**Implement HTM Persistance**

Nabeela Maham

nabeela.maham@stud.fra-uas.de

Kizito Onyema  
kizito.onyema@stud.fra-uas.de

Abstract— This report describes mainly the implementation of the serialization and deserialization of some of the class libraries in the NeoCortexApi including but not limited to *SparseObjectMatrix, InMemoryDistributedDictionary, SparseBinaryMatrix*

By this implementation, the NeocortexApi can Serialize and Deserialize its HTM Objects so that it can recall from where it stops. The Serializer records the instances of some of the HTM module to the stream and the Deserializer helps in creating the instance from the stream. The HTM keeps record of the instances of the HTM modules using the Streamwriter and then continues again from it’s last instance with the Streamreader. This is a new feature in the NeoCortexApi, as the current implemented and conducted serializations are still limited. As the NeoCortexApi is the mirror of a Human Brain, this work describes an aspect of the real life where a Human’s Brain coordinates itself and reminds the body of it last work continues his/her Work where it stopped; this emphasizes the NeoCortexApi’s cognitive ability, remembrance, continuity and efficiency.

Keywords— - serialization, deserilization, persistence

# **Introduction**

Hierarchical temporal memory (HTM) is a biologically constrained machine intelligence technology developed by Numenta. Originally described in the 2004 book. On Intelligence by Jeff Hawkins with Sandra Blakeslee, HTM is the distributed building and primarily used today for anomaly detection in streaming data. The technology is based on neuroscience and the physiology and interaction of pyramidal neurons in the neocortex of the mammalian (in particular, human) brain. The HTM algorithm is based on the well understood principles and core building blocks of the Thousand Brains Theory. In particular, it focuses on three main properties: sequence learning, continual learning and sparse distributed representations.

The Thousand Brains Theory is the core model-based, sensory-motor framework of intelligence putting together the neuroscience research developed over the almost two decades of research at Numenta and the Redwood Neuroscience Institute (founded in 2002 by Jeff Hawkins). It provides a unique interpretation of the high-level computation thought to happen throughout the neocortex and giving rise to intelligent behaviors. See Fig 1.1 and 1.2



*Fig 1.1 HTM Neuron [1]*



*Fig 1.2 HTM Structure [2]*

Since the HTM is based on the ideology of the neocortex in the Human Brain which has the ability of remembrance and continuity, such a function can also be created implemented as Persistence in the NeoCortexApi which could be achieved through serialization and deserialization; this means that the instances created by the HTM Modules a recorded at a point, and also willing to continue at any point in time with these points. We apply the serialization to save the data in the form of text file. We can use that data next time when you continue the experiment again in deserialization.

For the serialization, we have used the class HTMSerializer2.cs. There, we have applied different functions. It has serialized the property of integer, double, string, long, boolean and fetch the values for the deserialization. It can serialize the non-primitive data types such as cell, dictionary distal dendrite, synapse and list. Deserialize the value when we need to continue the experiment again.

# **Methods**



*Fig 1.3 Serialization and Deserialization [3]*

**Serialization**

Serialization is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization.

## How serialization works

The object is serialized to a stream that carries the data. The stream may also have information about the object's type, such as its version, culture, and assembly name. From that stream, the object can be stored in a database, a file, or memory.

### Uses for serialization

Serialization allows the developer to save the state of an object and re-create it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform actions such as:

* Sending the object to a remote application by using a web service
* Passing an object from one domain to another

Deserialization

Deserialization is the reverse process of serialization. It means you can read the object from byte stream. Here, we are going to use the method to deserialize the stream. See Fig 1.3

### **Classes Serialized And Deserialized**

***1. SparseObjectMatrix,***

Allows storage of array data in sparse form, meaning that the indexes of the data stored are maintained while empty indexes are not. This allows savings in memory and computational efficiency because iterative algorithms need only query indexes containing valid data.

*Listing 1 [SparseObjectMatrix](https://github.com/onyemakn/neocortexapi/blob/fda1274afab58109c1567b65ed5da21468c91d15/source/HtmPersistence/UnitTest2.cs" \l "L13-L36)*

// Create SparseObjectMatrix

// either by dicrect creation or running experiment

int[] dimensions = { 10, 10 };

SparseObjectMatrix<int[]> matrix = new(dimensions, false);

// Serialize

using (StreamWriter sw = new StreamWriter("ser.txt"))

{

matrix.Serialize(sw);

}

// Deserizlize

SparseObjectMatrix<int[]> matrixNew = new();

using (StreamReader sr = new StreamReader("ser.txt"))

{

matrixNew = SparseObjectMatrix<int[]>.Deserialize(sr);

HtmSerializer2.IsEqual(matrix, matrixNew);

}

In the above *SparseObjectMatrix* is serialized and deserialized, using StreamWriter, the instances of the SparseObjaectMatrix class are saved and written in a text file "ser.txt". Also, using StreamReader, the instances of the SparseObjaectMatrix class are read and saved in a text file "ser.txt"

***2. InMemoryDistributedDictionary***

This creates a larger dictionary across mutliple dictionaries. And it is aids for the purpose of testing. For this dictionary its special case deals with the number of nodes = 1. In this case dictionary is redused to a single dictionary, which matches with the original none-distributed implementation of SP and TM.

In here, the StreamWriter saves the instances of the numbers and values of the InMemoryDistributedDictionary class and writes into a file "InMem.txt". The StreamReader reads the instances of the keys and key values of the InMemoryDistributedDictionary class and saves it into a file "InMem.txt"

*Listing 2* [*InMemoryDistributedDictionary*](https://github.com/onyemakn/neocortexapi/blob/6b1b09ab6018053c421767f1e374d0fefc25dda4/source/NeoCortexEntities/InMemoryDistributedDictionary.cs#L403-L538)

public void Serialize(StreamWriter writer)

{

HtmSerializer2 ser = new HtmSerializer2();

ser.SerializeBegin(nameof(InMemoryDistributedDictionary<TKey, TValue>), writer);

// index 0

ser.SerializeValue(this.numElements, writer);

// index 1

ser.SerializeValue(this.currentDictIndex, writer);

// index 2

ser.SerializeValue(this.currentIndex, writer);

// index 3

ser.SerializeValue(this.dictCount, writer);

// Serialize dicList

ser.SerializeBegin(nameof(dictList), writer);

// index of dictionaries

int dictCnt = 0;

// looping through dictionaries in dictList

foreach (var dict in dictList)

{

ser.SerializeValue(dictCnt, writer);

foreach (var item in dict)

{

if (typeof(TKey) == typeof(int))

{

// Create Element with syntax Key\_\_Value

var writeValue = item.Key.ToString()+"\_\_"+item.Value.ToString();

ser.SerializeValue(writeValue, writer);

}

else

throw new NotSupportedException();

}

dictCnt++;

}

if (this.htmConfig != null)

{ this.htmConfig.Serialize(writer); }

ser.SerializeEnd(nameof(InMemoryDistributedDictionary<TKey, TValue>), writer);

}

public static InMemoryDistributedDictionary<int, int> Deserialize(StreamReader sr)

{

InMemoryDistributedDictionary<int, int> newDict = new InMemoryDistributedDictionary<int, int>();

***3. SparseBinaryMatrix***

Implements the sparse matrix which contains binary integer values only.

Here, the *SparseBinaryMatrix* is serialized and deserialized, using StreamWriter, the instances of the SparseObjaectMatrix class are saved and written in a text file "ser.txt". Also, using StreamReader, the instances of the *SparseBinaryMatrix* class are read and saved in a text file "ser.txt". This behavior can be seen in *Code Sample [6]*

*Listing 3 [SparseBinaryMatrix](https://github.com/onyemakn/neocortexapi/blob/fda1274afab58109c1567b65ed5da21468c91d15/source/HtmPersistence/UnitTest2.cs" \l "L64-L88)*

// Create SParse BinarySparseMatrix

// either by dicrect creation or running experiment

int[] dimensions = { 100 , 100 };

//IDistributedDictionary<int, int[]> dict = new();

SparseBinaryMatrix binaryMatrix = new(dimensions, false);

// Serialize

using (StreamWriter sw = new StreamWriter("Binary.txt"))

{

binaryMatrix.Serialize(sw);

}

// Deserizlize

SparseBinaryMatrix newBinary = new();

using (StreamReader sr = new StreamReader("Binary.txt"))

{

newBinary = SparseBinaryMatrix.Deserialize(sr);

HtmSerializer2.IsEqual(binaryMatrix, newBinary);

}

**HTM Models Called During Serialization**

***1. ModuleTopology***

At the initiation of the serialization process, the serialized class calls some of the existing HTM modules and Serializers which are in Heirachy. The ModuleTopology is part of the HTMmoduleTopology which gets and sets the topology of the HTM Models and then called by SparseObjectMatrix and SparseBinaryMatrix during the Serialization Process. These modules have Hierarchical functions in the HTM structure which also contributes to the creating of the instances of the HTM.

*Listing 4* [*ModuleTopology*](https://github.com/onyemakn/neocortexapi/blob/fda1274afab58109c1567b65ed5da21468c91d15/source/NeoCortexApi/Entities/SparseObjectMatrix.cs#L251-L259)

ser.SerializeBegin(nameof(SparseObjectMatrix<T>), writer);

ser.SerializeValue(this.IsRemotelyDistributed, writer);

if (this.ModuleTopology != null)

{ this.ModuleTopology.Serialize(writer); }

***2. htmConfig***

The *htmConfig* gets a list of Objects in the HTM Modules associated with Keys and Key values. The *htmConfig* is part of the HTM Module in the NeoCortexApi it’s main functionality is to aid the implementation of dictionary related functionalities in the HTM. With the list of it’s keys and Key values the *InMemoryDistributedDictionary* can save the various states to the Stream.

*Listing 5* [*htmConfig*](https://github.com/onyemakn/neocortexapi/blob/fda1274afab58109c1567b65ed5da21468c91d15/source/NeoCortexEntities/InMemoryDistributedDictionary.cs#L367-L379)

if (this.htmConfig != null)

{ this.htmConfig.Serialize(writer); }

ser.SerializeEnd(nameof(InMemoryDistributedDictionary<TKey, TValue>), writer);

***3. InMemoryArray***

The *InMemoryArray* Gets or sets the element at the specified index in the HTM Module Structure. The *InMemoryArray* is part of the HTM Module in the NeoCortexApi which holds the dimensions of the objects of the HTM models. These dimensions and indexes are part of the states saved in the stream for Serialization and Deserialization.

*Listing 6* [*InMemoryArray*](https://github.com/onyemakn/neocortexapi/blob/fda1274afab58109c1567b65ed5da21468c91d15/source/NeoCortexEntities/InMemoryArray.cs#L227-L236)

ser.SerializeBegin(nameof(InMemoryArray), writer);

ser.SerializeValue(this.backingArray, writer);

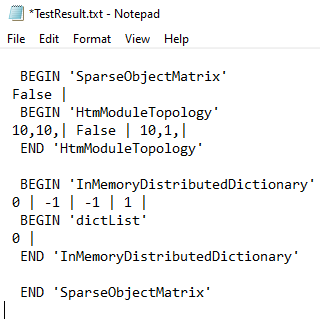
ser.SerializeValue(this.dimensions, writer);

ser.SerializeValue(this.numOfNodes, writer);

There are calls of distributed Hierarchy in the HTM when Serialization and Deserialization is initiated. These Hierarchies are the skeleton and building blocks of the various chains of the HTM and these are the various points from which the instances are drawn from. In the process of initiating Serialization, if an module is not clearly described and has a parameter that is not properly defined, it affects the whole system and create errors.

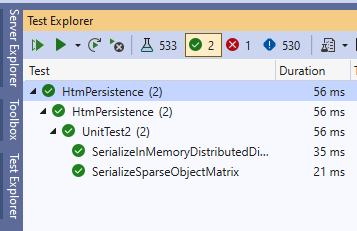
# **Results**

The following below shows the result of the implemented classes for Serialization and Deserialization in NeoCortexApi *SparseObjectMatrix, InMemoryDistributedDictionary, SparseBinaryDictionary* for Serialization and Deserialization in the NeoCortexApi. ThE Tests were conducted using UnitTesting method which also had some errors.

******

*Fig 1.4 UnitTest Result*

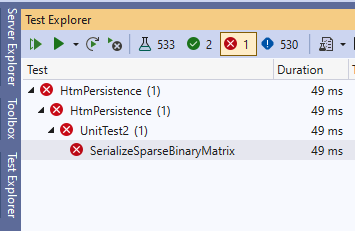
The above shows the Serialized and Deserialized values of the implemented classes. As you can see, some already existing HTM Serializers where also called in the process as described in this Report. The InMemoryDistributedDictionary shows some Key values and “dictList” parameter which it calls in the process of Serializing an Deserializing the Classes.



*Fig 1.5 Result From The Test Explorer*

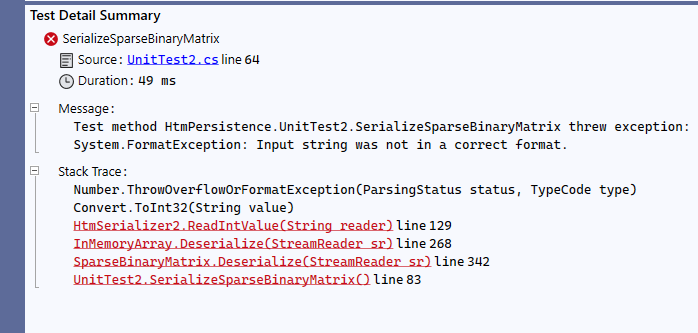
The explorer shows the success of the carried Tests of the various implemented classes; InMemoryDistributedDictionay and SparseObjectMatrix. In the Duration shows their execution times and helps to show their various performances. As stated about the implemented classes, the UnitTest of SparseBinaryMatrix threw an Exception Error.

***Encountered Errors***



*Fig 1.6 Error Encountered*

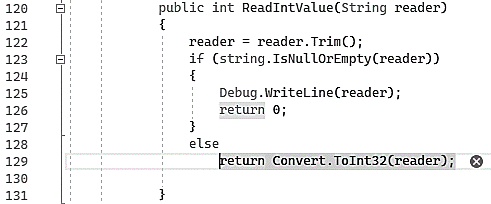
The UnitTest encountered errors in the process of serializing one of the classes due to the format of input string from one of the called Serializers in the HTM NeoCortexApi. See Test Detail summary below.



*Fig 1.7 Test Detail Summary*

The *SparseBinaryMatrix* could not serialize properly due to the string format used in the one of the major Serializers of the HTM modules in the NeoCortexApi which is the HtmSerializer2.cs. This Exception Error is thrown in line 129 in the HtmSerializer2.cs while serializing SparseBinaryMatrix. You can find the behavior here in this code Reference: Serialization Exception Error from HtmSerializer2.cs

*Fig 1.7 Exception Error (*[*HtmSerializer2.cs*](https://github.com/onyemakn/neocortexapi/blob/fda1274afab58109c1567b65ed5da21468c91d15/source/NeoCortexEntities/HtmSerializer2.cs#L129)*)*

**

# **Discussion**

The HTMSerializer2 class plays a major role in the implementation and serialization of existing Classes in the NeoCortexApi. During the Serialization, the lower classes SparseObjectMatrix,InMemoryDistributedDictionary, SparseBinaryMatrix calls the base HTM modules and classes InModuleTopology, htmConfig, InMemoryArray which holds the various states and instances to save to Stream. In the Deserialization a reverse process occurs. The various implemented Classes can be integrated to other parts of the NeoCortexApi. This creates a foundation for further implementations now made Possible for the specified classes in the Project, and can be further worked on for improvements, with more capabilities of saving larger instances and states. Also, the proper clarifications and distinctions of the dictionary keys and key values in the Process of the Serialization could give a greater meaning to the InMemoryDistributedDictionary. As a Memory based and functional Api, Serialization and Deserialization is very essential to boost the capabilities of the Api, making it an wholesome package. The NeoCortexApi must have the functional structure to recall it last states and continue working, whenever it initiates again in other to meet the capabilities of the Human Brain's NeoCortex.

# **References**

[1] Sequence learning and prediction in the neocortex | Numenta https://numenta.com/neuroscience-research/sequence-learning/   
  
[2] The Thousand Brains Theory of Intelligence | Numenta https://numenta.com/blog/2019/01/16/the-thousand-brains-theory-of-intelligence/   
  
[3] Serialization Filtering — Deserialization Vulnerability Protection | Albin Isaac  https://link.medium.com/EEXv86tYVob