^n$. This space is composed of n-dimensional binary vectors, that is, n-tuples of zeros and ones. For example, [1,1,0,1,0,0] represents a vector of $\mathbb{Z}_2^6$.

Depending on the context, these tuples can also be called points, patterns, addresses, or words. In this dissertation, a vector of $\{0, 1\}^n$ any of these terms may be used interchangeably according to the context. For a space with n dimensions, the number of vectors is given by $N = 2^n$. For example, with $n = 1$, the space comprises {[0], [1]} and therefore, $N = 2$.With $n = 2$ the space is composed of {[0,0], [0,1], [1,0], [1,1]}, giving $N = 4$. Kanerva represents the space itself also with N. For notational simplicity, I will follow the same convention here. The points of N can be geometrically visualized as the vertices of a hypercube of n dimensions which has its sides of length equal to 1.

It is important to notice that vectors of these spaces do not necessarily have any particular order. They are just vectors, not binary numbers. The properties of the vectors

required for SDM emerges from the distribution of their *distances* (see below), not from their binary number representation.

A summary of the main concepts of the space $\{0, 1\}^n$ follows; for a full description of the space see (Blumenthal & Menger, 1970; Kanerva, 1988). For the examples in the following paragraphs let us assume $n = 6$, $x = [1,0,0,1,1,0]$ and $y = [1,1,0,0,0,0]$.

Origin ***0***

The point with zero in every coordinate: $\boldsymbol{0} = [0,0,0,\ldots,0,0]$

Complement ***`x***

The complement of a vector x is the vector that has zeros where x has ones and vice versa. For example, $`x = [0,1,1,0,0,1]$

Norm $|\boldsymbol{x}|$

The norm of a binary vector is the number of ones that the vector has. For example, $|x| = 3$ and $|y| = 2$.

Difference ***x - y***

The difference of two vectors x and y is another vector that has ones in the dimensions where x and y differ and zeros in the dimensions where they agree. This operation is equivalent to the bitwise exclusive or (*XOR*) between x and y.

The difference is commutative in this space: $x - y = y - x$. In the example, $x - y = [0,1,0,1,1,0]$

Distance ***d(x, y)***

There are several distances that can be used in this space. The most common one, and also the one used in SDM, is the Hamming distance. The Hamming distance between *x* and *y* is the number of dimensions by which *x* and *y* differ. This is equivalent to the norm of the difference between *x* and *y*: $d(x, y) = |x - y|$. Moreover, since $(x - `x)$ is equal to the vector with all ones, $`x$ is the farthest point from *x* in the space.

The distance can be used as a similarity measure; two vectors of *N* are *similar* if they are close enough. Of course, this definition is relative, and this term in general is used in relation to other vectors; for example, if *x* and *y* are vectors, *S* is a set of vectors, and $y \in S$, we can say: "vector *x* is the most similar to *y* in *S*."

To implement SDM several similarity measures can be used, including other distances such as the Euclidean one. For the following discussion, if no other measure is explicitly indicated, wherever the term "distance" is used, the Hamming distance is assumed.

In the example, $d(x, y) = |x - y| = |[0,1,0,1,1,0]| = 3$

Betweenness ***x : y : z***

Point *y* is between *x* and *y* if and only if $d(x, z) = d(x, y) + d(y, z)$.

Using Hamming distance, any dimension *i* of *y* must be equal to the same dimension of *x* or *z*: if $x : y : z$ then $y_i = x_i$ or $y_i = z_i$

Based on this, it is easy to shown that the entire space *N* is between *x* and $`x$. In the example, there are several points between *x* and *y*. Al points *z* that follows the pattern [1,*,0,*,*,0] , where * can be either 0 or 1, are between *x* and *y*. (e.g., x : [1,1,0,0,1,0] : y.)

Orthogonality $x \perp y$

Two vectors are orthogonal, or indifferent, if and only if the distance between them is half of the number of dimensions: $d(x, y) = n/2$.

This property is commutative, if $x \perp y$ then $y \perp x$. It is easy to see that if a vector x is orthogonal to another vector y, x is also orthogonal to $\grave{y}$. If x is orthogonal to y, then x has exactly half of its dimensions equal to y. Therefore, the other half of the dimensions of x are equal to $\grave{y}$. Then $x \perp \grave{y}$.

Kanerva defines the *indifference distance* of the space $\{0, 1\}^n$ to be $n/2$. In the example, the indifference distance is 3 and $x \perp [1,1,0,0,1,1]$.

Sphere $O(r, x)$

A sphere[2] of radius r and center x is the set of points of N that are at most a distance r from x.

$O(r, x) = \{y \mid d(y, x) \leq r\}$. Spheres with radius n enclose the entire space N. For example, $O(1, x)$ = { [0,0,0,1,1,0], [1,1,0,1,1,0], [1,0,1,1,1,0], [1,0,0,0,1,0], [1,0,0,1,0,0], [1,0,0,1,1,1]}.

I already mentioned that N can be represented as the vertices of a hypercube of n dimensions. The distance between two points is the length of the shortest path across the edges of the hypercube that connects the corresponding vertices to these two points.

Kanerva (1988) defines a space (any metric space, not just binary spaces) as spherical if (1) each point x of the space has exactly one opposite $\grave{x}$, (2) all points of the

[2] Kanerva actually used circle for this concept. However, as we shall see later, sphere is a better name here.

space are between any point *x* and its opposite `*x*, and (3) each point in the space is isometrically equivalent to any other point; that is, for any two points *x* and *y* there exists a distance preserving transformation that maps *x* to *y*. The surface of a sphere is clearly a spherical space, as is *N*.

Based on this definition, Kanerva suggested the *sphere analogy*. Since *N* is spherical, the space is analogous to a three dimensional sphere with diameter 2*n*. The points *x* and `*x* are in the *poles* of this sphere (any point of the space can be *x*), the entire space lies between *x* and `*x*, and most of the space is in the *equator* (see Figure 1).

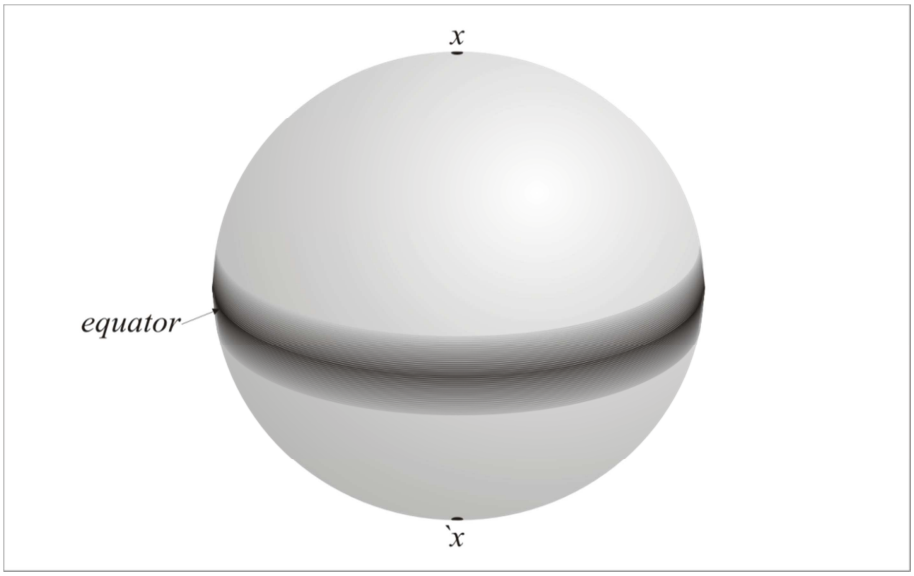


Figure 1. The sphere analogy. The space *N* is analogous to the surface of a 3-dimsional sphere. For any point *x*, most of the points in *N* are near of the equator, which is half way between *x* and `*x*. Adapted from (Jockel, 2009).

A circle on the surface of the 3-dimensional sphere with center at *x* is analogous to a sphere in *N*. The analogy is far from perfect: *N* has a discrete number of elements and the surface of the sphere is continuous, the minimal path between two points in *N* are not

unique, and a sphere in *N* is in general not convex. Nevertheless, the analogy is excellent for illustrating several properties of the space (Kanerva, 1988).

A very important property of *N* is the distribution of the distances from a randomly chosen point to the rest of the points of the space. Since *N* is spherical according to the definition above, any point could be in the origin (or translated to it), so I will consider the distances from the origin. Kanerva (1988) proved that these distances follow a binomial distribution, that can be approximated by a Normal distribution with mean distance equals to $n/2$ and standard deviation approximately equals to $\sqrt{n}/2$. Figure 2 summarizes this distribution for different values of *n*. It is easy to see that half of the space is closer than $n/2$ and the other half is farther than that distance. But it is counterintuitive that as the number of dimensions *n* increases, the distribution tends to highly concentrate the points at about the indifference distance $n/2$. For example, for $n = 1{,}000$, the mean distance is 500 and the standard deviation (*SD*) is about 15.8.

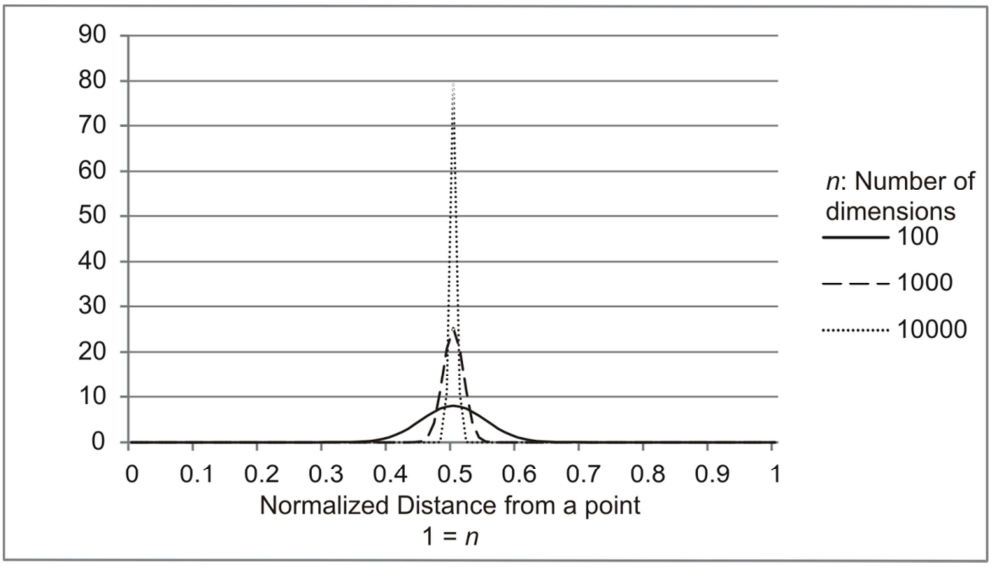


Figure 2. Distribution of Hamming distances in *N*. As the number of dimensions *n* increases, the distribution tends to highly concentrate the points at about the indifference distance $n/2$. Adapted from (Kanerva, 1988).

According to the Normal distribution, only one millionth of the space is closer than 422 bits or farther than 578 bits, since 5 *SD* is about 78 bits. Notice that points do not concentrate or cluster in the space, all points are isometrically equivalent, and the distances from any point to the rest of the space are concentrated at almost the indifference distance.

Randomly selected points of the space can represent unrelated points of interest, and due to the large size of the space, it is almost impossible to run out of vectors. Because of the distribution and the symmetry of the space, any two randomly chosen points will likely be almost at the indifference distance from each other, that is, they are almost orthogonal to each other. Kanerva named this remarkable property the *tendency to orthogonality* of the space.

Kanerva (2009) described another interesting example. Suppose we have two vectors A and B that only differ in 25% of their bits. This is unlikely to happen by chance, but they can be constructed in this way to represent related concepts (see Chapters 3 and 6). Based on A, we can create another vector C by changing 1/3 of the bits of A. C is just a noisy version A. One might think that C could become closer to B than to A, but this is very unlikely. If $d = 1/4$ and $e = 1/3$, then the distance between A and B is $d(A,B) = dn$, and the expected distance between C and B it is $d(C,B) = (d + e - 2de)n$. Thus, $d(C,B) = d(A,B) + (1 - 2d)en$. It is clear that the distance between C and B also increases. With $n = 1{,}000$, $d(A,C) = 333$ and $d(C,B) = 416$. The difference is more than 5 *SD*. If the dimensionality of the space is higher the effect is even more pronounced: with $n = 10{,}000$ the distance $d(A,C) = 3{,}333$ and $d(C,B) = 4{,}166$. In this case the difference is more than 16 *SD*.

These properties of high dimensional spaces are the basis of Sparse Distributed Memory. The following section describes the structure and functionality of SDM.

Memory Description

Here I present an introduction to SDM. Both leisurely descriptions (Franklin, 1995) and highly detailed descriptions (Kanerva, 1988, 1993) are available.

Conventional computer memories are accessed using the location, or address, of the data. A memory of this kind is just an array of fixed size registers; each register holds a word of the memory and the size of the register is called the word size. Each register is indexed by its address, and has a size that is known as the address size. In general, there is no relation between the address and the word stored at that particular register. Conversely, in SDM, a content addressable random access memory, the data in the memory are retrieved using the same content, or part of it, as the cue. Several authors, including Hawkins (2005), believe this is a fundamental characteristic of the human memory. In this kind of memory, called *associative memory*, instead of using a fixed, uninformative address to store the data, a meaningful vector is used as the address. In a special case of associative memory, called *auto-associative*, a data word stored in the memory is associated with itself. In other words, the data is stored using itself as an address. This can seem useless, but is actually quite convenient because it allows a word stored in the memory to be retrieved using an approximate or noisy version of itself (Kanerva, 2009). See Chapter 1 for more discussion about this subject.

We can imagine a conventional memory with an address size equal to its word size and use the memory as an associative memory. A problem arises with large word or the address sizes, such as the sizes described in the previous section. For example with

n = 1,000, Franklin (1995) compared the size of this memory with the number of atoms in the universe. It is evident that such a memory cannot be constructed. Moreover, even if it were possible to construct, the auto-associative characteristic could not be easily implemented. Nevertheless, high dimensional vectors are an attractive option to model concepts, events, and other similar entities, and SDM nicely addresses these problems.

SDM is built upon the properties of high dimensional spaces described on the previous sections. Here I will use high dimensional binary spaces in the order of 1,000, or 10,000 dimensions. Both addresses and words are binary vectors whose length equals the number of dimensions of the space. As an example, I will use binary vectors of 1,000 dimensions.

To calculate distances between two vectors in this space, the Hamming distance is used. As explained in the previous section, the distances from a point in the space to any other point are highly concentrated around half of the maximum distance. In our example, more than 99.9999% of the vectors lie at a distance between 422 and 578 from a given vector of the space.

Hard Locations

Since it is impossible to construct a memory with such huge address space, SDM is built with *hard locations*, the units of storage of the memory. Only hard locations can store data, and each hard location has a fixed address. A sparse uniformly distributed sample of all possible addresses of the space, on the order of 2^{20} of them, is chosen. This sample constitutes the addresses of the hard locations. The proportion of hard locations over the number of possible addresses of the space is very small, in the example on the order of 2^{-980}, the reason that the memory is called *sparse*. The number of addresses selected to

construct the memory is denoted by *m*. Hard locations are like islands in the vector space. As in the ocean, islands are just a tiny proportion of the entire surface of the ocean. Data storage is only possible in these islands.

To store data, each hard location has *counters*, one for each dimension. I denote as c_i the counter corresponding to dimension *i*. In the example, each hard location has 1,000 counters. A counter is just an integer register that can be incremented or decremented in steps of size one. According to a proof by Kanerva (1988), a range of -40 to 40 provides enough capacity for a SDM with 1,000,000 hard locations, as in this example. For other sizes this range may vary.

Each hard location can store several words but as a combination rather than distinct entities. The reconstruction of one of these words requires the participation of many hard locations in its storage and retrieval. For writing in an arbitrary address in SDM, the word is stored in several hard locations. This is radically different than the way a conventional memory works, where words are stored just in one location. To read from an arbitrary address in SDM, the output vector is a composite of the readings of several hard locations. This distributed storage is what makes SDM noise robust. The process of selecting which hard locations participate in a single reading or writing operation is called the *activation* of hard locations. An *activated hard location* is one that participates in a reading or writing operation. Kanerva (1988) uses the access sphere to determine which hard locations are active for a read or write operation (see below for details). Different activation mechanisms produce interesting variations on the original SDM. Several of these alternate activation mechanisms will be explored later in this chapter.

Writing and Reading Hard Locations

In order to understand how to read and write vectors in SDM, first it is necessary to know how to read and write a vector in a hard location. To write a word vector w in a hard location, for each dimension i, if the bit w_i of this dimension i in the word is 1, the corresponding counter c_i of that hard location is incremented. If it is 0, the counter is decremented. For example, if the word w = [1,0,0,1,0] is stored in a hard location, the first counter c_0 is incremented, c_1 is decremented, c_2 is decremented, and so on.

To read a word vector from a hard location, we compute a vector such that, for each dimension i, if the corresponding counter c_i in the hard location is positive, 1 is assigned to dimension i in the vector being read, otherwise 0 is assigned. For example, if the counters C of a hard location have the values [10,-5,11,-7,-8] the output word w is [1,0,1,0,0]. The chance that a word datum is exactly the address of a hard location is almost zero. However, words are written to their nearest hard locations. Next section explains how these hard locations are chosen and how the distributed storage takes place.

SDM Storage

Since a hard location stores words as a combination of all the stored words in it, reading it returns this combination that would be different than any of the stored words. SDM addresses this problem by reconstructing the original word using information from several hard locations.

To determine which hard locations are used to read or write, an *access sphere* is defined. The access sphere for an address vector is a sphere *O* with center at this address. The radius of the access sphere is defined in such a way that on average it encloses a small proportion *p* of the total number of hard locations. If *m* is the number of hard locations in the memory, the access sphere encloses *pm* hard locations. This value *p* is also the *probability of activation* of one hard location, that is, the probability that one hard location is in the access sphere of one particular reading or writing operation. Thus, the probability *p* determines unequivocally the radius of the access sphere. For example, for a SDM with 1,000 dimensions, and a probability of activation $p = 0.1\%$, the radius of the access sphere is 451. The access sphere will contain any hard location whose address is less than 451 away from the address vector. (See Figure 3.)

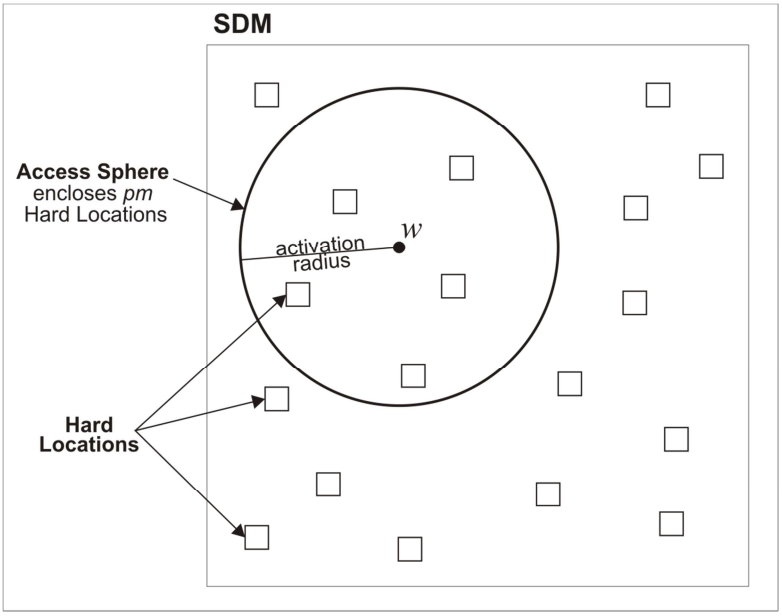


Figure 3. Access Sphere. Adapted from (Kanerva, 1993).

The activation of the hard locations can be achieved using other strategies; some of them are explored in following sections. To write a word vector *w* in any address of

the memory, the word is written to all hard locations inside the access sphere of the address *a*. Figure 4 shows the entire process.

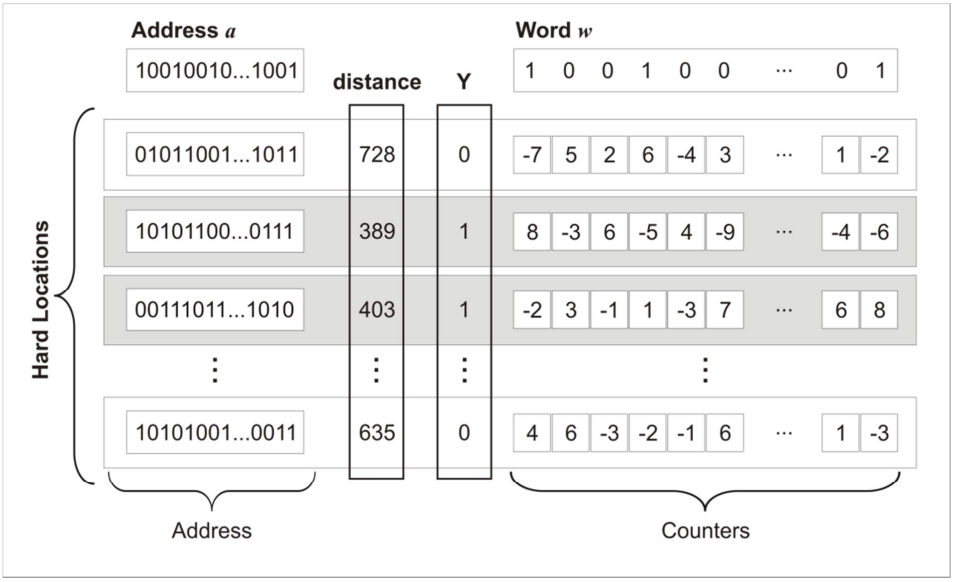


Figure 4. Writing hard locations. First the distance from the address vector *a* to each hard location's address is computed. Each dimension *j* of the vector Y is equal to 1 if the hard location *j* is into the access sphere of address *a*. The counters of activated hard locations (gray rows) are updated. If w_i is 0, the counter *i* of each active hard location is decremented. If w_i is 1, these counters are incremented.

First the distance from the address vector *a* to each hard location's address is computed. The activation vector *Y* is a binary vector of *m* dimensions, one for each hard location in the memory. The value of each dimension *j* is equal to 1 if the distance from *a* to the corresponding hard location *j* is less than the activation radius *r*, $d(a, hd_j) \leq r$. It is 0 otherwise. Finally, the word *w* is written to all activated hard locations, updating their counters.

SDM Retrieval

Reading SDM from any address consist of reading from all hard locations in the access sphere of the address vector, and combining them using a majority rule for each dimension. In other words, the output vector will have, in each dimension, a value equal to1 if the majority of the vectors read from the hard locations in the access sphere have a 1 in that dimension, and a value of 0 otherwise. An alternate procedure achieves a better result. By summing up the counters for each dimension of all hard locations in the access sphere, and then normalizing these sums using the mechanism explained above for reading a single hard location, one can produce the output vector without requiring the normalization of the readings of each hard location individually.

In general, SDM is used as an auto-associative memory, where the address vector is the same as the word vector, enabling the retrieval of a word from the memory using partial or noisy data as a cue. Suppose a vector *v'*, a partial or noisy version of a vector *v* stored in the memory, lies within a critical distance of *v* (see next section). If *v'* is used as address with which to cue the memory, the output vector, *v''*, will be closer to *v* than *v'*. If the process is repeated, using the vector *v''* as an address, the new reading will be even closer to *v*. After a few iterations, typically fewer than 10, the readings converge to the original vector. If the vector *v'* is farther away than the critical distance, the successive readings from the iterations will diverge. If the vector *v'* is about at the critical distance from *v*, the iterations yield vectors that are typically at the same critical distance from the vector *v*. This behavior mimics the "tip of the tongue" effect (Franklin, 1995). Figure 5 depicts the critical distance idea.

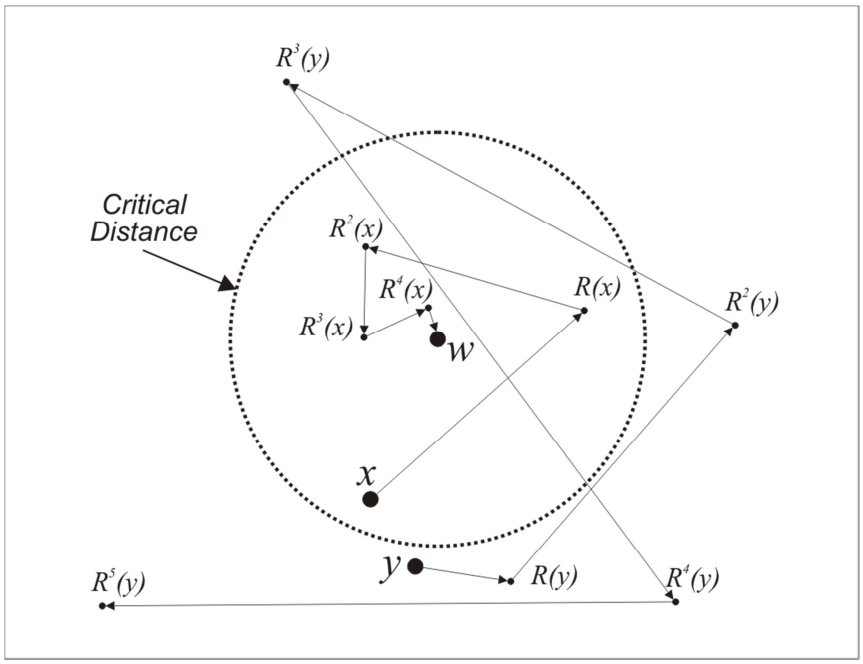


Figure 5. Critical distance. Convergence and divergence in iterative readings. Starting from *x*, which is within the critical distance, the stored word *w* is finally read. Starting from *y*, the sequence of readings diverges. $R^n(x)$ denotes the *n*-th in the sequence of readings. Redrawn from (Kanerva, 1988, p. 70).

Critical Distance, Fidelity, and Memory Capacity

Kanerva (1988) defined the critical distance as the distance beyond which divergence is more likely than convergence when reading SDM. It depends on the number of vectors already stored in the memory and on the number of hard locations that comprise the memory. He derived the expression for the critical distance as a function of the number of hard locations and the number of stored words. For example, a memory with one million hard locations, 10,000 stored words and an n = 1,000, has a critical distance of about 209.

Another important concept is the fidelity *P* that is the probability of correctly retrieving a bit of the output word. The memory fidelity is then the *n-th* power of *P*.

The memory capacity is defined as the number of stored words *T* for which the critical distance is zero. At this point, it is not possible to retrieve the stored words, even using the same word as the address. Kanerva calculated the SDM capacity (1988, 1993) by setting the memory fidelity to 0.5 and solving for *T*:

$$Capacity = \frac{m}{\left(\Phi^{-1}\left(2^{-\frac{1}{n}}\right)\right)^2} \quad (1)$$

where Φ is the normal distribution function and *m* the number of hard locations. For example, with $n = 1{,}000$ the capacity is approximately equals to $m / 10$, that is 100,000 words.

Other authors studied the capacity of SDM. Jaeckel (1989a) developed an approximate analysis that was also used also by Kanerva (1993). The most complete analysis of SDM's capacity was performed by Chou (1989). He derived the exact capacity of the memory in the general case. Keeler (1988) used Shannon's information capacity (Shannon & Weaver, 1949). In this theoretical framework, the capacity can be allocated to store more words or to tolerate more noise in the cues. He developed a mathematical model of the memory that helps to analyze the memory. A simple generalization of this mathematical model includes the binary Hopfield network (Hopfield, 1982) as a special case. Keller used this model to compare the capacity of both memories. He showed that both memories have the same capacity per storage element or counter. However, SDM presents an interesting advantage over Hopfield nets. In the former, the size of the words is independent of the number of storage elements; conversely, in the Hopfield nets the size of the words determines the capacity of the

memory. Doubling the number of hard locations in SDM doubles the capacity of the memory, independently of the dimensionality of the vectors.

Storing Sequences in SDM

When storing sequences of vectors in 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 output vector is the second element in the sequence, which is now used in turn as an address to read the memory again in order to retrieve the third element. This procedure is repeated until the whole sequence is retrieved. This mechanism uses the memory in a hetero-associative way, where the output is not necessarily similar to the cue vector. Kanerva (1988, 1993) showed that this procedure converges to the elements of the sequence. The problem with this mechanism for storing sequences is that it is not possible to use iterations to retrieve elements of the sequence from noisy input cues, yielding a far less robust memory. Another problem arises when the stored sequences have common elements, as in ABCD and FGCH. In the example, if the two sequences are stored with the described mechanism, cueing with the vector C will probably return an incorrect vector. Kanerva proposed the use of multiple *folds* to store sequences. Each fold is an entire SDM that stores the sequence of the k^{th} element ahead. That is, the next element is stored in the $fold_1$ with the current element as the address. The element two steps ahead is stored in $fold_2$ by using the current element as the address. The element *k* steps ahead is stored in that address in $fold_k$. The readings of all folds are combined to predict the next element. Jockel (2009) uses this procedure to store sequences for a robotic arm manipulation system. (See the following sections for details.) This procedure is clumsy, difficult to implement and wastes memory resources.

Kanerva (2009) proposed a better solution using hyperdimensional arithmetic, but some limitations and problems remain. In Chapter 4, I will discuss this problem in detail and introduce Extended Sparse Distributed Memory that addresses this issue with better results.

SDM Compared with Other Models

Matrix Notation of SDM

SDM can be described in terms of matrices and vector operations. For details see (Kanerva, 1993). This representation is useful for comparing the memory with correlation matrix memories, such as the Hopfield net (Hopfield, 1982) or Willshaw memories (Willshaw, 1981).

Figure 6 depicts the realization of SDM using matrices. The m x n matrix A in the left contains the address of one hard location in each row. The vector d, of size m, contains the distances from the cue vector x to each hard location address. The vector y, of size m, is the activation vector. If $d_i < r$, the activation radius, then y_i is 1, and 0 otherwise.

C is an n x m matrix that contains the counters of one hard location in each row. In order to write to the memory, the input vector is used to update the rows of the matrix C that correspond to the active hard locations. For reading from the memory, the vector s, of size n, has the sum of the counters corresponding to the rows in C, for the activated hard locations. This vector can be calculated as $s = C^{\mathrm{T}}y$. Finally, the binary vector z, the output vector, will have in dimension i a value 1 if $s_i > 0$ and 0 if $s_i < 0$. If $s_i = 0$, the output value is chosen randomly.

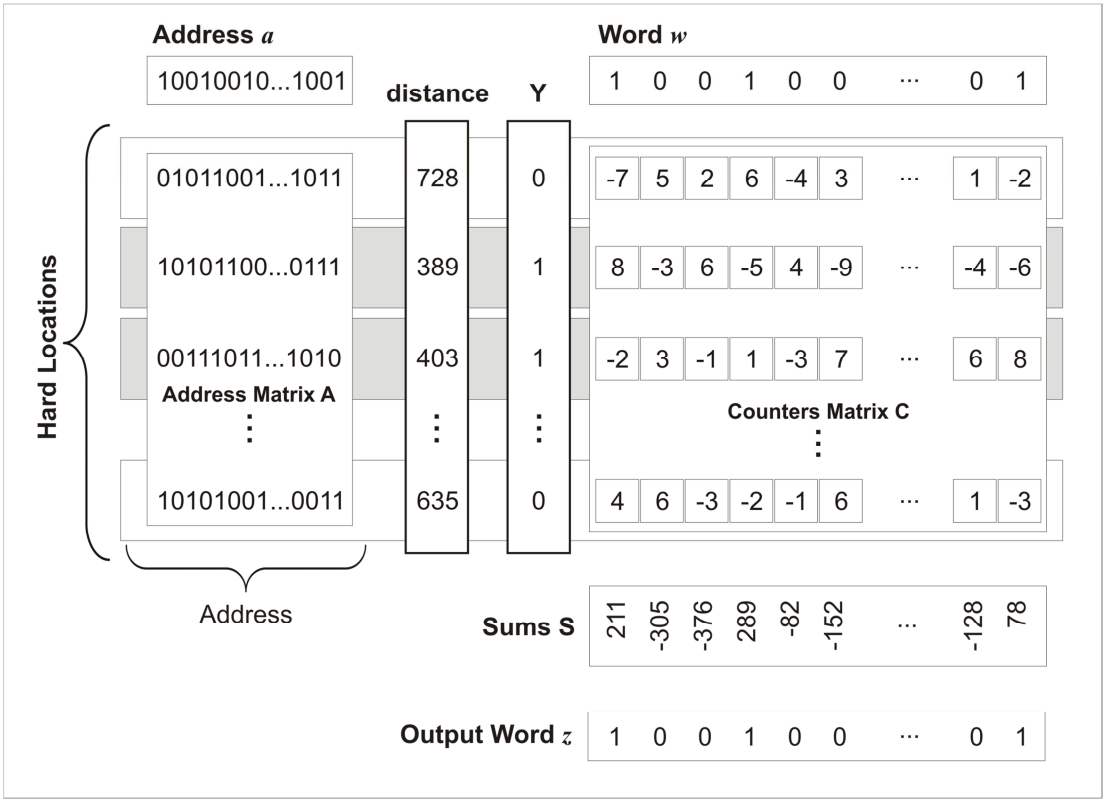


Figure 6. Realization of SDM using matrices. Redrawn from (Kanerva, 1993).

Artificial Neural Network

Some artificial neural networks (ANNs) exhibit characteristics similar to SDM, such as noise robustness, associativity, and so on. Kanerva described how SDM can be interpreted as a synchronous, fully connected, three layered feedforward artificial neural network. For details see (Kanerva, 1993). This interpretation is useful for comparing an SDM to a feedforward network. However, it is important to notice that an SDM has a completely different architecture and behavior than a feedforward ANN. In this view, the input layer is just the input vector *x*. The hidden layer corresponds to a vector *y* of size *m* that represents the active hard locations. The matrix *A* formed from the hard locations'

addresses corresponds to the matrix of synaptic weights between the input and hidden layers. The output layer is the output vector *z*. Finally, the matrix of synaptic weights between the hidden and output layers is determined by the matrix *C* of the counters of the hard locations. Figure 7 depicts this interpretation.

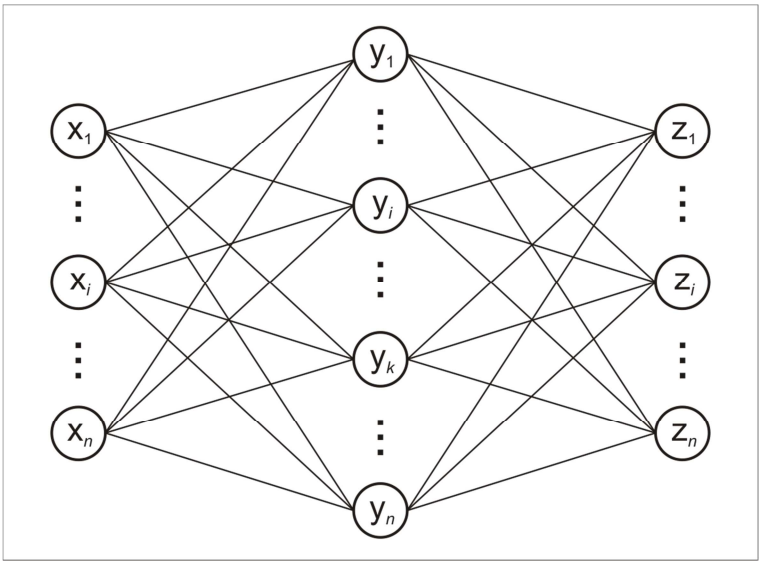


Figure 7. Description of SDM as an artificial neural network. The input layer X, is the input vector. The hidden layer is the activation vector *y* and the output layer is the output vector. The connections between X and Y are given by the hard location address. The connections between the hidden layer and the output layer are determined by the hard locations' counters. Redrawn from (Kanerva, 1993).

However, if we compare a three layer feedforward neural network trained with backpropagation and a SDM, they have several differences: first, SDM has the matrix *A* of synapses fixed and the matrix *C* allows only integer values. A feedforward network uses real values for the synaptic weights. Second, the activation function of the hidden units is completely different from the activation of hard locations. In SDM the hard locations are activated with a non-linear function and they only can take values 0 or 1. In back propagation networks, linear combinations of the inputs are used to activate the

hidden units and they can take real values. Finally, due to the mechanism and characteristics of SDM, its training is faster, compared to backpropagation trained networks. Even learning with just one or few repetitions is possible using SDM (Kanerva, 1993). On the other hand, a network trained by back propagation requires a large training set to learn.

Model of the Cerebellum

The functionality and features of SDM make this memory a good candidate to model episodic memory (Baddeley et al., 2001; Franklin et al., 2005). However, Kanerva partially modeled SDM after the structure of cortex of the cerebellum. I briefly compare them here; for details see (Kanerva, 1988, 1993). The main types of cells in the cerebellar cortex and its whole structure can be interpreted as parts of the SDM functionality.

Figure 8 shows a schematic view of the cerebellar cortex. There are two main types of inputs. The climbing fibers (Cl), which receive the signals from neurons in the brain stem, would have the same functionality as the word data input in SDM. The other kinds of inputs are the mossy fibers (Mo), which would have the same functionality as the address input in SDM. The granule cells (Gr), which receive inputs from the mossy fibers, would be equivalent to the hidden units in the SDM and work as address decoders. The Golgi cells (Go) could control the number of granule cells that fires at the same time, and could be interpreted as the control of activation of hard locations in SDM.

The axons of Purkinje cells (Pu) are the outputs of the model, and the synapses between the granule and the Purkinje cells would represent the counters of hard locations.

The comparison is far from perfect (Kanerva, 1993), but the similarities suggest that the cerebellar cortex can be interpreted as an associative memory and SDM is a plausible model of it.

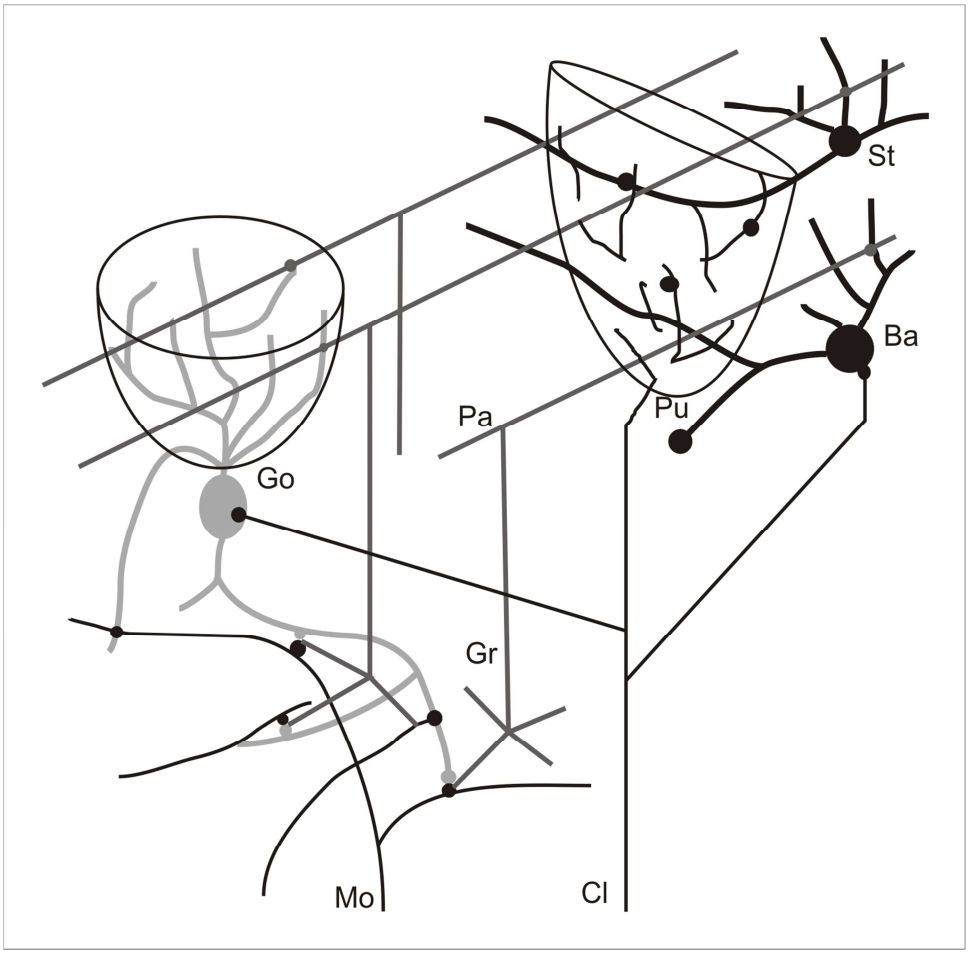


Figure 8. Schematic view of cerebellar cortex. Redrawn from (Kanerva, 1993).

Both Marr (1969) and Albus (1971) developed mathematical models of the cerebellum. Albus developed CMAC, Cerebellar Model Arithmetic Computer (Albus, 1981). CMAC is a sparse coarsely-coded associative memory algorithm designed to provide motor control for robotic manipulators. Both Marr's model and Abus' CMAC are

similar to SDM. Kanerva (1993) extensively compared SDM with these two models, showing that CMAC can be represented as a special case of the Jaeckel's hyper plane design (see next section for details).

Extensions and Improvements

Several authors have proposed different extensions and variations of SDM. In this section I will discuss some of the most influential ones. One of the critical steps in SDM's algorithm is the activation of hard locations. Many of the extensions described here address this issue. Others explore variations in the distribution of the hard location addresses in the space. Data in real applications are often not uniformly distributed, tending to cluster, which diminishes the performance of the memory. In these situations some hard locations may not be activated at all, resulting in wasting of their capacity. Other hard locations may be activated very frequently and again are wasted because their contents represent mostly noise. Most of the extensions discussed here address one or both of these issues.

Jaeckel's Selected-Coordinate Design

Jaeckel (1989a) introduced the selected-coordinate design as an alternate mechanism to activate hard locations. The rest of the model is exactly the same as in the original SDM. In this model, for each hard location a small number *k* of dimensions are randomly chosen, each being randomly assigned a value of zero or one with equal probability. For example, for an address space of 1,000 dimensions, 10 dimensions are chosen. A hard location is activated if only if the address to read or write matches all the *k* selected

dimensions values. The probability of activation p of one hard location is then $(0.5)^k$. In the example, it is approximately 0.001.

Jaeckel (1989a) showed that the capacity and fidelity of this model are slightly better than the original SDM. Another advantage over the original is the simplicity of the calculation of the activation. A hardware implementation using this model is simpler than the one corresponding to the original SDM.

Karlsson (1995) proposed a variation of Jaeckel's design restricting the selection of the selected dimensions for hard locations. With this simple change and the use of a lookup table he was able to speed up the process of activation of hard locations by several orders of magnitude.

Jaeckel's Hyperplane Design

In this second variation Jaeckel (1989b) dealt with skewed data, which are data with few ones. In this case fewer dimensions are selected, for example three, and all them must be one in the address to read or write in order to activate the hard location. By choosing the parameter k according to the proportion of ones in the data it is possible to achieve better results.

He also suggested *intermediate designs*. In these models only a fraction r of the selected dimensions need to match the address to read or write. By carefully choosing k and r depending on the number of ones in the data, it is possible to obtain a reasonable value for the activation probability p. Jaeckel (1989b) showed that the original SDM corresponds to one end of these intermediate designs and the selected-coordinate design corresponding to the other.

Dynamic Allocation

Several authors suggested allocating the hard locations using different distributions. Keller (1988) suggested choosing the addresses of the hard locations following the same distribution as the data. Jaeckel's hyperplane design (1989b) is inspired in this idea. Saarinen et al. (1991) improved memory utilization by distributing the hard addresses with Kohonen's self-organizing algorithm.

Other authors have proposed the use of genetic algorithms to distribute the hard location addresses. For example (Anwar, Dasgupta, & Franklin, 1999; Fan & Wang, 1997). Fan and Wang used a genetic algorithm to initialize the addresses of hard locations. Anwar et al. used a different fitness function to maximize the distance between hard locations. If each of these algorithms is seen as a neural network, the genetic algorithm changes the weights in the connections between the input layer and the hidden layer (matrix *A* in the ANN representation), while connections between hidden layer and the output layer (matrix *C* in the ANN representation) are updated with the standard SDM procedure.

Ratitch and Precup (2004) created the hard locations as needed, distributing the hard locations following the distribution of the data. Their design does not require allocating memory for hard locations that are not used, as is done in the original SDM. When data needs to be stored, new hard locations are created in the neighborhood of the input data if their number is less than a predefined value. If the predefined maximum number of hard locations has already been reached, an infrequently active hard location is first removed before creating a new one. The content of the hard location to be removed is combined with its nearest neighbor. Using similar ideas, Sutton and Whitehead (1993)

slowly move rarely active hard locations towards the address of data if the number of active hard locations for that data is below a certain value.

Helly, Willshaw, and Hayes (1997) proposed an alternative signal model that *propagates* the input data through the entire memory, decreasing the signal strength proportionately with the distance from the input. They also used a pruning mechanism similar to Sutton and Whitehead. This mechanism eliminated the requirement of predefining the access radius that best fit the data. They reported a notable improvement for non-random data over the original SDM.

Other Variations

Furber and colleagues (2004) developed an SDM version using spiking neurons. They used sparse codes, where only *n* of *m* bits are ones in the word vectors. They based their design on Jaeckel's hyperplane design for the activation of hard locations, using a Willshaw memory (Willshaw, 1981; Willshaw, Buneman, & Longuet-Higgins, 1969) as an alternative to counters for storage of the data. This design choice diminishes the capacity and noise robustness of the memory as pointed out by Kanerva in his analysis of SDM with one bit counters. However the most recently stored words in this model are easily retrieved, providing a good model for short term memory (Kanerva, 1988, pp. 75 - 76). Bose, Furber, and Shapiro (2005) extended this design to store sequences.

Ramamurthy, D'Mello, and Franklin (2006) introduced forgetting as part of an unsupervised learning mechanism. They decay the counters toward zero over time according to a sigmoid function, with the result that only sufficiently repeated vectors are preserved in the memory. 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 (D'Mello, Ramamurthy, & Franklin, 2005; Ramamurthy, D'Mello, & Franklin, 2004). This latter variation increased the performance for text based applications. Finally Anwar and Franklin (2005) introduced a model of SDM that can handle small cues, that is, vectors with a small number of dimensions.

Applications

Several applications were created using SDM as their main component or as a part of them. In this section, I present some representative applications in various domains. Of course, this sample by no means limits the possible applications to only these domains.

The properties of SDM make it good candidate for a cognitive agent's episodic memory model (Ramamurthy & Franklin, 2011). Various authors used SDM for speech and pattern recognition (Clarke, Prager, & Fallside, 1991; Fan & Wang, 1997; Joglekar, 1989; Meng et al., 2009). Others implemented prediction applications using SDM (Howell & Fowler, 1990; Rogers, 1990). And still others developed memory systems, especially procedural memory, for robot control applications (Jockel, 2009; Mendes, Coimbra, & Crisóstomo, 2009; Mendes, Crisostomo, & Coimbra, 2008; Rao & Fuentes, 1998).

LIDA Episodic Memory

The LIDA model (Baars & Franklin, 2009; Franklin & Patterson, 2006; Ramamurthy, Baars, D'Mello, & Franklin, 2006) is a comprehensive, conceptual and computational model covering a large portion of human cognition. Based primarily on Global Workspace theory (Baars, 1988) the model implements and fleshes out a number of psychological and neuropsychological theories. The LIDA model and its ensuing

architecture are grounded in the LIDA cognitive cycle (Baars & Franklin, 2003; Franklin et al., 2005). Every autonomous agent (Franklin & Graesser, 1997), be it human, animal, or artificial, must frequently sample (sense) its environment and select an appropriate response (action). More sophisticated agents, such as humans, process (make sense of) the input from such sampling in order to facilitate their decision making. The agent's "life" can be viewed as consisting of a continual sequence of these cognitive cycles. Each cycle constitutes a unit of sensing, attending and acting. A cognitive cycle can be thought of as a moment of cognition, a cognitive "moment". During each cognitive cycle the LIDA agent first makes sense of its current situation as best as it can by updating its representation of its current situation, both external and internal. By a competitive process, as specified by Global Workspace Theory (Baars, 1988), it then decides what portion of the represented situation is most in need of attention. Broadcasting this portion, the current contents of consciousness, enables the agent to choose an appropriate action and execute it. The different memories of the agent may also learn the broadcast content, completing the cycle.

LIDA includes several memory modules implemented in several different technologies. SDM 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 (Baddeley et al., 2001; Franklin et al., 2005). LIDA's transient episodic memory and declarative memory are implemented using variations of SDM (Ramamurthy, D'Mello et al., 2006; Ramamurthy et al., 2004; Ramamurthy & Franklin, 2011). The forgetting and consolidation mechanisms are interesting improvements for the episodic memory of cognitive agents

(Ramamurthy, D'Mello et al., 2006). When implementing the forgetting introduced in the previous section, the counters of each hard location of the episodic memory are decayed according to a sigmoid function. Counters with high values decay more slowly than counters with low values. Counters with high values are a consequence of highly repeated word vectors. Eventually, only counters with high values will remain and only these highly repeated words will be preserved in the memory. These words that are preserved in the episodic memory are *consolidated* to the declarative memory. The declarative memory, implemented with a second SDM, has exactly the same address for each hard location. The consolidation process is as follows: at predefined intervals the counters of each hard location of the declarative memory is updated with the counters of the corresponding hard location in the transient episodic memory. Declarative memory has a slower decay rate than episodic memory, preserving its contents for longer periods.

Pattern and Speech Recognition

Prager and Fallside (1989) and Clarke et al. (1991) implemented a short word recognition system based on continuous speech inputs. Testing the system with 133 small words, they reached a recognition accuracy of 95% without syntactic constraints. Their model used a variation of the original SDM that is able to represent real values. Each utterance of a vowel was represented by a 128-dimensional vector of real numbers.

Joglekar (1989) studied phonemes recognition with NETtalk data (Sejnowski & Rosenberg, 1986). He mapped hard locations directly to sample data to obtain the best results. Additionally, Danforth (1990) experimented with recognition of spoken digits. He represented the words with 240 bits. The results improved dramatically when some

words where used as addresses of hard locations. However his best results were achieved using Jaeckel's selected coordinate design.

Several authors implemented pattern recognition applications with SDM. Fan and Wang (1997) implemented a digit-recognition application using genetic algorithms to allocate the hard locations in the space. Meng et al. (2009) created a modified version of SDM that allocates hard location addresses with some of the data vectors improving the efficiency of the system. They also implemented the counters with only 2 bits but included a *tri-state* (high impedance) value. This design diminished the memory requirements and facilitated the hardware implementation while keeping the performance relatively high.

Prediction Applications

Rogers (1990) implemented a weather forecasting application using a combination of SDM and a genetic algorithm. He trained the system with 58,000 weather samples for the Australian coast. Each sample included features such as temperature, air pressure or cloud cover. The predictions using this mixed application outperform the results of the application using only SDM. Howell and Fowler (1990) developed a simple application that predicted academic success or failure for dental college students. They reported a performance of 68%, higher than similar studies of that time.

Perhaps the most promising prediction applications are related to sequence learning, and are strongly related with robot navigation, which is explored in the next section.

Robot Navigation and Manipulation

Several authors experimented with SDM as a main component of robot navigation systems. Rao and Fuentes (1998) created a system that employed a SDM combined with Brooks' subsumption architecture (Brooks, 1986) to learn adaptive navigational behaviors. They trained the system with vectors formed from the sensor data and motor inputs from the three most recent perceptions. The SDM was modified to self-organize the inputs in the address space.

Mendes et al. (2008, 2009) experimented with a robot vehicle that uses video images and motor information as sensory inputs. They utilized a modified SDM to predict the subsequent movements during autonomous navigation after training. Their SDM uses a randomized reallocation algorithm to dynamically allocate new hard locations as needed. The authors also compared several encoding methods for real or integer values when they are used with SDM. I will explore this issue in more detail in Chapter 5.

Jockel (2009) developed a robotic arm manipulation system based on the modified SDM of Mendes (2008) and Bose (2005). The memory dynamically allocates hard locations as needed and used buffers instead of counters. He also developed a multi-fold memory, as suggested by Kanerva (1988), for storage of sequences. Each fold is in fact an independent SDM, and the system can have multiple folds. The k^{th}*-fold* stores a prediction for the next element based on the element *k* prior steps in the sequence. The system combines the predictions of all folds to determine the next element. I will discuss a simpler approach for the same problem in Chapter 4.

In summary, Sparse Distributed Memory is an associative memory based on the properties of high dimensional binary spaces. It is composed of hard locations, the storage units of the memory. Its auto-associativity and noise robustness make it a good candidate for several applications, such as episodic memory for cognitive architectures, robot navigation controllers, and pattern recognition. Several authors developed variations and improvements: the forgetting mechanism, dynamic allocation of hard locations, and variations in the hard location activation mechanism are some of the extensions described in this chapter.

Chapter 3: Vector Representation

In Chapter 1, I mentioned the importance of the representation chosen for a system, and the degree to which the representation influences which task categories the system can compute optimally. In this work, I will discuss representations that help to perform challenging AI applications, in particular vector representations. In Chapter 1, I extensively described some of the basic operations required for these applications, and desirable properties of the representation and memory systems that readily support these basic operations. Plate (2003) described the properties of representation models in general and the ones suitable for connectionist systems in particular (pp. 2-16). Distributed representations in connectionist models are intimately related with vector representations. Here, I will summarize these concepts, and focus on representations based on long vectors and their properties.

In classic AI representations, there are two main approaches: the symbolic approach, and the connectionist approach that bases representations on the state of a simple network of units. Another representation model is the vector representation, built on vector spaces. Finally, some researchers claim that no representation is required at all (or at least, its importance is not as strong as the other approaches maintain) (Brooks, 1991).

Newell and Simon (1976) define physical symbol systems as follows:

> A physical symbol system consists of a set of entities, called symbols, which are physical patterns that can occur as components of another type of entity called an expression (or symbol structure). Thus, a symbol structure is composed of a number of instances (or tokens) of symbols related in some physical way (such as one token being next to another). At any instant of time the system will contain a collection of these symbol structures. Besides these structures, the system also contains a collection of processes that operate on expressions to produce other expressions: processes of creation, modification, reproduction and destruction. A

> physical symbol system is a machine that produces through time an evolving collection of symbol structures. Such a system exists in a world of objects wider than just these symbolic expressions themselves.

Basically, these systems are composed of entities (symbols) that can be instantiated (tokens), and of rules to manipulate them. Symbols are attractive representations for high-level problems such as planning or chess playing, but they seem less appropriate for other tasks, such as those required for challenging AI applications: object recognition, sequence learning, and so on. Further discussion of symbolic AI is beyond the scope of this work.

Connectionist systems, such as neural networks or semantic networks, can represent knowledge and data in several ways. The long-term knowledge (or data) representation is often based on the weights of the links between units. The different states or activation patterns of the units compose the short-term data representations.

The short-term representations in connectionist systems can in turn be subdivided into localist and distributed representations. In the former, each unit represents a single object, concept, or element of the system. The represented elements have a one-to-one correspondence to the system's units (Franklin, 1995, p. 132). The main advantage of a localist approach is the explicit representation of data. An external observer can easily interpret the activation of the units as the current representation of the system. For example, semantic networks and similar models, such as the Perceptual Associative Memory in the LIDA architecture (Ramamurthy & Franklin, 2011), follow this paradigm. Passing activation among units can explicitly implement constraint rules; or reinforcing the units' activation based on the activation of others can model similarity and composition of elements. Finally, localist representations are good candidates for input to

or output from a system. For example, in classification tasks, the output vector's dimensions represent the possible classification categories, where the value of each dimension denotes the probability of the probed element belonging to the corresponding category. Because of its explicit representation, these networks are easy to design according to the requirements of the system. Despite its advantages, this type of representation has several problems, mostly related to inefficiency. The one-to-one correspondence between items and units in the system implies that representing *n* items requires *n* units. For a system with few items, this may be reasonable, but it becomes impractical for large number of elements. Moreover, even similar items require an individual unit to represent each one. Something similar occurs with the connections between elements; their number can increase geometrically, producing in many cases a high degree of redundancy.

On the other hand, in distributed representations each item is represented by the activation of several units, and each unit can participate in the representation of a number of items (Franklin, 1995, p. 132; Hinton, McClelland, & Rumelhart, 1986). This representation is more efficient than the localist one. For example, 10 units can represent 2^{10} elements. The patterns of activation of the units comprise a vector, where each unit in the system corresponds to a dimension. The distributed representation is more compact and computationally efficient than the localist, but at the expense of explicitness. In an interesting alternative, the units themselves can represent explicit features of the item (e.g., is-red). The pattern of activation of several units distributively represents a particular item, but each unit locally represents a *microfeature* (Hinton et al., 1986). This intermediate model has some advantages. Similar elements have similar representations,

because they may share several features. This can lead to automatic generalization, since similar items will activate similar patterns of units, and the system will capitalize this reacting alike (Franklin, 1995, pp. 132-133). However, a system might require a large number of microfeatures in order to represent all possible items, making this model impractical.

Distributed representations can implement what Plate (2003) calls *explicit similarity*: similar elements have similar representations (p. 13). Several kinds of similarity measures can be used among vectors, e.g., the cosine, or the inverse of some distance, such as the Hamming (for binary vectors) or Euclidean distances.

Explicit similarity becomes even more advantageous using vectors that belong to high dimensional spaces (i.e., vector spaces with a large number of dimensions). Such spaces offer an enormous number of possible units' activation patterns, and the necessity for compact representations becomes less critical. There is no need for a one to one correspondence between patterns and items. For example, in a binary space with 1,000 dimensions, we can theoretically represent 2^{1000} different items, but this is highly unlikely. We can use just a fraction of the vectors in the space, say 2^{100} vectors distributed in the space, which still allows a gigantic number of possible representations. Even after adding some noise by introducing a few changes in one of these vectors, it can still represent the same item. In other words, a region of the space, instead of just one point, represents an item, creating a more noise robust representation that gracefully degrades as noise increases, and produces desirable properties such as pattern completion. (See Chapter 2 for an extended discussion on this subject.)

Distributed representations are generally associated with connectionist systems. However, we can abstract the representation from the implementation. A vector itself can represent an item without corresponding to the pattern of activation of units in a connectionist system. In many subfields of computer science vector representations constitute one of the main types of data structure. For example in machine learning, a vector of features–often of different data types–represents an element in a training set. A different approach, and closer to the focus of this work, utilizes vectors where all the dimensions share the same data type. Even with this uniformity, the number of possible representation models is limitless. The way to calculate or define the vector representation for an item, and the distance or similarity measurement define the representation and its properties. For example, in the last two decades a large number of semantic space models have emerged that use high dimensional vectors to represent words and texts. The most representative models include Latent Sematic Analysis (LSA) (Deerwester et al., 1990) based on statistical analysis; Random Indexing (Sahlgren, 2005),which employs random sparse vectors and random permutations; and BEAGLE (Jones & Mewhort, 2007), which computes vectors using circular convolution. For recent surveys of semantic space models see (Cohen & Widdows, 2009; Turney & Pantel, 2010).

In an even more generic view, vectors can represent any concept or element of interest: objects, features, rules, constraints, actions, etc. As explained above, when a vector belongs to a high dimensional space, interesting properties arise. For example, two randomly chosen points of the space are far away from each other, which Kanerva (1988)

defines as tendency to orthogonality, making them good candidates to represent unrelated concepts. For a complete discussion of this subject, see Chapter 2.

Reduced Descriptions

Here, I discuss the main ideas behind reduced descriptions. For further information, see Plate (2003).

One frequent criticism of distributed representations (or vector representations) is the difficulty they pose in the representation of complex structures. Performing high level cognitive tasks such as reasoning, planning, or action selection often involves structures with multiple elements. Implementations of these tasks frequently utilize structures such as sequences, hierarchies, and variable binding. Moreover, the elements of these structures can in turn be complex structures themselves. Of course, we can create these structures and use vectors as elements. But, in that case, the vectors become mere symbols, with a significant loss of expressive power. Hinton (1990) introduced the concept of reduced description, a method for encoding complex structures as single vectors. The main idea is to have a dual representation: the structure can be represented explicitly, with a vector for each component, or as a reduced description, where a single vector represents the whole structure. When the system focuses on a particular composite element, its constituent structure is represented in full, instantiating all the elements (vectors) that compose it. On the other hand, when the element participates in the structure of another element that has the current focus, it is represented with a single vector as a reduced description. See Figure 9.

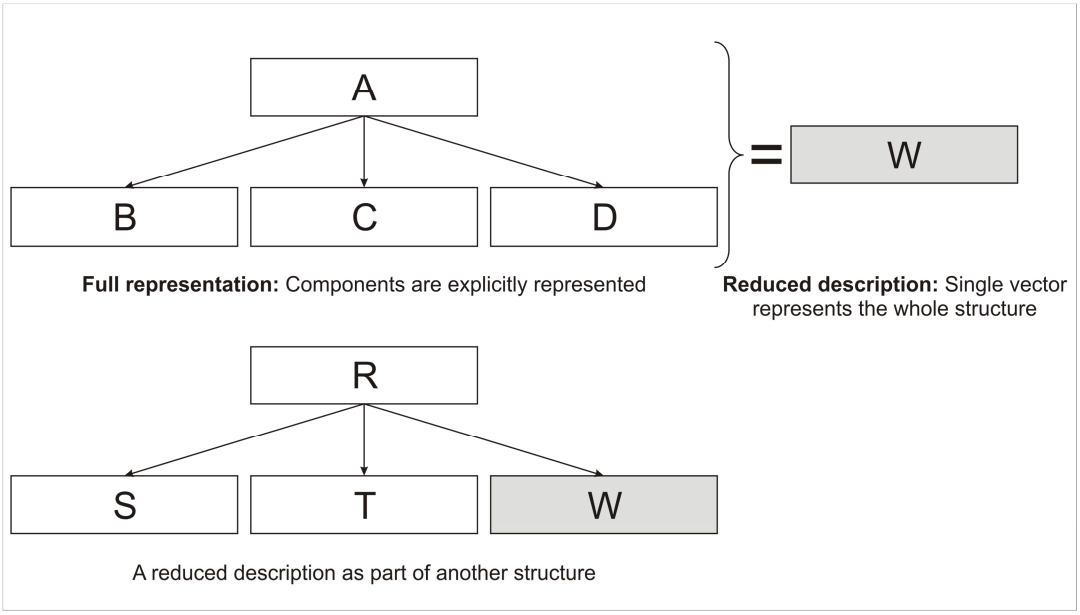


Figure 9. Reduced description. A complex structure has a dual representation: a full representation with an explicit structure where each element is a vector, and a reduced description, where a single vector represents the whole structure.

The reduced description is not a mere pointer to the full description, but a loosely compressed version of the original structure. Using pointers to create data structures has a long history in computer science. For example, a *struct* in the C programming language can have several elements, where some of them may be also pointers to other structures. Pointers help create lists, trees, or other data structures. Object oriented languages, such as Java, hide the pointers from the programmer using objects references, but they employ essentially the same mechanism: an object reference leads to the actual location of the object in memory. A pointer (or object reference) does not have any direct relationship with the data it points to. In other words, looking at the pointer rather than what it points to reveals nothing about the data. Furthermore, given an item (or part of it), it is not possible to locate it easily. Hash indexing is probably the traditional computer science technique most similar to reduced descriptions. Hashing allows the location of data to be

calculated from its content. However, the hashing usually does not provide any information about the content, and similar elements often have very dissimilar hashing values. The reduced descriptions, on the other hand, are abbreviated representations of the full data. Moreover, as we shall see, several operations can use directly the reduce descriptions without needing to recover the original data.

Plate (2003) analyses reduced descriptions from four desirable characteristics:

- *Representation adequacy*: The reduced description must be able to reconstruct or retrieve the full representation. Failing to this is analogous to a pointer that does not point to its data.
- *Reduction*: The reduced description must be smaller than the full representation. In general, the vectors used in vector representations are of a fixed size, and a single vector comprises a reduced description.
- *Systematicity*: The construction of the reduced description should be systematic. That is, the way to construct the reduced description must be well known and deterministic. This facilitates the reconstruction of the full representation.
- *Informativeness*: The reduced description should contain some information about the whole it represents. This allows its direct use for certain operations without retrieving the full representation (pp. 19-20).

Defining a reduced representation model determines basic operations that combine vectors and produce these required characteristics. The next section explores these basic operations in general, and the following sections describe some of the most relevant reduced description models.

Basic Operations to Combine Vectors

Many of the complex structures apply to AI problems pervasively; examples include sequences, hierarchies, and predicates (i.e., rules with variable binding). These structures, and probably others, can be constructed out of even simpler primitives such as binding and grouping. Binding is the assignment of one element, which is called the filler, to a particular role or position in the structure. For example, in a sentence, an element "Sue" can be bound to the *subject* role. Grouping is forming a set (or collection) of elements. For example, the structure to represent a sentence can be a collection of roles (bound to their fillers) where each role stands for a part of the sentence. In a similar way, a sequence can be modeled with the group of its elements, each of them bound to its position in the sequence. To create a reduced description model, we need to define binding and grouping operations[1], and a distance or similarity measure. Kanerva (2009) introduced more abstract names for these operations; he uses multiplication for binding, and sum for grouping, which simplifies the operations' notation. I will use this same convention here. The following summarizes the Kanerva's ideas of hyperdimensional arithmetic (Kanerva, 2009).

In general, the multiplication and sum operations don't necessarily correspond with the usual arithmetic operations, but they should have several properties in common. These properties, in turn, facilitate the achievement of the four characteristics of reduced description models described in the previous section. For example, the multiplication must be reversible; this allows *unbinding* the filler to reconstruct the original structure. I will use the operations defined in Spatter Code (Kanerva, 1994) as examples of the more

[1] Some systems can create reduced descriptions without explicitly defining these operations. For example see RAAM (Pollack, 1990).

general cases. This representation model uses bitwise XOR as multiplication; integer sum, in each dimension, followed by a normalization process, as sum; and Hamming distance as the distance measure.

If a vector A represents an element and vector B represents a role, the binding of A to B is given by:

$$C = A \otimes B \tag{2}$$

where $\otimes$ denotes the multiplication operator (e.g., XOR in Spatter Code). Multiplication by the *inverse* vector reverses this operation. The definition of the inverse vector depends on the multiplication operator used. In the XOR case, it is the same vector, but in other reduced description models (with a different multiplication operation) the inverse could be another vector[2]:

$$A = C \otimes B^{-1} \tag{3}$$

In the binary case using XOR, $B^{-1} = B$.

The multiplication must be commutative and associative:

$$A \otimes B = B \otimes A \tag{4}$$

$$(A \otimes B) \otimes C = A \otimes (B \otimes C) \tag{5}$$

Bitwise XOR fulfills these two properties. In some cases, a non-commutative multiplication becomes handy. Applying a random permutation by changing the order of

[2] Some versions of multiplication may not have an inverse for all possible vectors. This is analogous to 0 in the real numbers, which has no inverse.

the dimensions of one of the operands before computing the XOR produces an alternate non-commutative multiplication. This technique applies to binary spaces as well as other vector spaces; for more details, see (Kanerva, 2009; Plate, 2003). In general, a particular system employs a single random permutation that does not change for that particular system after its creation. Random permutations allow modeling other data structures such as sequences efficiently. See Chapter 4 for further details.

The multiplication also preserves distances:

$$d(A, B) = d(A \otimes C, B \otimes C) \tag{6}$$

This is easily verified for the XOR operation. The Hamming distance is the number of bits by which *A* and *B* differ. For example, if dimension *i* of vector C is 0, $A_i XOR\ C_i = A_i$. Similarly, $B_i XOR\ C_i = B_i$. If dimension *i* of C is 1, $A_i XOR\ C_i = \neg A_i$; and, $B_i XOR\ C_i = \neg B_i$. In both cases, the XOR operation preserves the difference between A_i and B_i, thus the distance between *A* and *B* is the same as the distance between *A XOR C* and *B XOR C*.

Interestingly, the multiplication in general produces a vector that is dissimilar to its operands:

$$A \otimes B \not\approx A \text{ and } A \otimes B \not\approx B \tag{7}$$

where $\not\approx$ denotes dissimilarity.

The sum must also be associative and commutative:

$$A + B = B + A \tag{8}$$

$$(A + B) + C = A + (B + C) \tag{9}$$

In Spatter Code, the sum is defined as the integer sum for each dimension of the vectors after they have been transformed into bipolar vectors with the zeros replaced by minus ones. A normalization function (e.g., a simple threshold function) yields a binary vector again. For each dimension, if the integer sum is positive, a one is assigned to that dimension, or zero otherwise. Actually, the sum defined in this way is not strictly associative, due to the normalization. But we can define a multi-operand sum that first computes the integer sum of all the operands for each dimension, and normalizes it (denoted by […]) at the end.

$$Sum(A, B, C, \dots) = [A + B + C + \cdots] \tag{10}$$

The resulting vector of the sum is similar to its operands:

$$A + B \approx A \; and \; A + B \approx B \tag{11}$$

Finally, multiplication has to distribute over sum:

$$A \otimes (B + C) = A \otimes B + A \otimes C \tag{12}$$

Random permutations (denoted by capital Greek letters: Π, Γ, etc.) can be used as a kind of multiplication. It is not a real multiplication, because one of the operands is not a vector, but different permutations can represent different roles. In this case, applying a permutation to a vector binds the vector to the role represented by the permutation. For example, if Π and Γ represent color and shape respectively,

$$A = \Pi(red) + \Gamma(square) \tag{13}$$

then *A* represents a red square.

Permutations also preserve distances, are commutative, associative, and distributive over the sum. Moreover, they have an interesting advantage over other multiplications: they preserve the vectors' density, defined as the relative number of zeros and ones. Some associative memories (e.g., Willshaw et al., 1969) and some representation models (e.g., Rachkovskij & Kussul, 2001) perform better with sparse vectors (i.e., vectors with few ones). Permutations work well for both sparse vectors and dense vectors, which have an equal number of zeros and ones (see Rachkovskij & Kussul, 2001 for further discussion on this subject).

Summing up, to create a reduced description we have to define multiplication and sum operations, as well as a distance measure in a vector space. The multiplication must be associative, commutative, and distributive over the sum. It must also preserve distance, and produce vectors dissimilar to its operands. The sum has to be associative (with some license) and commutative, and must produce vectors similar to its operands. These properties of the multiplication and the sum allow creating reduced description vectors, and performing the operations described later in the hyperdimensional section. A discussion about these properties can be found in (Kanerva, 2009; Plate, 2003).

Combining random permutations with some multiplications yields a non-commutative multiplication that is useful to model some structures. Permutations can be used as multiplications by themselves to model some bindings. Although it is not a hard

requirement, using high dimensional vectors enhances some of these properties. See Chapter 2 for details.

Spatter Code

Kanerva developed Spatter Code (1994, 2009) as a reduced description model based on large binary vectors. Vectors of high dimensional spaces tend to be orthogonal; making them good candidates for representing unrelated concepts (see Chapter 2 for details). Spatter Code defines the sum operation, also called superposition, as an integer sum in each dimension followed by a normalization process (in general, a threshold function). Bitwise XOR is the multiplication, or binding operation, and it employs the Hamming distance as a similarity measure.

Spatter Code can encode a set of elements using the sum operation. For example, three binary vectors *J*, *M*, and *T*, representing John, Mary, and talk respectively, can be combined to denote the event "John is talking with Mary":

$$E = [J + T + M] \tag{14}$$

The vector *E* captures the relationship between *J*, *M*, and *T*, but not the role that these elements have in the structure. A problem with this representation appears when the roles in the event or relationship matter. For example, the events: "John is looking at Mary," and "Mary is looking at John" have the same encoding:

$$[J + L + M] = [M + L + J] \tag{15}$$

Moreover, this representation suffers from the crosstalk effect (i.e., spurious representations produced by the superposition). For example, if we want to represent "blue car and red truck" with the vectors *B, C, R,* and *T*:

$$E = [B + C + R + T] \tag{16}$$

where phantom representations can appear: red car and blue truck. Using multiplication to bind elements to roles solves these problems. If we define vectors for the roles –*S* for agent, *A* for action, and *O* for object– a representation of the sentence "John is looking at Mary" follows:

$$E = [S \otimes J + A \otimes L + O \otimes M] \tag{17}$$

To extract the subject of the event *E*, we can multiply it by S^{-1} (in the binary case, $S^{-1} = S$). Thus,

$$S \otimes E = S \otimes [S \otimes J + A \otimes L + O \otimes M] \tag{18}$$

$$S \otimes E = [J + S \otimes A \otimes L + S \otimes O \otimes M] \tag{19}$$

$$S \otimes E = [J + N_1 + N_2] \tag{20}$$

where N_1 *and* N_2 can be considered as noise. Reading a cleanup memory that has *J*, *L* and *M* stored in it with $S \otimes E$, produces *J*, our answer. This example shows the necessity of a cleanup memory to work with reduced descriptions that helps recover the clean version of the vectors composing the reduced description. Spatter Code allows other operations that I will describe in the next sections.

A problem with Spatter Code arises due to the normalization after the sum. Remember that the sum operates over bipolar vectors (see previous section for details). After the sum, but before normalization, some dimension may be 0, and the normalization process–a threshold function centered on zero–must be defined randomly in these cases. When the sum comprises few operands, for example two, many dimensions of the sum vector are 0, introducing too much noise in the representation, making the representation brittle. This is a common problem with normalization in all reduced descriptions because this operation finally packs several vectors into one (of the same size and characteristics of the operands), producing some loss in the representation. Nonetheless this problem is more noticeable in the binary case than it is in HRR (or in the Modular Reduced Representation to be introduced in Chapter 5). In these other representations, summing two vectors can also produce undetermined values for some dimensions that must be determined randomly, as explained above for the binary case. The cases that produce this problem depend on the definition of the sum, but in general the problem appears when the values corresponding to one dimension in the combining vectors are complementary, that is, one value is the opposite of the other. In the binary case the 1 is the complement of the 0, generating this situation very often. Representations with more possible values for each dimension have more expressiveness, and the problem appears more infrequently.

Holographic Reduced Representation

Plate (1995, 2003) proposed the Holistic Reduced Representation (HRR), a reduced description model based on large vectors of real numbers. I describe here the operations

and requirements of the vectors of this representation model in some detail, which will be useful when comparing HRR with Modular Composite Representation in Chapter 6.

HRR uses the sum in each dimension as its superposition operation. The multiplication is a bit more complex. It utilizes circular convolution, an operation that resembles the convolution of vectors, but the result preserves the dimensionality of the operands. To decode circular convolution it uses circular correlation. Actually, correlation can be expressed as the convolution of a vector with the involution of the second operand (Plate, 2003, p. 97). To be consistent with the nomenclature, the involution of A will be represented by A^{-1}. In order for these operations to work as expected, having the properties described in previous sections, the possible values for each of the n dimension of the vector must be independently distributed with 0 mean and variance $1/n$. For example, a suitable distribution is a normal distribution $N(0,1/n)$.

Plate extensively demonstrated the operations and applications of HRR (Plate, 2003). All the operations described in previous sections can also be implemented using HRR. There is a practical limit to the number of vectors that can be combined into a single one before interference between the operands introduces so much noise that the combined vector becomes useless. HRR's interference limit, which can be about 12 elements, is greater than in the binary case. This makes HRR an interesting option for representing complex structures for hyperdimensional computing. However, the complicated operations that it uses, including circular convolution and circular correlation, the computational complexity of these operations, which is $O(n^2)$[3], and the requirements of the vectors make HRR less attractive.

[3] This can be improved to $O(n\log n)$ using FFT.

HRR in the Frequency Domain

Plate (2003) also proposed a modification of HRR in the frequency domain in which the space resulting from the Fourier-transformed vectors (pp. 145-151). The implementations of circular convolution and circular correlation in the frequency domain execute faster than in the *time* domain–the typical space of the vectors–even considering the time employed to transform the vectors to and from the frequency domain. Even better, creating the vectors directly in the frequency domain space avoids the transformations altogether. HRR in the frequency domain, also known as *circular* HRR, works with unitary complex numbers (i.e., complex numbers with modulus equal to one) as possible values in each dimension. Since these complex numbers all have modulus one, the dimensions of a circular HRR vector are determined by the angles of these complex numbers, which can be uniformly distributed on $(-\pi, \pi]$ without any constraint. The circular convolution in this domain is equivalent to the dimension-by-dimension sum of the angles, and the inverse of a vector corresponds to the negation of the angle in each dimension. Plate defines the superposition operation as the sum of the complex numbers, followed by a normalization that simply discards the modulus and takes only the angle of the resulting vector. Finally, circular HRR employs the mean of the cosines of the difference between corresponding angles as its similarity measure.

This representation has even better performance than the standard HRR. All the operations perform in linear time, and some of them introduce less noise. The only complaints raised by Plate were the more complex sum and similarity measure operations, and the difficulty introduced by working with angles in connectionist systems as opposed to working with real numbers. The Modular Composite Representation,

which can be compared with the HRR in the frequency domain, proposes alternatives that overcome these difficulties (see Chapter 6 for details).

As Kanerva (1996) points out, Spatter Code is equivalent to HRR in the frequency domain when the possible angles are restricted to 0 (equivalent to binary 0) and π (equivalent to binary 1). Modular Composite Representation, originally based on a generalization of Spatter Code, shares similarity with a special case of HRR in the frequency domain, as noted by Kanerva in a personal communication to the author. I will further discuss this similarity in Chapter 6.

Hyperdimensional Computing

Both, Kanerva (1994, 1996, 1998, 2009) and Plate (1995, 2003) describe several operations and experiments using Spatter Code and HRR. Kanerva (2009) presented a comprehensive and well organized review of these technologies and operations under the name of *hyperdimensional computing*. Here I will present a summary of these ideas. For more details and results, see (Kanerva, 1998, 2009; Plate, 2003). Some of the operations were already described in previous sections. I will repeat them here for completeness.

Binding

Binding tightly associates two vectors, creating a new vector that is dissimilar to both operands. Multiplication is used to perform this operation. For example, if A and B are vectors, then

$$C = A \otimes B \tag{21}$$

where C represents the binding between A and B.

Some representations require a non-commutative binding. In these cases, a variation of multiplication using random permutation fulfills the requirement:

$$\boldsymbol{C} = \Pi(\boldsymbol{A}) \otimes \boldsymbol{B} \tag{22}$$

where Π represents a random permutation.

Unbinding

Unbinding is the inverse of the binding operation. The unbinding operation allows finding the filler given the role, or a value given the variable. Multiplying the binding vector by the inverse of one of the constituents of the bond yields the other element:

$$\boldsymbol{A} = \boldsymbol{C} \otimes \boldsymbol{B}^{-1} \tag{23}$$

In the binary case, $\boldsymbol{B}^{-1} = \boldsymbol{B}$, but HRR (and other reduced description models) requires calculation of the inverse vector. When the non-commutative binding is used, we have two different unbinding operations, one for the retrieval of each operand:

$$\boldsymbol{A} = \Pi^{-1}\left(\boldsymbol{C} \otimes \boldsymbol{B}^{-1}\right) \text{ for the first operand, and} \tag{24}$$

$$\boldsymbol{B} = \boldsymbol{C} \otimes \Pi^{-1}\left(\boldsymbol{A}^{-1}\right) \text{ for the second operand.} \tag{25}$$

Grouping

Grouping, also known as superimposition or superposition, combines elements that form a set, record, or similar compositional structure. The sum operation followed by normalization (in most of the cases) produces grouping:

$$G = [A + B + C] \tag{26}$$

G, a vector that represents the composition of *A*, *B*, and *C*, is similar to each of its operands. An interesting combination of binding and grouping produces representations for records or relationships:

$$S = [A \otimes R_1 + B \otimes R_2 + C \otimes R_3] \tag{27}$$

where R_1, R_2, and R_3, are vectors that represent roles. For example, the representation for a geometric figure follows:

$$F = [circle \otimes Shape + red \otimes Color] \tag{28}$$

This same procedure can be used to represent relationships. Suppose the relation *parent* (*p*, *c*), and *A* is parent of *B*. The vector *R* represents this relationship:

$$R = [parent + p \otimes A + c \otimes B] \tag{29}$$

Adding a role vector for the type of relationship (the vector *relationType* in the following example) helps to retrieve this information using probing (see next section):

$$R = [relationType \otimes parent + p \otimes A + c \otimes B] \tag{30}$$

The vector *R* is different from *A* and *B*; this implies that two relationships with the same fillers are not similar. To make them similar, we can include the fillers (i.e., *A* and *B*) as new terms into the equation:

$$R = [relationType \otimes parent + p \otimes A + c \otimes B + A + B] \quad (31)$$

Now relationships with *A* and *B* as fillers will be similar, and the fillers can be used to cue the relationship. But introducing more terms in the composition of a vector makes it noisier and more brittle. For additional examples of representations of structures, see Plate (2003).

Probing

Superimposing (grouping) vectors does not easily allow reconstruction of the components of the resulting vector, but it does admit *probing*, or in other words, testing if the group vector includes a specific vector. Since the group vector is similar to its elements, the distance between *G* and *A* in the previous example must be less than the indifference distance, as defined in Chapter 2. Using a simple threshold function we can probe whether a vector is part of a group:

$$d(G, A) < Threshold \quad (32)$$

An even more interesting probe operation can produce the filler of a particular role in a group. Using the example of equation (28) in the previous section,

$$F \otimes Shape^{-1} = [circle \otimes Shape + red \otimes Color] \otimes Shape^{-1} \quad (33)$$

since multiplication is distributive over sum:

$$F \otimes Shape^{-1} = [circle \otimes Shape \otimes Shape^{-1} + red \otimes Color \otimes Shape^{-1}] \quad (34)$$

$$F \otimes Shape^{-1} = [circle + red \otimes Color \otimes Shape^{-1}] \quad (35)$$

$$F \otimes Shape^{-1} = circle + noise \approx circle \quad (36)$$

This operation produces an approximate, or noisy, version of *circle*. An auto-associative cleanup memory that stores the vectors known by the system (i.e., all the vectors used in the representations) can retrieve the original (clean) vector.

Analogies

There are two ways to use reduced descriptions: reconstructing the original structure, or using them as *holistic* vectors. Probing is an example of the former. Here I present an example of the second, which I borrowed from (Kanerva, 2009), that also exemplifies how to implement analogies using the properties of reduced descriptions.

Suppose we represent the relation between a country and its monetary unit:

$$A = [country \otimes USA + money \otimes Dollar] \quad (37)$$

$$B = [country \otimes Mexico + money \otimes Peso] \quad (38)$$

If we want to know what the dollar of Mexico is, we can simply multiply:

$$B \otimes (Dollar^{-1} \otimes A)^{-1} \approx Peso \quad (39)$$

More examples of *holistic* processing, including inference, multiple substitutions, and more complex analogies can be found in (Kanerva, 2009; Plate, 2003).

Mapping

Several of the operations of the previous sections yield approximate vectors that require an auto-associative memory to cleanup. Some vectors can be similar and valid for the system, for example a vector that represents a car, and a relation that include that vector as filler. In these cases, we may require separate memories for storage of simple vectors and composed vectors. A better solution takes advantage of the multiplication's distance preserving property. We can define a random vector to denote a *region* in the memory for simple elements, and another random vector for the relations' region.

To write to a particular region of the memory, we first multiply the vector by the region's *mapping* vector. To read from a particular region, first we multiply the cue vector by the mapping vector, and we multiply the result by the inverse of the mapping vector. The term region may be misleading. Actually, the mapping operation maps the whole space into the whole space, but in huge spaces such these, the chance that a mapped vector is similar to another vector in the system is almost zero. The mapping can also be done with random permutations.

Hierarchical Structures

Since the results of grouping and binding have the same dimensionality as their components, we can use them as components of other more complex structures. For example,

$$C = [circle \otimes Shape + red \otimes Color] \tag{40}$$

$$S = [square \otimes Shape + blue \otimes Color] \tag{41}$$

$$B = [bigger + bigO \otimes C + smallO \otimes S] \tag{42}$$

where *B* represents the relation *Bigger* (*bigO, smallO*) with *C* and *S* as fillers. The same procedure allows the representation of hierarchies. For example, a car, which is a compound object, includes elements, such as the motor and the wheels, that in turn can have their own structure.

Sequences

Several authors have proposed different ways of encoding sequences in distributed representations (for example see Kanerva, 2009; Murdock, 1983; Plate, 2003). Here I will describe a procedure to encode sequences in single vectors that resembles what I will later use for storing sequences in Extended SDM. In Chapter 4, I will extensively discuss the importance of sequences and review different ways to encode them.

To encode sequences as single vectors, we could use a role for each position in the sequence, but this is not practical because we would need to define as many vectors as a sequence could have elements, and this could become arbitrarily large. A better approach is to generate the role vectors recursively. Starting with a random vector *P* for the role of the first position in the sequence, the following roles are generated by simply multiplying the previous role by *P*.

$$S = [A \otimes P + B \otimes P \otimes P + C \otimes P \otimes P \otimes P] \tag{43}$$

or, in a more compact notation:

$$S = [C \otimes P + B \otimes P^2 + A \otimes P^3] \tag{44}$$

Interestingly, we can construct the vector *S* iteratively, adding one element at a time:

$$S_1 = A \otimes P \tag{45}$$

$$S_2 = [(S_1 + B)] \otimes P \tag{46}$$

$$S_3 = [(S_2 + C)] \otimes P \tag{47}$$

Notice that in the binary case, the inverse of a vector is itself, and a vector multiplied by itself produces a vector with all 0s, preventing the use of this technique. Nonetheless, a random permutation can replace both the random vector *P* and the multiplication, achieving the desired result. See Chapter 4 for details.

Other Models

Several authors have proposed models of memory based on vectors or similar distributed representations. Many of these modes use mathematical tools such as tensors (Dolan, 1989; Smolensky, 1990) to create role-filler representations. Other authors studied convolution-based models (Metcalfe, 1982; Murdock, 1983, 1993; Willshaw, 1981; Willshaw et al., 1969) that employ convolution to create the associations. The main problem with these techniques is that both tensors and convolution produce elements larger than the original elements, making difficult to create representations for complex structures with them. Nevertheless, some of them successfully model several human memory tasks. For example Murdock's TODAM (1983) and TODAM2 (1993), and Metcafe's CHARM (Metcalfe, 1982).

An interesting model is RAAMs (Pollack, 1990), a back propagation neural network that learns reduced descriptions of trees. Later, Chalmers (1990) designed a network based on RAAM able to create reduced descriptions of sentences, and holistically–without decoding–transform them into passive voice.

Rachkovskij and Kussul (2001) developed APNNs (Associative Projective Neural Networks), a special type of reduced description based on sparse binary vectors (i.e., binary vectors with few ones). They use an operation called Context Dependent Thinning to maintain the vector's density almost constant. The thinning operation consists of a carefully selected combination of random permutations. The results presented in (Rachkovskij, 2001) show that this model has similar characteristics to other reduced description modes such as the HRR and the Spatter Code.

Patyk-Lonska and colleges (2011) created a new reduced description, the GA model, which is similar to HRR, but based on geometric products instead of circular convolution. They report that GA's performance is superior to HRR's and similar to that of Spatter Code. However, some of the coding vectors produced by this model are larger than the operands, which discourage its application as a reduced description.

Even though they are not reduced descriptions by themselves, two models worth mentioning here for their relationships with HRR and Spatter Code respectively are BEAGLE (Jones & Mewhort, 2007) and Random Indexing (Sahlgren, 2005). Both are models of semantic spaces, and both represent words (and texts) with large vectors. BEAGLE utilizes circular convolution to create a vector representation that includes word order. Random Indexing uses binary vectors and captures the representation of

word order using random permutations. A comparison of both models can be found in (Recchia, Jones, Sahlgren, & Kanerva, 2010).

Chapter 4: Extended Sparse Distributed Memory

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 (Kurby & Zacks, 2008; Snaider et al., 2012; Sun & Giles, 2001). Kanerva, in his original work, described the use of SDM to store sequences (Kanerva, 1988). His procedure has the disadvantage of losing most of the auto-associative properties and noise robustness of the memory. Later he proposed hyperdimensional arithmetic as a new mechanism for storing sequences and other data structures such as sets and records (Kanerva, 2009). 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). Although interference is a desirable property of the memory because it mimics psychological effects, in this case it diminishes the capacity to retrieve sequences.

In this chapter, I propose a variant to the original SDM, called Extended Sparse Distributed Memory (ESDM), which is especially suitable for storing sequences and other data structures such as trees (Snaider & Franklin, 2011). This new extension considerably improves the performance of sequence storage of the memory as compared to both the original SDM memory sequence storage and the hyperdimensional arithmetic sequence storage version introduced by Kanerva (2009).

In the following section I describe the importance of sequence learning. Then I introduce Extended SDM, discussing several uses of this extension and its results. Several simulations are then presented and discussed. Finally, I propose some future directions.

Sequence Learning

Spatial-temporal sequence learning is one of the most important forms of learning for humans and animals (Starzyk & He, 2007; Sun & Giles, 2001). Sequences are used in procedural learning, to learn new skills, high level planning and problem solving.

For autonomous agents, time perception and representation are critical (Snaider et al., 2010, 2012), and sequence learning is a key component of these processes. An autonomous agent can be defined as "A system embedded in, and part of, an environment that senses its environment and acts on it over time in pursuit of its own agenda, so that its actions affect its future sensing" (Franklin & Graesser, 1997). We humans are good examples of autonomous agents, as are most animals, some mobile autonomous robots and some computer viruses. To be able to plan and foresee the result of an action, or group of actions, is a desired ability for many autonomous agents. From a cognitive point of view, time presents three major aspects: succession, duration, and temporal perspective (Block, 1990). Succession refers to the sequence of events from which an agent can perceive event order and succession.

Sun and Giles (2001) enumerate several domain problems where sequence learning is a main component: "inference, planning, reasoning, robotics, natural language processing, speech recognition, adaptive control, time series prediction, financial engineering, DNA sequencing, and so on." Each of these problems has its own particular issues that constrain the possible approaches. Even though there is a large body of research on engineering applications in sequence learning, in this work, I will focus on associative memories and related architectures.

Sun and Giles (2001) also classified sequence learning problems into four categories: sequence prediction, sequence generation, sequence recognition, and sequential decision making. Sequence prediction addresses the prediction of the next element based on previous elements of the sequence. Sequence generation focuses on the generation of the next element of the sequence, given the previous ones. This kind of problem is essentially the same as sequence prediction. Sequence recognition attempts to validate a sequence. This problem can also be transformed into one of the previous types of problems. Finally, sequential decision making addresses the selection of actions to accomplish a goal or to follow a trajectory. These latest sequence learning problems are essentially equivalent to planning problems. Here I will concentrate on the three first types of sequence learning problems.

Sun and Giles (2001) also characterized sequence learning models according to several dimensions such as the learning paradigm and the implementation paradigm. For example, the learning paradigm might be supervised, unsupervised or reinforcement based, while the implementation paradigm might be a neural network, a lookup table, a deterministic or stochastic model, and so on.

The *degree* of a sequence element is the number of previous elements required to unequivocally determine this element. The sequence degree is the maximum degree of any of its elements (Lawrence et al., 2006; L. Wang, 2000). For example, ABCDEF has a sequence degree one, since each element uniquely determines the next and therefore all have element degree one. On the other hand, the sequence A**BC**M**BC**H requires at least three elements to determine the next one for some of its elements: ABC establishes M, and DBC yields H. Thus the sequence has degree three. Sequences can be classified as

simple if they have degree one or *complex* otherwise (Lawrence et al., 2006). Complex sequences markedly increase the difficulty of the algorithms and applications for sequence learning (Araujo & Barreto, 2002; Lawrence et al., 2006; L. Wang, 1998, 2000). When several sequences with elements in common are stored in the memory, problems similar to those of complex sequences can arise. For example, if sequences A**BCD**E and F**GCD**H are stored in the memory, at least three previous elements are necessary to disambiguate the retrieval of these sequences, even if each sequence is simple (Araujo & Barreto, 2002).

Sun and Giles (2001) also described the major sequence learning approaches: neural networks, temporal difference methods, explicit symbolic planning, inductive logic programming, hidden Markov models, and evolutionary computation. Temporal difference methods, which include reinforcement learning methods such as Q-learning, were extensively reviewed and compared with correlated neural networks for sequence learning by Wörgötter and Porr (2004). In this work, I will focus on neural networks and related models. Kremer (2001) comprehensively reviewed the research in this area.

Neural networks, especially recurrent backpropagation networks, are widely used for sequence learning, for example (Giles, Horne, & Lin, 1995). Associative networks were also studied for this task. For example, L. Wang (2000) proposed hetero-associative networks such as bidirectional associative memory (BAM) or associative memories (L. Wang, 1998). Several authors implemented extensions of the Hopfield network to store sequences (Maurer, Hersch, & Billard, 2005). D. Wang and Yuwono (1995, 1996) developed a model based on short-term memory, implemented with self-organizing neural networks, that is able to successfully handle complex sequences. Similar

approaches, using self-organizing networks can be found in (Araujo & Barreto, 2002; Barreto & Araujo, 2004; Somervuo, 1999). Using associative memories for sequence storage is a long studied subject. Wang and Yuwono (1996) also described the problems of using several types of neural networks to store sequences, including Hopfield and Willshaw networks. Stringer and colleagues (Stringer, Rolls, Trappenberg, & de Araujo, 2003) studied hetero-associative continuous attractor networks to solve path-integration. Lawrence et al. (2006) discussed the advantages of using a combination of hetero-associative and auto-associative memory for sequence learning; they also provided a good review of associative sequence models.

Several recent works, based on the hierarchical organization of the neocortex and visual cortex, focus on learning and recognition of spatial and temporal patterns. This approach, generally referred to as a *deep learning* system, combines hierarchical networks with pattern recognition using different technologies such as neural and Bayesian networks. The basic idea is to detect pattern invariances in space and (in some models) in time in each level of the hierarchy, and to use the output of the lower layer as input for the higher ones. Features and patterns learned at a higher layer are non-linear combinations of patterns learned in lower ones. The higher the layer, the more abstract are the features of the data that they capture. Examples of these hierarchical models are: the Hierarchical Temporal Network (George, 2008; Hawkins & Blakeslee, 2007), HMAX (Riesenhuber & Poggio, 1999; Serre, Wolf, Bileschi, Riesenhuber, & Poggio, 2007), deep belief networks (Hinton, 2007; Hinton, Osindero, & Teh, 2006), and DeSTIN (Arel, Rose, & Coop, 2009).

Several models that use SDM for sequence learning were described in Chapter 2. Bose et al. (2005) developed a memory that learns sequences based on a SDM implemented with spike neurons. Jockel (2009) created a multi-fold SDM that performs sequence learning for a robotic arm manipulation system. The next section describes in detail the procedures proposed by Kanerva to store sequences in SDM.

Storing Sequences in SDM

When storing sequences of vectors in SDM, the address cannot be the same as the word, as it is in the auto-associative case. The vector that represents the first element of the sequence is used as address to read the memory. The output vector is the second element in the sequence. This second vector is used as an 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 mechanism for 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 (2009) 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, three of them are relevant to the present discussion and will be summarized here: multiplication of binary vectors defined as bitwise XOR, permutation, and sum with normalization. For a complete discussion of hyperdimensional computing and its operations see Chapter 3.

Bitwise XOR is the multiplication operation of binary vectors in hyperdimensional computing. 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 to the two original ones. Second, the XOR operation is reversible. Third, this operation preserves Hamming distances. For example, if *A, B, C* are binary vectors, and

$$A' = (A\ XOR\ C)\ and\ B' = (B\ XOR\ C)\ then\ d(A, B) = d(A', B') \qquad (48)$$

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 a single one in each row and column while the other positions contain zero. 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 is used (i.e., the value 0 is replaced by -1). 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 one in this dimension. If the value is negative, the normalized vector has a zero in this dimension. Ties are resolved at random. The sum with normalization has attractive properties: the resulting vector is similar to each of the vectors summed up; that is, the distance between them is less than the expected distance between any two vectors in the space. Also, XOR multiplication and random permutations distribute over the sum. For example:

$$[\Pi(A) + \Pi(B)] = \Pi([A + B]) \qquad (49)$$

$$[(A\ XOR\ C) + (B\ XOR\ C)] = ([A + B])\ XOR\ C \tag{50}$$

where $\Pi(x)$ denotes a random permutation and […] is the normalization operation.

In light of these properties, it is sometimes possible to retrieve the individual added vectors from the sum vector. This is feasible only if the number of summed vectors is small (e.g., three or fewer vectors). Even with this small number, interference between the vectors makes retrieval of the original vectors from the sum not very reliable.

Kanerva describes how to store sequences of vectors using hyperdimensional arithmetic (Kanerva, 2009). I will briefly describe this procedure and compare it with my implementation in the section "Storing sequences and other data structures". The main problem with this procedure is that it uses the sum operation, and thus it shares the same problems mentioned above for sums while reconstructing the sequence. It also uses permutation, and as we discussed before, this operation requires matrices that are outside of the binary vector domain. Nevertheless, permutations are easy to implement, and a reduced number of different permutations are required to obtain the desired functionality.

Extended SDM

Here I present a novel structure, built upon SDM, called 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 10). Formally, in a word of length m and with an address with length n, the first n bits of the word compose the address.

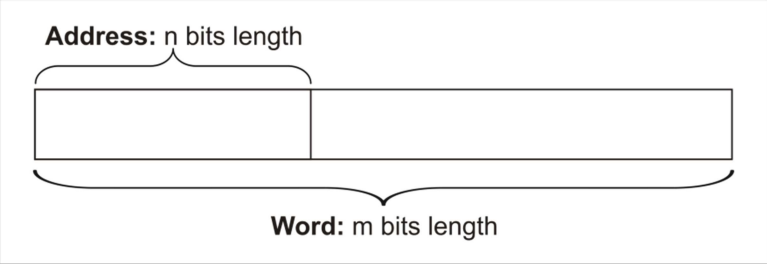


Figure 10. A word vector with its address section.

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 in Chapter 2, 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 a word vector of size *m*, and *M* is an *n* x *m* rectangular diagonal matrix with all ones 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 improve, the desirable characteristics of the SDM. First, with an initial vector as an 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 when some hard locations are corrupted or lost. Third, the previously discussed psychological characteristics of SDM are also present in ESDM. Finally, the hetero-associative part of the words in ESDM allows storing other data related to the address data but without interfering with it. This is a notable improvement over the original hetero-associative SDM that directly uses the current element as address of the next reading, preventing the use of iterations to retrieve the elements, and over the hyperdimensional version that relies on the flawed sum operation to achieve the same goal, but with far less effectiveness.

Lawrence et al. (2006) found similar conclusions with different associative memory architectures. They studied the advantages of using a combination of auto-associative and hetero-associative neural networks especially for sequence learning. In particular, they emphasized the importance of both the auto-associative and hetero-associative parts to achieve robust sequence memory. The auto-associative part provides noise robustness when cueing the memory with partial or noisy inputs, whereas the hetero-associative part points to the next element in the sequence.

Storing Sequences and Other Data Structures

In this chapter's introduction I mentioned two approaches suggested by Kanerva (1988, 2009) for storing sequences in SDM. I also mention that both approaches have important disadvantages that weaken the auto-associativity, content addressability and noise robustness properties of the memory.

The implementation of sequence storage in ESDM is straightforward and it eliminates the disadvantages mentioned. The most basic implementation uses addresses of length *n* and words of length 2*n*, as shown in Figure 11. 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 2*n*. 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.

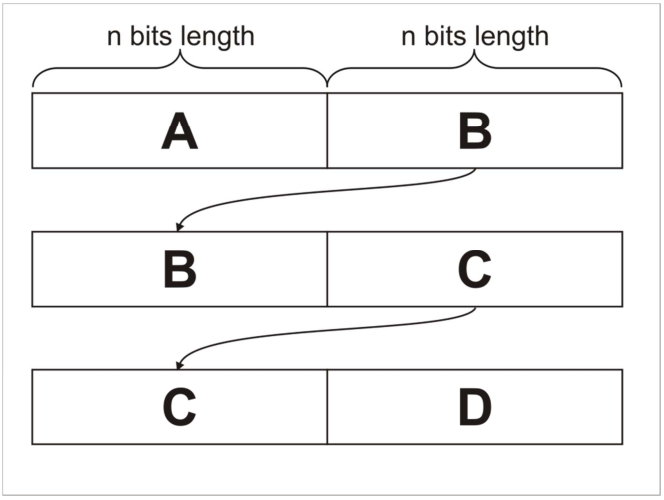


Figure 11. Basic sequence representation using 2*n* word vectors.

To retrieve the sequence, the initial vector of the sequence is used to read a word from the memory. This word is divided into two sections. The second section 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 an

address can have some noise, but the iterative 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 both of the sequences. One plausible solution is to use the same procedure proposed by Kanerva using hyperdimensional operations (Kanerva, 2009). 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. For example, if Π denotes a random permutation, then the address for the third element of the sequence is:

$$A_3 = \left[\Pi(E_1) + E_2\right] \tag{51}$$

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

$$A_{i+1} = \left[\Pi(A_i) + E_i\right] \tag{52}$$

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.

$$A'_{i+1} = k * \Pi(A'_i) + E_i \tag{53}$$

$$A_{i+1} = [A'_{i+1}] \tag{54}$$

where A' is the real vector with the sum before normalization.

The introduction of the scalar *k* has another critical function. The normalization required after the sum introduces excessive noise that diminishes the probability of recovering the sequence. The scalar *k* mitigates this effect. See the simulations section below for a discussion of this subject.

The equations (51), (52), (53) and (54) can be used in the original SDM, as suggested by Kanerva (2009). 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 output word.

Other data structures can be stored in ESDM in a similar way. For example, to store binary trees, addresses of length *n* and words of length 3*n* 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 mechanism similar 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 by 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.

Simulations and Experiments

For simulation and testing of the ESDM, I implemented several versions of the memory. One of them uses a database for the main storage of the hard locations, and a RAM cache to speed up storage and retrieval operations. This allows us to create large ESDMs, with millions of hard locations and word dimensions on the order of 1,000 or even 10,000 bits, even using modest computers. Another version implements the actor model for parallel and distributed execution. Finally, a GPU implementation runs in SIMD architecture with a notable performance gain. For more implementation details, see Chapter 7.

Several simulations were performed with the ESDM. First, the capacity and noise robustness of the extra bits of the words were compared with these same characteristics in the standard SDM. Second, the sequence storage and retrieval were tested for several

values of *k*. Third, retrieving sequences from intermediate elements was analyzed. Finally, experiments that test the retrieval of crossing sequences that have common elements were performed. In this section I present and discuss the details and results of these simulations.

ESDM Capacity and Noise Robustness

These simulations test the capacity of the memory and its noise robustness. Kanerva (1988) proved that the critical distance of SDM is a function of the number of words stored in the memory. He also proved that the maximum capacity of the memory is reached when the critical distance reaches zero, which is approximately equal to 10% of the number of hard locations for a memory with vectors of 1,000 dimensions. After this number it becomes impossible to retrieve a stored vector even when cueing the memory with the same vector. For a complete analysis of SDM capacity see (Chou, 1989; Kanerva, 1988; Keeler, 1988). Reading from ESDM is essentially the same as from SDM, except for discarding the extra bits of the word. Hence, convergence during a read in ESDM is the same as in SDM, and the critical distance and capacity are also similar to those of SDM. However, we need to show that the percentage of errors (changed bits) in the words read from ESDM is similar to the percentage of errors in the words read from standard SDM. If only the address part of the vectors stored in ESDM is used, the memory is equivalent to standard SDM, so the error comparison was performed between the address part and the whole word of the same simulation.

Several simulations were performed to test the percentage of errors in the output words. An ESDM with 200,000 hard locations, an address length of 1,000 dimensions and a word length of 2,000 dimensions (including the address) was used for the

simulations. The size of the memory, determined by the number of hard locations, was chosen to have enough hard locations in the access sphere for each read or write to support the desired properties of the ESDM, but to be as small as possible to limit the number of reads and writes required to perceive the effects of loading the memory. The size of the vectors was chosen to match those used by Kanerva (1988). For this particular simulation, a total of 10,000 random vectors were stored in the ESDM, which is roughly half of the memory capacity.

The storing of vectors in the memory was done in stages, writing 1,000 vectors in each stage. At the end of each stage, the vectors were read from the memory. For the readings, 10% of the bits of each vector address were changed randomly, and these noisy vectors were used as cues. Figure 12 and Table 1 show the results of this simulation.

An analysis of the retrieved vectors shows that the proportion of errors for the word and the address is constant and roughly proportional to the difference in size. This shows that using words that are longer than addresses does not affect the fidelity of the memory. Also, the percentage of retrieved vectors is consistent with the diminishing of the critical distance as more vectors are stored in the memory (Kanerva, 1988).

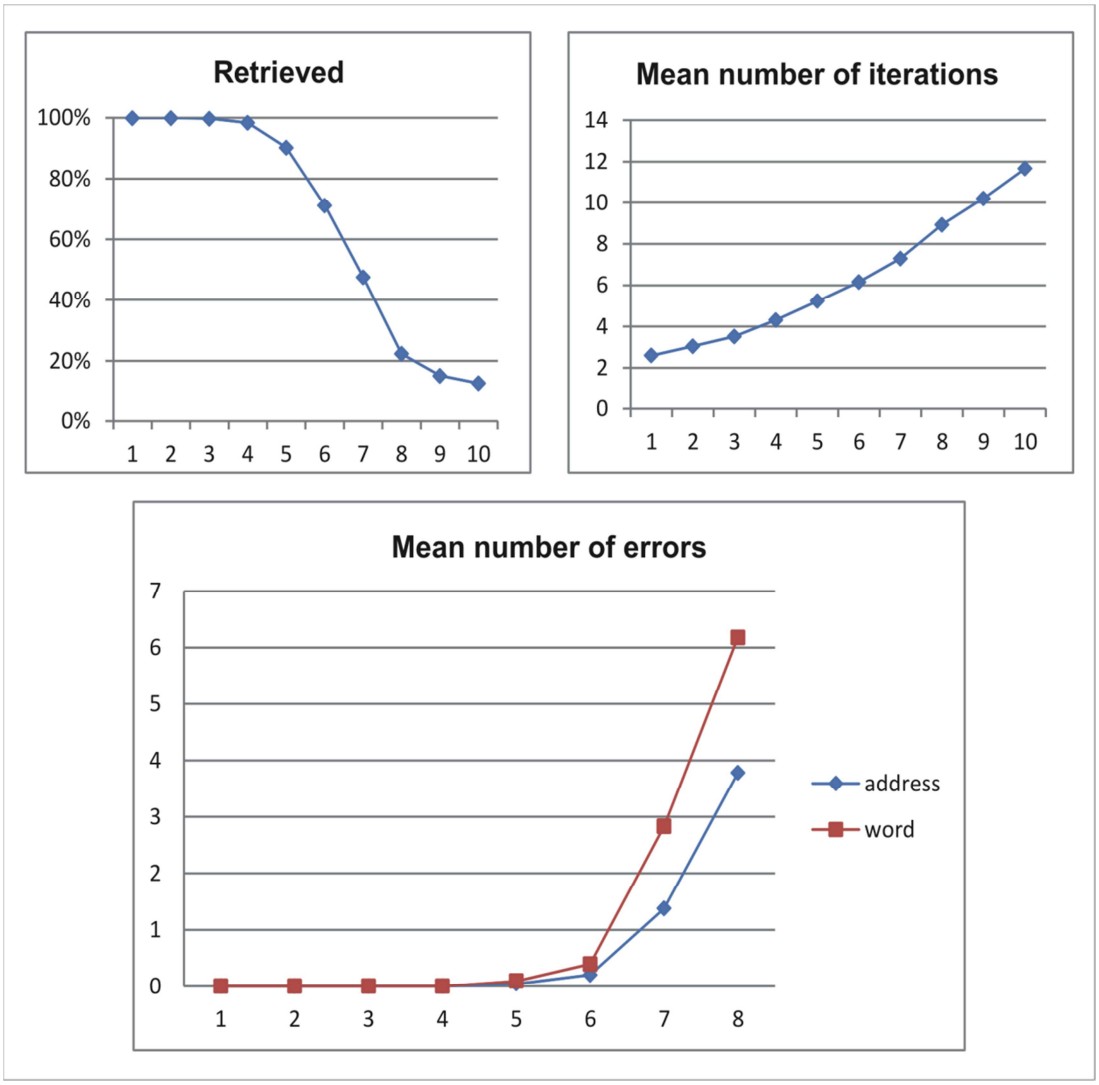


Figure 12. The percentage of retrieved vectors in each stage, the mean number of iterations required in each stage, and the number of errors (changed bits) in the address part and the whole word of the retrieved vectors in each stage.

Table 1

Simulation 1. ESDM capacity and noise robustness. In each stage, 1,000 vectors were stored. Then the same vectors were retrieved adding 10% noise to the cue (address). The number of iterations and the mean error are given for the retrieved vectors. The address part is equivalent to the standard SDM result.

Stage	Retrieved (%)	Iterations		Error mean	
		Mean	*SD*	Address	Word
1	100.00	2.59	0.49	0.00	0.00
2	100.00	3.04	0.24	0.00	0.00
3	99.80	3.51	0.59	0.00	0.00
4	98.40	4.31	0.90	0.00	0.00
5	90.30	5.23	1.25	0.04	0.09
6	71.20	6.16	1.41	0.20	0.39
7	47.60	7.30	1.62	1.37	2.83
8	22.30	8.24	1.58	3.78	6.18
9	15.00	9.50	1.83	1.15	1.60
10	12.60	11.09	3.34	1.54	2.47

Another simulation was performed to show the noise robustness of ESDM. The same ESDM was used as for the previous simulation, with 10,000 vectors already stored in the memory. The vectors were also preserved in a separate database so that they could be used as cues or compared with the retrievals from the ESDM. The simulation was performed in three stages. In each stage, one thousand vectors were randomly selected from the set of stored vectors, and the memory was read using the address part of these vectors with a variable amount of noise. The noise levels were as follows: 0% in the first stage, 5% in the second, and 10% in the third. Table 2 summarizes the results of this simulation.

Table 2
Simulation 2. ESDM capacity and noise robustness. In each stage, 1,000 vectors were retrieved from an ESDM with 10,000 stored vectors, and a variable amount of noise was added to the cue (address). The number of errors in the successfully retrieved vectors represents the average number of bits changed in each vector.

Stage	Noise (%)	Retrieved (%)	Error mean
1	0	100.00	0.286
2	5	97.00	4.784
3	10	14.80	2.439

The results of the experiments suggest a good performance of the memory: the number of successful retrievals was high with low levels of noise, and the error (number of changed bits in the retrieval) was very small, less than a bit on average. Even more, 93.3% of the vectors had zero errors in stage 1 and 79% of the retrievals in stage two had fewer than five errors. As expected, the number of retrieved vectors diminished when the vectors used as cues reach the critical distance. Notice that the critical distance is the distance at which the probability of convergence to the stored value is 50%. The critical distance is a function of the number of hard locations and the number of stored vectors in the memory. For the ESDM used in this experiment, with a load of 50% of its capacity, distances of 100 bits (10% of the address length) from the original vectors are beyond the critical distance. See Kanerva (1988) for details.

Sequences

I performed several simulations to test sequences stored in ESDM. In each simulation, 50 or 100 sequences of 20 elements each were stored. As in the previous simulations, ESDM memories with 200,000 hard locations, an address length of 1,000 dimensions and a word length of 2,000 dimensions (including the address) were used for these simulations. A

new ESDM with a memory load between 5% to 10% of the memory capacity was used for each simulation. This prevented interference among stored vectors. I considered a sequence successfully retrieved if all elements were retrieved with a small amount of noise (less than 5%).

The first simulation stored and successfully retrieved 49 out of 50 sequences; however, the same approach failed to retrieve a single sequence in a run with 100 sequences. Interference produced by memory load, 10% in this case, does not suffice to explain this result. Rather, the normalization after the sum in equation (52) enables an effect that distorts the address. The sum has only two binary vectors as operands in the address calculation. When the two operands differ in the value of a single dimension, the algorithm randomizes this dimension's value. In the average case when using a random uniform distribution of vectors, excessive noise in 50% of the bits prevents successful retrieval of the element.

To avoid this problem, equations (53) and (54) were used. Since one of the operands has a smaller weight than the other, the sum has no undetermined dimensions, and the problem disappears. In a simulation where 100 sequences were stored using equations (53) and (54) with $k = 0.8$, all the sequences were restored without error.

The use of the parameter k has other interesting consequences due to the fact that the weight of the previous elements diminishes as the sequence advances. It is possible to "step into" the sequence in the middle. However, more than one element may be required for the cue. For smaller values of k, fewer elements are required as part of the cue to step into the sequence. Conversely, if two (or more) sequences have common elements, the

probability of retrieving the correct sequence increases as *k* approaches one. The value of *k* is then a tradeoff between these two desirable properties.

Several simulations with different values of *k* were performed. First, the "step into" property was tested. Three simulations with values of *k* equal to 0.7, 0.8 and 0.9 respectively were performed. One hundred sequences with 20 elements each were stored in each simulation. Then, 10 of the stored sequences were chosen, and for the elements of these sequences, the number of cue elements required to be able to step into the sequence at that element was evaluated. To avoid transitory effects, only elements after the fifth were used as points to step into. Table 3 shows the results of these simulations.

Table 3
Effect of k on stepping into the sequence. In each stage, the simulation evaluated the number of cue elements required to step into the sequence at different points.

Stage	*k*	Required Elements	
		Mean	*SD*
1	0.7	1.085	0.280
2	0.8	2.697	0.679
3	0.9	6.000	1.265

As expected, the number of required elements in the cue increases as *k* increases. The best value of *k* depends on the degree of the sequences that memory stores. The higher the required degree, the higher must be the value of *k*.

Another series of simulations was performed to evaluate the retrieval of sequences with common elements, that is, sequences that intersect. Four simulations with values of *k* between 0.9 and 0.6 respectively were performed. Ten pairs of sequences with 20 elements each were stored in each simulation. The sequences in each pair had a common element. In every case, the intersection was after the fourth element in the

sequences. A number of random vectors were stored in the memory so as to achieve a load of 10% of the capacity of the memory.

Each of these sequences was then retrieved from the memory, and the number of successfully recovered sequences noted. With all of these values of *k*, all sequences were successfully retrieved. This result shows that the feature of correctly retrieving intersecting sequences is invariant over the value of *k*. However, equations (53) and (54) suggest that if two sequences have more than one consecutive element in common, higher values of *k* will perform better.

Notice that when *k* is equal to or less than 0.5, the first term in equation (53) is always less than one and it does not contribute to the final value after normalization in equation (54). As a consequence, the next address is only a function of the previous element, so that most elements after the intersecting element are not able to be retrieved. This is because of the interference produced by the common element.

Comparing the results of the last two groups of simulations, a balance between the two characteristics, step into and crossing of sequences is achieved with a value of *k* between 0.6 and 0.8. Of course, the selection of the value of *k* depends on the requirements of the application of the ESDM.

Long Sequences

A series of experiments further demonstrates the capacity of this memory for sequence storage. Using an Extended SDM with 1,000,000 hard locations, an address length of 1,000 dimensions, and a word length of 2,000 dimensions, 50 sequences with 100 random elements each were stored in the memory using a parameter *k* equal to 0.8. Then, the sequences were retrieved adding 10% noise to the cue vectors. All sequences were

recovered from the memory without any error. I performed the same experiment with 100 sequences using a similar memory configuration, obtaining the same result. Another experiment stored 10 sequences of 1,000 elements each in a memory with identical configuration. As in the previous experiments, all sequences were retrieved without errors when the memory was read after adding 10% noise to the cue vectors.

Each of these experiments utilizes a number of vectors that is approximately 10% of the theoretical memory capacity. If the number of sequences increases, the performance would diminish. Nevertheless, this possible decrease in performance would be due to the capacity limit and not because of the sequence storage mechanism.

Another experiment demonstrates the crossing sequence learning capability of the memory for long sequences. Using a predefined set of vectors as an *alphabet*, 10 sequences with 100 elements (each of them chosen from the alphabet) were stored in the memory. The results varied depending on the alphabet's size and the parameter *k*. Using a parameter *k* equal to 0.7 and an alphabet of 20 elements, no sequence was retrieved correctly. On the other hand, using $k = 0.9$ and alphabet with 40 vectors, every sequence was retrieved almost without errors. Only 8 out of the 1,000 elements that composed the 10 sequences presented errors. Finally, the same experiment with $k = 0.9$ and 20 elements in the alphabet had an intermediate result. Only 16 of the retrieved vectors resulted in more than 10% of errors, and 962 vectors had less than 1% of bits with errors. These results are consistent with the expected interference among similar vectors when the alphabet is small, which produce a large number of the crossings between the sequences.

Summing up, these experiments demonstrate that the capabilities of the sequence learning mechanism are preserved even when long sequences are used. The mechanism's

performance degrades when the total number of vectors approaches the memory's maximum capacity, or when the size of the alphabet of possible vectors to construct the sequences is small, which produces more interference among the vectors.

Conclusions

Here I have presented an extension of the original SDM that addresses several of its difficulties with storing compound data structures like sequences, trees and records. ESDM preserves the desirable, biologically inspired, properties of the original. It is also still 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 simulations successfully tested the performance of the ESDM in several scenarios. The importance of the parameter *k* was shown not only for the storage of simple sequences but also for enhancing performance when stepping into in the middle of sequences, and for enabling accurate retrieval in the case of common elements in different sequences.

ESDM is compatible with other improvements already studied, such as the introduction of the "don't care" symbol (D'Mello et al., 2005; Ramamurthy et al., 2004), or the forgetting mechanism (Ramamurthy, D'Mello et al., 2006; Ramamurthy & Franklin, 2011). Incorporating this forgetting mechanism is a natural direction for further development of this architecture. Other possible variations of ESDM already studied for SDM include dynamic allocation (Ratitch & Precup, 2004) of hard locations and distribution of hard locations according to the data (Anwar et al., 1999; Fan & Wang, 1997).

Chapter 5: Integer Sparse Distributed Memory

Sparse distributed memory (SDM) (Kanerva, 1988) is based on large binary vectors, and has several desirable properties. It is distributed, auto-associative, content addressable, and noise robust. For details see Chapter 2.

The original SDM uses binary vectors for both addresses and data words. This usage results in several limitations. First, real data are not always Boolean, making representations using more than two values desirable. A possible solution for this limitation is to use several dimensions of the word vectors to represent one feature, but this approach does not fit very well with the structure of SDM. In the distance calculation, a difference in any dimension has the same weight as that of any other dimension, but if several bits (i.e., dimensions) are used to represent a single feature, the weight of the bits should not be the same.

Mendes and colleagues (2009) evaluated several binary encodings to use with SDM in robot navigation tasks, and reported their difficulties and limitations. Using binary numbers coding some transitions have Hamming distances that incorrectly reflect the difference between the features. For example, the Hamming distance between seven (0111) and eight (1000) is 4 instead of the desired distance of 1.

They also reported the performance of the Gray code, which only partially mitigates this effect. The best solution that they proposed is to use a sum code, in which, for example, 3 is represented as 111 and 5 as 11111. This coding substantially increases the dimensionality of the memory. Interestingly, they report that grouping bits and processing them as integers produces excellent performance. However, their implementation diminishes some of the desirable properties of SDM. The extension

proposed in this paper directly uses integer vectors, achieving similar performance but without the disadvantages reported by Mendes.

Another disadvantage of binary vectors is the loss of information due to the noise introduced into the representation by the normalization used in combining vectors. Vectors can be summed up dimension by dimension (for this operation, vectors belonging to $\{0; 1\}^n$ are replaced by vectors of $\{-1; +1\}^n$). This operation produces a vector belonging to $\mathbb{Z}^n$. The normalization process reduces the resultant to a vector that is also in $\{-1, 1\}^n$ but with significant loss of information. See for example (Kanerva, 2009; Snaider & Franklin, 2011; Snaider & Franklin, 2012a). I extensively discussed this issue in Chapter 3.

Here I introduce a new version of SDM, the Integer Sparse Distributed Memory (Integer SDM) (Snaider & Franklin, 2012b). This version is based on large vectors, on the order of thousands of dimensions, where each dimension has a range of possible integer values. This memory has properties similar to the original SDM noise robustness, auto-associativity, and being distributed. A further extension of Integer SDM permits words and addresses of different lengths, which is particularly useful for the reliable storage of sequences and other data structures (see Chapter 4). In addition, this memory avoids the limitations imposed by binary representation, as described above, allowing a better encoding of non-binary data and alleviating the normalization problem when combining several vectors. This memory also fits the requirements of the Modular Composite Representation to be introduced in Chapter 6.

Integer Sparse Distributed Memory

The structure and operations of Integer SDM are similar to that of SDM (see Chapter 2). However, the words and addresses used by Integer SDM are large vectors of integers rather than binary vectors. The possible values for each dimension are in a defined integer range. For example, the range of values can be {-8, 7}, {0, 15}, or any other range. However, for simplicity, we will work with ranges with 0 as the lower bound and $r - 1$ as the upper bound. Although there is no theoretical limit to the size of the range, the storage requirement of the memory increases proportionally with the range's size. More formally, Integer SDM works within a multidimensional space with vectors $v \in \mathbb{Z}_r^n$, where n is the number of dimensions of the space and r is the size of the range of values for each dimension. The dimensions of the space follow modular arithmetic: the greatest possible value for a dimension is $r - 1$, and the next value after $r - 1$ is 0.

Integer SDM is composed of hard locations. As in SDM, a small fraction of all possible addresses $a \in \mathbb{Z}_r^n$ are chosen at random (with equal probability) as addresses for the hard locations. Each hard location has a fixed address and counters, resembling the structure of SDM. However, hard locations in Integer SDM have a different arrangement of counters: each dimension has r counters, one for each possible value in that dimension (see Figure 13). I define C_i as the group of counters corresponding to the dimension i, and $\boldsymbol{C}_i^v$ as the counter corresponding to dimension i and value $v \in \{0, r-1\}$.

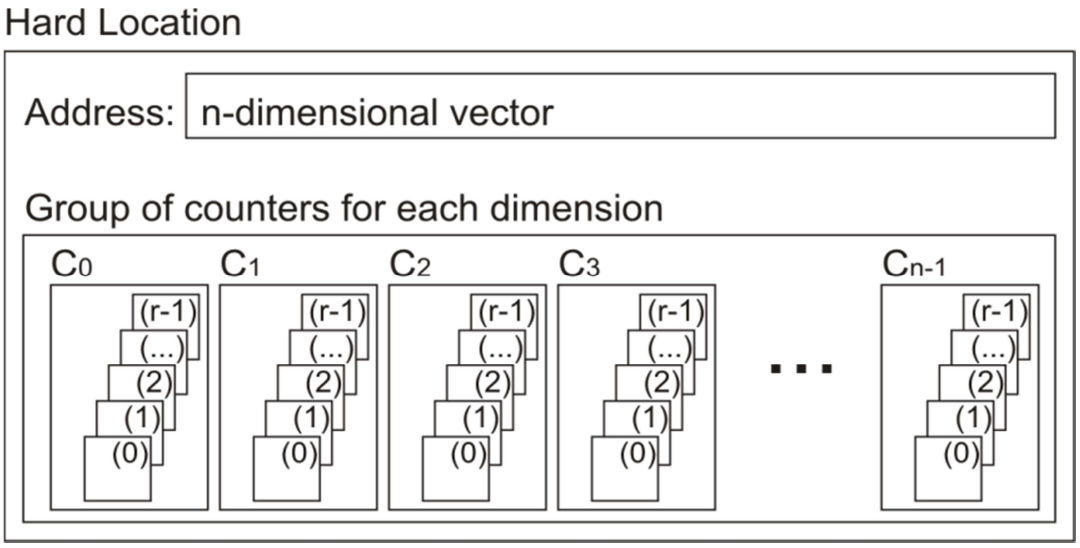


Figure 13. Structure of an Integer SDM hard location. Each hard location has an address that is an *n*-dimensional vector belonging to $\mathbb{Z}_{\boldsymbol{r}}^{\boldsymbol{n}}$, and counters for storing data. The counters are organized into groups. There is a group of counters for each dimension of the vector space of words, *n* in this example. Each group has *r* counters, one for each of the possible values in each dimension of the word vectors.

To read or write a word *w*, first the access sphere of the address is determined. Any similarity measure for vectors in the space can be used as distance, including any norm, but the measure need not define a metric on the space.

The distance used here is an extension of the Euclidean or Manhattan metric. The distance between two vectors is defined as:

$$d(u, v) = \sqrt{\sum_i (\Delta_i)^2} \tag{55}$$

for the extended Euclidean metric, and:

$$d(u, v) = \sum_i \Delta_i \tag{56}$$

for the extended Manhattan distance, where:

$$\Delta_i = \min\left(mod_r(u_i - v_i), mod_r(v_i - u_i)\right). \tag{57}$$

Since each dimension in the space follows modular arithmetic, each dimension *i* is like a circle with two possible *paths* between the values u_i and v_i. Notice that Δ_i is the shorter of the two.

The geometric interpretation of this space is on the surface of a hypersphere, and the variation of the Euclidian distance is equivalent to the distance between two points on the surface of the hypersphere. See Figure 14.

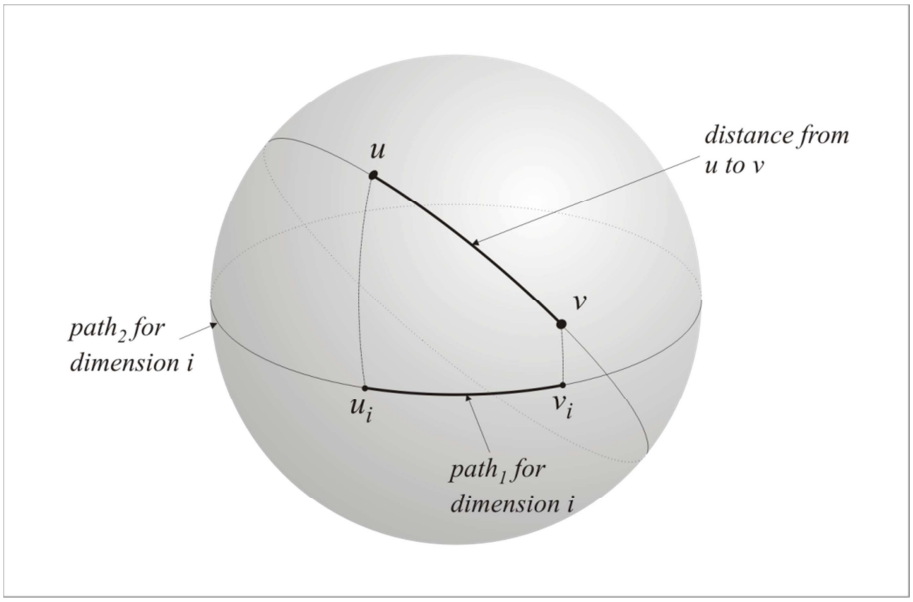


Figure 14. Euclidean distance from *u* to *v* on the surface of a sphere. For the distance calculation, when projecting onto dimension *i*, there are two possible *paths.* The shortest one ($path_1$ in this example) is used for calculating the distance.

The radius of the access sphere is defined in such a way that on average it encloses a small proportion *p* of the total number of hard locations. If *m* is the number of hard locations in the memory, the access sphere encloses *pm* hard locations. This value *p* is also the probability of activation of one hard location, that is, the probability that one hard location participates in a particular reading or writing operation. Since the hard locations are uniformly distributed in the space, the probability *p* unambiguously determines the radius of the access sphere. An *activated* hard location with respect to a given operation is one that participates in a specified reading or writing operation. Figure 15 illustrates the structure of the Integer SDM.

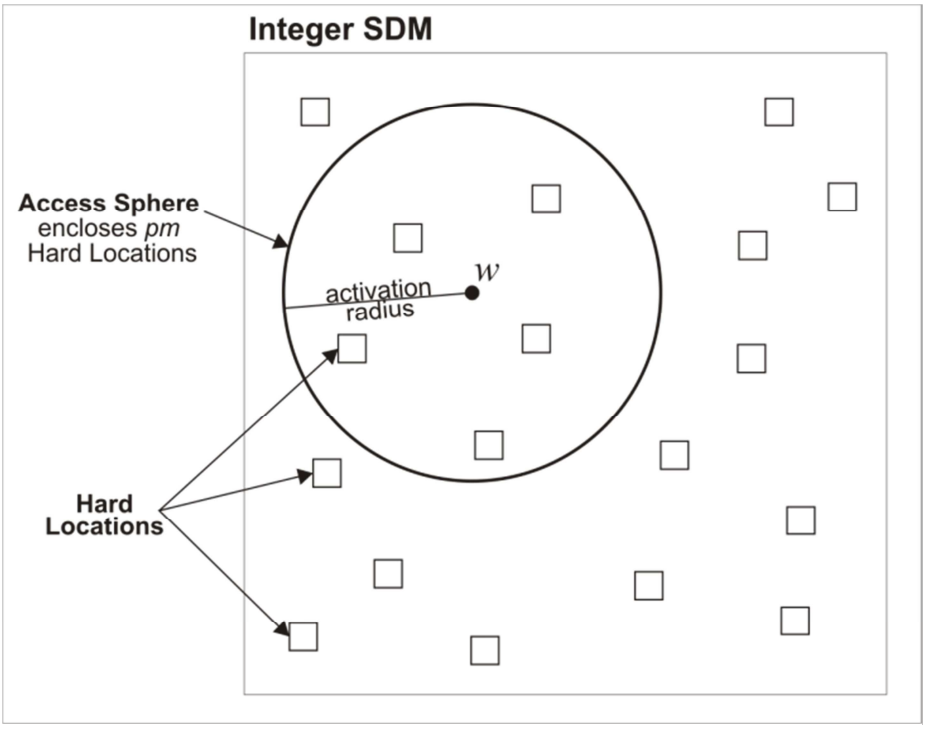


Figure 15. Integer SDM structure. The addresses of hard locations are uniformly distributed in the space of $\mathbb{Z}_r^n$. The access sphere of *w* encloses *pm* hard locations. These *pm* hard locations are active when *w* is read or written.

For writing the word *w* in the memory, the counters of each hard location in the access sphere are updated using the following rule:

$$C_i^v \; is \; incremented \iff v = w_i$$

where w_i is the value of the dimension *i* of the word *w*. Notice that only one of the *r* counters in each dimension is incremented for a given hard location; this process is repeated for each hard location in the access sphere.

Reading from the memory begins by determining the hard locations in the access sphere in the same way as when writing. Then the counters corresponding to each of the *r* values in each dimension are summed up over all hard locations in the access sphere:

$$S_i^v = \sum_{\substack{HL \in \\ Access \\ Sphere}} C_i^v \tag{58}$$

where S_i^v is the sum of the counters for dimension *i* and value *v*.

Finally, for each dimension a majority rule is applied among the values, and the value *v* corresponding to the maximum S_i^v is assigned to z_i, the value of the *i-th* dimension of the output vector.

$$z_i = index(v) \; of \; max\left(S_i^0 .. S_i^{r-1}\right) \tag{59}$$

where z_i is the value of *i* dimension of the output vector. This vector *z* can be used as an address to read from the memory again, iterating in the same way as in the original SDM. See Chapter 2 for details.

The complexity of the reading (or writing) operation of the memory is *O*(*mn* + *prmn*). The first term corresponds to the calculation of the distance from *w* to each hard

location, and the second term corresponds to the reading (or writing) of the counters in the hard locations. Since in general $pr << 1$, the first term dominates. When the number of hard locations m is too large, the implementation is likely to be slow. However, the algorithm is easily parallelizable to be executed in multithreading or SIMD architectures (e.g., using GPUs). Moreover, other methods for activating the hard locations have been studied for SDM; these can be adapted for Integer SDM also. See for example (Jaeckel, 1989a, 1989b; Karlsson, 1995). These alternatives would greatly reduce the time complexity of the algorithm.

Radius of the Access Sphere

Here I will analyze the calculation of the access sphere radius that corresponds to a particular value of p when the variant of the Manhattan distance is used. In this section the term distance refers to the variant of the Manhattan distance introduced in the previous section. To calculate the radius of the access sphere as a function of p, we need the distribution of the distances from a given point to all the other points in the space. Since the space is symmetrical, any point is equivalent to any other one. For notational simplicity, we will calculate the distribution with respect to the origin (the vector with each dimension equal to 0). In Chapter 6, I will give a proof for the following approximation to this distribution for the case when r is even. The result is similar, but not exactly the same, when r is odd.

If the dimensions of all vectors are independent and uniformly distributed in $\{0, r-1\}$ and r is even, then the distribution of Manhattan distances from a given vector to the rest of the vectors of the space can be approximated by a normal distribution with parameters:

$$D \sim N\left(\frac{nr}{4}, \frac{n\left(r^2+8\right)}{48}\right) \tag{60}$$

With this distribution we can calculate the radius of the access sphere; it is simply the value of the distance that encloses a proportion p of the space:

$$radius \approx \sqrt{\frac{n\left(r^2+8\right)}{48}}\ \Phi^{-1}(p) + \frac{nr}{4} \tag{61}$$

where Φ^{-1} is the inverse of the normal distribution function. For example, with n = 1,000, r = 16, and p = 0.001 the radius of the access sphere is approximately 3,771.

Fidelity and Capacity

The fidelity of this memory–the probability of retrieving a written word–is better than the fidelity of the original SDM with the same number of hard locations and the same number of stored words. This improvement is due to more precise storage in each hard location. Since each dimension is independent of the others, we can choose any dimension to analyze φ, the fidelity of one of the dimension; the result will be the same for all other dimensions. For convenience, we select dimension 0. Suppose the stored value for dimension 0 of word w is k, or $w_0 = k$. To read w_0 incorrectly from memory, at least one of the sums S_0^v for the incorrect values ($v \neq k$), must be greater than S_0^k. The value of the sums for incorrect values is due to the contribution of other words written in the memory that share some of the same hard locations used to store w. Assuming the

other words written in the memory are uniformly distributed in the space[1], the noise produced by the interference of these written words is distributed in *r* possible values. This diminishes the expected value and variance of the $S_0^v \; for \; v \neq k$. Then the probability of having $M_{S^v} = max\left(S_0^v | v \neq k\right) > \; S_0^k$ is less than in the original SDM for the same number of words stored in the memory. This increment in the fidelity of the memory also increments its capacity: more words can be stored before the effect of interference is noticed. This compensates for the additional requirements of memory storage to implement the counters of this memory as compared to the original SDM.

The theorem at the end of this section derives the following approximate formula for φ the fidelity of this memory.

$$\varphi = \int_{-\infty}^{\infty} \phi\left(\frac{u-\lambda_k}{\sqrt{\lambda_k}}\right) \Phi\left(\frac{u-\lambda_v}{\sqrt{\lambda_v}}\right)^{r-1} \; du \tag{62}$$

where

$$\lambda_v = \frac{mtp^2}{r} \text{ and } \lambda_k = mp + \frac{mtp^2}{r} \tag{63}$$

The value *t* is the number of vectors stored in the memory.

Figure 16 depicts the probability density functions (pdf) of S_0^v, M_{S^v}, and S_0^k when one of the vectors is recalled, for an Integer SDM with 1,000,000 hard locations, *r* = 16, *p* = 0.001, and *t* = 400,000. The fidelity φ of one dimension is the probability that $M_{S^v} > S_0^k$. In this example φ = 0.99993.

[1]This assumption is reasonable for the purpose of estimating the capacity of the memory. However, the memory can store vectors even if its hard locations are not uniformly distributed, but the capacity may be diminished. See Kanerva (1993) for a similar analysis for SDM.

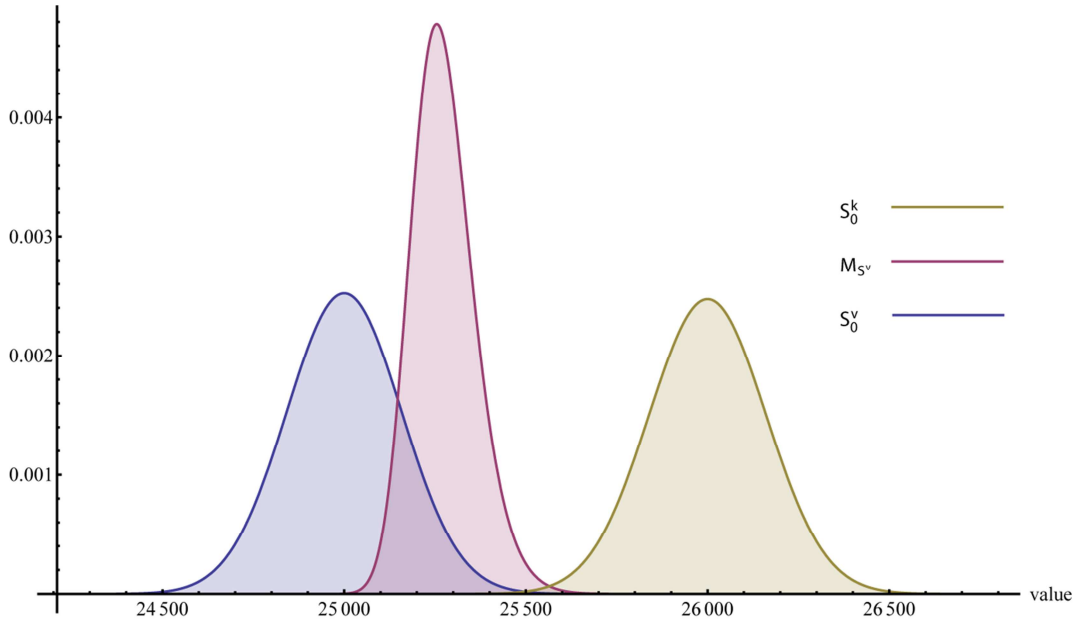


Figure 16. Pdf's of S_0^v, M_{S^v}, and S_0^k for a Integer SDM with 1,000,000 hard locations, r = 16, p = 0.001, and t = 400,000.

Figure 17 shows φ, the probability that one dimension is retrieved correctly, as a function of t, the number of stored vectors. If $t \approx 550{,}000$ then $\varphi = 0.999$. A standard SDM with the same number of hard locations will reach this same fidelity after storing about 105,000 vectors (Kanerva, 1993).

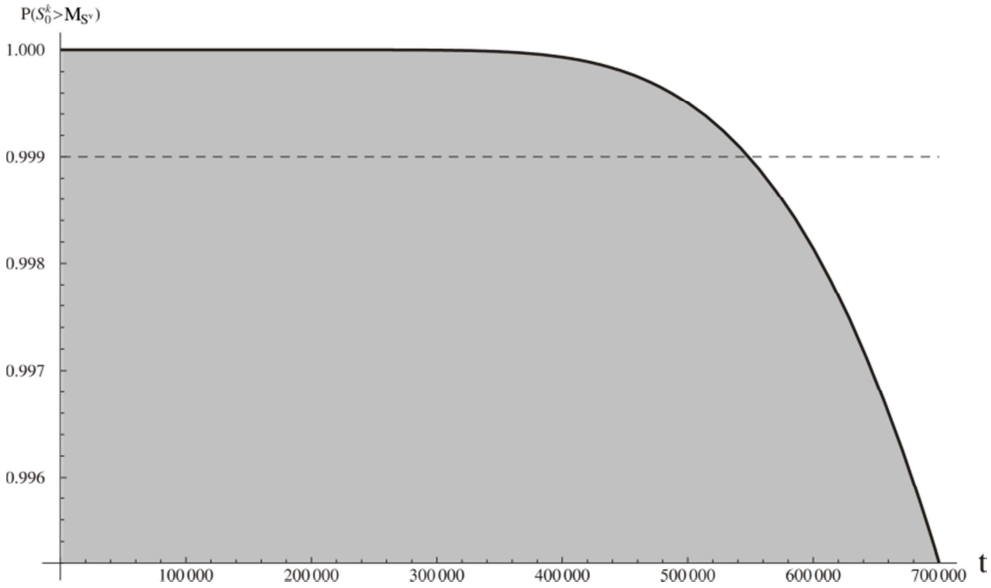


Figure 17. Fidelity of one dimension as a function of *t*, the number of vectors stored in the memory. For a Integer SDM with 1,000,000 hard locations, $r = 16$, $p = 0.001$, and $t \approx 550{,}000$ the fidelity is $\varphi = 0.999$.

Theorem: The fidelity φ of one dimension, which is the probability of retrieving a dimension correctly, can be approximated by:

$$\varphi = \int_{-\infty}^{\infty} \phi\left(\frac{u-\lambda_k}{\sqrt{\lambda_k}}\right) \Phi\left(\frac{u-\lambda_v}{\sqrt{\lambda_v}}\right)^{r-1} du \tag{64}$$

where

$$\lambda_v = \frac{mtp^2}{r} \text{ and } \lambda_k = mp + \frac{mtp^2}{r} \tag{65}$$

Proof: We will write into the memory a vector *w* and a set T of vectors. All these vectors are uniformly distributed in the space. We will use *t* to denote the size of the set T. Then, we will read from the memory in the address *w* retrieving *w'*, and we will calculate the

probability of *w'=w*. Since all the dimensions of *w* are independent (the same is true for all vectors in T), we can analyze the fidelity φ of dimension 0, that is the probability of correctly retrieving the value for the dimension 0 (i.e., $w_0' = w_0$), and use this to calculate the probability of correctly retrieving *w*.

$$\varphi = P\left[w_0' = w_0\right] \tag{66}$$

$$P[w' = w] = \varphi^n \tag{67}$$

where *n* is the number of dimensions of *w*.

Suppose that $w_0 = k$, and remember from above that:

$$w_0' = index(v) of \max\left(S_0^{(0)} .. S_0^{(r-1)}\right) \tag{68}$$

When we read *w*, *w*e want S_0^k, the sum of counters corresponding to dimension 0 and value *k*, to be greater than M_{S^v}, the maximum of all the other sums corresponding to dimension 0 and values different than *k*. In other words,

$$M_{S^v} = \max\left(S_0^v | v \neq k\right), \tag{69}$$

and in order to recall the correct value *k* of w_0', we need $M_{S^v} < S_0^k$. If we define $G = \left(M_{S^v} - S_0^k\right)$, then

$$\varphi = P[G < 0] \tag{70}$$

I will first analyze M_{S^v}. Consider the hard locations that are activated when *w* is written or read. Since *p* is the probability of activation of a hard location during one

reading or writing operation, and the vectors in T are independent of *w*, the probability of activation of a hard location in the access spheres of both *w* and a vector in T is p^2. The distribution of the values of the counter $C_0^v | v \neq k, v \in \{0, r-1\}$ for a hard location activated in the write operation for one of the vectors in T has a Bernoulli distribution with probability $p_v = \frac{p^2}{r}$. Then, for the *t* writes of the vectors in T, the distribution of $C_0^v | v \neq k$ for any hard location has a Binomial distribution:

$$C_0^v \sim B\left(t, \frac{p^2}{r}\right) \tag{71}$$

We will have *r* - 1 counters, corresponding to an *incorrect* value in dimension 0, for each hard location in the access sphere of *w* with the Binomial distribution defined as in (71). The sum of these counters for all hard locations in the access sphere of *w* (when we read *w*) is:

$$S_0^v \sim B\left(mt, \frac{p^2}{r}\right) \tag{72}$$

The probability mass function (pmf) of this sum is:

$$P\left\{S_0^v = x\right\} = f_0^v(x) = \binom{mt}{x}\left(\frac{p^2}{r}\right)^x\left(1 - \frac{p^2}{r}\right)^{mt-x} \tag{73}$$

and the cumulative distribution function (cdf) is:

$$P\left\{S_0^v \leq x\right\} = F_0^v(x) = \sum_{i=0}^{x}\binom{mt}{i}\left(\frac{p^2}{r}\right)^i\left(1 - \frac{p^2}{r}\right)^{mt-i} \tag{74}$$

Note that:

$$f_0^v(x) = F_0^v(x) - F_0^v(x-1) \tag{75}$$

For the n[th] order statistic, the cdf of the maximum of *n iid* random variables $X_i \sim F(x)$ is:

$$F_{max}(x) = [F(x)]^n \tag{76}$$

and its pmf is:

$$f_{max}(x) = [F(x)]^n - [F(x-1)]^n \tag{77}$$

In our case, we have $r-1$ random variables S_0^v, each of which has a cdf defined as in equation (74), so:

$$F_{max}(x) = \left[\sum_{i=0}^{x} \binom{mt}{i} \left(\frac{p^2}{r}\right)^i \left(1-\frac{p^2}{r}\right)^{mt-i}\right]^{r-1} \tag{78}$$

and, $f_{max}(x)$ can be calculated with:

$$f_{max}(x) = (r-1)\left\{\binom{mt}{x}\left(\frac{p^2}{r}\right)^x \left(1-\frac{p^2}{r}\right)^{mt-x} \left[\sum_{i=0}^{x} \binom{mt}{i}\left(\frac{p^2}{r}\right)^i \left(1-\frac{p^2}{r}\right)^{mt-i}\right]^{r-2}\right\} \tag{79}$$

S_0^k, corresponds to the sum of the counter for the *correct* value for dimension 0 when reading from address *w*. We can express S_0^k as:

$$S_0^k = S_0^{k_w} + S_0^{k_T} \tag{80}$$

where $S_0^{k_w}$corresponds to that part of the sum of the counters for the value *k* due to the word *w*, and $S_0^{k_T}$ is the contribution due to the other vectors in T. $S_0^{k_w}$ has also a Binomial distribution:

$$S_0^{k_w} \sim B(m, p) \tag{81}$$

The probability mass function (pmf) of this sum is:

$$P\left[S_0^{k_w} = x\right] = f_0^{k_w}(x) = \binom{m}{x} p^x (1-p)^{m-x} \tag{82}$$

And, $S_0^{k_T}$ has also a binomial distribution identical to S_0^v:

$$S_0^{k_T} \sim B\left(mt, \frac{p^2}{r}\right) \tag{83}$$

and its probability mass function (pmf) is:

$$P\left[S_0^{k_T} = x\right] = f_0^{k_T}(x) = \binom{mt}{x} \left(\frac{p^2}{r}\right)^x \left(1 - \frac{p^2}{r}\right)^{mt-x} \tag{84}$$

We can rewrite (70) as

$$\varphi = P\left[M_{S^v} - \left(S_0^{k_w} + S_0^{k_T}\right) < 0\right] \tag{85}$$

$f_{S_0^k}(x)$ can be computed as the convolution of $f_0^{k_w}(x)$ and $f_0^{k_T}(x)$:

$$f_{S_0^k}(x) = \sum_{i=0}^{x} f_0^{k_w}(i) f_0^{k_T}(x-i) \tag{86}$$

$$f_{S_0^k}(x) = \sum_{i=0}^{x} \left[\binom{m}{i} p^i (1-p)^{m-i}\right] \left[\binom{mt}{x-i} \left(\frac{p^2}{r}\right)^{x-i} \left(1 - \frac{p^2}{r}\right)^{mt-(x-i)}\right] \tag{87}$$

We can rewrite G as

$$G = M_{S^v} - \left(S_0^{k_w} + S_0^{k_T} \right) \tag{88}$$

Thus, to calculate $f_G(x) = P[G = x]$, we have to compute the cross-correlation between $f_{S_0^k}(x)$ and $f_{max}(x)$:

$$f_G(x) = \sum_{i=-mt}^{mt} f_{S_0^k}(i)\, f_{max}(x+i) \tag{89}$$

and

$$F_G(x) = \sum_{u=-\infty}^{x} \sum_{i=-mt}^{mt} f_{S_0^k}(i)\, f_{max}(u+i) = \sum_{i=-mt}^{mt} f_{S_0^k}(i)\, F_{max}(x+i) \tag{90}$$

Finally, to calculate φ:

$$\varphi = F_G(0) = \sum_{i=-mt}^{mt} f_{S_0^k}(i)\, F_{max}(i) \tag{91}$$

Although equation (91) yields an exact solution for φ, computing it is difficult. Alternatively, $F_G(x)$ can be derived by approximating $S_0^{k_w}$, $S_0^{k_T}$ and S_0^v with Poisson distributions:

$$S_0^v \simeq Poiss\left(\lambda_v = \frac{mtp^2}{r} \right) \tag{92}$$

$$S_0^{k_w} \simeq Poiss(\lambda_{kw} = mp) \tag{93}$$

$$S_0^{k_T} \simeq Poiss\left(\lambda_{kT} = \frac{mtp^2}{r}\right) \tag{94}$$

From (93) and (94):

$$\begin{aligned} S_0^k &= S_0^{k_w} + S_0^{k_T} \simeq Poiss(\lambda_{kw} + \lambda_{kT}) \\ &\simeq Poiss\left(\lambda_k = mp + \frac{mtp^2}{r}\right) \end{aligned} \tag{95}$$

The distributions of S_0^v and S_0^k can be further approximated to normal distributions:

$$S_0^v \simeq N(\lambda_v, \lambda_v) \tag{96}$$

$$S_0^k \simeq N(\lambda_k, \lambda_k) \tag{97}$$

The cross correlation between $f_{S_0^k}(x)$ and $f_{max}(x)$ is

$$f_G(x) = \int f_{S_0^k}(u) f_{max}(x+u)\, du \tag{98}$$

and the cdf of G is:

$$F_G(x) = \int_{-\infty}^{x} \int f_{S_0^k}(u) f_{max}(z+u)\, du\, dz \tag{99}$$

$$F_G(x) = \int_{-\infty}^{\infty} f_{S_0^k}(u) \left(\int_{-\infty}^{x} f_{max}(z+u)\, dz \right) du \tag{100}$$

$$F_G(x) = \int_{-\infty}^{\infty} f_{S_0^k}(u) F_{max}(x+u)\, du \tag{101}$$

$$F_G(x) = \int_{-\infty}^{\infty} \phi\left(\frac{u-\lambda_k}{\sqrt{\lambda_k}}\right) \Phi\left(\frac{x+u-\lambda_v}{\sqrt{\lambda_v}}\right)^{r-1} du \tag{102}$$

Finally,

$$\varphi = F_G(0) = \int_{-\infty}^{\infty} \phi\left(\frac{u-\lambda_k}{\sqrt{\lambda_k}}\right) \Phi\left(\frac{u-\lambda_v}{\sqrt{\lambda_v}}\right)^{r-1} du \tag{103}$$

which proves the theorem □.

Experiments and Results

For the simulation and testing of the Integer SDM I implemented the memory using a custom database for the main storage of the hard locations, and a ram cache to speed up the storing and retrieving operations. This allows us to create large Integer SDMs, with hundreds of thousands of hard locations, and with word dimensions on the order of 1,000 or 10,000 dimensions, even using modest computers. For more detail about the implementation of Integer SDM, see Chapter 7.

Several simulations were performed to test the percentage of errors in the output words. For the simulations I used an Integer SDM with 100,000 hard locations and a word length of 1,000 dimensions, where $r = 16$ and the value in each dimension is in the range of $\{0 - 15\}$. I used a probability of activation $p = 0.001$ that approximately

corresponds to a radius of the access sphere of 65[2], when the Euclidean distance variant is employed. The size of the memory, determined by the number of hard locations, was chosen to have enough hard locations in the access sphere for each read or write to support the desired properties of the Integer SDM, but to be as small as possible so as to limit the number of reads and writes required to perceive the effects of loading the memory. For this particular simulation, a total of 5,000 random vectors were stored in the Integer SDM. The vectors were also preserved in a separate database so they could be used as cues or compared with the retrievals from the Integer SDM.

The simulation was performed in four stages. In each stage, 100 vectors were randomly selected from the set of 5,000 stored vectors, and the memory was cued using these vectors with some amount of noise, that is with some number of randomly selected dimensions that were changed from the original. The amount of noise in each stage was: 5% in the first stage, 10% in the second, 20% in the third, and 30% in the last. In stages 1 and 2, 100% of the vectors were retrieved. Stage three had only one retrieval error, and stage 4 produced 65% correct retrievals. Table 4 summarizes these results. The same experiment using the variation of Manhattan distance had similar results: 100% of the vectors were correctly retrieved in the first three stages and 65% in the fourth (see Table 5). The graceful degradation in the performance shown in these experiments is similar to that observed in the original SDM (Kanerva, 1988). Based on these results, the Manhattan distance is preferred due to its simplicity. Consequently, the rest of the experiments described here utilize the Manhattan distance.

[2] The radius of the access sphere was obtained empirically. For 1,000 random points, the *pm* closest hard locations–100 in this experiment–were determined, and the farthest one was recorded. The average of these recorded values was 65.

Table 4
Simulation 1. Integer SDM capacity and noise robustness. In each stage 100 vectors were retrieved from an Integer SDM with 5,000 stored vectors, and a variable amount of noise was added in the cue (address). Euclidean distance was used for this simulation.

Stage	Noise (%)	Retrieved (%)
1	5	100.00
2	10	100.00
3	20	99.00
4	30	65.00

Table 5
Simulation 2. Integer SDM capacity and noise robustness. In each stage 100 vectors were retrieved from an Integer SDM with 5,000 stored vectors, and a variable amount of noise was added in the cue (address). Manhattan distance was used for this simulation.

Stage	Noise (%)	Retrieved (%)
1	5	100.00
2	10	100.00
3	20	100.00
4	30	65.00

Another series of experiments further tested the noise robustness and capacity of the memory. These experiments used Integer SDMs with 50,000, 100,000, and 200,000 hard locations respectively. In each of them, vectors were stored in stages, and then samples were retrieved adding different amounts of noise for each sample. I considered a retrieval to be correct when the output vector of a reading operation has no errors. Figure 18 illustrates the results of these experiments that clearly show the performance of the memory for different configurations and how it diminishes gracefully as the noise or the number of stored vectors increases.

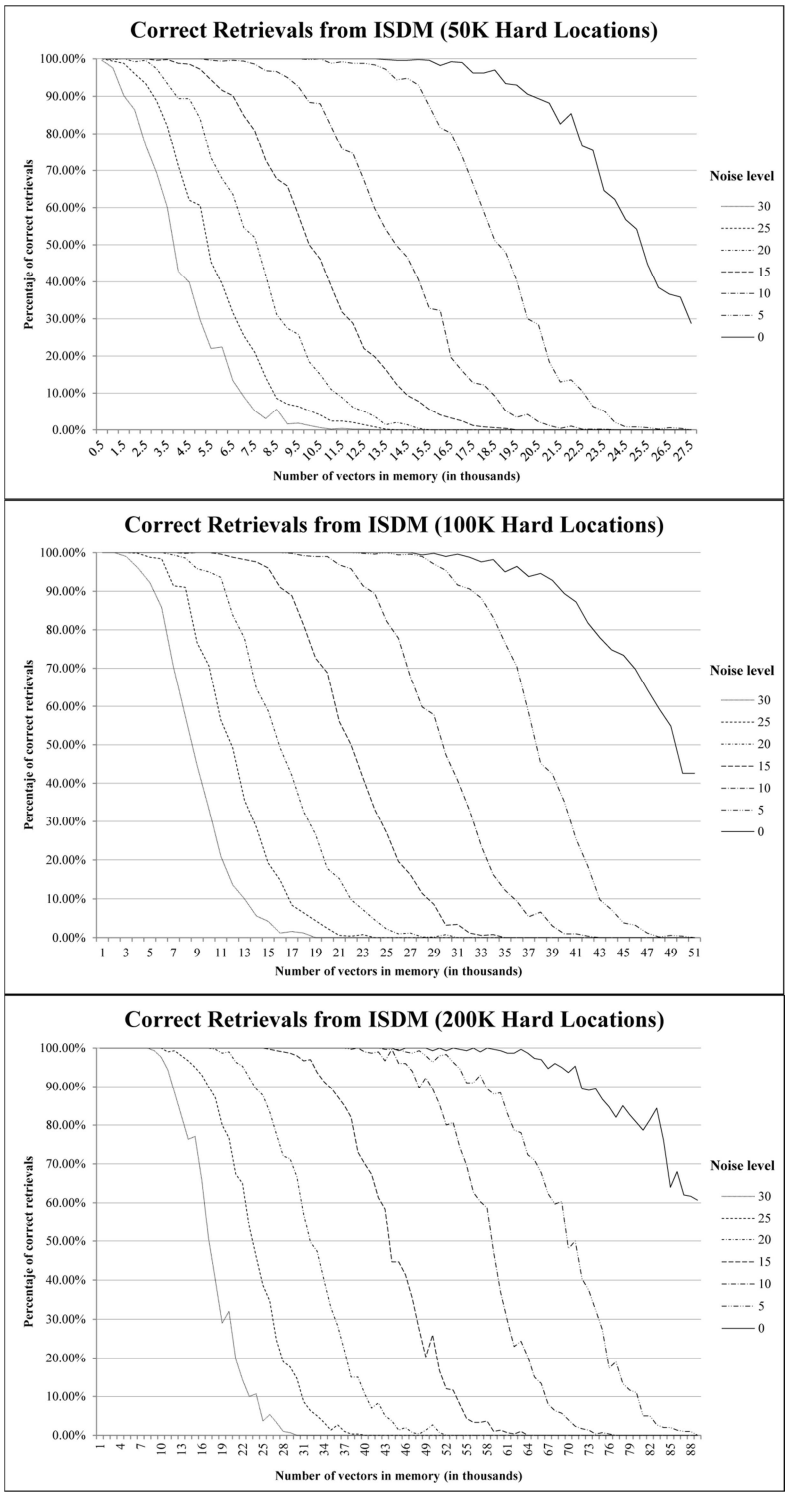


Figure 18. Retrievals from Integer SDMs with different configurations. The graphs show the retrieval rate with various levels of noise added to the cue vector for each memory configuration.

In a similar experiment, I measured the number of dimensions that differed between the stored word and the retrieved word when no noise is introduced. In a memory with 100,000 hard locations, $r = 16$, and $p = 0.001$, the results matched the theoretical expected values of φ (see Figure 19).

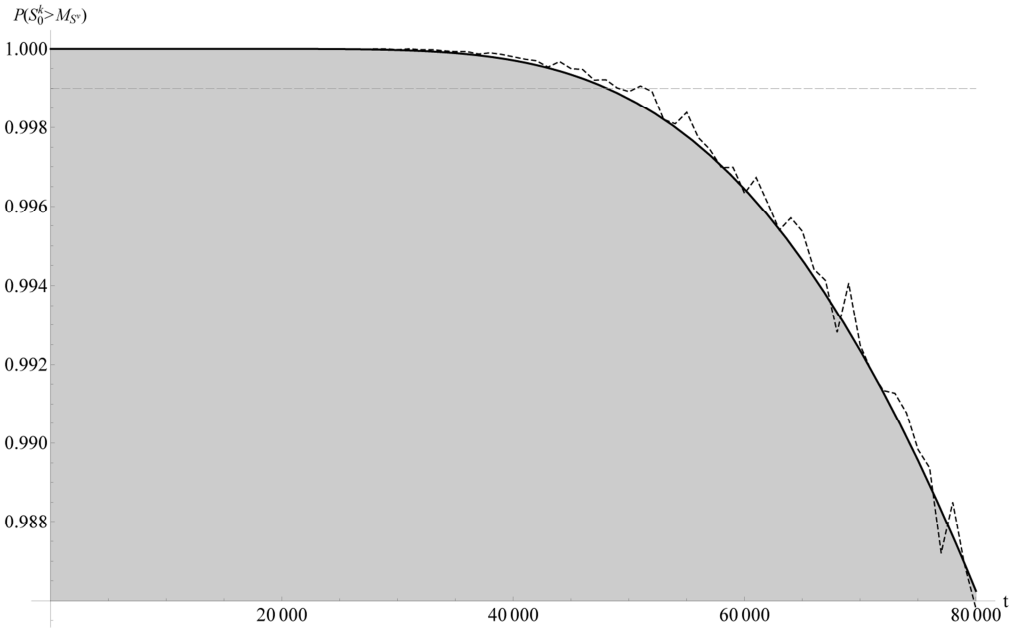


Figure 19. Comparison of theoretical value of φ (solid line) and the measured value (dark dashed line) for different values of *t*, the number of stored vectors in the memory. The light dashed line corresponds to probability 0.999, which is the value that Kanerva uses to define the capacity of the original SDM.

This experiment matches the theoretical predictions quite well, but due to the approximations in the analysis, the correspondence for all configurations is not as close as in this example. For example, the same experiment for a memory with 200,000 hard locations has a deviation from the curve of around 10%. This discrepancy may be due to the approximations in the analysis, or the slight correlation between words stored in one particular hard location. Further work will explore this effect in greater detail.

Nevertheless, the intuitions given by this theoretical analysis offer useful predictions about the memory's performance.

Another experiment demonstrated the generalization characteristics of the memory. Figure 20(a) depicts 12 images. The images are 33 x 33 pixels, gray scale, with 16 possible gray tones. For each image, one vector of 1,089 dimensions representing the information of the image was stored in the memory. Each of these vectors was saved in the memory only once. The memory used for this experiment is similar to that used in the previous experiment. It has 100,000 hard locations with addresses of 1,089 dimensions, $r = 16$ and $p = 0.001$. Notice that the images are intended to facilitate the visualization of the experiment; I do not argue that this is the best way to store or retrieve images. The memory was then cued using the new vector depicted in Figure 20(b). This vector is different from all the stored ones. The output vector's image is displayed in Figure 20(c). It is not in the training set either, and results from the interference of the stored vectors.

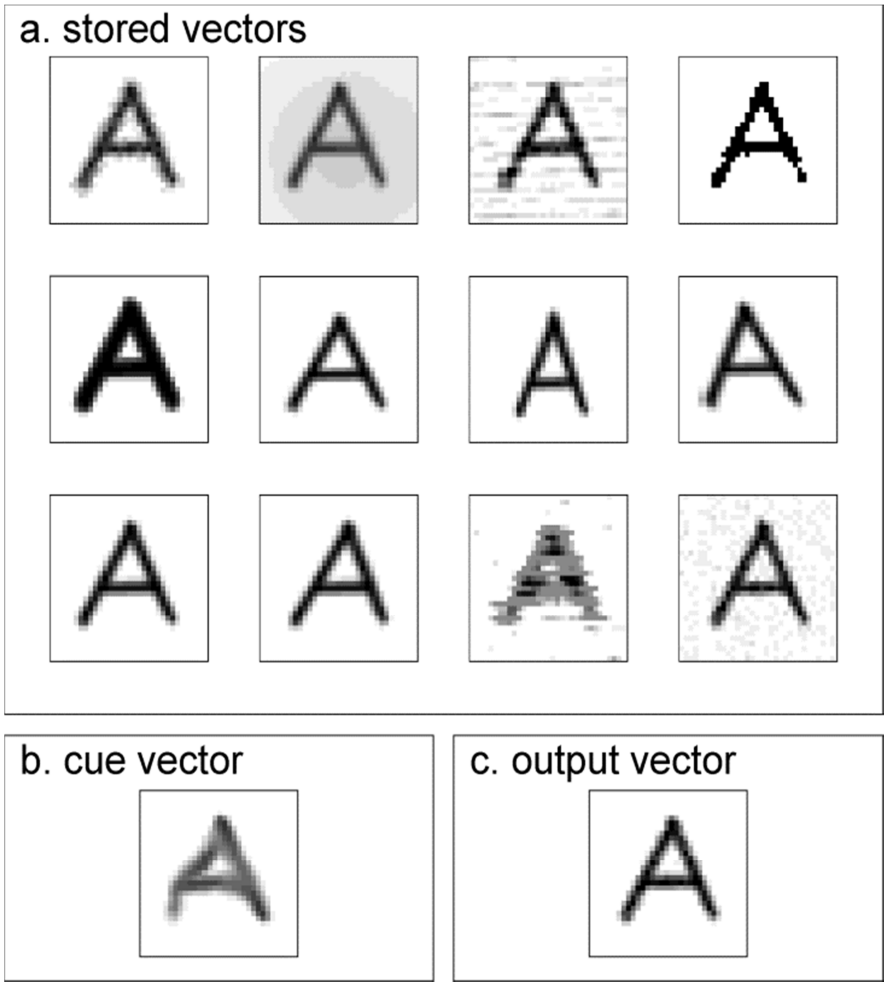


Figure 20. Generalization and pattern formation. (a) Images corresponding to vectors stored in the memory as a training set for the experiment. Each of these vectors was stored once in the Integer SDM. (b) Image corresponding to the vector used to cue the memory. (c) Image corresponding to the output vector read from the memory using (b) as cue. Vectors of images (b) and (c) are not in the training set (a).

Extensions

Integer SDM is compatible with other improvements already studied, such as the forgetting mechanism (Ramamurthy, D'Mello et al., 2006; Ramamurthy & Franklin, 2011), and the Extended SDM presented in Chapter 4.

Forgetting in Integer SDM

The structure of the Integer SDM is particularly suitable for implementing forgetting. Counters of all hard locations may be decayed, that is decremented, every several operations. The decaying procedure could use a sigmoid function to compute the decrement of each counter. In this way, vectors that do not receive sufficient reinforcement would eventually be forgotten.

One possible improvement of this decaying mechanism would be to increment the counters by more than one in the writing operation. For example, each time a counter must be incremented as a result of a writing operation in the memory, the counter would be incremented by 10 instead of only by 1. The operation of the memory does not change, but now the decaying of the counters will be smoother.

Extended Integer SDM

Another extension, which has already been implemented, is applying the same concepts as in Extended SDM (see Chapter 4). The main idea of this memory structure is the use of vectors with different lengths for the addresses and the words. This extension dramatically improves capability of the memory to store sequences and other data structures. Several of the experiments described in Chapter 4 have been reproduced using integer vectors with similar results.

This extension is particularly interesting in comparison with the implementation described by Jockel (2009) that uses SDM for a robotic arm manipulation system. This application requires vectors encoding non-binary data and sequences of these vectors. This architecture is composed of a multilayer SDM memory, and several encodings were tested. The resulting architecture is more complex and limited than the Integer SDM

presented here. Extended Integer SDM, a combination of Extended and Integer SDM's, could directly handle integer vectors and sequences with intersections.

Other Extensions

Other designs of activation of hard locations, such as Jaeckel's selected coordinate design (Jaeckel, 1989a), can also be implemented with Integer SDM. This can improve the signal-to-noise ratio as in the original SDM. Along the same lines, other distances can be used in the space such as the cosine operator.

Conclusions

In this chapter I have presented a new version of SDM, the Integer SDM, that overcomes the limitations of the original SDM resulting from its use of binary vectors. This memory preserves the desirable, biologically inspired properties of the original. It is also noise robust, auto-associative, and distributed. It degrades gracefully when some hard locations fail, or when the memory approaches its maximum capacity. It is also able to generalize patterns due to interference of several similar vectors. These properties make Integer SDM a good candidate for modeling episodic memory in autonomous agents.

The integer representation has several advantages over the binary one. The encoding of values is simpler, avoiding undesirable effects of other encodings (Jockel, 2009; Mendes et al., 2009), and diminishes the effect of normalization when several vectors are combined, for example in the storing and retrieval of sequences (Snaider & Franklin, 2011).

Several extensions of the Integer SDM were also presented. Some of them are already implemented such as the extended vectors for sequence storing. Others, such as the forgetting mechanism, are partially implemented.

Many applications can benefit from the advantages of this memory over the standard SDM. The already-mentioned robotic arm manipulation system is one of them. The episodic memory for the LIDA cognitive architecture (Franklin & Patterson, 2006; Ramamurthy, Baars et al., 2006; Ramamurthy & Franklin, 2011) is implemented with SDM. Integer SDM could offer a better implementation for episodic memory in this architecture. I also argue that Integer SDM could be used to implement other memory modules in this architecture, such as procedural memory or perceptual memory. Integer SDM is a good candidate as a cleanup memory for use with Modular Composite Representation, described in Chapter 6.

Chapter 6: Modular Composite Representation

In Chapter 3, I discuss vector representations in general, and reduced descriptions, a mechanism for encoding complex structures as single vectors, in particular. The main idea behind reduced descriptions is to have a dual representation: the complex structure can be represented explicitly, with a vector for each component, or as a reduced description, where a single vector represents the whole structure.

This chapter introduces the Modular Composite Representation (MCR): a new reduced description model that employs long integer vectors. This representation paradigm has properties similar to Spatter Code (Kanerva, 1994), which uses binary vectors, and to Holographic Reduced Representations (HRR) (Plate, 1995, 2003), based on vectors of real or complex numbers. This new model satisfies the four desirable characteristics of reduced descriptions analyzed by Plate (2003) and discussed in Chapter 3: representation adequacy (full descriptions can be reconstructed from the reduced ones), reduction (the reduced descriptions have a size similar to their components), systematicity (the process of constructing the reduced description must be well known and deterministic), and informativeness (the reduced description encloses information about the whole it represents)(p. 19). MCR also provides explicit similarity; that is, similar elements have similar representations.

Modular composite representation generalizes the ideas implemented in Spatter Code: the operations employed in MCR are equivalent to the XOR and integer sum defined in Spatter Code (see Chapter 3 for details), but extended to the modular integer space. As Kanerva noted in a personal communication with the author, MCR also correlates with HRR in the frequency domain, which we will explore later in this chapter.

High-dimensional vector spaces have interesting properties that make them attractive for representation models. The distribution of the distances between vectors in these spaces and the huge number of possible vectors allow a noise-robust representation model where the distance between vectors represents the similarity (or dissimilarity) of the concepts they represent. In Chapters 2 and 3, I extensively described the properties of high dimensional spaces in general, and the binary case in particular. In order to qualify as a reduced description representation model, MCR must define grouping and binding operations, as well as a similarity measure (or distance). These operations must fulfill additional properties discussed in Chapter 3. Notice also that although MCR requires for some operations an associative memory for cleaning up the result vectors, it does not need to be an Integer SDM; any associative memory can fulfill this requirement. MCR only requires using modular integer vectors and the operations among them defined in this chapter.

The following subsections describe the vector space used in MCR, its basic operations, and its similarity measure. Next, I describe several experiments and compare their results with those of Plate using HRR. Then I analyze the expected value and variance of some expressions, and conclude with contrasting MCR with Spatter Code and HRR.

Modular Integer Vectors

MCR utilizes large modular integer vectors, as introduced in the chapter on Integer SDM (Chapter 5). These vectors have a defined integer range of possible values for each dimension. For example, the range of values can be {–8, 7} or {0, 15}. Although any range of values is possible, for simplicity in the notation and analysis, I will use ranges

with 0 as the lower bound and $r - 1$ as the upper bound, and only even values of r. In more formal notation, MCR employs vectors within multidimensional space, $v \in \mathbb{Z}_r^n$, where n is the number of dimensions of the space and r is the size of the range of values for each dimension. The dimensions of the space follow modular arithmetic. The greatest possible value for a dimension is $r - 1$ and the next value after $r - 1$ is 0.

Figure 21 serves to clarify the following definitions of possible relations between values. The complement of a value is another value such that their sum equals r. For example, if $r = 16$, the complement of 3 is 13. The opposite of a value is the value in its *antipode*, which is calculated by adding $r/2$ to it.

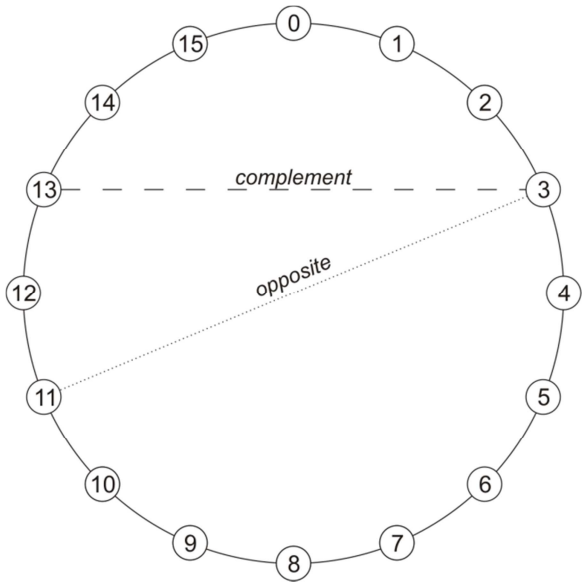


Figure 21. The possible values for one dimension of a modular integer vector with $r = 16$. The complement of a value is another value such that their sum equals r. The opposite of a value is the value in its *antipode*, that is, the value plus $r/2$.

Several integer arithmetic operations have their corresponding modular versions. The modular sum corresponds to the arithmetic sum modulo *r*:

$$a_r + b_r = mod_r(a + b) \tag{104}$$

where $mod_r(\ldots)$ is the reminder of the integer division by *r*. For example, if $r = 16$, the modular sum of 6 and 12 is 2. The modular subtraction is defined in a similar way:

$$a_r - b_r = mod_r(a - b) \tag{105}$$

Subtraction can also be expressed as the sum of the complement. To show this we can add *r* inside the mod_r term, which does not alter the result:

$$a_r - b_r = mod_r(r + a - b) \tag{106}$$

or

$$a_r - b_r = mod_r(a + (r - b)) \tag{107}$$

where $(r - b)$ is the complement of *b*. Other operations such as multiplication and division also have equivalents in modular arithmetic, but MCR does not utilize them.

The individual values in each dimension of the vectors used in MCR do not have to follow any particular distribution: they can be randomly chosen from $\{0, r - 1\}$. In contrast, HRR vectors must follow a normal distribution with specific parameters; otherwise, the operations defined in HRR to combine vectors do not produce the desired results. See Chapter 3 and Plate (2003) for further discussion about this subject. Nonetheless, to construct useful models, vectors that represent unrelated concepts ought

to have *distant* representations, and random vectors that are uniform distributed in the space tend to be far apart from each other.

Manhattan Distance in a Modular Space

MCR utilizes a variation of the Manhattan distance introduced in Chapter 5:

$$d(u, v) = \sum_i \Delta_i \tag{108}$$

where

$$\Delta_i = \min\left(mod_r(u_i - v_i), mod_r(v_i - u_i)\right). \tag{109}$$

Similar to SDM (see Chapter 2), in which the binary vector space has a large number of dimensions, the distances from a given vector to the rest of the vectors in the space tend to concentrate highly at half of the maximum distance. Kanerva called this effect the space's *tendency to orthogonality*.

In order to analyze the behavior and properties of the modular integer vectors employed in MCR, it is useful to know the distribution of the distances among the vectors in the space. The following theorem approximates this distribution for the case when r is even. The result is similar, but not exactly the same, when r is odd.

Theorem: If the dimensions of all vectors are independent and uniformly distributed in $\{0, r-1\}$ and r is even, then the distribution of Manhattan distances from a given vector to the rest of the vectors of the space can be approximated by:

$$D \sim N\left(\frac{nr}{4}, \frac{n\left(r^2+8\right)}{48}\right) \tag{110}$$

Proof. The dimensions of the vectors are independent and uniformly distributed in $\{0, r-1\}$. The distance from the origin to a vector $v \in \mathbb{Z}_r^n$ will be the sum of n random iid variables $X_i = \Delta_i$, where $\Delta_i = \min\left(mod_r(0 - v_i), mod_r(v_i - 0)\right)$.

The possible values of X_i are between 0 and $r/2$ and X_i does not have a uniform distribution since values 0 and $r/2$ have half of the probability of the other possible values. This is because the modular property of the space (and the distance calculation). For example, if $r = 16$, the maximum difference in dimension i between v ant the origin is 8, and the only possible value of v_i is 8. The same is true for a difference of 0. For other possible values of the difference, for example 4, there are 2 possible values of v_i: 4 and 12. More formally, since adding r to the argument of the mod_r function does not alter the result, we can rewrite the expression of X_i as

$$X_i = \min\left(mod_r(r - v_i), mod_r(v_i)\right) \tag{111}$$

The values of v_i are uniformly distributed in $\{0, r-1\}$. If $v_i = 0$, then both arguments of the min function are zero; thus $X_i = 0$. For all other possible values of v_i none of the arguments of min is zero, thus $v_i = 0$ is the only value that produces $X_i = 0$, and then $P(X_i = 0) = 1/r$.

For values of $v_i \in \{1, r-1\}$ the argument of the two mod_r functions are positive and less than r. So, we can rewrite the expression of X_i as

$$X_i = \min(r - v_i, v_i) \text{ where } v_i \in \{1, r-1\} \tag{112}$$

It is easy to see that the maximum value of $X_i = r/2$. If $v_i \leq r/2$, then $r - v_i \geq r/2$, and then $X_i = v_i$, which is less than or equal to $r/2$. On the other hand, if $v_i \geq r/2$, then $r - v_i \leq r/2$, and then $X_i = r - v_i$ which is less than or equal to $r/2$. Notice also that for $X_i = r/2$, either $r - v_i = r/2$ or $v_i = r/2$. But, $r - v_i = r/2$ implies that $v_i = r/2$. Thus, only this value produces $X_i = r/2$, and then $P(X_i = r/2) = 1/r$.

Finally, each value $x \in \{1, r/2 - 1\}$ of X_i is produced by exactly two values of v_i. In effect,

$$x = \min(r - v_i, v_i) \Rightarrow r - v_i = x \text{ or } v_i = x \text{ where } 1 \leq x \leq r/2 - 1 \tag{113}$$

Following reasoning similar to that of the previous paragraph, it is clear that exactly one value of v_i less than $r/2$ and one greater that $r/2$ satisfy the second half of the previous expression for each value of x such that $1 \leq x \leq r/2 - 1$. Then, $P(X_i = x) = 2/r$, where $1 \leq x \leq r/2 - 1$.

Summing up, the distribution of X_i follows

$$P(X_i = x) = \begin{cases} \frac{1}{r} & x = 0, \frac{r}{2} \\ \frac{2}{r} & 1 \leq x < \frac{r}{2} - 1 \\ 0 & otherwise \end{cases} \tag{114}$$

Since the distribution of X_i is symmetric on $\{0, r/2\}$, the expected value of X_i is half of its possible values, that is, $r/4$. The variance of the distribution of X_i requires some

more analysis. We introduce the simplifying substitution, $r' = r/2$. Then, the variance of X_i will be

$$\sigma^2 = \frac{1}{r'}\sum_{i=1}^{r'-1}\left(i - \frac{r'}{2}\right)^2 + \left(\frac{1}{2r'}\left(0 - \frac{r'}{2}\right)^2\right) + \left(\frac{1}{2r'}\left(r' - \frac{r'}{2}\right)^2\right) \tag{115}$$

$$\sigma^2 = \frac{1}{r'}\sum_{i=1}^{r'-1}\left(i - \frac{r'}{2}\right)^2 + 2\left(\frac{1}{2r'}\left(0 - \frac{r'}{2}\right)^2\right) = \frac{1}{r'}\sum_{i=0}^{r'-1}\left(i - \frac{r'}{2}\right)^2 \tag{116}$$

$$\sigma^2 = \frac{1}{r'}\sum_{i=0}^{r'-1}\left(i^2 - ir' + \frac{r'^2}{4}\right) = \frac{1}{r'}\sum_{i=0}^{r'-1}(i^2) - \sum_{i=0}^{r'-1}(i) + \frac{r'^2}{4} \tag{117}$$

$$\sigma^2 = \frac{(r'-1)(2r'-1)}{6} - \frac{(r'-1)r'}{2} + \frac{r'^2}{4} = \frac{r'^2+2}{12} \tag{118}$$

Substituting back r, the variance of X_i is

$$\sigma^2 = \frac{r^2+8}{48} \tag{119}$$

Since $X_1,\ldots, X_n$ are independent and identically distributed and

$$D = \sum_{i=1}^{n} X_i \tag{120}$$

it follows from the central limit theorem that for large number of dimensions n we can approximate the distribution of the distances by a normal distribution with mean $n\mathrm{E}[X_i]$ and variance $\mathrm{var}(X_i)n$. In conclusion, the distribution of distances from the origin (or any other point) to the rest of the points of the space is:

$$D \sim N\left(\frac{nr}{4}, \frac{n\left(r^2+8\right)}{48}\right) \quad (121)$$

which proves the theorem □.

When *n* is large, for example 1,000 or 10,000, the ratio between the mean and the standard deviation of the distance distribution tends to be large, with values concentrated around half of the maximum distance. For example, when n = 1,000 and r = 16, the distribution of the distances is well-approximated by a normal distribution with a standard deviation of 74.16 and mean distance of 4,000. Dividing the mean by the standard deviation–about 54 in this example–yields the number of standard deviations between a vector and the bulk of the space. Notice that per the normal distribution, 99.9999% of the vectors of the space lie within five standard deviations of the mean, corresponding to distances between 3,630 and 4,370 in the current example. The probability of a random vector of being closer than 3,000 is almost zero (~10^{-43}), which is a useful property that helps to make the model extremely robust.

Basic Operations

Chapter 3 presented the basic vector operations employed by reduced description models to combine into a single vector other vectors that represent the elements of a complex structure. Two basic operations, grouping and binding, constitute the heart of the reduced description models. Grouping (or sum) operation is used to create sets or groups of elements, and binding (or multiplication) creates representations for bonds among elements, such as in the role-filler case. Given that the required properties of these operations (described in Chapter 3) are responsible for the behavior and characteristics of

the reduced description models, each model can define these operations according to the characteristics of its vector space. In this way, we can abstract the reduced description model ideas and its basic operations to explore problems and applications independently of the reduced description implementation. For example, consider the following expression that represents a red circle:

$$F = [circle \otimes Shape + red \otimes Color] \tag{122}$$

where *circle*, *Shape*, *red*, and *Color* are vectors, and the symbols $\otimes$ and + represent the binding and grouping operations respectively. This expression can work in any reduced description model with appropriate definitions for grouping and binding.

The rest of this section defines the binding and grouping operations used in MCR. These definitions fulfill all the requirements described in Chapter 3, enabling MCR as a reduced description system able to perform hyperdimensional computing expressions and applications. Chapter 3 and Kanerva (2009) introduced many of these hyperdimensional computing applications.

The binding (or multiplication) of modular integer vectors is defined as the modular sum in each dimension. For example, the multiplication of two vectors *A* and *B* $\in \mathbb{Z}_{16}^{n}$, with values for dimension *i* equal 10 and 12 respectively, produces a new vector *C* with dimension *i* equals to 6.

$$C_i = mod_r(A_i + B_i) \tag{123}$$

This operation resembles the bitwise XOR used in Spatter Code.[1]

[1] Actually, XOR is a special case of the modular sum when $r = 2$.

The unbinding operation is simply the modular subtraction in each dimension, or the modular sum of the first operand with the complement of the second operand in each dimension. This leads to the definition of the inverse vector in this model. The inverse of the vector A is another vector A^{-1} such that each dimension i of A^{-1} is the complement of the value of A in the same dimension:

$$A_i^{-1} = mod_r(r - A_i) \tag{124}$$

This multiplication operation has all the properties described in Chapter 3: It is associative, commutative, distributive over the sum (see below), and preserves distances. Given that the definition of the MCR vector multiplication employs the modular sum in each dimension, it inherits its associativity and commutativity properties. For example, when adding the values of dimension i of two vectors, $mod_r(A_i + B_i) = mod_r(B_i + A_i)$. Also, for this operation it holds that

$$mod_r\big(A_i + \ mod_r(B_i + C_i)\big) = \ mod_r(\ mod_r(A_i + B_i) + C_i) \tag{125}$$

These properties also lead to the distance-preserving property of this multiplication.

Theorem: The multiplication of MCR vectors defined above preserves the distance between vectors. Given three MCR vectors A, B, and C, the following equality holds:

$$d(A, B) = d(A \otimes C, B \otimes C) \tag{126}$$

Proof. Suppose the distance between A and B is d. From equations (108) and (109)

$$d = \sum_i \min\left(mod_r(A_i - B_i), mod_r(B_i - A_i)\right) \tag{127}$$

After multiplying A and B by C, the first operand of the *min* function becomes

$$mod_r\left(mod_r(A_i + C_i) - mod_r(B_i + C_i)\right) \tag{128}$$

Applying the associativity and commutative properties of the modular sum produces the following expression:

$$mod_r\left(mod_r(A_i - B_i) + mod_r(C_i - C_i)\right) = mod_r(A_i - B_i) \tag{129}$$

which is identical to the original expression before the multiplication. Applying the same procedure to the second operand produces a similar result. Consequently,

$$d(A, B) = d(A \otimes C, B \otimes C) \tag{130}$$

which proves the theorem □.

This multiplication produces vectors that tend to differ from the operands.

$$A \otimes B \approx A \text{ and } A \otimes B \approx B \tag{131}$$

Later in this chapter I will explore the expected value and variance of the vectors produced by the multiplication.

The grouping (or sum) operation is a bit more difficult to define. In fact, there are several options for this operation. To correctly evaluate the different options, we have to consider that producing vectors similar to its operands is the most important characteristic of the grouping operation. This similarity allows identifying a composed vector from

some of its elements, and vice versa, a fundamental property of reduced description models. The first alternative consists of the *average* of the operands' values for each dimension, choosing randomly among the closest ones if the average produces a non-integer value. This value corresponds to the middle point on the arc between the two values corresponding to each operand on the circle of Figure 21. For example, if we group the vectors A and $B \in \mathbb{Z}_{16}^{n}$ with values for dimension i 10 and 12 respectively, the result has a value 11 in that dimension. Applying this operation to all dimensions produces a new vector that is approximately equidistant from its operands. A problem arises when the vectors to group have opposite values for one dimension, since the average then has two possible values that must be defined by chance. For example, the average for a particular dimension of vectors with values 5 and 13 can be either 9 or 1.

The lack of associativeness in the average operation generates further difficulties when grouping several vectors, as illustrated in following example. In the same modular space with $r = 16$, the average of values 0, 7 and 8 yields 5; however, averaging 7 and 8 first and then grouping with 0 produces a different result (4). Associating the values in other ways produces yet other results. Even worse, if the values of the operands lie in different semicircles (see Figure 21), the average must consider the two possible paths between values (i.e., the two arcs on the circle that connect the values in one direction or another), picking the one that minimizes the distances from the resulting value to the operands, overcomplicating the operation. An interesting solution utilizes a mechanism similar to the sum operation defined for HRR in the frequency domain (Plate, 2003, p. 146). Let us consider each possible value as a vector of unit length in a plane, called an *equivalent vector*. The center of the circle in Figure 22 corresponds to the coordinate's

origin in this plane. For example, the equivalent vector for the value zero is (0, 1) and the value seven corresponds to the equivalent vector ($\sqrt{2}, -\sqrt{2}$). The sum operation involves two steps to calculate each dimension *i*: the equivalent vector sum and the normalization. The first step consists of calculating the rectangular sum (i.e., their vector sum) of the equivalent vectors corresponding to the values of each operand for dimension *i*. Second, the normalization process calculates each dimension of the group vector as the closest value corresponding to the resultant vector normalized to length one. Since the dimensions have only *r* possible values, a table with the equivalent vectors' components and the tangent of their angles can speed up the calculation and normalization processes. Figure 22 shows the representation of the equivalent vectors and a couple of examples of grouping.

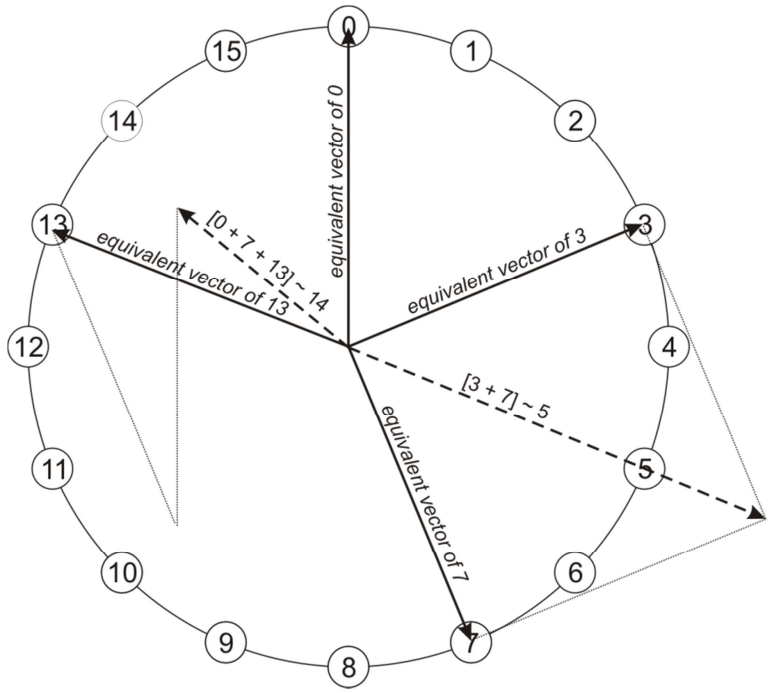


Figure 22. Equivalent vectors and examples of grouping.

We can attach a weight to some of the vectors when we group them by multiplying the corresponding vectors of their dimension values by a scalar or weight. For example, suppose we want to group the vectors A and B with weights w_A and w_B . For each dimension i we have to sum the equivalent vectors a_i and b_i corresponding to the values A_i and B_i respectively, multiplying a_i by the scalar w_A and b_i by the scalar w_B.

$$C_i = \text{value}_r(w_A a_i + w_B b_i) \tag{132}$$

where $\text{value}_r(x)$ produces the closest value corresponding to the vector x.

As in the binary case explained in Chapter 3, we can extend the definition of this sum for the case of more than two operands by simply summing, in each dimension, all the equivalent vectors of the operands for each dimension before normalizing. Grouping several operands in this way produces more consistent results than summing and normalizing in each individual group operation between two operands. Figure 22 depicts the result for combining three vectors that have values 0, 7, and 13 respectively for a given dimension.

Interestingly, the length of the resultant vector gives an idea of the quality of the resulting value for that dimension: a longer resultant vector is more likely to represent an almost mid-point between the operands' values than a shorter one. Similar values have equivalent vectors with similar directions. Adding these equivalent vectors will produce a new vector with length approximately equal to the sum of the operands' lengths. On the other hand, a short resulting vector indicates that several opposite (or near opposite) equivalent vectors comprise the operands, producing a resulting vector dissimilar to some (or all) of these values. Figure 22 illustrates examples of both situations. Finally, it is

worthy of mention that using this definition of sum produces the same result as the average version in the case of grouping only two vectors.

The final option for grouping is similar to the one used in Spatter Code (Kanerva, 2009): applying a majority rule in each dimension. This simple technique works only when combining several vectors because with few operands, the chances of equal values in one dimension in several vectors is small, producing an undefined value in that dimension that must be determined randomly.

Comparing these options for the grouping operation, clearly the sum of equivalent vectors emerges as the most appropriate one. The other options have serious flaws, including more complex algorithms, or the introduction of more noise in the result. When combining only two vectors, the average of each dimension, which produces the value corresponding to the midpoint of the shorter arc between the two values in the circle of values, is still useful due to its simplicity. The complexity of the sum, defined as the addition of equivalent vectors, is $O(nt)$ where n is the number of dimensions of the vector and t is the number of vectors to group. However, this operation requires calculating the components of the vectors representing the values of each operand and each dimension, which involves calculating the sine and the cosine of the angle of the equivalent vector of each value and an arctangent at the end, which could be computationally expensive (i.e., a large constant in the time complexity). Nevertheless, since there are only r possible and predefined values for each dimension, using tables for the two components and the tangent of the equivalent vectors greatly alleviates this problem.

This grouping operation has the required properties described in Chapter 3. Since the rectangular sum of vectors is commutative and associative, the grouping operation

shares these properties. Actually, as in the binary sum described in Chapter 3, this operation is not strictly associative because of the normalization after each sum. However, using the expanded definition for several operands as defined above mitigates this problem. Finally, the multiplication distributes over the sum. We can interpret the multiplication as a *rotation* of the circle of values for each dimension. Clearly, rotating equivalent vectors and then adding them produces a resulting vector identical to the result of first adding the equivalent vectors and then rotating.

Hyperdimensional Computing with Modular Composite Representation

In this section, I will use an example, which Plate (2003) introduced when presenting HRR, of encoding events with MCR, allowing us to compare the results from both models (pp. 128-134). This example employs 512-dimensional vectors with an r of 16.

As pointed out in Chapter 3, some hyperdimensional operations produce noisy versions of the target vector, requiring a cleanup memory with all the vectors used in the experiment to produce the correct vector. When required, this example will use a hash table data structure to maintain all the vectors, and an exhaustive search procedure that computes the distances from a given vector to all the vectors in the table, returning the closest ones. At the end of this section, I present the results from the same experiments using Integer SDM as cleanup memory.

The example requires some base vectors (vectors representing features other vectors are composed of) that are independently and uniformly distributed in the space. The expected distance between these vectors is around the mean distance $nr/4$ (2,048 in this example). Composing some of these base vectors by grouping and binding them defines more complex elements. For clarity, base vectors will be divided into three

categories: event types, object features, and role features.The event type category includes the vectors **cause**, **eat**, and **see**. The object feature category comprises **being**, **human**, **state**, **food**, **fish**, and **bread**. Finally, **object** and **agent** constitute the role features group. The following formulas define the token and role vectors for this example:

$$\mathbf{mark} = \mathbf{being} + \mathbf{human} + \mathbf{id}_{mark} \tag{133}$$

$$\mathbf{john} = \mathbf{being} + \mathbf{human} + \mathbf{id}_{john} \tag{134}$$

$$\mathbf{paul} = \mathbf{being} + \mathbf{human} + \mathbf{id}_{paul} \tag{135}$$

$$\mathbf{luke} = \mathbf{being} + \mathbf{human} + \mathbf{id}_{luke} \tag{136}$$

$$\mathbf{thefish} = \mathbf{food} + \mathbf{fish} + \mathbf{id}_{the\_fish} \tag{137}$$

$$\mathbf{thebread} = \mathbf{food} + \mathbf{bread} + \mathbf{id}_{the\_bread} \tag{138}$$

$$\mathbf{hunger} = \mathbf{state} + \mathbf{id}_{hunger} \tag{139}$$

$$\mathbf{thirst} = \mathbf{state} + \mathbf{id}_{thirst} \tag{140}$$

$$\mathbf{eat}_{\boldsymbol{agt}} = \mathbf{agent} + \mathbf{id}_{eat\_agent} \tag{141}$$

$$\mathbf{eat}_{obj} = \mathbf{object} + \mathbf{id}_{eat\_object} \quad (142)$$

Other role vectors, such as $\mathbf{see}_{agt}$, have similar definition expressions. The construction of these vectors using these expressions produces similar vectors within each category which are also dissimilar to vectors in other groups. For example, the vectors **mark** and **paul** are similar, and both are dissimilar to **thebread**. The id vectors are also random vectors (generated in the same way as the base vectors) that help to discriminate the vectors within the same group. We can considerer **fish** as a **being**, and construct the **fish** vector accordingly, but I follows Plate's example where he defined the **fish** vector with the expression above.

Table 6 summarizes the distances among representative vectors in the example. The distance between a vector and itself is always zero. Notice that in HRR, this is not always the case (Plate, 2003, p. 130): a vector can have a distance from itself different than zero. (However, in the HRR frequency domain, this distance is always zero.)

Table 6
Distances among some vectors of the example. The diagonal, with distances equal to 0, corresponds to the distance of a vector with itself. Notice that vectors with common features, such as the vectors that represent persons, are close (see text for a definition of "close").

	mark	john	paul	luke	thefish	thebread	hunger	thirst
mark	0							
john	1078	0						
paul	1101	1113	0					
luke	1121	1125	1088	0				
thefish	2008	1978	2027	1965	0			
thebread	2102	2084	2099	2077	1502	0		
hunger	2033	2027	2044	2046	2033	2009	0	
thirst	2036	2012	1995	1975	2068	2034	1345	0

Vectors with common features, such as vectors that represent persons, have small distances between them. According to equation (110), the distance distribution of the vectors in the space has a *SD* approximately equal to 50 and a mean of 2,048. The likelihood that **mark** and **john** are within distance 1,078 of each other by chance alone is almost zero ($\sim 10^{-69}$). The distances among unrelated vectors cluster around 2,048, the indifference distance.

Using the token and role vectors, we can create vectors representing different events. Table 7 describes the events of this example and the equations used to create the corresponding MCR vectors. These equations are just one of many available options. For example, binding each event type vector (such as **eat**) with an event type role vector (e.g., $\mathbf{event}_{type}$) will facilitate the decoding of the event type.

Table 7
Events created using the token and role vectors of the example.

Event	Equation
Mark ate the fish.	$S_1 = \text{eat} + \text{eat}_{\text{agt}} \otimes \text{mark} + \text{eat}_{\text{obj}} \otimes \text{thefish}$
Hunger caused Mark to eat the fish.	$S_2 = \text{cause} + \text{cause}_{\text{agt}} \otimes \text{hunger} + \text{eat}_{\text{obj}} \otimes S_1$
John ate.	$S_3 = \text{eat} + \text{eat}_{\text{agt}} \otimes \text{john}$
John saw Mark.	$S_4 = \text{see} + \text{see}_{\text{agt}} \otimes \text{john} + \text{see}_{\text{obj}} \otimes \text{mark}$
John saw the fish.	$S_5 = \text{see} + \text{see}_{\text{agt}} \otimes \text{john} + \text{see}_{\text{obj}} \otimes \text{thefish}$
The fish saw John.	$S_6 = \text{see} + \text{see}_{\text{agt}} \otimes \text{thefish} + \text{see}_{\text{obj}} \otimes \text{john}$

Table 8 lists the distances between the vectors that represent the events S_1 to S_6. The equations used to construct these vectors influence their similarity to each other. For example, S_4, S_5, and S_6 have short distances between each other, reflecting their similarity. S_6 is farther from S_5 than S_4 even though S_5 and S_6 share the same elements;

the difference in roles accounts for this. Including the agent and object fillers as extra terms in the equation increases the similarity between events with the same elements, even if they participate in different roles. For example, the definition of S_5 would change to:

$$\mathbf{S_5 = see + see_{agt} \otimes john + see_{obj} \otimes thefish + john + thefish} \qquad (143)$$

Table 8
Distances among vectors representing the events described in Table 7.

	S_1	S_2	S_3	S_4	S_5	S_6
S_1	0					
S_2	1947	0				
S_3	1159	2002	0			
S_4	1995	2036	1830	0		
S_5	1858	1983	1839	1085	0	
S_6	2025	2024	2036	1390	1443	0

The decoding using probing works as follows: multiplying the event vector by the inverse of the role produces a vector similar to the filler vector, or in other words, the filler vector plus a small amount of noise. An auto-associative memory that contains all the vectors of the system works as a cleanup memory, which returns the closest vector to the one produced by the decoding. Table 9 shows the closest items in the cleanup memory of the example to the vectors resulting from the unbinding of several expressions. For example, in the first row, the unbinding of the agent of S_1 produces a vector closest to **mark**, the correct vector. The other vectors representing persons (**luke**, **john,** and **paul**) are closer than chance (the indifference distance is 2,048), but farther away than **mark** by about 7 *SD*.

Table 9
Results of unbinding elements from the event vectors.

Description	Expression	Rank of distances			
1. Agent of eating of S_1	$\mathbf{S_1} \otimes \mathbf{eat}_{agt}^{-1}$	**mark** (1181)	**luke** (1491)	**paul** (1523)	**john** (1593)
2. Agent of S_1	$\mathbf{S_1} \otimes \mathbf{agent}^{-1}$	**mark** (1554)	**luke** (1652)	**paul** (1660)	**john** (1778)
3. Object of S_1	$\mathbf{S_1} \otimes \mathbf{eat}_{obj}^{-1}$	**thefish** (1166)	**food** (1629)	**fish** (1666)	**thebread** (1837)
4. Agent of S_2	$\mathbf{S_2} \otimes \mathbf{cause}_{agt}^{-1}$	**hunger** (1187)	**state** (1572)	**thirst** (1737)	**human** (1897)
5. Object of S_2	$\mathbf{S_2} \otimes \mathbf{cause}_{obj}^{-1}$	$\mathbf{S_1}$ (1209)	**eat** (1620)	$\mathbf{S_3}$ (1628)	$\mathbf{S_5}$ (1908)
6. Agent of object of S_2	$\mathbf{S_2} \otimes \mathbf{cause}_{obj}^{-1} \otimes \mathbf{eat}_{agt}^{-1}$	**mark** (1666)	**luke** (1804)	**paul** (1806)	**john** (1866)
7. Object of object of S_2	$\mathbf{S_2} \otimes \mathbf{cause}_{obj}^{-1} \otimes \mathbf{eat}_{obj}^{-1}$	**thefish** (1659)	**food** (1886)	**fish** (1887)	$\mathbf{eat}_{agt}$ (1939)
8. Object of S_3	$\mathbf{S_3} \otimes \mathbf{eat}_{obj}^{-1}$	**see** (1927)	$\mathbf{see}_{agt}$ (1947)	$\mathbf{S_6}$ (1959)	**state** (1966)
9. John's role in S_4	$\mathbf{S_4} \otimes \mathbf{john}^{-1}$	$\mathbf{see}_{agt}$ (1124)	**agent** (1459)	$\mathbf{eat}_{agt}$ (1634)	$\mathbf{see}_{obj}$ (1640)
10. John's role in S_5	$\mathbf{S_5} \otimes \mathbf{john}^{-1}$	$\mathbf{see}_{agt}$ (1120)	**agent** (1497)	$\mathbf{eat}_{agt}$ (1664)	$\mathbf{cause}_{agt}$ (1724)
11. John's role in S_6	$\mathbf{S_6} \otimes \mathbf{john}^{-1}$	$\mathbf{see}_{obj}$ (1129)	**object** (1527)	$\mathbf{eat}_{obj}$ (1637)	$\mathbf{cause}_{obj}$ (1715)

Plate (2003) explained the difference between the *chunking* mechanism and the *holistic* processing with the following example (p. 134). Chunking involves a sequence of operations. For example, the expression in line 5 can decode S_1, the object of S_2, which is itself a composite vector. By first cleaning up the vector S_1, and then applying the

expression in line 1, we obtain **mark**, the agent of S_1. On the other hand, using holistic processing produces the same result in one operation, as showed by the expression in line 6, which yields the final result directly without decoding the intermediate vector S_1. Chunking produces less noise than holistic processing, but requires an extra cleanup operation.

Also interesting, the expression in line 8 returns random vectors, which are almost the indifference distance from any vector used in the system, because S_3 does not have an object component, and the expressions of lines 10 and 11 that correctly decode John's role in similar events.

MCR can employ Integer SDM as cleanup memory. Performing this same experiment using Integer SDM with a word length of 512 dimensions, 100,000 hard locations and a radius of activation of 1,925 (see Chapter 5 for details) produces results similar to those reported above, with a few notable considerations. Some of the expressions in Table 9, in particular lines 2, 6, and 7, return vectors with an elevated level of noise compared to the target vector, producing retrieval errors in a few of the runs. Increasing the radius of activation of the hard locations in the memory mitigates this problem. The rest of the expressions yield vectors that retrieve the correct values in all the trials. To simulate extra data, 1,000 random vectors were preloaded in the memory.

MCR can model other data structures, representations, and applications as described in Chapter 3. By adapting the procedures presented Chapter 4, MCR can represent sequences and related structures efficiently. Moreover, the use of random permutations is completely compatible with MCR, which allows employing them as an alternative to the multiplication described in this chapter. Using MCR, it is possible to

reproduce all the experiments described by Plate (2003) and Kanerva (2009). I have already reproduced some of them with similar results to the ones reported by Plate and Kanerva. Since these experiments do not contribute to the current discussion, additional repetition of experiments and further analysis of them are unnecessary.

Normalized Distance and Similarity

The distance defined for MCR has an inconvenient dependence on *n*, the dimensionality of the vectors, and *r*, the number of possible values, making difficult to compare the performance of MCR models with different values for these parameters. A *normalized distance* independent of *r* and *n*, denoted by *d'*, becomes useful for these comparisons:

$$d'(A, B) = d(A, B)\frac{4}{nr} \tag{144}$$

Its distribution is approximately normal with the following mean and variance:

$$D' \sim N\left(1, \left(\frac{1}{3} + \frac{8}{3r^2}\right)\frac{1}{n}\right) \tag{145}$$

The minimum normalized distance is zero, as in the non-normalized distance, but using *d'* the value one corresponds to the indifference distance, and the value two to its maximum. The distribution of *D'* clearly shows that its variance diminishes proportionally with *n* without bound, allowing the creation of a model with a distance distribution variance as low as desired. Notice that a model with a small variance has high noise robustness, accuracy, and reliability. The variance also diminishes when incrementing *r*; however, when *r* becomes large, the second term in the sum tends to zero, and 1/3 dominates. If *r* is 16 or greater, the value of the variance tends to the

maximum possible (for a given value of *n*). The worst value corresponds to r equal two, the binary case. See Figure 23 for details.

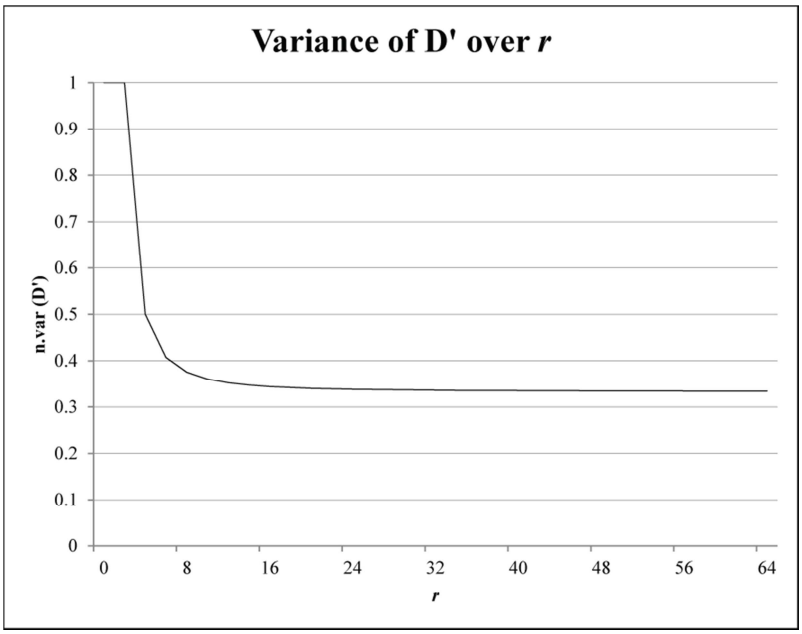


Figure 23. Variance of D’ over *r*.

The *similarity* among vectors, defined as

$$sim(A, B) = 1 - d'(A, B) \tag{146}$$

is particularly handy for comparing results to those of models that uses other similarity measures, such as the cosine. A vector has a similarity of one with itself, and zero similarity when compared with vectors at the indifference distance (corresponding to a normalized distance of one). The distribution of similarities of one vector with all the other vectors in the space is almost the same as that of *D’*, but with a mean equal to zero:

$$Sim \sim N\left(0, \left(\frac{1}{3} + \frac{8}{3r^2}\right)\frac{1}{n}\right) \quad (147)$$

Expected Value and Variance of the Similarity of Selected Expressions

Plate (2003) discussed the means and variances of different similarity measures among several prototypical expressions of HRR in the frequency domain (pp. 267-271). Here I will compare those results with the calculations using MCR. The experiments employ 512-dimensional vectors, matching the configuration used by Plate, and r =16.

Table 10 shows the theoretical values and the experimental results using MCR for several expressions that were also described by Plate (2003) using HRR (p. 271). Notice that the operations in the expressions are deterministic. In other words, with the same vectors *A*, *B*, *C*, and *D*, the expressions always produce the same results. The means and variances in the table compare the analytical estimates and experimental results after calculating each expression multiple times with different random vectors.

Due to the properties of the multiplication described previously, multiplying a vector *A* by another vector *B*, and then by its inverse B^{-1} yields exactly the same vector *A*, which explains the theoretical results of the expressions with mean 1 and variance 0. The rest of the expressions in the table compute the similarity between unrelated vectors, thus they follow the distribution of equation (147) with r = 16, and a mean equal to 0 and variance normalized by n equal to $\frac{1}{3} + \frac{8}{3r^2} = 0.34375$. The experimental values in the table show the results of 50,000 runs for each expression, all of which closely match the analytical results.

Table 10
Means and variances of selected expressions for a MCR model with n = 512 and r = 16. The experimental results correspond to 50,000 runs. The variance is normalized by multiplying by *n.*

Expression	Similarity			
	Analytic		Experimental	
	mean	n . var	mean	n . var
$sim(A, A)$	1	0.00000	1.0000	0.0000
$sim(A, B)$	0	0.34375	0.0000	0.3435
$sim(A, A \otimes B)$	0	0.34375	0.0000	0.3429
$sim(A, A \otimes C)$	0	0.34375	0.0000	0.3466
$sim\left(A, A \otimes A \otimes A^{-1}\right)$	1	0.00000	1.0000	0.0000
$sim\left(B, A \otimes B \otimes A^{-1}\right)$	1	0.00000	1.0000	0.0000
$sim\left(A, A \otimes B \otimes A^{-1}\right)$	0	0.34375	0.0000	0.3430
$sim\left(C, A \otimes B \otimes A^{-1}\right)$	0	0.34375	-0.0002	0.3412
$sim\left(C, A \otimes B \otimes C^{-1}\right)$	0	0.34375	0.0001	0.3463
$sim\left(D, A \otimes B \otimes C^{-1}\right)$	0	0.34375	-0.0001	0.3428

HRR in the frequency domain (Plate, 2003), described in Chapter 3, has the same means for each of these expressions, but with higher variances, 0.5 compared to 0.34375 in MCR (pp. 145-151). When $r = 2$, MCR is equivalent to Spatter Code, which has a variance of 1 for these same expressions. In conclusion, MCR is more noise robust than either HRR or Spatter Code for models using vectors with the same size n. Notice that in the limit as r approaches infinity, the normalized variance of the similarity of vectors in MCR tends to 1/3; the value for $r = 16$ is not far from this theoretical minimum, and values greater than 16 do not significantly improve the normalized variance. In

consequence, r = 16 is a good choose for constructing MCR vectors, enabling the representation of such values with only 4 bits per dimension, and also limiting the storage requirements of Integer SDM memories, which increases linearly with r (see Chapter 5 for details).

For the grouping operation, the analytical calculation of the means and variances are harder to obtain. Here I present the analysis for grouping two vectors, and measure the similarity to one of the operands. I also present the experimental results for grouping 2 to 15 vectors.

To analyzing the mean and variance of

$$sim(A, A + B) \tag{148}$$

we can rewrite (148) as

$$sim(A, A + B) = 1 - d'(A, A + B) \tag{149}$$

According to the definition of the sum, the output of grouping two vectors has a distance to any of the operands equal to half of the distance between them.

$$sim(A, A + B) = 1 - \frac{d'(A, B)}{2} \tag{150}$$

Given that d' approximately follows the normal distribution in equation (145), $sim(A, A + B)$ also distributes normally:

$$Sim(A, A + B) \sim N\left(\frac{1}{2}, \frac{1}{4}\left(\frac{1}{3} + \frac{8}{3r^2}\right)\frac{1}{n}\right) \tag{151}$$

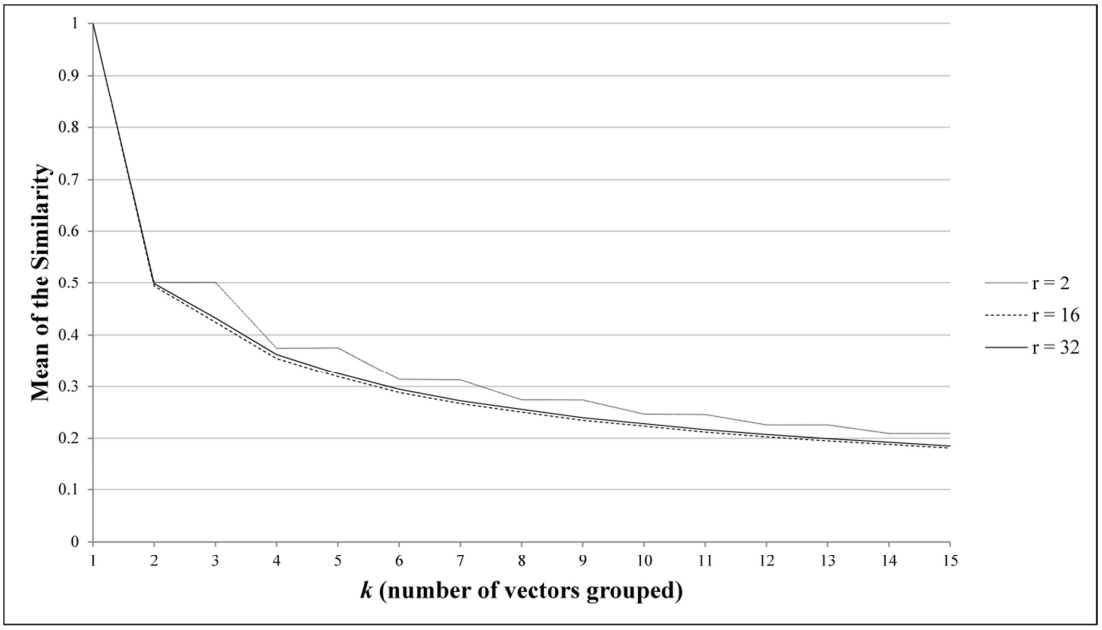
The normalized variance of the similarity given by grouping two vectors and one of its operands is approximately 0.086 for r = 16, and 0.084 for r = 32. The mean

reported by Plate (2003) using HRR is higher (0.6366) (p. 270). However, due to the difference in the variance of the distance distributions of the two spaces (remember that HRR uses cosine as similarity measure), both models have almost the same probability of presenting the mean or less similarity between the sum vector and one of its operands just by chance. In other words, the cdf (cumulative density function) for the distributions of the similarity measure for HRR and MCR, have almost the same value for similarity, equal to 0.6366 and 0.5 respectively:

$$\text{cdf}(Sim_{HRR}(A, A+B) = 0.6366) \cong \text{cdf}(Sim_{MCR}(A, A+B) = 0.5) \quad (152)$$

Furthermore, the normalized variance of this similarity using MCR is smaller than in HRR: 0.086 as compared to 0.0947, which makes MCR more noise robust and accurate compared to HRR for a given dimensionality.

Figure 24 shows the experimental results of the similarity between a random vector and the same vector grouped with $k - 1$ other random vectors, for values of k between one and fifteen. The experiments use vectors with 512 dimensions and r with values 2, 16, and 32. The data correspond to 10,000 runs for each value of k. The results for $k = 2$ confirms the theoretical analysis. Additionally, compared with HRR, MCR has better variances and similar means, considering the cdf of the distributions of the similarity functions, also making it less noisy for grouping. (See Plate, 2003 for details.)



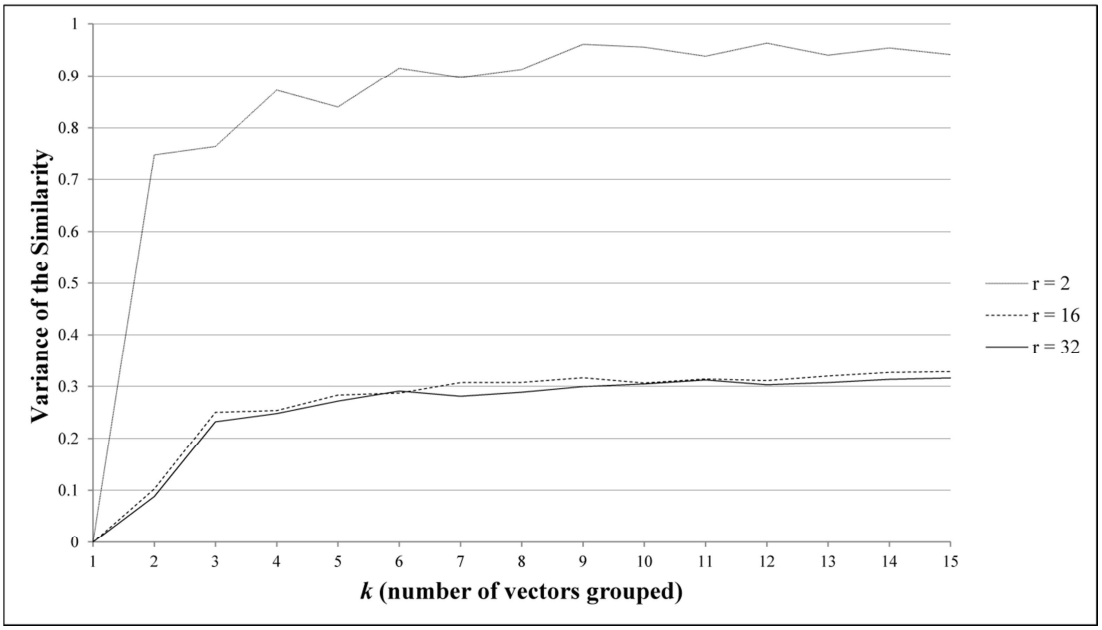


Figure 24. Means and variances of the similarity between a random vector and the same vector grouped with $k - 1$ other random vectors. The vectors have 512 dimensions and three values for r are evaluated (2, 16, and 32). The data correspond to 10,000 runs for each value of k.

Summary of Comparisons: MCR, HRR and Spatter Code

MCR shares many properties with HRR and Spatter Code. All three enable reduced descriptions. Actually, MCR is a generalization of Spatter Code that uses integer modular vectors instead of binary vectors. When $r = 2$, MCR becomes equivalent to Spatter Code. The analytical and experimental results show that MCR is more reliable and accurate than Spatter Code for a given number of dimensions; however, Spatter Code utilizes simpler operations, which would be an advantage for some applications. The representational expressiveness of MCR would be considered a further advantage over Spatter Code in applications that require the encoding of non-binary data. (See for example Jockel, 2009).

Although HRR has a rich expressive representation and very good performance when combining and decoding structures from holistic vectors, it utilizes complex operations, such as circular convolution, that have time complexities of the order of $O(n^2)$ or $O(n\text{Log } n)$. HRR in the frequency domain, also known as circular HRR, has better overall performance, can perform the operations in $O(n)$ time, and has more stable variances and results than the normal HRR. As Kanerva pointed out in a personal communication with the author, under an interpretation of the values in MCR as discretized angles, the binding and grouping operations of both models are similar. However, each model utilizes a different distance (or similarity) measurement, which explains the variations in performance between the two models. The development of MCR was inspired as an extension of Spatter Code, and as such, the simplicity of its design. The circular HRR was derived from the normal HRR, producing a more

cumbersome base for the model. Finally, MCR can readily utilize Integer SDM as its cleanup memory, whereas HRR has no specific auto-associative memory available.

Conclusions

MCR is a new reduced description representation that balances representational expressiveness and implementational simplicity. It has all the required and desirable characteristics of reduced descriptions described in Chapter 3: representation adequacy, reduction, systematicity, and informativeness. Moreover, it implements explicit and structural similarity, which allows the holistic processing of several operations, avoiding the need to reconstruct the structure prior to processing.

The experiments and analysis detailed herein have demonstrated MCR's performance in a number of scenarios, empirically validating its anticipated noise robustness, representation expressiveness, and holistic processing capability. The analysis of the means and variances for the similarities of representative operations suggests that MCR has better performance for these operations than HRR or Spatter Code using vectors with the same number of dimensions. Nevertheless, the accuracy of any of these models can be increased without bound by enlarging the dimensionality of the vectors.

To perform the experiments in this chapter I developed a script parser and interpreter that allows writing the expressions and operations of MCR in a simple language, and running it embedded within a Java program. This greatly facilitates the creation and running of experiments and applications that use MCR. Chapter 7 describes this scripting language and its implementation in more detail.

Chapters 1 and 3 discuss several challenging AI applications that would benefit from MCR. Some of the characteristics of this vector representation–noise robustness,

explicit similarity, and structural similarity–can facilitate the implementation of such applications. The simplicity of this model's operations and its performance make it an attractive option over other models. Moreover, its natural integration with Integer SDM as cleanup memory offers a further advantage.

A promising project, Vector LIDA, would implement the LIDA cognitive architecture (Franklin & Patterson, 2006; Snaider, McCall, & Franklin, 2011) using MCR vectors as its main representation for data structures. Some of the advantages over the current implementation, which employs nodes and links in a graph-like structure, includes a more realistic and biologically plausible model, better integration with the episodic memory, which already uses a vector based SDM memory, better integration with other low level perceptual processing (such as HMAX Serre et al., 2007), better scalability, and easier learning mechanisms. For further details, see Chapter 8.

Chapter 7: Implementations

This chapter describes several implementations of the Extended SDM, the Integer SDM, and the MCR interpreter. The technologies used include database storage with least recently used (LRU) cache, parallel and distributed support using the Akka framework (Subramaniam, 2011), an implementation of the actors model (Hewitt, Bishop, & Steiger, 1973), and parallel processing using Graphic Processors Units (GPUs) (Che et al., 2008; NVIDIA, 2012). The MCR interpreter was created using the Java Compiler Compiler (Javacc), a parser generator for Java[1].

Modern computers have multi-core CPUs executing instructions in parallel. Furthermore, GPUs, which can perform billions of parallel vector operations per second, can speed up applications, such as Extended SDM and Integer SDM, that have vector data structures as their main components. Such applications that could only run in high-end supercomputers a few years ago, can now execute efficiently on desktops or laptops due to the parallel processing power of modern GPU devices.

Although of polynomial time complexity $O(nm)$, where n and m represent the number of dimensions of the vectors and the number of hard locations respectively, Extended SDM and Integer SDM algorithms often execute slowly as the result of a large number of hard locations. (See Chapters 4 and 5 for details.) Similarly, the storage requirement of these models also increases linearly with n and m. The implementations discussed here explore alternatives to mitigate these drawbacks.

I chose Java for these implementations for several reasons. First, Java is a mature and solid object oriented language with countless proven libraries and frameworks that

[1] JavaCC is an open source Java parser generator. The source code and more information can be found at http://javacc.java.net/

facilitate the implementation of standard tasks such as persistence, logging, and networking. The virtual machine paradigm, central to the Java technology, enables the execution of the same program (without requiring a recompilation) in different platforms and operating systems. This improves the availability of the system and speeds up the development. For example, a Windows based machine has served as developing platform, but several experiments were performed in a Linux based, High Performance Computer. Although traditional machine-code compiled languages, such as C and C++, might produce optimized code, the just-in-time compiler and other advanced Java technologies have the potential to achieve similar performance (Oracle, 2010). Finally, the LIDA Framework, a project closely related with this work, is also implemented in Java, which biased the selection of Java.

Although many previous software implementations of SDM and its extensions have utilized arrays as their fundamental data structure (Kanerva, 1993), the software described here follows an object oriented approach. In the typical realization of SDM, the addresses of the hard locations compose one array, whereas a second array implements their counters. This simple implementation performs efficiently when the system runs in a single processor, and the data structures hold in the physical memory. However, using an object oriented paradigm facilitates the implementation of more sophisticated realizations that take advantage of multithreading, distributed processing, and the memory hierarchy.

The rest of this chapter discusses the object oriented design of SDM and its variations, the hard locations' cache, and a couple of parallel instantiations. Finally, a description of the MRC's parser and interpreter completes the chapter.

Object Oriented Design

The SDM design proposed here employs several design patterns (Gamma, Helm, Johnson, & Vlissides, 1995) to improve its flexibility. For example, the widely used factory pattern offers a standardized approach to the creation of vectors and other elements in the system. The main difference between the standard array-based implementation and the current one consists in the modeling of each hard location as an object. The hard location class has an attribute for its address, a binary vector, and another attribute referencing the counters, an *n*-size array of bytes. Several methods, mostly getters and setters, help to encapsulate the class behavior. Figure 25 displays the UML class diagram of the main classes and interfaces of the Extended SDM implementation. Two interfaces, `SparseDistributedMemory`[2] and `HardLocation`, define the conceptual behavior of the memory, and two abstract classes, `AbstractSparseDistributedMemory` and `AbstractHardLocation`, provide the implementation of their common functionalities. Finally, the concrete classes (at the bottom of the diagram) provide the specific components for a couple of variations: the normal implementation and the cached version. Notice that this design encapsulates the bulk of the functionality in the two abstract classes, whereas only a few methods are delegated to the concrete classes. The abstract hard location class includes its address, its counters, and generic methods such as the distance calculation, and accessors and mutators (i.e., getters and setters). The basic concrete class for hard locations, `HardLocationImpl`, needs only to create the counters, inheriting most of its functionality from its superclass.

[2] Class names in Java follow the camel case practice.

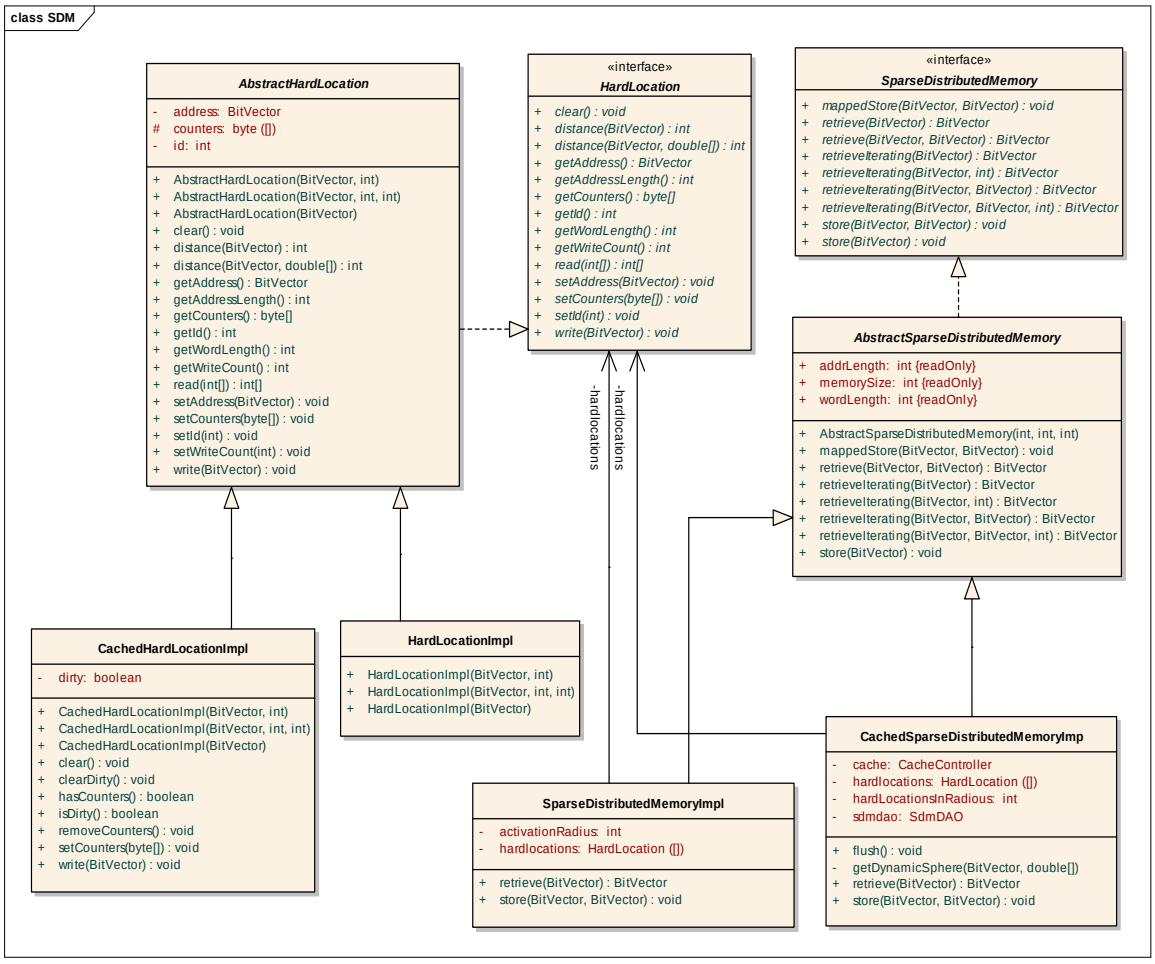


Figure 25. UML class diagram of SDM main classes. For clarity's sake, some class members were not included in the diagram.

The abstract sparse distributed memory class provides the most complex functionality of the memory, including methods for iteratively reading, and others for applying mappings while writing. The concrete classes must provide the most basic methods for reading and writing, which in turn, are called from the methods defined in the abstract class. This design facilitates reuse of code in enhancements such as the implementation of a cached version of the memory. See the next subsection for details.

Finally, the addresses and other bit vector data utilize the `BitVector` implementation of the Colt Java library[3], providing a fast, compact implementation of many bit-vector operations. The library represents binary vectors with an array of longs[4], performing several operations 64 bits at a time.

The basic Integer SDM implementation has a similar design, except that it uses a new `SdmVector` implementation instead of the `BitVector`, and a `Counter` interface (and related classes) to encapsulate counter functionality. I will discuss them in more detail in the distributed and multithreading subsection. This object oriented design provides the basis for the more advanced designs presented here, which would have been more difficult to implement using the standard array-based design.

Cached Implementation

The storage requirements of these implementations increases proportionally with *m*, number of hard locations, *n*, the dimensionality of the space, and in the case of Integer SDM, *r*, the number of possible values in each dimension. A cache design mitigates this requirement, allowing the execution of these applications in computers with moderate RAM capacity. The addresses of the hard locations require some memory, but their counters constitute the major memory-consuming elements in these applications.

Analyzing the reading and writing algorithms, both consist of comparisons of all hard locations' addresses to the target address, followed by a reading or update of a small fraction of the hard locations' counters. There is no significant advantage to storing the

[3] The Colt library is a set of open source libraries for high performance scientific and technical computing in Java developed at CERN, the European Organization for Nuclear Research. For more information see http://acs.lbl.gov/software/colt/

[4] long is a 64 bit integer data type in Java.

addresses in a secondary storage and then caching them, since both reading and writing operations require all of them. Moreover, they have a modest memory footprint compared to the counters. Thus, the addresses are instantiated directly in RAM. On the other hand, the counters are never required all at the same time, and during an iterating reading operation (see Chapter 2), the counters of many hard locations are repeatedly accessed, making them good candidates for caching.

The Extended SDM and Integer SDM cached implementations utilize a LRU cache for the hard locations' counters. The memory instantiates all the hard locations, including their address vectors, but not their counters. A cache controller provides the counters as needed. The first time a hard location is accessed for reading or writing, the cache controller creates an array with empty counters and assigns it to the hard location; subsequently, it retrieves the counters' values from a secondary memory. The controller keeps track of which hard locations have instantiated counters, saving and removing them when the space is required. A DAO class, which implements the data access object design pattern, encapsulates the communication with secondary storage, enabling the controller to interact with different secondary memories, such as databases or files, without modifying the cache logic. Figure 26 shows the UML class diagram of the cache main components. The cache controller employs the `SdmDAO` interface to become independent of the DAO implementation. The `CachedHardLocationImpl` and `CachedSparseDistributedMemoryImpl` classes have small additions to their standard counterparts, such as getter and setter implementations to access the counters in support of the cache mechanism.

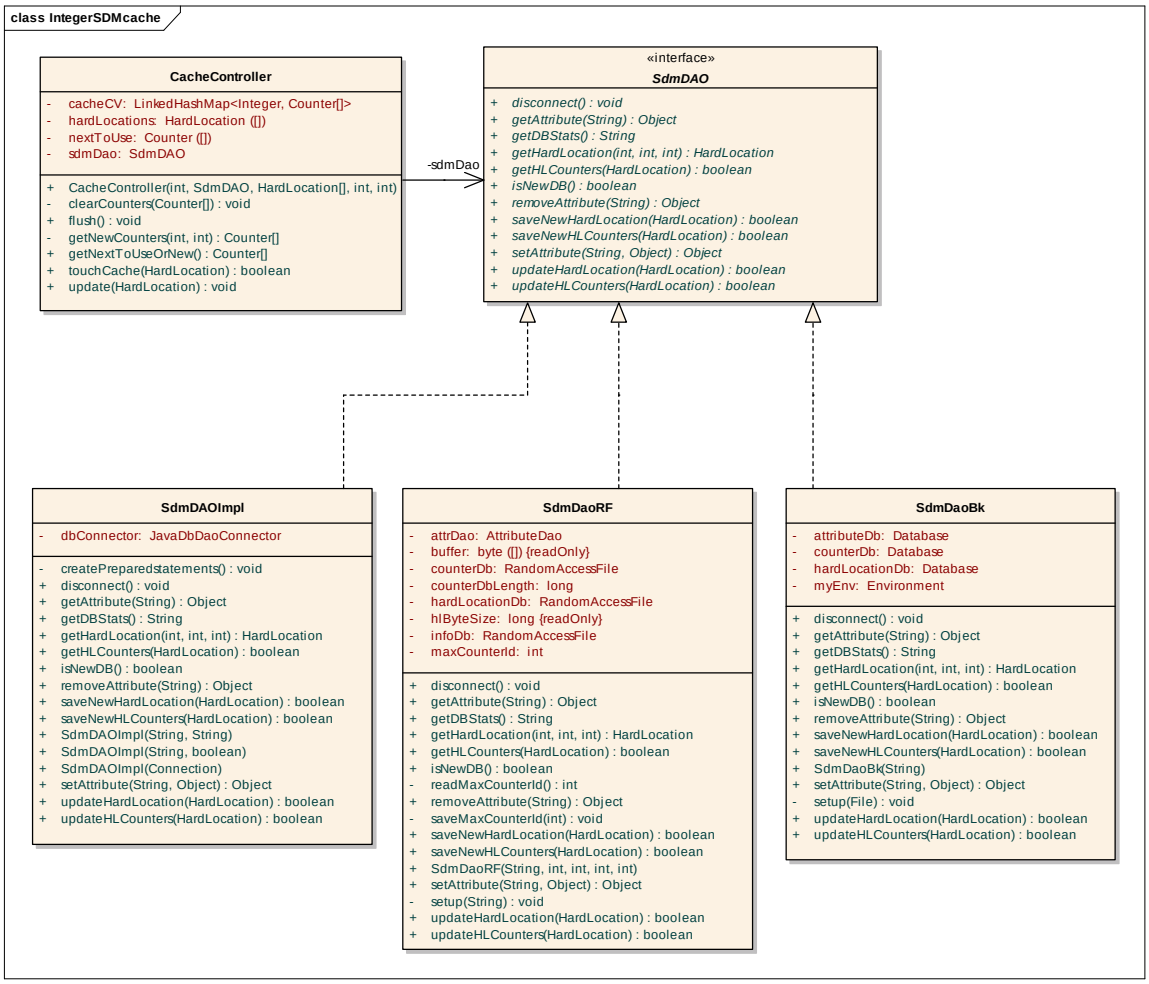


Figure 26. UML class diagram of the cache's main components. For the sake of clarity, some class members were not included in the diagram.

Three different secondary storage mechanisms were tested: relational database management system (RDBMS), non-relational databases management system (non-RDBMS), and plain data files. The RDBMS implementation stores the hard location's information in a couple of tables, and standard SQL queries provide access to their data. This design can employ any RDBMS engine supported by Java, which constitutes one of its main advantages. Two database engines were used in the simulations: JavaDB and MySQL. The former, completely implemented in Java, has the potential of embedding

the database engine in the application, enabling standalone execution in any Java platform. The second database engine, MySQL, is one of the most popular and widely available ones. Both RDBMSs worked correctly, without significant difference in performance. MySQL was only 5% faster than JavaDB. Nevertheless, the scalability and clustering capabilities of MySQL make it preferable for implementing large SDM and similar memories.

The non-RDBMSs, which employ key-value stores very similar to map data structures, have lately gained momentum in the industry. Many leading web-based applications utilize this storage paradigm because of its simplicity, robustness, performance, and scalability. In addition to these advantages, many of the non-RDBMSs use simple byte arrays as their native data type, which fits naturally to SDM technology requirements. The experiments implemented here utilize Berkeley DB (Olson, Bostic, & Seltzer, 1999), one of the first databases in this category.

In spite of all the rationale in favor of this kind of database in the context of the applications of this work, the results did not show any significant difference with the RDBMS implementations. After a careful analysis, the overhead produced by copying the byte arrays to and from the database driver emerges as the main cause of this unexpected result.

Finally, the plain file implementation outperforms the other two implementations. The `SdmDaoRF` class stores the hard location's data in a pair of random access files in the file system. Minimizing the data copy operations and optimizing the file system calls by reserving the whole disk space requirement at the beginning, the memory achieved a performance up to five times better than the other two models.

Although the results strongly bias the selection of the plain file approach, further testing, including other database models, is required before discarding the database implementations. Moreover, the scalability and distributivity characteristics of database systems, both relational and non-relational, make them attractive choices for distributed environments, even in light of the above-mentioned disadvantages.

Any of these cached implementations suffice to run the SDM variations described in this work, even on computers of modest capabilities. They also provide a persistence mechanism as a valuable side effect of the cache architecture: After performance of a simulation, the secondary storage preserves the memory information, thus a future simulations can reuse the stored data.

Parallel and Distributed Implementations

In the last decade, parallel processing has become ubiquitous. Nowadays, it is common to have multi-core CPUs executing instructions in parallel, even in desktop and laptop computers. Furthermore, Graphic Processors Units (GPUs), which can perform billions of parallel vector operations per second, are often found even in mid-range computers. Cloud computing, a metaphor for the delivery of computing processing as a utility service, provides cheap, almost unlimited processing power that, in general, relies on multithreading and distributed processing. This paradigm is an attractive option for memory- and processing-intensive AI applications, including the SDM extensions described in this dissertation. For example, Google Inc. has recently proposed a *cloud robotics* platform to help developing smart robots using the processing power of cloud computing (Guizzo, 2011).

Instead of using the low level threading support of Java, the multithreading implementation presented here utilizes Akka, an actors model framework (Hewitt et al., 1973). The actors model, a theoretical model of concurrent computation, defines actors as its primitive elements. The actors communicate only through messages, and there is no global state of the system. In response to received messages, an actor can modify its local state, send messages to other actors, and create new actors. The object paradigm differs from the actor model in that the former typically executes sequentially and the latter is inherently concurrent and asynchronous. The Akka framework implements the actors model in Java (and in Scala), abstracting from some of the inherent complexities of concurrent programming. Furthermore, this framework hides the implementation details of distributed execution from the programmer; after defining the actors, they can be executed locally or distributively over a network.

The Integer SDM implementation using Akka defines a number of classes for actors and messages. Whereas the messages are simple objects that encapsulate each operation and do not need further analysis, the new actor classes and the changes in some of the base classes require additional discussion. By dividing the functionality of the sparse distributed memory class, the implementation better supports the concurrent design. The new `HardLocationPool` interface and its several implementations encapsulate the control of the hard location's collection, leaving only the high level functionality of the memory to the `SparseDistributedMemoryImpl` class. The hard location pool's variants implement different functionalities, including the cached and multithreaded versions. Figure 27 displays the UML diagram corresponding to this new design.

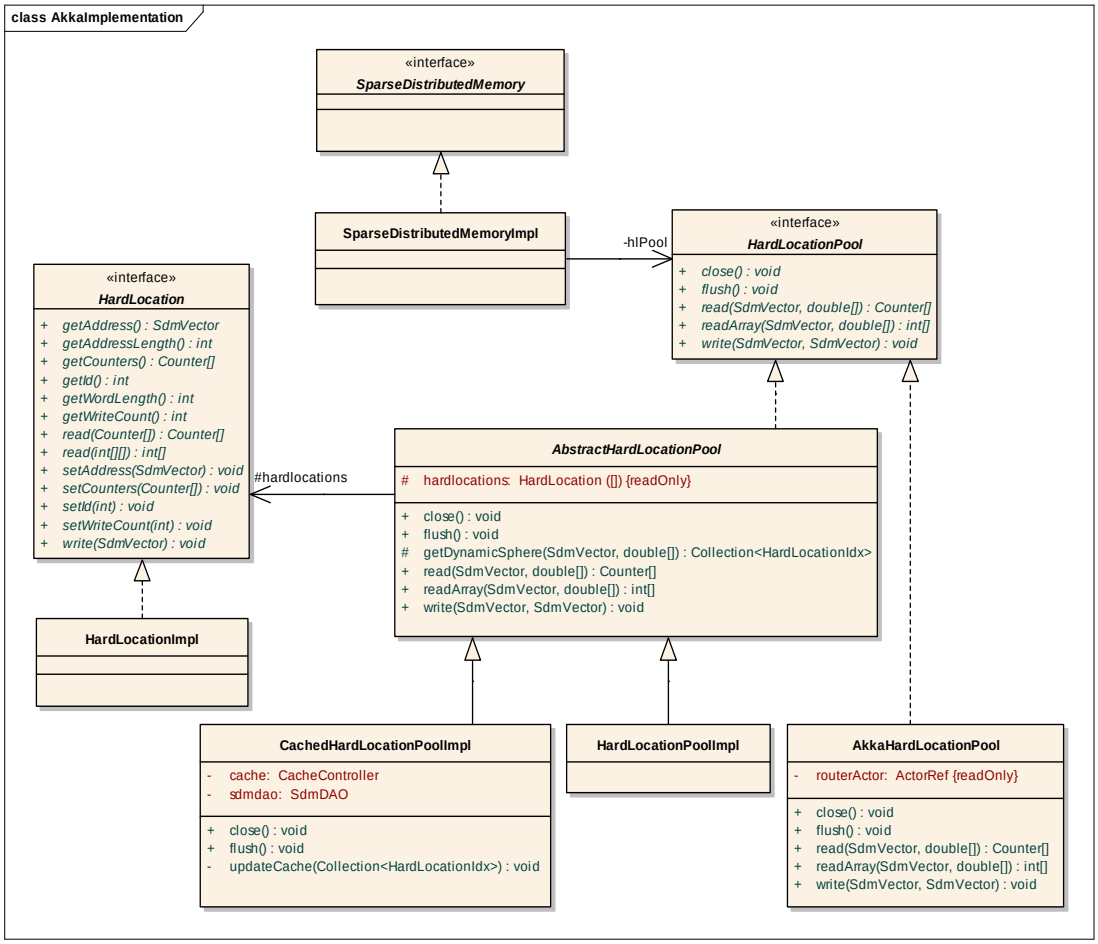


Figure 27. UML class diagram of some of the classes that support the Akka actor implementation. To improve clarity, some class members were not included in the diagram.

The `AkkaHardLocationPool` connects the SDM with the Akka framework. This class has an `SdmRouterActor` which in turn includes a collection of `SdmActor` actors. Each `SdmActor` actor has a hard location pool. Furthermore, the router actor can include other router actors in its actor collection, implementing a tree-like structure of actors that resembles the composite design pattern (Gamma et al., 1995). Each leaf of this tree has a hard location pool, and due to the actor model capabilities, the access to each pool executes concurrently. Some of the router actors (and its `SdmActor` children) can

actually be remotely instantiated, making the design distributed as well. See Figure 28 for details. When the sparse distributed memory class invokes `AkkaHardLocationPool`'s read or write methods, it sends a message to the router actor, which in turn broadcasts the message to its children. In response to the message, the children actors of type `SdmActor`, concurrently read from or write to their own hard location pool, and send a message back to the sender. The children actors of type `SdmRouterActor` broadcast the message down the hierarchy.

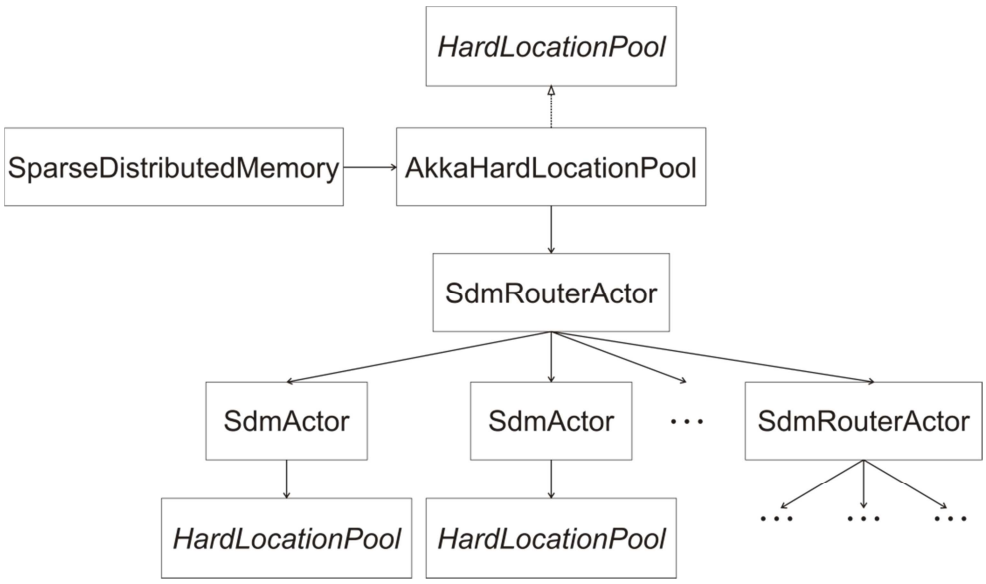


Figure 28. Hierarchy of actors used in the SDM Akka implementation.

Several Integer SDM experiments utilized this implementation, using both multithreading and distributed support, running in a high performance computer (HPC), which consists of a Beowulf (Linux) cluster of 133 Penguin Computing compute nodes. The nodes used for the experiments have 8 processors (2.5Ghz AMD Opteron 2380's) and 32GB of memory, and are connected via DDR InfiniBand. The experiments

employed configurations with one node (8 processors), two nodes (16 processors), and four nodes (32 processors). The performance using one node was almost five times faster than running the same experiment without concurrency. The framework and threading overhead explain why the performance does not achieve the theoretical eight-fold enhancement. Using two or four nodes (up to 32 processors) allows creating large Integer SDM instances, impossible to achieve in smaller configurations. Although the performance degrades due to the communication overhead, the experiments prove the viability of this design for distributed computing.

GPU Processing Support

GPUs, originally created for graphic processing, have expanded their application spectrum to other computation intensive fields, such as physics and AI, which have benefited from their parallel processing capabilities. These devices comprise many simple *cores* that can execute the same code with different data in parallel, following the SIMD architecture, and making them ideal for vector or matrix processing. The GPUs work as coprocessors of the host processor. A program using this paradigm has sections that run sequentially on the host, and other sections that run in parallel on the GPU. This architecture has a memory hierarchy that comprises a global memory common to all processes in the GPU, a private memory for each GPU's core, and a memory space shared by the running cores. Although optimizing aspects such as data copy and memory allocation across this hierarchy can improve the overall performance, these considerations are outside the scope of the present work. To implement GPU support, the de facto standard in the industry and academia is the CUDA GPU programming toolkit, developed by NVIDIA for their GPUs (Che et al., 2008; NVIDIA, 2012).

The Extended SDM implementation with CUDA support utilizes parallel processing to calculate the distances among vectors when the access sphere is determined (see Chapters 2 and 4), and to access the counters of the hard locations in the reading and writing operations. Two new classes, `CudaHardLocationPool` and `CudaUtils`, encapsulate most of the code that interfaces with the *kernels*, the CUDA subroutines.

The addresses and counters of the hard locations were allocated in the global memory of the GPU, minimizing the memory copy to and from the host. Five kernels, developed in C with CUDA extensions, provide the algorithms for the functionality of the memory: `initSdm`, `write`, `read`, `normalize`, and `getDistance`. The `initSdm` kernel creates the hard locations in the GPU memory. The `write` and `read` kernels perform the basic operations of the memory, supporting the low level details of the `HardLocationPool` interface (see Figure 27 for details). The `read` kernel produces a vector of integers with the sums of the counters in each dimension corresponding to the hard locations in the access sphere. This vector must be normalized to obtain the output binary vector, but due to the parallel execution of the kernel in the GPU, the normalization must be performed using a separate kernel. Finally, the `getDistance` kernel calculates the distance between two vectors, and the `read` and `write` kernels call it to determine which hard locations are inside the access sphere.

This CUDA implementation was tested using a GPU NVIDIA GeForce GTX 560 Superclocked 2048 MB GDDR5 with 336 CUDA cores. The experiments run with this hardware showed an impressive gain in performance. In an Extended SDM with 500,000 hard locations, an address size of 1,000, and word size of 2,000, the reading and writing operations ran 52 times faster than when the object oriented implementation (see above)

was used. This result may be further improved optimizing the memory usage and fine tuning the thread execution. Moreover, the newest GPUs have up to 3,073 CUDA cores[5], almost ten times more than the one employed here. Using these new GPUs would improve the performance of this implementation even more. These results demonstrate the feasibility of these memories for real time applications with a large number of hard locations, such as robot controllers or visual recognition.

MCR Parser and Interpreter

Although hyperdimensional computing (Kanerva, 2009) using MCR vectors can be implemented using general-purpose programming languages (GPL) such as Java or C++, the syntaxes and native structure of these languages obfuscate the simplicity of the hyperdimensional computing expressions (see Chapters 3 and 6 for examples of MCR expressions). A specific scripting language, that allows writing MCR expressions, was developed using Javacc, a Java parser generator that produces a parser in Java code from a grammar specification. A runtime interpreter, implemented also in Java, can evaluate the MCR scripting language and maintain the MCR vectors in memory.

Figure 29 shows an example of the MCR scripting language, which reproduces the expressions of the hyperdimensional computing example presented in Chapter 6. Variables, such as **cause**, **idmark**, and **luke**, represent vectors. The plus sign (+) stands for the grouping operation, whereas the product sign (*) represents the binding operator. The * has precedence over the +, and parentheses can be used to force a desired operation's evaluation order. The exclamation symbol (!) produces the inverse of its

[5] The top of the line NVIDIA GPU as today is GeForce GTX 690. See http://www.geforce.com for details.

succeeding vector, useful for the probe operation (See Chapters 3 and 6 for details). Also, the slash (`/`) is equivalent to multiplication by the inverse of the second operand, which is a compact syntax for probing.

Several instructions complement the scripting language: `newrnd()` creates new random vectors, `print()` outputs a message to the console, `printd()` displays the distance among two vectors, and `rank()` displays the rank of closest vectors in the system, that is, all the vectors assigned to a variable so far, to a given vector. Appendix B lists the complete grammar definition of the MCR scripting language in Javacc format.

The MCR interpreter runs inside a Java program, and can process expressions defined in a text file or embedded in the code as strings. The execution of the interpreter can be interleaved with normal Java code, rendering it unnecessary to include in the scripting language the typical control structures, such as if and for-loops, found in most GPLs. In effect, if we need to repeat one or several MCR expressions, we can wrap the interpreter execution by a for-loop in the Java code. A hash table data structure, with the vector's variable names as keys, holds the vectors created by the interpreter. Since both the Java code and the interpreter have access to this hash table, the Java code can manipulate these vectors, or even create new ones that are included in the interpreter repertory. This easy interaction between the native Java code and the MCR interpreter enables the creation of experiments and applications using the "best of both worlds": the simplicity of the MCR scripting language combined with the power and versatility of Java.

```
//Create random base vectors
newrnd(cause, eat, see, being, human,state,food,fish,bread,object,agent);

//Create random id vectors
newrnd(idmark,idjohn,idpaul,idluke);
newrnd(idthebread,idthefish,idhunger,idthirst,ideatagt,ideatobj,
            idseeagt,idseeobj,idcauseagt,idcauseobj);

//people vectors
mark = being + human + idmark;
john = being + human + idjohn;
paul = being + human + idpaul;
luke = being + human + idluke;

//Other vectors
thefish = food + fish + idthefish;
thebread = food + bread + idthebread;
hunger = state + idhunger;
thirst = state + idthirst;

eatagt = agent + ideatagt;
eatobj = object + ideatobj;

seeagt = agent + idseeagt;
seeobj = object + idseeobj;

causeagt = agent + idcauseagt;
causeobj = object + idcauseobj;

//events
s1 = eatagt * mark + eatobj * thefish + eat;
s2 = cause +causeagt * hunger + causeobj*s1;
s3 = eat + eatagt*john;
s4 = see + seeagt*john+seeobj*mark;
s5 = see + seeagt * john + seeobj* thefish;
s6 = see + seeagt * thefish  + seeobj* john;

//probes
printd (s1, eat);
print("");
rank("s1/eatagt",s1*!eatagt,5);
rank("s1/agent",s1/agent,5);
rank("s1/eatobj",s1/eatobj,5);

rank("s2/causeagt",s2/causeagt,5);
rank("s2/causeobj",s2/causeobj,5);
rank("s2/causeobj/eatagt",s2/causeobj/eatagt,5);
rank("s2/causeobj/eatobj",s2/causeobj/eatobj,5);

rank("s3/eatobj",s3/eatobj,5);
rank("s4/john",s4/john,5);
rank("s5/john",s5/john,5);
rank("s6/john",s6/john,5);
```

Figure 29. Example of MCR scripting expressions. This example reproduces the experiment presented in Chapter 6.

Conclusions

This chapter presents the software implementations of the technologies introduced in this dissertation. Cached versions of the Extended and Integer SDM allow running these memories even in modest computers. The various parallel implementations introduced here, including multithreading, distributed, and SIMD variants, have demonstrated the feasibility of SDM and related models, to take advantage of the incipient trend of parallel computing. Further work must address optimization of these designs to improve their performance and scalability. Finally, the MCR scripting language interpreter simplifies the implementation of experiments and applications based on MCR vectors, without the burden of the syntactic overhead of the host language.

Chapter 8: Conclusions

Cognitive software agents, robot controllers, and other similar challenging AI applications have several basic operations in common. These operations, described in Chapter 1, include pattern recognition when partial and noisy cues are used, sequence learning, generalization of patterns, and Hebbian learning. A memory system for these applications can facilitate the implementation of these operations. SDM has proven to be a good candidate. It possesses some of the desirable features for memory systems listed in Chapter 1: content addressability, auto-associativity and hetero-associativity, robustness to noise, generalization, clustering, pattern recognition, sequence learning, resilience to memory damage, one-shot and incremental learning, forgetting, and high dimensionality. The SDM extensions presented in this dissertation, which further improve these features, and MCR, the new reduced description model introduced in this work, integrate a set of technologies with the potential to address the complexities of challenging AI applications.

The rest of this chapter will describe some further directions and possible applications of the technologies introduced here, followed by a discussion of their limitations. Finally, I will summarize the conclusions and cite this author's papers related to this work. Appendix A includes a complete list of papers written by the author.

Further Directions

Several extensions and variations of Extended SDM and Integer SDM are natural paths of further development. First, a forgetting mechanism (Ramamurthy, D'Mello et al., 2006), which will help to preserve only the most often repeated elements in the memory, would improve the unsupervised learning capability of the memory. Only correct inputs

and associations are likely to be repeated frequently, and then incorrect inputs would decay away from the memory without any supervision. By balancing the new inputs and the decay rate, this mechanism would also prevent the memory from approaching its maximum capacity.

Other designs of SDM hard location activation, like Jaeckel's selected coordinate design (Jaeckel, 1989a, 1989b), can also be implemented with these SDM extensions, improving the signal to noise ratio. Moreover, other designs, such as the ones proposed by (Anwar et al., 1999; Fan & Wang, 1997; Keeler, 1988; Ratitch & Precup, 2004) and reviewed in Chapter 2, utilize variations in the distribution of the hard locations that improve the performance of SDM when the data to be stored are not uniformly distributed in the space. Exploring these variations is also an attractive further direction.

Random indexing (Sahlgren, 2005), a semantic space model that creates semantic vectors by combining random vectors associated with each word, is a possible application of Extended SDM. In the random indexing model, each word has two associated vectors: a random vector, and a semantic vector, the latter being the result of combining the random vectors of other words related to this one. The process can be iterative, refining the semantic vector as new related words appear. Extended SDM has the potential to produce semantic vectors directly during word storage. The data vector (see Chapter 4 for details) can hold the random vector and the semantic vector in two sections that are updated whenever new data arrives. With this implementation, the memory would still preserve its noise robustness capability, and would additionally create the semantic vectors that relate the words according to their meaning.

In recent years, several models of the so called *deep learning* systems, such as HMAX (Serre et al., 2007), HTM (George, 2008), DeSTIN (Arel et al., 2009), and deep belief nets (Hinton, 2007; Hinton et al., 2006), have emerged. These models, based on the hierarchical organization of the neocortex, and of the visual cortex, focus on learning and recognition of spatial and temporal patterns. They detect pattern invariances in space and (in some models) in time in each level of the hierarchy. The output of a lower layer provides the input for the higher ones. The higher the layer, the more abstract are the features they capture of the data. A possible deep learning system could use layers of Extended Integer SDMs. The memory that implements each layer stores the input vectors, and its interference and generalization properties facilitate the creation and detection of patterns from similar vectors (see an experiment of Chapter 5 that shows this mechanism). Finally, the sequence storage mechanism described in Chapter 4 helps to learn temporal patterns in each layer.

Vector LIDA

A promising project that I called Vector LIDA would intensively utilize the technologies presented in this dissertation. This project would implement the LIDA architecture (Franklin & Patterson, 2006; Snaider et al., 2011) using MCR vectors as its main representation for data structures, and the various extensions of SDM presented here for its main memory mechanisms. The LIDA architecture was briefly introduced in Chapter 2, and a recent description of the model can be found in (Franklin, Strain, Snaider, McCall, & Faghihi, in press). For reference, Figure 30 depicts the structure of the LIDA model, including its modules and their interactions.

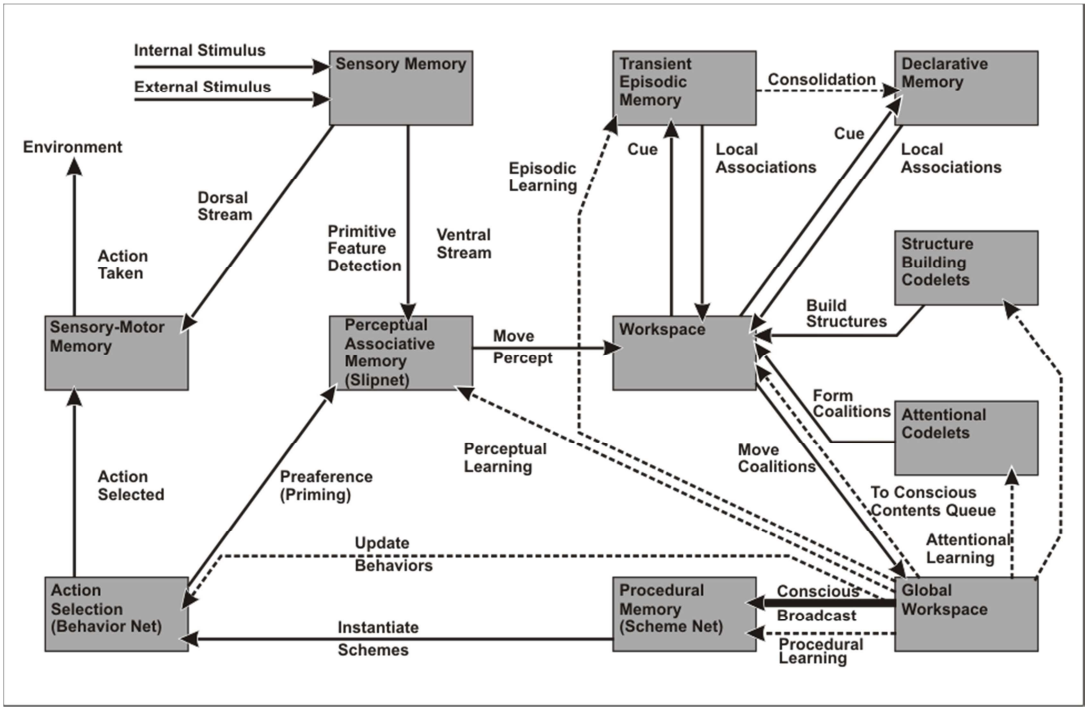


Figure 30. LIDA cognitive model diagram. The boxes represent the different modules of the model, and the arrows the interactions among them.

The current version of the LIDA model utilizes nodes and links in a graph-like structure (node structure) as its main data structure. This implementation introduces several problems. First, comparing node structures can be computationally expensive. Moreover, some of LIDA's processes require approximate comparisons of the node structures, which can be even harder to compute. MCR vectors can represent information such as that contained in node structures, but unlike node structures, MCR vectors have an innate approximate comparison property, as explained in detail in Chapters 3 and 6.

Second, some modules in the LIDA model, such as perceptual associative memory, episodic memory, and procedural memory, require the implementation of learning mechanisms. These mechanisms must be able to learn new node structures in an instructionalist learning mode, and reinforce previous ones via reinforcement learning.

The current model uses a value attached to each node and link, called base level activation, that helps to implement reinforcement learning. However, the model does not have a generic strategy for the learning of new elements, and the current implementations of several modules do not scale well. The SDM variations presented here have both the required learning mechanisms (instructionalist and reinforcement) integrated into their basic functionality. Learning new vectors (instructionalist learning) simply consists of storing the vector in the memory using its standard storage procedure. When the same vector is stored several times (reinforcement), the hard locations' counters corresponding to the values of each dimension of this vector will have larger counts, making it resistant to interference by other vectors stored in the memory. This effect would improve implementing a forgetting mechanism.

Moreover, the current episodic memory module in LIDA already employs a SDM memory as its base implementation. The problem of translating back and forth from node structures to vectors in episodic memory disappears when using MCR vectors as the main data structure of LIDA. Furthermore, the sequence storage mechanism of Extended SDM would enable the episodic memory module to store composite events, sequences of simpler events, improving the event-learning capability of the episodic memory module.

Third, MCR vectors have the potential of implementing directly Barsalou's perceptual symbol system (1999), which uses symbols grounded in sensory and motor information. Although the current LIDA model employs a version of perceptual symbols, it does not exploit their capability for expressiveness, and they have a limited impact on the functionality of the whole system. Nodes in LIDA are grounded in sensory data. The activation of a node depends on the activation of its child nodes, which eventually are

activated from sensory data. However, a node (without considering its children) does not represent any specific sensory or motor information by itself, so its grounding feature is seldom employed in the LIDA model processes. Moreover, the simulator idea, central to the perceptual symbol system theory, is hard to implement using nodes and node structures. On the other hand, constructing MCR vectors from sensory and motor information using hyperdimensional computing operations would produce representations that have many of the perceptual symbols' characteristics described by Barsalou (1999). Similar sensory information would yield similar representations, and the holistic processing operations of MCR could facilitate the implementation of the simulators described in his model. Interestingly, MCR vectors with role-filler components for each modality have the potential to integrate several modalities in a single representation, addressing the so called binding problem. For example, the MCR vector *B*

$$B = [birdImage \otimes Visual + birdSong \otimes Auditory] \tag{153}$$

may represent the integration of the data from the visual and auditory modalities. Notice that the vectors *birdImage* and *birdSong* would be in turn reduced descriptions also.

Fourth, the hierarchical networks described in the previous section provide biologically plausible mechanisms with which to perceive both spatial and temporal patterns from low level sensory data, making them attractive for modeling low level perception between sensory memory and perceptual associative memory (PAM) in LIDA (see Figure 30). Since these models in general produce high dimensional vectors as output, interfacing them with Extended Integer SDM memories for implementing PAM

would be simpler, more scalable, and more noise robust than with the current implementation. HMAX (Serre et al., 2007) is probably the most biologically realistic hierarchical model for this function, since their authors designed it following the biological data as accurately as possible, but other models such as HTM (George, 2008) or DeSTIN (Arel et al., 2009) are also possible options. Furthermore, these hierarchical models have the potential of detecting spatial-temporal patterns in other modules, such as the workspace or perceptual memory, and they would seamlessly integrate with MCR vectors. For example, attention and structure-building codelets (see Figure 30) can be implemented with these hierarchical networks so as to detect patterns in the workspace, and build coalitions and complex structures, respectively, with these patterns. A similar implementation for procedural memory, using hierarchical networks, could improve the detection and learning of temporal patterns that eventually became sequences of actions or *behavior streams* (D'Mello, Ramamurthy, Negatu, & Franklin, 2006). These hierarchical network models, combined with MCR vectors and Extended Integer SDM, have the potential to provide a primary detection algorithm in LIDA.

Finally, using MCR vectors would produce a more biologically plausible model through its synergy with other models, such as the hierarchical networks mentioned above, Barsalou's perceptual symbols, Fuster's *cognits* (2006), and several neurodynamical theories (Franklin et al., in press). I have already described (see above) how to implement perceptual symbols, and how their construction addresses the binding problem. A discussion follows of the relationship between MCR vectors and both cognits and neurodynamical theories.

Fuster defined a cognit as an abstraction of a network of neurons. Its representation power comes from the neurons that compose it and specially the relationship between its component neurons. He extensively describes how different memory types (e.g., episodic, perceptual, motor, etc.) can be interpreted as hierarchies of cognits. He pointed out that cognits in one level of this hierarchy can be a composition of other cognits from several levels in the hierarchy. MCR vectors may be used as an abstraction of the cognit model. They are also distributed, can combine elements of various levels of the memory hierarchy in a single vector, and their hyperdimensional operations can combine and associate cognits represented as vectors.

Franklin and colleges (Franklin et al., in press) have compared several neurodynamical theories with the LIDA model. By interpreting the brain as a dynamical system, the representations would be trajectories in the phase space (pattern of activation space) of one or several cell assemblies. These trajectories can in turn interfere with and influence the trajectory in the phase space of other cell assemblies. A MCR vector would model not only a pattern of activation of a cell assembly, but also a trajectory of these patterns. For example, if a single neuron in a cell assemble has a sequence of activations in a trajectory of 4 steps (e.g., 1011, where one and zero mean high firing rate and low firing rate respectively), we may code this sequence as a single value (11 in the example) and assign this value to one dimension in our MCR vector. Employing the same procedure for each neuron in the cell assembly, produces a MCR vector that represent the trajectory of the pattern. Using an Integer SDM as a cleanup memory can produce a previously stored vector from a partial vector, which would model the oscillatory and self-organizing properties of the dynamical system interpretation. Using random

permutation or multiplication produces a new vector that would model the influence from one cell assembly to another. Although these ideas are still under development, using MCR vectors and the memories proposed herein has enormous potential to model representations and cognitive processes in a more biologically plausible way.

Summing up, some of the advantages of Vector LIDA over the current implementation include a more realistic and biologically plausible model, better integration with its episodic memory, better integration with other low level perceptual processing (such as HMAX Serre et al., 2007), better scalability, and easier learning mechanisms.

Limitations

The proposed memory models have the several advantages described herein; however, they have also some limitations. First, the performance of the memories degrades if the stored vectors are not uniformly distributed in the space. The possible variations in the hard location activation mechanism mentioned in the previous section would mitigate this issue, but a more extensive study has to confirm the expected improvement.

Second, the memories discussed in this work only produce a single vector as a result of the reading operation. Although this is enough for a broad range of uses, some applications (e.g., the procedural memory module in vector LIDA) could require retrieving the set of closest vectors in the memory. A multilayer hierarchical memory might provide a possible path for addressing this issue.

Third, Integer SDM used as a cleanup memory for an MCR reduced description model does not always yield the expected vector due to the excessive noise introduced by

the MCR operations (see examples in Chapter 6). Other ways to improve the noise robustness of the memory need to be explored to solve this problem.

Finally, MCR vectors can integrate several vectors into one, but if the number of combined vectors is too large, the composite vector becomes useless due to the noise introduced in the representation. Exploring sparse vector representations–vectors with a small number of significant dimensions compared to the total number of dimensions–might improve the performance of MCR vectors.

Summary of Conclusions

The first variation of SDM presented here, Extended SDM, increases the hetero-associativity feature of the memory without diminishing its auto-associativity. This variation is particularly efficient for learning sequences and other data structures such as trees. Furthermore, the novel mechanism for sequence storage described in Chapter 4 allows the inclusion of sequences of degree greater than one, crossing sequences–sequences with common elements–and sequence recall from a middle point to the end. Previously, this kind of sequence learning was only possible in SDM with complex architectures such as the one described by Kanerva (1988) or the one implemented by Jockel (2009). I also analyzed the effect of the parameter *k* (see Chapter 4) to fine-tune the behavior of the memory for sequence learning. This parameter controls the number of previous elements required to retrieve the next element in a sequence, thereby controlling the grade of the sequences that the memory can learn. Two papers have already been accepted or published discussing this memory and its applications: (Snaider & Franklin, 2011; Snaider & Franklin, 2012a).

Another extension presented here, the Integer SDM, extends the domain of the memory to accept integer vectors, with a range of possible values for each dimension. Real world data are often non-binary, thus a memory able to store values other than binary can be more effective for applications that use such values. The integer representation has several advantages over the binary one. The encoding of values is simpler, avoiding undesirable effects of other encodings (Jockel, 2009; Mendes et al., 2009), and it diminishes the effect of normalization when several vectors are combined, for example in the storage and retrieval of sequences (Snaider & Franklin, 2011). The benefits of this model are retained when merged with Extended SDM into a combination SDM possessing integer vectors, better support for hetero-associativity, and improved sequence learning.

Integer SDM as a cleanup memory is also a good companion for the Modular Composite Representation. Reduced descriptions using large vectors, such as Spatter Code and HRR, require an auto-associative memory to clean up not only noisy input vectors, but also those produced as the result of operations between other vectors. These operations, such as sum or multiplication, often produce noisy versions of the target vectors. The auto-associative memory helps clean up these vectors.

Both theoretical and empirical analyses of the capacity of Integer SDM were presented in this dissertation. The results of the experiments match the theoretical predictions, and demonstrate the potential of the system. A first paper describing this memory has already been published (Snaider & Franklin, 2012b). A second paper that describes the theoretical analysis of this memory, and related experiments, has been submitted for review (Snaider, Franklin, Strain, & George, in review).

I also defined and empirically tested Modular Composite Representation (MCR), a new reduced description representation based on modular integers. It improves on two earlier reduced description models (Hinton, 1990): the binary Spatter Code (Kanerva, 1994) and the Holographic Reduced Representation (Plate, 1995, 2003). The former uses large binary vectors and simple operations, such as XOR, to produce a reduced description model able to represent complex structures or hierarchies as a whole. The use of binary vectors limits the model's expressiveness, and some required operations such as normalization introduce excessive noise into the vectors that can diminish the performance of the model. On the other hand, Holographic Reduced Representation (HRR), based on large real-numbered vectors, has a rich representation capability, but it requires complex operations such as circular convolution. Moreover, the vectors must follow a normal distribution for each dimension, which further complicates its use. MCR is an intermediate point between these two models, balancing representational expressiveness and implementational simplicity.

The detailed presentation of MCR includes a complete description of the model and its operations. Some examples of different uses and applications were also presented, including the integration of Integer SDM as a cleanup memory. The experiments and analysis detailed herein have demonstrated MCR's performance in a number of scenarios, empirically validating its anticipated noise robustness, representational expressiveness, and holistic processing capability. The analysis of the means and variances for the similarities of representative operations suggests that MCR has better performance for these operations than either HRR or Spatter Code using vectors with the same number of

dimensions. A paper describing the MCR model is in review (Snaider & Franklin, in review).

Chapter 7 demonstrates that the extensions of SDM presented here are well suited for parallel implementation. Several implementations were described and tested. The first realization uses a least recently used (LRU) cache and a database. Another implementation uses a state of the art parallel framework, the Akka framework, which implements the actors model (Hewitt et al., 1973). This implementation, able to run as a multithreading application or in a distributed architecture, outperforms the single-thread implementation, proving the potential of these SDM variations for running in parallel and on distributed hardware. Finally, a third implementation explores the parallel vector architecture supported by modern GPUs. This computational paradigm has a SIMD (Single Instruction Multiple Data) structure that is ideal for SDMs due their vector structure.

Finally, I described further directions and possible applications of this research, including the use of the extended SDM, Integer SDM, and MCR representations as the main technologies for implementing the LIDA cognitive architecture. A paper introducing the LIDA computational framework, the base for future developments, has already been published (Snaider et al., 2011). I am preparing a position paper that includes the requirements for representations involved in challenging AI applications as described in Chapter 1, and the advantages of the vector LIDA project. This project shows how all the technologies that comprise this work can be used together to enhance their features. Other possible extensions include deep learning using Extended SDM and a multi-layered version of these memories.

References

Albus, J. S. (1971). A theory of cerebellar function. *Mathematical Biosciences, 10*, 25-61.

Albus, J. S. (1981). *Brains, Behavior, and Robotics*: BYTE/McGraw-Hill.

Altmann, E. M., & Gray, W. D. (2002). Forgetting to remember: the functional relationship of decay and interference. *Psychological Science, 13*(1), 27-33.

Anwar, A., Dasgupta, D., & Franklin, S. (1999). *Using Genetic Algorithms for Sparse Distributed Memory Initialization.* Paper presented at the International Conference Genetic and Evolutionary Computation (GECCO).

Anwar, A., & Franklin, S. (2005). A Sparse Distributed Memory Capable of Handling Small Cues, SDMSCue. In M. K. Ng, A. Doncescu, L. T. Yang & T. O. Leng (Eds.), *High Performance Computational Science and Engineering: IFIP TC5 Workshop on High Performance Computational Science and Engineering (HPCSE), World Computer Congress, August 22–27, 2004, Toulouse, France* (pp. 23–38). New York, NY: Springer Science+Business Media Inc.

Araujo, A. F. R., & Barreto, G. A. (2002). Context in temporal sequence processing: a self-organizing approach and its application to robotics. *IEEE Transactions on Neural Networks, 13*(1), 45-57.

Arel, I., Rose, D., & Coop, R. (2009). DeSTIN: A Scalable Deep Learning Architecture with Application to High-Dimensional Robust Pattern Recognition. *Proc. of the AAAI 2009 Fall Symposium on Biologically Inspired Cognitive Architectures (BICA).*

Baars, B. J. (1988). *A Cognitive Theory of Consciousness*. Cambridge, UK: Cambridge University Press.

Baars, B. J., & Franklin, S. (2003). How conscious experience and working memory interact. *Trends in Cognitive Sciences, 7*(4), 166-172.

Baars, B. J., & Franklin, S. (2009). Consciousness is computational: The LIDA model of Global Workspace Theory. *International Journal of Machine Consciousness, 1*(1), 23-32.

Baddeley, A. D., Conway, M., & Aggleton, J. P. (2001). *Episodic Memory*. Oxford, UK: Oxford University Press.

Barreto, G. A., & Araujo, A. F. R. (2004). Identification and control of dynamical systems using the self-organizing map. *IEEE Transactions on Neural Networks, 15*(5), 1244-1259.

Barsalou, L. W. (1999). Perceptual symbol systems. *Behavioral and Brain Sciences, 22*, 577–609.

Block, R. (1990). *Cognitive Models of Psychological Time*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Blumenthal, L. M., & Menger, K. (1970). *Studies in geometry*. San Francisco, CA: Freeman.

Bose, J., Furber, S. B., & Shapiro, J. L. (2005). Spiking neural sparse distributed memory implementation for learning and predicting temporal sequences. *Lecture Notes in Computer Science, 3696/2005*, 115 - 120.

Brooks, R. A. (1986). A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation, RA-2*(1), 14-23.

Brooks, R. A. (1991). Inteligence without represention. *Artificial Intelligence, 47*, 139-159.

Brown, J. (1958). Some tests of the decay theory of immediate memory. *Quarterly Journal of Experimental Psychology, 10*, 12-21.

Chalmers, D. J. (1990). Syntatic Transformations on Distributed Representations. *Connection Science, 2*(1-2), 53-62.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., & Skadron, K. (2008). A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing, 68*(10), 1370-1380.

Chou, P. A. (1989). The capacity of the Kanerva associative memory. *IEEE Trans. Information Theory, 35*(2), 281-298.

Clarke, T., Prager, R. W., & Fallside, F. (1991). The modified Kanerva model: Theory and results for real-time word recognition. *IEE Proceedings of Radar and Signal Processing, 138*(1), 25-31.

Cohen, T., & Widdows, D. (2009). Empirical distributional semantics: Methods and biomedical applications. *Journal of Biomedical Informatics, 42*(2), 390-405.

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*.

D'Mello, S. K., Ramamurthy, U., Negatu, A., & Franklin, S. (2006). A Procedural Learning Mechanism for Novel Skill Acquisition. In T. Kovacs & James A. R.

Marshall (Eds.), *Proceeding of Adaptation in Artificial and Biological Systems, AISB'06* (Vol. 1, pp. 184–185). Bristol, England: Society for the Study of Artificial Intelligence and the Simulation of Behaviour.

Danforth, D. G. (1990). *An empirical investigation of sparse distributed memory using discrete speech recognition* (No. Technical report 90.18): Research Institute for Advanced Computer Science, NASA Ames Research Center.

Deerwester, S. C., Dumais, S. T., Furnas, G. W., Landauer, T. K., & Harshman, R. A. (1990). Indexing by latent semantic analysis. *Journal of the American Society of Information Science, 41*(6), 391-407.

Dolan, C. P. (1989). *Tensor manipulation networks: connectionist and symbolic approaches to comprehension, learning, and planning*. Los Angeles, CA: UCLA, Computer Science Department.

Ebbinghaus, H. (1885). *Memory: A contribution to experimental psychology*. New York, NY: Dover.

Fan, K. C., & Wang, Y. K. (1997). A genetic sparse distributed memory approach to the application of handwritten character recognition. *Pattern Recognition Letters, 30*(12), 2015-2022.

Fei-Fei, L., Fergus, R., & Perona, P. (2006). One-Shot learning of object categories. *IEEE Trans. Pattern Analysis and Machine Intelligence, 28*(4), 594 - 611.

Foundalis, H. E. (2006). *PHAEACO: A Cognitive Architecture Inspired by Bongard's Problems. PhD Thesis.*, Indiana University, Indiana.

Foundalis, H. E., & Martinez, M. (2007). A Generalization of Hebbian Learning in Perceptual and Conceptual Categorization. In S. Vosniadou, D. Kayser, & A. Protopapas (Eds.), *Proceedings of the European Cognitive Science Conference 2007*. Delphi, Greece.

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

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

Franklin, S., & Graesser, A. C. (1997). Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents *Intelligent Agents III* (pp. 21–35). Berlin: Springer Verlag.

Franklin, S., & Patterson, F. G. J. (2006). The LIDA Architecture: Adding New Modes of Learning to an Intelligent, Autonomous, Software Agent *IDPT-2006 Proceedings*

(Integrated Design and Process Technology): Society for Design and Process Science.

Franklin, S., Strain, S., Snaider, J., McCall, R., & Faghihi, U. (in press). Global Workspace Theory, its LIDA Model and the Underlying Neuroscience. *Biologically Inspired Cognitive Architectures, 1*.

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

Fuster, J. M. (2006). The cognit: A network model of cortical representation. *International Journal of Psychophysiology, 60*(2), 125-132.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. M. (1995). Design Patterns: Elements of Reusable Object-Oriented Software: Addison-Wesley Professional.

George, D. (2008). *How the brain might work: A hierarchical and temporal model for learning and recognition. PhD Thesis.*, Stanford University.

Giles, C. L., Horne, B. G., & Lin, T. (1995). Learning a Class of Large Finite State Machines with a Recurrent Neural Network. *Neural Networks, 8*(9), 1359-1365.

Guizzo, E. (2011). Robots with their heads in the clouds. *IEEE Spectrum, 48*(3), 16-18.

Hawkins, J. (2005). *On Intelligence*. New York, NY: Owl Books.

Hawkins, J., & Blakeslee, S. (2007). Why can't a computer be more like a brain. *IEEE Spectrum, 44*(4), 20-26.

Hely, T., Willshaw, D. J., & Hayes, G. (1997). A new approach to Kanerva's sparse distributed memory. *IEEE Transactions on Neural Networks, 8*(3), 791-794.

Hewitt, C., Bishop, P., & Steiger, R. (1973). *A universal modular ACTOR formalism for artificial intelligence*. Paper presented at the Proceedings of the 3rd international joint conference on Artificial intelligence.

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

Hinton, G. E. (2007). Learning multiple layers of representation. *TRENDS in Cognitive Sciences, 11*(10), 428-434.

Hinton, G. E., McClelland, J. L., & Rumelhart, D. E. (1986). Distributed representations. In D. E. Rumelhart & J. L. McClelland (Eds.), *Parallel Distributed Processing:*

Explorations in the Microstructure of Cognition. Volume 1: Foundations (pp. 77-109). Cambridge, MA: MIT Press.

Hinton, G. E., Osindero, S., & Teh, Y. (2006). A fast learning algorithm for deep belief nets. *Neural Computation, 18*, 1527-1554.

Hofstadter, D. R. (1995). *Fluid Concepts and Creative Analogies*. New York, NY: Basic Books.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *In Proceedings of the National Academy of Science, 79*, 2554-2558.

Howell, R. M., & Fowler, D. (1990). A Neural Network as an Instrument of Prediction. In R. A. Miller (Ed.), *Procedings of the Annual Symposium on Computer Application in Medical Care* (pp. 299-302). Washington DC: IEEE Computer Society Press.

Jaeckel, L. A. (1989a). *An Alternative Design for a Sparse Distributed Memory.* (No. Report RIACS TR 89.28): Research Institute for Advanced Computer Science, NASA Ames Research Center.

Jaeckel, L. A. (1989b). *A Class of Designs for a Sparse Distributed Memory* (No. Report RIACS TR 89.30): Research Institute for Advanced Computer Science, NASA Ames Research Center.

Jockel, S. (2009). *Crossmodal Learning and Prediction of Autobiographical Episodic Experiences using a Sparse Distributed Memory. PhD Thesis.*, University of Hamburg, Hamburg.

Joglekar, U. D. (1989). *Learning to read aloud: A neural network approach using sparse distributed memory* (No. RIACS 89.27): Research Institute for Advanced Computer Science, NASA Ames Research Center.

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

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

Kanerva, P. (1993). Sparse Distributed Memory and related models. In M. H. Hassoun (Ed.), *Associative Neural Memories: Theory and Implementation* (pp. 50-76). New York, NY: Oxford University Press.

Kanerva, P. (1994). The binary spatter code for encoding concepts at many levels. In M. Marinaro & P. Morasso (Eds.), *ICANN '94: Proceedings of International*

Conference on Artificial Neural Networks (Vol. 1, pp. 226–229). London, UK: Springer-Verlag.

Kanerva, P. (1996). Binary Spatter-Coding of Ordered K-Tuples. In C. von der Malsburg, W. von Seelen, J. C. Vorbrüggen & B. Sendhoff (Eds.), *ICANN 96 Proceedings of the 1996 International Conference on Artificial Neural Networks* (Vol. 1112, pp. 869-873): Springer.

Kanerva, P. (1998). Dual Role of Analogy in the Design of a Cognitive Computer. In K. Holyoak, D. Gentner & B. Kokinov (Eds.), *Advances in Analogy Research: Integration of Theory and Data from the Cognitive, Computational, and Neural Sciences (Proc. Analogy'98 workshop)* (pp. 164-170). Sofia, Bulgaria: New Bulgarian University.

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

Karlsson, R. (1995). A fast activation mechanism for the Kanerva SDM memory. *Proceedings of the RWC Symposium*, 69-70.

Keeler, J. D. (1988). Comparison between Kanerva's SDM and Hopfield-type neural networks. *Cognitive Science, 12*, 299-329.

Keppel, G., & Underwood, B. J. (1962). Proactive inhibition in short-term retention of single items. *Journal of Verbal Learning and Verbal Behavior, 1*, 153-161.

Kremer, S. C. (2001). Spatiotemporal Connectionist Networks: A Taxonomy and Review. *Neural Computation, 13*(2), 249-306.

Kruschke, J. K. (2003). Attention in Learning. *Current Directions in Psychological Science 12*, 171-175.

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

Lawrence, M., Trappenberg, T., & Fine, A. (2006). Rapid learning and robust recall of long sequences in modular associator networks. *Neurocomputing, 69*(7-9), 634-641.

Leveille, J., Ames, H., Chandler, B., Gorchetchnikov, A., Livitz, G., Versace, M., et al. (2011). *Invariant object recognition and localization in a virtual animat*. Paper presented at the International Conference on Cognitive and Neural Systems (ICCNS) 2011.

Likharev, K. K. (2009). *Biologically Inspired Computing in CMOL CrossNets*. Paper presented at the AAAI Fall Symposium on Biologically Inspired Cognitive Architecture.

Logan, G. D. (2002). An Instance Theory of Attention and Memory. *Psychological Review, 109*, 376–400.

Lopez, I., Sanz, R., Moreno, F., Salvador, R., & Alarcon, J. (2007). *From Cognitive Architectures to Hardware: A Low Cost FPGA-Based Design Experience.* Paper presented at the IEEE International Symposium on Intelligent Signal Processing. WISP 2007.

Marr, D. (1969). A theory of cerebellar cortex. *Journal of Physiology, 202*, 437-470.

Maurer, A., Hersch, M., & Billard, A. G. (2005). Extended hopfield network for sequence learning: Application to gesture recognition. *Proceedings of the ICANN 2005*.

McGeoch, J. A. (1932). Forgetting and the law of disuse. *Psychological Review, 39*, 352-370.

Mendes, M., Coimbra, A., & Crisóstomo, M. (2009). Assessing a Sparse Distributed Memory Using Different Encoding Methods. *Proceedings of the World Congress on Engineering, 1*, 1-6.

Mendes, M., Crisostomo, M., & Coimbra, A. P. (2008). *Robot navigation using a sparse distributed memory.* Paper presented at the IEEE International Conference on Robotics and Automation. ICRA 2008.

Meng, H., Appiah, K., Hunter, A., Yue, S., Hobden, M., Priestley, N., et al. (2009). *A modified sparse distributed memory model for extracting clean patterns from noisy inputs.* Paper presented at the International Joint Conference on Neural Networks (IJCNN), Atlanta, GA, USA.

Metcalfe, J. (1982). A composite holographic associative recall model. *Psychological Review, 89*, 627-661.

Murdock, B. B. (1983). A Distributed Memory Model for Serial-Order Information. *Psychological Review, 92*(1), 316-338.

Murdock, B. B. (1993). TODAM2: A Model for the Storage and Retrieval of Item, Associative, and Serial-Order Information. *Psychological Review, 100*(2), 183-203.

Newell, A., & Simon, H. A. (1976). Computer Science as Empirical Inquiry: Symbols and Search. *Communications of the ACM, 19*(3), 113-126.

NVIDIA. (2012). *CUDA C Programming Guide V.4.2*. Santa Clara CA: NVIDIA.

Olson, M., Bostic, K., & Seltzer, M. (1999). Berkeley DB *Proccedings of the Usenix Annual Technical Conference*. Monterey, CA.

Oracle. (2010). *Java Programming Language, SL-275-SE6 G.2.*

Patyk-Lonska, A., Czachor, M., & Aerts, D. (2011). A comparison of geometric analogues of holographic reduced representations, original holographic reduced representations and binary spatter codes *Procedings of the Federated Conference on Computer Science and Information Systems (FedCSIS), 2011*

Peterson, L. R., & Peterson, M. J. (1959). Short-term retention of individual verbal items. *Journal of Experimental Psychology, 58*, 193-198.

Plate, T. A. (1995). Holographic Reduced Representations. *IEEE Transactions on Neural Networks, 6*(3), 623-641.

Plate, T. A. (2003). *Holographic Reduced Representation: distributed representation of cognitive structure*. Stanford: CSLI.

Polikar, R., Udpa, L., Udpa, S., & Honavar, V. (2001). Learn++: An incremental learning algorithm for supervised neural networks. *IEEE Transactions on System, Man and Cybernetics (C), Special Issue on Knowledge Management, 31*(4), 497-508.

Pollack, J. B. (1990). Recursive Distributed Representations. *Artificial Intelligence, 46*(1-2), 77-105.

Prager, R. W., & Fallside, F. (1989). The modified Kanerva model for automatic speech recognition. *Computer Speech and Language, 3*(1), 61-81.

Rachkovskij, D. A. (2001). representation and Processing of Structures with Binary Sparse Distributed Codes. *IEEE Transactions on Knowledge and Data Engineering, 13*(2), 261-276.

Rachkovskij, D. A., & Kussul, E. M. (2001). Binding and Normalization of Binary Sparse Distributed Representations by Context-Dependent Thinning. *Neural Computation, 13*(2), 411-452.

Ramamurthy, U., Baars, B. J., D'Mello, S. K., & Franklin, S. (2006). *LIDA: A Working Model of Cognition.* Paper presented at the 7th International Conference on Cognitive Modeling, Trieste, Italy.

Ramamurthy, U., D'Mello, S. K., & Franklin, S. (2006). Realizing Forgetting in a Modified Sparse Distributed Memory System. In C. Schunn & S. Lane (Eds.),

Proceedings of the 28th Annual Conference of the Cognitive Science Society (pp. 1992–1997). Mahwah, NJ: Lawrence Erlbaum Associates.

Ramamurthy, U., D'Mello, S. K., & Franklin, S. (2004). Modified Sparse Distributed Memory as Transient Episodic Memory for Cognitive Software Agents *Proceedings of the International Conference on Systems, Man and Cybernetics*. Piscataway, NJ: IEEE.

Ramamurthy, U., & Franklin, S. (2011). *Memory Systems for Cognitive Agents*. Paper presented at the Proceedings of Human Memory for Artificial Agents Symposium at the Artificial Intelligence and Simulation of Behavior Convention (AISB'11).

Rao, R. P. N., & Fuentes, O. (1998). Hierarchical Learning of Navigational Behaviors in an Autonomous Robot using a Predictive Sparse Distributed Memory. *Machine Learning, 31*, 87-113.

Ratitch, B., & Precup, D. (2004). Sparse distributed memories for on-line value-based reinforcement learning. *Lecture Notes in Computer Science (LNCS), 3201*, 347-358.

Recchia, G. L., Jones, M. N., Sahlgren, M., & Kanerva, P. (2010). Encoding sequential information in vector space models of semantics: Comparing holographic reduced representation and random permutation. In S. Ohisson & R. Catrambone (Eds.), *Proceedings of the 32nd Annual Cognitive Science Society*. Austin, TX.

Riesenhuber, M., & Poggio, T. (1999). Hierarchical models of object recognition in cortex. *Nature Neuroscience, 2*, 1019-1025.

Robertson, P., & Laddaga, R. (2011). *Learning to Find Structure in a Complex World.* Paper presented at the Biological Inspired Cognitive Architectures 2011, Washington DC.

Rogers, D. (1990). Predicting weather using a genetic memory: a combination of Kanerva's sparse distributed memory with Holland's genetic algorithms. *Advances in Neural Information Processing Systems, 2*, 455-464.

Saarinen, J., Pohja, S., & Kaski, K. (1991). Self-organization with Kanerva's sparse distributed memory. In T. Kohonen, K. Mäkisara, O. Simula & J. Kangas (Eds.), *Artificial Neural Networks ICANN-91* (Vol. 1, pp. 285-290). Helsinki, Finland: Elsevier/North-Holland.

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

Sejnowski, T. J., & Rosenberg, C. R. (1986). *NETtalk: A Parallel Network that Learns to Read Aloud* (No. Report JHU/EECS-86/01): Department of Electrical Engineering and Computer Science, Johns Hopkins University.

Serre, T., Wolf, L., Bileschi, S., Riesenhuber, M., & Poggio, T. (2007). Robust Object Recognition with Cortex-Like Mechanisms. *IEEE Transations on Pattern Analysis and Machine Intelligence, 29*(3), 411-426.

Shannon, C., & Weaver, W. (1949). *The mathematical theory of communication*. Urbana, IL: University of Illinois Press.

Sims, C. R., & Gray, W. D. (2004). *Episodic versus semantic memory: An exploration of models of memory decay in the serial attention paradigm.* Paper presented at the 6th international conference on cognitive modeling -- ICCM2004, Pittsburgh, PA.

Smolensky, P. (1990). Tensor Product Variable Binding and the Representation of Symbolic Structures in Connectionist Systems. *Artificial Intelligence, 46*(1-2), 159-216.

Snaider, J., & Franklin, S. (2011). *Extended Sparse Distributed Memory.* Paper presented at the Biological Inspired Cognitive Architectures 2011, Washington DC.

Snaider, J., & Franklin, S. (2012a). Extended Sparse Distributed Memory and Sequence Storage. *Cognitive Computation, 4*(2), 172-180.

Snaider, J., & Franklin, S. (2012b). *Integer Sparse Distributed Memory.* Paper presented at the The 25th Florida Artificial Intelligence Research Society Conference FLAIRS-25, Marco Island, FL.

Snaider, J., & Franklin, S. (in review). Modular Composite Representation.

Snaider, J., Franklin, S., Strain, S., & George, E. O. (in review). Integer Sparse Distributed Memory: Analysis and Results.

Snaider, J., McCall, R., & Franklin, S. (2010). *The Immediate Present Train Model Time Production and Representation for Cognitive Agents*. Paper presented at the AAAI Spring Symposium 2010 on "It's All In the Timing".

Snaider, J., McCall, R., & Franklin, S. (2011). *The LIDA Framework as a General Tool for AGI*. Paper presented at the The Fourth Conference on Artificial General Intelligence.

Snaider, J., McCall, R., & Franklin, S. (2012). Time Production and Representation in a Conceptual and Computational Cognitive Model. *Cognitive Systems Research, 13*(1), 59-71.

Somervuo, P. (1999). Redundant Hash Addressing of Feature Sequences Using the Self-Organizing Map. *Neural Processing Letters, 10*(1), 25-34.

Starzyk, J. A., & He, H. (2007). Anticipation-Based Temporal Sequences Learning in Hierarchical Structure. *IEEE Transactions on Neural Networks, 18*(2), 344-358.

Stringer, S. M., Rolls, E. T., Trappenberg, T. P., & de Araujo, I. E. T. (2003). Selforganizing continuous attractor networks and motor function. *Neural Networks, 16* (2), 161-182.

Subramaniam, V. (2011). *Programming Concurrency on the JVM: Mastering Synchronization, STM, and Actors* Pragmatic Bookshelf.

Sun, R., & Giles, C. L. (2001). Sequence learning: From recognition and prediction to sequential decision making. *IEEE Intelligent Systems, 16*(4), 67-70.

Sutton, R. S., & Whitehead, S. D. (1993). Online learning with random representations *Proceedings of the 10th International Conference on Machine Learning* (pp. 314-321): Morgan Kaufmann.

Tenenbaum, J. B., Kemp, C., Griffiths, T. L., & Goodman, N. D. (2011). How to grow a mind: statistics, structure, and abstraction. *Science, 331*(6022), 1279-1285.

Tulving, E. (1968). Theoretical issues in free recall. In T. R. Dixon & D. L. Horton (Eds.), *Verbal Behaviour and General Behaviour Theory*. Englewood Cliffs, NJ: Prentice Hall.

Turney, P. D., & Pantel, P. (2010). From Frequency to Meaning: Vector Space Models of Semantics. *Journal of Artificial Intelligence Research, 37*, 141-188.

Versace, M., & Chandler, B. (2011). MoNETA: A Mind Made from Memristors. *IEEE Spectrum, December 2011*.

Wang, D., & Yuwono, B. (1995). Anticipation-based temporal pattern generation. *IEEE Trans. Syst., Man, Cybern, 25*(4), 615-628.

Wang, D., & Yuwono, B. (1996). Incremental Learning of Complex Temporal Patterns. *IEEE Transactions on Neural Networks, 7*(6), 1465-1481.

Wang, L. (1998). Learning and retrieving spatio-temporal sequences with any static associative neural network. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, 45*(6), 729-738.

Wang, L. (2000). Heteroassociations of spatio-temporal sequences with the bidirectional associative memory. *IEEE Transactions on Neural Networks, 11*(6), 1503-1505.

Willshaw, D. J. (1981). Holography, associative memory, and inductive generalization. In G. E. Hinton & J. A. Anderson (Eds.), *Parallel Models of Associative Memory* (pp. 83-104). Hillsdale, NJ: Erlbaum.

Willshaw, D. J., Buneman, O. P., & Longuet-Higgins, H. C. (1969). Non-holographic associative memory. *Nature, 222*(5197), 960-962.

Winston, P. H. (1992). *Artificial Intelligence* (3rd ed.). Boston, MA: Addison Wesley.

Wörgötter, F., & Porr, B. (2004). Temporal sequence learning, prediction, and control: A review of different models and their relation to biological mechanisms. *Neural Computation, 17*(2), 245-319.

Appendix A: Author's Refereed Publications

Snaider, J., & Franklin, S. (2012a). Extended Sparse Distributed Memory and Sequence Storage. *Cognitive Computation, 4*(2), 172-180.

Franklin, S., Strain, S., Snaider, J., McCall, R., & Faghihi, U. (in press). Global Workspace Theory, its LIDA Model and the Underlying Neuroscience. *Biologically Inspired Cognitive Architectures, 1*.

Snaider, J., McCall, R., & Franklin, S. (2012). Time Production and Representation in a Conceptual and Computational Cognitive Model. *Cognitive Systems Research, 13*(1), 59-71.

Snaider, J., & Franklin, S. (2012). *Integer Sparse Distributed Memory.* Paper presented at the The 25th Florida Artificial Intelligence Research Society Conference FLAIRS-25, Marco Island, FL.

Snaider, J., & Franklin, S. (2011). *Extended Sparse Distributed Memory.* Paper presented at the Biological Inspired Cognitive Architectures 2011, Washington DC.

Snaider, J., Olney, A. M., & Person, N. (2011). *Nonverbal Action Selection for Explanations Using an Enhanced Behavior Net*. Paper presented at the the 11th Conference of Intelligent Virtual Agents 2011.

Snaider, J., McCall, R., & Franklin, S. (2011). *The LIDA Framework as a General Tool for AGI*. Paper presented at the The Fourth Conference on Artificial General Intelligence.

Snaider, J., McCall, R., & Franklin, S. (2010). *The Immediate Present Train Model Time Production and Representation for Cognitive Agents*. Paper presented at the AAAI Spring Symposium 2010 on "It's All In the Timing".

Snaider, J., McCall, R., & Franklin, S. (2009). *Time Production and Representation in a Conceptual and Computational Cognitive Model*. Paper presented at the AAAI Fall Symposium 2009 on Biologically Inspired Cognitive Architecture.

Snaider, J., Proaño, A., López De Luise, D., & Stegmayer, G. (2008). *Autenticación Facial Inteligente*. Paper presented at the X Workshop de Investigadores en Ciencias de la Computación. La Pampa, Argentina.

Franklin, S., Madl, T., D'Mello, S. K., & Snaider, J. (in review). LIDA: A Systems-level Architecture for Cognition, Emotion, and Learning. *Transactions on Autonomous Mental Development*.

Snaider, J., Franklin, S., Strain, S., & George, E. O. (in review). Integer Sparse Distributed Memory: Analysis and Results.

Snaider, J., & Franklin, S. (in review). Modular Composite Representation.

Appendix B: MCR Scripting Language Javacc Grammar

```
/**
 * JavaCC template file created by SF JavaCC plugin 1.5.17+ wizard for
 * JavaCC 1.5.0+
 *
 * @author Javier Snaider
 *
 */
options
{
  JDK_VERSION = "1.6";
  static = false;
}

PARSER_BEGIN(McrParser)
package edu.memphis.ccrg.mvsdm.mcr.parser;
import edu.memphis.ccrg.mvsdm.parser.nodes.*;
import java.util.ArrayList;
import java.util.List;

public class McrParser
{
}

PARSER_END(McrParser)

SKIP :
{
  " "
| "\r"
| "\t"
}
/* OPERATORS */
TOKEN :
{
< PLUS : "+" >
}
TOKEN:
{
< INV : "!" >
}
TOKEN:
{
< MULOP : "*"|"/" >
}

TOKEN:
{
< EXP : "^" >
}
TOKEN:
{
< SEP : ";" >
}
```

```
TOKEN:
{
< NEW : "newvector" >
|   < NEWRNDVECTOR : "newrndvector" >
|   < NEWRND : "newrnd" >
|   < PRINT : "print" >
|   < PRINTDISTANCE : "printd" >
|   < PRINTRANK : "rank" >
}
TOKEN:
{
< EQUALS : "=" >
}
TOKEN:
{
< CR : "\n" >
}
TOKEN:
{
< LPAREN : "(" >
}
TOKEN:
{
< RPAREN : ")" >
}

TOKEN:
{
< COMMENT: "//" (~["\n"])* >
}
TOKEN :
{
< CONSTANT : ("-")? < NUMBER >("." < NUMBER >)(["E","e"] ("-")? <
NUMBER >)? >
|   < INTEGER : (("-")? < NUMBER >) >
|   < ID : <LETTER> (<LETTER>|<DIGIT>)*  >
|   < NUMBER : (< DIGIT >)+ >
|   < #DIGIT : [ "0"-"9" ] >
|   < #LETTER: ["_","a"-"z","A"-"Z"] >
|   < STRING_LITERAL: "\"" (~["\"","\\","\n","\r"] | "\\"
(["n","t","b","r","f","\\","\'","\""] | ["0"-"7"] (["0"-"7"])? | ["0"-
"3"] ["0"-"7"] ["0"-"7"]))* "\"" >
}

Program program ():
{
    Statement stmt=null;
    Program prog=new Program();
}
{
    (
        (stmt=statement(){
            if (stmt!=null){
                prog.addStatement(stmt);
            }
         })+
```

```
    )
    <EOF>
    {
    return prog;
    }
}

Statement statement():
{
    VecExpression vexp=null;
    VecExpression vexp2=null;
    List<String> ids = new ArrayList<String>();
    Token token=null;
    Token token2=null;
}
{
(token = < ID > <EQUALS > vexp=vectorExpression() < SEP > )
    {
    return new Assignment(new VectorIdentifier(token.image),vexp);
    }

|LOOKAHEAD(3)
(< PRINT > "(" vexp = vectorExpression() ")" < SEP > )
    {
    return new PrintVector(vexp);
    }

|LOOKAHEAD(3)
(< PRINT > "(" token = < STRING_LITERAL > ")" < SEP > )
    {
    return new
PrintObject(token.image.substring(1,token.image.length()-1));
    }

|  (< PRINTDISTANCE > "(" vexp = vectorExpression() "," vexp2 =
vectorExpression() ")"
< SEP > )
    {
    return new PrintVectorDistance(vexp,vexp2);
    }
|LOOKAHEAD(3)
(< PRINTRANK > "(" vexp = vectorExpression() "," token = < INTEGER >
")" < SEP > )
    {
    return new PrintRank(vexp, new Integer (token.image));
    }

|LOOKAHEAD(3)
(< PRINTRANK > "(" token = < STRING_LITERAL > ","
vexp = vectorExpression() "," token2 = < INTEGER > ")" < SEP > )
    {
    return new PrintRank(token.image.substring(1,token.image.length()
1),vexp, new Integer (token2.image));
    }

|   ( < NEWRND > "(" token=< ID >
```

```
        "," token2=< ID >
        {
      ids.add(token2.image);
      }
        )*
    ")" < SEP > )
    {
        ids.add(0,token.image);
      return new NewRnd(ids);
    }

| < SEP >
| < CR >
{
return null;
}
| < COMMENT >< CR >
{
return null;
}
}

VecExpression vectorExpression():
{
    VecExpression oper1=null;
    VecExpression oper2=null;
    Token token=null;
    char op;
    List<VecExpression> ops = new ArrayList<VecExpression>();
}
{
LOOKAHEAD(2)
  oper1=term() (op=addop() oper2=term()
    {
    ops.add(oper2);
    })*
    {
        if(ops.size()>0){
            ops.add(0,oper1);
            return  new SumOp(ops);
        }else{
            return oper1;
        }
    }

|LOOKAHEAD(3)
< NEW > "(" ")"
{
return new NewVectorFact(false);
}
|LOOKAHEAD(4)
< NEW > "(" oper1= vectorExpression() ")"
{
return new NewVectorFact(oper1);
}
|   < NEW > "(" oper1= vectorExpression() "," token=< INTEGER > ")"
```

```
{
return new NewVectorFact(oper1,new Integer(token.image));
}
|  < NEWRNDVECTOR > "("")"
{
return new NewVectorFact(true);
}

|LOOKAHEAD(2)
    token=< ID >
{
return new VectorIdentifier(token.image);
}
}

char addop():
{
}
{
<PLUS>
{
return '+';
}
}

VecExpression term():
{
    VecExpression oper1=null;
    VecExpression oper2=null;
    Token token;
}
{
oper1=factor() (token=< MULOP > oper2=factor()
    {
    oper1= new MulOp(oper1,oper2,(token.image.charAt(0)=='/'));
    })*
{
return oper1;
}
}

VecExpression factor():
{
    VecExpression oper1=null;
}
{
oper1=icp()
{
return oper1;
}
|< INV > oper1=icp()
{
return new InvOp(oper1);
}
}
```

```
VecExpression icp():
{
    VecExpression oper1=null;
    Token token = null;
}
{
token=< ID >
{
return new VectorIdentifier(token.image);
}
|< LPAREN > oper1=vectorExpression() < RPAREN >
{
return oper1;
}
}
```