Information Technology Course
Module: Project
by Prof. Dr. Andreas Pech/
Dipl. Ing. Damir Dobric

Refactoring open-source project NeoCortexAPI

Quang Nhat Bui - 1148709
email: buinhatquang31011996@gmail.com

***Abstract*— Recently, the phrase "Artificial Intelligence" (AI) has gained popularity. AI is used in a variety of applications, including categorization, pattern recognition, prediction, and so on. Current machine learning or deep learning algorithms are largely math-based and do not accurately reflect the human brain. Numenta created a novel idea called Hierarchical Temporal Memory (HTM) that is similar to the structure and operation of the cerebral cortex. The HTM system can identify spatial patterns and continuously learn successive inputs. Initially, the algorithm is implemented in C++ and Python. The HTM community later ported the technology to Java. However, there is no existing version of HTM in C#. It is understandable that .NET only started targeting multiple platforms recently and there are more developers get familiar with C#. Thus, Damir Dobric created NeoCortexAPI as a port version of htm.java and made open sourced in 2021 to help C# developers to approach the cutting edge of sequence learning algorithm that truly learn in online fashion.**

Keywords—Hierarchical Temporal Memory, NeoCortexAPI, .NET, open-source

I. Introduction

Hierarchical Temporal Memory (HTM) emerged as a neocortical abstraction [1]. It differs from traditional machine learning algorithms and neural networks, which are developed from the ground up on mathematical formulations. The algorithm is accessible in three languages: C++, Python, and Java. HTM, on the other hand, does not have a C# implementation. The NeoCortexAPI, which was released as open source in early 2021, is addressing this issue by bringing the sequence learning method to.NET. The intention of this paper is to document the NeoCortexAPI as well as explain important components of HTM. The paper begins by discussing the notion of Sparse Distributed Representation (SDR), and then presents all of the HTM components with pseudo code for implementation. Finally, there will be a discussion of how NeoCortexAPI could be used in the future.

II. Sparse Distribution Representation

Even though neurons in the neocortex are extensively linked, inhibitory neurons ensure that only a small fraction of neurons are active at any given moment. As a result, in the brain, information is always expressed by a small proportion of activated neurons within a huge population of neurons. A "sparse distributed representation" is the name given to this type of encoding. The term "sparse" refers to the fact that only a tiny fraction of neurons is activated at any given moment. The term "distributed" refers to the fact that numerous neurons must be activated to express anything. To express the whole message, a single activated neuron must be evaluated within the context of other neurons [2].

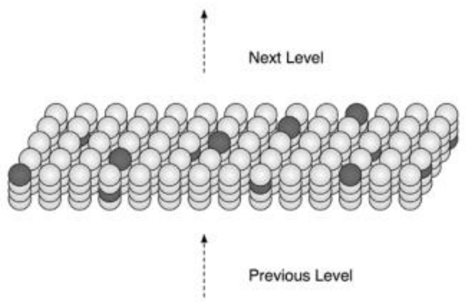

Fig. 1 Small region of SDR with active cells in dark gray *[2]*

As illustrated in Fig. 1, an SDR is a large array of bits, the majority of which are turned off (0s) and just a few are switched on (1s). As two SDRs are deemed to have comparable meaning if they have numerous overlapping locations of on bits, each in SDR represents some meaning. The more bits they share, the more comparable the data or the less the gap between two SDRs.

III. HTM Components

HTM system consists of three main components: encoder, spatial pooler, temporal memory. On top of that, there are cortex layer, region and network. A layer contains several HTM components, a region contains many layers and a network consists of many region.

A. Encoder

Encoders are tools that convert data sources into SDRs without altering the semantic content [3]. Each encoder will be assigned to a certain data type. If the input data is blood pressure, for example, a blood pressure encoder is required. The following encoders are presently supported by the project: Scalar encoder, Category encoder, and DateTime encoder. A new encoder will be necessary if a new type of data is to be fed into the HTM system.

In NeoCortexAPI, the following code snippet is used to configure an encoder:

```
Dictionary<string, object> encoderSettings = new
Dictionary<string, object>();
encoderSettings.Add("W", 25);
encoderSettings.Add("N", (int) 1024);
encoderSettings.Add("MinVal", (double)0);
encoderSettings.Add("MaxVal", (double)100);
encoderSettings.Add("Radius", (double)-1);
encoderSettings.Add("Periodic", (bool)false);
encoderSettings.Add("ClipInput", (bool)true);
encoderSettings.Add("Name", "TestScalarEncoder");
```

The encoder settings will then be fed into an encoder, in this case, a scalar encoder:

```
ScalarEncoder encoder = new
ScalarEncoder(encoderSettings);

int[] result = encoder.Encode(input);
```

For a given input the result can be visualized as a picture with the help of NeoCortexUtils library:

```
int[,] twoDimenArray =
    ArrayUtils.Make2DArray<int>(
        result,
        (int)Math.Sqrt(result.Length),
        (int)Math.Sqrt(result.Length));
var twoDimArray =
    ArrayUtils.Transpose(twoDimenArray);

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

The output SDRs of the scalar encoder from the inputs 3.6, 3.8, and 17.6 are presented in Fig. 2 where the yellow bricks are 0s and the black blocks are 1s. TABLE *I* describes the meaning of the encoder parameters.

TABLE I ENCODER PARAMETERS [4]

Parameter	Description
N	The number of bits in the output. Must be greater than or equal to W
W	The number of bits that are set to encode a single value - the “width” of the output signal restriction: w must be odd to avoid centering problems.
MinVal	The minimum value of the input signal.
MaxVal	The upper bound of the input signal. (input is strictly less if Periodic == True)
Radius	Two inputs separated by more than the radius have non-overlapping representations. Two inputs separated by less than the radius will in general overlap in at least some of their bits. It can be described as the radius of the input.
Resolution	Two inputs separated by greater than, or equal to the resolution are guaranteed to have different representations.
Periodic	If true, then the input value “wraps around” such that MinVal = MaxVal. For a periodic value, the input must be strictly less than MaxVal, otherwise MaxVal is a true upper bound.
ClipInput	if true, non-periodic inputs smaller than minVal or greater than maxVal will be clipped to minVal/maxVal

For the specific type of input data, a custom encoder is required. NeoCortexAPI offers an abstract class from which new implementation should derive. The class name is NeoCortexApi.Encoders.EncoderBase. This base encoder has predefined properties for configuration all the necessary functionality for an encoder to be functional in NeoCortexAPI framework. For example, if you want to create a blood pressure encoder, the following code snippet can be used:

```
namespace NeoCortexApi.Encoders
{
    public class BloodPressureEncoder : EncoderBase
    {
        public override int Width { get; }

        public override bool IsDelta { get { return
false; } }

        public override int[] Encode(object
inputData)
        {
            // Implementation of the blood pressure
encoder
            // Output is 1D array SDR
        }

        public override List<T> GetBucketValues<T>()
        {
            throw new NotImplementedException();
        }
    }
}
```

B. *Spatial Pooler*

Spatial Pooler (SP) is a learning algorithm that is designed to replicate the neurons functionality of human brain. Essentially, if a brain sees one thing multiple times, it is going to strengthen the synapses that react to the specific input result in the recognition of the object. Similarly, if several similar SDRs are presented to the SP algorithm, it will reinforce the columns that are active according to the on bits in the SDRs. If the number of training iterations is big enough, the SP will be able to identify the objects by producing different set of active columns within the specified size of SDR for different objects.

To get started with SP in NeoCortexAPI, there are several parameters which can be initialized through HtmConfig.

```
int inputBits = 100;
int numColumns = 1024;

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,

    // initialize other parameters

};
```

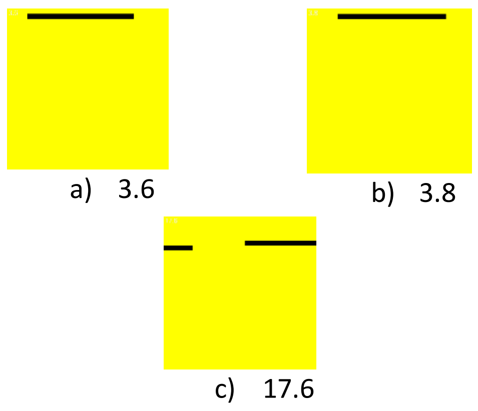

Fig. 2 SDR visualization of several encoded inputs

For compatibility among the implementations of the HTM system in various programming languages, the original parameter initialization through key value pairs is kept and can be done with the following snippet:

```
int inputBits = 100;
int numColumns = 1024;

Parameters p = Parameters.getAllDefaultParameters();
p.Set(KEY.INPUT_DIMENSIONS, new int[] { inputBits
});
p.Set(KEY.COLUMN_DIMENSIONS, new int[] { numColumns
});
p.Set(KEY.RANDOM, new ThreadSafeRandom(42));
p.Set(KEY.CELLS_PER_COLUMN, 25);
p.Set(KEY.GLOBAL_INHIBITION, true);
p.Set(KEY.LOCAL_AREA_DENSITY, -1);
p.Set(KEY.NUM_ACTIVE_COLUMNS_PER_INH_AREA, 0.02 *
numColumns);
p.Set(KEY.POTENTIAL_RADIUS, (int)(0.15 *
inputBits));

// set other spatial pooler parameters
```

Although key value pairs are still supported for initializing Spatial Pooler, it is recommended to use HtmConfig because it is easier to track which parameters have been initialized in the constructor with auto property initializers (supported since C# 6.0, current is C# 10) and the help of InteliSense in Visual Studio. Furthermore, because the parameter name is already declared in the class, any misspelling will result in compiled errors that are easily noticed by the compiler. Parameter mistakes, unlike key value pair initialization, will result in runtime issues that are difficult to identify and correct.

For example, the following configuration will result in compiled error:

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

    CellPerColumn = 25,
    GlobalInhibition = true,
    LocalAreaDensity = -1,
    NumActiveColumnsPerInhArea = 0.02 * numColumns,
    PotentialRadius = (int)(0.15 * inputBits),

    // other parameters
};
```

The parameter CellsPerColumn is misspelled, but the compiler can detect it and prevent the project from being compiled, as well as reporting the issue to the user. With this information, the user may quickly correct the error.

With the key value pairs initialization, the parameters is set using `Set(string key, object val)`. The parameter dictionary will be filled with keys of type string as a result of this. If there are any mistakes, there will be no errors after building the project because everything is compiled. The runtime error will occur only when the parameters are applied. Although a static class named `NeoCortexApi.Entities.KEY` was established to circumvent the problem, there is no requirement that developers use the static class instead of passing the argument as a literal string, which results in the error seen below:

```
public void Set(string key, object val)
{
    paramMap[key] = val;
}

Parameters p = Parameters.getAllDefaultParameters();
p.Set("random", new ThreadSafeRandom(42));
p.Set("inputDimensions", new int[] { inputBits });
p.Set("cellPerColumn", 5);
p.Set("columnDimension", new int[] { 500 });

//p.apply(mem);
```

Another critical component of the NeoCortexApi system is connections. It denotes the possible relationship, i.e. the persistence value, between the output columns and the input SDR bits. The Spatial Pooler and Temporal Memory algorithms rely on these connections.

The parameters can be applied to the Connections in both cases as follow:

- Parameters initialization with HtmConfig:

```
var htmConfig = new HtmConfig(new int[] { 5 }, new
int[] { 5 })
{
    // default values of parameters is set in the
constructor
}
Connections mem = new Connections(htmConfig);
```

- Parameters initialization with key value pairs:

```
Parameters p = Parameters.getAllDefaultParameters();
Connections mem = new Connections();
p.apply(mem);
```

The meaning of HTM parameters is described in *TABLE II.*

TABLE II SPATIAL POOLER PARAMETERS [4]

Parameter Name	Description
POTENTIAL_RADIUS	Defines the radius in number of input cells visible to column cells. It is important to choose this value, so every input neuron is connected to at least a single column. For example, if the input has 50000 bits and the column topology is 500, then you must choose some value larger than 50000/500 > 100.
POTENTIAL_PCT	Defines the percent of inputs withing potential radius, which can/should be connected to the column.
GLOBAL_INHIBITION	If TRUE global inhibition algorithm will be used. If FALSE local inhibition algorithm will be used.
INHIBITION_RADIUS	Defines neighbourhood radius of a column.
LOCAL_AREA_DENSITY	Density of active columns inside of local inhibition radius. If set on value < 0, explicit number of active columns (NUM_ACTIVE_COLUMNS_PER_INH_AREA) will be used.
NUM_ACTIVE_COLUMNS_PER_INH_AREA	An alternate way to control the density of the active columns. If this value is specified then LOCAL_AREA_DENSITY must be less than 0, and vice versa.
STIMULUS_THRESHOLD	One mini-column is active if its overlap exceeds overlap threshold θ_o of connected synapses.
SYN_PERM_INACTIVE_DEC	Decrement step of synapse permanence value withing every inactive cycle. It defines how fast the NeoCortex will forget learned patterns.
SYN_PERM_ACTIVE_INC	Increment step of connected synapse during learning process
SYN_PERM_CONNECTED	Defines Connected Permanence Threshold θ_p, which is a float value, which must be exceeded to declare synapse as connected.
DUTY_CYCLE_PERIOD	Number of iterations. The period used to calculate duty cycles. Higher values make it take longer to respond to

	changes in boost. Shorter values make it more unstable and likely to oscillate.
MAX_BOOST	Maximum boost factor of a column.

The Spatial Pooler algorithm is initialized in the below code snippet:

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

And the computation of Spatial Pooler is simply by calling `sp.Compute(int[] input, bool learn)`. The result of this method is the short form of the output SDR from the Spatial Pooler. The algorithm starts from an SDR with random distributed connections from each column to the input space. Each connection between an input bit and an output column is assigned a random permanence value between 0 and 1 [5]. The input bits connected to the given column are activated if their permanence values are greater than the connected permanence threshold θ_p defined in the parameter initialization. Each Output column will be active if a certain number of input bits connected to that column, are activated in the input space. With this connections, Spatial Pooler will be able to represent different inputs as SDRs but still be able to maintain the semantic information of the input. The similarity of information is measured by the overlap score of the output columns. The more overlap the two outputs have, the more similar they are.

When learning is turned on, only some columns that have the high overlap score will be chosen to be active and will be allowed to learn, other columns remain unchanged. The active columns will have its connections, which overlap the input, reinforced by increment the synaptic permanence value. Meanwhile, other connections which does not matched the input will have their permanence value diminished. The permanencies are limited from 0 to 1 [5].

Internally, Spatial Pooler also has two separate mechanisms that deal with the granularity of the output SDR and the SDR topology representation. The first one is called boosting. Normally, without boosting, Spatial Pooler will have a limited number of active columns represent different inputs, or their active duty-cycles is close to 1. Other dormant columns will never be active during the whole learning phase. This mean the output SDR can describe less information about the set of input. Boosting mechanism will allow more columns to participate in expressing the input space [5]. The second mechanism is local inhibition. Typically, most of the current encoded information do not hold the spatial meaning. As a result, global inhibition will be applied, which means the winner columns is chosen based on the overlap rank of all other columns in the output SDR. Local inhibition, on the other hand, is critical when the input space has topology, that is, when nearby input neurons convey information from comparable sub-regions of the input space [5]. As the output SDR is spatially linked to the input space, it enables the Spatial Pooler to represent the spatial aspect of the input data.

In practice, several inputs are presented to the Spatial Pooler with learning over a large number of iterations, which may be accomplished with the following code:

```
int maxSPLearningCycles = 1000;

for (int cycle = 0; cycle < maxSPLearningCycles;
cycle++)
{
    //
    // This trains the layer on input pattern.
    foreach (var input in inputs)
    {
        // Learn the encoded SDR pattern.
        int[] activeColumns = sp.Compute(input,
true) as int[];

        var actCols = activeColumns.OrderBy(c =>
c).ToArray();

        // Store the input/output in a dictionary
and calculate the similarity with other output of
the same input later on.
    }
}
```

Note that the output of the Spatial Pooler algorithm contains the index of active columns rather than the whole SDR array.

Spatial Pooler's boosting mechanism allows all columns to be consistently used throughout all patterns [6]. However, even if the columns have already learned a patterns, the boosting mechanism is still active and encourage other inactive columns to express themselves and cause the Spatial Pooler to forget the input. To overcome this issue, the new homeostatic plasticity controller is introduced to the Spatial Pooler to switch off boosting after the learning enter the stable state. The research shown that the output SDRs of the Spatial Pooler remain unchanged during its lifespan.

The Spatial Pooler is extended with the following code:

```
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)
        {
            // INSTABLE STATE
            // This should usually not happen.
            isInStableState = false;
        }
        else
        {
            // STABLE STATE
            // Here you can perform any action if
required.
            isInStableState = true;
        }
    });
SpatialPooler sp = new SpatialPooler(hpa);
```

C. *Temporal Memory*

The spatial output of Spatial Pooler can then be fed into Temporal Memory to study the sequential pattern of the input. The HTM parameters for Temporal Memory are presented in TABLE *III*.

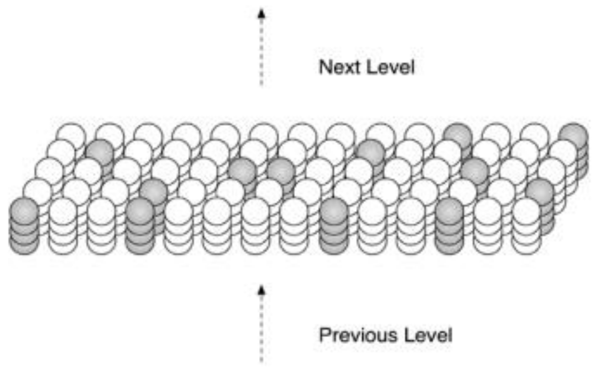

Fig. 3 SDR with bursting when there is no prior state [2]

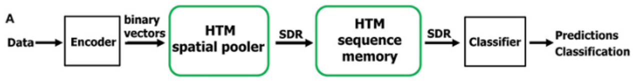

Fig. 4 An end-to-end HTM system [5]

TABLE III TEMPORAL MEMORY PARAMETERS [4]

Parameter	Description
CELLS_PER_COLUMN	Number of cells per columns in the SDR
ACTIVATION_THRESHOLD	The activation threshold of a segment. If the number of active connected synapses on a distal segment is at least this threshold, the segment is declared as active one.
LEARNING_RADIUS	Radius around cell from which it can sample to form distal connections.
MIN_THRESHOLD	If the number of synapses active on a segment is at least this threshold, it is selected as the best matching cell in a bursting column.
MAX_NEW_SYNAPSE_COUNT	The maximum number of synapses added to a segment during learning.
MAX_SYNAPSES_PER_SEGMENT	The maximum number of synapses that can be added to a segment.
MAX_SEGMENTS_PER_CELL	The maximum number of Segments a Cell can have.
INITIAL_PERMANENCE	Initial permanence value for a synapse.
CONNECTED_PERMANENCE	If the permanence value for a synapse is ≥ this value, it is "connected".
PERMANENCE_INCREMENT	If a segment correctly predicted a cell's activity, the permanence values of its active synapses are incremented by this amount.
PERMANENCE_DECREMENT	If a segment correctly predicted a cell's activity, the permanence values of its inactive synapses are decremented by this amount.
PREDICTED_SEGMENT_DECREMENT	If a segment incorrectly predicted a cell's activity, the permanence values of its active synapses are decremented by this amount.

The Temporal Memory can be initialized as follow:

```
TemporalMemory tm = new TemporalMemory();
tm.Init(mem);
```

Similar to Spatial Pooler, Temporal memory has a same method to calculate the output SDR: `tm.Compute(int[] activeColumns, bool learn)`. This method takes the result of active columns in the stable output SDR from the Spatial Pooler (training is off) with learning option to produce a ComputeCycle object which holds the information of winner cells and predictive cells.

Temporal Memory algorithm predicts what the next input SDR will be based on sequences of Sparse Distributed Representations (SDRs) produced by the Spatial Pooling algorithm [7]. Each column in the SDR consists of many cells. Each cell can have three states: active, predictive, and inactive. These cells should have one proximal segment and many distal dendrite segments. The proximal segment is the connection of its the column and several bits in the input space. The distal dendrite segments represent the connection of the cell to nearby cells [2]. When a certain input is fed into the HTM system with no prior state or there is no context of the input, every cell in the active column is active. This is called bursting (see Fig. 3). With the prior state, the algorithm will choose winner cell for each column based on the context of the previous input. From these winner cells, other cells will have the predictive state when the connections to the current active cells in the distal segment of those cells reach a certain value of ACTIVATION_THRESHOLD.

With learning is turned on, if the predictive cells become active in the next input, the permanence values of its active synapses will be incremented, and its inactive ones will be diminished. However, if the system predicts some wrong cells, the cells' active synapses will be punished as their synaptic permanence is reduced. Without learning, there are no change in the synapses of the segments.

The following code illustrates how to start using Temporal Memory to learn a sequence:

```
List<int[]> activeColumnsList = new List<int[]>();
// This activeColumns are the SpatialPooler output
SDRs in short form
for (int i = 0; i < maxCycles; i++)
{
    foreach (var activeColumns in activeColumnsList)
    {
        ComputeCycle cc = tm.Compute(activeColumns,
learn: true);
        // ComputeCycle contain the information of
ActiveCells, WinnerCells and PredictiveCells which
can be used to aggregate the result of the learning
process
    }
}
```

To interpret the Temporal Memory output, a classifier is utilized to match the input to the predicted SDR result. This classifier includes an algorithm for comparing the SDR output of various input sequences and storing them as key value pairs. The classifier, as shown in Fig. 4, is the last component in the HTM system that allows it to be employed in many prediction applications.

The following code snippet demonstrates the use of HtmClassifier:

```
HtmClassifier<string, ComputeCycle> cls = new
HtmClassifier<string, ComputeCycle>();
for (int i = 0; i < maxCycles; i++)
{
    foreach (var activeColumns in activeColumnsList)
    {
        ComputeCycle cc = tm.Compute(activeColumns,
learn: true);

        // The input label is captured during the
encoding phase
        cls.Learn(input, cc.PredictiveCells);
    }
}
```

D. *CortexNetwork, CortexRegion and CortexLayer*

The human cerebral cortex is divided into multiple layers, according to neuroscience. NeoCortexApi, a model based on biology, also introduces CortexLayer, which works as one layer of the cortex. The layer is made up of multiple IHtmModules that are implemented by Encoder, Spatial Pooler, and Temporal Memory. The addition of this component makes the code more presentable and opens up new possibilities for combining many layers together for more complicated applications. The layer combination is implemented using CortexRegion. When all cortical areas are combined, we have a CortexNetwork that resembles the whole cerebral cortex of the human brain.

The following pseudo code demonstrates the use of CortexNetwork, CortexRegion and CortexLayer:

```
CortexNetwork net = new CortexNetwork("my cortex");

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

// the modules are all initialized and configured
EncoderBase encoder = new ScalarEncoder(settings);
SpatialPooler sp = new SpatialPoolerMT(hpa);
TemporalMemory tm = new TemporalMemory();

layer1.AddModule("encoder", encoder);
layer1.AddModule("sp", sp);
layer1.AddModule("tm", tm);

region0.AddLayer(layer1);
net.AddRegion(region0);
```

It is simple to run the Network, Region, or Layer by using the `Compute` method on the component. Specifically, the CortexLayer provides a method to get the result of each IHtmModule using `GetResult(string moduleName)`, allowing freedom in utilizing the component while being presentable.

IV. Future Extension and Applications

The NeoCortexAPI is ported from htm.java but it has its own development. Specifically, the Spatial Pooler was extended with the Homeostatic Plasticity Controller to allow the algorithm to exit new-born state once the learning has stabilized [6]. For future extension, when some layers are initialized and placed into a CortexRegion, their configuration is currently done manually. It is a good idea to create an algorithm that will automatically setup the layers after the initial one.

Until date, the NeoCortexAPI has been underutilized in many classification and prediction applications. As a result, it is instructive to observe how it reacts in real-world classification problems.

References

[1] J. Mnatzaganian, E. Fokoué and D. Kudithipudi, "A Mathematical Formalization of Hierarchical Temporal Memory’s Spatial Pooler," Frontiers in Robotics and AI, vol. 3, p. 81, 2017.

[2] J. Hawkins, S. Ahmad and D. Dubinsky, "Hierarchical Temporal Memory (HTM) Whitepaper," Numenta, Inc., Redwood City, California, United States of America, 2011.

[3] S. Purdy, "Encoding Data for HTM Systems," Numenta, Inc., Redwood City, California, United States of America, 2016.

[4] D. Dobric, "C# Implementation of Hierarchical Temporal Memory Cortical Learning Algorithm in .NET Core and more.," 2018. [Online]. Available: https://github.com/ddobric/neocortexapi. [Accessed 2021].

[5] Y. Cui, S. Ahmad and J. Hawkins, "The HTM Spatial Pooler—A Neocortical Algorithm for Online Sparse Distributed Coding," Frontiers in Computational Neuroscience, vol. 11, p. 111, 2017.

[6] D. Dobric, A. Pech, B. Ghita and T. Wennekers, "Improved HTM Spatial Pooler with Homeostatic Plasticity Control," PEARL, University of Plymouth, Plymouth, England, 2021.

[7] J. Hawkins, S. Ahmad, S. Purdy and A. Lavin, "Temporal Memory Algorithm," in Biological and Machine Intelligence (BAMI), Numenta, 2016-2020.