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

Implement the New Spatial Pattern Learning Experiment

Fahim Talukdar
fahim.talukdar@stud.fra-uas.de

A.S.M Saiem Solimullah
a.solimullah@stud.fra-uas.de

Nowrin Tasnim Sinthia
nowrin.sinthia@stud.fra-uas.de

Ashraf Uddin
ashraf.uddin@stud.fra-uas.de

Abstract*— Spatial pattern learning, particularly in the context of algorithms like the Spatial Pooler in Hierarchical Temporal Memory (HTM) models, refers to the process of learning patterns of activation in a spatially distributed manner. In HTM theory, the Spatial Pooler is a crucial component responsible for learning spatial patterns in the input data. It receives sparse binary input signals (often encoded from sensory data) and learns to create a sparse distributed representation (SDR) of the input in a high-dimensional space. Spatial pattern learning is valuable for a wide range of applications, including sensory data processing, anomaly detection, pattern recognition, and predictive modeling. Its efficiency, robustness, and adaptability make it a powerful tool for building intelligent systems that can learn from and interact with complex real-world environments. The previous experiment of Spatial Pattern Learning wasn't efficient as it didn't give the SDR (mini columns) of inputs 51 to 99 for the first 40 cycles and ran for 1000 iterations which isn't time efficient. This new Spatial Pattern Learning, at first, it will resolve the issue of not giving the SDR for inputs 51 to 99, then it will break the iterations by matching some criteria as well as give statistical information and better representation of SDRs to show how it changes in each cycle of Spatial Pattern Learning.

I. Introduction

Human brain continuously receives a vast amount of information from the real-world scenario through different sensors. By creating synaptic connections with a portion of the presynaptic neurons, each cortical neuron must make sense of a deluge of time-varying inputs. With the combination of huge number of neurons, a convoluted sheet of neural tissue forms which called neocortex. Our perception and behaviour are influenced by the way groups of neurons activate collectively.

Hierarchical Temporal Memory (HTM) is a framework which explains certain structural and algorithmic characteristics of the neocortex [1]. It learns the procedure that occurs in one cortex of the brain also worked on continuous stream of input patterns which represent input sequences based on the input stream's recursive pattern [2].

HTM is constructed with three parts named as Encoder, Spatial Pooler and Temporal Memory. Encoder encodes the data and send this data to spatial pooler and then from there spatial pooler set an algorithm for which column of neocortex should be activated. Temporal Memory works by forming associations between patterns of activity in the neocortex over time. It learns sequences of patterns and makes predictions about what is likely to happen next based on past observations. This is achieved through the formation of predictive connections between neurons, which allow the system to anticipate future states based on the current input and past context.

Spatial Pattern Learning using Sparse Distributed Representations (SDRs) represents a pivotal area within the realm of artificial intelligence, drawing inspiration from the brain's remarkable ability to process and recognize spatial patterns which is based on the fundamentals of hierarchical temporal memory (HTM). In HTM, Sparse Distributed Representations are crucial data structures, posing a central challenge in their construction and management. The Spatial Pooler Algorithm transforms a sequence of input bits into the SDR representation. To understand this representation, these SDRs are influenced by the structure of the human brain neocortex, specifically the activated columns on the brain membrane surface. In essence, the algorithm helps convert information into a format inspired by the brain's architecture, contributing to the study of intelligence in artificial systems.

The key concept of our study is that Sparse Distributed Representations, can act as a common syntax for different inputs. This project tried to show that, after training for a certain limit this algorithm can be form a stable state from where the detection of the input data can provide much accuracy for different inputs.

II. Literature Review

A. Hierarchical Temporal Memory

The Hierarchical Temporal Memory (HTM) is a machine learning algorithm inspired by how the neocortex, a part of the human brain, organizes and processes information [3]. It mimics how our brain handles various types of information like vision, audio, and behavior. The HTM network has the ability to learn from input patterns, recognize them, and make predictions based on what it has learned.

The name "Hierarchical Temporal Memory" suggests its three main features: hierarchy, temporal, and memory. In terms of hierarchy, an HTM network is structured in multiple levels organized hierarchically. Information from lower levels is combined at higher levels, creating more complex components. Regarding temporal aspects, an HTM network can learn both spatial (related to space) and temporal (related to time) patterns from a continuous stream of data. This means the network's output at a given time not only depends on the current input but also on previous ones.

The ability of HTM to predict patterns based on previously trained data patterns. HTM receives a unique pattern after a few iteration of the program it compares the previous patterns to the current pattern. Input patterns should not repeat, and the uniqueness should be maintained.

Unlike storing all data in a single large memory bank, HTM distributes information among columns within each level. This distributed approach enhances efficiency and allows the network to process information in a way that mirrors the brain's functioning. In essence, HTM is a machine learning algorithm that takes inspiration from the human brain's structure and functions, particularly in its ability to learn from various inputs, recognize patterns, and make predictions.

B. Sparse Distributed Representations (SDR's)

SDR's are the language of the brain [4]. The way brain presents any kind of information is called Sparse Distributed Representation. Means to build or present something like the way brain works, it is needed to create a system which can function like the SDR's.

This Sparse Distributed representations (SDRs) of input patterns are used in HTM's algorithm. Using an amount of active bits, it produces SDRs internally. These bits have semantic value. As a result, two inputs with equivalent semantic meaning must have equal active bit representation in SDR, which plays an important role in HTM learning.

Sparse Distributed Representations refer to binary data representations that consist of numerous bits, with only a small fraction being active (set to 1). Within these representations, each bit holds semantic significance, and this meaning is spread across the various bits. In essence, Sparse Distributed Representations encode information in a binary format where a majority of the bits remain inactive, each carrying a part of the overall meaning. Figure 2 represents cells with active columns (red dots) for a defined input 1.

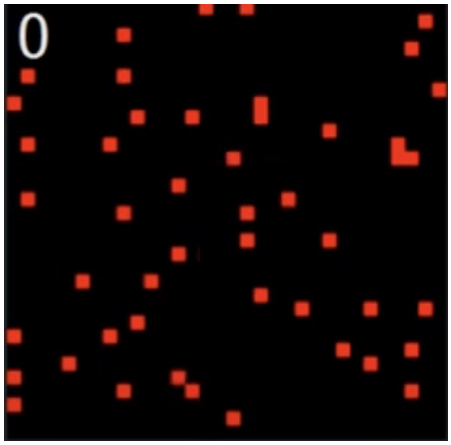

Figure 1: Sparse Distributed Representation

Sparse Distributed Representations, serve as a means to express patterns of activity within a network. Every input that enters the network undergoes a transformation into an SDR. Following this transformation, the network's hierarchy of nodes processes the SDR to formulate predictions regarding forthcoming inputs.

In simpler terms, SDRs are utilized to capture and depict the activity patterns within a network. They act as a coded representation of the inputs, allowing the network to analyze and make predictions about what might come next based on these transformed representations.

C. Encoder

Encoder is an important component within a system responsible for converting the raw data from its original format into a suitable representation. Encoders generally apply scaling, binning, mathematical transformations, or other methods to transform input data into a format that reduces dimensionality and duplication while maintaining pertinent information.

In this program, Encoders generates SDRs with a constant number of bits 'N' and a fixed number of active (1's) bits 'W', regardless of what they represent. An encoder should always provide the same output for the specific input. If the encoders offered various SDR bit lengths, comparisons and other operations would not be possible [5].

D. Spatial Pooler:

Spatial Pooler is another core component of Hierarchical temporal memory. The Spatial Pooler continuously converts input patterns into SDRs in an HTM system. The HTM temporal memory learns temporal sequences of these SDRs and makes predictions for future inputs [6]. It works in three phases namely overlap, inhibition and learning.

Spatial Pooler which is used to create SDRs addresses the challenge of representing active neurons from sensory or motor organs, facilitating the cortex in learning sequential patterns. It takes input vectors of varying sizes and transforms them into sparse vectors of uniform size, effectively normalizing the representation. The output of the Spatial Pooler signifies mini-columns, specifically the pyramidal neurons in the cortices.

This algorithm has some specific properties, including preserving overlap properties, ensuring similar inputs produce comparable outputs in the columns. The Spatial Pooler processes input data, converting it into active columns. In this process, each mini-column is connected to a set number of synapses from the input to learn sequences. An overlap score is then computed, and if it surpasses a threshold value, the column is activated. Otherwise, it remains inactive. Essentially, the Spatial Pooling Algorithm plays a crucial role in pattern learning by transforming input data into a structured representation that possesses to specific properties and thresholds.

III. DESCRIPTION OF EXITING ALGORITHM

Existing algorithms implement an experiment that demonstrates how a NeoCortexApi's Spatial Pooler (SP) can learn spatial patterns. The SP learns by setting a set of input values by iteratively processing them until it achieves stable

representations for all inputs. Here, in the existing experiment to build a slice of cortex 1024 mini columns used with 200 input bits. Also, a set of configuring parameter used for this learning process as shown in figure *.

In the current Spatial Pattern Learning algorithm, at first it is initialzing the necessary values of “HtmConfig” and “Encoder”. After this, 100 random input values was created and then “HtmConfig”, “Encoder” and “inputValues” are passed into “var sp = RunExperiment(cfg, encoder, inputValues)” this method which returns column list of every input values by using spatial pooler algorithm.

In spatial pooler algorithm, at first it establishes the connections by “HtmConfig” parameters as well as creates the memory.

The variable isInStableState is set to false because, when the program gives the column list of inputs, that will not be stable, and it need to be in stable state for each inputs. That is why, in spatial pooler algorithm, homeostatic plasticity controller algorithm is used.

HomeostaticPlasticityController extends the default Spatial Pooler algorithm. The purpose of HomeostaticPlasticityController is to set the SP in the new-born stage at the begining of the learning process. In this stage the boosting is very active, but the SP behaves instable. After this stage is over, the HomeostaticPlasticityController will be controlling the learning process of the SP. Once the SDR generated for every input gets stable, the HomeostaticPlasticityController will fire an event that notifies the code that SP is stable now that means isInStableState will set to true.

As we know, in Hierarchical Temporal Memory, it has three layers (Encoder, Spatial Pooler and Temporal Memory). Here the code is for spatial pattern learning more accurately, how actually Spatial Pooler learns patterns when the input is encoded by the encoder. So, in this program, two layers is involved. One is an Encoder, and the other one is Spatial Pooler. This is done by adding the cortexLayer. First cortexLayer will add Encoder then it will add Spatial Pooler by using following code.

It creates the instance of the neo-cortex layer. Algorithms will be performed inside of that layer.

```
CortexLayer<object, object> cortexLayer = new CortexLayer<object, object>("L1");
```

Add encoder as the very first module. This model is connected to sensory input cells that receive the input. Encoder will receive the input and forward the encoded signal to the next module.

```
cortexLayer.HtmModules.Add("encoder", encoder);
```

The next module in the layer is Spatial Pooler. This module will receive the output of the encoder.

```
cortexLayer.HtmModules.Add("sp", sp);
```

After this, the existing algorithm takes 1000 iterations for the learning process. In the learning process, at first each input is encoded by Scalar Encoder Algorithm then for each input, which columns will be activated, that is done by the spatial pooler algorithm and the column list stored in “activeColumns”. For the first 40 iterations, the column list of each input will be unstable and HPC will control these column list. Those inputs which have columns, HPC will try to make them stable and other inputs which don't have columns, HPC will boost them so that every input have activated columns.

After 40 iterations, all input will have column list and HPC will make them stable so that for each input the activation of columns doesn't change. This number of iterations can be modified from the code.

The algorithm HPC setting stable state to true for each input if that input has similarity of 97% column list for consecutive 50 cycles. The existing program checking the boolean variable isInStableState is set to true or not for 5 cycles then breaks the loop. In this project, we have changed these static values by using a certain condition to exit from this 1000 iterations loop which has been described later in this paper.

IV. Investigating the reason for not getting mini-columns list for input 51 to 99

When the program is run, for first 40 iterations, not getting the mini column list for input 51 to input 99. The investigation started by investigating the Homeostatic Plasticity Controller.

```
HomeostaticPlasticityController hpa = new HomeostaticPlasticityController(mem, inputValues.Count * 40,
    (isStable, numPatterns, actColAvg, seenInputs) =>
    {
        // Event should only be fired when entering the stable state.
        // Ideal SP should never enter unstable state after stable state.
        if (isStable == false)
        {
            Debug.WriteLine($"INSTABLE STATE");
            // This should usually not happen.
            isInStableState = false;
        }
        else
        {
            Debug.WriteLine($"STABLE STATE");
            // Here you can perform any action if required.
            isInStableState = true;
        }
    });
```

At first, it was considered that the problem might lie in the value passed, 'inputValues.Count * 40'. Reducing the value from 40 to 30 would eliminate the mini column list for the initial 30 cycles. Further reduction of the value to 1 would generate all the inputs mini column list from iteration 1 onwards. However, running the program without the Homeostatic Plasticity Controller did not resolve the issue. Hence, it became evident that the problem did not stem from the Homeostatic Plasticity Controller boosting the input’s column for a specific number of iterations, which is currently set at 40 in the program. Thus, reducing the number of iterations is not a logical solution.

Subsequently, the investigation delved into the Spatial Pooler algorithm. There was a consideration that perhaps input 51 to input 99 were not inhibiting the mini columns,

prompting scrutiny of inhibition algorithms. The program employs a local inhibition algorithm for this purpose.

```
public virtual int[] InhibitColumnsLocal(Connections c, double[] overlaps, double density)
{
   return InhibitColumnsLocalOriginal(c, overlaps, density);
   //return InhibitColumnsLocalNewApproach(c, overlaps);
   //return inhibitColumnsLocalNewApproach2(c, overlaps, density);
   //return inhibitColumnsLocalNewApproach3(c, overlaps, density);
   //return InhibitColumnsLocalNewApproach11(c, overlaps, density);
   //return InhibitColumnsLocalNew(c, overlaps, density);
}
```

In this program, there are in total 6 different local inhibition algorithms present. Tried all the algorithms one by one.

- **inhibitColumnsLocalNewApproach(c, overlaps)** this function doesn't take density into account and gives error when there is no columns for input 51.
- **inhibitColumnsLocalNewApproach2(c, overlaps, density)** this function generates all the columns for input 0 from the beginning. For the first few inputs, it generates too many columns and after few inputs, the other input's columns are low and from this function the columns SDR list are not coming from input 51 to input 99. After cycle 41, it generates a lot of columns instead of 0.02*numColumns.
- **inhibitColumnsLocalNewApproach3(c, overlaps, density)** this function doesn't generate any columns for the inputs.
- **inhibitColumnsLocalNewApproach11(c, overlaps, density)** this function generates columns from the beginning for all the inputs, but the main problem is, the columns SDR list length is also a lot instead of 0.02*numColumns.
- **inhibitColumnsLocalNew(c, overlaps, density)** it also behaves like this function of **inhibitColumnsLocalNewApproach2(c, overlaps, density)**.

From our findings, the best one is **InhibitColumnsLocalOriginal(c, overlaps, density)** which is already implemented, because it generates 0.02*numColumns length of columns per input for all the inputs after 40 cycles. But the problem of not generating the columns for input 51 to input 99 for the first 40 cycles was still there.

Upon thorough investigation of all the algorithms, it became apparent that there was insufficient overlap occurring for winning columns associated with inputs 51 to 99. The inhibition algorithm functions in a manner where it inhibits columns only when there is a minimum of one column for each input. Consequently, if there are no columns for a particular input, inhibition for that input does not occur. This observation suggests that inputs failing to generate columns lack sufficient strength. Hence, the issue does not lie within the inhibition algorithm itself.

So, started to investigate the function where the boosting of columns with low overlap happening. In the function, at first, getting the weak columns by comparing overlap duty cycle and minimum percent overlap duty cycle. If overlap duty is less than minimum percent overlap duty cycle, then weak columns are detected. For each weak column, it retrieves the corresponding Column object and its associated Pool, adjusting the permanences of synapses to enhance overlap. This adjustment is performed by increasing the synaptic permanences using a specified increment value. Finally, the function updates the permanences for the column's sparse potential synapses, effectively boosting the responsiveness of these columns to input patterns.

```
/// <summary>
      /// This method increases the permanence values of synapses of columns whose
      /// overlap level is too low. Such columns are identified by having an
      /// overlap duty cycle (activation frequency) that drops too much below those of their peers.
      /// The permanence values for such columns are increased.
      /// </summary>
      /// <param name="c"></param>
public virtual void BoostColsWithLowOverlap(Connections c)
{
   // Get columns with too low overlap.
   var weakColumns = c.Memory.Get1DIndexes().Where(i => c.HtmConfig.OverlapDutyCycles[i] < c.HtmConfig.MinOverlapDutyCycles[i]).ToArray();

   for (int i = 0; i < weakColumns.Length; i++)
   {
      Column col = c.GetColumn(weakColumns[i]);

      Pool pool = col.ProximalDendrite.RFPool;
      double[] perm = pool.GetSparsePermanences();

ArrayUtils.RaiseValuesBy(c.HtmConfig.SynPermBelowStimulusInc, perm);
      int[] indexes = pool.GetSparsePotential();


col.UpdatePermanencesForColumnSparse(c.HtmConfig, perm, indexes, true);
      //UpdatePermanencesForColumnSparse(c, perm, col, indexes, true);
   }
}
```

At first, maybe the boosting algorithm has the problem by not boosting the weak columns enough. the following equation was tried for boosting the weak columns [5]:

$$b_i = e^{-\beta(\bar{a}_i(t) - \langle \bar{a}_i(t) \rangle)} \quad (1)$$

Here, b_i is the boosting signal, β refers to boosting strength, $\bar{a}i(t)$ is the OverlapDutyCycle[i] and $<\bar{a}i(t)>$ refers to MinimumOverlapDutyCycles[i]. The code of this equation is given below:

```
public virtual void BoostColsWithLowOverlap2(Connections c)
{
   // Get columns with too low overlap.
   var weakColumns = c.Memory.Get1DIndexes().Where(i =>
c.HtmConfig.OverlapDutyCycles[i] <
c.HtmConfig.MinOverlapDutyCycles[i]).ToArray();

   foreach (var weakColumnIndex in weakColumns)
   {
      Column col = c.GetColumn(weakColumnIndex);

      // Get the boosting signal for the weak column based on its
overlap duty cycle.
      double boostingSignal = CalculateBoostingSignal(col,
c.HtmConfig);

      // Adjust the synaptic connections (permanences) associated
with the weak column.
      AdjustPermanencesForColumn(col,
c.HtmConfig.SynPermBelowStimulusInc, boostingSignal,c);
   }
}

private double CalculateBoostingSignal(Column col, HtmConfig
config)
{
   double overlapDutyCycle =
config.OverlapDutyCycles[col.Index];
   double minOverlapDutyCycle =
config.MinOverlapDutyCycles[col.Index];
   double BoostBeta = 100;
   // Calculate the boosting signal based on the difference between
the current overlap duty cycle and the minimum overlap duty cycle.
   //double boostingSignal = 1 / (1 + Math.Exp(-BoostAlpha *
(overlapDutyCycle - minOverlapDutyCycle)));
   double boostingSignal = Math.Exp(-BoostBeta *
(overlapDutyCycle - minOverlapDutyCycle));

   return boostingSignal;
}

private void AdjustPermanencesForColumn(Column col, double
synPermBelowStimulusInc, double boostingSignal, Connections c)
{
   Pool pool = col.ProximalDendrite.RFPool;
   double[] permanences = pool.GetSparsePermanences();

   // Boost the permanences associated with the weak column.
   for (int i = 0; i < permanences.Length; i++)
   {
      permanences[i] += synPermBelowStimulusInc *
boostingSignal;
   }

   // Update the permanences for the column.
   col.UpdatePermanencesForColumnSparse(c.HtmConfig,
permanences, pool.GetSparsePotential(), true);
}
```

After its implementation, the program was executed, yet no significant changes were observed in boosting low overlap columns.

Subsequently, a consideration arose that perhaps accelerating the reduction of weak columns could facilitate the emergence of mini columns for inputs 51 to 99.

Therefore, an investigation commenced to understand the mechanism behind the generation of weak columns. It was found that the weak columns are generated in the following line of code:

```
// Get columns with too low overlap.
var weakColumns = c.Memory.Get1DIndexes().Where(i =>
c.HtmConfig.OverlapDutyCycles[i] <
c.HtmConfig.MinOverlapDutyCycles[i]).ToArray();
```

From this line of code, two things are responsible for generating weak columns. One is Overlap Duty Cycles and the other one is Minimum Overlap Duty Cycles. Our assumption is that if we can increase the value of OverlapDutyCycles[i] as well as decrease the value of MinOverlapDutyCycles[i] then the weak columns will be reduced faster than before.

So, tried to figure out how Overlap Duty Cycles and Minimum Percent Overlap Duty Cycles are calculated. Overlap Duty Cycles calculated by this equation:

$$dutyCycle = \frac{(\text{period}-1)*\text{dutyCycle}+\text{newValue}}{\text{period}} \quad (2)$$

Here, the value period is equal to duty cycle period which value is coming from HtmConfig and it was initially equal to 100. If we make this duty cycle period equal to 1000 then the value of dutyCycle does not change that much but it increases the value a little bit and by increasing a little, it affects the reducing of weak columns as well as making stability faster.

After this, investigate the function where Minimum Percent Overlap Duty Cycle is calculated. Here, found that Minimum Overlap Duty Cycle is calculated by following equation:

$$MinimumOverlapDutyCycle = maxOverlapDuty * \text{MinimumPercent OverlapDutyCycles} \quad (3)$$

In this equation, the value of Minimum Percent Overlap Duty Cycles is coming from HtmConfig also. Initially, the value was 1. Its value was too big in comparison to the program because even if for the first 40 iterations of boosting through Homeostatic Plasticity Controller, it can't boost the weak columns as a result after 40 iterations, getting mini columns for input 51 to input 99 without reducing the weak columns. Because even if reducing the number of iterations or increasing the number of iterations, will get the same mini columns after defined iterations. Decreasing the value of Minimum Percent Overlap Duty Cycles may lead to the assumption that mini columns will emerge for inputs 51 to 99, as it will diminish the weak columns. Additionally, for the initial 40 iterations, the Homeostatic Plasticity Controller can boost the columns, contributing to program stability and faster execution overall.

So, tried to change the values of duty cycle period and minimum percent overlap duty cycle so that we can see by which value we are getting the mini columns for input 51 to input 99 as well as stability faster.

For DutyCyclePeriod = 1000	numColumns 1024	maxBoost= 5
	minOctOverlapCycles	All SDR Available
minOctOverlapCycles	0.1	2
minOctOverlapCycles	0.2	2
minOctOverlapCycles	0.3	2
minOctOverlapCycles	0.4	2
minOctOverlapCycles	0.45	2
minOctOverlapCycles	0.5	2
minOctOverlapCycles	0.6	4
minOctOverlapCycles	0.7	4
minOctOverlapCycles	0.8	40
minOctOverlapCycles	0.9	40
minOctOverlapCycles	1	41

Figure 2: Changing the values of minOctOverlapCycles

The figure indicates that reducing the value of minOctOverlapCycles (minimum percent overlap duty cycles) will result in obtaining mini column values between cycle 2 and cycle 3. A graph representation will offer a clearer understanding of this relationship.

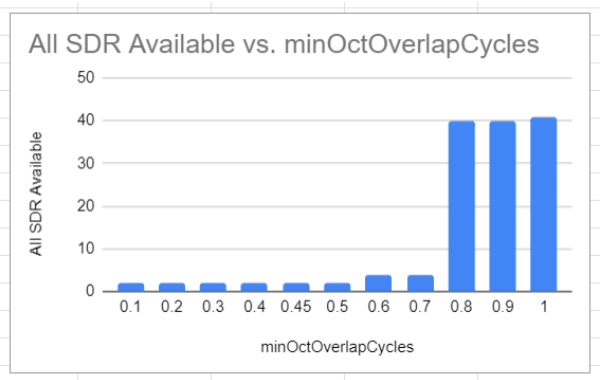

Figure 3: Graph of All SDR Available vs minOctOverlapCycles

V. METHODOLOGY

The project Spatial Pattern Learning is developed using C# .Net Core in Microsoft Visual Studio 2022 IDE (Integrated Development Environment) is used as a new model to understand and implement the functioning of Spatial Pattern Learning.

Spatial Pattern Learning particularly refers to the process of learning patterns of activation of mini columns in a spatially distributed manner. In Spatial Pattern Learning, the spatial pooler learns which input patterns are commonly associated and should be represented together. It adjusts the connections between input bits and columns of neurons based on the statistical properties of the input patterns it receives over time.

In the previous experiment, the learning phase runs for 1000 iterations even if the program gets stable. In this new experiment, we have defined the criteria for how long data should be learned and when the program should exit from the loop after matching certain criteria.

In this new experiment, after the program gets stable, comparing the mini column list for each cycle and determining whether the mini column list changed or not and looking for 100 consecutive iterations. If there is any change in mini column list, look for another 100 consecutive iterations and if there is no change in mini column list for 100 consecutive iterations, break the loop and print the dictionary for last 100 iterations in the end.

Checked 100 iterations because the variable isInStableState is true, after 50 or 60 and sometimes 70 iterations, it turns false again that's why choose 100 iterations.

In each iteration, showing how many iterations are same for mini column of each input as well as showing when an input is getting stable and which iteration it gets stable. After the program gets stable, showing how many iterations are stable. Generating bit maps for each input for better understanding of how mini column list is changing in each iteration and visualization of stability in mini column list for each input.

First, after the program gets stable, store the mini column list for each input in the dictionary.

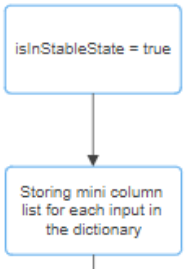

Figure 4: Storing the mini columns list for each input in the dictionary.

Before comparing the mini column list, checking whether all the input has mini column list or not.

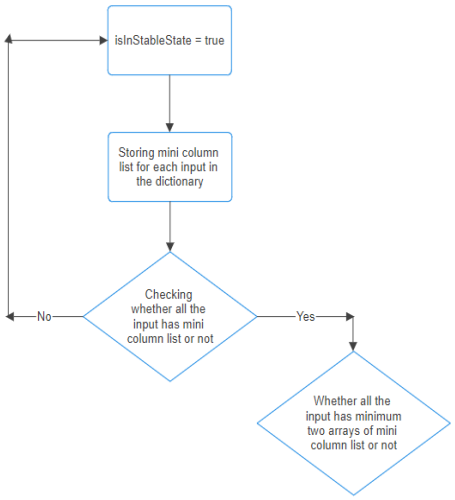

Figure 5: Checking if an input has mini columns or not.

If all the input has a mini column list, then check whether all the input has minimum two arrays of mini columns or not because for comparing, need minimum two arrays.

If all the input has a minimum of two arrays, compare the arrays. So, if all the input has a minimum of two arrays as well as SDRs then first, take the last two cycles SDR (mini column list) values of that input and check whether the length of the arrays is same or not. If not, then there will be a Boolean variable which will be responsible for making the

decision whether an input has been same for last two cycles or not. So, if the length of two arrays is not same then that Boolean variable will be set to false. If the length of two arrays is same, then compare each value of those two arrays and check whether all the values are same or not. If same, then the Boolean variable will set to true and if not satisfied then the Boolean variable will set to false.

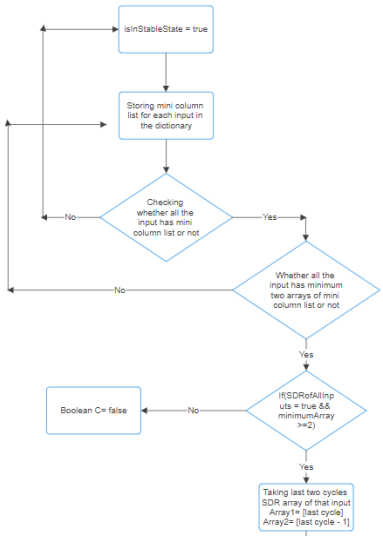

Figure 6: Taking the last two cycles SDR for comparing.

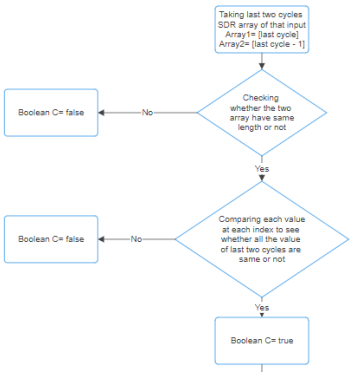

Figure 7: Getting Boolean C is true or false.

If Boolean variable is true, that means the consecutive two cycles have same values without any changes. So, counting for cycles will be increased because this counting is needed to see whether a certain number of cycles have same mini column list or not. If Boolean variable gets false, then counting for cycle will set to zero again because needed consecutive minimum number of cycles must be matched for breaking the loop.

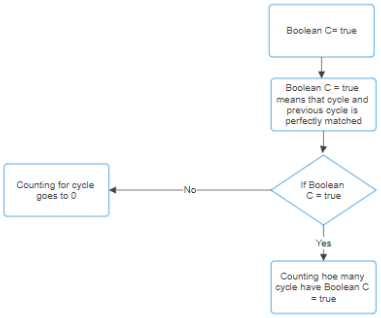

Figure 8: Counting the cycle number if Boolean C is true.

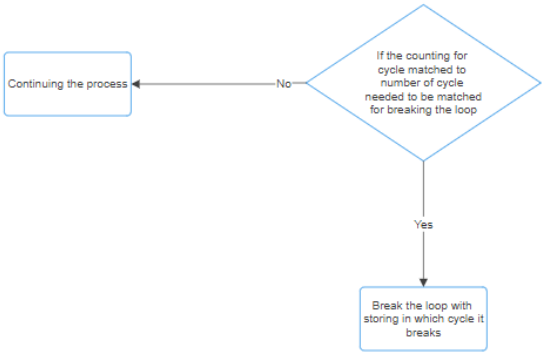

Figure 9: Breaking the loop if condition met.

Further statistical information:

Firstproviding which input is getting stable at which cycle. For this, checking whether an inputs recent iteration to its previous iteration's similarity is 100% or not. So, if similarity is 100% for consecutive 50 iteration, then state that input is stable as well as storing the iteration number to show in which iteration the input gets stable.

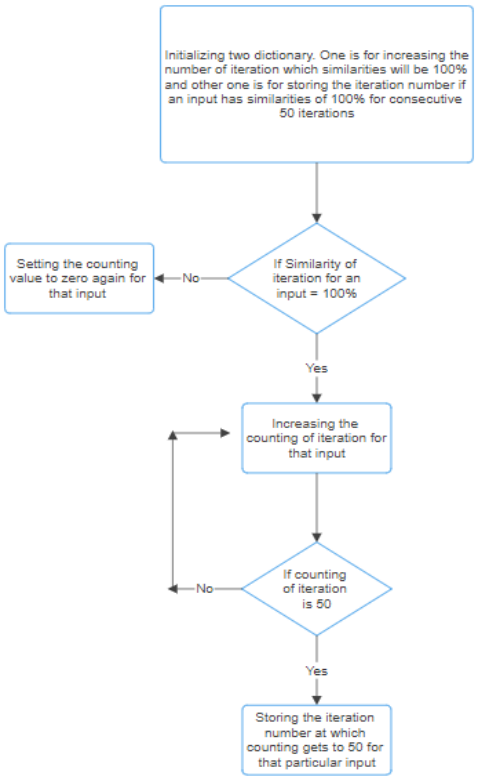

Figure 10: Increasing the number of same iterations of an input and storing the cycle number if counting of iteration gets 50.

Generating Bit Maps to see how actually mini column list is changing throughout the iterations for an input. For this, first created an array of size 1024 because column size is 1024. Then initialize every index with value 0 and then turns 0 to 1 for active columns (mini column list). Then made this 1D array into 2D array. After this, creating a directory dynamically called "Outputs" where dynamically created folders for input 0 to input 99. In each folder the Bit Map is

saved as an image to see the changes in mini column list in each iteration.

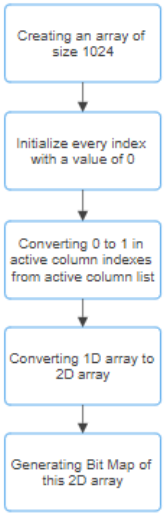

Figure 11: Generating BitMaps for input.

After this, create a dictionary to save the final SDR (mini columns) for the inputs. Then, create a dictionary to store the inputs will be activated for a column for example column 0 will be activated for input 93, input 94 and input 95.

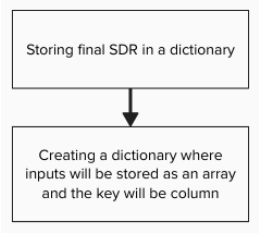

Figure 12: A column will be activated for which inputs.

Generating bitmaps of connected input bits for a column.

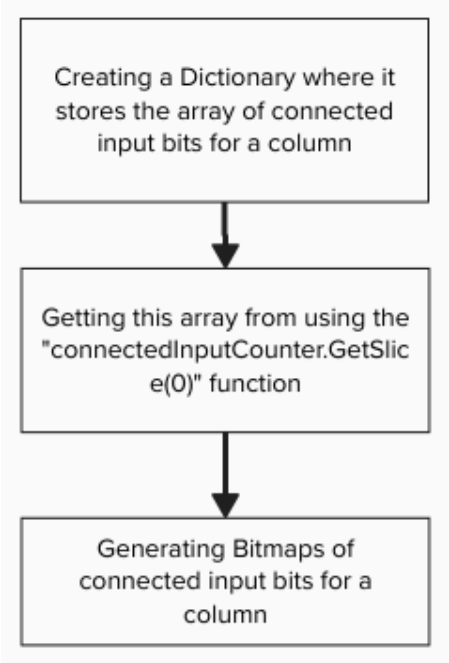

Figure 13: Flowchart of Generating bitmaps of connected input bits for a column.

VI. RESULT

As the investigation shows, if the value of minimum percent overlap duty cycles decreases then will get the mini columns from the starting of the iterations as well as increasing the value of duty cycle period. So, setting the value of duty cycle period to 1000 and minimum percent overlap duty cycles to 0.45 gives the best output. As getting all the values from cycle 3 and getting stability from cycle 384 which almost 70 cycles less than the previous implementation.

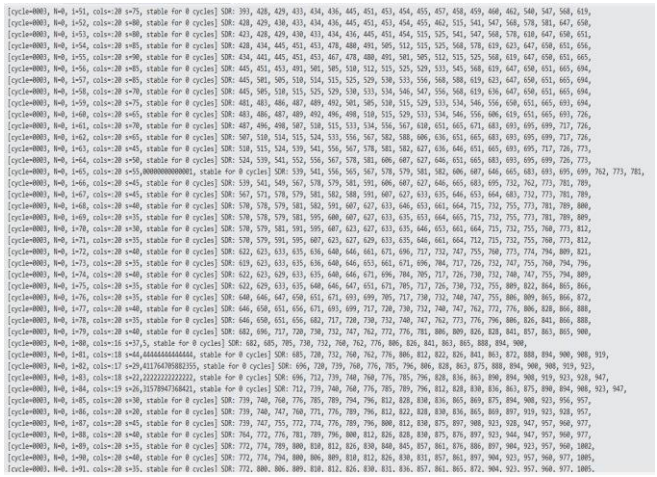

Figure 14: Getting mini columns for all inputs.

This picture (Figure 14) shows getting mini columns for all the inputs from cycle 3.

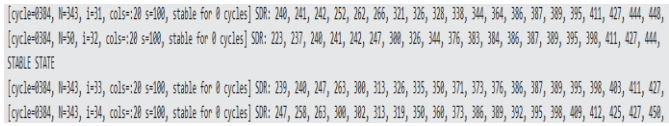

Figure 15: Stability on cycle number 384.

This image (Figure 15) shows stability on cycle number 384.

After getting the stability on 384 cycle that means the variable isInStableState = true then storing the column list of each input in the dictionary. The dictionary is storing column list like this:

input:0 cycle 384:7, 24, 29, 43, 46, 59, 62, 70, 102, 112, 114, 116, 118, 148, 154, 155, 933, 953, 982, 1012, cycle 383:7, 24, 29, 43, 46, 59, 62, 70, 102, 112, 114, 116, 118, 148, 154, 155, 933, 953, 982, 1012,

input:1 cycle 384:7, 12, 24, 29, 37, 39, 46, 54, 59, 95, 102, 118, 125, 148, 154, 155, 953, 961, 998, 1012, cycle 383:7, 12, 24, 29, 37, 39, 46, 54, 59, 95, 102, 118, 125, 148, 154, 155, 953, 961, 998, 1012,

After breaking the cycle, printing the dictionary at the end of the program.

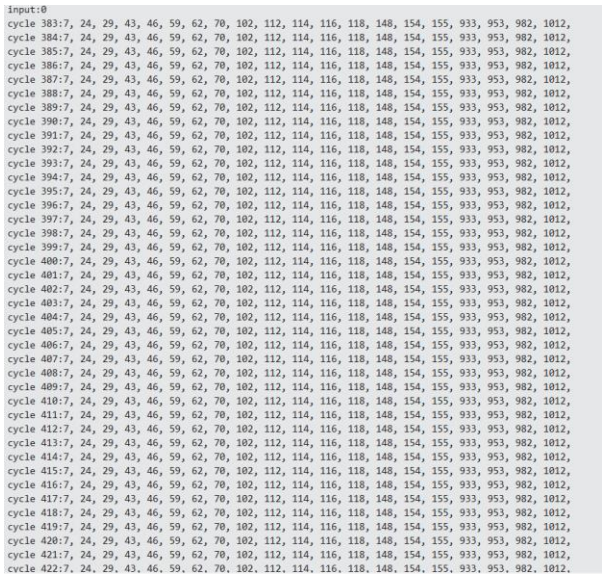

Figure 16.1: Printing the dictionary for input (last 100 cycles)

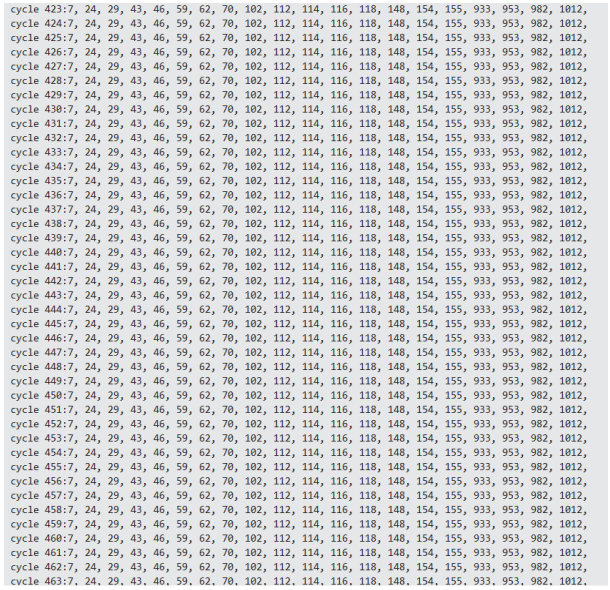

Figure 16.2: Printing the dictionary for input (last 100 cycles)

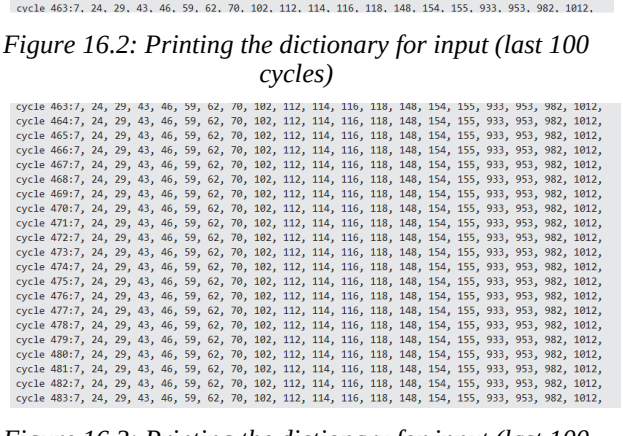

Figure 16.3: Printing the dictionary for input (last 100 cycles)

These three images are for the last 100 cycles mini columns values of input 0.

Finally, the all the inputs stable mini column list is printed.

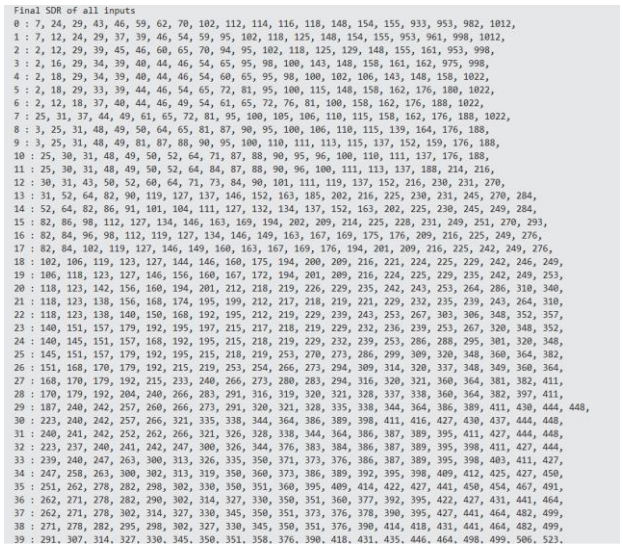

Figure 17.1: All the inputs SDR (mini columns)

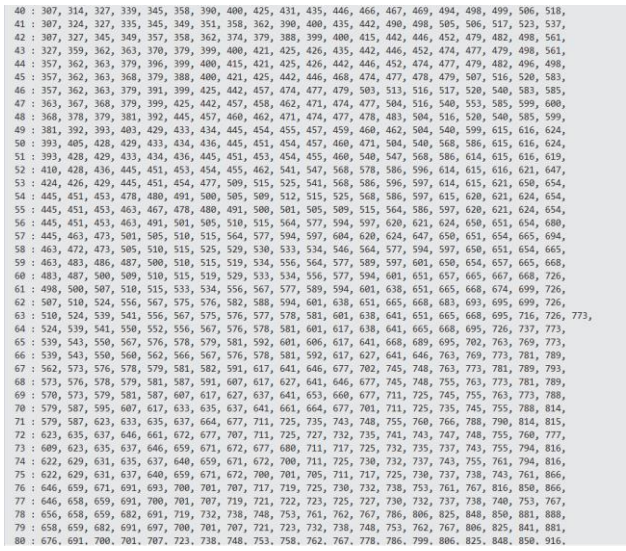

Figure 17.2: All the inputs SDR (mini columns)

Figure 17.3: All the inputs SDR (mini columns)

Previous implementation's output was:

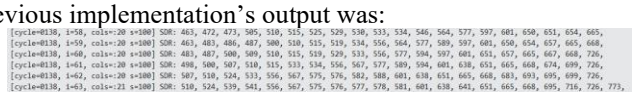

Figure 18: Previous output of the implementation Spatial pattern Learning

After implementing new Spatial Pattern Learning Experiment, the output is:

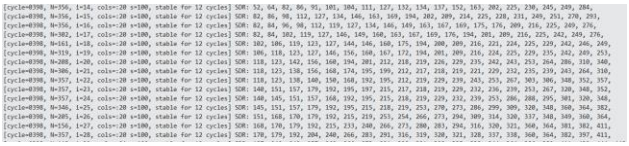

Figure 19: After implementing new Spatial Pattern Learning

Here, N = how many iterations the SDR for the input was not changed. After isInStableState is true, counting and showing how many cycles are stable for all the inputs.

After each cycle, show whether a input is stable or not and if a input is stable then at which cycle it gets stable.

```
input 0: Stable Input and stable on 102 cycle
input 1: Stable Input and stable on 134 cycle
input 2: Stable Input and stable on 93 cycle
input 3: Stable Input and stable on 94 cycle
input 4: Stable Input and stable on 92 cycle
input 5: Stable Input and stable on 92 cycle
input 6: Stable Input and stable on 93 cycle
input 7: Stable Input and stable on 92 cycle
input 8: Stable Input and stable on 92 cycle
input 9: Stable Input and stable on 138 cycle
input 10: Stable Input and stable on 100 cycle
input 11: Stable Input and stable on 116 cycle
input 12: Stable Input and stable on 92 cycle
input 13: Stable Input and stable on 92 cycle
input 14: Stable Input and stable on 92 cycle
input 15: Stable Input and stable on 92 cycle
input 16: Stable Input and stable on 92 cycle
input 17: Stable Input and stable on 146 cycle
input 18: Stable Input and stable on 287 cycle
input 19: Stable Input and stable on 129 cycle
input 20: Stable Input and stable on 240 cycle
input 21: Stable Input and stable on 142 cycle
input 22: Stable Input and stable on 91 cycle
input 23: Stable Input and stable on 91 cycle
input 24: Stable Input and stable on 91 cycle
input 25: Stable Input and stable on 102 cycle
input 26: Stable Input and stable on 243 cycle
input 27: Stable Input and stable on 292 cycle
input 28: Stable Input and stable on 91 cycle
input 29: Stable Input and stable on 330 cycle
input 30: Stable Input and stable on 151 cycle
input 31: Stable Input and stable on 91 cycle
input 32: Stable Input and stable on 384 cycle
input 33: Stable Input and stable on 91 cycle
input 34: Stable Input and stable on 91 cycle
input 35: Stable Input and stable on 91 cycle
input 36: Stable Input and stable on 113 cycle
input 37: Stable Input and stable on 192 cycle
input 38: Stable Input and stable on 270 cycle
input 39: Stable Input and stable on 91 cycle
input 40: Stable Input and stable on 226 cycle
```

Figure 20.1: Stability cycle number for all inputs

```
input 41: Stable Input and stable on 91 cycle
input 42: Stable Input and stable on 91 cycle
input 43: Stable Input and stable on 91 cycle
input 44: Stable Input and stable on 91 cycle
input 45: Stable Input and stable on 114 cycle
input 46: Stable Input and stable on 91 cycle
input 47: Stable Input and stable on 91 cycle
input 48: Stable Input and stable on 91 cycle
input 49: Stable Input and stable on 192 cycle
input 50: Stable Input and stable on 91 cycle
input 51: Stable Input and stable on 91 cycle
input 52: Stable Input and stable on 144 cycle
input 53: Stable Input and stable on 91 cycle
input 54: Stable Input and stable on 91 cycle
input 55: Stable Input and stable on 197 cycle
input 56: Stable Input and stable on 91 cycle
input 57: Stable Input and stable on 186 cycle
input 58: Stable Input and stable on 327 cycle
input 59: Stable Input and stable on 152 cycle
input 60: Stable Input and stable on 292 cycle
input 61: Stable Input and stable on 151 cycle
input 62: Stable Input and stable on 91 cycle
input 63: Stable Input and stable on 91 cycle
input 64: Stable Input and stable on 91 cycle
input 65: Stable Input and stable on 292 cycle
input 66: Stable Input and stable on 91 cycle
input 67: Stable Input and stable on 151 cycle
input 68: Stable Input and stable on 158 cycle
input 69: Stable Input and stable on 231 cycle
input 70: Stable Input and stable on 231 cycle
input 71: Stable Input and stable on 372 cycle
input 72: Stable Input and stable on 292 cycle
input 73: Stable Input and stable on 91 cycle
input 74: Stable Input and stable on 153 cycle
input 75: Stable Input and stable on 153 cycle
input 76: Stable Input and stable on 372 cycle
input 77: Stable Input and stable on 153 cycle
input 78: Stable Input and stable on 91 cycle
input 79: Stable Input and stable on 91 cycle
input 80: Stable Input and stable on 98 cycle
input 81: Stable Input and stable on 231 cycle
```

Figure 20.2: Stability cycle number for all inputs

```
input 81: Stable Input and stable on 231 cycle
input 82: Stable Input and stable on 151 cycle
input 83: Stable Input and stable on 151 cycle
input 84: Stable Input and stable on 91 cycle
input 85: Stable Input and stable on 91 cycle
input 86: Stable Input and stable on 91 cycle
input 87: Stable Input and stable on 151 cycle
input 88: Stable Input and stable on 104 cycle
input 89: Stable Input and stable on 91 cycle
input 90: Stable Input and stable on 91 cycle
input 91: Stable Input and stable on 91 cycle
input 92: Stable Input and stable on 91 cycle
input 93: Stable Input and stable on 91 cycle
input 94: Stable Input and stable on 231 cycle
input 95: Stable Input and stable on 91 cycle
input 96: Stable Input and stable on 91 cycle
input 97: Stable Input and stable on 91 cycle
input 98: Stable Input and stable on 189 cycle
input 99: Stable Input and stable on 129 cycle
```

Figure 20.3: Stability cycle number for all inputs.

When input is not getting stable:

```
input 70: Stable Input and stable on 231 cycle
input 71: Not stable Input
input 72: Stable Input and stable on 292 cycle
input 73: Stable Input and stable on 91 cycle
input 74: Stable Input and stable on 153 cycle
```

Figure 21: If a cycle is not stable.

After each cycle, show by percentage how many inputs are stable for that cycle.

When a cycle is 100% stable:

```
input 97: Stable Input and stable on 91 cycle
input 98: Stable Input and stable on 189 cycle
input 99: Stable Input and stable on 129 cycle
100% stable
```

Figure 22: If a cycle is 100% stable.

When a cycle is not 100% stable:

```
input 97: Stable Input and stable on 91 cycle
input 98: Stable Input and stable on 189 cycle
input 99: Stable Input and stable on 129 cycle
97% stable
```

Figure 23: If a cycle is not 100% stable.

Generating BitMaps for better understanding of changing of mini columns for each cycle of an input.

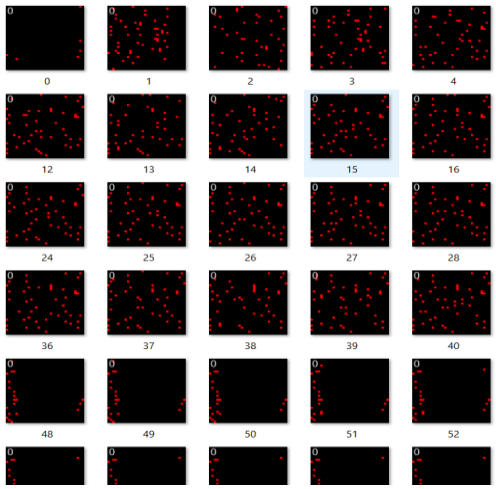

Figure 24: BitMaps of an input for each cycle.

Here's a YouTube video link for better understanding.

Showing a column will be activated for which inputs. Here, only showing first 30 columns output but in the code output there will be all the columns.

```
0 : 93, 94, 95,
1 :
2 : 2, 3, 4, 5, 6,
3 : 8, 9,
4 : 92,
5 : 91, 92, 93,
6 :
7 : 0, 1, 96, 98, 99,
8 : 92, 93, 94, 95, 96, 97,
9 :
10 : 92, 93,
11 :
12 : 1, 2, 6,
13 :
14 : 96, 97, 98, 99,
15 :
16 : 3,
17 : 95,
18 : 4, 5, 6,
19 :
20 : 92, 93, 94, 95, 96, 97,
21 :
22 :
23 : 96,
24 : 0, 1,
25 : 7, 8, 9, 10, 11,
26 :
27 :
28 :
29 : 0, 1, 2, 3, 4, 5,
30 : 10, 11, 12,
```

Figure 25: Output of a column will be activated for which inputs.

First, showing the output of connected input bits for a column. Here, also showing the output for only first 30 columns but in the code output there will be output of all the columns.

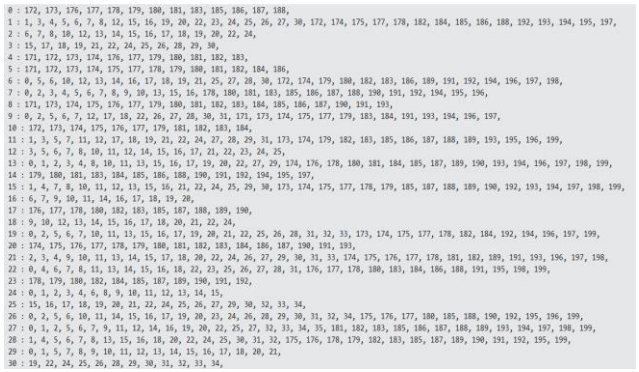

Figure 26: Connected input bits for a column.

Then generating bit maps of connected input bits for a column for better representation.

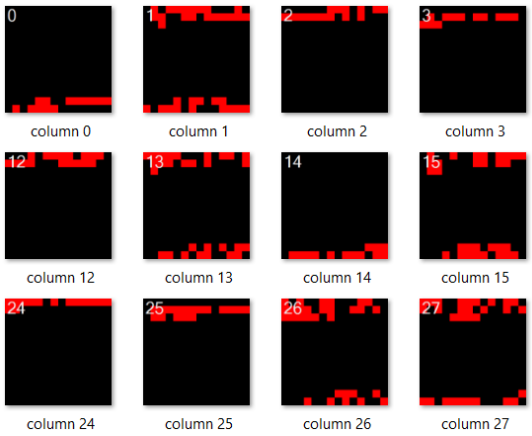

Figure 27: Generating bit maps of connected input bits for a column.

Here is other statistical information about Spatial Pattern Learning:

First, changing the parameter value of "minOctOverlapCycle" because by changing this parameter, show that at which cycle where all the inputs mini columns are present.

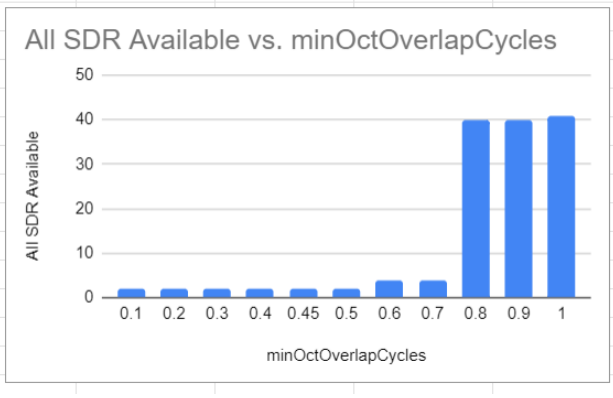

Figure 28: All SDR Available vs minOctOverlapCycles.

Changing the parameter W (15,21,31,61,81,91) and getting the stability like this:

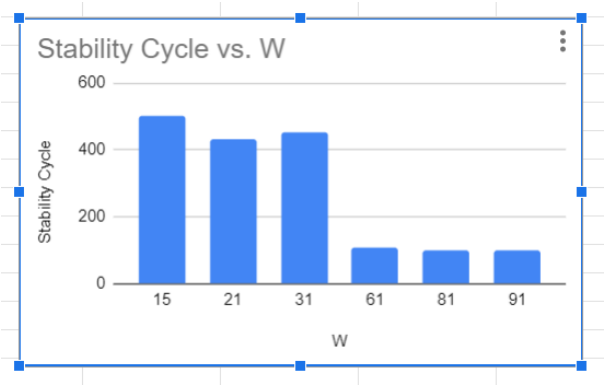

Figure 29: Stability Cycle vs W (Population)

Changing the values of NumColumns (512, 1024, 2048) and getting the stability of cycles like this:

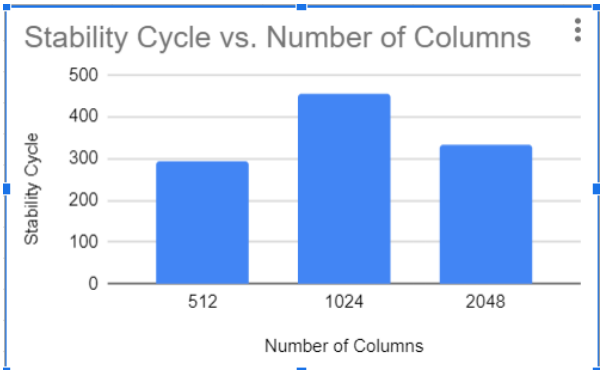

Figure 30: Stability Cycle vs NumColumns

Changes other parameters also such as Stimulus Threshold, Activation Threshold and maxBoost but didn't see any changes that much in stability.

At the end, did some unit tests for the functions that were created and used. Not all functions have unit tests, those functions that were returning something only these functions were tested, and all test passed successfully.

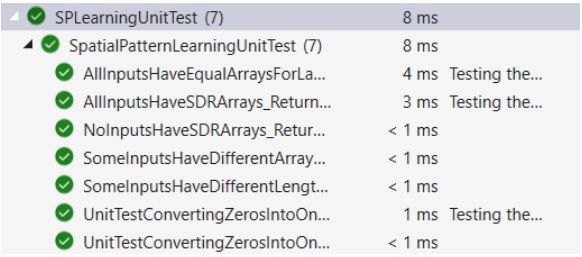

Figure 31: Unit test passed successfully.

VII. Conclusion

The SpatialPatternLearning experiment has provided invaluable insights into the capabilities and dynamics of spatial pattern learning within Spatial pooler. Through meticulous experimentation and analysis, demonstrated the efficacy of the program. Our project was to implement new spatial pattern learning. At first, investigated why first 40 cycles don't give mini columns list for inputs 51 to input 99. This is happening because of the boosting of low overlap columns. There increasing of overlap duty cycles value and decreasing the value of minimum overlap cycles solved the problem. After this, the main task of the program was, it should exit from the loop by a certain condition after the variable isInStableState is set to true which implemented successfully in the program. It ensures that after the variable isInStableState is true, then it's stability will not change by checking consecutive 100 cycles. At the end, the dictionary is written. It provides, how many iterations the SDR for the input was not changes. After isInStableState is true, counting and showing how many cycles are stable. After each cycle, whether a input is stable or not and if a input is stable then at which it gets stable as well as showing by percentage that how many inputs are stable for that cycle. Generating bitmaps for each input of every cycle to represent how actually the SDRs are changing through cycles for each input. Provided other statistical information, for example changing the parameters, how the program effects stability as well as at which cycle all the input have mini column list by showing graphs.

VIII. References

[1] J. Hawkins, S. Ahmad and D. Dubinsky, "Cortical learning algorithm and hierarchical temporal memory.," [Online]. Available: http://numenta.org/resources/HTM_CorticalLearning Algorithms.pdf.

[2] Ahmad, Subutai and a. J. Hawkins, "Properties of sparse distributed representations and their application to hierarchical temporal memory," 2015. [Online]. Available: arXiv preprint arXiv:1503.07469.

[3] J. Hawkins and S. A. a. D. Dubinsky, "Hierarchical Temporal Memory including HTM Cortical Learning Algorithms," Numenta, December 2011. [Online].

[4] "Numenta platform for intelligent computing (nupic)," [Online]. Available: http://numenta.org/nupic.html, commit bd8e6a9.

[5] Cui, Yuwei, S. Ahmad and a. J. Hawkins, "The HTM spatial pooler—A neocortical algorithm for online sparse distributed coding.," Frontiers in computational neuroscience 11 : 272195, 2017.

[6] Mnatzaganian, James, E. Fokoué and a. D. Kudithipudi, "A mathematical formalization of hierarchical temporal memory's spatial pooler.," *Frontiers in Robotics and AI : 81,* 2017.