Part I

SPARSE DISTRIBUTED MEMORY: A CROSS-PLATFORM, MASSIVELY PARALLEL, OPEN SOURCE REFERENCE IMPLEMENTATION

1 INTRODUCTION

Sparse Distributed Memory (SDM) [10] is a mathematical model of long-term memory that has a number of neuroscientific and psychologically plausible dynamics. It has been applied in many different fields, like pattern recognition [16, 20], noise reduction [14], handwriting recognition [7], robot automation [19, 13], and so forth. Despite all those applications, there is not a reference implementation which would allow one to replicate the results published in a paper, to check the source code for details, and to improve it. Thus, even though intriguing results have been achieved using SDM, it requires great effort of researchers to improve someone's work.

It is our belief that such a tool could bring orders of magnitude more researchers and attention if they were able to use the model, at zero cost, with an easy to use high-level language such as python. Neuroscientists interested in long-term memory storage should not have to worry about high-bandwidth vector parallel computation. This new tool provides a ready to use system in which experiments can be executed almost as soon as they are designed.

Our motivation was our own effort in order to run our models. As there is no reference implementation, we had to implement our own and run several simulations to ensure that our implementation was correct and bug free. Thus, we had to deviate from our main goal — which was to test our models — and to focus in the implementation itself. Furthermore, new members in our research group had to go through different source codes developed by former members.

The main contribution of this work is a reference implementation which yields (i) orders of magnitude gains in performance, (ii) has several backends and operations, (iii) has automatic tests, (iv) is cross-platform, and (v) is easily extended to fulfill other research models. Our reference implementation may, hopefully, accelerate research into the model's dynamics and make it easier for readers to replicate any previous results and easily understand the source-code of the model. Moreover, new machanisms and applications are also introduced.

SDM, extensions of it, has been used in many applications. For example, Snaider and Franklin [21] extended SDM to efficiently store sequences of vectors and trees. Rao and Fuentes [19] used SDM in an autonomous robot. Meng et al. [14] modified SDM to clean patterns from noisy inputs. Linhares et al. [12] showed that SDM respects the limits of short-term memory discussed by Miller [15] and Cowan [6].

2

SPARSE DISTRIBUTED MEMORY

Sparse Distributed Memory (SDM) is a mathematical model for cognitive memory published by Kanerva [10]. It introduces many interesting mathematical properties of n-dimensional binary space that, in a memory model, are psychologically plausible. Most notable among these are the tip-of-the-tongue phenomenon, conformity to Miller's magic number [12] and robustness against loss of neurons.

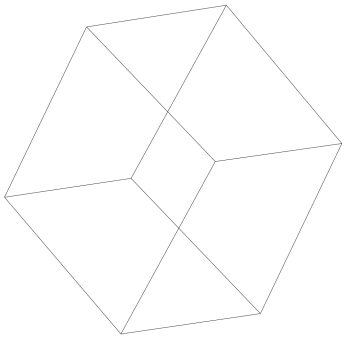
The data and address space belong to binary space and are represented by a sequence of bits, called bitstrings. The distance between two bitstrings is calculated using the Hamming distance. It is defined for two bitstrings of equal length as the number of positions at which the bits are different. For example, 00110_b and 01100_b are bitstrings of length 5 and their Hamming distance is 2. One has to be careful when thinking intuitively about distance in SDM because the Hamming distance does not have the same properties of, say, the Euclidean distance.

The graph composed of $\{0,1\}^n$ nodes and links between nodes iff their Hamming distance is one is called the *hypercube graph*, or Q_n, as in Figure 1. Though Kanerva has derived many combinatorial properties of the space, I believe that this is a aesthetically appealing object on its own, and beautiful results can be found in the graph-theoretical literature. A good survey is found in Harary et al. [9].

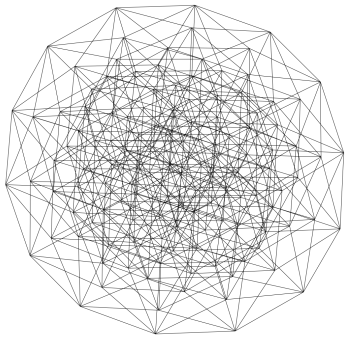
Here is an interesting question that I leave for further research: A hypercube with n dimensions can be divided by two hypercubes with $n-1$ dimensions. Is there an algorithm that separates the area of each hard-location in such a form that there exists a function mapping each bitstring in $\{0,1\}^n$ to the set of hard locations it belongs to? In other words... though this would break Kanerva's assumption of a uniformly distributed set of hard locations for a perfectly symmetrical set of hard locations, there could be large performance gains if such a mapping function from a bitstring to its corresponding set of nearest hard locations exists.

Unlike traditional memory used by computers, SDM performs read and write operations in a multitude of addresses, also called neurons. That is, the data is not written, or it is not read in a single address spot, but in many addresses. These are called activated addresses, or activated neurons.

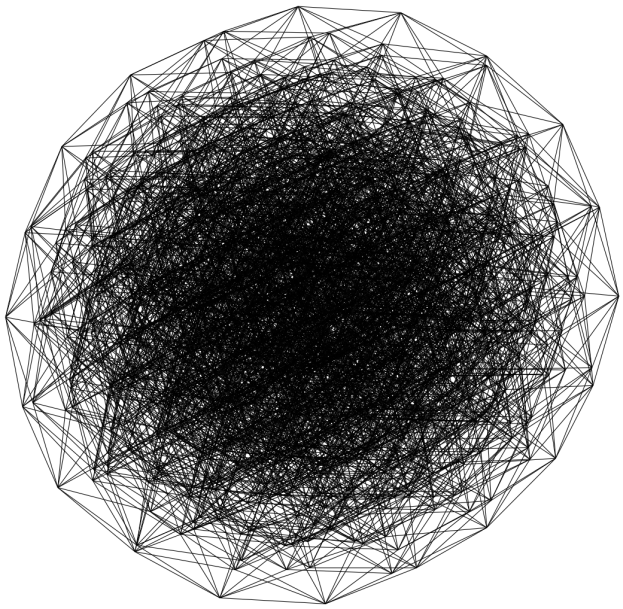
The activation of addresses takes place according to their distances from the datum. Suppose one is writing datum η at address ξ, then all addresses inside a circle with center ξ and radius r are activated.



(a) Q_3



(b) Q_7



(c) Q_{10}

Figure 1: Here we have Q_n, for $n \in \{3, 7, 10\}$. Each node represents a possible bitstring in $\{0, 1\}^n$, and two nodes are linked if the bitstrings differ by a single dimension. A number of observations can be made here. First, the number of nodes grows as 2^n as n grows; which makes use of the entire space infeasible as $n >> 20$. Another interesting observation, better seen in the figures below, is that most of the space lies at the center, at a distance of around 500 from any given vantage point.

So, η will be stored in all these activated addresses, which are around address ξ, such as in Figure 2. An address ξ' is inside the circle if its hamming distance to the center ξ is less than or equal to the radius r, i.e. $distance(\xi, \xi') \leqslant r$.

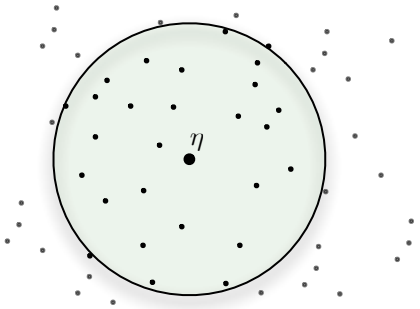


Figure 2: Activated addresses inside access radius r around center address.

Every time write or read in SDM memory activates a number of addresses with close distance. The data is written in these activated addresses or read from them. These issues will be addressed in due detail further on, but a major difference from a traditional computer memory is that the data are always stored and retrieved in a multitude of addresses. This way SDM memory has robustness against loss of addresses (e.g., death of a neuron).

In traditional memory, each datum is stored in an address and every look up of a specific datum requires a search through the memory. In spite of computer scientists having developed beautiful algorithms to perform fast searches, almost all of them do a precise search. That is, if you have an imprecise clue of what you need, these algorithms will simply fail.

In SDM, the data space is the same as the address space, which amounts to a vectorial, binary space, that is, a $\{0,1\}^n$ space. This way, the addresses where the data will be written are the same as the data themselves. For example, the datum $\eta = 00101_b \in \{0,1\}^5$ will be written to the address $\xi = \eta = 00101_b$. If one chooses a radius of 1, the SDM will activate all addresses one bit away or less from the center address. So, the datum 00101_b will be written to the addresses 00101_b, 10101_b, 01101_b, 00001_b, 00111_b, and 00100_b.

In this case, when one needs to retrieve the data, one could have an imprecise cue at most one bit away from η, since all addresses one bit away have η stored in themselves. Extending this train of thought for larger dimensions and radius, exponential numbers of addresses are activated and one can see why SDM is a distributed memory.

When reading a cue η_x that is x bits away of η, the cue shares many addresses with η. The number of shared addresses decreases as the cue's distance to η increases, in other words, as x increases.

This is shown in Figure 3. The target datum η was written in all shared addresses, thus they will bias the read output in the direction of η. If the cue is sufficiently near the target datum η, the read output will be closer to η than η_x was. Repeating the read operation increasingly gets results closer to η, until it is exactly the same. So, it may be necessary to perform more than one read operation in order to converge to the target data η.

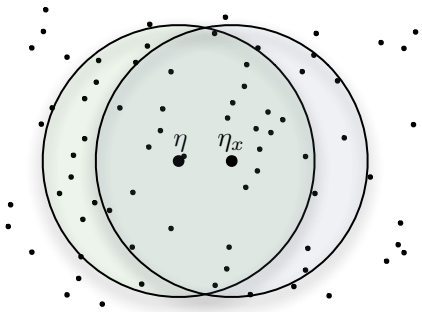


Figure 3: Shared addresses between the target datum η and the cue η_x.

The addresses of the $\{0,1\}^n$ space grows exponentially with the number of dimensions n, i.e. $N = 2^n$. For $n = 100$ we have $N \approx 10^{30}$, which is incredibly large when related to a computer memory. Furthermore, Kanerva [10] suggests n between 100 and 10,000. Recently he has postulated 10,000 as a desirable minimum N (personal communication). To solve the feasibility problem of implementing this memory, Kanerva made a random sample of $\{0,1\}^n$, in his work, having N' elements. All these addresses in the sample are called hard-locations. Other elements of $\{0,1\}^n$, not in N', are called virtual neurons. This is represented in Figure 4. All properties of read and write operations presented before remain valid, but limited to hard-locations. Kanerva suggests taking a sample of about one million hard-locations.

Using this sample of binary space, our data space does not exist completely. That is, the binary space has 2^n addresses, but the memory is far away from having these addresses available. In fact, only a fraction of this vectorial space is actually instantiated. Following Kanerva's suggestion of one million hard-locations, for $n = 100$, only $100 \cdot 10^6/2^{100} = 7 \cdot 10^{-23}$ percent of the whole space exists, and for $n = 1,000$ only $100 \cdot 10^6/2^{1000} = 7 \cdot 10^{-294}$ percent.

Kanerva also suggests the selection of a radius that will activate, on average, one one thousandth of the sample, which is 1,000 hard-locations for a sample of one million addresses. In order to achieve his suggestion, a 1,000-dimension memory uses an access radius $r = 451$, and a 256-dimensional memory, $r = 103$. We think

that a 256-dimensional memory may be important because it presents conformity to Miller's magic number [12].

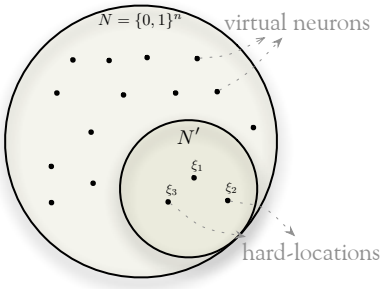


Figure 4: Hard-locations randomly sampled from binary space.

Since a cue η_x near the target bitstring η shares many hard-locations with η, SDM can retrieve data from imprecise cues. Despite this feature, it is very important to know how imprecise this cue could be while still giving accurate results. What is the maximum distance from our cue to the original data that still retrieves the right answer? An interesting approach is to perform a read operation with a cue η_x, that is x bits away from the target η. Then measure the distance from the read output and η. If this distance is smaller than x we are converging. Convergence is simple to handle, just read again and again, until it converges to the target η. If this distance is greater than x we are diverging. Finally, if this distance equals x we are in a tip-of-the-tongue process. A tip-of-the-tongue psychologically happens when you know that you know, but you can't say what exactly it is. In SDM mathematical model, a tip-of-the-tongue process takes infinite time to converge. Kanerva [10] called this x distance, where the read's output averages x, the critical distance. Intuitively, it is the distance from which smaller distances converge and greater distances diverge. In Figure 5, the circle has radius equal to the critical distance and every η_x inside the circle should converge. The figure also shows a convergence in four readings.

The $\{0,1\}^n$ space has $N = 2^n$ locations from which we instantiate N' samples. Each location in our sample is called a hard-location. On these hard-locations we do operations of read and write. One of the insights of SDM is exactly the way we read and write: using data as addresses in a distributed fashion. Each datum η is written in every activated hard-location inside the access radius centered on the address, that equals datum, $\xi = \eta$. Kanerva suggested using an access radius r having about one one thousandth of N'. As an imprecise cue η_x shares hard-locations with the target bitstring η, it is possible to retrieve η correctly. (Actually, probably more than one read is necessary to retrieve exactly η.). Moreover, if some neurons

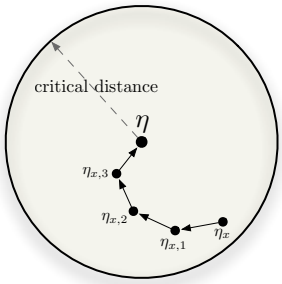


Figure 5: In this example, four iterative readings were required to converge from η_x to η.

η	0	1	1	0	1	0	0
ξ_{before}	6	-3	12	-1	0	2	4
	⇓ -1	⇓ +1	⇓ +1	⇓ -1	⇓ +1	⇓ -1	⇓ -1
ξ_{after}	**5**	**-2**	**13**	**-2**	**1**	**1**	**3**

Table 1: Write operation example in a 7-dimensional memory of data η being written to ξ, one of the activated addresses.

are lost, only a fraction of the datum is lost and it is possible that the memory can still retrieve the right datum.

A random bitstring is generated with equal probability of 0's and 1's in each bit. One can readily see that the average distance between two random bitstrings has binomial distribution with mean $n/2$ and standard deviation $\sqrt{n/4}$. For a large n, most of the space lies close to the mean and has fewer shared hard-locations. As two bitstrings with distance far from $n/2$ are very improbable, Kanerva [10] defined that two bitstrings are orthogonal when their distance is $n/2$.

The write operation needs to store, for each dimension bit which happened more (0's or 1's). This way, each hard-location has n counters, one for each dimension. The counter is incremented for each bit 1 and decremented for each bit 0. Thus, if the counter is positive, there have been more 1's than 0's, if the counter is negative, there have been more 0's than 1's, and if the counter is zero, there have been an equal number of 1's and 0's. Table 1 shows an example of a write operation being performed in a 7-dimensional memory.

The read is performed polling each activated hard-location and statistically choosing the most written bit for each dimension. It consists of adding all n counters from the activated hard-locations and, for each bit, choosing bit 1 if the counter is positive, choose bit 0 if the counter if negative, and randomly choose bit 0 or 1 if the counter is zero.

2.1 NEURONS AS POINTERS

One interesting view is that neurons in SDM work like pointers. As we write bitstrings in memory, the hard-locations' counters are updated and some bits are flipped. Thus, the activated hard-locations do not necessarily point individually to the bitstring that activated it, but together they point correctly. In other words, the read operation depends on many hard-locations to be successful. This effect is represented in Figure 6: where all hard-locations inside the circle are activated and they, individually, do not point to η. But, like vectors, adding them up points to η. If another datum ν is written into the memory near η, the shared hard-locations will have information from both of them and would not point to either. All hard-locations outside of the circle are also pointing somewhere (possibly other data points). This is not shown, however, in order to keep the picture clean and easily understandable.

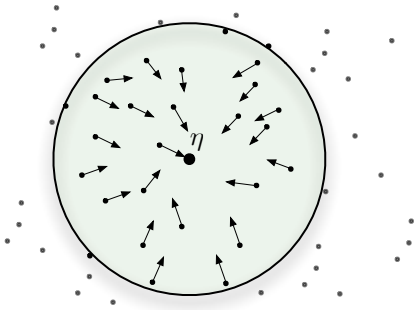


Figure 6: Hard-locations pointing, approximately, to the target bitstring.

2.2 CONCEPTS

Although Kanerva does not mention concepts directly in his book [10], the author's interpretation is that each bitstring may be mapped to a concept. Thus, unrelated concepts are orthogonal and concepts could be linked through a bitstring near both of them. For example, "beauty" and "woman" have distance $n/2$, but a bitstring that means "beautiful woman" could have distance $n/4$ to both of them. As a bitstring with distance $n/4$ is very improbable, it is linking those concepts together. Linhares et al. [12] approached this concept via "chunking through averaging".

Due to the distribution of hard-locations between two random bitstrings, the vast majority of concepts is orthogonal to all others. Consider a non-scientific survey during a cognitive science seminar, where students asked to mention ideas unrelated to the course brought up terms like birthdays, boots, dinosaurs, fever, executive order, x-rays, and so on. Not only are the items unrelated to

cognitive science, the topic of the seminar, but they are also unrelated to each other.

For any two memory items, one can readily find a stream of thought relating two such items ("Darwin gave dinosaurs the boot"; "she ran a fever on her birthday"; "isn't it time for the Supreme Court to x-ray that executive order?", ... and so forth). Robert French presents an intriguing example in which one suddenly creates a representation linking the otherwise unrelated concepts of "coffee cups" and "old elephants" [8].

This mapping from concepts to bitstrings brings us two main questions: (i) Suppose we have a bitstring that is linking two major concepts. How do we know which concepts are linked together? (ii) From a concept bitstring how can we list all concepts that are somehow linked to it? This second question is called the problem of spreading activation.

2.3 READ OPERATION

In his work, Kanerva proposed and analyzed a read algorithm called here Kanerva's read. His read takes all activated hard-locations counters and sum them. The resulting bitstring has bit 1 where the result is positive, bit 0 where the result is negative, and a random bit where the result is zero. In a word, each bit is chosen according to all written bitstrings in all hard-locations, being equal to the bit more appeared. Table 2a shows an example of Kanerva's read result bitstring.

Daniel Chada, one member of our research group, proposed another way to read in SDM, in this work called Chada's read. Instead of summing all hard-location counters, each hard-location evaluates its resulting bitstring individually. Then, all resulting bitstrings are summed again, and the same rule as Kanerva applies. Table 2b shows an example of Chada's read result bitstring. The counter's values are normalized to 1, for positive ones, or -1, for negative ones, and the original values are the same as in Table 2a.

The main change between Kanerva's read and Chada's read is that, in the former, a hard-location that has more bitstrings written has a greater weight in the decision of each bit. In the latter, all hard-locations have the same weight, because they can contribute to the sum with only one bitstring.

A member of my Master's committee, professor Paulo Murilo, has proposed a generalized reading operation (personal communication), which covers both Kanerva's and Chada's read — and opens a new venue of potential discoveries. He proposed summing all hard-location counters raised to the power of z while holding the original sign of the counter (positive or negative). Thus, Kanerva's read would be the same as $z = 1$, while Chada's would be

ξ_1	-2	12	4	0	-3
ξ_2	-5	-4	2	8	-2
ξ_3	-1	0	-1	-2	-1
ξ_4	3	2	-1	3	1
$\sum$	**-5**	**10**	**4**	**3**	**-5**
	⇓	⇓	⇓	⇓	⇓
	0	1	1	1	0

(a) Kanerva's read example

ξ_1	-1	1	1	1	-3
ξ_2	-1	-1	1	1	-1
ξ_3	-1	1	-1	-1	-1
ξ_4	1	1	-1	-1	1
$\sum$	**-2**	**1**	**0**	**0**	**-2**
	⇓	⇓	⇓	⇓	⇓
	0	1	1	1	0

(b) Chada's read example

Table 2: Comparison of Kanerva's read and Chada's read. Each ξ_i is an activated hard-location and the values come from their counters. Gray cells' value is obtained randomly with probability 50%.

the same as $z = 0$. Hence, we will here explore how SDM would behave with other values of z, such as 0.5, 2, and 3.

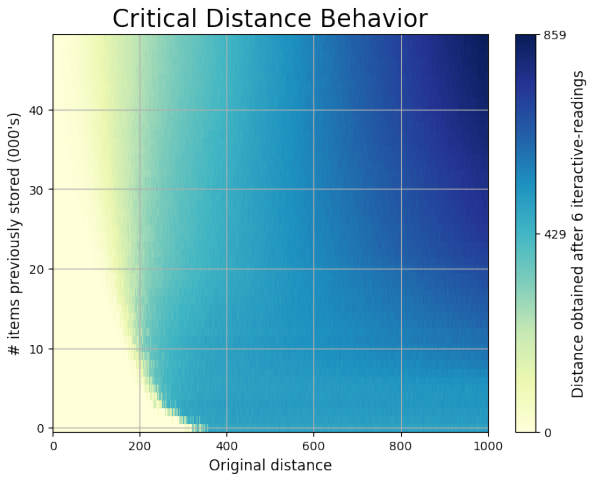
2.4 CRITICAL DISTANCE

Kanerva describes the critical distance as the threshold of convergence of a sequence of read words. It is "the distance beyond which divergence is more likely than convergence"[10]. Furthermore, Kanerva explains that "a very good estimate of the critical distance can be obtained by finding the distance at which the arithmetic mean of the new distance to the target equals the old distance to the target"[10]. In other words, the critical distance can be equated as the edge to our memory, the limit of human recollection.

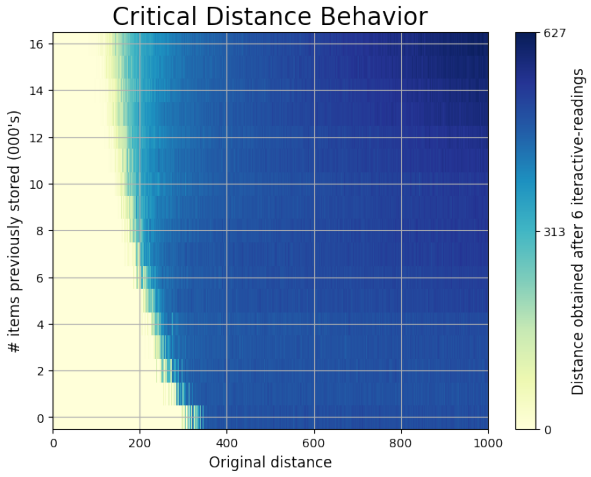
In his book, Kanerva analyzed a specific situation with $n = 1000$ ($N = 2^{1000}$), 10 million hard-locations, an access-radius of 451 (within 1000 hard-locations in each circle) and 10 thousand writes of random bitstrings in the memory. As computer resources were very poor those days, Kanerva couldn't make a more generic analysis.

Starting from the premise of SDM as a faithful model of human short-term memory, a better understanding of the critical distance may shed light on our understanding of the thresholds that bind our own memory.

Figure 7 compares the critical distance behavior under different scenarios. This replicates our previous results Brogliato [1] and Brogliato et al. [2] and is a first part of the process of framework validation, to which we throw our attention next.



(a) Kanerva's original model



(b) Chada's read

Figure 7: How far, in hamming distance, is a read item from the original stored item? Kanerva demonstrated that, after a small number of iterative readings (6 here), a critical distance behavior emerges. Items read at close distance converge rapidly; whereas farther items do not converge. Most striking is the point in which the system displays the tip-of-tongue behavior. Described by psychological moments when some features of the item are prominent in one's thoughts, yet the item still cannot be recalled (but an additional cue makes convergence 'immediate'). Mathematically, this is the precise distance in which, despite having a relatively high number of cues (correct bits) about the desired item, the time to convergence is infinite. Heatmap colors display the hamming distance the associative memory is able to cleanly converge to—or not. In the x-axis, the distance from the desired item is displayed. In the y-axis, we display the read operation's behavior as the number of items registered in the memory grows. These graphs are computing intensive, yet they can be easily tested by readers in our provided jupyter notebooks. Note the different scales.

3

FRAMEWORK ARCHITECTURE

The framework implements the basic operations in a Sparse Distributed Memory which may be used to create more complex operations. It is developed in C language and the OpenCL parallel framework — which may be loaded in many platforms and programming languages — with a wrapper in Python. The Python module makes it easy to create and execute simulations in a Sparse Distributed Memory and works properly in Jupyter Notebook [11]. It works in both Python 2 and Python 3.

We split the SDM memory in two parts: the hard-location addresses and the hard-location counters. Thus, the addresses (bitstrings) of the hard-locations are stored in one array, while their counters in another. This makes possible to create multiple SDMs using the same address space, which would save computational effort to scan a bitstring in all the SDMs — since they share the same address space, the activated hard-locations will be same in all of them. As the slowest part of reading and writing operations is scanning the address space, the performance benefits are high.

Each part may be stored either in the RAM memory or in a file. The RAM memory is interesting for quick experiments, automated tests, and others scenarios in which the SDM may be lost, while the file is interesting for a long-term SDM, like creating an SDM file with 10,000 random writes, which will be copied over and over to run multiple experiments. The file may also be sent to another researcher or may be published with the paper to let others run their own checks and verify the results. In summary, the framework fits many different uses and necessities.

Let a SDM memory with N dimensions and H hard-locations. Then, in a 64-bit computer, the array of hard-location addresses will use $H \cdot 8 \cdot \lceil N/64 \rceil$ bytes of memory, and there will be $H \cdot N$ hard-location counters. For example, in a SDM memory with 1,000 dimensions and 1,000,000 hard-locations, using 32-bit integers for the counters, the array of addresses will use 122MB of memory and the counters will use 3.8 GB of memory.

Basic operations were grouped in four sets: (i) for bitstrings, (ii) for addresses, (iii) for counters, and (iv) for memories (SDMs). Operations include creating new bitstrings, flipping bits, generating a bitstring with a specific distance from a given bitstring, scanning the address space using different algorithms, writing a bitstring to a counter, writing in an SDM, reading from an SDM, and iteratively reading from an SDM until convergence.

3.1 BITSTRING

Bitstrings are the main structure of SDM. The addresses are represented in bitstrings, as well as the data. A bitstring is stored as an array of integers. Each integer may be 16-bit, 32-bit, or 64-bit long, depending on the configuration. By default, each integer is 64-bit long.

For instance, a 1,000-bit bitstring will have $\lceil 1000/64 \rceil = 16$ integers. These integers will have a total of $16 \cdot 64 = 1,024$ bits. The remaining 24 bits are always zero, so they do not affect the result of any operation. Bitstrings store neither how many bits they have nor the array length. These pieces of information are only stored in the address space.

3.1.1 *The distance between two bitstrings*

The distance between two bitstrings is calculated by the hamming distance, which is the number of different bits between them.

It is calculated counting the number of ones in the exclusive or (xor) between the bitstrings, i.e., $d(x, y) =$ number of ones in $x \oplus y$. There are several algorithms to calculate the number of ones [22], but the performance depends on the processor. So, we have implemented three different algorithms and one may be selected through compiling flags. The default algorithm is to use a builtin instruction from the compiler.

An alternative algorithm available is the lookup. It pre-calculates a table of distances — by default, it is a 16-bit table which uses 65MB of RAM, but one may increase to a 32-bit table which uses 4GB of RAM —, which is accessed a few times. Finally, to calculate the distance between two bitstrings, it sums the distance of each 16-bit part of the bitstrings, i.e., $d(x[0 : N-1], y[0 : N-1]) = d(x[0 : 15], y[0 : 15]) + d(x[16 : 31], y[16 : 31]) + \ldots$. As each distance is calculated in $O(1)$, this algorithm runs in $O(\lceil bits/16 \rceil)$.

3.2 ADDRESS SPACE

An address space is a fixed collection of bitstrings, and each bitstring represents a hard-location address. They store the number of bitstrings, as well as the number of bits, number of integers per bitstring, and the number of remaining bits.

Bitstrings are stored in a contiguous array of 64-bit integers, as shown in Figure 8. Hence, basic pointer arithmetic provides us with performance improvements in their access, as processors realize fechts of contiguous chunks of memory [18].

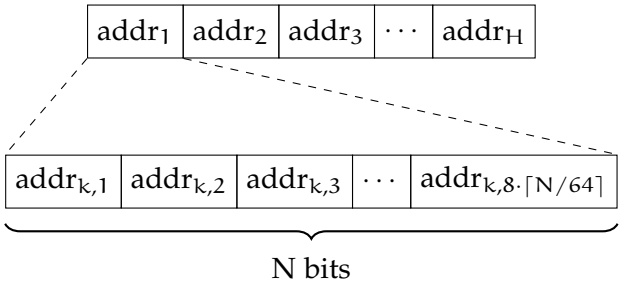


Figure 8: Address space's bitstrings are stored in a contiguous array. In a 64-bit computer, each bitstring is stored in a sub-array of 64-bit integers, with length $8 \cdot \lceil N/64 \rceil$.

The scan for activated hard-locations is performed in an address space. It returns the indexes of the bitstrings which were inside the circle. Then, each operation uses this index in a different way.

3.2.1 *Scanning for activated hard-locations*

Scanning for the activated hard-locations is a problem similar to well-known problems in computational geometry called "range reporting in higher dimensions". In this case, none of the known algorithms is able to solve our problem faster than $O(n)$. The algorithm which seems to best fit in our problem consumes $O(n)$ space and runs in $O(\log^d(n))$ [4], which is really slower than $O(n)$ when, for instance, $n = 1,000,000$ and $d = 1,000$. For a review of the algorithms, see Chan et al. [3].

In 2014, there was published a solution to fast search in hamming space which seems applicable to our problem Norouzi et al. [17]. It provides a fast search when $r/b < 0.11$ or $r/b < 0.06$, where r is the radius and b is the number of bits. But, in our case, for a 1,000 bits SDM, $r/b = 0.451$, which changes the runtime to $O(n^{0.993})$. This is really close to $O(n)$, but with a larger constant. Hence, $O(n)$ is still faster.

It is intriguing that none of those algorithms is able to solve our scanning problem. The idea behind those computational geometry algorithms is roughly to split the search space in half each step, which would take $O(\log(n))$ to go through the whole space. But this approach does not work because of the high number of dimensions (i.e., 1,000) and because the hardlocations' addresses are randomly sampled from the $\{0,1\}^d$ space. So, each addresses' bit itself splits the hardlocations in half, but it does not split the search space in half since both halves still must be convered by the algorithm. For instance, let's say we have 1,000 dimensions with 1,000,000 hardlocations, and we are scanning within a circle with radius 451, then after checking the first bit we have two cases: (i) for the half with the same first bit, we must keep scanning with radius 451; and

(ii) for the half with a different first bit, we must keep scanning with radius 450. The search space has not been split in half because both halves have been covered (and one of them should have been skipped).

Hence, as our best approach is to scan through all hardlocations, we may distribute the scan into many tasks which will be executed independently. The tasks may be executed in different processes, threads, or even computers. They may also run in the CPU or in the GPU. In this case, we may take into account both the time required to distribute the tasks and the time to receive their results.

The framework implements three main scanner algorithms: linear scanner, thread scanner, and OpenCL scanner. The linear scanner runs in a single core, is the slowest one, and was developed only for testing purposes; the thread scanner runs at the CPU in multiple threads sharing memory (and our recommendation is to use the number of threads equals to twice the number of CPU cores); and the OpenCL scanner runs in multiple GPU cores and support multiple devices. The speed of a scan depends on the CPU and GPU devices, thus the best approach to choose which scanner is best for one's setup is to run a benchmark.

The OpenCL must be initialized, which just copies the address space's bitstrings to the GPU's memory. Then, many scans may be executed with no necessity to upload the bitstrings again.

3.3 COUNTERS

Each hard-location has one integer of data per bit. For instance, each hard-location of a 1,000 bits SDM has 1,000 bits. Those integers are stores in a counter.

A counter is an array of integers which stores the data of all hard-locations. So, the counter's array has $N \cdot H$ integers.

When two counters are added in a third counter, there may occur an overflow. Counters may have overflow protection depending on compiling options. By default, there is no overflow check for performance reasons. It is not supposed to be a problem because each counter is a signed 32-bit integer which can store any number between -2,147,483,648 and 2,147,483,647 (and they will not overflow with less than $2^{31} - 1$ divided by the number of activated hard-locations). For instance, when $N = 1,000$, $H = 1,000,000$, and $r = 451$, the average number of activated hard-locations is 1,000 and it would require at least one million writes to be possible to overflow a counter. Note also that it would be improbable for a hard location dimension to be this biased before the complete saturation of the memory.

3.4 READ AND WRITE OPERATIONS

The reading and writing operations are executed in two steps: first, the address space is swept looking for the activated addresses; then, the operation is performed in the counters. Reading operation assemblies the bitstring according to the counters of the activated addresses, while the writing operation changes the counters.

The iterated reading keeps reading until it gets exactly the same bitstring (or the number of maximum interations has been reached), i.e., it performs $\eta_{i+1} = \text{read}(\eta_i)$ and stops when $\eta_{k+1} = \eta_k$. If the initial bitstring is inside the critical distance of η, it will converge to η, but, if it is not, it will diverge and reach the maximum number of iterations.

The framework is available both Kanerva's read and Chada's read. The latter was implemented according to a generalization proposed by physics professor Paulo Murilo of UFF during the discussion of my master's defense. The generalization brings a parameter z, which is the exponent of the counter. In this case, the results are floating point instead of integer, which considerably reduces performance. When $z = 1$, it is exactly as the Kanerva's read. When $z = 0$, it is the Chada's read. We also explored how SDM would behave for different values of z.

There is also another special read operation: the weighted reading. The weight is retrieved from a lookup table of integers based on the distance between the hard-location's address and the reading address. The weight is multiplied by the counter's value and summed. The rest of the read operation is exactly the same.

4

RESULTS (I): FRAMEWORK VALIDATION

The framework has been validated comparing its results with the expected results from Kanerva [10]. Thus, we run simulations which were then compared to the theoretical analysis conducted some decades ago.

The objective here is twofold: (i) a single command will install the framework, and (ii) another single command will run (and display) the desired figures obtained the simulation. This will allow potential users to become familiarized with the system and its underlying code with barely any learning curve beyond scientific python and jupyter notebooks.

One particular analysis of interest is that of the distance read at a point α. Suppose an SDM is trying to read an item written at α, but the cues received so far lead to a point of distance d from α. As one reads at $\alpha + d$, a new bitstring β is obtained, leading to our question: what is the new distance from α to β? Is it smaller or larger than d? That, of course, depends on the ratio between d and the number of dimensions of the memory. As we have found out, there are some deviations from Kanerva's original theoretical analysis and the results obtained by simulation.

4.0.1 *Some initial anomalous results*

As we ran the simulations reflecting some of Kanerva's graphs, one in particular struck our attention: The new distances obtained after a read operation were not perfectly predicted by the theoretical model, and we propose that this is due to interaction effects between different attractors.

Kanerva [10] originally predicted a ~500-bit distance after a point (Fig. 9). The original prediction considered that the read distance would decline when inside the critical distance in increase afterwards, converging to a ~500-bit distance. At this point, each read would lead to a different, orthogonal, ~500-bit distance point.

Our preliminary results show that the theoretical prediction is not accurate. There are interaction effects from one or more of the attractors created by the 10,000 writes, and these attractors seem to raise the distance beyond ~500 bits (Fig. 10). Our results were obtained using a 1,000 bit, 1,000,000 hard-location SDM with exactly 10,000 random bitstrings written into it, the same used by Kanerva.

But, when we reduced the number of random bitstrings written in the SDM from 10,000 to only 100, the results reflected very well the

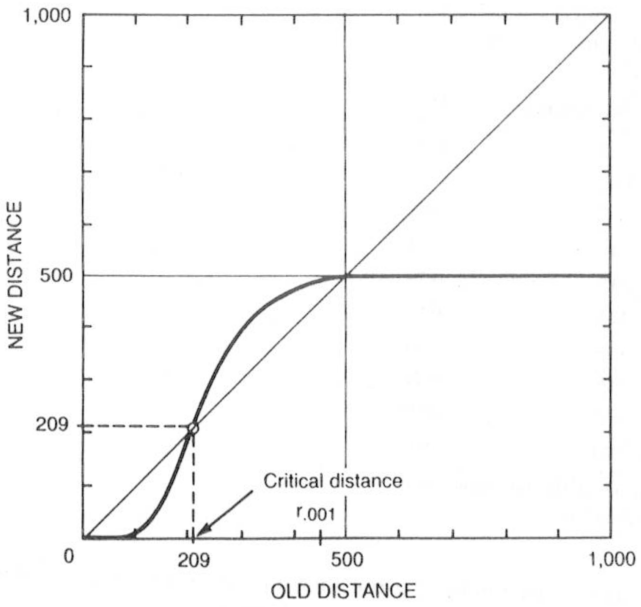


Figure 7.3
New distance to target as a function of old distance.

Figure 9: Kanerva's original Figure 7.3 (p. 70) predicting a ~500-bit distance after a point.

Kanerva's theoretical expectation (Fig. 11). This result strengthens our hypothesis that the disparities in the computational results are due to the interaction effect of high numbers of different attractors.

To obtain the results from Fig. 10 and 11, we had to write 10,000 random bitstrings to an SDM, and then randomly choose one of those bitstrings to be our origin. Finally, we randomly flipped some bits from the origin bitstring and executed a reading operation in the SDM. Thereby, in order to show the interaction effects more clearly, we wrote a handmade bitstring to the SDM which had all bits inverted in relation to the origin bitstring — their hamming distance was equal to 1,000. Our handmade bitstring was acting as an opposite attractor, and one can see the accelerating effects towards convergence to both attractors: the origin and the handmade bitstrings (Fig. 12). Here we had the exact same configuration of Figure 10, with the addition of the single opposite attractor.

Obviously, these small deviations from Kanerva's original theoretical predictions deserve a qualification. Kanerva was working in the 1980s and the 1990s, and had no access to the immense computational power that we do today. It is no surprise that some small interaction effects should exist as machines allow us to explore the ideas of his monumental work.

Physicist Paulo Murilo observed that the models of Kanerva-read ($z = 1$) and Chada-read ($z = 0$) were simple variations of the exponent

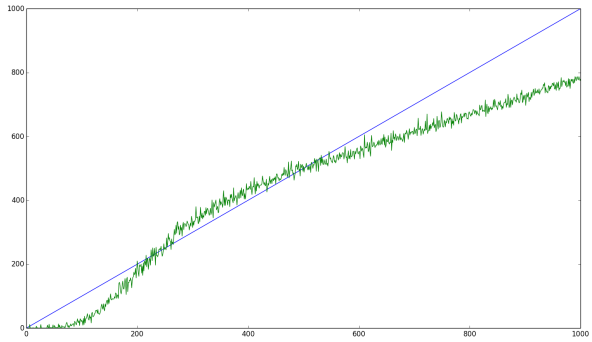


Figure 10: Results generated by the framework diverging from Kanerva's original Table 7.2. Here we had a 1,000 bit, 1,000,000 hard-location SDM with exactly 10,000 random bitstrings written into it, which was also Kanerva's configuration.

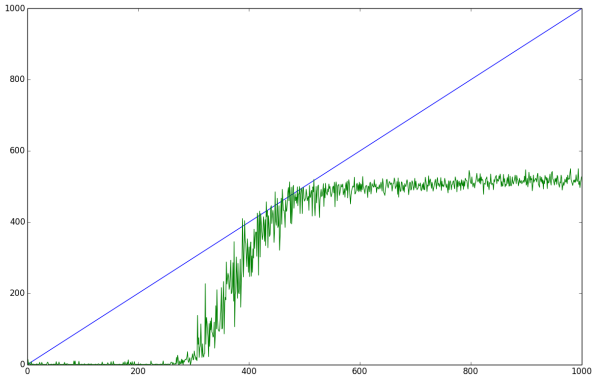


Figure 11: Results generated by the framework similar from Kanerva's original Table 7.2. It was a 1,000 bit, 1,000,000 hard-location SDM with exactly 100 random bitstrings written into it.

z, which suggests experimenting with different values. The results, however, have not yielded performance improvements. Though for $z \leqslant 1$ results are comparable to $z = 1$, for $z > 1$, the system shows a clear deterioration, with a smaller distance to convergence and higher divergence at large-distance reads. This is shown in Figure 13.

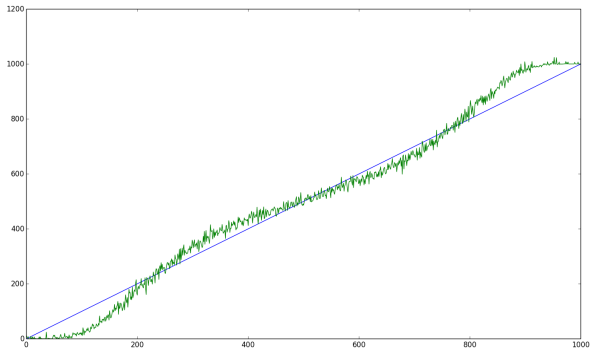
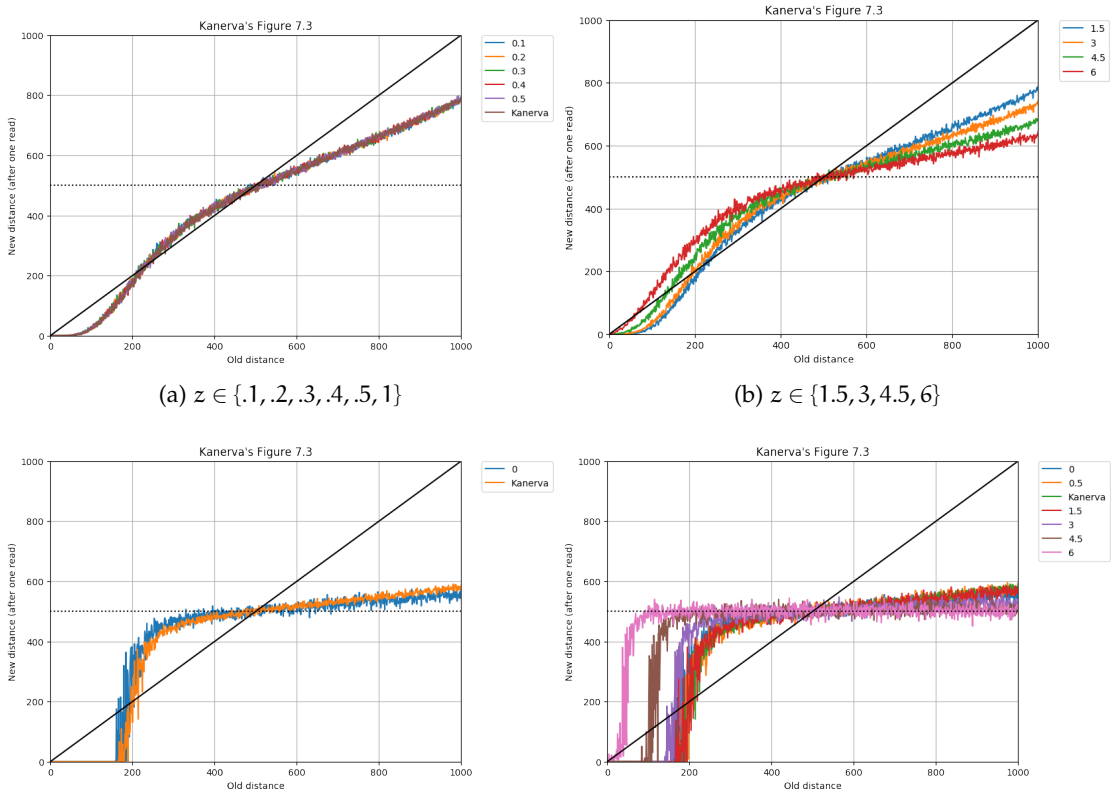


Figure 12: This graph shows the interaction effects more clearly. As we include an opposite bitstring, one can see the accelerating effects towards convergence to both attractors: the origin and the opposite. Here we have the exact same configuration of Figure 10, with the addition of the single opposite attractor.



(a) $z \in \{.1, .2, .3, .4, .5, 1\}$

(b) $z \in \{1.5, 3, 4.5, 6\}$

(c) $z \in \{0, 1\}$

(d) $z \in \{0, 0.5, 1, 1.5, 3, 4.5, 6\}$

Figure 13: (a) and (b) show the behavior of a single read; (c) and (d) present the effects of 6 iterative reads. As stated previously, we can see a deterioration of convergence, with lower critical distance as $z > 1$. Another observation can be made here, concerning the discrepancy of Kanerva's Fig 7.3 and our data. It seems that Kanerva may not have considered that a single read would only 'clean' a small number of dimensions *after the critical distance*. What we observe clearly is that with a single read, as the distance grows, the system only 'cleans' towards the orthoghonal distance 500 after a number of iterative readings.

5

RESULTS (II): PERFORMANCE

Our intention is to provide comparative performance metrics under different computation engines (CPU, GPU, etc) and different operating systems (Linux, MacOs, Windows, etc). Performance can be measured as the average number of scans of all hard locations per second, reads per second, writes per second, etc.

Our first device is a personal MacBook Pro Retina 13-inch Late 2013 with a 2.6GHz Intel core i5 processor, 6GB DDR3 RAM, and Intel Iris GPU. We also intend to test on machines such as the iMac with dedicated GPU, MacPro with dedicated GPU, and personal computers under Linux with dedicated GPUs.

Beyond that, we are running as state-of-the-art devices: (i) an Amazon EC2 p3.xlarge with Intel Xeon E5-2686v4 processor, 61GB DDR3 RAM, and NVIDIA K80 GPU, and (ii) an Amazon EC2 p3.8xlarge with Intel Xeon E5-2686v4 processor, 488GB DDR3 RAM, and 8x NVIDIA K80 GPU.

6

RESULTS (IV): SUPERVISED CLASSIFICATION APPLICATION

Supervised classification problem consists of categorize data into groups after seeing some samples from each group. First, it is presented pieces of data with their categories. The algorithm learns from these data, which is known as learning phase. Then, new pieces of data are presented and the algorithm must classify them into the already known groups. It is named supervised because the algorithm will not create the groups itself. It will learn the groups from during the learning phase, in which the groups have already been defined and the pieces of data have already been classified into them.

Although this problem has already been studied (REF), our intention here is to show that a pure SDM may also be used to classify data. Fan and Wang [7] has used SDM to solve a classification problem, recognizing handwriting letters from images, but he used a mix of genetic algorithm with SDM, which is very different from the original SDM described by [10]. Even though his algorithm has classified properly, we were intrigued whether a pure SDM would also classify successfully.

Hence, we have developed a supervised classification algorithm based on a pure SDM as our main memory. Our goal was to classify noisy images into their respective letters (case sensitive) and numbers. For some examples, see Figure 14.

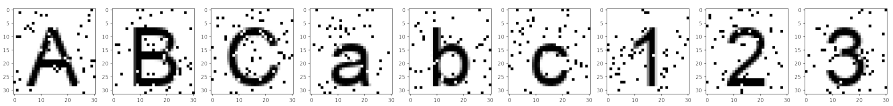


Figure 14: Examples of noisy images with uppercase letters, lowercase letters and numbers.

The images had 31 pixels of width and 32 pixels of height, totaling 992 pixels per image. Each image was mapped into a 1,000 bit bitstring in which the bits were set according to the color of each pixel of the image. So, white pixels were equal to bit 0, and black pixels to bit 1. The 8 remaining bits were all set to zero. This was a bijective mapping (or one-to-one mapping), i.e., there was only one bitstring for each image, and there was only one image for each bitstring.

A total of 62 classification groups have been trained in the SDM. For each of them, it was generated a random bitstring. Thus, the groups' bitstrings were orthogonal between any two of them. There is one

image for each of the 62 groups in Figure 15. Notice that the SDM has never seen a single image with no noise.

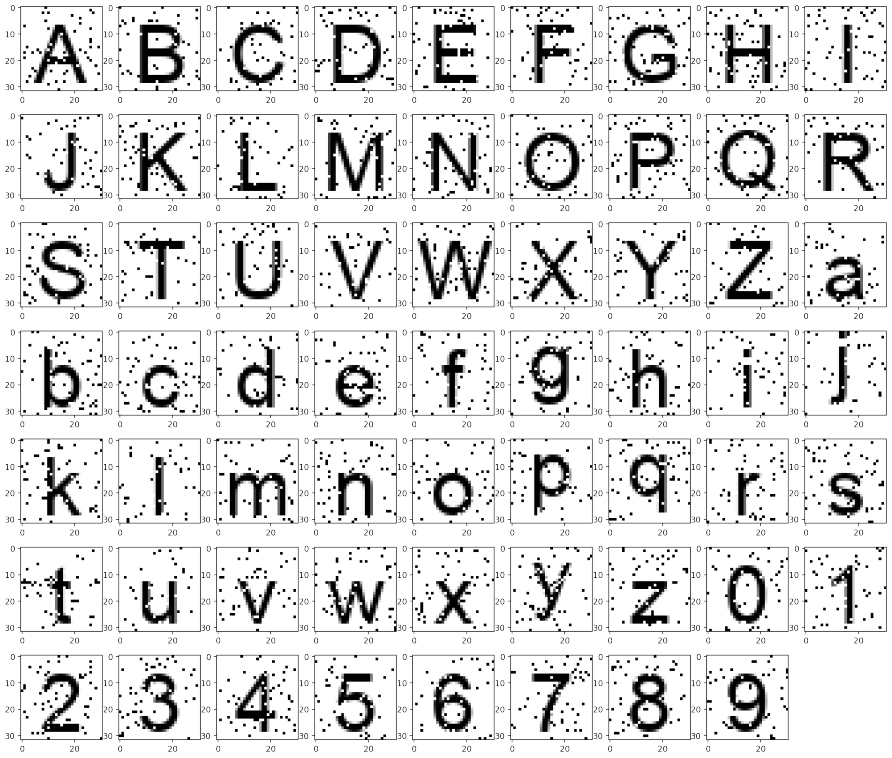


Figure 15: One noisy image for each of the 62 classification groups.

The association of images to groups was stored as sequences in SDM, as detailed by Kanerva [10] in Chapter 8. During the learning phase, the image bitstrings were stored pointing to their groups bitstrings, i.e., write(addr=bs_image, datum=bs_label). Thus, in order to classify an unknown image, we only had to read from its address and check which group has been found.

During the learning phase, we have generated 100 noisy images for each character. The images had 5% of noise, i.e., 5% of their pixels have been randomly flipped. For example, see the generated images for letter A in Figure 16. Then, we have wrote the classification group bitstring into the bitstring associated to each noisy image, i.e., write(bs_image, bs_label). For a complete image training set, see Appendix XYZ.

Finally, we have assess the performance of our classifier. We had done it in three different scenarios: high noise (20%), low noise (5%) and no noise. See Figures 17 and 18 for images with 20% noise and no noise. The low noise scenario had the same noise as the training set. For each scenario, we had classified 620 unknown images with 10 images per group.

The performance was calculated as the percentage of hits for each group. We did not expected the same performance for all groups

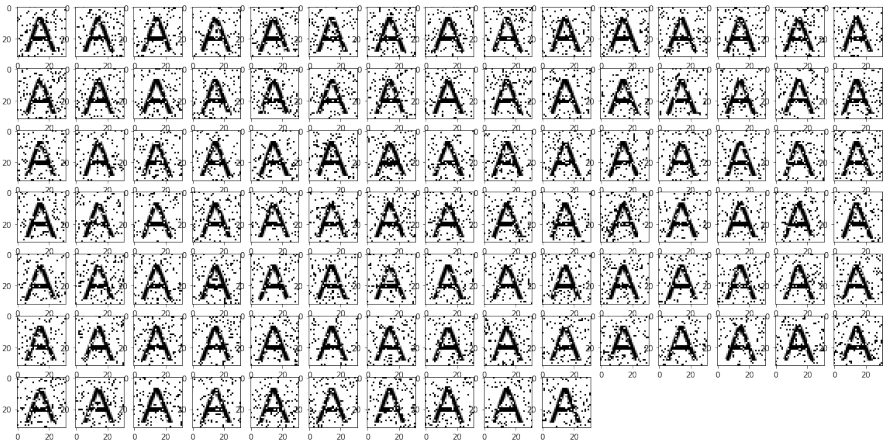


Figure 16: 100 noisy images generated to train label A.

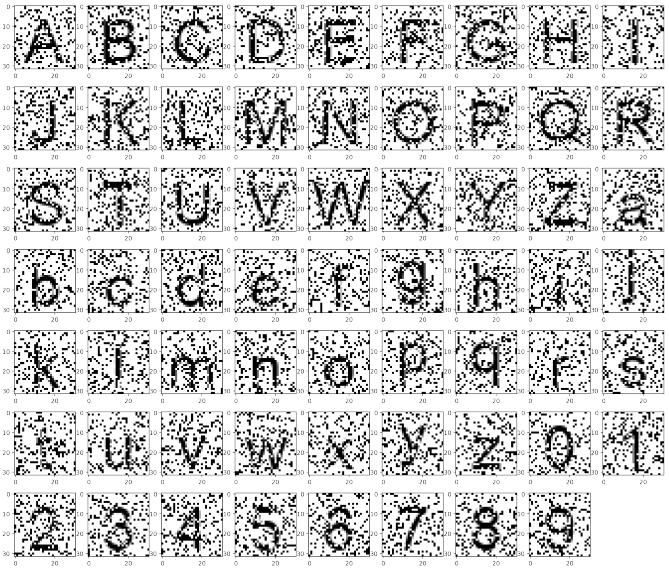


Figure 17: Images generated using a 20% noise for the high noise scenario.

because some groups become very similar to other depending on the noise level, and this similarity may even confuse a person (see Figure 19).

In the no noise scenario, the classifier has hit all characters, except letter "l" which was wrongly associated to the group of "i". We believe that it happened because the classifier had never seen an image with no noise and the difference between the images of "l" and "i" is smaller than the critical distance. So, both groups have been merged and it would converge to only one of them. In our simulation, it happened to be the group of "i".

In the low noise scenario, it has made few mistakes. It correctly classified all images but some from characters "b", "e", "f", "l", "t", and "9". It completely classified "l" images to the "i" group. In the

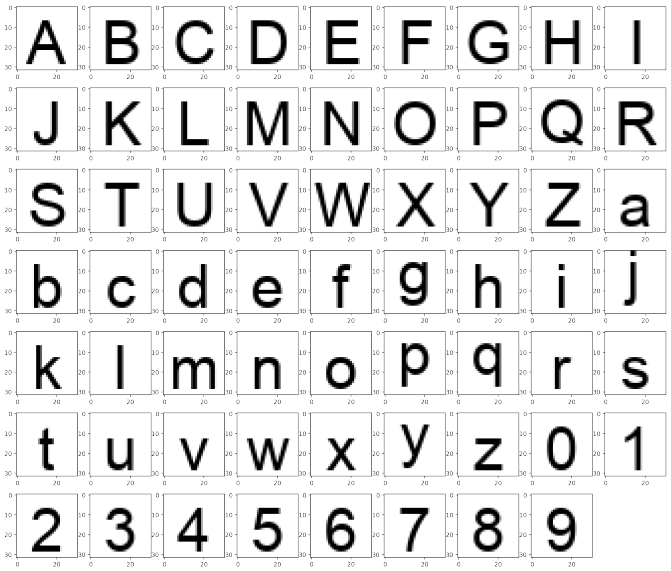


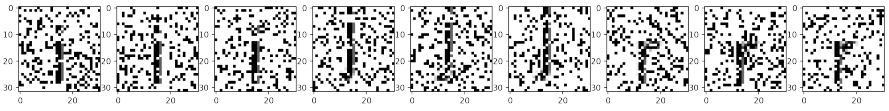
Figure 18: Images generated for the no noise scenario.

other cases, it made just a few mistakes. See Figure 20 to check the images and their classification.

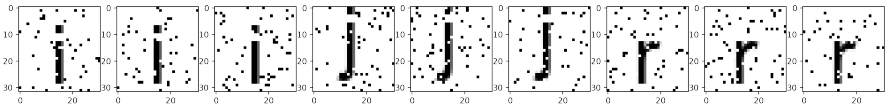
The high noise scenario is the most interesting, because, even in a high noise level, the classifier has hit most of the characters. It has hit all images for 44 out of 62 groups, and made at least one miss for the other 18 groups. The misses may be seen in details in Figure **??**.

The critical distance plays an important role in the classification error. As we have 62 groups and each have been trained with 100 images, there were 6,200 writes to the memory. When an image is being classified, it will have to converge to a group, and the convergence depends on the distance between this image and the images from the training set, i.e, in the noise level.

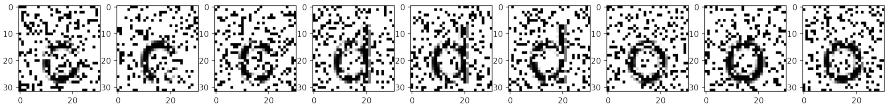
These results show that the SDM may be used as a supervised classification algorithm. Although we do not believe that the mapping between images and bitstrings are even close to the way human cognition deals with images, we believe the results are interesting and useful to many possible real world problems.



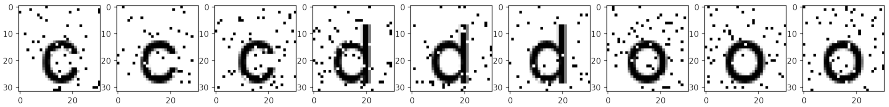
(a) “i”, “l”, and “r” with 20% noise.



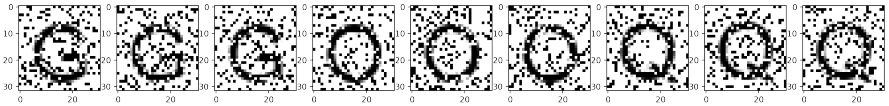
(b) “i”, “l”, and “r” with 5% noise.



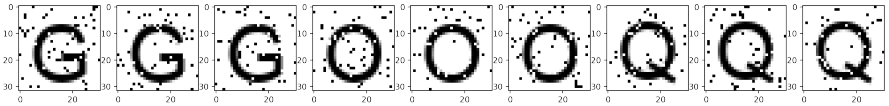
(c) “c”, “d”, and “o” with 20% noise.



(d) “c”, “d”, and “o” with 5% noise.

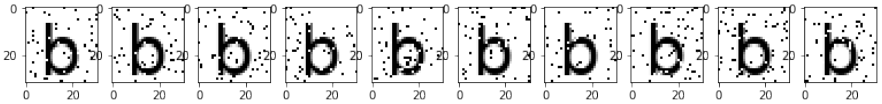


(e) “G”, “O”, and “Q” with 20% noise.

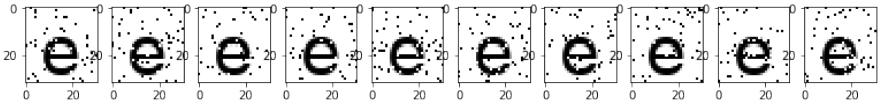


(f) “G”, “O”, and “Q” with 5% noise.

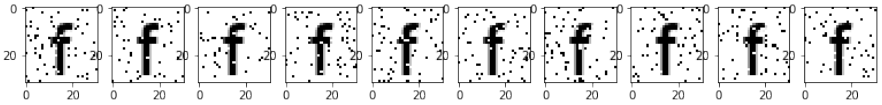
Figure 19: Images of different characters which may be confusing depending on the noise level.



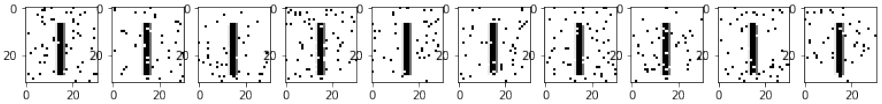
(a) Images from character “b which were classified as [b, b, b, h, b, o, b, h, b, b], respectively. It has made 3 misses.



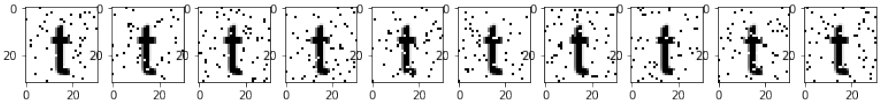
(b) Images from character “e which were classified as [e, e, e, e, e, e, e, e, o, e], respectively. It has made 1 miss.



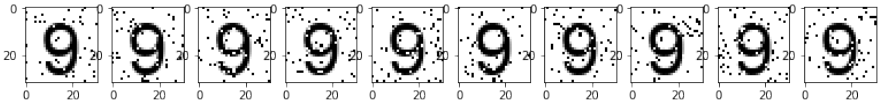
(c) Images from character “f which were classified as [i, f, f, I, I, I, f, f, f, f], respectively. It has made 4 misses.



(d) Images from character “l” which were classified as [i, i, i, i, i, i, i, i, i, i], respectively. It has missed them all, as if both groups have been merged.

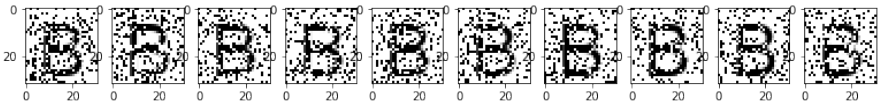


(e) Images from character “t” which were classified as [t, t, t, t, t, t, t, i, t, t], respectively. It has made 1 miss.

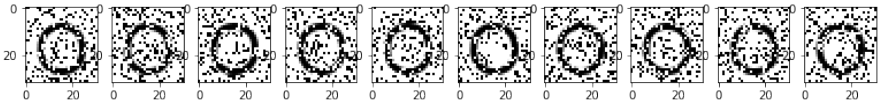


(f) Images from character “9” which were classified as [9, 9, o, 9, 9, 9, o, o, 9, 9], respectively. It has made 3 misses.

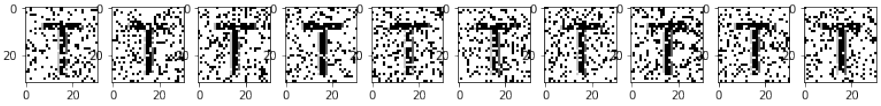
Figure 20: Characters in the low noise scenario in which the classifier has made at least one mistake. In all the other cases, it correctly classified the images. We may notice that the groups of “i” and “l” have been completely merged by the classifier, because it cannot distinguish them, not even with no noise.



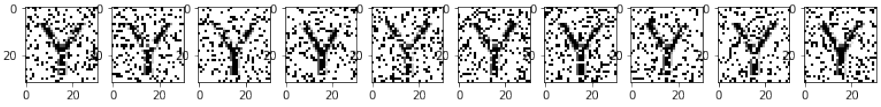
(a) Images from character “B” which were classified as [S, B, B, B, B, B, B, B, B, B]. It has made 1 mistake.



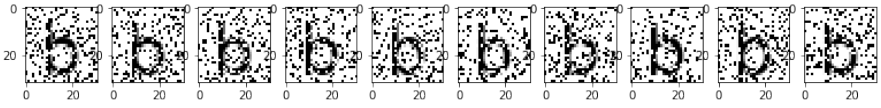
(b) Images from character “O” which were classified as [G, G, O, O, O, O, O, O, O, O]. It has made 2 mistakes.



(c) Images from character “T” which were classified as [T, T, T, T, T, I, T, T, T, T]. It has made 1 mistake.



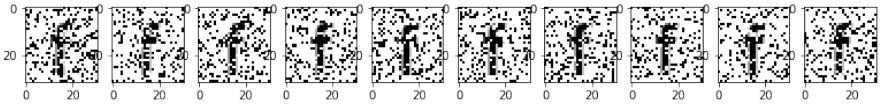
(d) Images from character “Y” which were classified as [Y, I, Y, Y, Y, Y, Y, Y, Y, Y]. It has made 1 mistake.



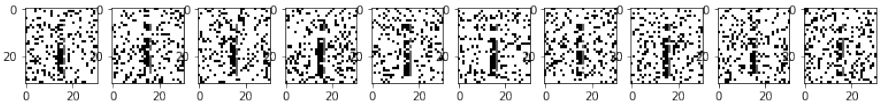
(e) Images from character “b” which were classified as [o, o, o, b, o, h, h, b, b, o]. It has made 7 mistakes.

(f) Images from character “c” which were classified as [c, c, c, c, c, o, c, c, c, o]. It has made 2 mistakes.

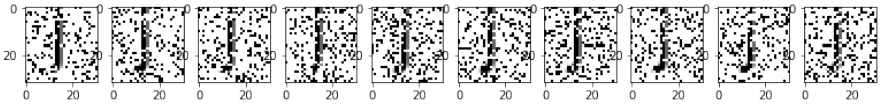
(g) Images from character “e” which were classified as [e, o, e, o, o, o, e, o, o, e]. It has made 6 mistakes.



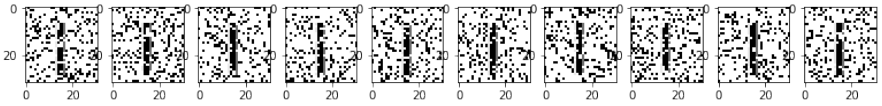
(h) Images from character “f” which were classified as [I, I, I, I, i, I, I, I, I, I]. It has missed them all.



(i) Images from character "i" which were classified as [i, i, i, I, i, i, i, i, I, i]. It has made 2 mistakes.

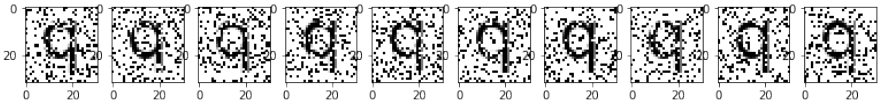


(j) Images from character "j" which were classified as [j, j, j, I, I, j, j, j, j, I]. It has made 3 mistakes.

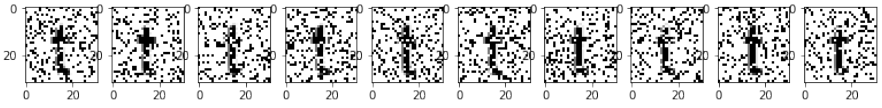


(k) Images from character "l" which were classified as [I, i, I, I, I, I, i, I, I, i]. It has missed them all.

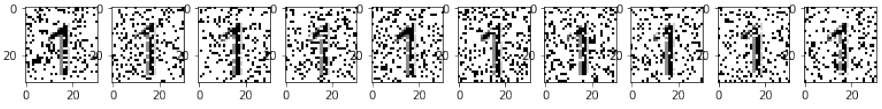
(l) Images from character "n" which were classified as [u, n, n, n, n, n, u, u, u, h]. It has made 5 mistakes.



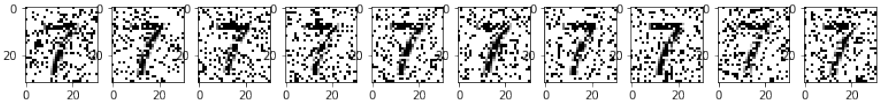
(m) Images from character "q" which were classified as [q, q, q, q, q, q, q, q, q, g]. It has made 1 mistake.



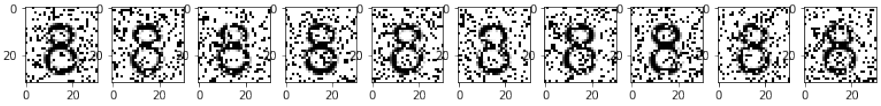
(n) Images from character "t" which were classified as [I, r, I, i, I, i, i, i, I, i]. It has missed them all.



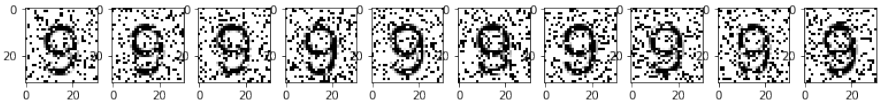
(o) Images from character "1" which were classified as [1, I, 1, I, 1, 1, I, I, 1, I]. It has made 5 mistakes.



(p) Images from character "7" which were classified as [7, 7, 7, I, 7, I, I, 7, 7, 7]. It has made 3 mistakes.



(q) Images from character "8" which were classified as [8, 6, 6, 6, 8, d, 8, 8, d, 6]. It has made 6 mistakes.



(r) Images from character "9" which were classified as [9, 0, 6, 0, 9, 0, 0, 9, 0, 0]. It has made 7 mistakes.

Figure 19: Characters in the high noise scenario in which the classifier has made at least one mistake. In all the other cases, it correctly classified the images.

7

RESULTS (III): SUPERVISED IMAGE NOISE FILTERING APPLICATION

Image noise filtering consists in removing the noise from an input, in out case an image. Our images are black & white images and the noise is generated randomly flipping some of their pixels from black to while and vice versa. In Figure 20, we may see an image with different levels of noise, from 0% to 45% in steps of 5%. It makes no sense to apply 50% of noise because it would absolutely randomize the image.

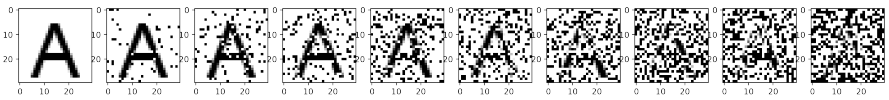


Figure 20: Progressive noise into letter "A", from 0% to 45% in steps of 5%.

The images have 30 x 30 pixels, totaling 900 pixels per image. Each image is mapped into a 1,000 bit bitstring in which the bits are set according to the color of each pixel of the image. White pixels are equal to bit 0, and black pixels to bit 1. The 100 remaining bits are all set to zero. This is a bijective mapping (or one-to-one) from images and bitstrings, i.e., there is one, and only one, bitstring for each image, and vice versa.

In the learning phase, noisy images are generated and they are written into SDM chunked with their labels. The chunk was calculated using the exclusive or (XOR) operator. So, the image bitstring was written to the address of its bitstrings XOR its label bitstring — write($addr = bs_{image} XOR bs_{label}, datum = bs_{image}$).

Finally, in order to remove the noise of a new image, first we have to classify it (possibly using the already presented classification algorithm), and then we just have to read from the chunked address until it converges.

8

RESULTS (V): THE POSSIBILITY OF UNSUPERVISED REINFORCEMENT LEARNING

Reinforcement learning has increasing prominence in the media after AlphaZero has won all games from both the best chess grandmasters in the world and the best chess engines. What is incredible about these victories is that AlphaZero has almost no knowledge about chess game and has learned all its movement playing against itself for 4 hours. Basically, it knowns only the valid movements and had to learn everything from scratch, which it did using a reinforcement learning algorithm.

Reinforcement learning is a machine learning algorithm which learns from the rewards of its actions. So, it receives the game state as input, then it decides which action will be taken, and finally it learns from the rewards of all the actions it has chosen. In theory, it learns after each reward feedback it receives, improving its decision over time and presenting intelligent behavior. A positive reward would indicate that the chosen action should be encouraged. While a negative reward would indicate the opposite. In some algorithms, there may be a neutral reward which would indicate that the chosen action was neither positive nor negative. How each type of reward should be handled depends on each algorithm.

We have done some experiments with an SDM as a memory for a TicTacToe player. Basically, it receives the current board state and returns which action should be played. In the end of the game, it receives the sequences of boards and the winner, and is supposed to learn from them.

Our algorithm to decide what should be player is very simples: it read the current board from SDM. If the reading converges to another board, it chooses the movement which would bring the current board to the one read from SDM. If the reading does not converge, it just play randomly.

After a game has finished, it is time to learn from its decisions. Thus, if SDM wins the game, it will write the whole sequence of boards to SDM. Let $b_0, b_1, b_2, \ldots, b_n$ be the board sequence of the game (see Figure 21). Then it will write $b_0 \to b_1 \to b_2 \to \cdots \to b_n$, with possibly different weights for each transition. If it loses, it will reverse the board (replace X by O and vice versa), and will act as if it had won. Hence, it will learn which sequences lead to victory. When a new board appears, it may have already seen that situation and will decide according to the sequences which goes towards victory. This is our positive reward learning.

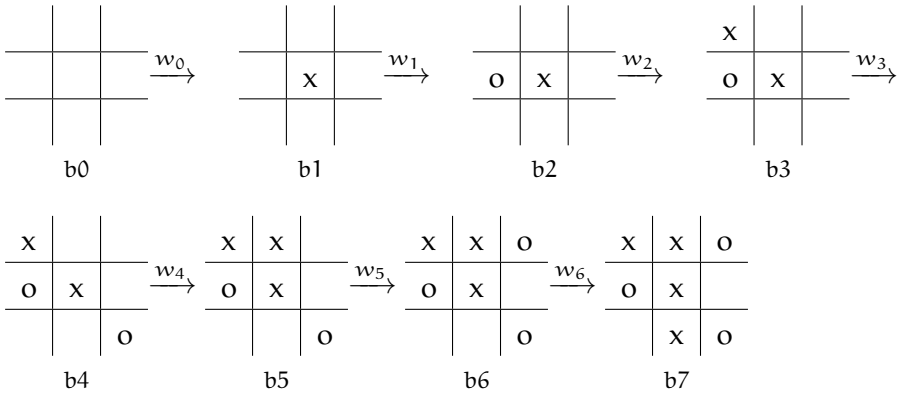


Figure 21: Example of a game with 7 movements in which X wins.

It is also important to learn when a draw happens — after all, it is better to tie than to lose, right? In this case, the sequence of boards is also written to SDM, but with no weight at all. So, if the board has appeared both in a tie sequence and in a winning sequence, it would be more likely to choose the winning one because it was written with greater weight. This is our neutral reward learning.

Finally, we also want to prevent losing games. So, when it loses a game, it will stimulate movements different from the chosen ones. Thus, for each transition $b_k \to b_{k+1}$ made by its action, it will write all possible transitions from b_k but b_{k+1}.

Internally, every board is mapped into a random bitstring and passed to SDM. Thus, SDM knows nothing about the boards themselves. It knowns only about their transition and which ones would lead to either a victory or a draw. As every two boards are orthogonal, SDM does not known whether two boards are consecutives or not. The only link between two boards is the transition written in SDM.

After all, SDM knowns nothing about the boards themselves and yet it may learn how to play TicTacToe.

In order to properly run the discussed algorithms, it is necessary to have two SDMs: a 0-fold and a 1-fold SDM. In the 0-fold SDM, every bitstring is written to its own address. In the 1-fold SDM, every bitstring points somewhere else. So, the transitions are written in the 1-fold SDM, while the boards themselves are written to the 0-fold SDM. The boards are written only once in the 0-fold, no matter how many times they appear. The transitions may be written more than once in the 1-fold SDM, because it would reinforce that transition.

In more details, the next movement decision consists in one read from the 1-fold SDM, resulting in a bitstring. Then this bitstring is used in an interative reading from the 0-fold SDM, which will converge to the bitstring associated with the next board. If it does

not converge to any board, than SDM will choose a random movement and learn from it.

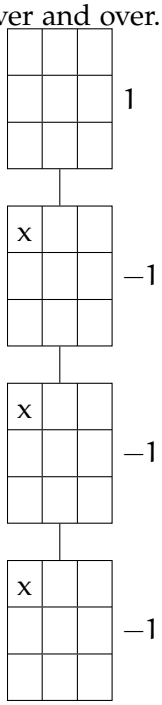
The weight used when writing a winning sequence is calculated using ...

— talk about player generations —

8.0.1 *Training*

It is an unsupervised algorithm because SDM learns playing against an opponent, who may be another SDM player, a human, or a player whose movements are always aleatory.

Thus, in order to train a SDM player, we just have to keep it playing over and over.



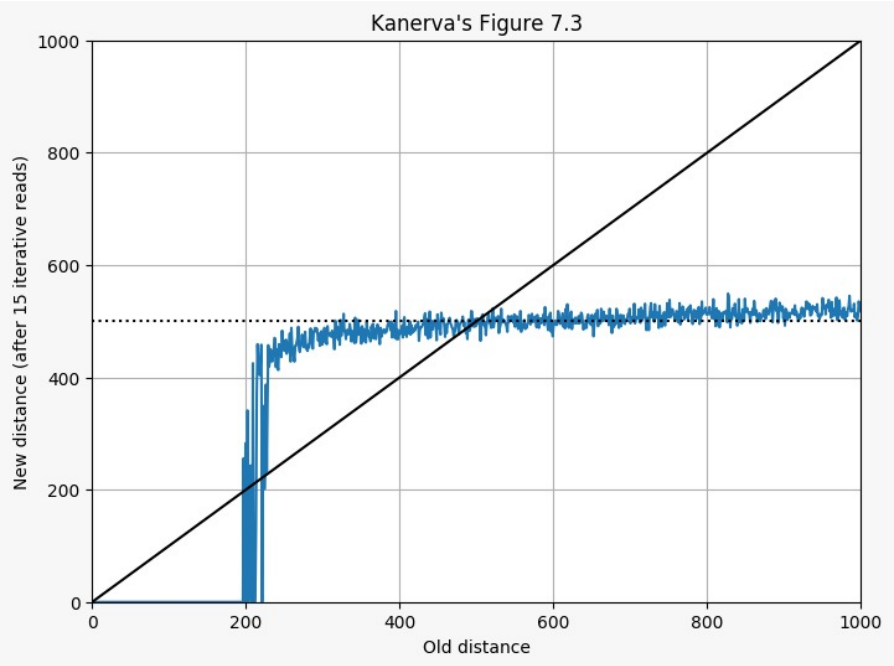
8.0.2 *Results*

9

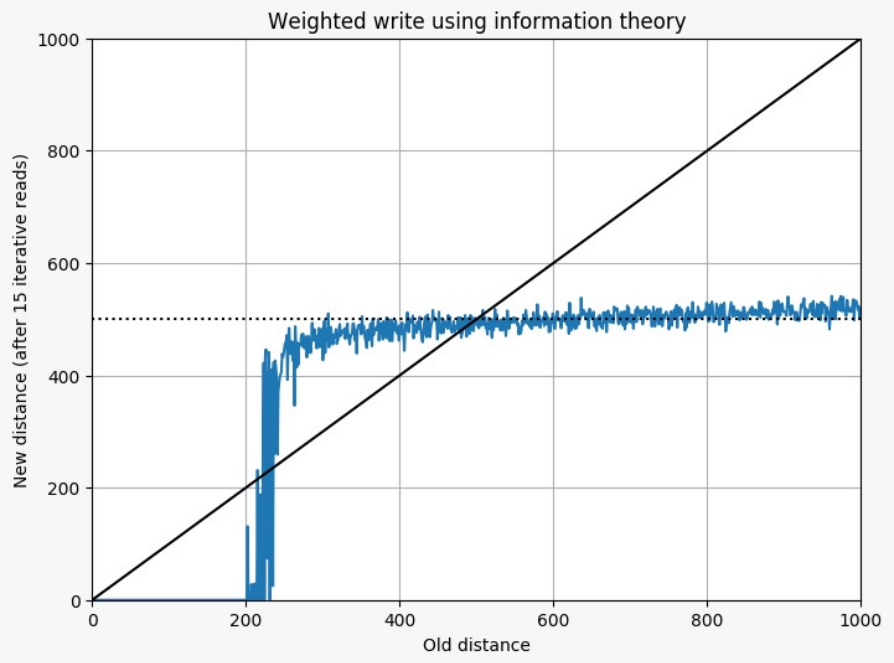
RESULTS (VI): INFORMATION THEORETICAL WRITE OPERATION

My advisor, Alexandre Linhares, has proposed another read operation: the weighted reading. In it, the sum of the counter's value is weighted based on the distance between the hard-location's address and the reading address. The logic behind it is to vary the importance of each hard-location inside the circle. It is only natural that one encodes an item in near hard locations with a stronger signal, and a natural candidate for this signal function is the amount of information contained in the distance between the item and each hard location. Closer hard locations have lower probabilies and therefore should encode more information. We know that the probability of a distance d is $\binom{N}{d}/2^N$, and, acordingly, we have tested how SDM behaves when write weights are calculated using the amount of information given by each bitstring, i.e., $w(d) = -\log_2\left(2^{-N}\binom{N}{d}\right) = N - \log_2\binom{N}{d}$, as in information theory [5].

The results can be seen in Figure 22 and seem promising. It seems that the critical distance increases by a number of bits. Note that 10 additional bits imply an attractor 2^{10} of the size of the original. Another point to keep in mind is that, since the modulus of the vectors are not uniform in this approach, that the shape of the attractor may have large asymmetries.



(a) Kanerva's model



(b) Write process weighted by the amount of information contained in the distance between the written bitstring and each hard location

Figure 22: (a) and (b) show the behavior of the critical distance under Kanerva's model and the information-theoretic one, respectively.

10 CONCLUSION

Sparse Distributed Memory is a viable model of human memory, yet it does require researchers to (re-)implement a number of parallel algorithms in different architectures.

We propose to provide a new, open-source, cross-platform, highly parallel framework in which researchers may be able to create hypotheses and test them computationally through minimal effort. The framework is well-documented for public release at this time (http://sdm-framework.readthedocs.io), it has already served as the backbone of Chada's Ph.D. thesis. The single-line command "pip install sdm" will install the framework on posix-like systems, and single-line commands will let users test the framework, generate some of the figures from Kanerva's theoretical predictions in their own machines, and — if interested enough —, test their own theories and improve the framework, and the benchmarks used to evaluate the framework, in open-source fashion. It is our belief that such work is a necessary component towards accelerating research in this promising field.

10.1 FUTURE WORK

Here are some ideas that have been considered during this work, but have had to be left for future research.

10.1.1 *Multiple levels*

10.1.2 *Classification with context using sequences — for words instead of only letters*

10.1.3 *Symmetrical Hard Locations*

Here is an interesting question that I leave for further research: A hypercube with n dimensions can be divided by two hypercubes with $n-1$ dimensions. Is there an algorithm that separates the area of each hard-location in such a form that there exists a function mapping each bitstring in $\{0,1\}^n$ to the set of hard locations it 'belongs to'? In other words... though this would break Kanerva's assumption of a uniformly distributed set of hard locations for a perfectly symmetrical set of hard locations, there could be large performance gains if such a mapping function from a bitstring to its corresponding set of nearest hard locations exists.

We have generated a Docker image, which makes it even easier to explore the framework. After running the container, a Jupyter Notebook is available with sdm-framework and other tools already installed. The simulations run in this thesis are promptly available to be re-executed and explored. We invite readers to take a look and explore a little bit.

11

APPENDIX

BIBLIOGRAPHY

[1] M. S. Brogliato. Understanding the critical distance in sparse distributed memory. Master's thesis, Escola Brasileira de Administração Pública e de Empresas - EBAPE, Fundação Getulio Vargas, 2011. forthcoming.

[2] Marcelo S Brogliato, Daniel M Chada, and Alexandre Linhares. Sparse distributed memory: understanding the speed and robustness of expert memory. *Frontiers in human neuroscience*, 8: 222, 2014.

[3] Timothy M Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the ram, revisited. In *Proceedings of the twenty-seventh annual symposium on Computational geometry*, pages 1–10. ACM, 2011.

[4] Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[5] Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[6] N. Cowan. The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, 24:87–185, 2000.

[7] Kuo-Chin Fan and Yuan-Kai Wang. A genetic sparse distributed memory approach to the application of handwritten character recognition. *Pattern Recognition*, 30(12):2015–2022, 1997.

[8] R. M. French. When coffee cups are like old elephants, or why representation modules dont make sense. In A. Riegler and M. Peschl, editors, *Proceedings of the 1997 International Conference on New Trends in Cognitive Science*, pages 158–163. Austrian Society for Cognitive Science, 1997.

[9] Frank Harary, John P Hayes, and Horng-Jyh Wu. A survey of the theory of hypercube graphs. *Computers & Mathematics with Applications*, 15(4):277–289, 1988.

[10] P. Kanerva. *Sparse Distributed Memory*. MIT Press, 1988.

[11] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al.

Jupyter notebooks-a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90, 2016.

[12] A. Linhares, D. M. Chada, C. N. Aranha, and . The emergence of miller's magic number on a sparse distributed memory. *Public Library of Science (PLOS) One*, 6(1):e15592, Jan 2011. doi: 10.1371/journal.pone.0015592.

[13] Mateus Mendes, Manuel Crisóstomo, and A Paulo Coimbra. Robot navigation using a sparse distributed memory. In *Robotics and automation, 2008. ICRA 2008. IEEE international conference on*, pages 53–58. IEEE, 2008.

[14] Meng et al. A modified sparse distributed memory model for extracting clean patterns from noisy inputs. *Proceedings of International Joint Conference on Neural Networks*, June 2009.

[15] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63:81–97, 1955.

[16] Kenneth A Norman and Randall C O'reilly. Modeling hippocampal and neocortical contributions to recognition memory: a complementary-learning-systems approach. *Psychological review*, 110(4):611, 2003.

[17] Mohammad Norouzi, Ali Punjani, and David J Fleet. Fast exact search in hamming space with multi-index hashing. *IEEE transactions on pattern analysis and machine intelligence*, 36(6):1107–1119, 2014.

[18] Ram Pai, Badari Pulavarty, and Mingming Cao. Linux 2.6 performance improvement through readahead optimization. In *Proceedings of the Linux Symposium*, volume 2, pages 105–116, 2004.

[19] Rajesh Rao and Olac Fuentes. Hierarchical learning of navigational behaviors in an autonomous robot using a predictive sparse distributed memory. *Machine Learning*, pages 87–113, 1998.

[20] Rajesh PN Rao and Dana H Ballard. Natural basis functions and topographic memory for face recognition. In *IJCAI*, pages 10–19, 1995.

[21] Javier Snaider and Stan Franklin. Extended sparse distributed memory. Paper presented at the Biological Inspired Cognitive Architectures 2011, Washington D.C. USA.

[22] Henry S Warren. *Hacker's delight*. Pearson Education, 2013.