Monadic Memory - Getting Started

Peter Overmann
31 Jul 2022

Monadic Memory is a new auto-associative memory for binary Sparse Distributed Representations (SDRs).
It can be used for clean up, clustering or pooling of a sequence of tokens.

This algorithm takes an SDR x as input and searches the memory for a similar (as measured by Hamming distance) SDR. If a similar SDR has been stored before, that earlier version is returned, otherwise the result is taken to be x.
Therefore the algorithm always returns a value similar to the input,

The implementation combines two mirrored Dyadic Memory instances which share a common hidden layer comprised of random SDRs r, storing x->r in the one memory and r->x in the other memory. As expected from an associative memory, a roundtrip x->r->x' produces a "cleaner" version x' of x.

Monadic Memory has the same capacity as Dyadic Memory. For typical values n=1000 and p=10, it can hold around 500k random patterns.

Monadic Memory Algorithm

```
MonadicMemory[f_Symbol, {n_Integer, p_Integer}] :=
  Module[ {D1, D2, items = 0},

   DyadicMemory[ D1, {n, p}];
   DyadicMemory[ D2, {n, p}];

   (* random SDR *)
   f[] := SparseArray[ RandomSample[ Range[n], p] → Table[1, p], {n}];

   (* store and recall x *)
   f[x_SparseArray] := Module[ {r, hidden},

     r  = D2[D1[D2[D1[x]]]]; (* two roundtrips *)

     If[ HammingDistance[x, r] < p / 2 , Return[r]];

     items++;
     hidden = f[];
     D1[ x → hidden]; D2[ hidden → x];

     x (* return input value *)
    ];

   f["Items"] := items;
  ];
```

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]
 ]
```

Configuration

```
Get[ $UserBaseDirectory <> "/TriadicMemory/dyadicmemoryC.m"]
```

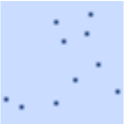
```
(* use Mathematica code if the C command line tool is unavailable:
  Get[  $UserBaseDirectory <> "/TriadicMemory/dyadicmemory.m"]
*)
```

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

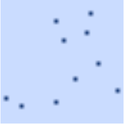
```
MonadicMemory[ M, {n, p}];
```

Store and recall a random SDR

```
SDRPlot [x = M[]]
```



```
SDRPlot[M[x]]
```



Recall the stored value from noisy input

```
SDRPlot[M[SDRNoise[x, 1 - p / 2]]] (* remove bits -- pepper noise *)
```

```
SDRPlot [M[SDRNoise[x, p / 2 - 2]] ] (* add bits -- salt noise *)
```

```
M["Items"]
```

1

Capacity testing: Store random tokens

For n = 1000 and p = 10, the algorithm can store about 500k random tokens.

```
k = 500 000;
```

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

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

{816.594, Null}

```
M["Items"]
```

499 998

Recall stored patterns and calculate retrieval accuracy

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

{636.609, Null}

The number of stored items has not (significantly) increased, while the algorithm keeps learning during recall.

```
M["Items"]
```

500 001

Most tokens were perfectly recalled, a few have small errors.

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

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

Store and recall a random SDR

```
SDRPlot [x = M[]]
```

```
SDRPlot[M[x]]
```

Recall the stored value from noisy input

```
SDRPlot[M[SDRNoise[x, 1 - p / 2]]](* remove bits -- pepper noise *)
```

```
SDRPlot [M[SDRNoise[x, p / 2 - 2]] ] (* add bits -- salt noise *)
```