Investigation of the HTM Layer Stability

Masters in Information Technology
Individual Project
Sahana Prasad
1324329
sahana.prasad@stud.fra-uas.de

***Abstract* — The main objective of this project is to describe Spatial Pooler (SP) algorithm in Hierarchical Temporal Memory (HTM). SP is a very important component of HTM which is trained for several iterations in order to generate effective Sparse Distributed Representation (SDR) for the given set of input sequences. Initially, during an observation the created SDRs were being forgotten as training progressed. An internal boosting mechanism led by homeostatic plasticity process caused this instability in learning method. This is because different set of SDRs were being produced after unspecified number of iterations. New approach has been discussed in this project which introduces Homeostatic Plasticity Controller (HPC) into SP algorithm that deactivates the boosting mechanism once SP becomes stable in New-born stage.**

Keywords— Hierarchical Temporal Memory (HTM), Spatial Pooler (SP), Temporal Memory (TM), Homeostatic Plasticity Controller (HPC),* Sparse Distributed Representation *(SDR), Sequence Learning, HTM Classifier.

I. Introduction

HTM is an unsupervised learning neural network model inspired by the human brain and based on the computational principles of single cortical columns in the Neocortex [1]. HTM has the highest forecast accuracy and precision. In addition, HTM demonstrates a number of additional appealing qualities for real-world sequence learning, such as rapid adjustment to changes in the information stream, sensor noise robustness, and fault tolerance. In comparison to other neural models, these features make HTM an excellent contender for sequence learning [1].

The Hierarchical Temporal Memory Cortical Learning Algorithm (HTM CLA) is a type of learning algorithm where neurons are arranged in layers to form regions [1]. Areas, columns, and mini-columns are organized hierarchically and can be linked together to construct more complex networks that perform higher-level cognitive tasks like invariant representations, pattern and sequence recognition, and so on. The HTM CLA is made up of three elements, one of which is an encoder that turns user input into a binary stream of size n, also known as an input vector or array of SP. The second component is SP, which can take an input vector and generate an SDR for it. The third section contains Temporal Memory (TM), which can also refer to the system's cognitive capabilities [1]. The basic goal of TM is to learn any sequence of SDRs that are given as input. The combination of these three parts forms a cortical sublayer capable of memorizing multiple sequences, but memory is limited, as it is in the brain. Building and controlling these SDRs is the most important aspect of HTM [1].

HTM CLA's two main algorithms are the SP and TM algorithms. Sensory inputs are connected to mini-columns in the SP. It's in charge of learning spatial patterns by encoding them into Sparse Distributed Representations. The resulting SDR is then sent into the TM technique, which reflects the encoded spatial pattern [1]. The TM is in charge of memorizing SDR sequences. It monitors and learns the SDRs design from the SP, and then forecasts a new esteem for the relative arrangements based on previous data. HTM CLA at that point receives vectors of dynamic cells from TM and performs advanced cognitive functions such as prediction, inference, and learning [1]. The previous version of the SP was insecure, causing learned patterns to be forgotten and re-learned throughout the learning process. The stable and unstable stable states of the SP alternated. Furthermore, the instability is linked to a single pattern rather than a group of patterns, according to the investigation. The SP can achieve stability of one SDR pattern p1 whereas another SDR pattern p2 can become unstable. All the experiments that depend on spatial pattern recognition requires a stable SP [1]. It can be noted that, the TM method uses the SP's SDRs as an input, and an unstable SP will cause the TM algorithm to forget learned sequences. In this study, the SP's instability is explored, and a SP modification is presented to increase the algorithm's stability.

II. Methodology

A. Overview

The goal of HTM CLA is to allow the machine to perform advanced capabilities and cognitive activities in the same way as the human brain does. This time-based prediction system is designed to forecast future information based on the information that has already been given into it [3]. HTM requires user input in order to memorize and anticipate. A scalar value, date-time string, or an image can be used as input. The encoder is then used to convert the input to SDR, which can then be learned by the HTM classifier [3].

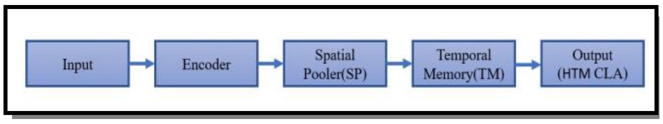


Figure 1: HTM CLA workflow

The SP is an HTM algorithm capable of learning spatial patterns. The SP receives a cluster of bits as an input on a regular basis and converts it to the SDR. TM is another aspect of HTM CLA. TM takes the SDR output from SP as an input. TM is an algorithm that learns how the SP method shapes SDRs and predicts what another input SDR will be [3].

B. Spatial Pooler algorithm

The SP algorithm employs a homeostatic plasticity-inspired column-boosting method. This mechanism is considered to be very important for cortical layer stability [1]. Homeostatic plasticity ensures the functional stability of neuronal networks. It coordinates circuit connection alterations and balances network stimulation and inhibition [1].

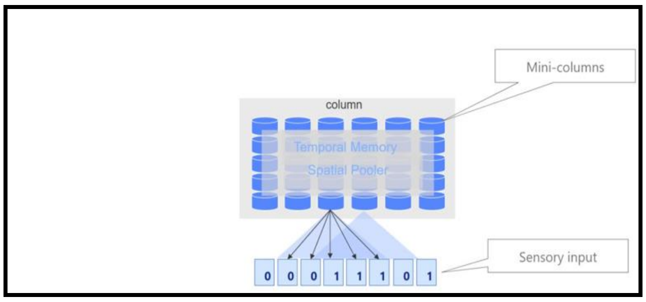


Figure 2: Column with SP and TM [3]

In HTM, the excitation process is directly implemented by the algorithms SP and TM, which set cells to an inactive or predictive state. SP also includes two different inhibition algorithms: There are two types of inhibition: global and local [1]. Cells in the neighbourhood of the currently processing cell are either activated or suppressed using inhibition algorithms. The boosting feature of the SP maintains track of column activity and guarantees that all columns are used consistently across all patterns. As this process is active all of the time, it can increase columns that have previously been learnt SDRs [1].

When this occurs, the SP will temporarily “forget” some previously learnt patterns. If the SP is shown the forgotten pattern again, it will begin to learn it again. For example, if the SDRs given input pattern set remains constant during entire course of the SP's life cycle, then SP is said to be stable [1]. The similarity between all SDRs having same input pattern set is 100 percent because SP is known for learning patterns quickly. Because it does not necessitate a lengthy training process, this behaviour is ideal for real-world applications. The learnt SDR does not vary over time if the similarity is 100 percent. It is important to note that, after certain number of iterations SP tends to forget the learnt SDR and it starts to learn new patterns all over again. So the new set of SDRs are being presented which overlaps with the old SDR values. As the previous SDR values are forgotten completely, there is drop in accuracy from 100% to less than 5%. If the similarity between the inputs is less than 100%, the generated SDRs will be different. This suggests that the SP is unstable [1].

Initially, the learnt patterns were fluctuating between stable and unstable state which is considered to be not so useful. The SP establishes and maintains a steady SDR right from the outset of the learning process (unchanged) [1]. SDR will alter after that until the SP returns to a stable state SP. Usually, cell SDRs which have large number of active columns tend to overlap more than cell SDRs which have smaller active cells [1]. The boosting mechanism in the Neocortex, which is inspired by homeostatic plasticity, solves this problem by enhancing quiet mini-columns and suppressing excessively active mini-columns. Sensory inputs that are regularly related to the cortical layer L4 are used by the SP. This work enhances the SP algorithm and introduces the infant stage of the HTM and SP based on this discovery [1].

C. Spatial Pooler during New-born Stage

The main goal of this research is to add an extra algorithm to SP that does not interfere with the current SP approach in order to stabilize the SP while still allowing it to leverage plasticity. A new component, the Homeostatic Plasticity Controller (HPC), implements the algorithm employed by the extended SP [1]. The controller is "connected" to the SP’s existing implementation so that each iteration's computation along with their corresponding SDRs can be sent to the controller. The controller continues to boost the SP until it achieves a stable condition, which is measured in repeats [1]. The SP is in the so-called New-born stage during this time, and once it hits a stable state, the new algorithm disables the boosting and notifies the application of the status change. The controller keeps track of how many mini-columns are present in the overall pattern. Controller also monitors all mini-columns so that when there are visibly same generated SDRs carrying same number of mini-columns then it means that SP has attained its stability. Once the SP has attained the stability, it will break out of New-born stage and jumps into next operation without any boosting mechanism [1].

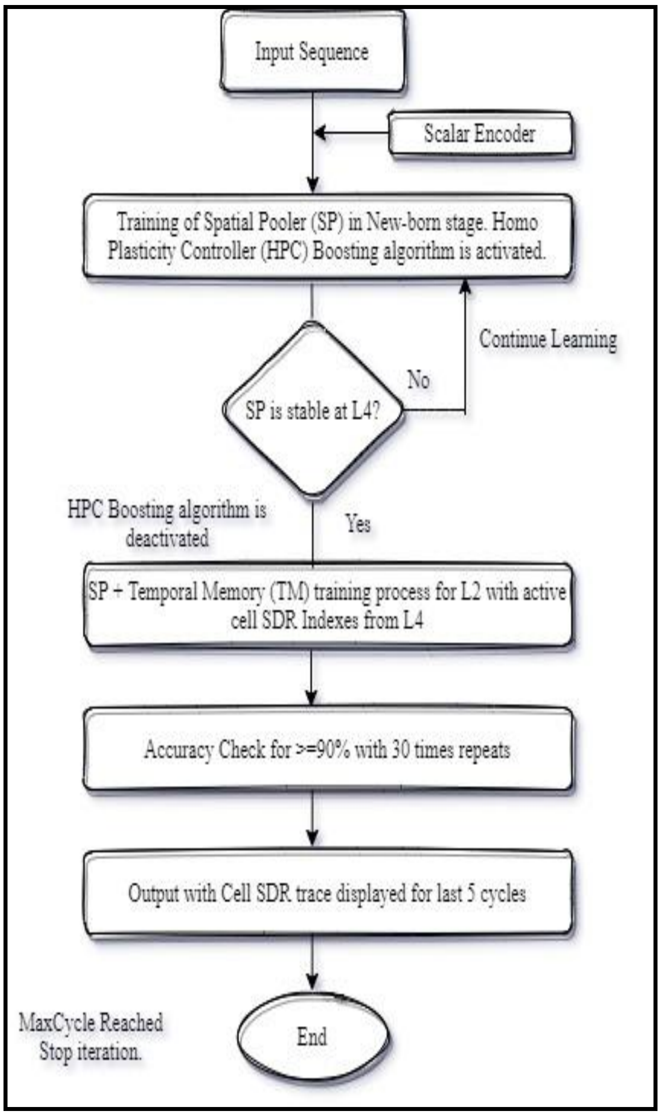

Input Sequence
Scalar Encoder
Training of Spatial Pooler (SP) in New-born stage. Homo Plasticity Controller (HPC) Boosting algorithm is activated.
Continue Learning
No
SP is stable at L4?
HPC Boosting algorithm is deactivated
Yes
SP + Temporal Memory (TM) training process for L2 with active cell SDR Indexes from L4
Accuracy Check for >=90% with 30 times repeats
Output with Cell SDR trace displayed for last 5 cycles
MaxCycle Reached Stop iteration.
End

Figure 3: Proposed Architecture model for Sequence Learning

D. *Implementation*

The entire experiment is written in C# and uses all of the NeocortexApi package's functionality as well as the Microsoft.Net Core 3.1 foundation. The entire project code is open-source, and the current source code can be seen on the GitHub link provided. The entire procedure is carried out in a file called *SequenceLearning.cs*. The coding approach has been broken down into six stages, which are as follows -

- Initialization of Parameters, Scalar Encoders and other values.
- Training of Spatial Pooler in New-born stage [1].
- Training of Spatial Pooler + Temporal Memory on the given input pattern set.
- To obtain top 3 Predicted Input Values.
- Accuracy check and Match/Mismatch Count.
- To display cell state trace.

1. *Initialization of Parameters, Scalar Encoders and other values*

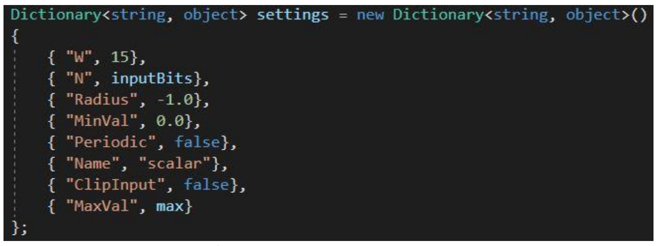
To configure the HTM network, this experiment uses 2048 columns and each input value is encoded with 100 bits [3]. In NeocortexApi some default crucial perimeters are needed to be declared before initializing a scalar encoder –

I. W = It denotes the width for number of bits that are set to encode a single value. It must be set to odd value in order to avoid the centering problems.

II. N = Denotes number of input bits on the output window. It must be N >= W.

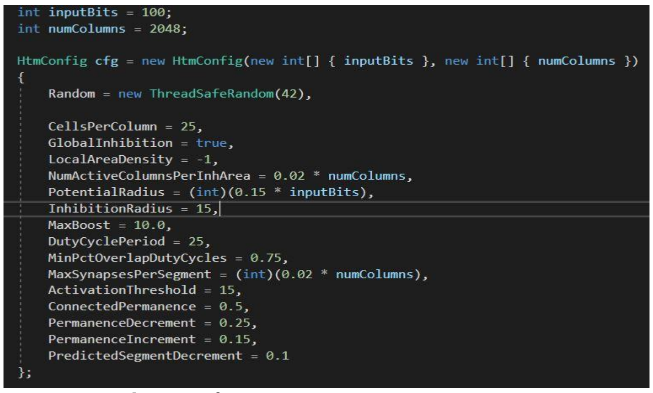
III. Max and Min values = Defines input range. The input sequence at the lower layer must be within that specified range.

IV. Radius = Defines (W/N). So we set a value of -1, to neglect the effect of range.



```
Dictionary<string, object> settings = new Dictionary<string, object>()
{
    { "W", 15},
    { "N", inputBits},
    { "Radius", -1.0},
    { "MinVal", 0.0},
    { "Periodic", false},
    { "Name", "scalar"},
    { "ClipInput", false},
    { "MaxVal", max}
};
```

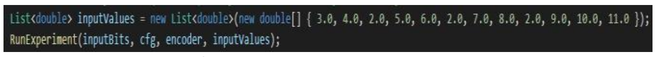
Figure 4: Initialization of Scalar Encoders



```
int inputBits = 100;
int numColumns = 2048;

HtmConfig cfg = new HtmConfig(new int[] { inputBits }, new int[] { numColumns })
{
    Random = new ThreadSafeRandom(42),

    CellsPerColumn = 25,
    GlobalInhibition = true,
    LocalAreaDensity = -1,
    NumActiveColumnsPerInhArea = 0.02 * numColumns,
    PotentialRadius = (int)(0.15 * inputBits),
    InhibitionRadius = 15,
    MaxBoost = 10.0,
    DutyCyclePeriod = 25,
    MinPctOverlapDutyCycles = 0.75,
    MaxSynapsesPerSegment = (int)(0.02 * numColumns),
    ActivationThreshold = 15,
    ConnectedPermanence = 0.5,
    PermanenceDecrement = 0.25,
    PermanenceIncrement = 0.15,
    PredictedSegmentDecrement = 0.1
};
```

Figure 5: Initialization of Parameters

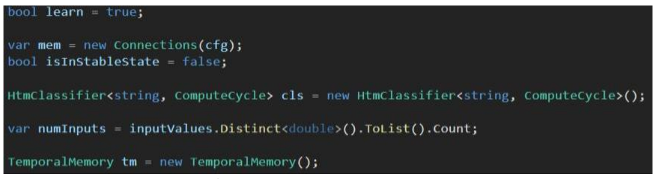
The data stream from the training sequence is used to populate the *inputValues* list. This variable is required to keep track of the entire training sequence during a Sequence Learning experiment. It can be customized. To perform the remaining tasks, the *RunExperiment()* function is invoked with all of the required inputs.



```
List<double> inputValues = new List<double>(new double[] { 3.0, 4.0, 2.0, 5.0, 6.0, 2.0, 7.0, 8.0, 2.0, 9.0, 10.0, 11.0 });
RunExperiment(inputBits, cfg, encoder, inputValues);
```

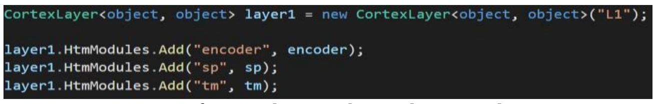
Fig 6: Input Sequence for Sequence learning

The code below initializes Booleans, Cortex layers for Connections, TM and the HTM Classifier. The Cortex Layers is a pipeline that integrates the encoder, SP, and TM components of HTM [3]. It aids in the separation of the preparation and execution zones.



```
bool learn = true;

var mem = new Connections(cfg);
bool isInStableState = false;

HtmClassifier<string, ComputeCycle> cls = new HtmClassifier<string, ComputeCycle>();

var numInputs = inputValues.Distinct<double>().ToList().Count;

TemporalMemory tm = new TemporalMemory();
```

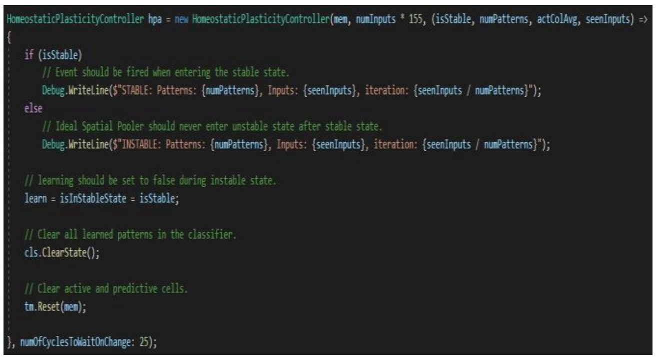
Figure 7: Initialization of other important parameters



```
CortexLayer<object, object> layer1 = new CortexLayer<object, object>("L1");

layer1.HtmModules.Add("encoder", encoder);
layer1.HtmModules.Add("sp", sp);
layer1.HtmModules.Add("tm", tm);
```

Figure 8: Integration of Cortex layer with encoder, SP and TM

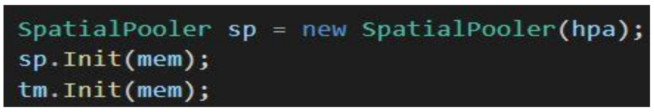
In order to introduce the "New-born" effect, *HomeostaticPlasticityController* is included in the SP algorithm. Once the SP has attained the required stable state for all input patterns, this effect monitors the learning process and deactivates the boosting mechanism (New-born effect) [1].



```
HomeostaticPlasticityController hpa = new HomeostaticPlasticityController(mem, numInputs * 155, (isStable, numPatterns, actColAvg, seenInputs) =>
{
    if (isStable)
        // Event should be fired when entering the stable state.
        Debug.WriteLine($"STABLE: Patterns: {numPatterns}, Inputs: {seenInputs}, iteration: {seenInputs / numPatterns}");
    else
        // Ideal Spatial Pooler should never enter unstable state after stable state.
        Debug.WriteLine($"INSTABLE: Patterns: {numPatterns}, Inputs: {seenInputs}, iteration: {seenInputs / numPatterns}");

    // learning should be set to false during instable state.
    learn = isInStableState = isStable;

    // Clear all learned patterns in the classifier.
    cls.ClearState();

    // Clear active and predictive cells.
    tm.Reset(mem);

}, numOfCyclesToWaitOnChange: 25);
```

Figure 9: *HomeostaticPlasticityController* in SP algorithm

In addition, the *hpa* parameter for HPC Algorithm is initialized and provided to SP so that the trained SP can attain a stable state in the new born stage.

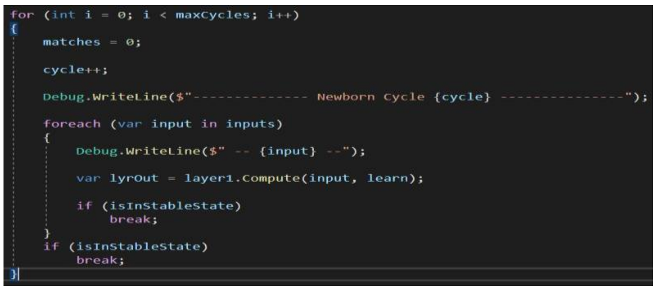


```
SpatialPooler sp = new SpatialPooler(hpa);
sp.Init(mem);
tm.Init(mem);
```

Fig 10: SP and TM initializations

2. *Training of Spatial Pooler in New-born stage*

SP will be quickly trained with input data from Sequence Stream iterating per cycle, and boosting will take place using the HPC Algorithm in NeocortexApi until SP reaches a stable state. The iteration cycle will break once SP has attained it's stability in New-born stage. [1].

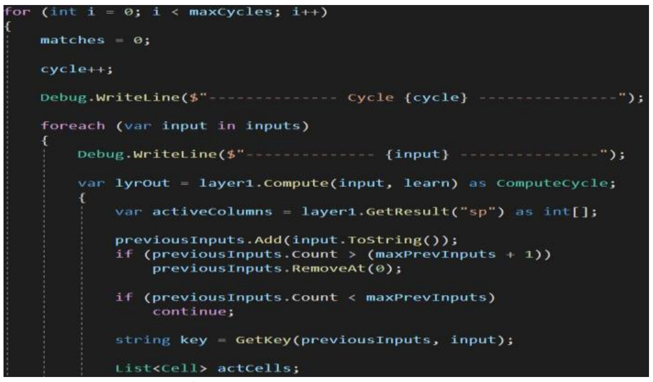


```
for (int i = 0; i < maxCycles; i++)
{
    matches = 0;

    cycle++;

    Debug.WriteLine($"-------------- Newborn Cycle {cycle} ---------------");

    foreach (var input in inputs)
    {
        Debug.WriteLine($" -- {input} --");

        var lyrOut = layer1.Compute(input, learn);

        if (isInStableState)
            break;
    }
    if (isInStableState)
        break;
}
```

Figure 11: SP in New-born stage

3. *Training of Spatial Pooler + Temporal Memory on the given input pattern set*

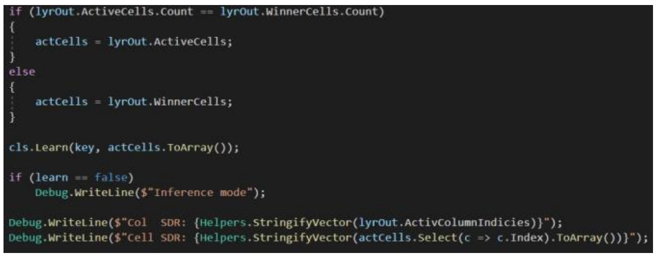
When SP has stabilized, the SP+TM training procedure for layer L2 will commence, using the active cell SDR indexes from layer L4 for each input during each cycle [3]. An HTM Classifier is required to generate output from layer L2. The *learn()* method in the HTM classifier in NeocortexApi accepts a key as a parameter, which is a string of potential sequences based on the current input, and memorizes the SDR indexes for that key. As a result, the code is written in this manner to achieve the required result [3].



```
for (int i = 0; i < maxCycles; i++)
{
    matches = 0;

    cycle++;

    Debug.WriteLine($"-------------- Cycle {cycle} ---------------");

    foreach (var input in inputs)
    {
        Debug.WriteLine($"-------------- {input} ---------------");

        var lyrOut = layer1.Compute(input, learn) as ComputeCycle;
        {
            var activeColumns = layer1.GetResult("sp") as int[];

            previousInputs.Add(input.ToString());
            if (previousInputs.Count > (maxPrevInputs + 1))
                previousInputs.RemoveAt(0);

            if (previousInputs.Count < maxPrevInputs)
                continue;

            string key = GetKey(previousInputs, input);

            List<Cell> actCells;
```

Figure 12: Training of SP+TM

tm.Reset(mem) is introduced to line 279 of the source code in order to speed up the learning process, even if it will not be able to predict the exact first element in every output cycle.

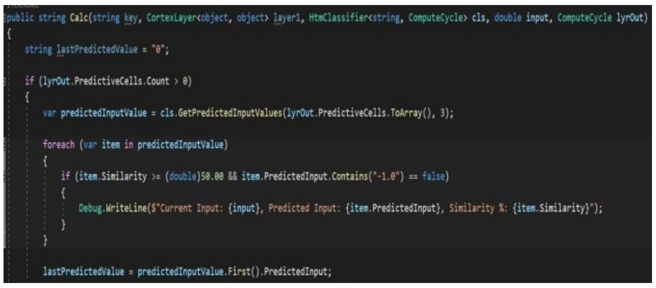


```
if (lyrOut.ActiveCells.Count == lyrOut.WinnerCells.Count)
{
    actCells = lyrOut.ActiveCells;
}
else
{
    actCells = lyrOut.WinnerCells;
}

cls.Learn(key, actCells.ToArray());

if (learn == false)
    Debug.WriteLine($"Inference mode");

Debug.WriteLine($"Col  SDR: {Helpers.StringifyVector(lyrOut.ActivColumnIndicies)}");
Debug.WriteLine($"Cell SDR: {Helpers.StringifyVector(actCells.Select(c => c.Index).ToArray())}");
```

Figure 13: Cell and Column SDRs

4. *To obtain top 3 Predicted Input Values*

A new version of the HTM classifier was released, which gets the best three predicted values from a list of index input values which has similarity of more than 50% [3].



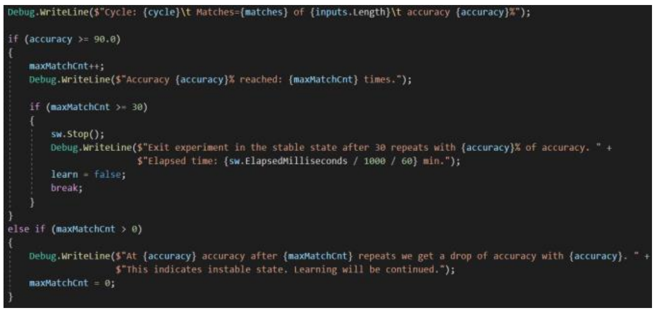
```
public string Calc(string key, CortexLayer<object, object> layer1, HtmClassifier<string, ComputeCycle> cls, double input, ComputeCycle lyrOut)
{
    string lastPredictedValue = "0";

    if (lyrOut.PredictiveCells.Count > 0)
    {
        var predictedInputValue = cls.GetPredictedInputValues(lyrOut.PredictiveCells.ToArray(), 3);

        foreach (var item in predictedInputValue)
        {
            if (item.Similarity >= (double)50.00 && item.PredictedInput.Contains("-1.0") == false)
            {
                Debug.WriteLine($"Current Input: {input}, Predicted Input: {item.PredictedInput}, Similarity %: {item.Similarity}");
            }
        }

        lastPredictedValue = predictedInputValue.First().PredictedInput;
```

Figure 14: New version of HTM classifier method to obtain top 3 predicted values in sequence learning

The method takes the key string representing the sequence and memorizes the SDR for that key. It doesn't matter what this value is. It should only provide information about the input. The classifier goes through all of the memorized SDR's hash values and tries to match the best ones with the highest amount of bits in the SDR. Finally, the classifier provides an array of inputs that are the most similar [3].

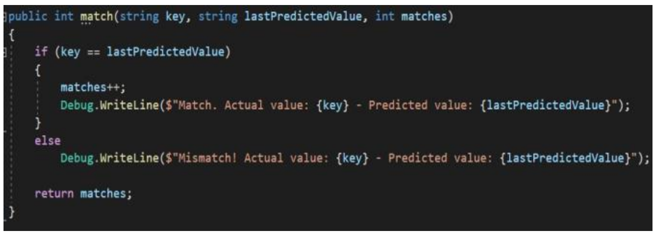
5. *Accuracy check and Match/Mismatch count*

This is a critical stage where, after the New-born stage, TM begins to learn cell SDRs in order to achieve accuracy of >=90%. This is then evaluated, and the number of times this accuracy has been used is counted. This is done to ensure that prediction accuracy is perfected, so that after learning, a machine will either produce an accurate forecast result or not. So, if the necessary accuracy is achieved after 30 repetitions, the machine has been perfectly trained with the input sequence, and the program is terminated. There should almost never be a situation when accuracy suffers. If this happens, retrain the sequences or select the appropriate set of input sequences. In Figure 16, it is checked in the function if key value is equal to the *lastPredictedValue*. If the answer is affirmative, the match variable will be increased. Otherwise, the key value is not equal to *lastPredictedValue*, indicating that learning is still in its early stages or that learning is unstable.



```
Debug.WriteLine($"Cycle: {cycle}\t Matches={matches} of {inputs.Length}\t accuracy {accuracy}%");

if (accuracy >= 90.0)
{
    maxMatchCnt++;
    Debug.WriteLine($"Accuracy {accuracy}% reached: {maxMatchCnt} times.");

    if (maxMatchCnt >= 30)
    {
        sw.Stop();
        Debug.WriteLine($"Exit experiment in the stable state after 30 repeats with {accuracy}% of accuracy. " +
                    $"Elapsed time: {sw.ElapsedMilliseconds / 1000 / 60} min.");
        learn = false;
        break;
    }
}
else if (maxMatchCnt > 0)
{
    Debug.WriteLine($"At {accuracy} accuracy after {maxMatchCnt} repeats we get a drop of accuracy with {accuracy}. " +
                $"This indicates instable state. Learning will be continued.");
    maxMatchCnt = 0;
}
```

Figure 15: Accuracy check and Match/Mismatch count

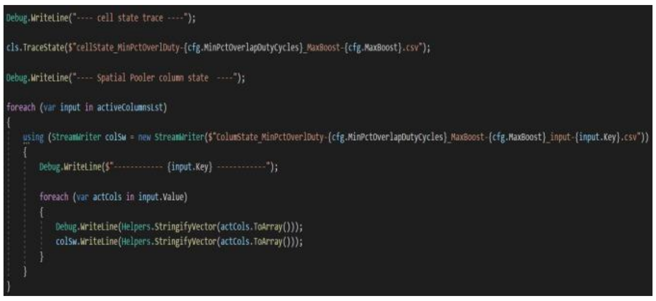


```
public int match(string key, string lastPredictedValue, int matches)
{
    if (key == lastPredictedValue)
    {
        matches++;
        Debug.WriteLine($"Match. Actual value: {key} - Predicted value: {lastPredictedValue}");
    }
    else
        Debug.WriteLine($"Mismatch! Actual value: {key} - Predicted value: {lastPredictedValue}");

    return matches;
}
```

Figure 16: Method to check key value and lastPredictedValue

6. To display cell state trace

Over every index input values, the cell state trace displays the cell SDR values for the last 5 cycles, after it attains accuracy of >=90% up to 30 iterations. This can be used to plot statistical results to see whether there is any instability during the learning process.



```
Debug.WriteLine("---- cell state trace ----");

cls.TraceState($"cellState_MinPctOverlDuty-{cfg.MinPctOverlapDutyCycles}_MaxBoost-{cfg.MaxBoost}.csv");

Debug.WriteLine("---- Spatial Pooler column state  ----");

foreach (var input in activeColumnsLst)
{
    using (StreamWriter colSw = new StreamWriter($"ColumState_MinPctOverlDuty-{cfg.MinPctOverlapDutyCycles}_MaxBoost-{cfg.MaxBoost}_input-{input.Key}.csv"))
    {
        Debug.WriteLine($"----------- {input.Key} -----------");

        foreach (var actCols in input.Value)
        {
            Debug.WriteLine(Helpers.StringifyVector(actCols.ToArray()));
            colSw.WriteLine(Helpers.StringifyVector(actCols.ToArray()));
        }
    }
}
```

Figure 17: Method to display cell state trace on output window

III. Experiment Observation and Results

Several sorts of input sequences are used in more than fifteen investigations. The majority of the time, L4 is stable, and the specified precision has been obtained. All of the unit-tested sequences as well as the output of the findings have been posted to GitHub. Though we received 100 percent accuracy for these sequences each time, we didn't get a perfect match for every cycle. It's because at line 279 of the source code, we implemented *tm.Reset(mem)*, which does not predict the exact first element in every cycle. Even though it's interesting to remark, learning happens more quickly here. All created SDRs should remain unmodified for the duration of the SP instance after the SP reaches a stable state [1]. The SP is said it be stable if all the active cells in SDRs is same for certain iterations [1]. It is also said to be stable if specified number of stable iterations is achieved for all SDRs [1].

Let's take a look at experiment results. In this case, we have input sequence as shown –

```
inputValues =  ({ 0.0, 1.0, 0.0, 2.0, 3.0, 4.0, 5.0, 6.0, 5.0,
4.0, 3.0, 7.0, 1.0, 9.0, 12.0, 11.0, 12.0, 13.0, 14.0, 11.0,
12.0, 14.0, 5.0, 7.0, 6.0, 9.0, 3.0, 4.0, 3.0, 4.0, 3.0, 4.0 });
```

To get better training results, it is always a good approach to use bigger input sequences (>10). In the below result block we have Index 0, Index 1, and Index 2 at 100% similarity in this example, which are then provided as anticipated input values during the training phase in order to select the top three highest predicted values. As seen below, the similarity percent may not be high during the early phases of the learning process. However, as learning progresses, the percent of similarity increases, and it finally reaches the requisite accuracy. It's also worth noting that one pattern leads to another and so on during the learning process. This example output result was captured at cycle 173, after the input sequence had been stable for 30 repetitions with no loss of precision. The New-born stage lasts until cycle 106, at which point it breaks out and SP+TM training begins. The rest of the index values were presumably less than 50%, so they were not chosen to be displayed here. For this HTM network design, it is noticed that TM provides all stable and active cell SDR.

```
------------- Cycle 173 ---------------

------------- 6 ---------------

Active segments: 40, Matching segments: 40


Col SDR: 503, 543, 546, 570, 575, 615, 618, 638, 674, 681, 698, 717, 725,
730, 745, 746, 750, 772, 774, 777, 778, 788, 799, 801, 834, 846, 853, 856,
860, 863, 871, 874, 875, 882, 883, 891, 893, 894, 900, 915, Cell SDR:
12583, 13598, 13656, 14256, 14384, 15389, 15469, 15966, 16858, 17036,
17461, 17931, 18144, 18262, 18625, 18651, 18768, 19323, 19367, 19434,
19461, 19712, 19982, 20029, 20853, 21174, 21345, 21413, 21521, 21585,
21795, 21867, 21879, 22061, 22093, 22299, 22342, 22357, 22510, 22876,


Match. Actual value: 5-4-3-7-1-9-12-11-12-13-14-11-12-14-5-7-6-9-3-4-
3-4-3-4-0-1-0-2-3-4-5-6 - Predicted value: 5-4-3-7-1-9-12-11-12-13-14-
11-12-14-5-7-6-9-3-4-3-4-3-4-0-1-0-2-3-4-5-6

Item length: 40        Items: 34

Predictive cells: 40    10653, 10861, 10906, 11480, 12500, 12669, 13011,
13651, 14100, 14270, 14390, 15267, 15647, 15963, 16538, 16673, 16743,
17046, 17471, 17479, 17525, 17757, 17890, 18025, 18184, 18479, 18543,
18580, 18679, 18873, 18944, 19218, 19319, 19351, 19435, 19491, 19718,
19751, 19785, 19863,

>indx:0    inp/len: 4-3-7-1-9-12-11-12-13-14-11-12-14-5-7-6-9-3-4-3-4-3-
4-0-1-0-2-3-4-5-6-5/40 ,Same Bits = 40      , Similarity% 100      10653,
10861, 10906, 11480, 12500, 12669, 13011, 13651, 14100, 14270, 14390,
15267, 15647, 15963, 16538, 16673, 16743, 17046, 17471, 17479, 17525,
17757, 17890, 18025, 18184, 18479, 18543, 18580, 18679, 18873, 18944,
19218, 19319, 19351, 19435, 19491, 19718, 19751, 19785, 19863,

>indx:1    inp/len: 4-3-4-3-4-0-1-0-2-3-4-5-6-5-4-3-7-1-9-12-11-12-13-14-
11-12-14-5-7-6-9-3/40 ,Same Bits = 40      , Similarity% 100      10653,
10861, 10906, 11480, 12500, 12669, 13011, 13651, 14100, 14270, 14390,
15267, 15647, 15963, 16538, 16673, 16743, 17046, 17471, 17479, 17525,
17757, 17890, 18025, 18184, 18479, 18543, 18580, 18679, 18873, 18944,
19218, 19319, 19351, 19435, 19491, 19718, 19751, 19785, 19863,

>indx:2    inp/len: 4-3-4-0-1-0-2-3-4-5-6-5-4-3-7-1-9-12-11-12-13-14-11-
12-14-5-7-6-9-3-4-3/40 ,Same Bits = 40      , Similarity% 100      10653,
10861, 10906, 11480, 12500, 12669, 13011, 13651, 14100, 14270, 14390,
15267, 15647, 15963, 16538, 16673, 16743, 17046, 17471, 17479, 17525,
17757, 17890, 18025, 18184, 18479, 18543, 18580, 18679, 18873, 18944,
19218, 19319, 19351, 19435, 19491, 19718, 19751, 19785, 19863,

Current Input: 6

Predicted Input: 4-3-7-1-9-12-11-12-13-14-11-12-14-5-7-6-9-3-4-3-4-3-4-
0-1-0-2-3-4-5-6-5    Similarity %: 100

Predicted Input: 4-3-4-3-4-0-1-0-2-3-4-5-6-5-4-3-7-1-9-12-11-12-13-14-
11-12-14-5-7-6-9-3    Similarity %: 100

Predicted Input: 4-3-4-0-1-0-2-3-4-5-6-5-4-3-7-1-9-12-11-12-13-14-11-
12-14-5-7-6-9-3-4-3    Similarity %: 100
```

When the TM has learned all of the SDR patterns and has achieved an accuracy of (30 times), a 'cell state trace' is generated on the debug window at the end of the experiment, which can be used to compare column/cell activity. Consider the following cell state trace created for the input sequence -

```
---- cell state trace ----

0-1-0-2-3-4-5-6-5-4-3-2-1-9-12-11-12-13-14-11-12-14-5-7-6-9-3-4-3-4-
3-4

7594, 8438, 8532, 8638, 9209, 9367, 9631, 9688, 10242, 10262, 10854,
10914, 11303, 11485, 12086, 12519, 12660, 14266, 14377, 14457,
14580, 15102, 15551, 15630, 15738, 15767, 16009, 16052, 16124,
16155, 16474, 16548, 16714, 16725, 16829, 16934, 17044, 17199,
17247, 17767,

7594, 8438, 8532, 8638, 9209, 9367, 9631, 9688, 10242, 10262, 10854,
10914, 11303, 11485, 12086, 12519, 12660, 14266, 14377, 14457,
14580, 15102, 15551, 15630, 15738, 15767, 16009, 16052, 16124,
16155, 16474, 16548, 16714, 16725, 16829, 16934, 17044, 17199,
17247, 17767,

7594, 8438, 8532, 8638, 9209, 9367, 9631, 9688, 10242, 10262, 10854,
10914, 11303, 11485, 12086, 12519, 12660, 14266, 14377, 14457,
14580, 15102, 15551, 15630, 15738, 15767, 16009, 16052, 16124,
16155, 16474, 16548, 16714, 16725, 16829, 16934, 17044, 17199,
17247, 17767,

7594, 8438, 8532, 8638, 9209, 9367, 9631, 9688, 10242, 10262, 10854,
10914, 11303, 11485, 12086, 12519, 12660, 14266, 14377, 14457,
14580, 15102, 15551, 15630, 15738, 15767, 16009, 16052, 16124,
16155, 16474, 16548, 16714, 16725, 16829, 16934, 17044, 17199,
17247, 17767,

7594, 8438, 8532, 8638, 9209, 9367, 9631, 9688, 10242, 10262, 10854,
10914, 11303, 11485, 12086, 12519, 12660, 14266, 14377, 14457,
14580, 15102, 15551, 15630, 15738, 15767, 16009, 16052, 16124,
16155, 16474, 16548, 16714, 16725, 16829, 16934, 17044, 17199,
17247, 17767,
```

During the learning process, a cell state trace is generated for each index value. For SDR 1, SDR 2, SDR 3, SDR 4, SDR 5, they are generated five times. Cell SDR values from the previous 5 cycles are always used. In this situation, the input sequence corresponds to cell SDR from cycle 169 to cycle 173. The cell state trace for cycle 173 may already be seen in the diagram above.

After the experiment, we plot the learned SDRs to see whether there is any instability. Cell SDRs for two input sequences are depicted in the following result -

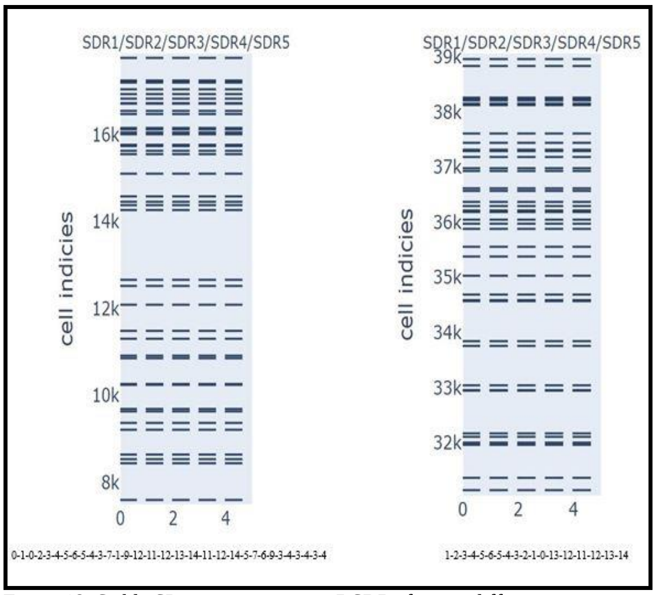


Figure 18: Stable SP state representing 5 SDRs for two different input patterns

In both of these results, SP is in stable state representing 5 SDRs for two different input pattern, at the end of 30 iterations. It can also be observed that after SP becomes stable, it remains steady until accuracy is reached. For various input sequences, many such representations have been made on GitHub.

IV. GITHUB LINK

Link to Source Code And Output Results:

https://github.com/PrasadSahana/neocortexapi/tree/master/source/Samples/NeoCortexApiSample

Link guide on generation of visualized SDR Plots:

https://github.com/PrasadSahana/neocortexapi/blob/master/source/Samples/NeoCortexApiSample/Statistical%20Analysis/Investigation%20of%20HTM%20layer.md

V. CONCLUSION

The goal of this study is to look at the instability of the HTM SP method, which has to learn spatial patterns in an unsupervised manner. The original SP already incorporates some type of HPC, as stated in the previous sections [1]. The HPC deactivates boosting mechanism after which SP reaches a stable state. After that SP+TM trains a large number of input sequences by improving the quality of learning that can be depended upon [1]. In conclusion, the number of active cell SDRs remains consistent after 30 cycles. This paper also includes a C# code solution based on the proposed algorithm, as well as a brief description of the entire process of using Neocortexapi to accomplish this.

ACKNOWLEDGMENT

I would like to sincerely thank Prof. Damir Dobric for giving me an opportunity to closely work with him on this topic. I would also like to thank Prof. Dr. Andreas Pech for all the guidance and continuous support throughout this project.

REFERENCES

[1] Dobric, D., Pech, A., Ghita, B., & Wennekers, T. (2021). Improved HTM Spatial Pooler with Homeostatic Plasticity Control.

[2] Numenta. 2021. Numenta Resources on Temporal Memory. [online] Available at: <https://numenta.com/resources/biological-and-machine-intelligence/temporal-memory-algorithm/> [Accessed 16 September 2021].

[3] GitHub. 2021. GitHub - ddobric/neocortexapi: C#.NET Core Implementation of Hierarchical Temporal Memory Cortical Learning Algorithm.. [online] Available at: <https://github.com/ddobric/neocortexapi> [Accessed 16 September 2021].