Information Technology Course
Module Software Engineering
by Damir Dobric / Andreas Pech

# MIGRATION OF SPATIAL POOLER TO .NET CORE PARTS 2a, 2b and 3

Asmade ERASMUS SEMESTER STUDENT KABORE
asmade.kabore@gmail.om

***Abstract*— This paper presents the results from the study of the Spatial Pooler tutorial. It is a tutorial which presents some features of the spatial pooler. The goal of this research project was to study the implementation algorithms of those features' in Python in order to migrate them to .net core. HTM spatial pooler (SP) is a key component of HTM networks that continuously encodes streams of sensory inputs into SDRs. The SP is a core component of HTM networks. In an end-to-end HTM system, the SP transforms input patterns into SDRs in a continuous online fashion. The HTM temporal memory learns temporal sequences of these SDRs and makes predictions for future inputs [1].**

**Key words – Spatial Pooler, Hierarchical Temporal Memory, SDR (Sparse Distributions Representations)**

## I. INTRODUCTION

Spatial Pooler is a component of Hierarchical Temporal Memory (HTM). Hierarchical temporal memory (HTM) is a theoretical framework that models a number of structural and algorithmic properties of the neocortex [2]. Spatial Pooler converts binary inputs in Sparse Distributed Representations.
Numenta presented a tutorial of some python algorithms of the features of the HTM Spatial Pooler. My project purpose was to write these algorithms especially Parts 2a, 2b and 3 in C# for .Net Core. To conduct this study, I implemented each part in a UnitTest project. Then the final solution of the project is comprised of 3 unitTests projects. The paper is organized as follows: We first introduce the methodology used to reach this goal and then we will discuss the implementations of our test methods.
In Part 2a, we will show graphically that the input overlap of 2 identical input binary vectors is 1 and will decrease as we add noise to one of the 2 vectors.
In part 2b, we will show graphically how the output overlap changes with respect to the input overlap. The input overlap decreases when we add noise to one of the input vectors. The goal is to then to show how the output overlap decreases as we add noise to one of the input vectors.
In part 3, we will train our Spatial Pooler by presenting it random binary vectors a certain number of times. And after the training or learning, we will see that our Spatial Pooler shows some resilience to noise when we reproduce the graph in Part 2b.

All the three UnitTests almost use the same methods. They are all well commented.

## II. METHODS

### A- Spatial Pooler initialization

To initialize the Spatial Pooler, we have to assign values to some of its parameters. In order to understand our initialization, we will present some parameters which are documented by NUMENTA :

**POTENTIAL_RADIUS:**
This parameter indicates the number of input bits that each column can potentially be connected to. It's the number of input bits that are visible

to each column. A large enough value will result in 'global coverage', meaning that each column can potentially be connected to every input bit. This parameter defines a square (or hyper square) area: a column will have a max square potential pool with sides of length 2 * potential Radius + 1. The default value is 16.

**GLOBAL_INHIBITION: (bool)**
If true, then during inhibition phase, the winning columns are selected as the most active columns from the region as a whole. Otherwise, the winning columns are selected with respect to their local neighborhoods.
Using global inhibition boosts performance x60. The default value is False.

**NUM_ACTIVE_COLUMNS_PER_INH_AREA: (float)**
An alternate way to control the density of the active columns. If numActiveColumnsPerInhArea is specified then localAreaDensity must be less than 0, and vice versa. When using numActiveColumnsPerInhArea, the inhibition logic will insure that at most 'numActiveColumnsPerInhArea' columns remain ON within a local inhibition area (the size of which is set by the internally calculated inhibitionRadius, which is in turn determined from the average size of the connected receptive fields of all columns). When using this method, as columns learn and grow their effective receptive fields, the inhibitionRadius will grow, and hence the net density of the active columns will *decrease*. This is in contrast to the localAreaDensity method, which keeps the density of active columns the same regardless of the size of their receptive fields. The default value is 10.

**SYN_PERM_INACTIVE_DEC: (float)**
The amount by which an inactive synapse is decremented in each round.
Specified as a percent of a fully-grown synapse. The default value is 0.008.

**SYN_PERM_ACTIVE_DEC: (float)**
The amount by which an active synapse is incremented in each round.
Specified as a percent of a fully-grown synapse. The default value is 0.05.

**InputDimensions:**
A sequence representing the dimensions of the input vector. Format is (height, width, depth, ...), where each value represents the size of the dimension. For a topology of one dimension with 100 inputs use 100, or (100,). For a two-dimensional topology of 10x5, we use (10,5). Default is (32, 32).

**ColumnDimensions:**
A sequence representing the dimensions of the columns in the region. Format is (height, width, depth, ...), where each value represents the size of the dimension. For a topology of one dimension with 2000 columns, we use 2000, or (2000,). For a three-dimensional topology of 32x64x16, we use (32, 64, 16). Default ``(64, 64)``.

Here is the initialization in python from the tutorial of Numenta.

```
sp = SP
(inputDimensions
= 1000,
                    columnDimensions = 2048,
                    potentialRadius =
                    int(0.5*1000),
                    numActiveColumnsPerInhArea
                    = int(0.02*2048),
                    globalInhibition = True,
                    seed = 1,
                    synPermActiveInc = 0.01,
                    synPermInactiveDec = 0.008
                    )
```

SP is the spatial pooler class implemented in python and while creating an instance, we initialize the parameters this way. To have more details about all the parameters and their default values and even the full implementation of the Spatial Pooler class in Python, please go to this Numenta GitHub link:
https://github.com/numenta/nupic/blob/master/src/nupic/algorithms/spatial_pooler.py

For our project, a "Helpers.cs" file was added to all UnitTests projects in which the parameters can be initialized. In order to get the same results in C# as in python, we set the parameters in C# with the same values. All the other parameters except the input and column dimension are set with default values.

```
parameters.Set(KEY.POTENTIAL_RADIUS, 500);
            parameters.Set(KEY.POTENTIAL_PCT, 0.5);
            parameters.Set(KEY.GLOBAL_INHIBITION, true);
            parameters.Set(KEY.LOCAL_AREA_DENSITY, -1.0);

parameters.Set(KEY.NUM_ACTIVE_COLUMNS_PER_INH_AR
EA, 40.96);
            parameters.Set(KEY.STIMULUS_THRESHOLD, 0.0);
            parameters.Set(KEY.SYN_PERM_INACTIVE_DEC,
0.008);
            parameters.Set(KEY.SYN_PERM_ACTIVE_INC, 0.01);
            parameters.Set(KEY.SYN_PERM_CONNECTED, 0.1);

parameters.Set(KEY.MIN_PCT_OVERLAP_DUTY_CYCLES,
0.001);

parameters.Set(KEY.MIN_PCT_ACTIVE_DUTY_CYCLES, 0.1);
            parameters.Set(KEY.DUTY_CYCLE_PERIOD, 1000);
            parameters.Set(KEY.MAX_BOOST, 0.0);
parameters.Set(KEY.RANDOM, rnd);
```

## B- Implementations of methods

As some methods are implemented by many lines of code, we will not provide methods codes in this paper. We will just show their signature and explain what they do. To have access to all methods implementation for each project, please visit my project section in GitHub with the following:
**https://github.com/UniversityOfAppliedSciencesFrankfurt/se-dystsys-2018-2019-softwareengineering/tree/Asmade-Kabore**

In Python version of the algorithms, some lists were used to store the results of the methods but in our case, the results are saved in text files. For that, we used a method called WriteArrayToFile. This method signature is as follows:

```
private static void WriteArrayToFile (string
filename, double [] data)
```

First parameter is the name of the text file in which the second parameter data will be saved. Data in the text file will be separated by "|". The text files are automatically created in UnitTestSpatialPoolerPart2b\bin\Debug\netcoreapp2.1 for example for UnitTest of Part 2b.
We will see later in this section what kind of data are saved for each part.
As said in the introduction, it is shown graphically how the input overlap in part 2a decreases as we add noise to one of the 2 input binary vectors. So, we needed to plot line graph of the input overlap with respect to the noise. In python, it is easily implemented but in C# in order to gain time, we decided to run python plotting script directly in UnitTest.
A method called runPythonCode provided by Mr. Dobric was used for that purpose. Here is Its signature:

```
public static void runPythonCode(string
filename1, string filename2)
```

This method parameters are text files that contain data we want to plot. Inside this method, we call the plotting script and then give it the data as arguments for the command prompt.

But before we are able to use these 2 methods, data should be first obtained and processed. Methods were implemented for that purpose too.
As we will see it through this paper, the spatial pooler should show some resilience to noise. Some noise quantified in percentage will then be added to one of the input binary vectors. Adding 5% of noise to an input binary of 100 bits size vector means that out the 100 bits, we will randomly take 5 and flip their state. For instance, if a selected bit is 1, it will be set to 0.
We can come to the conclusion that considering 2 binary identical input vectors, if a noise level of 100% is added to one of the two, we obtain 2 opposite vectors, one being the negative of the other.
A method called corruptVector was implemented for that purpose:

public static int[] corruptVector(ref int[] vector, float noiseLevel)

The parameters are the binary vector to which noise will be added and the noise level and the return value is a corrupted vector (vector to which noise was added).

We needed to compute the percentage of overlap between 2 binary vectors. Then, a method called percentOfOverlap was implemented.

the input overlap between two binary vectors is defined as their dot product. This value is normalized by dividing it by the minimum number
of active inputs.
The dot product of two vectors a = [$a_1$, $a_2$, …, $a_n$] and b = [$b_1$, $b_2$, …, $b_n$] is defined as:
$a*b = a_1b_1+a_2b_2+…+a_nb_n$

A method called dotProduct calculating and

returning the dot product of 2 vectors was implemented:

```
public static float dotProductMethod(int[] x1,
int[] x2)
        {

            float dotProductResult = 0;

 for (int i = 0, j = 0; i < x1.Length; i++,
j++)

            {

          dotProductResult += x1[i] * x2[j];

            }

            return dotProductResult;
        }
```

Once, we have the dot product of the 2 input vectors, we determine the vector with the minimum number of 1 or the vector with less active columns. A method was implemented to take a binary vector and returns the number of its active columns. Here it is:

```
public static int count_nonzero(int[] x)
            {
int numberOfNonZeros = 0;
for (int j = 0; j < x.Length; j++)
                {
                    if (x[j] == 1)
                    {
                        numberOfNonZeros += 1;
                    }

                }

                return numberOfNonZeros;
            }
```

Our vectors are binary ones and are then composed of 0 and 1. Let's give an example to understand the input overlap calculation.
Assuming that we have 2 binary vectors X and Y defined as:

X = [0, 1, 1, 0, 1, 1, 0]

Y = [1, 1, 0, 1, 1, 0, 1]

The number of "1" in X1 is 4 and the one in X2 is 5. The minimum is then 4 which means we will obtain the percent overlap by dividing the dot product by 4.

The dot product will then be:

$X*Y = X_1*Y_1 + X_2*Y_2 + … + X_nY_n$

Here n = 7

$X*Y = 0*1 + 1*1 + 1*0 + 0*1 + 1*1 + 1*0 + 0*1$

$= 0+1+0+0+1+0+0$

$= 2$

The percent overlap is then 2/4 = 0.5 = 50%

There are methods that are only implemented and used in part 3:

public static void resetVector(int[] x1, int[] x2) which copies the content of vector x1 to x2 giving 2 identical vectors.

public static int[] reverseSort(int[] x) which will sort the elements of the vector x in a reverse order.

And the last method whose implementation will be shown below takes a 2-dimensional vector and the number of row we want and then returns a one dimensional vector:

```
public static T[] GetRow<T>(T[,] matrix, int row)
        {
            var columns = matrix.GetLength(1);
            var array = new T[columns];
            for (int i = 0; i < columns; ++i)
                array[i] = matrix[row, i];
            return array;
        }
```

If we have for instance the following matrix (3x3):

| | | | |
|---|---|---|---|
| 0 | 1 | 1 | ← ROW 0 |
| 0 | 0 | 1 | |
| 1 | 0 | 0 | |

GetRow(matrix,0) will return the first row which is:

| | | |
|---|---|---|
| 0 | 1 | 1 |

This method will be very useful when it will come to train the Spatial Pooler with a variety of binary vectors.

## III- Implementations and results of each part

### A- Part 2a

In this section, the goal was to show graphically how the percent overlap decreases as we add noise to one of the input vectors.

- **Code architecture:**

Part 2a is implemented in a UnitTest project called

UnitTestSpatialPoolerPart2a which is itself part of the UnitTestSpatialPooler solution.
The project folder contains 3 files:

-Helpers.cs for the Spatial Pooler parameters configurations.
-create-plots-Part2a.py which is python script that will create graphs from data obtained in the UnitTestSP2a.cs.

-UnitTestSP2a.cs which contains all the implementations of methods in the test class public class UnitTestSP2a. The main method or entry method is the test method public void TestMethod1().

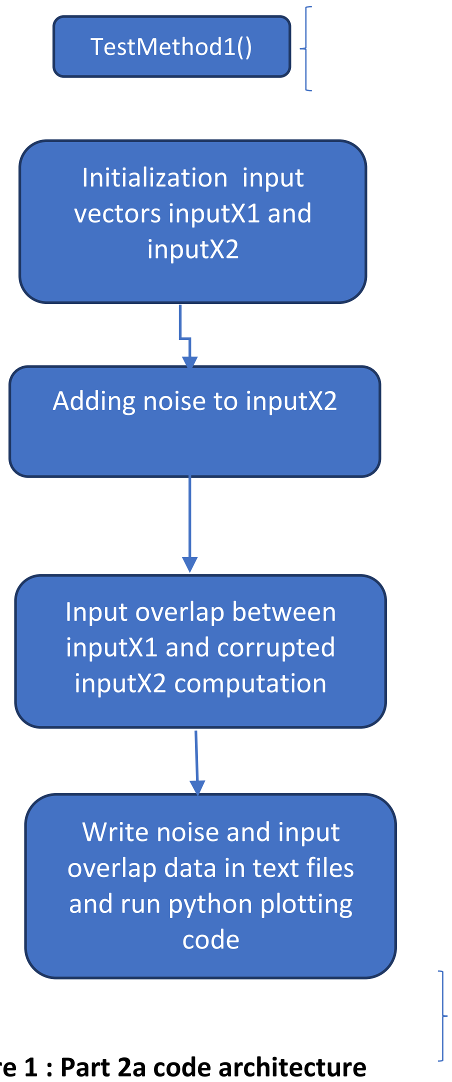

TestMethod1()
Initialization input vectors inputX1 and inputX2
Adding noise to inputX2
Input overlap between inputX1 and corrupted inputX2 computation
Write noise and input overlap data in text files and run python plotting code

**Figure 1 : Part 2a code architecture**

- **Implementation in C#**

Reference to UnitTestSpatialPoolerPart2a project:
**https://github.com/UniversityOfAppliedSciencesFrankfurt/se-dystsys-2018-2019-softwareengineering/tree/Asmade-Kabore/MyMachineLearningProject/UnitTestSpatialPoolerPart2a**

The test is implemented in TestMethod1().
We define 2 input vectors inputX1 and inputX2 which are integer arrays of size 1000.
And then we will feed them with random binary numbers (0 and 1).
We had to add noise to one of our input vectors before computing their input percent overlap and then write these 2 data to text files through WriteArrayToFile (string filename, double [] data) method. The data have to be then in arrays. That's why we used 2 arrays one for the noise level and the other one for the input percent overlap.

```
noiseLevelArray = new double [13];
double [] percentOverlapArray = new double [13];

float [] Noise = new float [] {0, 5, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100};
```

Then, we will take each noise level using a for loop in which inputX1 vector will be corrupted at each iteration.
Then through a for loop, we go through the array containing the level of noise. At each iteration, the vector input X2 is corrupted with the noise level whose index corresponds to the iteration number.
Then we pass the inputX2 vector we just corrupted and the inputx1 to the percentOverlap method and we put the result in an array that we named percentOverlapArray. When we exit the loop, we write data in percentOverlapArray in a text file named percentInputOverlap.txt and noise level in noiseLevel.txt. The interest is to show graphically how the percent overlap of the 2 vectors behaves in function of the noise level. We pass these 2 data contained in these text files to the method runPythonCodePart2a that will launch the python script create-plots-Part2a.py to plot the desired graph.

- **Result in C# .NET CORE**

The below figure (figure 2) shows the input overlap between 2 identical binary vectors in function of the noise level added to one of them.
0 percent noise level means that the vector remains the same, whereas 100 percent noise Level means the vector is the logical negation of the original vector.
The graph shows that the relationship between overlap and noise level is linear and monotonically decreasing.

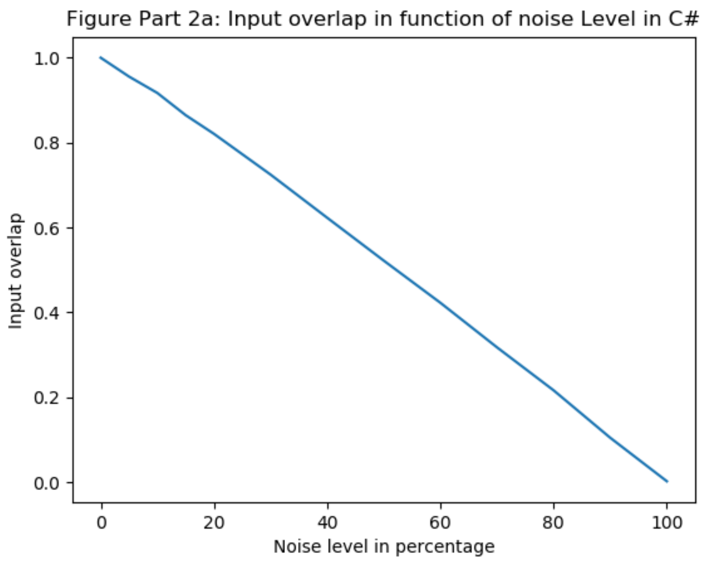


**Figure 2: input overlap in function of noise from C#**

- **Result of Python part 2a algorithm test**

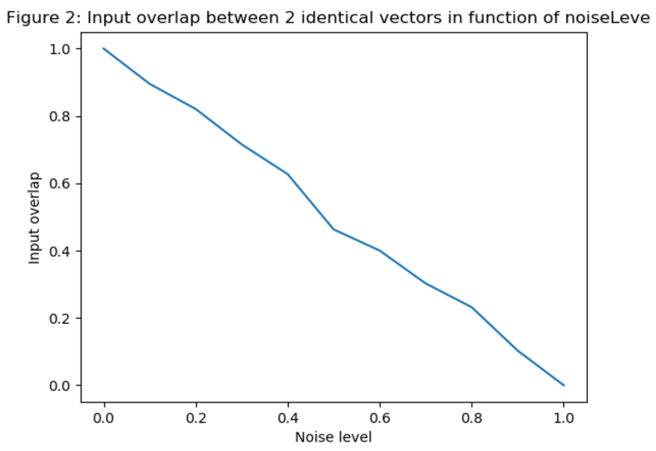


**Figure 3 : input overlap in function of noise in Python**

The difference of this graph with the C# one is the slope. The one in C# is linear whereas the python one is not entirely linear. This means

there is some resilience to noise in python algorithm.
We can’t get exactly the same graphs as we don’t use the same input vectors which are fed with random binary numbers.

**B – Part 2b**

In this part, the goal was to show how the output overlap behaves in function of the input overlap of 2 vectors.
The output overlap is the overlap of the columns that become active when fed to the Spatial Pooler.  Learning is turned off and we observe the output of the SP as we input two binary vectors. We will see that 2 identical vectors will have the same active columns.
The output overlap decreases when the input one decreases, thus when we add noise.

**Code architecture:**

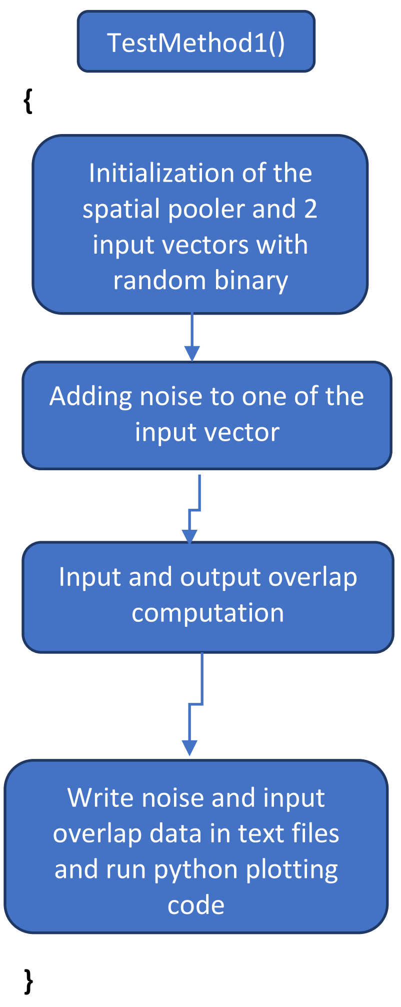

TestMethod1()
{
Initialization of the spatial pooler and 2 input vectors with random binary
Adding noise to one of the input vector
Input and output overlap computation
Write noise and input overlap data in text files and run python plotting code
}

**Figure 4 : Part 2b code architecture**

- **Implementation in C#**

Part 2b is implemented in a UnitTest projet called UnitTestSpatialPoolerPart2b. It contains 3 files:
Helpers.cs where parameters of the spatial pooler are set.
-Create-plots-Part2b.py which is used to plot graphs
-UnitTest2b.cs where test is implemented as shown below

```
namespace UnitTestSpatialPoolerPart2b
    {
      [TestClass]
      public class UnitTest2b
        {
         [TestMethod]
         public void TestMethod1()
        {
         //Main CODE
         }

              // Methods implementations
              }
        }
```

Reference to UnitTestSpatialPoolerPart2b Project:
[https://github.com/UniversityOfAppliedSciencesFrankfurt/se-dystsys-2018-2019-softwareengineering/tree/Asmade-Kabore/MyMachineLearningProject/UnitTestSpatialPoolerPart2b](https://github.com/UniversityOfAppliedSciencesFrankfurt/se-dystsys-2018-2019-softwareengineering/tree/Asmade-Kabore/MyMachineLearningProject/UnitTestSpatialPoolerPart2b)

Our main code is implemented in TestMethod1(). As we have to see how the output overlap changes with the input one, we needed to use 2 input Vectors inputX1 and inputX2 with size 1000 and 2 output vectors outputX1 and outputX2 with size 2048. InputX1 is first fed up with random binary numbers and then it is copied to inputX2 so that these 2 vectors have the same values. the 2 output vectors are initialized with 0 meaning that all their columns are inactive. Noise Levels are in an array as Part 2a
and Input and output overlap results will be put in arrays.

```
float[] Noise = new float[] { 0, 5, 10, 15, 20,
40, 50, 60, 70, 80, 90, 100 };
double[] percentOverlapArrayInput = new
double[Noise.Length];

double[] percentOverlapArrayOutput = new
double[Noise.Length];
```

The output overlap is the overlap of the columns that become active when fed to the Spatial Pooler. means before computing the output percent overlap, we will have to fed our 2 output vectors to Spatial Pooler which has to be initialized.
Parameters of the SP are initialized in Helpers.cs
and we get these parmeters values in UnitTest2b.cs with the following instruction:

```
var parameters = Helpers.GetDefaultParams();

parameters.setInputDimensions(new int[] { 1000 });
parameters.setColumnDimensions(new int[] { 2048 });
var sp = new SpatialPooler();
var mem = new Connections();
parameters.apply(mem);
sp.init(mem);
```

And then like in part 2a, we use a for loop with which each each iteration will correspond to a noise level.
What is done at each iteration is that
for example, for the first iteration when j = 0, we will retrieve from the Noise array, the value whose index corresponds to 0 and use it to corrupt inputX2 like this:

```
inputX2 = corruptVector(ref inputX1, Noise[j]);
```

Before the for loop, inputX2 = inputX1
That's why inputX1 is used in method corruptVector. inputX1 is the reference vector to which we add noise.
After that, we fed the two output vectors to the SP:

```
sp.compute(mem, inputX1, outputX1, false);
sp.compute(mem, inputX2, outputX2, false);
```

After these 2 computations, some columns of outputX1 and outputX2 will be activated so that we can know compute their output overlap as follows:

```
percentOverlapArrayOutput[j]=
percentOverlap(outputX1, outputX2);
```

When noise level is 0, it means inputX1 and inputX2 are identical which leads to an input overlap of 1. At the same time, outputX1 and outputX2 will be identical after feeding them to the spatial pooler. This will result in an output overlap of 1. We will check this in the next next section for results.

- **Result in C# .NET CORE**

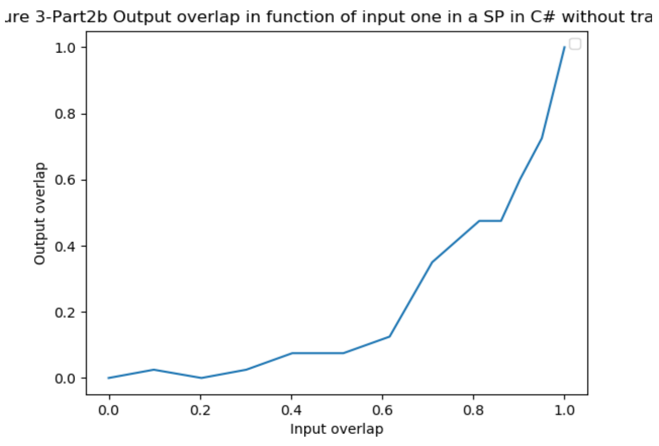


**Figure 5: output overlap in function of input overlap in a SP without training**

It shows the output overlap between two sparse representations in function of their input overlap. When noise level is 0, we have two identical binary vectors inputX1 and inputX2 which will result in the same active columns.
Then, we feed it to the SP and estimate the output overlap between the two representations in terms of the common active columns between them.
As we can see in figure 7, our line graph shows the expected results. When the input overlap decreases meaning that we are adding noise to one of the input vectors, then the output overlap also decreases.

➢ **Obtained Result in Python**

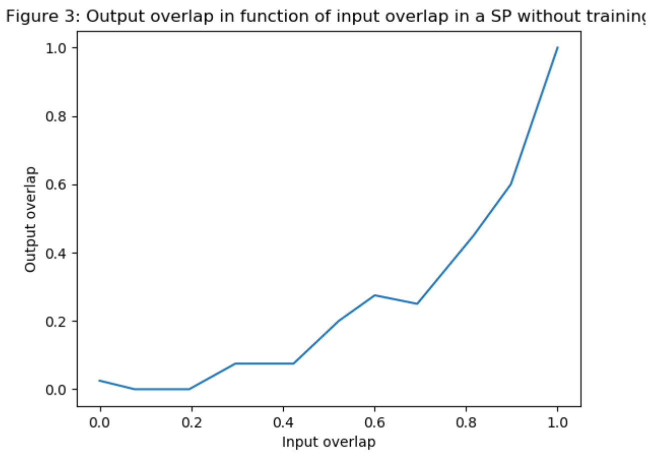


**Figure 6: output overlap in function of input overlap Python**

Our graph from C# code behaves as expected. The 2 graphs have the same behavior but are not exactly the same. That's due to the fact that we don't use the same vectors in C# as in python as they are fed with random numbers.

## C - Part 3

After training, a Spatial Pooler can become less sensitive to noise. For this purpose, we train the Spatial Pooler by turning learning on, and by exposing it to a variety of random binary vectors. We will then expose the SP to a repetition of input patterns in order to make it learn and distinguish them once learning is over.
This will result in robustness to noise in the inputs. In this section we will reproduce figure 5 and figure 6 seen in Part 2b section after the SP has learned a series of inputs.
We will present 10 random vectors to the SP, and repeat these 30 times.
Later we will change the number of times we do this to see how it changes the last plot[3].

➢ Implementation in UnitTest3

Part 3 is implemented in a UnitTest projet called UnitTestSpatialPoolerPart3. It contains 4 files:
-Helpers.cs for the Spatial Pooler parameters configurations.
-create-plots-Part3-fig4a.py which is python script that will create graphs.
-create-plots-Part3-fig4b.py
-UnitTestSP3.cs which contains all the implementations of methods in the test class public class UnitTestSP3. The main method or entry method is the test method public void TestMethod1().

Reference to UnitTestSpatialPoolerPart3 project:

https://github.com/UniversityOfAppliedSciencesFrankfurt/se-dystsys-2018-2019-softwareengineering/tree/Asmade-Kabore/MyMachineLearningProject/UnitTestSpatialPoolerPart3

As mentioned above, the SP will be exposed to a variety of binary input vectors. In our code, the variable numExamples =10 (by default) represents the number of vectors.
After that, we declare a 2-dimensional input binary vector called inputVectors and feed it with random binary numbers.
Each row of the 2-D inputVectors corresponds to one of the numExamples input vectors.
Then, we initialize and configure the spatial pooler the same way as in section Part 2b.

➢ Overlaps before training

An input binary vector called inputVector and feed with random binary numbers is used to train the SP but without learning.

```
int [] inputVector = Helpers.GetRandomVector(
1000
parameters.Get<Random> (KEY.RANDOM));
```

And then, we define a vector called activeCols that

will be presented to the spatial pooler. activeCols is initialized with 0 meaning that all its columns are inactive. After showing it to the SP with InputVector some of its columns will be activated. And after that we get the overlap score that will be sorted in a reverse order before being written to the text file overlapBeforeTraining.txt.

```
int [] activeCols = new int [2048];
sp.compute(mem, inputVector, activeCols, false);
 var overlaps = sp.calculateOverlap(mem,
 inputVector);
overlaps = reverseSort(overlaps);
    WriteIntArrayToFile("overlapBeforeTraining.txt",
overlaps);
```

The overlap score for a column is the number of connections to the input that become active when presented with a vector.

And after that, we determine the overlap scores of each active column in activeCols vector and add it to the list activeColScores.
And for graph purpose, the number of active columns is determined and written to a text file named numberOfActCols.txt.

- Overlaps after training

  - training of the spatial pooler

we are going to train the SP by presenting 10 random vectors to it and repeat these 30 times. Yo notice in the portion of code below that learning is turned on.

```
for (int i = 0; i < epochs; i++)
        {
            for (int k = 0; k < numExamples ; k++)

            {
inputVectorsRowk = GetRow(inputVectors, k);
            outputColumnsRowk = GetRow(outputColumns, k);
sp.compute(mem, inputVectorsRowk, outputColumnsRowk, true);
            }
        }
```

Epochs = 30 and numExamples=10

- overlaps computation

After training the SP, we compute the overlap (array) and sort it reversively and write it to the text file overlapAfterTraining.txt.
And then we call runPythonCode2 to launch create create-plots-Part3-fig4a.py.

```
runPythonCode2("overlapBeforeTraining.txt",
"overlapAfterTraining.txt", "numberOfActCols.txt"
);
```

- Input overlap and output overlap after training

  It is implemented the same way as in part 2b.
  Only the input vectors and output vectors names are different.

We will change change Epochs and numExamples and show the graphs in the next section.

## 1. Epochs = 30 and numExamples=10

- Result in C# .NET CORE

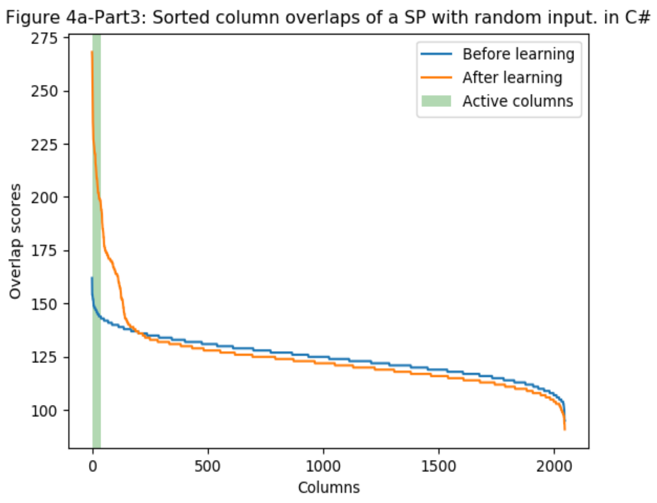


**Figure 7: sorted columns overlaps scores**

Figure 7 above shows the sorted overlap scores of all columns in a spatial pooler with random input, before and after learning. The top 2% of columns with the largest overlap scores, comprising the active columns of the output sparse representation are highlighte green.

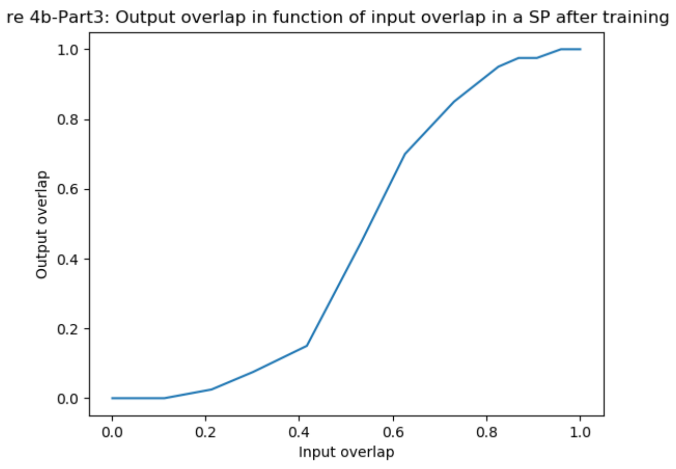


**Figure 8: output overlap in function of input overlap**

How robust is the SP to noise after learning?
Figure 8 shows again the output overlap between two binary vectors in function of their i overlap as seen in Part2b. After training, the SP exhibits more
robustness to noise in its input, resulting in a -almost- sigmoid curve.
This implies that even if a previous input is presented again with a certain amount of noise its sparse representation still resembles its original [4].

- Results obtained in Python

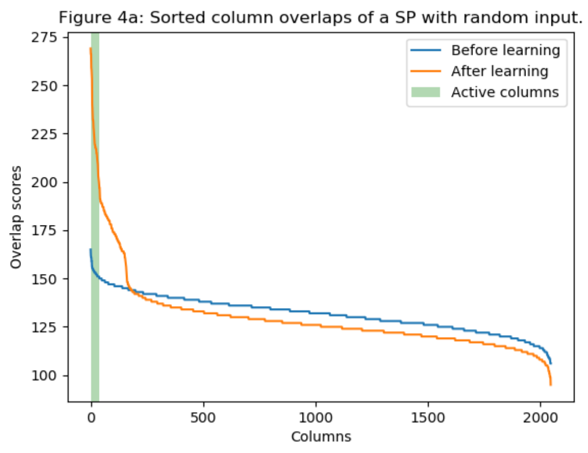


**Figure 9: sorted columns overlaps scores**

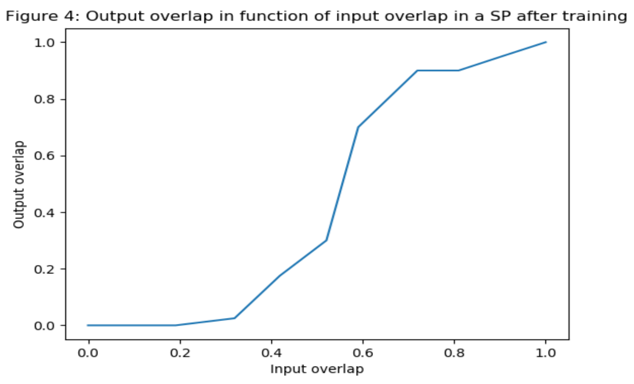
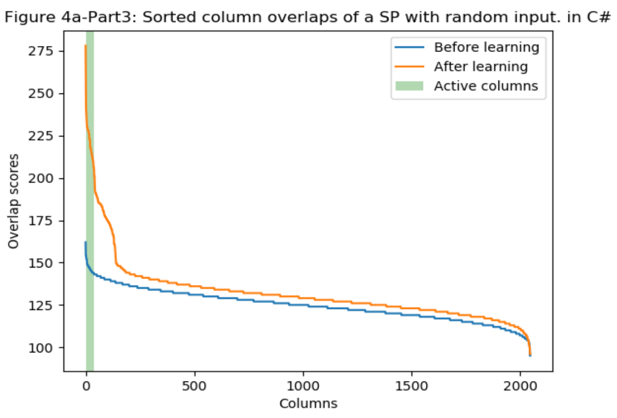


**Figure 10: output overlap in function of input overlap**

**Epochs = 20 and numExamples=10**
**Only from C#**



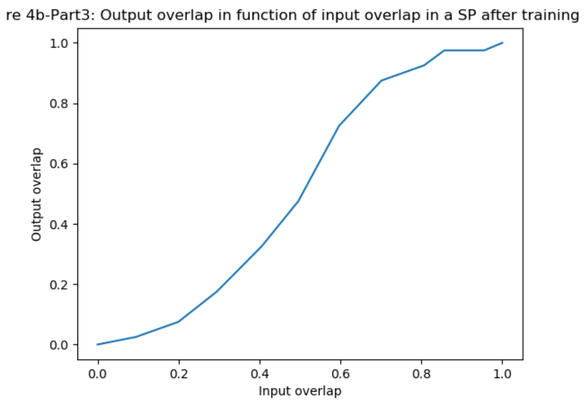
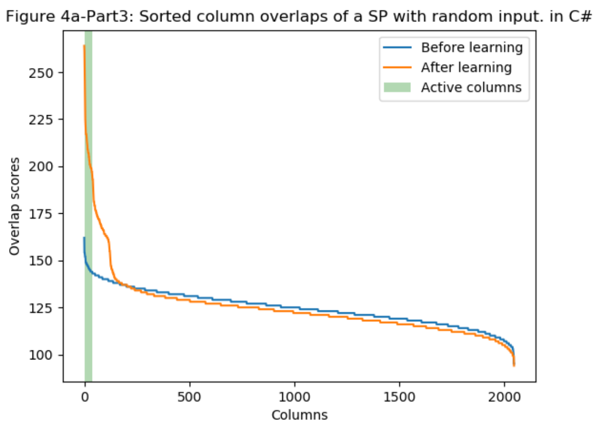


**Figure 10: With epochs = 20 in C#**

### Epochs = 10 and numExamples=10
### Only from C#



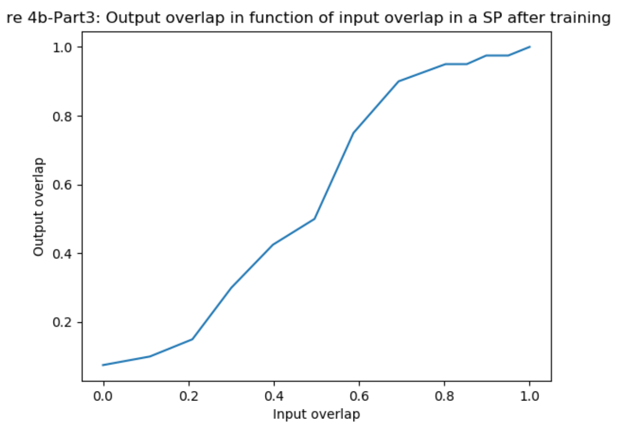
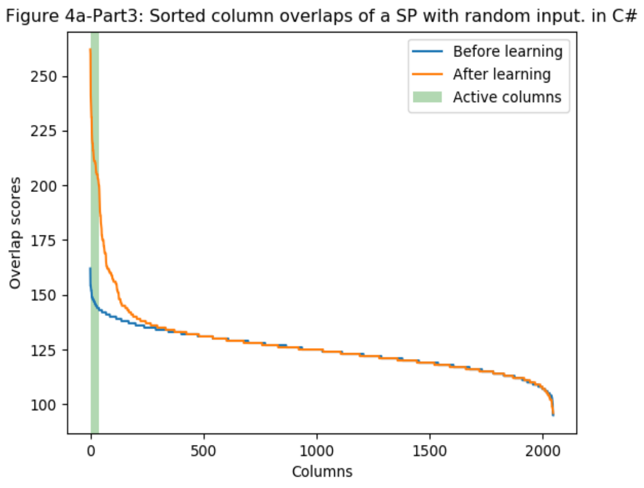


**Figure 12: with epochs = 10 in C#**

### Epochs = 60 and numExamples=10
### Only from C#



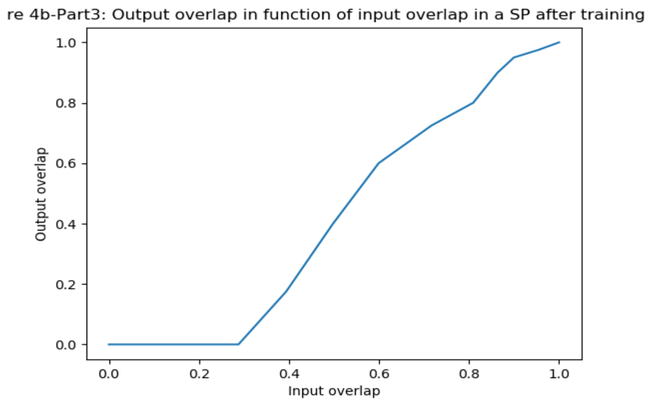


**Figure 13: With epochs = 60 in C#**

## IV- Conclusion and discussion

In this paper, we described some features of the HTM spatial pooler. Inspired by computational principles of the neocortex, the goal of the HTM spatial pooler is to create SDRs and support essential neural computations such as sequence learning and memory. The model satisfies a set of important properties, including tight control of output sparsity, efficient use of mini-columns, preserving similarity among inputs, noise robustness. Our line diagrams are a little bit different from the python ones. One reason may be the fact we didn't use the same input vectors. They are all fed with random numbers. It will be insterresting to use the same input vectors and see whether the graphs are exactly similar or not. For part 3, it will also be interesting to modify the number of examples of input vectors that are used to train the Spatial Pooler to see how it behaves and see if there is a relationship between the number of examples. The HTM spatial pooler leads to a flexible sparse coding scheme that can be used in practical machine learning applications.

## V- References

[1] Cui, Y., Liu, L., McFarland, J., Pack, C., and Butts, D. (2016b). Inferring Cortical variability from local field potentials. *J. Neurosci.* 36, 4121–4135. doi: 10.1523/JNEUROSCI.2502-15.2016

[2] Numenta White paper 1-68

[3], [4] GITHUB of NUMENTA