Getting Started: Dyadic Memory

Peter Overmann

20 June 2022

The Dyadic Memory algorithm described in this Mathematica notebook is a sparse distributed memory (SDM) for storing heteroassociations x->y, where x and y are sparse distributed representations (SDR).

SDRs are sparse binary hypervectors of dimension n which is typically greater than 1000, and a sparse population p which is typically 1 to 3 percent of n.

In the implementation presented here, x and y may have different dimensions and sparsity.

The concept of an SDM was introduced by Pentti Kanerva in 1988. The present algorithm, discovered in 2021 and published in the paper Triadic Memory, has the functionality of the original SDM, but is implemented in an entirely different way, based on a biologically plausible combinatorial neural networks.

Asymmetric Dyadic Memory

Hetero-associative memory for storing associations x->y, where x and y are sparse distributed representations (SDRs) of dimensions n1 and n2, respectively.
No assumption is made about the sparse populations of x and y when storing an association.
When recalling y for a given SDR x, the result will have the target sparse population p2.

```
DyadicMemory[f_Symbol, {n1_Integer, p1_Integer}, {n2_Integer, p2_Integer}] :=
  Module[ {W, connections},

   (* memory initialization (using sparse data if n2 >= 2000 *)
   W[_] = If[ n2 < 2000, Table[0, n2], SparseArray[ _ → 0, n2]];

   (* random association x→y *)
   f[] := SparseArray[ RandomSample[ Range[n1], p1] → Table[1, p1], {n1}] ->
     SparseArray[ RandomSample[ Range[n2], p2] → Table[1, p2], {n2}];

   (* 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] := ((W[#] += y) & /@ connections[x];);

   (* recall y, given an address x *)
   f[ SparseArray[ _ → 0, n1]] = SparseArray[_ → 0, n2];

   f[x_SparseArray] := Module[ {v, t },
     t = Max[1, RankedMax[v = Plus @@ (W /@ connections[x]), p2]];
     SparseArray[Boole[# ≥ t] & /@ v]
    ]

   (* implementation
    note: connection counters are addressed via the downvalues of W,
   simulating a sparse data structure *)
  ];
```

Noise

Adding salt or pepper noise to an SDR.

```
SDRNoise[x_SparseArray, bits_Integer] := Module[ {p},

  If[bits ≥ 0,
   (* salt noise, adding bits *)
   p = Union[Flatten[x["NonzeroPositions"]],
     Table[ RandomInteger[{1, Length[x]}], bits]];
   SparseArray[ p → Table[1, Length[p]], {Length[x]} ],

   (* pepper noise, removing bits *)
   p = Most[ArrayRules[x]];
   SparseArray[ RandomSample[ p,  Length[p] + bits], Length[x]]
  ]
 ]
```

Visualization

Plot an SDR as a square image, padding with zeros if necessary.

```
SDRPlot[ x_SparseArray ] :=
 Module[ {w, d},
  w = Ceiling[Sqrt[Length[x]]];
  d = Partition[PadRight[Normal[x], w^2], {w}] /.
    {1 → {0.04, 0.18, 0.42}, 0 → {0.79, 0.86, 1.0}};
  Image[d, ImageSize → 2 * w]
 ]
```

Initialize a memory instance

```
n = 1000;
p = 10;
```

```
DyadicMemory[ M, {n, p}, {n, p}];
```

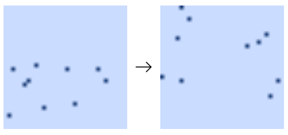
Create a random association and store it in memory

```
r = M[]
```

```
SparseArray[ Specified elements: 10
             Dimensions: {1000} ] → SparseArray[ Specified elements: 10
                                                 Dimensions: {1000} ]
```

```
x = First[r];
y = Last[r];
```

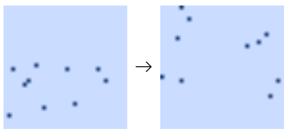
```
SDRPlot /@ (x → y)
```



```
M[x → y]
```

Recall the stored value

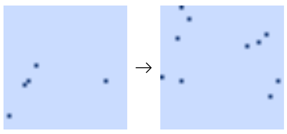
```
SDRPlot /@ (x → M[x] )
```



Recall the stored value from noisy input

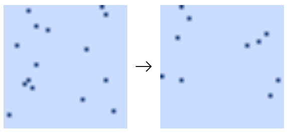
```
x = SDRNoise[x, - p / 2]; (* remove half of the "on" bits *)
```

```
SDRPlot /@ (x → M[x] )
```



```
x = SDRNoise[x, p]; (* add p random bits *)
```

```
SDRPlot /@ (x → M[x] )
```



Capacity testing: Store k random associations

For n1 = n2 = 1000 and p1 = p2 = 10, the memory can store and perfectly recall about 500k random associations.

```
k = 500 000;
```

```
data = Table[ M[], k];
```

```
M /@ data; // AbsoluteTiming
```

```
{152.912, Null}
```

Recall stored values and compute Hamming distances to expected outputs

```
out = HammingDistance[ M[ First[#]], Last[#]] & /@ data; // AbsoluteTiming
```

```
{143.626, Null}
```

```
Sort[Tally[out]]
```

```
{{0, 499 998}, {1, 1}, {2, 1}}
```