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

Approve Prediction of Multisequence Learning

Vasanthakumar Ramesh Lakshmi

vasantkumar.lakshmi@stud.fra-uas.de

***Abstract*—Most artificial networks today depend upon dense representations, whereas biological networks rely on sparse representations. Hierarchical Temporal Memory (HTM) is a specific realization of the Thousands Brains Theory that generates motor commands for interacting with the environment and testing predictions. In this experiment, we will calculate accuracy of Multisequence Learning algorithm which is based on HTM, this accuracy is calculated with randomly generated sequences or dataset and the sub-sequences or test sequences which are subset of those sequences or dataset. To complete this experiment, we see how randomly generated sequences are learned whose learning accuracy close to 100% and calculated the prediction accuracy of the subsequence to compare the results. The experiment in total learned 1000 different sequences of 25 elements and where tested with 200 subsequence which resulted into 36 test sequences giving 100% accuracy, 58 test sequences giving 75% accuracy and final average accuracy of 200 runs is 61.64% which can be consider decent as the experiment tried to learn the randomness function.**

Keywords—AI, HTM, sparse dense representation , SDRs, spatial pooler, API, multisequence learning, neocortex.

I. INTRODUCTION

The neocortex is the seat of intelligent thought in the mammalian brain. High level vision, hearing, touch, movement, language, and planning are all performed by the neocortex. Given such a diverse suite of cognitive functions, you might expect the neocortex to implement an equally diverse suite of specialized neural algorithms. This is not the case. The neocortex displays a remarkably uniform pattern of neural circuitry. The biological evidence suggests that the neocortex implements a common set of algorithms to perform many different intelligence functions [1].

Multi-sequence learning using HTM involves training the algorithm to recognize and predict patterns across multiple input sequences. To implement multi-sequence learning using HTM, we first need to encode the input data into Sparse Distributed Representations (SDRs), which can be done using scalar encoder. Once the data is encoded, the spatial pooler creates sparse representations of the input sequences, which are then fed into the temporal memory component for learning and prediction. Multisequence learning using HTM is a powerful approach for recognizing and predicting patterns across multiple input sequences.

In this project, we have tried to implement new methods along Multisequence Learning algorithm [2]. The new methods are to create dataset and test dataset as per configuration, automatically reading the dataset from the given path, we also have test data in another file which needs to be read for later testing the subsequence in similar. Multisequence Learning takes multiple sequences for learning. After learning is completed, calculation of accuracy of predicted element with the test sequence. These test sequences are subsets of the sequences used during learning.

II. LITERATURE SURVEY

A. *Sequence*

A particular order in which related things follow each other.

B. *Hierarchical Temporal Memory*

The objective of HTM is to emulate the hierarchical structure and learning process of the brain. It is comprised of a network of nodes organized hierarchically, where each node corresponds to a group of neurons in the neocortex. These nodes acquire the ability to identify patterns in sensory information and generate predictions based on their prior experiences. Subsequently, the predictions are evaluated against the input data to refine the node's model and enhance its predictive precision.

As per Hawkins, the neocortex learns and makes predictions by forming a hierarchical structure of columns, each containing a set of neurons that recognize patterns in sensory input. These columns communicate with each other in a hierarchical manner, with higher-level columns representing more abstract concepts [3]. An SDR consists of a large array of bits of which most are zeros, and a few are ones. Each bit carries some semantic meaning so if two SDRs have more than a few overlapping one-bits, then those two SDRs have similar meanings [4].

C. *Sparse Distributed Representation*

In the context of HTM, SDRs are used to represent patterns of activity in the network. Each input to the network is transformed into an SDR, which is then processed by the network's hierarchy of nodes to make predictions about future input.

Hawkins and Ahmad proposed that SDRs, which are binary vectors with a small number of active bits (ones) out of a large number of total bits, are a natural way to represent sparse, distributed patterns of activity in the neocortex [5].

D. *Encoder*

The encoder is designed to take in raw data inputs and convert them into SDRs that represent the relevant features of the input data. The encoding process involves several steps, including dimensionality reduction, noise reduction, and normalization. The encoder also learns to recognize patterns in the input data and adapts to changes in the input distribution over time.

The SP encoder is designed to handle temporal data and uses a sliding window approach to capture temporal patterns in the input data. It first converts the input data into a continuous stream of binary values, which are then fed into the HTM network as a sequence of SDRs [6].

E. *Spatial Pooler*

The Spatial Pooler is a type of unsupervised learning algorithm that takes in high-dimensional input data and creates a lower-dimensional SDR that represents the relevant features of the input data. It does this by first assigning a random set of weights to each input feature and then computing the overlap between each input and the set of weights. The features with the highest overlap are then selected and included in the SDR [7].

F. *Temporal Memory*

The functioning of Temporal Memory involves preserving a collection of active cells that embody the present context of the input data. Upon receiving fresh input patterns, the active cells undergo modifications according to the similarity between the input and the current context. Additionally, the algorithm manages a set of connections between cells that represent the sequence of patterns that were encountered previously [7].

Using these connections, Temporal Memory can predict the next likely input pattern based on the current context. If the prediction is correct, the algorithm reinforces the connections between the cells that were active during the predicted sequence. If the prediction is incorrect, the algorithm adjusts the connections to reduce the likelihood of that sequence occurring in the future.

G. *Multisequence Learning*

Multisequence learning is an HTM based algorithm that involves learning and predicting multiple sequences of patterns simultaneously. Multisequence learning is achieved by using separate Temporal Memory modules to learn and predict each sequence of patterns.

The key idea behind this approach [8] is to use a hierarchical structure of nodes to learn and predict sequences of patterns at different levels of abstraction. At the lowest level, each Temporal Memory module learns and predicts the raw sensory input from a single modality. At higher levels, the nodes learn and predict sequences of patterns that combine information from multiple modalities.

III. METHODOLOGY

In the experiment, we tried to achieve calculating accuracy of the of the predicted sequence with respected to actual sequence. Which means that for a given test sequence, the element preceding in the sequence should match the predicted element in the sequence. To add on more features, the dataset used to train the sequence is automated where a configuration is used to create/save the training sequences and the test sequences are created out of these training sequences, in simpler terms the test sequences are subsequences of training sequences or subset of training sequences. Another feature is that all the logs of training sequences with training accuracy and logs of testing the sequences during predictions are saved to report file for analysis and concluding results.

The Figure 1 gives an overview of the experiment and helps to understand the flow of data in the experiment.

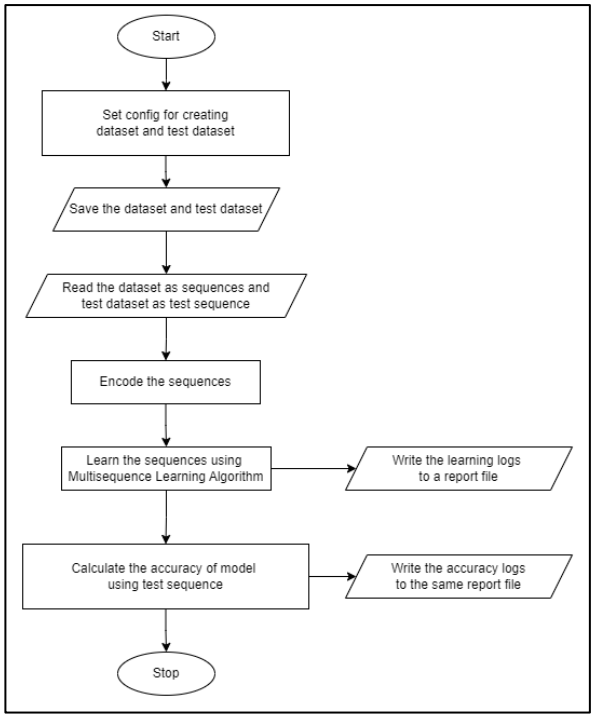

Figure 1 Flow chart of the experiment

The following methods and models were used to preprocess, learn and postprocess as part of the experiment.

A. Sequence model

A sequence is a data model which consists of sequence name and numeric sequence. The object of list of sequence data model is nothing but multiple sequences. The subsequence is subset of sequence and hence used same data model shown in Source code 1.

```
// Model of sequence
public class Sequence
{
   // name of sequence
   public String name { get; set; }
   // sequence itself
   public int[] data { get; set; }
}
```

Source code 1 Data Model of the Sequence

Below is a sample of Dataset 1 used as training dataset and Dataset 2 as the test dataset used to calculate accuracy during prediction.

```
[
  {
    "name": "S1",
    "data": [ 4, 5, 6, 8, 9, 10…… 32, 33, 34 ]
  },
  {
    "name": "S2",
    "data": [ 4, 5, 6, 7, 8, 9 …… 32, 33, 34 ]
  },
  {
    "name": "S3",
    "data": [ 5, 6, 7, 8, 9, 10 ……  31, 33, 34 ]
  }
]
```

Dataset 1 Training dataset

```
[
  {
    "name": "T1",
    "data": [ 24, 25, 27, 28, 29 ]
  },
  {
    "name": "T2",
    "data": [ 13, 15, 16, 17, 18 ]
  },
  {
    "name": "T3",
    "data": [ 19, 21, 22, 23, 24 ]
  }
]
```

Dataset 2 Testing dataset

B. ConfigOfSequence model

This is another data model used as configuration for creating the data/sequence in Sequence model.

```
// Model of configuration used to create sequence
public class ConfigOfSequence
{
   // count of the sequence
   public int count { get; set; }
   // length/size of each sequence
   public int size { get; set; }
   // length/size of each test sequence
   public int testSize { get; set; }
   // start value of sequence
   public int startVal { get; set; }
   // end value of sequence
   public int endVal { get; set; }

   // constructor
   public ConfigOfSequence(int Count, int Size,
      int TestSize, int StartVal, int EndVal)
   {
      this.count = Count;
      this.size = Size + 3;
      this.testSize = TestSize;
      this.startVal = StartVal;
      this.endVal = EndVal;
   }
}
```

Source code 2 Data Model of the ConfigOfSequence

The constructor in Source code 2 which is data model of ConfigOfSequence adds 3 to the size of Sequence since when the sequence is generated 3 elements are randomly deleted to add more random factor.

C. FetchHTMConfig method

Here in

Source code 3, we save the HTMConfig which is used for Hierarchical Temporal Memory to Connections.

```
/// <summary>
/// HTM Config for creating Connections
/// </summary>
/// <param name="inputBits">input bits</param>
/// <param name="numColumns">number of
columns</param>
/// <returns>Object of HTMConfig</returns>
public static HtmConfig GetHtmConfig(int inputBits,
int numColumns)
{
   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,

        // Learning is slower than forgetting in this case.
        PermanenceDecrement = 0.25,
        PermanenceIncrement = 0.15,

        // Used by punishing of segments.
        PredictedSegmentDecrement = 0.1
    };

    return cfg;
}
```

Source code 3 FetchHTMConfig method

Through this provision, multiple configs can be created, and different models can be trained. This gives some space to scale up the experiment.

D. GetEncoder method

This method is used to get the encoder setting, as per *inputBits* and *max* which is maximum value to be encoded. We are using *ScalarEncoder* [2] since we are encoding numeric values.

E. ReadDataset method

This method is used to read the dataset file when the full path is given. It returns an object of list of Sequence which is later used to save dataset as well as test dataset. The method used JSON deserializer to read the dataset file.

F. SaveDataset method

This method is used to save the dataset whether it be training dataset or test dataset. It takes in object of list of Sequence and used JSON serializer to serialize the object and saves the data.

G. GetLogFile and WriteLog methods

The *GetLogFile* method is used to generate unique log file name and empty file which is used later to save training logs as well as accuracy logs when predicting the sequence. The WriteLog method appends the logs which means formatting of logs needs to be done before writing to file.

H. IsCreateDatasetValid method

This method is used to validate the *ConfigOfSequence* model as seen in Source code 2, some cases such as minimum size of sequence must be 3 since a single/double element can not be treated as sequence, or maybe the start value of sequence can not be greater than the end sequence. It can also be used to force experiments to run a minimum length of 9 elements in a sequence and so on.

I. CreateDataset and CreateSequences methods

The *CreateDataset* method is a wrapper method to *CreateSequences* method which validates the *ConfigOfSequence* before starting to create any sequences.

J. getSyntheticData method

This is the most important method which is responsible for creating sequences as per configuration and randomly removes 3 elements to add more randomness to the dataset.

K. IsCreateTestDatasetValid method

This method is similar to *IsCreateDatasetValid* and is responsible for validating configurations related to creating test dataset. Some of the checks are like the size of test dataset can not be greater than the size of training dataset.

L. CreateTestDataset, SelectRandomSequence and CreateTestSequence methods

The method CreateTestDataset is a wrapper method to *SelectRandomSequence* and *CreateTestSequence*, which also calls *IsCreateTestDatasetValid* method to check that before creating any test sequences is the configuration valid or not.

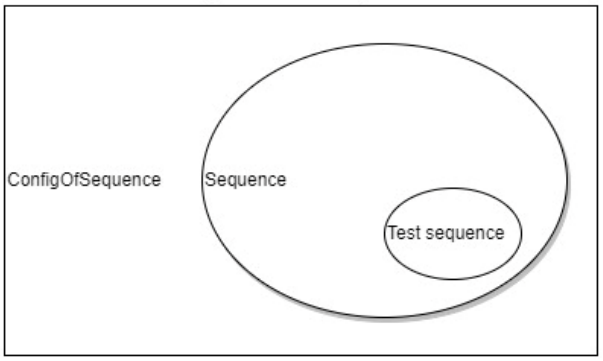

Figure 2 Relation between Sequence and Test Sequence

All the test sequences are created from training sequences and are part of the sequence or subsequence. Which also means that the test sequences are subset of the train sequences. The following is represented in Figure 2 using the Venn's diagram.

When test sequences are created, we randomly select the train sequence randomly using the *SelectRandomSequence* method and then *CreateTestSequence* method selects a subsequence from it.

M. CreateSaveData method

This a wrapper method to define all the parameters used to create training and test dataset and it saves the files in a dataset directory. The *ConfigOfSequence* is set and initialized in this method.

N. PredictNextElement method

This method is our main task of the project where the test sequence calls the predict function to predict the next element and compare predicted next element with the actual next element.

The accuracy percentage is calculated as the number of matching predictions made divided by total number of predictions made for an element in the subsequence multiplied by 100%.

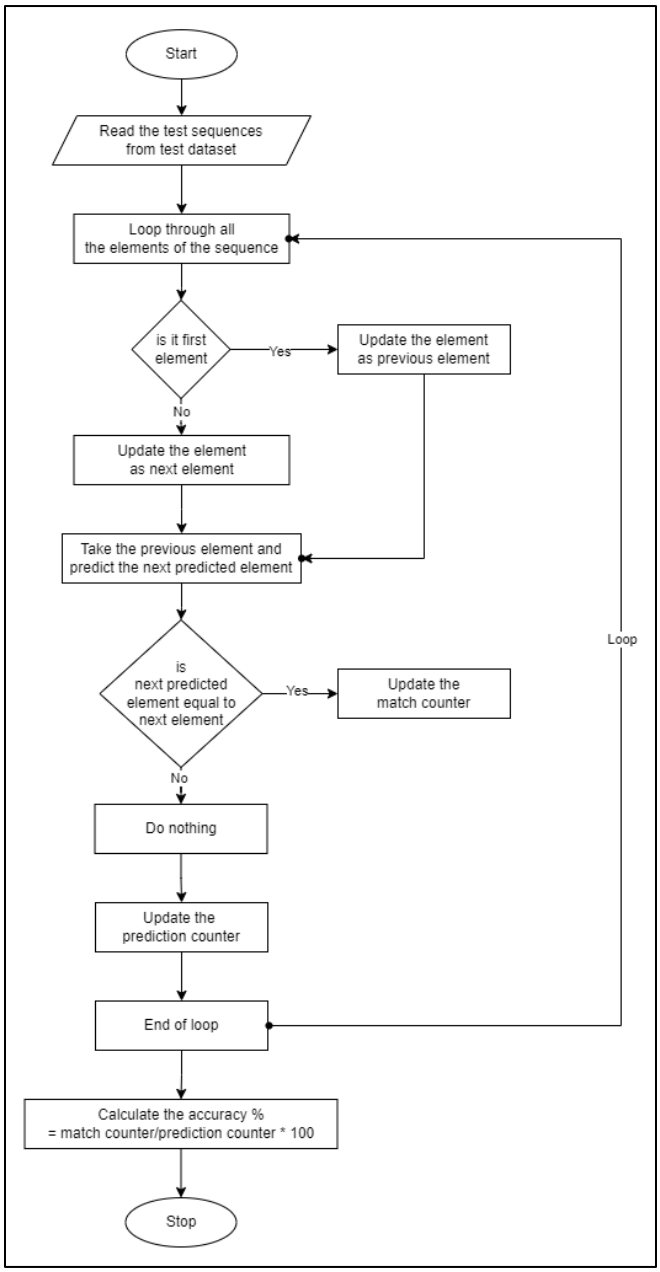

Figure 3 Flow chart for calculating accuracy

The Figure 3 shows detailed flow of the process of calculating the accuracy.

IV. DISCUSSIONS

The experiment was running around 20 times which learned over 1000 different sequences in total of size 25 each and tested with 200 total sub-sequence. Each run has a different dataset and test dataset which can be found in the dataset directory and full report of runs are in reports directory.

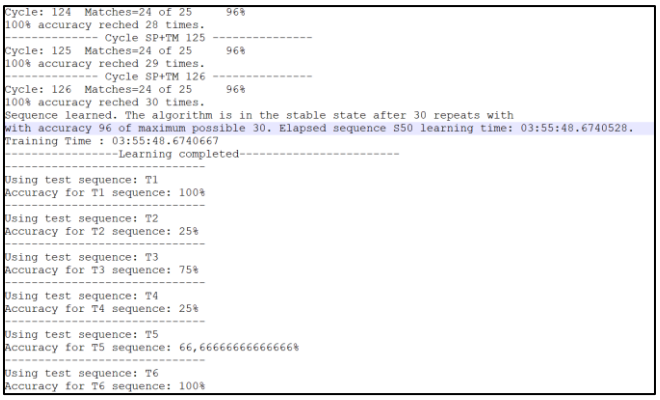

Figure 4 Sample report used for detailed analysis

Figure 4 which is the sample report shows us the training accuracy as the sequence was learned efficiently and was stopped learning after there was nothing to learn and in the end the test sequence was used to predict the accuracy of the learned model.

Average time for each run of experiment was around 3.5-4hr. Counting some stats from the plot as 8 runs have 0% accuracy, 36 runs have 100% accuracy which we expect, 58 runs have 75% accuracy. The average of the 200 runs was 61.64%, which we expect to be decent. The Figure 5 gives us visual representation of the accuracy of all 200 runs.

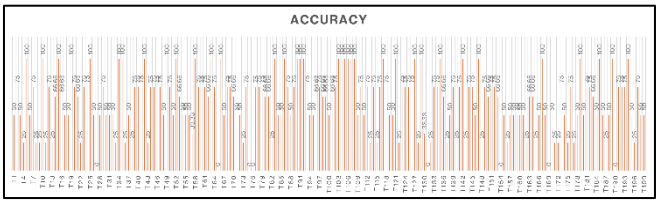

Figure 5 Visual representation of accuracy for all runs

Further improvements can be made as the *ConfigOfSequence* is initialized as parameterized constructor and populating values in it can be more flexible as a configuration file and ran run will multiple configurations. For flexibility configurations will help to give results based on variety and can be run in cloud environments.

V. REFERENCES

[1] J. Hawkins, “Hierarchical Temporal Memory White Paper,” Numenta, 12 September 2011. [Online]. Available: https://numenta.com/neuroscience-research/research-publications/papers/hierarchical-temporal-memory-white-paper/.

[2] D. Dobric, "NeoCortexApi : https://github.com/ddobric/neocortexapi".

[3] J. Hawkins and S. Blakeslee, On Intelligence, An Owl Book.

[4] S. Puddy, "Encoding Data for HTM Systems," Arxiv, 18 February 2016. [Online]. Available: https://arxiv.org/abs/1602.05925.

[5] J. Hawkins and S. Ahmad, "Why Neurons Have Thousands of Synapses, a Theory of Sequence Memory in Neocortex," 30 March 2016. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fncir.2016.00023/full.

[6] S. Ahmad, *Hierarchical Temporal Memory*, Scholarpedia, 2012.

[7] J. Hawkins and G. Dileep, *Hierarchical Temporal Memory*, 2009.

[8] S. Ahmad, *Real-Time Multimodal Integration in a Hierarchical Temporal Memory Network*, 2015.