ML 19/20-5.10 Validate Memorizing Capabilities of Spatial Pooler

Dipanjan Saha, Pradosh Kumar Panda, Rina Yadav
dipanjan.saha@stud.fra-uas.de, ppanda@stud.fra-uas.de, rina.yadav@stud.fra-uas.de

***Abstract*— The main objective of the project is to describe memorizing capabilites as the ability to recall last impression of input sequence. It utilizes Hierarchical temporal memory (HTM) to generate SDR's. We have described how input vectors are generated for different learning sequences and shown output for the learning sequences using encoder. We have discussed different test cases to verify the memorizing capabilities of the Spatial Pooler using different learning sequences. To check the similarities of this sequences we have used hamming distance and overlap plotting.**

Keywords: HTM, memorizing capabilities, spatial pooler, Hamming Distance

I. Introduction:

The Spatial Pooler algorithm is part of the emerging HTM (Hierarchical Temporal Memory) technology, a biology-inspired computational model designed to research intelligence[1]. HTM programming uses SDRs (Sparse Distribution Representation) as the main data structures. Building and controlling these SDRs is the core challenge of science. Spatial Pooler Algorithm converts an sequence of bits into an SDR representation. To order to achieve a better view of knowledge, these SDRs take influence from their neuroscience predecessors – the human brain neocortex – because it is a series of excited columns on the brain membrane surface.

In order to achieve this representation, the Spatial Pooler transforms all inputs feed into its SDR. Such SDRs may be perceived to be the system's interpretation of the data being fed. In general, the application of the Spatial Pooler for new patterns involves the alternation of the permanence (remaining unchanged indefinitely) values by Hebbian laws. Under this law, the permeance values of the active SP column from the feed inputs are strengthened while the inactive columns are "punished"[2]. The reuse of all mini columns in the space pooler by the HTM region enhances the potential ability to froget, as learning progresses, that the original dataset would be lost in representation [2], to provide space for more recent inputs. The property in this work is defines as Concentration of Congintion. Cognitive capacity is the total amount of information the brain is capable of retaining at any particular moment. This amount is finite, so we can say our total capacity is only ever 100%. How much of one's cognitive capacity is being used towards a particular task at any given time is called the cognitive load. Doing activities that are habitual does not create a heavy cognitive load, so several tasks can be done at once.
This work focused on and isolating pure symbols learning patterns from sequence learning, including temporal memory. This implies that each pattern is unique and, aside from their value, there is no sequential relation with time frames.This work is formulated in six tests in order to understand the data collection, to verify the memorizing capacity of the Spatial Pooler.

II. Methodology:

In this context, memorizing capabilities are described as the ability to remember the last memory of a learning sequence. In the case of software implementation, this ensures that after learning has finished, the spatial pooler will remember the reported SDR of one value to be the same as the newly calculated SDR of that value. If these two SDRs are the same or show a strong degree of resemblance, it can be inferred that the memory of the input value is retained. All the test cases were made by keeping major factors in mind which are Campare Number, Gap and Overlap Bitmap. The explaniation is as follows:

1. **Compare Number**: It is specifically used in this work which refers to the vector under observation. That is, we are taking one vector for observation in a whole learning. The idea is to see the differences in the outputs for the vector.
2. **Overlap Bitmap**: It is a bitmap which shows the comparison between two outputs. As mentioned before, we have the outputs of a specific vector in the learning. That is, if there are 4 sequences of that specific vector is present in the learning, then we would have three overlap bitmaps, which represent the comparisons between two consecutive sequences. These bitmaps are showed in the document in an orderly manner. If we have taken 1.0 as the Compare Number, then we will record the output SDRs for this vector that we have got in the learning. And we would compare those outputs, by overlapping and finding the hamming distance.
3. **Gap**: It is the number of vectors or sequences that are there between two sequences of the vector under observation or compare number.

III. Implementation and Code

i. ***Generating Input Vectors***:

The input parameters for the input plane those have been taken are as follows:

```
var parameters = GetDefaultParams();
parameters.Set(KEY.POTENTIAL_RADIUS, 64 * 64);
parameters.Set(KEY.POTENTIAL_PCT, 1.0);
parameters.Set(KEY.GLOBAL_INHIBITION, false);
parameters.Set(KEY.STIMULUS_THRESHOLD, 0.5);
parameters.Set(KEY.INHIBITION_RADIUS,
(int)0.25 * 64 * 64);
```

```
parameters.Set(KEY.LOCAL_AREA_DENSITY, -1);
parameters.Set(KEY.NUM_ACTIVE_COLUMNS_PER_INH_
AREA, 0.1 * 64 * 64);
parameters.Set(KEY.DUTY_CYCLE_PERIOD,
1000000);
parameters.Set(KEY.MAX_BOOST, 5);
Console.WriteLine("Fetched all default
parameters\n");

parameters.setInputDimensions(new int[] { 32,
32 });
parameters.setColumnDimensions(new int[] { 64,
64 });
parameters.setNumActiveColumnsPerInhArea(0.02
* 64 * 64);
```

1. **Number of active columns**: The number of columns which can be active in a specific area
2. **Column Dimensions**: The number of columns which would be taken in the input plane on which the input vectors will be plotted

The input vectors are generated one by one for each input sequence provided in the test case. The method '**GetEncodedSequence**' in used to get the input vector plotted in the input directory as png files.

```
var inputVectors =
GetEncodedSequence(inputSequence, minVal,
inputs.getMaxIndex(), inputs, inFolder);
```

The '**inputSequence**' is the input integer array which is the sequence, '**minVal**' is the minimum value of the sequence, '**inputs.getMaxIndex()**' gets you the maximum value in the sequence and the 'inputs' is the object of the class '**InputParameters**' which contains all the test inputs

ii. GetEncodedSequence:

A new object of the class 'ScalarEncoder' is declared which exists in the 'NewCortexApi'. It is provided with a dictionary of input parameters which are provided while calling method

```
ScalarEncoder encoder = new ScalarEncoder(new
Dictionary<string, object>()
{
{ "W", inputs.getWidth()},
{ "N", 1024},
{ "Radius", inputs.getRadius()},
{ "MinVal", min},
{ "MaxVal", max},
{ "Periodic", false},
{ "Name", "scalar"},
{ "ClipInput", false},
});
```

For each and every element (starting index) in the input sequence, one encoding is done using the 'Encode()' method of 'ScalarEncoder' class and the result is appended in a list called 'sdrList'. This list is returned as the return parameter of the method and is stored in a variable 'inputVectors'.

```
foreach (var i in inputSequence)
{
var result = encoder.Encode(i);
sdrList.Add(result);
int[,] twoDimenArray =
ArrayUtils.Make2DArray<int>(result,
(int)Math.Sqrt(result.Length),
(int)Math.Sqrt(result.Length));
var twoDimArray =
ArrayUtils.Transpose(twoDimenArray);
int counter = 0;
if (File.Exists(outFolder + @"\\" + i +
@".png"))
{
counter = 1;
while (File.Exists(outFolder + @"\\" + i + @"-
" + counter.ToString() + @".png"))
{
counter++;
}}
```

For each element in the sequence, one vector is plotted by converting the result into a 2D array and providing it to the method '**DrawBitMap()**' in the '**NeoCortexUtils**' class. The colors for the active and the inactive cells are provided and also the directory where the bitmap file will be generated

```
if (counter == 0)
NeoCortexUtils.DrawBitmap(twoDimArray, 1024,
1024, $"{outFolder}\\{i}.png", Color.Yellow,
Color.Black, text: i.ToString());
else
NeoCortexUtils.DrawBitmap(twoDimArray, 1024,
1024, $"{outFolder}\\{i}-{counter}.png",
Color.Yellow, Color.Black, text:
i.ToString());
```

iii. Generating the Output:

After getting the '**inputVectors**', they are iterated by a 'for' loop and outputs are generated for each and every vector in the sequence. It is done using the '**Compute**' method in the '**SpatialPoolerMT**' class in the '**NeoCortex**' project.

```
for (int j = 0; j < activeArray.Length; j++)
{
 if (activeArray[j] > max)
 max = activeArray[j];
}
```

Then the number of rows and columns in the output plane is calculated by fetching the maximum value in the 'activeArray' which is fetched from the '**Compute**' method.

```
var activeArray = sp.Compute(inputVectors[i],
true) as int[];
```

This is done in order to get a proper aspect ratio in the output plane. A new 2D array is created to plot the output vector. In the 'activeArray', where ever the active column is found, '1' is assigned and the rest is assigned with '0'. Then the same method as before to draw the bitmap

‘DrawBitMap()’ is called by providing the 2D array, output directory and the respective colors for active and inactive colors (black and green in this case)

```
NeoCortexUtils.DrawBitmap(out2dimArray,  1024,
1024,
"{outFolder}\\{outFolderCount}\\{count}.png",
Color.Black,        Color.Green,        text:
inputSequence[i].ToString());
```

iv. *Overlap Plotting and Hamming Distance plotting:*

While providing the test cases, an input parameter, ‘**CompareNumber**’ is provided. It’s the element in the sequence which we want to study. And all of the comparisons will be done for this value only

```
inputs.setCompareNumber(1.0);
```

A new 2D array ‘record2Darray’ is created which will contain the previous output 2D vector for the element in the sequence under experiment. First, it’s checked whether the input sequence for which the processing is going on is the ‘**compareIndex**’ or not, Then only we can proceed with the comparision

```
if (inputSequence[i] == compareIndex)
{
if (recordOutput != null)
{
hammingDistance =
MathHelpers.GetHammingDistance(recordOutput,
comparingArray, false);
record2Darray = new int[recordOutput.Length,
recordOutput.Length];
for (int j = 0; j < recordOutput.Length; j++)
{
for (int k = 0; k < recordOutput.Length; k++)
record2Darray[j, k] =
Convert.ToInt32(recordOutput[j][k]);
}}
```

If it’s the same element, then output is recorded in the ‘**record2Darray**’ which can be compared with later on if the same element ever reoccurs. At the same time the ‘**hammingDistance**’ is obtained by calling the ‘**GetHammingDistance()**’ method in the ‘MathHelpers’

```
for (int j = 0; j < rowHam; j++)
{
for (int k = 0; k < rowHam; k++)
{
if (limit < hammingDistance.Length)
{
hammingArray[j, k] =
Convert.ToInt32(hammingDistance[limit]);
limit++;}}}
```

The hamming distance is then converted into a 2D array, ‘hammingArray’ to plot it. Two different methods are called to plot the Hamming Distance and the Overlap bitmaps. (‘DrawBitmapHamming’ and ‘DrawBitmapOverlap’).

v. *DrawBitmapHamming:*

The 2D array provided to this method contains the Hamming Distances in percentage. Depending on the percentages the colors are provided. The color varies from ‘Black’ to ‘White’. If the hamming distance is 100%, then it’s ‘**White**’, if 0%, then ‘**Black**’. And the variation is in every 10%.

```
else if (twoDimArray[Xcount, Ycount] >= 90)
{
myBitmap.SetPixel(Xcount * scale + padX,
Ycount * scale + padY,
(Color)convert.ConvertFromString("#e6e6e6"));
k++;
```

vi. *DrawBitmapOverlap:*

This method receives two 2D arrays which are to be compared and plotted together. And the rest of the parameters are similar to the previously mentioned method. The matching elements are colored ‘Blue’ and the mismatching elements are colored yellow and green for the first and second array respectively. And the inactive locations will be colored black as before.

```
if (twoDimArray1[Xcount, Ycount] == 1 &&
twoDimArray2[Xcount, Ycount] == 1)
{
myBitmap.SetPixel(Xcount * scale + padX,
Ycount * scale + padY, Color.Blue
k++;
}
```

IV. Test Cases and Results

The test cases were made keeping in mind that the patterns should always be different and the repetition of the same vector in multiple sequences should also be different. The other parameters apart from these only impact on the resolution of the output/input. The result for every test case is shown

i. *Test Case 1:*

The input sequence which is taken into consideration for Test Case 1 is as follows:

```
new double[] { 1.0, 3.0, 2.0 },
new double[] { 3.0, 1.0, 4.0 },
new double[] { 4.0, 1.0}
```

The campare number in this case which is considered is ‘1.0’.

```
inputs.setCompareNumber(1.0);
```

In this learning, it was intended that the vector ‘1.0’ should be present in all of the sequences and with relatively similar repetition

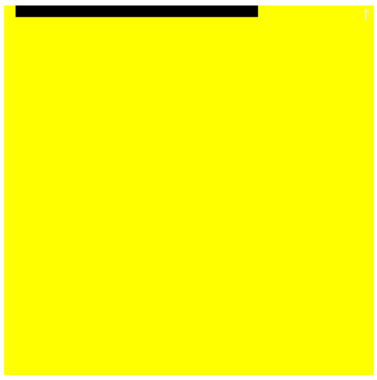

Figure 1: Input Vector pattern for ‘1.0’

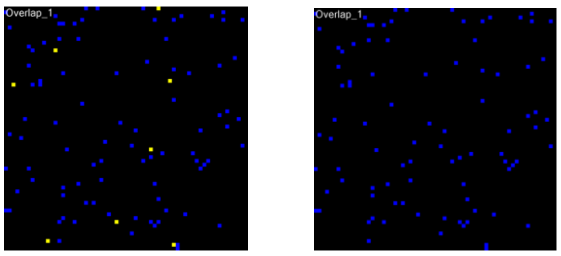

Figure 2 : Different Output Camparsion

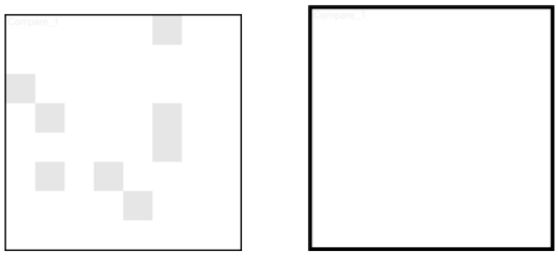

Figure 3 : Hamming Distance

Observation drawn in Test Case 1 **:**

Even though the vector’s repetitions were almost equal, the fist repetition showed some changes, however, the second time it was able to recall the learning.

ii. *Test Case 2:*

The input sequence which is taken into consideration for Test Case 2 is as follows

```
new double[] { 1.0, 1.0, 1.0 },
new double[] { 2.0, 2.0},
new double[] { 4.0, 4.0, 4.0, 4.0}
new double[] { 3.0, 2.0, 1.0}
```

The campare number in this case which is considered is ‘1.0’.

```
inputs.setCompareNumber(1.0);
```

In this learning, it was intended that the vector ‘1.0’ should be repeated in one sequence only and then to make in reappear after 2 sequences.

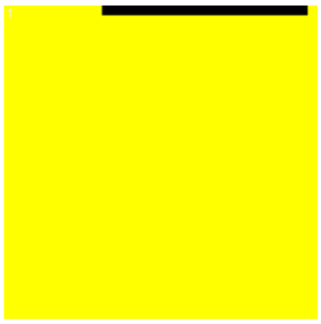

Figure 4 : Input Vector pattern for ‘1.0’

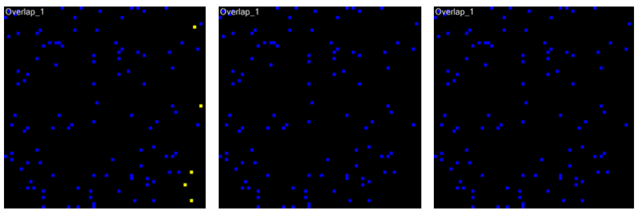

Figure 5 : Different Output Camparsion

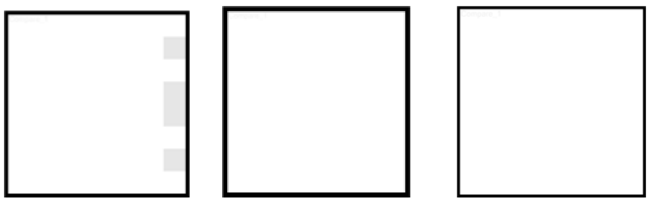

Figure 6 : Hamming Distance

Observation drawn in Test Case 2 :

This time from the comparisons it can be seen that except the first time it was able to recall the previous learning. Even though the last repetition was after a significant period, it completely coincided with the previous plot. Curious thing to see is, this time also it showed some changes in the first repetition.

iii. *Test Case 3:*

The input sequence which is taken into consideration for Test Case 3 is as follows

```
new double[] { 2.0, 1.0 , 3.0, 4.0},
new double[] { 3.0, 5.0 },
new double[] { 4.0, 4.0, 2.0, 1.0}
```

The campare number in this case which is considered is ‘1.0’.

```
inputs.setCompareNumber(1.0);
```

In this learning, it was intended that the vector ‘1.0’ should be repeated only **once** after one sequence in between

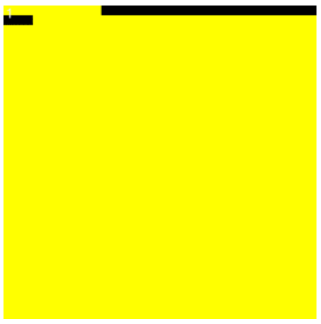

Figure 7 : Input Vector pattern for '1.0'

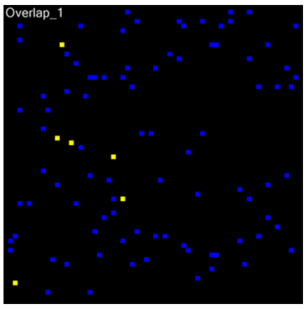

Figure 8 : Different Output Camparsion

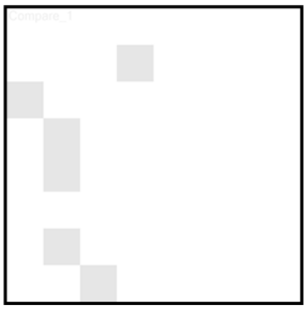

Figure 9 : Hamming Distance

Observation drawn in Test Case 3 :

With the reduction in repetition and with the increase of the in between period, it can be observed that the recall process has highly impacted and the output is very different from one another.

iv. ***Test Case 4:***

The input sequence which is taken into consideration for Test Case 4 is as follows

```
new double[] { 1.0, 1.0, 1.0 },
new double[] { 2.0, 2.0, 2.0 },
new double[] { 3.0, 3.0, 3.0 },
new double[] { 4.0, 4.0, 4.0 },
new double[] { 1.0, 1.0, 1.0 }
```

The campare number in this case which is considered is '1.0'.

```
inputs.setCompareNumber(1.0);
```

In this learning, it was intended that the vector '1.0' should have two repetitive sequences which will be separated by multiple other sequences. This was to find out the differences in the changes in the outputs within the sequences and outside the sequences

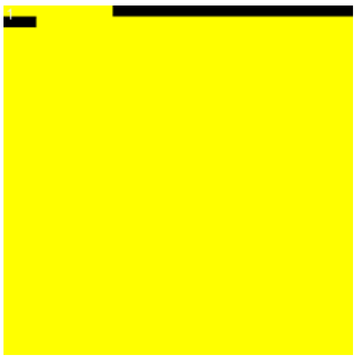

Figure 10 : Input Vector pattern for '1.0'

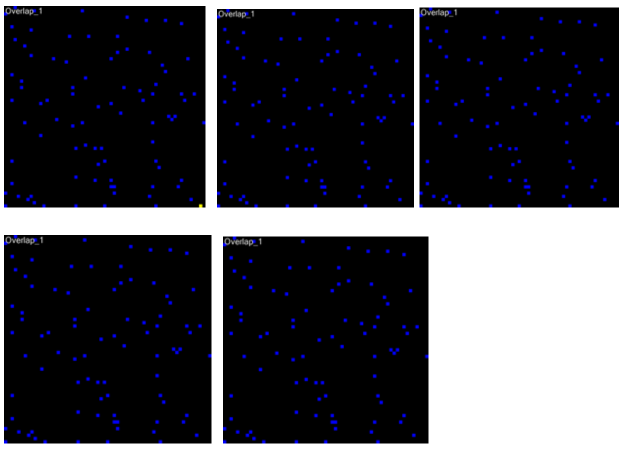

Figure 11 : Different Output Camparsion

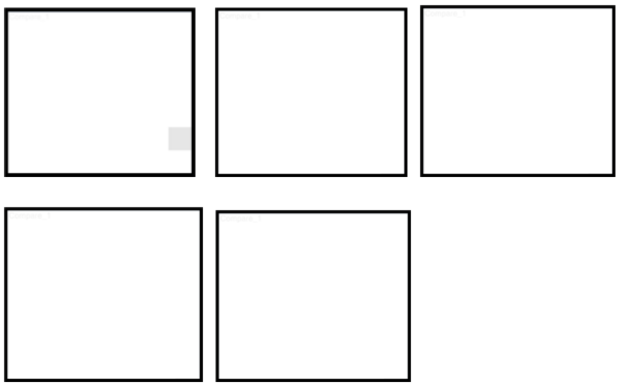

Figure 12 : Hamming Distance

Observation drawn in Test Case 4 :

This is the best result so far that we've got. There is a huge gap between the two repetitive sequences, but still, it was able to recall them properly. The periodic repetition seems to increase the probability of the recalling process. However, again the first repetition showed some changes as previously it did.

v. ***Test Case 5:***

The input sequence which is taken into consideration for Test Case 5 is as follows

```
new double[] { 1.0, 2.0, 3.0, 4.0 },
new double[] { 1.0, 2.0, 3.0, 4.0 },
new double[] { 1.0, 2.0, 3.0, 4.0 },
new double[] { 1.0, 2.0, 3.0, 4.0 }
```

The campare number in this case which is considered is '4.0'.

```
inputs.setCompareNumber(4.0);
```

In this learning, it was intended that the vector '4.0' shouldn't have any specific pattern of repetition and will have varying distances for each repetition

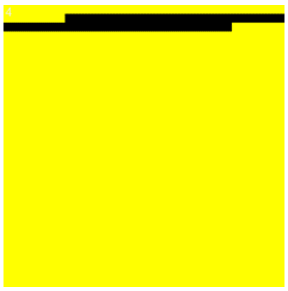

Figure 13 : Input Vector pattern for '4.0'

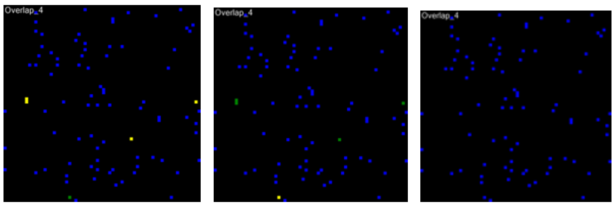

Figure 14 : Different Output Camparsion

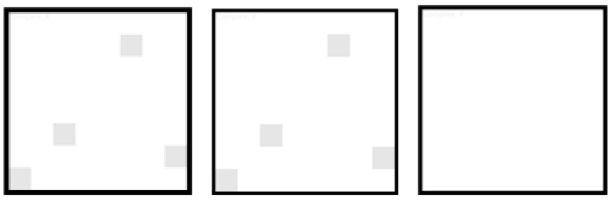

Figure 15 : Hamming Distance

Observation drawn in Test Case 5 :

This time the reading are very random and not following the pattern as they were previously doing. The first two repetitions were completely showed different results and not been recalled properly. However, the last repetition was able recall the learning successfully.

vi. ***Test Case 6:***

The input sequence which is taken into consideration for Test Case 6 is as follows

```
new double[] { 5.0, 3.0, 1.0, 4.0 },
new double[] { 3.0, 4.0, 2.0 },
new double[] { 1.0, 5.0, 3.0, 2.0, 1.0 },
new double[] { 5.0, 4.0, 1.0, 4.0 }
```

The campare number in this case which is considered is '6.0'

```
inputs.setCompareNumber(6.0);
```

In this learning, it was intended that the vector '6.0' should be provided with the maximum gap from its repeating vector than in other test cases.

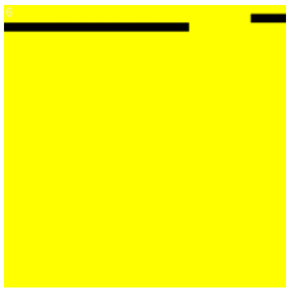

Figure 16 : Input Vector pattern for '6.0'

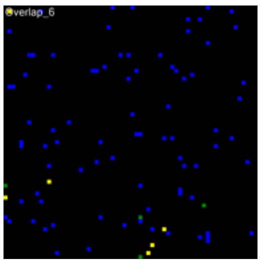

Figure 17 : Different Output Camparsion

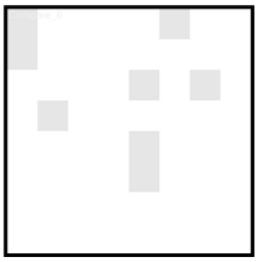

Figure 18 : Hamming Distance

Observation drawn in Test Case 6 :

This is one of the conclusive results that we have got so far. In this scenario, we have avoided the periodic repetition and identical sequences. And also provided the maximum periodical gap between the learnings. And the result being that, it completely failed to recall the learning

V. Conculsion

Even though the results of the learnings were not very patterned and were completely random at times, some specific points can be noted on the memorizing and recalling of the Spatial Pooler.

The most noticeable point was that when ever, the learnings are periodically repetitive, it provides the best result in terms of recalling the learning.

Secondly, the period gap between two repetitions seems to be impacting the memorizing capacity. Also, it was noted that the first repetition was also showed the most difference in the learning outputs than the others.

Hence, the number of encounter of the repetitions impacts the memorizing capacity as well. The more it encounters the same vector, the more accurately it produces the result.

VI. Refernces

[1] Hawkins, J. et al. 2016-2020. Biological and Machine Intelligence. Release 0.4. Accessed athttps://numenta.com/resources/biological-and-machine-intelligence/.

[2] Y. Cui, S. Ahmad and J. Hawkins, "The HTM Spatial Pooler—A Neocortical Algorithm for Online Sparse Distributed Coding", *Frontiers in Computational Neuroscience*, vol. 11, 2017. Available: 10.3389/fncom.2017.00111 [Accessed 25 March 2020]

[3] http://nupic.docs.numenta.org/0.6.0/spatial-pooler.html