Improve Unit Tests(HTMClassifier)

Debasish Dutta
Matriculation No. 1343753

***Abstract*— This study provides Hierarchical Temporal Memory (HTM), a machine learning approach that uses Spatial Pooler, Scalar Encoder and Temporal memory where Unit Tests are done on the learned data to be congruent with the expected output. Hierarchical Temporal Memory (HTM) theory, which represents the structural and algorithmic aspects of neocortex, has recently developed a new paradigm in machine intelligence. There is still a lot of work to be done on the HTM algorithm's inference of patterns and structures recognized by the algorithm. The agility after testing the sequentially learned data while coinciding them with the input and output is the actual goal.**

Keywords— scalar encoder, spatial pooler, neocortex, temporal memory.

I. INTRODUCTION

HTM is essentially a theory on how the human brain functions. Three brain features are critical in the development of HTM. To begin with, the brain is a hierarchical organization by nature. Signals flow in both ways along the hierarchy. Additionally, there is signal flow within the region. Second, all of the information stored in the brain is temporal. All aspects of brain learning revolve around the concept of time. Finally, the human brain functions primarily as a memory system. Over time, we try to remember and predict patterns. In a way, all of the cells and their connections are storing the patterns that have been observed through time.

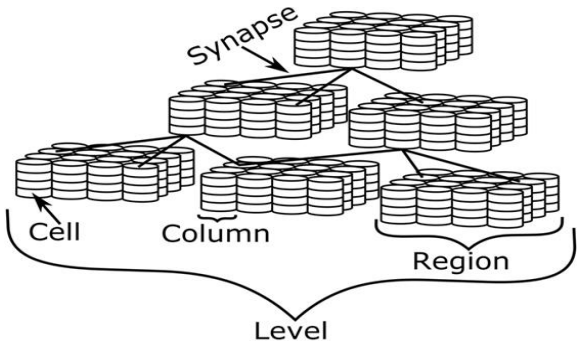

Figure 1: Illustration of the many levels of detail in HTM.

Humans use the neocortex to learn sequences and predict the future, which is why Hawkins and George (2007) developed hierarchical temporal memory (HTM). It should be able to produce generalized representations for similar inputs in its idealized form. HTM should be able to perform time-dependent regression using its learned representations. Many applications utilizing spatiotemporal data would benefit greatly from such a system. Cui et al. (2016) used HTM to predict taxi passenger counts using time-series data. They used HTM for anomaly detection as well (Lavin and Ahmad 2015). The evolving nature of HTM's algorithmic definition and the lack of a formalized mathematical model have hampered its popularity in the machine learning community.

A high-level HTM model of the neocortex's structure and function for example, an HTM region is made up of many columns, each containing multiple cells, just like cortical minicolumns do. A level is made up of regions. The full network shown in Figure 1 is made up of levels stacked in a tree-like structure. To form feedforward and neighboring connections, HTM uses synapses, both proximal and distal.

HTM cortical learning algorithm is the current version (Hawkins et al., 2011). Its two main algorithms are the spatial pooler (SP) and the temporal memory (TM), respectively (TM). Assume that the input is a sparse distributed representation (SDR) (Ahmad and Hawkins, 2015). To put it another way, the SP is a mapping function from one feature domain to another. One SDR per input domain should be used in the feature domain. Similar to self-organizing maps, the algorithm is a type of unsupervised competitive learning algorithm (Rumelhart et al., 1985). (Kohonen, 1982). It learns sequences and predicts outcomes. In this algorithm, connections are formed between previously active cells, following Hebb's rule (Hebb 1949). A sequence can be learned by making those connections. The TM can then make predictions based on its learned knowledge of sequences.

Since HTM is a neocortical abstraction, it lacks a formal mathematical definition. It's difficult to comprehend the algorithm's key features and how to improve it. On the algorithmic part of HTM, there is little research. Hawkins and Ahmad (2016) recently proposed a framework for the TM. Ahmad and Hawkins (2015) formalized the SP in some ways. Lattner (2014) linked the SP to vector quantization to provide an initial understanding of it. However, he did not generalize his work to account for local inhibition. However, certain components of the algorithm, like boosting, were not included in Byrne (2015). Leake et al. (2015) discussed the SP initialization. His focus was on the network initialization, but he did provide some insight into how initialization may impact initial calculations within the network. A complete framework for HTM's SP is provided, as well as examples of

its use in machine learning. With an algorithmic framework, engineers can develop scalable HTM hardware with less effort.

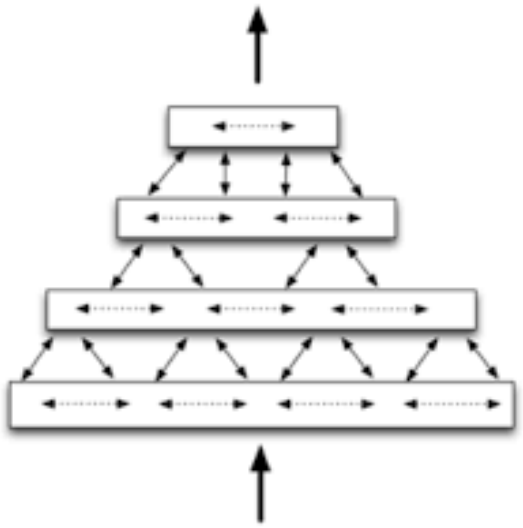

Figure 2: In HTM, there are many levels of organization.

The hierarchical structure of HTM is depicted in Figure 2. In HTM, the cell serves as the fundamental unit of hierarchy. The cells are arranged in rows and columns. These columns unite to form a region, and regions combine to form hierarchies, and hierarchies combine to form hierarchies. If the cells are in the same region or different regions, they might be linked to each other. Cells learn and store the data's temporal sequence, while columns reflect the data's semantics using SDR representations.

They can be linked to other cells in the same area or to regions at higher or lower levels if they are located in the same region. The cells learn and store the temporal sequence of the data, whilst the columns signify the semantics of the data by using SDR representations in their respective columns.There are numerous advantages to learning through the use of a hierarchical structure.

The areas at lower levels could be used to learn some fundamental traits, while the regions at higher levels could be used to learn high-level predictions from the regions at lower levels. In general, higher-altitude places are more resistant to noise. Binary bits are used to represent the data in the input.

Columns in the regions create connections with these binary bits, and each cell in turn establishes connections with cells in other columns based on the context of the temporal sequence of the sensory inputs received in the region in which it is located. It is planned to use the output of the lower area as input for the region in the upper hierarchy.

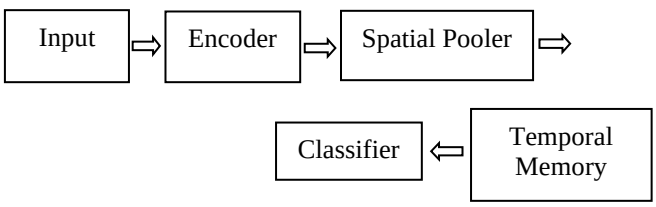

Figure 3: The NuPIC Pipeline's process flow diagram.

The NuPIC framework has an HTM Classifier implementation. Figure 2 depicts the NuPIC framework's workflow. Input data is initially encoded into binary bits by encoders. The encoders use the semantics of the variable to turn the data into binary bits. Bits with the same value will overlap. The spatial pooler is the learning function of the columns in the region.

A stable representation of the input patterns is formed by connecting the input bits to columns. "It is possible for the cells of a column to learn how the input represents time by using a temporal pooler. Using active columns in the hierarchy's upper area, a classifier tries to deduce the output.

Structures and sequences can be learned and represented using memory predictions in HTM regions. A classification result from the HTM classifier is far from satisfactory. However, knnClassifier and CLAClassifier are two classifiers in the NuPIC framework. To determine the output class, the old knnClassifier looked at its closest neighbors and used that information.

For large datasets, this method demands that every single data point be saved in memory, which is extremely time-consuming. Furthermore, there is no guarantee that it will perform as expected. The most recent version of Classifier maps the SDR to the input region and attempts to rebuild the target variable from it. This strategy, on the other hand, does not appear to be very effective.

Finally, with the help of biomimetic algorithms like HTM, programmers want to create an artificial neural network that mimics the brain's structure and functionality. The algorithm's hierarchical ascending layers of cellular areas, as depicted in Fig. 1, allow the network to capture spatial and temporal information. Each region in the HTM is made up of cells, which are grouped in columns to represent biological minicolumns.

Only like the pyramidal neurons, the cell in HTM is just an abstract representation. Many connections connect each pyramidal neuron, which is why they are classified as "proximal, distal, and apical" integration zones (or segments).

An important role of the proximal segment is that it receives feed-forward input, such as watching cell activity at lower levels of the hierarchy or receiving sensory inputs. Neuronal action potentials are often generated when proximal activity is sensed. When it comes to the cell's surroundings (contextual input) and higher-level hierarchy (feedback input), the distal and apical portions are devoted to these tasks.

II. TEST CASES & RESULTS

Through the GetPredictedInputValues() method, I've worked on or tested three public methods. The following is a breakdown of the methods' workings.

Some functions, on the other hand, didn't necessitate unit testing because they don't have any expected return values.

The following are examples of some of those techniques. ActiveMap2.Clear() is present in the initial method ClearState, but it lacks the functionality needed to be tested in a unit test. Learn(TIN input, Cell[] output) is the second function's Learn(TIN input, Cell[] output) where the learning process begins after this technique, from whence we acquired the sequences that determine the HTM region parameters.

Scalar encoder, spatial pooler, temporal memory, input, and other encoders all function simultaneously in these locations, and the hierarchical temporal memories play a crucial part in obtaining the required output from the system.

```
public void ClearState()
 {
        m_ActiveMap2.Clear();
 }
public void Learn(TIN input, Cell[] output)
 {
 }
```

Table 1: Non-returning htmClassifier methods

```
namespace NeoCortexApi.Classifiers
{
   public class HtmClassifier<TIN, TOUT> :
IClassifier<TIN, TOUT>
   {
      private int maxRecordedElements = 10;

      private List<TIN> inputSequence = new List<TIN>();

      private Dictionary<int[], int> inputSequenceMap =
new Dictionary<int[], int>();

      private Dictionary<TIN, List<int[]>> m_AllInputs =
new Dictionary<TIN, List<int[]>>();
      public void ClearState()
   private bool ContainsSdr(TIN input, int[] sdr)
private int GetBestMatch(TIN input, int[] cellIndicies, out
double similarity, out int[] bestSdr)
public void Learn(TIN input, Cell[] output)
public List<ClassifierResult<TIN>>
GetPredictedInputValues(Cell[] predictiveCells, short
howMany = 1)
public TIN GetPredictedInputValue(Cell[] predictiveCells)
public void TraceState(string fileName = null)
private string ComputeHash(byte[] rawData)
private static byte[] FlatArray(Cell[] output)
private static int[] GetCellIndicies(Cell[] output)
private int PredictNextValue(int[] activeArr, int[]
predictedArr)
}
```

This code snippet is the public class of HtmClassifier where apart from GetPredictedInputValues, all of the methods in the HtmClassifier class are either private or have no return type. As a result, verifying the entire HtmClassifier class's methodology is difficult. So, I build GetPredictedInputValues unit testing methods.

- The Initial

The very first standard method on which I worked is CheckHowManyOfGetPredictedInputValues() where I have checked that the data sent through 'howMany' parameters are equal to the 'res' variable value which is counted by Count==howMany. So, the goal we set for our GetPredictedInputValues() method is being met by this unit test.

With HPC, the TM will learn cells for patterns quickly in the pretrained SP with HPC. In that situation, the starting sequence 4-5-6 might have the same SDR as the starting sequence 1-2-3-4-5-6, and so forth. As a result, 4-5-6 will be returned instead of 1-2-3-4-5-6, as seen in the diagram below. The first matching sequence is always returned by HtmClassifier. The reason for this is that 4-5-6 will be memorized first, therefore it will match as the first one.

```
public void CheckHowManyOfGetPredictedInputValues(int
howMany)
      {
         sequences = new Dictionary<string,
List<double>>();
         sequences.Add("S1", new List<double>(new
double[] { 0.0, 1.0, 2.0, 3.0, 4.0, 2.0, 5.0, }));

         LearnHtmClassifier();

         var lyrOut = layer.Compute(1, false) as
ComputeCycle;

         var res =
htmClassifier.GetPredictedInputValues(lyrOut.PredictiveCe
lls.ToArray(), Convert.ToInt16(howMany));

         Assert.IsTrue(res.Count==howMany);
      }
```

1. This code snippet checks that the 'howMany' parameter value is equal to the Count==howMany variable value.

- The Second

The next function, NoExceptionIfCellsCountIsZero(), demonstrates that if we send cells count '0', it checks for the exception. CheckHowManyOfGetPredictedInputValues(), on the other hand, does not throw an exception in this instance. So, it is maintained that the method htm.Classifier.GetPredictedInputValues() returns the precise number of predicted input values that we sought to forecast.

```
public void NoExceptionIfCellsCountIsZero()
      {
         Cell[] cells = new Cell[0];
         var res =
htmClassifier.GetPredictedInputValues(cells, 3);
         Assert.AreEqual(res.Count,0);
      }
```

2. NoExceptionIfCellsCountIsZero() where it is showing that if we send cells count '0', it checks for the exception.

- The Third

The third and the last unit test plays very important role for the HTMClassifier. I examine the input sequence and output sequence is same after learning our data through the method LearHtmClassifier(). For example, here in the code below is demonstrating that the input sequence-3 is equal to output sequence-3 after checking through the predicted method, GetPredictedInputValues(). Assuming the previous step correctly anticipated the current value, we have a match. It is impossible to foresee the sequence's first element (a single element). If we achieve 30 repetitions with 100percentage accuracy, the trial is over. As a result, the first element will always begin at the beginning of the learning process.

```
public void CheckNextValueIsNotEmpty()
      {
         sequences = new Dictionary<string,
List<double>>();
         sequences.Add("S1", new List<double>(new
double[] { 0.0, 1.0, 2.0, 3.0, 4.0, 2.0, 5.0, }));

         LearnHtmClassifier();

         //var tm = layer1.HtmModules.FirstOrDefault(m =>
m.Value is TemporalMemory);
         //((TemporalMemory)tm.Value).Reset(mem);

         var lyrOut = layer.Compute(1, false) as
ComputeCycle;

         var res =
htmClassifier.GetPredictedInputValues(lyrOut.PredictiveCe
lls.ToArray(), 3);

         var tokens = res.First().PredictedInput.Split('_');
         var tokens2 = res.First().PredictedInput.Split('-');
         var predictValue =
Convert.ToInt32(tokens2[tokens.Length - 1]);
         Assert.IsTrue(predictValue > 0);
      }
```

3. It tests the input sequence and output sequence are same after learning our data through the method LearHtmClassifier().

W/N is the ratio of the width to the number of input bits in an html document (Htm Sparsity). Iteratively running this unit test saves the cycle in which we achieve a 100 percent match for the first time at Sparsity=0.18. The values of W and N are flexible, however the ratio must be 0.18 or less. With the exception of the parent/outer loops, which are defined with the number of readings desired in the result, this program has two loops (a loop within a loop). Despite the fact that the child loop/inner loop has 460 cycles, it is terminated as soon as we receive a 100 percent match, i.e. for max=10, we receive ten out of ten correct answers. The parent loop is then incremented by one, and the loop continues for the specified number of times (in our case we used 1000 - 10000 loops). We discovered that the optimal HTM Sparsity for max=10 is 0.18" in this case.

III. Discussion

Using Spatial Pooler, Scalar Encoder, and Temporal Memory (HTM), this work proposes a machine learning solution in which the unit test are developed. Machine intelligence has recently evolved a new paradigm in the form of Hierarchical Temporal Memory (HTM) theory, which represents the structural and algorithmic components of the neocortex. I propose three new unit tests, which I validate and benchmark against existing ones. Using datasets from the spatial pooler, the scalar encoder, and the temporal memory repository, I evaluated the performance of my suggested unit tests for the HTM-Classifier. The HTM algorithm's inference of patterns and structures that the system recognizes still needs a lot of development.

My future research will focus on further enhancing the classifier design and investigating the application of the HTM-Classifier for HTM Classifier issues, among other things. In the realm of machine learning, Deep Learning is a vital component of the use of Artificial Intelligence. The effectiveness of these systems is determined by their behavior and performance. The goal of my future research will be to improve the performance of this sort of system by combining a Random Forest and HTM Cortical Learning Algorithm version.

IV. References

[1] Abdullah M. Zyarah, Kevin Gomez, and Dhireesha Kudithipudi (2000). End-to-End Memristive HTM System for Pattern Recognition and Sequence Prediction.

[2] Dobrick, D. (2021). *Improved HTM Spatial Pooler with.* Retrieved from University of Plymouth: https://pearl.plymouth.ac.uk/bitstream/handle/10026.1/17130/Improved%20HTM%20spatial%20pooler%20with%20homeostatic%20plasticity%20control.pdf?sequence=1&isAllowed=y

[3] Jie Cheng & Russell Greiner (2001). Learning Bayesian Belief Network Classifiers: Algorithms and System

[4] Xia Zhituo, Ruan Hao, Wang Hao (2012). A Content-Based Image Retrieval System Using Multiple Hierarchical Temporal Memory Classifiers

[5] Santiago Fernández, Alex Graves, Juergen Schmidhuber (2008). Phoneme recognition in TIMIT with BLSTM-CTC

[6] Yuwei Cui, Chetan Surpur, Subutai Ahmad, and Jeff Hawkins (2016).
A comparative study of HTM and other neural network models for online sequence learning with streaming data.

[7] S. Hochreiter and J. Schmidhuber (1970), Long short-term memory, Neural Computing.