1999

Using Genetic Algorithms for Sparse Distributed Memory Initialization

A. Anwar
D. Dasgupta
S. Franklin

Follow this and additional works at: https://digitalcommons.memphis.edu/ccrg_papers

Recommended Citation

This Document is brought to you for free and open access by the Cognitive Computing Research Group at University of Memphis Digital Commons. It has been accepted for inclusion in CCRG Papers by an authorized administrator of University of Memphis Digital Commons. For more information, please contact khggerty@memphis.edu.
Using Genetic Algorithms for Sparse Distributed Memory Initialization

Ashraf Anwar$^{1,2}$
Institute for Intelligent Systems
The University of Memphis
Memphis, TN 38152
aanwar1@memphis.edu

Dipankar Dasgupta$^3$
Institute for Intelligent Systems
The University of Memphis
Memphis, TN 38152
dasgupta@msci.memphis.edu

Stan Franklin$^{1,2}$
Institute for Intelligent Systems
The University of Memphis
Memphis, TN 38152
stan.franklin@memphis.edu

Abstract- In this paper, we describe the use of Genetic Algorithms to initialize a set of hard locations that makes up the storage space of Sparse Distributed Memory (SDM). SDM is an associative memory technique that uses binary spaces, and relies on close memory items tending to be clustered together, with some level of abstraction and with details blurring. An important factor in the physical implementation of SDM is how many hard locations are used, which greatly affects the memory capacity. It is also dependent somehow on the dimension of the binary space used. For the SDM system to function appropriately, the hard locations should be uniformly distributed over the binary space. We represented a set of hard locations of SDM as population members, and employed GA to search for the best (fittest) distribution of hard locations over the vast binary space. Fitness is based on how far each hard location is from all other hard locations, which measures the uniformity of the distribution. The preliminary results are very promising, with the GA significantly outperforming random initialization used in most existing SDM implementations. This use of GA, which is similar to the Michigan approach, differs from the standard approach in that the object of the search is the entire population.

1 Introduction

Genetic Algorithms (GA) are finding increasing number of applications in a variety of search, optimization, machine learning and other problems (Dasgupta and McGregor 1992; Dasgupta and Michalewicz 1997; Davis 1991; Fan and Wang 1997). They are usually used whenever there is a large search space, and no efficient algorithm, for trimming or pruning it. Sparse Distributed Memory (SDM) is an associative memory model invented by Kanerva (1988a), which we think is a good model for the human memory. SDM has a vast binary space of possible addresses in the semantic space, whereas for any practical application only a very small portion of this space can actually exist. This makes SDM a perfect match for GA search. Having so many possibilities for assignment of actual (hard) locations to addresses suggests the use of GA. GA proved to be much more superior to the random initialization method that is widely used.

The only SDM implementation that used non-random initialization of hard locations, is the GSDM work (Fan and Wang 1997), in which they initialized the locations based upon the data set to be stored. This can serve to reduce the memory requirement for the implementation, whereas limiting generalization, and learning capabilities, as the memory becomes full. They used the mutation and crossover genetic operators to allow for change or noise in the assigned addresses, based upon the selected existing addresses, if any, until all available memory locations are assigned to the available input data set.

Our approach is totally different, and independent of theirs. The idea here is to keep the general purpose huge memory SDM as is, but to invoke GA for better distribution of the hard locations over the huge semantic space. This approach has two main advantages. It maintains the original functionality of SDM and, at the same time, allows for non-biased distribution of the hard locations over the semantic space, which allows for a domain independent and general associative memory and learning.

There are also neural network implementations of SDM (Hely et al 1997; Joglekar 1989; Scott et al 1993), in which weights for the hidden layer (the hard locations) are initialized randomly. Such random initialization of the hidden layer weights corresponds initially to the random initialization of hard locations in traditional SDM, but allows for dynamic address change as learning or training takes place so that the hard locations correspond to the addresses of the training patterns.

SDM has proved successful in modeling associative memory (Anwar 1997; Hely 1994; Keeler 1987; Scott 1993). Associative memory is typically needed for any smart autonomous agent for various reasons (Glenberg 1997; Kosslyn 1992). Cognitive and Conscious agents typically need such a memory (Franklin 1997; Franklin and Graesser 1997). Cmattie, a clerical software agent, uses associative memory to keep track of departmental seminar announcements (Bogner 1999; McCauley 1998;
2 Sparse Distributed Memory

2.1 A Brief Tour with SDM

Sparse Distributed Memory, SDM, is the work of Pentti Kanerva (1988a;b; 1992) which gets its name from the sparse allocation of storage locations in the vast binary address space, and the distributed nature of information storage and retrieval as we will explain later. For more details and discussions, see (Franklin 1995) for a brief overview, (Willshaw 1990) for a useful commentary, or (Anwar 1997, Chapter 2) for a more detailed summary of its mechanism. Many evaluations, extensions, and enhancements have been suggested for SDM (Evans and Surkan 1991; Karlsson 1995; Kristoferson 1995a;b; Rogers 1988b; Ryan and Andreae 1995).

The auto-associative version of SDM is truly an associative memory technique where the contents and the addresses belong to the same space and are used alternatively. The dimension of the space determines how rich each word is. Another important factor is how many real memory locations are in the space. We call these hard locations. Features are represented as one or more bits. Groups of features are concatenated to form a word, which becomes a candidate for writing into SDM. When writing, a copy of this binary string is placed in all close-enough hard locations. When reading, a cue would reach all close enough hard locations and get some sort of aggregate or average return from them. Reading is not always successful. Depending on the cue and the previously written information, among other factors, convergence or divergence during a reading operation may occur. If convergence occurs, the pooled word will be the closest match, possibly with some sort of abstraction, of the input reading cue. On the other hand, when divergence occurs, there is no relation, in general, between the input cue and what is retrieved from SDM.

2.2 SDM from the Inside Out

SDM is a content addressable memory that, in many ways, is ideal for use as a long-term associative memory. Content addressable means that items in memory can be retrieved by using part of their contents as a cue, rather than having to know their addresses in memory.

Boolean geometry is the geometry of Boolean spaces. A Boolean space is the set of all Boolean vectors (that is, vectors composed of zeros and ones) of some fixed length, n, called the dimension of the space. Points in Boolean space are Boolean vectors. The Boolean space of dimension n contains 2^n Boolean vectors, each of length n. The number of points increases exponentially as the dimension increases. Boolean geometry uses a metric called the Hamming distance, where the distance between two points is the number of coordinates at which they differ. Thus \( d((1,0,0,1,0), (1,0,1,1,1)) = 2 \). The distance between two points will measure the similarity between two memory items, closer points being more similar. We may think of these Boolean vectors as feature vectors, where each feature can be only on, 1, or off, 0. Two such feature vectors are closer together if more of their features are the same. For n = 1000, most of the space lies between distance 422 and distance 578 from a given vector. In other words, almost all the space is far away from any given vector. A Boolean space implementation is typically sparsely populated, an important property for the construction of the model, and the source of part of its name.

A sphere is the set of all points within some fixed distance, the radius, from its center. Spheres in Boolean space are quite different in one respect from the Euclidean spheres we are used to. Points of an Euclidean sphere are uniformly distributed throughout. For \( r = n/2 \), most of the points in a sphere in a Boolean space lie close to its boundary.

SDM address space is enormous. It contains 2^n locations, where n is the dimension of the binary space. For n=1000, one cannot hope for such a vast memory. On the other hand, thinking of feature vectors, a thousand features wouldn't deal with human visual input until a high level of abstraction had been reached. A dimension of 1000 may not be all that much; it may, for some purposes, be unrealistically small. Kanerva proposes to deal with this vast address space by choosing a uniform random sample, size \( 2^{20} \), of locations, that is, about a million of them. These he calls hard locations. With \( 2^{20} \) hard locations out of \( 2^{1000} \) possible locations, the ratio is \( 2^{-980} \), very sparse indeed. In addition, the distance from a random location in the entire address space to the nearest hard location will fall between 411 and 430 ninety-eight percent of the time, with the median distance being 424. The hard locations are certainly sparse.

SDM is distributed in that many hard locations participate in storing and retrieving each datum, and one hard location can be involved in the storage and retrieval of many data. This is a very different beast than the store-one-datum-in-one-location type of memory to which we're accustomed. Each hard location, itself a bit vector of length 1000, stores data in 1000 counters, each with range -40 to 40. We now have about a million hard locations, each with a thousand counters, totaling a billion counters in all. Numbers in the range -40 to 40 will take most of a byte to store. Thus we are talking about a billion bytes, a gigabyte, of memory. Quite a lot, but not out of the question.

How do these counters work? Writing a 1 to the counter increments it; writing a 0 decrements it. A datum, \( \xi \), to be
written is a bit vector of length 1000.\(^4\) To write \(\xi\) at a given hard location \(x\), write each coordinate of \(\xi\) to the corresponding counter in \(x\), either incrementing it or decrementing it.

Call the sphere of radius 451 centered at location \(\zeta\) the access sphere of that location. An access sphere typically contains about a thousand hard locations, with the closest to \(\zeta\) usually some 424 bits away and the median distance from \(\zeta\) to the hard locations in its access sphere about 448. Any hard location in the access sphere of \(\xi\) is accessible from \(\xi\). With this machinery in hand, we can now write distributively to any location, hard or not. To write a datum \(\xi\) to a location \(\zeta\), simply write \(\xi\) to each of the roughly one thousand hard locations accessible from \(\zeta\). Distributed storage.

With our datum distributively stored, the next question is how to retrieve it. With this in mind, let’s ask first how one reads from a single hard location, \(x\). Compute \(\zeta_x\), the bit vector read at \(x\), by assigning its \(i\)th bit the value 1 or 0 according as \(x\)‘s \(i\)th counter is positive or negative. Thus, each bit of \(\zeta_x\) results from a majority rule decision of all the data that have been written on \(x\). The read datum, \(\zeta_x\), is an archetype of the data that have been written to \(x\), but may not be any one of them. From another point of view, \(\zeta_x\) is the datum with smallest mean distance from all data that have been written to \(x\).

Knowing how to read from a hard location allows us to read from any of the \(2^{1000}\) arbitrary locations. Suppose \(\zeta\) is any location. The bit vector, \(\xi\), to be read at \(\zeta\), is formed by pooling the data read from each hard location accessible from \(\zeta\). Each bit of \(\xi\) results from a majority rule decision over the pooled data. Specifically, to get the \(i\)th bit of \(\xi\), add together the \(i\)th bits of the data read from hard locations accessible from \(\zeta\), and use half the number of such hard locations as a threshold. At or over threshold, assign a 1. Below threshold assign a 0. Put another way, pool the bit vectors read from hard locations accessible from \(\zeta\), and let each of their \(i\)th bits votes on the \(i\)th bit of \(\xi\).

We now know how to write items into memory, and how to read them out. But what’s the relation between the datum in and the datum out? Are these two bit vectors the same, as we’d hope? Let’s first look at the special case where the datum \(\xi\) is written at the location \(\xi\). This makes sense since both are bit vectors of length one thousand. Kanerva offers a mathematical proof that reading form \(\xi\) recovers \(\xi\) (Kanerva 1988a). Here’s the idea of the proof. Reading from \(\xi\) recovers archetypes from each of some thousand hard locations and takes a vote. The voting is influenced by the \(~1000\) stored copies of \(\xi\) and, typically, by about 10,000 other stored data items. Since the intersection of two access spheres is typically quite small, these other data items influence a given coordinate only in small groups of ones or twos or threes, i.e. white noise. The thousand copies drown out this slight noise and \(\xi\) is successfully reconstructed.

Not all of the stored items are needed to recover it. Iterated reading allows recovery when reading from a noisy version of what’s been stored. Again, Kanerva offers conditions (involving how much of the stored item is available for the read) under which this is true, and a mathematical proof. Reading with a cue that has never been written to SDM before gives, if convergent, the closest match stored in SDM to the input cue, with some sort of abstraction. Since a convergent sequence of iterates converges very rapidly, while a divergent sequence of iterates bounces about seemingly at random, comparison of adjacent items in the sequence quickly tells whether or not a sequence converges. Thus, this memory is content addressable, provided we write each datum with itself as address.

The above discussion, based on the identity of datum and address, produced a content addressable memory with many pleasing properties. It works well for reconstructing individual memories.

3 The GA Design and Implementation

3.1 Population Formation

Binary encoding was used since we are dealing with binary spaces. For various values of chromosome length (binary space dimension), the population size (number of hard locations) was determined.

To be noted here is that the number of hard locations employed in a certain SDM greatly affects the capacity of the resulting associative memory. It is a design decision after all. In our experiments, we took the size of our storage memory (number of hard locations), to be the same as the dimension (chromosome length), resulting in a non-linear occupation ratio. This makes the storage size equal to the binary log of the vast semantic address space. For \(n=10\), we have \(2^{100}\) possible addresses over the semantic space, out of which we only assign 100 hard locations. This is different from other occupation ratios employed in other SDM implementations, but again it is after all a design decision based mainly on the memory capacity required.

For \(n=10\), 1110011000 is an example of an address or a hard location to be assigned. If we were to have only two hard locations in such a case, then the best two assigned addresses will be at 10 hamming distance from each other, e.g. 1111111111 and 0000000000. The best population will have individuals that are as far apart as possible from each other on the average, with a small enough standard deviation. It helps also to have each pair of individuals at least a certain threshold apart.

The size of the search space is clearly \(2^n\). It is immediately apparent how such a space becomes too huge for any traditional search technique as \(n\) becomes large.

\(^4\) We will try to adhere to Kanerva’s convention of using lower case Greek letters for locations and for data, and lower case Roman letters for hard locations. The Greek letters will include \(\xi\) (xi), \(\eta\) (eta), and \(\zeta\) (zeta).
Even a traditional heuristic search technique will find it too hard to arrive at a good coverage of the space in an efficient manner.

3.2 The Fitness Measure

The fitness function for an individual $m$ is given by:

$$\sum \text{Hamming Distance}(m,n) \quad \text{for all } n \neq m$$

In other words, it is the sum of the hamming distances between that individual and all others, subject to two constraints:

i. The individual is unique in the population (no repetition). This is achieved by checking each new child to be added to a generation against previous members of the generation. If the child is not unique, it is discarded, and the genetic operators are reused to produce another unique child.

ii. The individual is at least $k$ bits apart from all other individuals (threshold for hamming distance where $k>0$). Similar precautions are followed to preserve this condition just as in i. To be noted here, is that the threshold is typically not high, in order to avoid unnecessary reiteration.

So, the target is to maximize the mean fitness per generation according to the above two constraints, while preserving a reasonably low standard deviation.

Note that our goal is a whole population rather than a particularly fit individual as is usually the case. Actually, there is a whole spectrum between the two extremes of group fitness, the Michigan approach, and individual fitness, the Pittsburgh approach (Grefenstette 1987; 1988; 1994; Schaffer and Grefenstette 1985). Our group fitness measure is similar to the Michigan approach.

3.3 The Genetic Operators

Many parameter tunings were considered and compared. The following set of parameters tends to produce the best results among all sets considered. More testing and experimenting with different values of GA parameters is being done.

Roulette Selection (Davis 1991; Goldenberg 1989), was used, where individuals get selected in proportion to their fitness. Ranked Replacement was considered where we would replace the worst parent always, and only if the child is better than the parent. Elitism was employed to keep the best fitting individual. Invert Mutation with 2% mutation rate was used along with One-point Crossover with 1.0 rate.

The above operators can never span the entire search space for large values of $n$ (space dimension) due to the exponential nature of the space, which becomes too large for any computational algorithm to cover as $n$ increases. However, it is reasonable to assume that they will span enough representatives of the various regions in the space.

We have to consider the very huge size of the space before we can even dream about spanning the entire space with any means. GA however can play the role of a good heuristic, picking representatives out of vast regions in the search space. This is similar in some sense to the basic functionality of SDM when reading or writing, since we consider only actual sparse hard locations existing in access spheres (see Section 2.2 above for explanation).

3.4 Implementation

The GA software package Sugal version 2.1 was used on Solaris UNIX over Sparc 5 platform. We had to modify the fitness function, and replacement strategies in the source code to suit our complex group fitness function, and the constraints imposed on the GA search. The fitness function now is a function of the entire generation, not only one individual. Also the replacement strategy was modified to observe the constraints mentioned in sec 3.2.

4 Results

When $n$ is both the space dimension as well as the number of hard locations, it can be shown that the optimal mean fitness is $\lceil n/2 \rceil$ (For $n=2$, 00 and 11 are optimal, with mean fitness of 2. For $n=3$, 000, 011, 101 are optimal with mean fitness of 4. For $n=4$, 0000, 1111, 1010, 0101 are optimal, with mean fitness of 8). Apparently, as long as that condition holds, and for small values of $n$, we may be able to use a deterministic algorithm. But as $n$ becomes large, and the number of hard locations increases significantly over $n$, the need for a heuristic like GA arises.

Table 1 shows the improvement GA has over random initialization in terms of the average run time, taken over 20 sample runs, required to reach the target mean fitness with a specific standard deviation (SD). The times shown in the table are in the format “h:mm:ss:dd”. Each set of two rows gives the results of GA vs. random initialization for 6%, 4%, and 2% standard deviation from the target mean fitness, for a certain number of hard locations. The second row in each set explicitly gives the improvement gained by GA in percentage.

The target is to reach certain group fitness with a limit on the standard deviation of the individuals. The lower the standard deviation, the better is the resulting population, and the harder is the task. Note that for non-significant SD, both methods reach target almost instantly. In this case, the random approach may then perform as well as the GA, or even slightly better in rare occasions. When tighter constraints are in place, the GA significantly outperforms random initialization. The improvement can be as high as about 41000 times faster, or 4,114,200%. The non-monotonic improvement of GA can be due in part to the huge size of the search space, which is never entirely spanned by any means, for large values of $n$. 

Table 1: Average Runtimes for GA Initialization compared to Random Initialization.

<table>
<thead>
<tr>
<th>No. of Hard Locations</th>
<th>Random 6% SD</th>
<th>GA 6% SD</th>
<th>Random 4% SD</th>
<th>GA 6% SD</th>
<th>Random 2% SD</th>
<th>GA 2% SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0:00:00.09</td>
<td>0:00:00.01</td>
<td>0:00:07.00</td>
<td>0:00:00.14</td>
<td>0:03:16.80</td>
<td>0:00:01.83</td>
</tr>
<tr>
<td>GA Gain</td>
<td>800%</td>
<td>4,900%</td>
<td>10,654%</td>
<td>10,654%</td>
<td>10,654%</td>
<td>10,654%</td>
</tr>
<tr>
<td>20</td>
<td>0:00:00.01</td>
<td>0:00:00.01</td>
<td>0:00:04.60</td>
<td>0:00:00.03</td>
<td>6:34:11.10</td>
<td>0:00:04.60</td>
</tr>
<tr>
<td>GA Gain</td>
<td>0%</td>
<td>15,233%</td>
<td>514,054%</td>
<td>514,054%</td>
<td>514,054%</td>
<td>514,054%</td>
</tr>
<tr>
<td>30</td>
<td>0:00:00.10</td>
<td>0:00:00.07</td>
<td>0:00:00.80</td>
<td>0:00:00.04</td>
<td>6:57:08.65</td>
<td>0:00:04.30</td>
</tr>
<tr>
<td>GA Gain</td>
<td>43%</td>
<td>2,000%</td>
<td>581,962%</td>
<td>581,962%</td>
<td>581,962%</td>
<td>581,962%</td>
</tr>
<tr>
<td>40</td>
<td>0:00:00.09</td>
<td>0:00:00.10</td>
<td>0:00:02.50</td>
<td>0:00:00.10</td>
<td>8:00:00.10</td>
<td>0:00:00.70</td>
</tr>
<tr>
<td>GA Gain</td>
<td>-10%</td>
<td>2,400%</td>
<td>4,114,200%</td>
<td>4,114,200%</td>
<td>4,114,200%</td>
<td>4,114,200%</td>
</tr>
<tr>
<td>50</td>
<td>0:00:00.40</td>
<td>0:00:00.20</td>
<td>0:00:02.10</td>
<td>0:00:00.40</td>
<td>8:41:03.76</td>
<td>0:00:01.60</td>
</tr>
<tr>
<td>GA Gain</td>
<td>100%</td>
<td>425%</td>
<td>1,953,885%</td>
<td>1,953,885%</td>
<td>1,953,885%</td>
<td>1,953,885%</td>
</tr>
</tbody>
</table>

Note the increased time requirements of both the GA and the random approach, as the problem size (dimension and number of hard locations) increases, which was certainly expected. Apparently, for large dimensions, random initialization can only match GA, if the space is too sparse. This explains why existing SDM applications were able to manage with the random initialization so far. But, by using our approach to remove this limitation, we can deploy SDM with small dimensions, and relatively large number of evenly distributed hard locations.

5 Discussion

The advantage of our approach is immediately obvious. To our knowledge, this is the first use of GA for such a search problem. The random initialization approach was the one used in most SDM implementations (Anwar 1997; Hely et al 1997; Hong and Chen 1991; Rao and Fuentes 1996; 1998; Surkan 1992). Also it was the original method suggested by Kanerva (Kanerva 1988a), and discussed elsewhere (Denning 1989; Franklin 1995; Willshaw 1990). A few neural network implementations of SDM were also used (Hely et al 1997; Joglekar 1989; Scott et al 1993), with random initialization of link weights with small values, followed by incremental correction using standard Backpropagation (Haykins 1994). Practical implementations of SDM are not rare. They vary from traditional implementations (Anwar 1997; Manevitz and Zemach 1997; Rogers 1989b), to parallel tree-structure realization (Hamalainen et al 1997), to connection machine implementation (Rogers 1988a; 1989a; Turk 1995; Turk and Gorz 1994), or to neural network simulation (Scott et al 1993). Keeler made a comparison between SDM and Hopfield neural network (Keeler 1988). Jaeckel also proposed a design variation for SDM (Jaeckel 1989a; b).

6 Future Research

The use of Parallel Genetic Algorithms (PGA) or Structured Genetic Algorithms (sGA) to divide the search space more efficiently, with migration between subspaces, will be considered and evaluated. The use of a multi-modal niching GA to evolve independent sub-populations out of each a candidate is picked for hard location assignment, is currently under investigation. Also, more tuning of the current GA parameters may well help obtain even better results. To be noted here, is that the straightforward mathematical algorithm to allocate arbitrary number of hard locations in a binary space with large dimension, while observing the maximum hamming distance constraint, has exponential time complexity. An SDM augmented with GA for the assignment of the addresses as illustrated here, or to reduce memory requirement and to assign addresses based on the available inputs (Fan, and Wang 1997), may well be a key to a better understanding and simulation of the human mind.

7 Conclusions

GA significantly outperforms the random initialization of Sparse Distributed Memory in almost all cases. The advance is more apparent when all the hard locations are required to have a minimum fitness that is close enough to the target mean fitness.

We emphasize here that we are dealing with group fitness, not individual fitness. In other words, the entire generation is considered as a whole when measuring the fitness. When we reach the stopping criterion, the entire generation is considered as a whole. We would like to point out that this approach of group fitness, which is similar to the Michigan approach, is indeed different from the common practice in GA, or the Pittsburgh approach.
The use of GA for allocation of hard locations in SDM proves to be very efficient and far more superior to currently used techniques.

Bibliography


Bogner, Myles, Ramamurthy, Uma, and Franklin, Stan (In Press) Consciousness and Conceptual Learning in a Socially Situated Agent.


Chou, P. (1989) The Capacity of the Kanerva Associative Memory is Exponential. Stanford University, CA, USA.


Keeler, J. (1987) Capacity for patterns and sequences in Kanerva's SDM as compared to other associative memory models. RIACS-TR-87.29, NASA Ames Research Center, Moffett Field, CA, USA.


