Triadic Memory — A Fundamental Algorithm for Cognitive Computing

Peter Overmann

Email: ai@peterovermann.com

Posted online on 02 April 2022; revised 20 April 2022.

How does the brain store and compute with cognitive information? In this research report, I revisit Kanerva's Sparse Distributed Memory theory and present a neurobiologically plausible implementation, founded on combinatorial connectivity patterns of large neural assemblies. This type of neural network gives rise to a new efficient algorithm for content-addressable memory. Extending this new kind of memory to store crossassociations (i.e. triples of items) leads to the discovery of the triadic memory algorithm, which turns out to be a powerful building block for cognitive computing and possibly the long sought-for universal algorithm of human intelligence. As proof of concept, I develop a semantic triplestore, analogy-finding operators, autoencoders for sparse hypervectors, and a basic temporal memory. Software implementations are included in this report for reference.

Sparse distributed memory

Associative memory as a research subject has been spanning half a century and the fields of cognitive psychology, neuroscience, artificial neural networks, and computer science. There is empirical evidence that the brain recalls memories by searching through a network of data items which are connected by associations. An associative memory uses contextual input cues, which can be incomplete or noisy, to retrieve and reconstruct previously stored information, often in a sequential or recursive manner.

A key feature of associative memory is the universality between the stored information and the storage addresses. In technical applications, the concept of associative memory has found practical use in the form of content-addressable computer memory systems.

In his seminal work on Sparse Distributed Memory, Kanerva (1988) postulated that the brain represents cognitive concepts as points in a hyperdimensional space. The dimensionality of this space would be in thousands. Long binary vectors (or bit strings) serve as the fundamental data structure for storing and processing pieces of information in this mathematical framework.

At any moment, only a relatively small subset of the brain's neurons are active. This observation leads us to the working assumption that binary vectors representing cognitive patterns are sparsely populated. We also observe sparsity in the time domain: Compared to a computer's clock, the brain operates at low speed. Sparse coding is the basis for storage and processing efficiency. Computing with sparse binary hypervectors has mathematical properties that are well suited for modeling cognitive processes. For example, superimposing two mental concepts would be analogous to the union (logical *OR*) of two sparse hypervectors. The Hamming distance between two such vectors serves as measure for the distance between the corresponding concepts in our mind.

Associative memory systems are content-addressable: The same kind of data object (sparse binary hypervectors in the case of SDM) is universally used as memory address as well as the pattern to be stored. For a heteroassociation $x \to y$, a pattern y is stored at a memory address x. For an autoassociation $x \to x$, the same pattern x is used as address and as content.

Information storage is distributed across many locations, and each of those locations can hold pieces of multiple patterns. Thus a distributed memory is robust to a partial loss of storage locations and transmission noise.

Like the human brain, an associative memory system can retrieve memories from incomplete or noisy input cues. Starting from a noisy input address, a less noisy version of the stored pattern is recalled. Using this in turn as a memory address gives an even cleaner pattern. This way, a sequence of recall steps will incrementally remove noise and converge to the originally stored pattern.

Storing a large number of patterns in a distributed manner requires a similarily large number of memory locations, arranged in intermediate units between the input and the output layers. In neural network terminology this is a hidden layer. In a neurobiological realization, those hidden units should actually be quite visible, and likely consist of a very common neuron type.

How does information from the input layer get distributed over hidden units? Kanerva hypothesized that there's a large set of fixed address vectors, randomly distributed over the space of possible input vectors, each pointing to a hard storage location. In order to store or recall a pattern, SDM calculates the Hamming distance of the input address to those fixed address vectors and selects some nearest neighbors as addresses for distributed storage. This mathematically elegant concept utilizes the binomial statistical distribution of distances between random points in the space of sparse binary hypervectors.

While the original SDM design exhibits the features one would desire from an associative memory, it has the shortcoming that its addressing mechanism requires massive computation, which also has no immediate biological equivalent. Jaeckel (1989) proposed a class of memory designs with an improved algorithmic complexity.

In the following, I present a neural memory that realizes the desired features and properties of SDM using an efficient and biologically plausible addressing mechanism.

Combinatorial neural networks

In order to construct a neural associative memory, we need to look beyond the cellular scale at which individual neurons interact with each other. An individual neuron can perform only very basic computations. Complex properties of a memory system will emerge at a large scale, in an assembly of possibly millions of neurons.

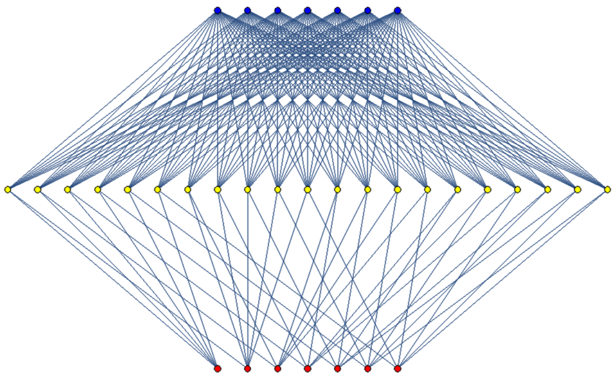
At the scale of a large neural networks, the information flow can be modelled, in a very good approximation, as binary signals. In this sense, nature has found a way to emulate digital computation.

Our challenge is now to design a network structure which exhibits the features of an associative memory. How would we connect millions of neurons in a systematic way to store a similarly large number of sparse binary hypervectors?

Specifically, we want to store associations $x \rightarrow y$ where x and y are sparse binary vectors of dimension n. It's a well-understood fact that a two-layer neural network with an $n \times n$ connection matrix has a very limited memory capacity. Information storage needs to be distributed across a large intermediate layer. So our approach is to construct a three-layer neural network that functions as a pattern associator.

The solution is surprisingly simple: To track all possible correlations among the active bits of the input pattern, there needs to be one hidden unit for every possible pairwise combination of input bits. With an input dimension n, the hidden layer then consists of $n(n\text{-}1)/2$ units, each permanently connected to a unique pair of inputs. Each hidden node has a variable connection to every output bit. Those variable connections are the storage locations of the associative memory.

The following graph illustrates this network topology for $n = 7$, with the combinatorically connected input nodes shown at the bottom, 21 hidden nodes at the center, and 7 fully connected output nodes at the top:



From this generic, fully connected network described above one can quickly derive variants by introducing some kind of topology in the input or output layers. For example, with input bits arranged in a two-dimensional grid, only pairs of nodes that lie within the radius of a receptive field would activate hidden nodes. Generally, such variants have a smaller hidden layer and therefore a lower information capacity.

Is a combinatorial neural network design a plausible model for how the brain stores long-term information? Marr (1969) and Albus (1971) independently developed the notion of the cerebellum as an associative memory, describing the connectivity and interactions of the different neuron types observed in the cerebellar cortex. Kanerva (1992) pointed out the close analogies between the Marr–Albus theory and SDM.

Our three-layer combinatorial network resembles the structure of the cerebellar cortex in the following way: The input layer models mossy fibers, which have pairwise forward connections to granule cells representing the intermediate layer. (In Marr's original model, granule cells are connected to up to five input fibers.) The output layer would be the equivalent of Purkinje cells which receive input, through their huge number of synapses, from the intermediate layer.

In order to test the hypothesis of combinatorial connectivity, one would need to look for substructures in the cerebellum where a number of input fibers connects to approximately the squared number of granule cells, and where excitation patterns indicate pairwise connectivity.

An efficient associative memory algorithm

Doing a multitude of experiments with different learning algorithms verfied that a combinatorial neural network naturally behaves like an associative memory, regardless of the specific mechanism. Whether one uses physiologically-informed synaptic activation models or discrete operations optimized for computer systems, a sparse distributed memory will emerge from the network's topology. With the goal in mind to provide efficient software components for cognitive computing, I'll present the most minimalistic learning algorithm that yields the desired memory features:

We begin with a blank slate memory where all storage locations are initialized to zero. To store a heteroassociation $x \to y$ in memory, we determine the set of hidden nodes that get activated by pairs of active input nodes, i.e the non-zero bits in x. Then we simply increment the connection weights between active hidden nodes and active output nodes (the non-zero bits in y). Conversely, decrementing the same counters would delete a previously stored association from memory.

Recalling a stored pattern is equally straightforward: Given an input address x, we again find the set of activated hidden nodes and calculate, for each possible output node, the sum of the respective counters. The resulting n-dimensional scoring vector v indicates the probability for each output node to be part of the originally stored pattern.

A binary output vector y with target sparse population p is reconstructed as follows: We find the p^{th} largest component t in v and construct the binary vector $y_i = [\, v_i \,]_p$ so that $y_i = 1$ if $v_i \geq \max(1,t)$, and $y_i = 0$ otherwise.

A complete reference implementation is included below. *Mathematica* (a language I helped develop) was used for discovering and exploring this algorithm:

```
HeteroAssociativeMemory[f_Symbol, {n_Integer, p_Integer}] := Module[ {T, connections},
   (* memory initialization *)
   T[_] = Table[0, {n}];

   (* connections from input to hidden layer  *)
   connections[x_SparseArray] := Module[ {k = Sort[Flatten[x["NonzeroPositions"]]]},
     Flatten[Table[ {k[[i]], k[[j]]}, {i, 1, Length[k] - 1}, {j, i + 1, Length[k]}], 1] ];

   (* store x→y *)
   f[x_SparseArray, y_SparseArray] :=
    Module[{h = Normal[y]}, (T[#] += h) & /@ connections[x];];

   (* recall y, given an address x *)
   f[x_SparseArray] := Module[ {v, t },
     t = Max[1, RankedMax[v = Plus @@ (T /@ connections[x]), p]];
     SparseArray[Boole[# ≥ t] & /@ v] ];

   (* random vector with dimension n and sparse population p *)
   f[] := SparseArray[RandomSample[Range[n], p] → Table[1, {p}], {n}];
  ];
```

Several notes about this algorithm are in order:

The time complexity of storing or recalling patterns with dimension n and a sparse population p is $O(n\, p^2)$, so the performance as a memory system is more efficient for sparser vectors.

There are no hidden parameters that may require tuning to particular tasks.

Each pattern is learnt in one shot – no multiple training iterations are needed. Different associations can be added to memory in any order, always yielding the same memory state.

The core of the algorithm is built mainly upon vector additions. It neither requires the expensive matrix multiplications typically found in neural-network-based learning methods, nor the kind of sequential search methods commonly used in content-addressable memory systems. The computing performance of this algorithm is bound by the computer system's memory access speed rather than its arithmetic performance.

Using the reference implementation configured with $n = 1{,}000$ and $p = 10$ (i.e. a sparsity of one percent) running on a small desktop computer, we can store or retrieve about 6,000 patterns per second. The memory capacity in this configuration is about one million random associations.

The memory's noise tolerance depends on the number of stored patterns. When filled with 100,000 random patterns (at $n = 1{,}000$ and $p = 10$), all stored information can be perfectly retrieved even if half of the address bits are randomly removed, or up to 15 address bits are randomly added. As progressively more patterns are stored in memory, noise tolerance decreases.

When exposed to a number of random autoassociations $x \to x$, the algorithm quickly learns an identity mapping and becomes useless. For building a practical auto-associative memory, we can work around this behavior by inserting random vectors w and storing chains of the form $x \to w \to x$. This setup exhibits the features we would expect from an auto-associative SDM, in particular the ability to clean up noisy input.

Summarizing what we've found so far: There is an unexpectedly simple and computationally highly efficient implementation of SDM, based on a combinatorial neural network which shows similarities to structures observed in the cerebellar cortex.

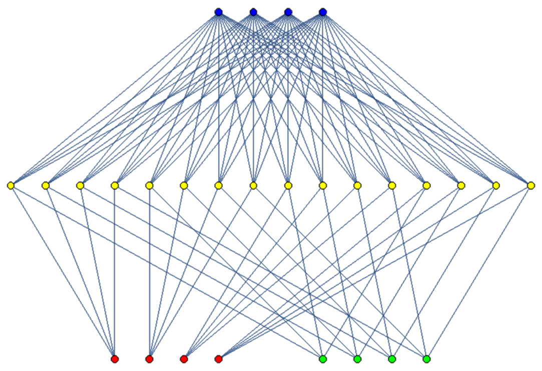
While an auto-associative or hetero-associative SDM is a useful tool for storing items or simple linked lists of items, it's not capable of computing with sparse distributed representations on its own. A universal computing architecture requires algebraic operations between sparse distributed representations, in particular certain equivalents of addition and multiplication. This has been addressed with frameworks such as Vector Symbolic Architectures (VSA, Gayler 2003) and Hyperdimensional Computing (Kanerva 2009), which locate this functionality outside the memory system. In the following, I'll show that the complementary aspects of SDM on one side and VSA or Hyperdimensional Computing on the other side can be subsumed and unified in a new kind of memory architecture.

Triadic memory

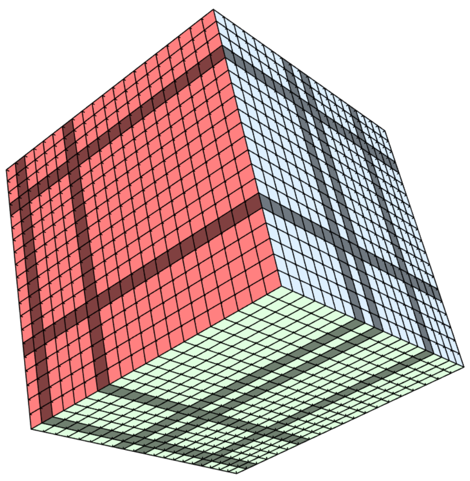
Thus far, we've learnt that there exists a biologically plausible neural implementation of SDM, an associative distributed memory for sparse hypervectors, which is based on a novel kind of combinatorial neural network. The corresponding algorithm works efficiently as a content-addressable hetero-associative memory system. An auto-associative memory can be built on this basis by inserting a layer of intermediate random vectors. Now, having solved this challenge which dates back to the 1980s, it turns out there's something even more interesting to be discovered: A memory that stores crossassociations, the combination of two vectors producing a third vector, has an even simpler architecture than a hetero-associative memory.

Why investigate crossassociations? Associating a pair of items to result in a third item is an ubiquitous function in cognition, neural systems, computer science and in engineering applications such as sensor data fusion. In the context of content-addressable memory systems, a cross-associative memory would be a database that requires two keys to retrieve a value. To my knowledge, no attempt has been made to design such a system.

In the context of hyperdimensional vectors, a crossassociation is a mapping $\{x, y\} \rightarrow z$ from two input vectors to an output vector. Within the framework of our combinatorially connected neural networks, there is a straightforward way to model such a cross-associative memory: Every node in the hidden layer is activated by a different pairing of nodes chosen from the two input vectors. This connectivity pattern is illustrated here for $n = 4$:



The hidden layer consists of n^2 nodes, so there are n^3 storage locations connecting the hidden layer with the output layer. Visualizing those storage elements as a cube, we see that a sparse vector corresponds to a set of p parallel planes in this cube (p being the sparse population, or vector norm).



For a given triple of input and output vectors, each with population p and each representing one of the three axes of the cube, there will be p^3 activated storage locations, corresponding to the elements where three orthogonal planes intersect.

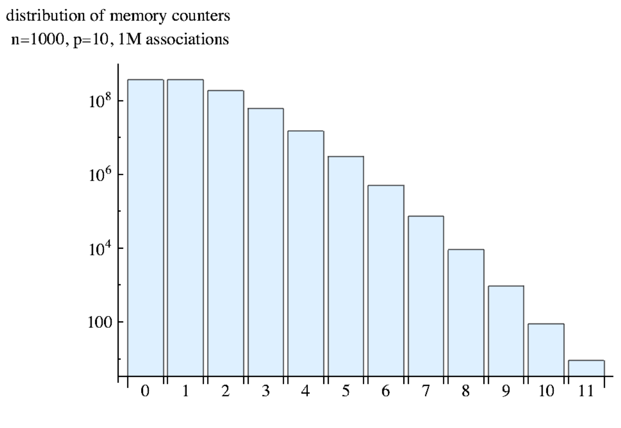
We started with the idea of storing an association $\{x, y\} \to z$ that connects two input vectors to one output vector. But obviously the shape of this memory cube does not distinguish between input and output. Any two of the three primary axes can be used for input, with the third axis taking the role of output. To account for this observation, we'll use the triplet notation *{x,y,z}* for an association between three vectors.

The learning algorithm can be expressed in mathematical notation as follows: The memory cube is a tensor of rank 3 with (high) dimensions $n \times n \times n$, consisting of non-negative integers T_{ijk} with zero initial value. Adding a triple *{x, y, z}* of binary sparse vectors to the memory corresponds, in component notation, to the mapping $T_{ijk} \to T_{ijk} + x_i\, y_j\, z_k$. Deletion from memory is realized with the inverse operation $T_{ijk} \to T_{ijk} - x_i\, y_j\, z_k$.

To recall z for given vectors x and y, we just need to calculate $z_k = [x_i\, y_j\, T_{ijk}]_p$. Similarily, the first and second parts of a triplet are given by $x_i = [y_j\, z_k\, T_{ijk}]_p$ and $y_j = [x_i\, z_k\, T_{ijk}]_p$. This notation implies summation over indices that appear twice in the same term. The operator $[\ldots]_p$, as defined in the previous section, produces a binary vector with target sparse population p.

To understand why and how this works, let's consider a tensor T_{ijk} which is a superposition of various triples, but does not contain the term $x_i\, y_j\, z_k$. Adding this triple updates the this tensor to $T_{ijk} + x_i\, y_j\, z_k$. The query for z_k is then calculated as $[x_i\, y_j\, (T_{ijk} + x_i\, y_j\, z_k)]_p = [x_i\, y_j\, T_{ijk} + x_i\, y_j\, x_i\, y_j\, z_k]_p = [x_i\, y_j\, T_{ijk} + p^2\, z_k]_p \approx [\, p^2\, z_k]_p = z_k$. This is true as long as the average value of the counters T_{ijk} stays below 1. From this we can quickly estimate the memory's capacity: The probability for one component T_{ijk} to be incremented when adding a random triple association is $(p/n)^3$. Therefore the memory can hold and perfectly recall up to $(n/p)^3$ random triples. At a typical sparsity of one percent, the capacity is one million associations — independently of the vector dimension.

The chart below shows the distribution of memory counters T_{ijk} for n = 1,000 and p = 10, after adding one million random associations. Circa 36.78 percent (or 1/e) of the counters remains unused, about the same number of counters has value 1, and the rest shows an exponentially decreasing distribution up to a value of 11. This means in an ideal implementation a 4-bit memory counter would suffice, and the entire memory would occupy half a gigabyte of computer storage.



If we consider each triple vector stored in memory a hyperedge connecting two nodes to a third node, then the entire memory becomes a holistic superposition of directed hypergraphs. Traversing the network by means of the query function defined above is therefore a probabilistic operation, as the outcome of a query is the most likely superposition of nodes that were originally added to the memory. The sparsity constraint effectively cuts off less probable contributions. The combination of a graph structure with probabilistic decision making evidently resembles the concept of a Bayesian network.

What we're discovering here is arguably the simplest possible realization of an SDM, or content-addressable memory in general. The unexpected simplicity of this algorithm emerges as we moved from autoassociations and heteroassociations to crossassociations that involve triples of sparse distributed representations.

In the following, I'll refer to this algorithm as *triadic memory* — borrowing this term from Feldman (1981) who used it in a different context.

A reference implementation in *Mathematica* language is shown below. There are separate functions for recalling the first, second, or third part in a triple association, or triad.

```
TriadicMemory[f_Symbol, {n_Integer, p_Integer} ] := Module[ {T, tup},
  (* initialize memory cube *)
  T = Table[ 0, {n}, {n}, {n}];

  (* memory addresses activated by input vectors *)
  tup[x__] := Tuples[ Flatten[ #["NonzeroPositions"] ] & /@ {x}];

  (* binarize a vector at sparsity target p *)
  f[0] = SparseArray[{0}, {n}];
  f[x_] := Module[{t = RankedMax[x, p]}, SparseArray[Boole[# ≥ t] & /@ x]];

  (* store {x,y,z} *)
  f[x_SparseArray, y_SparseArray, z_SparseArray] := (++ T[[##]] & @@@ tup[x, y, z];);

  (* recall x, y, or z *)
  f[_ , y_, z_] := f[ Plus @@ ( T[[All, #1, #2]] & @@@ tup[y, z])];
  f[x_, _ , z_] := f[ Plus @@ ( T[[#1, All, #2]] & @@@ tup[x, z])];
  f[x_, y_, _] := f[ Plus @@ ( T[[#1, #2, All]] & @@@ tup[x, y])];

  (* random vector with dimension n and sparse population p *)
  f[] := SparseArray[ RandomSample[ Range[n], p] → Table[1, {p}], {n}];
 ]
```

The above program instantiates a named triadic memory and its access functions. For practical tasks, several such memories may be instantiated and combined into circuits (an example will be given later on). A triadic memory `M` provides the atomic operation `M[x,y,z]` for storing a triplet $\{x, y, z\}$, and the three functions `M[_,y,z]`, `M[x,_,z]`, and `M[x,y,_]` for recalling one part (denoted here by the blank character) by specifying the other two parts of the triplet. As already mentioned, we can also implement a function that subtracts a triple from memory.

A triadic memory's capacity is comparable to the capacity of a hetero-associative memory with the same configuration. Chosing a vector dimension $n = 1{,}000$ and sparse population $p = 10$, we can store about a million triples of random vectors and perfectly recall them, even in the presence of a few bits of noise in the input.

In the terminology of Vector Symbolic Architectures, storing a triple $\{x, y, z\}$ binds a value to a variable and assigns the result to a third vector. Which of the three positions we take to be the variable, the value or the result is up to interpretation — any of the six possible permutations can be chosen.

The binding operation is reversible: After binding y to x and linking the result to z, we can extract y by recalling the second position $\{x, \_, z\}$. Also, binding distributes over vector addition: Individually storing $\{x, y, a\}$ and $\{x, y, b\}$ in memory has the same effect as storing $\{x, y, a{+}b\}$, with the sum (or bundle in VSA terminology) of two binary vectors being computed as the elementwise logical OR. With those two properties, reversibility and distribution over addition, the binding operation is effectively a kind of multiplication.

As the triadic memory incorporates bundling and variable binding as intrinsic features, we can now build a foundation for hyperdimensional computing based solely on the memory system — without the need for an external binding operation. In other words, triadic memory merges the concepts of SDM and VSA into a single unified framework.

Semantic triples and analogy

The Resource Description Framework (RDF), a W3C specification for storing and exchanging semantic data on the web, represents semantic facts as triples of the form {*subject, predicate, object*} — a data structure that fits naturally into the triadic memory framework. Implementing an efficient and scalable RDF triplestore has been a well-known challenge. With triadic memory, we've now a content-addressable triplestore at our hands that scales to millions of items, with a constant query time independent of the number of items. In this section, we'll explore how semantic triples can be mapped into a triadic memory.

As a proof of concept, we set up a small movie database involving titles, years, director names and countries:

```
{TG, title, The General}          {MT, title, Modern Times}
{TG, year, 1926}                  {MT, year, 1936}
{TG, director, Keaton}            {MT, director, Chaplin}
{TG, country, USA}                {MT, country, USA}
```

Each data item (subjects, predicates and objects) shall be mapped to a random hyperdimensional vector, which are mutually nearly orthogonal and don't encode any semantic content or similarity. Instead, similarity between items will emerge from their proximity in the semantic graph. The three parts of a triplet are a priori first-class objects: It's an arbitrary choice to assign the first position to represent objects, the second position to represent predicates, and the third to represent objects —we're adopting the RDF convention just for sake of clarity.

Having encoded all data items as random vectors, we can now store them directly in a triadic memory. Because the underlying operation is an addition, it does not matter in which order triples are committed to the memory. As already mentioned, also subtraction is possible atomic operation.

To query the triadic memory, two parts of a triplet need to be given to retrieve the third part. For instance, querying `{MT, _, 1936}` gives `year`. This works even if the input vectors are noisy or superpositions of the originally encoded items. The result of a query may be an approximate version of the precise answer due to crosstalk from unrelated associations, or naturally if the input was unprecise. Chaining a few queries together, using the output of one query as the input to the next, typically improves accuracy step by step, resulting in a clean answer. This is a general and well-understood feature of associative memory systems.

An important feature of triadic memory, which distiguishes it from auto-associative or hetero-associative memories, is the ability to test if an association has been already stored. This property essentially enables unsupervised learning, as we can now build a memory system that learns an item only in the case it has not been seen before. For example, one might want to test if *{x,y,_}* is identical to *z* before deciding to store the triple *{x,y,z}* in memory.

Triadic memory stores a holistic superposition of triples, unlike a traditional database which keeps items as separate records. In the above example, the query `{_, country, USA}` retrieves a sparse hyperdimensional vector that is a superposition of `TG` and `MT`. This bundle can then be used in a subsequent operation to query further information, simultaneously on both items it contains. Conversely, a traditional database would return a table of the two entities `TG` and `MT`, over which a subsequent query would then need to iterate. What's an expensive *join* operation in a conventional triplestore is done in one go in triadic memory.

At this point, we've seen some first evidence that triadic memory stores and processes information in a way that resembles how the mind works. To take this analogy a step further, let's explore how one might construct analogies between semantic facts stored in the memory. Arguably, the ability to think in analogies is at the foundation of cognition. Our mind constantly seeks to find connections and relations between pieces of information. In the framework of triadic memory, this would correspond to a mechanism that finds a connecting path, or mapping, between structures in the semantic graph.

Circling back to our movie database, let's explore how to construct basic analogy queries (see Kanerva (2010) for a similar discussion in the context of VSA). For instance, we could ask what's the analogy to `Keaton` in `MT` — without providing additional cues that finding the answer involves relations with `MT` or the `director` predicate. In order to traverse the semantic graph, we need to know at least what predicates are known about each subject. A simple way to represent such an index (or rudimentary schema) might look like this:

```
{TG, predicate, title}              {MT, predicate, title}
{TG, predicate, year}               {MT, predicate, year}
{TG, predicate, director}           {MT, predicate, director}
{TG, predicate, country}            {MT, predicate, country}
```

We also record what predicates are known for each object:

```
{title, predicate, The General}     {title, predicate, Modern Times}
{year, predicate, 1926}             {year, predicate, 1936}
{director, predicate, Keaton}       {director, predicate, Chaplin}
{country, predicate, USA}
```

The above index can be stored either in the same triadic memory or in a separate memory instance (a hetero-associative memory would equally work for this purpose). Our notation does not distinguish between those possibilities.

It's now straightforward to infer the equivalent of `Keaton` in `MT`: The two-step query `{MT,{_,predicate,Keaton},_}` finds `Chaplin`, which is the expected answer. Generally, the function `{`*sub*`, {_, predicate, `*obj*`}, _}` constructs a direct analogy between *sub* and *obj*.

A slightly more complicated query pattern connects the movie title `Modern Times` with `Keaton`:
`{{_,{_,predicate,Modern Times},Modern Times,{_,predicate,Keaton},_}` resolves to `Chaplin`. This is a query pattern that connects two objects to produce a third object.

As a last example, let's take two objects, `1936` and `USA`, and search a subject they have in common: `{_,{{_,{_,predicate,USA},USA},_,1936},1936}` finds `MT` to be the connecting node.

Autoencoders and unsupervised learning

Many types of autoencoders have been devised for unsupervised learning tasks. An autoencoder converts an input to a compressed code (a "bottleneck"), then decodes this intermediate representation in order to reconstruct a version of the input which retains only its essential features while reducing noise. Typical autoencoder algorithms require training, achieved by backpropagating reconstruction errors.

How would we build an autoencoder for high-dimensional sparse distributed representations? As a starting point, we take a pair $\{a_1, a_2\}$ of sparse hypervectors, each with dimension n and sparse population p, which should be around one percent of n. The pair $\{a_1, a_2\}$ can be considered a vector of dimension $2n$, consisting of two segments of equal length and sparsity, resulting in a total sparse population $2p$. A 2-to-1 autoencoder would map $\{a_1, a_2\}$ to an intermediate code c, then uses c to reconstruct $\{a_1, a_2\}$. This can easily be achieved with triadic memory:

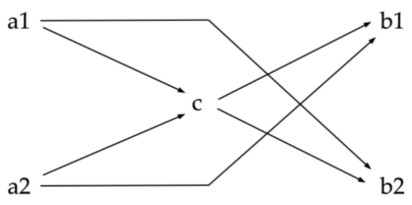
The first step is to query $c = \{a_1, a_2, \_\}$, then we calculate the reverse mappings $b_1 = \{\_,a_2,c\}$ and $b_2 = \{a_1,\_,c\}$. If the combination $\{b_1, b_2\}$ is similar to $\{a_1, a_2\}$, we know this or a similar triad is already stored in memory, and we take $\{b_1, b_2\}$ to be the reconstructed result. Owing to the roundtrip query via the internal code, the pair $\{b_1, b_2\}$ will be a "cleaner" version of $\{a_1, a_2\}$. If $\{b_1, b_2\}$ is dissimilar to $\{a_1, a_2\}$, we generate a random vector c, store the new triple $\{a_1, a_2, c\}$ in memory, and consider the input as result. Similarity is measured by the combined Hamming distance $H(a_1,b_1) + H(a_2,b_2)$. Here we use the fact that two sparse vectors whose Hamming distance is less than $p/2$ can be considered statistically related.

In essence, this algorithm tests if the memory knows an association between a pair of vectors. If an association is found, it calculates a version of the input which is consistent with what is known in memory. Otherwise, a new association with a newly deployed random vector is added to the memory.

The corresponding *Mathematica* program consists of just a few lines, to be inserted in the above `TriadicMemory` code. Of the three possible permutations, here is the function that autoencodes the first two positions of a triple:

```
(* autoencoder for {a1,a2} *)
f[ a1_SparseArray, a2_SparseArray, Automatic ] := Module[ {b1, b2, c},
   z = f[a1, a2, _]; b1 = f[_, a2, c]; b2 = f[a1, _, c];
   If [ HammingDistance[ a1, b1] + HammingDistance[a2, b2] > p ,
    f[a1, a2, c = f[] (* new random vector *) ]; {a1, a2, c}, {b1, b2, c}]];
```

The information flow inside this 2-to-1 autoencoder is sketched below:



Although this algorithm serves the same purpose as traditional autoencoders, there are some noteworthy differences: First, it does not require offline training before deployment. This program continually learns new information in each step while recognizing patterns it has already enountered in a similar shape, and produces output that is close, or even identical, to previously learnt information.

Another difference is the encoding method: When there's new information to be learnt, a new random vector gets generated and assigned to represent it in the memory. This encoding strategy does not depend on the type or meaning of data to be encoded.

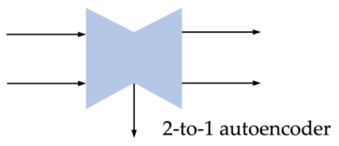
Apart from the vector dimension *n* and the target sparse population *p*, there are no parameters that may require tuning.

This internal code *c* is effectively a higher-level condensed representation of the input pair $\{a_1, a_2\}$, thus creating hierarchy. By layering multiple autoencoders, each realized as a separate instance of a triadic memory, we can now easily build up deep hierarchies with increasingly abstract, and increasingly stable, representations of low-level input.

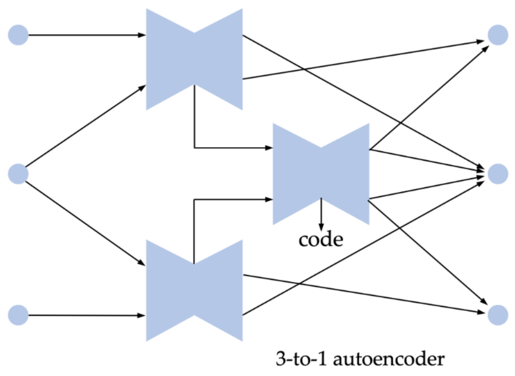
The most radical difference with respect to traditional autocoders is the absence of a Hebbian-style learning mechanism. As descendants of the classic perceptron, today's family of autoencoders is fundamentally based on synaptic learning, that is fitting real-valued network connection weights to achieve a desired outcome. Undoubtedly, such networks have evolved into impressive applications of linear algebra techniques. But it's hard to see how a biological neuron would acquire and persist high-precision synaptic weights, adjust its learning rate to a specific task, or even implement something like the backpropagation of errors.

In a triadic memory, the equivalents of synapses are binary. Learning is based on the deployment of a preexisting random vectors (as we've seen in the autoencoder algorithm), and on counting the number of association patterns in which those random vectors are involved. In this model, learning and memory are separated: Binary synapses of random seed vectors are the locus of memory, wiring and firing together as postulated by Hebb. But the locus of learning is linked to the random vector as a whole, not to the strengths of individual vector elements. My hypothesis is that dendritic segments of pyramidal cells are the neural equivalents of the random seed vectors in triadic memory, and that a dendrite somehow keeps track of the number of associations to which it contributes (Gallistel and King (2009) advocated a theory compatible with this view).

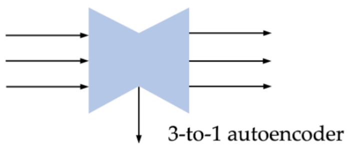
To conclude the discussion of autoencoders, we assemble a 3-to-1 autoencoder which compresses three vectors $\{a_1, a_2, a_3\}$ to an internal code (reducing dimensionality by a factor 3) and gives a reconstructed version of the input. This is our first example for a multi-level circuit based on triadic memory components. The building block to be used here is the 2-to-1 autoencoder introduced above, for which we'll use this circuit diagram symbol:



In a feed-forward circuit diagram, each connecting arrow carries an entire n-dimensional sparse hypervector. With these prerequisites, we can design a 3-to-1 autoencoder as follows:



The three modules used in this circuit must be based on one and the same triadic memory instance in order to properly construct the output. If we're only interested in an encoding of the input, those three modules can be realized as independent memories. A 3-to-1 autoencoder module may be used, for example, to encode three consecutive letters in a word or three consecutive words in a sentence. For such purposes, we would introduce a new circuit diagram symbol which encapsulates the above diagram:

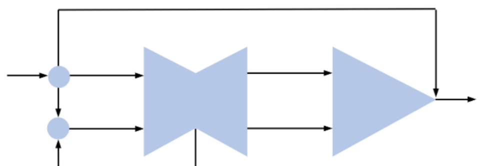


The design of the 3-to-1 autoencoder gives us a first glimpse of what it takes to do cognitive modelling and computing on the foundation of triadic memory. Starting from atomic memory operations, we can create modular building blocks, recursively combine them into higher-level modules, and thus achieve deep neural network architectures for hyperdimensional computing.

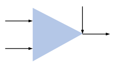
Sequence learning and temporal memory

How would an associative memory learn and replay sequences of hyperdimensional vectors? This is a key question, as the brain has a inherent capability to learn temporal, spatial and all kinds of abstract sequences. A computational model of the brain needs to be able to ingest sequential data patterns, incrementally update its memory as novel information becomes available, and make the most probable prediction for future input at each step. Technical applications of online machine learning include time series prediction, sequence continuation, and anomaly detection.

To represent a sequence, a memory system needs to associate each element with a condensed version of the preceding elements. Such a system incorporates a feedback loop that accumulates the context which is needed to predict the next step. A concrete implementation based on triadic memory is a recurrent neural network with two stages:


temporal memory

Let's turn our attention to the first stage: A 2-to-1 autoencoder combines the current input with the superposition of the previous input and a feedback signal which is the internal code from the previous time step. The superposition is calculated as the element-wise logical OR of the two vectors. So one of the inputs to the autoencoder has twice the regular sparse population, which essentially represents deep context (the depth being determined by the sparse population *p*). The output of the autoencoder, which is a stabilized version of its input, serves as the input for the second stage. The second stage then learns and predicts the outcome based on the context provided by the first stage.

The above diagram makes use of a new circuit component  which is a basic triadic memory module with twofold functionality: Given a triple {*x*,*y*,*z*}, it adds the triple to the memory if the result of the query {*x*,*y*,_} is not identical to *z*. When given an input pair {*x*,*y*}, the output is the result of the query {*x*,*y*,_}. Like the autoencoder introduced earlier, this is a core component to be used in unsupervised learning scenarios. In the temporal memory network shown above it is invoked twice in each time step: First to associate the current input with the output from the first stage, second to make a prediction based on the updated output from the first stage. The two stages must be based on separate triadic memory instances.

Our associative temporal memory processes one long sequence of sparse hypervectors, learning new input vectors and subsequences on the go. When it recognizes a subsequence and makes the correct prediction, it won't update the memory. In cases where context is ambiguous, the prediction can be a superposition of possible outcomes — which is the behavior we should expect from an associative sequence memory.

Computational experiments with this temporal memory show an interesting, almost human-like behavior. It easily learns repetitions of simple patterns such as `ABCD`, or of more complex patterns like `ABCDAB`. When switching to repetitions of a new pattern, say `XXYZ`, it will predict some continuation of the previous pattern for a few steps and then adjust to the new pattern. Long sequences of many distinct items, such as words, are easier to remember than long sequences of few items, such as the decimal digits. To give an example for a difficult sequence: Trying to memorize the first hundred digits of π, after about five iterations the temporal memory accurately predicts most digits.

Hawkins (2004) outlined a hierarchical temporal memory model whose core function is a memory prediction mechanism, located in columnar units of the cerebral cortex. Our temporal memory achieves a similar if not identical functionality, and it may turn out that it is a computationally equivalent — yet considerably simpler — version of Hawkins' framework. Following this path, the consequence would be that a full-scale model of the neocortex can be built from some hundred thousands of triadic memory instances forming a deep hierarchical recurrent network. As the necessary software components are now readily available, the next challenge is to come up with scalable design patterns for building larger cognitive circuits.

Conclusions

The research journey reported in this paper began with revisiting the idea of Sparse Distributed Memory (SDM) and the discovery of a computationally efficient algorithm based on a new kind of combinatorial neural network. This new algorithm realizes a content-addressable memory system for hyperdimensional sparse binary vectors, and may serve as a computational model for aspects of the cerebellar cortex.

Further exploration of this hetero-associative neural memory revealed the existence of a related memory network that stores crossassociations, i.e. relations between triples of items. Somewhat unexpectedly, this triadic memory has a more coherent mathematical formulation than its hetero-associative counterpart, and it offers rich features not found in traditional associative memories:

— Triadic memory stores a holistic superposition of triples of hyperdimensional sparse binary vectors.

— It has the features of an associative content-addressable memory, in particular tolerance to noise.

— It emulates elements of Bayesian networks.

— Triadic memory is based on a new kind of artificial neural network.

— Its learning mechanism resembles dendritic rather than Hebbian synaptic learning.

— It's the simplest possible realization of an SDM.

— It incorporates properties of Vector Symbolic Architecture (VSA).

— The memory lends itself to storing and computing with semantic triples.

— One part of a triple can be recalled by specifying the other two parts, in any direction.

— It can be tested whether an association is already stored in memory, which enables unsupervised learning.

— The vector dimension n and sparse population p are the only configuration parameters.

— The memory's capacity is approximately $(n/p)^3$.

— The underlying cubic memory shape is an $n \times n \times n$ array of small non-negative integers.

— A large class of possible algorithms can be built upon basic triadic memory motifs.

To demonstrate the power of the triadic memory algorithm, I've shown how to build a semantic triplestore, basic analogy operators, autoencoders and a temporal memory based on this new framework. It cannot be emphasized enough that all of the above has been implemented with just a few lines of program code, proving that triadic memory is a feasible foundation for cognitive computing. From this starting point, we can now model increasingly complex cognitive processes, develop machine learning solutions that emulate human cognition, and ultimately aim at building a model of the brain.

If we follow Mountcastle's (1978) idea that all human intelligence is based on just one universal algorithm, then triadic memory should be considered a promising candidate.

Acknowledgements

I am especially obliged to Pentti Kanerva for commenting on earlier versions of this text. I would also like to thank Stephen Wolfram for inspiring the search for simple programs with interesting behavior.

References

J. Albus (1971). *A theory of cerebellar functions*. Mathematical Biosciences 10:25

M. Feldman (1981). *Triadic Memories*. Universal Edition 17326

C. Gallistel, A. King (2009). *Memory and the Computational Brain. Why cognitive science with transform neuroscience*. John Wiley & Sons

R. Gayler (1998). *Multiplicative binding, representation operators, and analogy*. Poster abstract. In: Holyoak, Gentner, Kokinov (editors). *Advances in analogy research*. Sofia: New Bulgarian University

J. Hawkins, S. Blakeslee (2004). *On Intelligence*. Times Books, Henry Holt and Company, New York

L. Jaeckel (1989). *An alternative design for a sparse distributed memory*. RIACS Technical Report 89.28

L. Jaeckel (1989). *A class of designs for a sparse distributed memory*. RIACS Technical Report 89.30

P. Kanerva (1988). *Sparse distributed memory*. MIT Press, Cambridge MA

P. Kanerva (1992). *Sparse Distributed Memory and Related Models*. RIACS Technical Report 92.10

P. Kanerva (2009). *Hyperdimensional Computing: An Introduction to Computing in Distributed Representation with High-Dimensional Random Vectors*. Cognitive Computing 1:139

P. Kanerva (2010). *What We Mean When We Say "What's the Dollar of Mexico?": Prototypes and Mapping in Concept Space.* Quantum Informatics for Cognitive, Social, and Semantic Processes. AAAI Fall Symposium

D. Marr (1969). *A theory of cerebellar cortex*. Journal of Physiology 202:437

V.B. Mountcastle (1978). *An Organizing Principle for Cerebral Function: The Unit Model and the Distributed System.* In: Edelman, Mountcastle (editors). *The Mindful Brain.* MIT Press

Companion software is available on GitHub at https://github.com/PeterOvermann.