***ML 22/23-5 Refactoring HtmSerializer***

Erdi Tras   
erdi.tras@stud.fra-uas.de

Khaled Kandil  
[Khaled.kandil@stud.fra-uas.de](mailto:Khaled.kandil@stud.fra-uas.de)

Maria Sanz Pina Liam Hosier  
maria.spso31@gmail.com

*Abstract*—A theoretical framework called hierarchical temporal memory (HTM) offers a way to simulate a number of the Neocortex's basic computing ideas. Hierarchical Temporal Memory (HTM) serialization refers to the process of converting HTM objects, such as HTM models or network configurations, into a format that can be stored or transferred between systems. HTM serialization typically involves converting the object into a binary or text-based format that can be easily transmitted or stored, while preserving the key properties of the object. The resulting serialized object can then be used to recreate the original HTM object, either on the same or a different system. Deserialization is opposite of the serialization. HTM serialization unit tests are automated tests that are designed to verify that the serialization and deserialization functionality of an HTM serializer works as expected. A software testing process known as unit testing involves testing each individual unit or component of a software system separately to make sure they function as intended. The purpose of this project is refactoring the code of the HTMSerializer File. This means making the code more legible and understandable.

Keywords—hierarchical temporal memory, Neocortex, Serialization, Deserialization, Unit Test, Refactoring

# Introductıon

A Hierarchical Temporal Memory (HTM) serializer is a software component that is designed to convert HTM objects into a serialized format. HTM is a machine learning algorithm developed by “Numenta” that is based on principles of neuroscience and is designed to recognize patterns in data streams, such as those found in sensor data or natural language.

A developing machine learning approach called hierarchical temporal memory (HTM) may make it possible to make predictions on spatiotemporal data. The neocortex-inspired algorithm does not yet have a complete mathematical foundation. In this work, the spatial pooler (SP), a crucial learning component in HTM, is brought together under a single, overarching framework. In order to determine the level of permanence updating, a maximum likelihood estimator for the basic learning mechanism is proposed. The study of the boosting processes reveals that they constitute a secondary learning mechanism. The SP is shown to perform remarkably well on categorical data in both spatial and categorical multi-class categorization. A comparison between HTM and well-known algorithms like competitive learning and attribute bagging is made. There are ways to use the SP for both dimensionality reduction and classification. Evidence from experiments shows that the SP may be utilized for feature learning when the appropriate parameterizations are applied. [1]

Several of the neocortex structures and functions are modeled by HTM at a high level. Its structure resembles that of the cortical minicolumns, where an HTM region is made up of numerous columns made up of various numbers of cells each. A level is formed by one or more regions. The whole network shown in Figure 1 is made up of levels that are stacked hierarchically in a tree-like structure. Synapses are used to create feedforward and adjacent connections in HTM, respectively. Proximal and distal synapses are used in these connections.

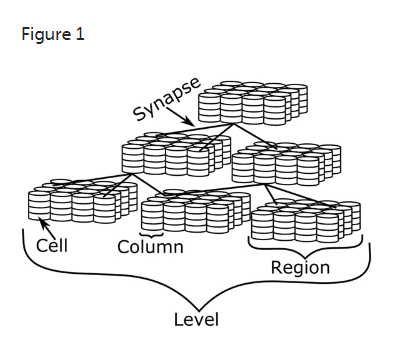


Figure 1: Depiction of HTM

The HTM cortical learning method was succeeded by the present version of HTML. The spatial pooler (SP) and the temporal memory algorithm are the two main algorithms in the current version of HTM (TM). An SDR is a binary vector that typically has a sparse number of active bits or a bit with the value "1," and the SP is in charge of receiving input in the form of an SDR and producing a new SDR. The SP can be thought of as a function that maps the input domain to a new feature domain in this way. Similar SDRs from the input domain should be represented by a single SDR in the feature domain. The algorithm uses a type of vector quantization that resembles self-organizing maps. It is a type of unsupervised competitive learning algorithm. Making predictions and learning sequences are tasks for the TM. Using this technique, connections are made between cells that have previously been active. The development of those linkages could lead to the learning of a sequence. The TM can then make predictions using the sequences it has learned about. [1]

The second generation of HTM learning algorithms, often referred to as cortical learning algorithms (CLA), was drastically different. It uses a sparse distributed representations data structure to describe brain activity and a more biologically accurate neuron model. The data structure's parts are binary, 1 or 0, and the number of 1 bits is minimal relative to the number of 0 bits (often also referred to as cell, in the context of HTM). Its HTM generation consists mostly of a sequence memory algorithm that learns to record and anticipate complex sequences and a spatial pooling technique that generates sparse distributed representations (SDR).

The cerebral cortex's layers and minicolumns are discussed and partially modeled in this latest generation. Each HTM layer is made up of a number of intricately interconnected minicolumns, which should not be confused with an HTM level of an HTM hierarchy. A fixed percentage of the minicolumns in an HTM layer's sparse distributed representation are active at any given time [clarification needed]. A minicolumn is a collection of cells with a same receptive field. A few of the cells in each minicolumn can recall many past states. There are three possible states for a cell: active, inactive, and predictive. [2]

Since HTM was first created as a neocortical abstraction, it lacks a formal mathematical formulation. Without a mathematical foundation, it is challenging to comprehend the main traits of the program and how it might be enhanced. Generally speaking, very little research has been done on the mathematics underlying HTM. [1]

**-Spatial Pooling**

Each minicolumn receptive field consists of a fixed number of inputs drawn at random from a much larger pool of node inputs. Certain minicolumns will be more or less related with the active input values depending on the (particular) input pattern. The most active minicolumns are chosen by spatial pooling, and other minicolumns close to the active ones are inactivated (inhibited). A stable set of minicolumns is typically activated by similar input patterns. Each layer's memory consumption can be altered to learn more intricate spatial patterns or less sophisticated ones.[2]

-**Active, inactive and predictive cells**

As was already established, a cell (or neuron) of a minicolumn may be in an active, inactive, or predictive state at any given time. Cells are inactive at first. They will be the sole cells to become active in the current time step if one or more active minicolumn cells are in the predicted state. The cells in the active minicolumn are made active if none of them are in the predicted state (which occurs during the first time step or when the activation of this minicolumn was unexpected). As a cell becomes active, it gradually establishes connections with cells in the area that have typically been active for a number of earlier time steps. In this way, a cell can recognize a recognized sequence by determining whether the cells it is attached to are active. This cell enters the predictive state in expectation of one of the few future inputs of the sequence if a significant number of related cells are active. A layer's output contains minicolumns that are both active and predictive.

Because minicolumns are active over extended periods of time, the parent layer observes more temporal stability.[2]

**-Comparing HTM and neocortex**

HTM is an effort to mimic the operation of a set of cortical areas in the neocortex that are connected hierarchically. The hippocampus is distantly similar to the highest HTM level, while a portion of the neocortex corresponds to one or more HTM levels. A single HTM node may stand in for a collection of cortical columns in a specific area. Although the HTM is largely a functional model, there have been various attempts to link its algorithms to the organization of neural connections in the neocortex's layers. Six horizontal layers arranged vertically compose the neocortex. It is

incorrect to equate the six layers of neocortical cells with the tiers of an HTM hierarchy. [2]

# Methods

**-Serialization and Deserialization**

Serialization is the process of converting an object or data structure into a format that can be easily transmitted or stored. In the case of HTM, serialization allows trained HTM models or network configurations to be saved, loaded, and shared between systems.

An HTM serializer typically consists of a set of functions or methods that are used to convert HTM objects into a standardized format, such as a binary or text-based format. The serialized HTM objects can then be transmitted, stored, or loaded into an HTM system for further processing. HTM serializers can be implemented in a variety of programming languages and are often used in conjunction with other HTM software components, such as HTM learning algorithms or anomaly detection systems.

The process of serialization involves changing an object's state into a format that can be stored or transferred. Serialization's counterpart, deserialization, transforms a stream into an object. These procedures work together to make it possible to store and send data. Deserialization is opposite of the Serialization. [3]

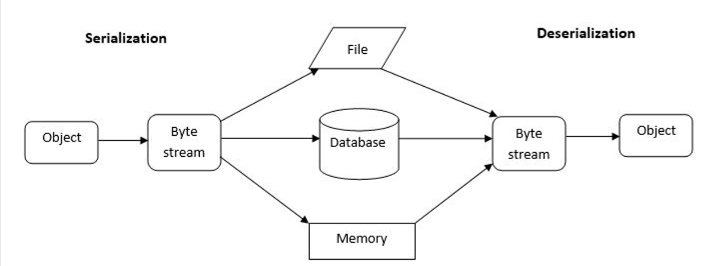


Figure 2: Serialization and Deserialization of an Object

In C#, a serializer is used to convert an object to a format that can be easily transported or stored, such as JSON or XML. The most commonly used serialization methods in C# are:

**a) Binary Serialization:** This method converts an object to a binary format, which can be easily transported or stored. It is useful for situations where data needs to be sent over a network or saved to a file.

Because binary serialization maintains type fidelity, the entire state of the object is preserved, and when you deserialize, an exact copy is produced. The status of an object can be preserved using this type of serialization between application calls. For instance, serializing an object to the Clipboard allows you to share it between many apps. An item can be serialized to a stream, a disk, memory, across the network, and other locations. To transfer things “by value” from one computer or application domain to another, remote access requires serialization. [3]

**b) XML and SOAP Serialization:** This method converts an object to an XML format, which can be easily read and understood by both humans and machines. It is useful for situations where data needs to be exchanged between systems or platforms.

Just public properties and fields are serialized using XML and SOAP, and type fidelity is not maintained. When you wish to supply or consume data without limiting the program that consumes the data, this is helpful. XML is a popular option for Web-based data sharing because it is an open standard. Being an open standard, SOAP is a desirable option. [3]

**c) JSON Serialization:** This method converts an object to a JSON format, which is a lightweight data-interchange format that is easy for humans to read and write and easy for machines to parse and generate.

JSON serialization serializes only public properties and does not preserve type fidelity. An appealing option for data sharing over the web is the open standard JSON. [3]

All above serialization method are supported by .NET framework, you can use any of them based on your requirement and compatibility.

Our task is refactoring the code of the HTMSerializer File. This means making the code more legible and understandable. In order to do that, we are going to separate the methods that use serialize and deserialize in two classes. The class 1 will correspond to those methods that are formatting the code while the second class will correspond to the actual serializing and deserializing methods. These methods are used in a wide variaty of projects. Our task is trying to encapsulate these methods so every project can use them (is like a library, you can take it and adapt it to your own work).

The outline of this project can mainly be described in four parts as following:

1) Analyzing the serializer and solving the errors of UnitTests.

2) Splitting the Formatting code methods from the Serialization code methods (SerializeValue/DeserializeValue from all other methods).

3) Implementing all formatting methods with create an interface IHtmSerializationFormatter and class HtmDefaultTextFormatter which interface contains all formatting methods. Then, copy all formatting methods from HtmSerializer into that class and doing refactoring.

4) Improving Unit Tests after refactoring.

-**Serialize() and Deserialize() Methods**

Two methods named Serialize() and Deserialize have been implemented inside the classes in under NeoCortexApi directory in HTM and they are the two main methods of our project. This Seralize function creates an instance of the HtmSerializer class and initiates functions called SerializeBegin and SerializeEnd.

The methods Serialize Begin and Serialize End are both methods that use sw.Write(). In the project it says we should name this methods SerializeValue, but they are not serializing any value itself, just saying when the start and end of the serialized object is, so its logical to call them SerializeObject instead Also we can make the code more compact by doing only one method:

public void SerializeObject(String typeName, StreamWriter sw, bool isBegin)

{

sw.WriteLine();

if (isBegin)

{

sw.Write($"{TypeDelimiter} BEGIN '{typeName}' {TypeDelimiter}");

}

else

{

sw.Write($"{TypeDelimiter} END '{typeName}' {TypeDelimiter}");

}

Figure 3: Calling SerializeBegin and SerializeEnd in

SerializeObject class

To make the class start and end in a text file clear to the user, serialize Begin and serializeEnd are used. The SerializeValue method is used to implement serialization for fundamental properties like int, double, bool, string, array, list, and dictionary. SerializeValue has no output. It writes the output in the stream in this case StreamWriter which is an argument in every SerializeValue methods. The method takes three parameters as input: the object to be serialized, the type of the object, and the StreamWriter to write the serialized object to. We can see in figure 4 the typical format of a SerializeValue method.

Public void SerializeValue(int val,StreamWriter sw)

Figure 4: Method SerializeValue (header)

On the other hand we have the deserialize methods called DeserializeValue() or other names. These methods are generic methods that deserializes a value of type T from a StreamReader. We need StreamReader and the value of the ParameterDelimiter for output.

public T DeserializeValuE<T>(streamReader sr)

Figure 5: Method DeserializeValue (header) from the line 1286 of HTMSerializer

The current serializer implementation contains the serialization logic code and serialization formatting code. Formatting code is in methods that use sw.Writer() or sw.Read() methods. Also, name all these methods SerializeValue and DeserializeValue. Formatting code methods are needed to perform the refactoring. Then, the interface and the class to be created containing these formatting methods from HtmSerializer. So, all formatting code methods and other serialization code methods are listed below.

|  |  |
| --- | --- |
| Formatting Codes | Other Serialization/  Deserialization Codes |
| SerializeBegin() | ReadBegin() |
| SerializeEnd() | ReadEnd() |
| SerializeValue(int) | ReadIntValue() |
| SerializeValue(double) | ReadDoubleValue() |
| SerializeValue(string) | ReadStringValue() |
| SerializeValue(long) | ReadLongValue() |
| SerializeValue(bool) | ReadBoolValue() |
| SerializeValue(object) | Serialize/Deserialize  KeyValuePair() |
| SerializeValue(array) | Serialize/Deserialize  Dictionary() |
| SerializeValue  (double[]) | Serialize/DeserializeObject() |
| SerializeValue(int[]) | Serialize/Deserialize  DistalDendrite() |
| SerializeValue(cell[]) | Serialize/Deserialize  HtmConfig() |
| SerializeValue  (Dictionary) | Serialize/Deserialize  IEnumerable() |

Table 1: Splitting formatting methods from other methods

**-Unit Tests**

HTM serialization unit tests are automated tests that are designed to verify that the serialization and deserialization functionality of an HTM serializer works as expected.

Unit testing is a software testing methodology in which individual units or components of a software system are tested in isolation to ensure that they work correctly. In the case of an HTM serializer, unit tests would typically involve testing the serializer's ability to convert HTM objects, such as HTM models or network configurations, into a serialized format and then back into the original object without loss of data or functionality.

HTM serialization unit tests might include the following types of tests:

(a) **Serialization tests:** These tests would verify that the serializer correctly converts an HTM object into a serialized format.

(b) **Deserialization tests:** These tests would verify that the deserializer correctly converts a serialized HTM object back into its original form.

(c) **Round-trip tests:** These tests would verify that the serializer and deserializer together can correctly serialize and deserialize an HTM object, with no loss of data or functionality.

(d) **Error handling tests:** These tests would verify that the serializer and deserializer correctly handle error conditions, such as invalid input data or unexpected data formats.

HTM serialization unit tests are an important part of the software development process, as they help to ensure that the serializer works correctly and is free of bugs or errors. By automating these tests, developers can catch problems early in the development process and reduce the likelihood of bugs being introduced into the codebase.

Sometimes some unit tests related to our project may give errors when we debug. We are also making some changes in the source code to eliminate these errors. Let’s see some examples from source code changeable to solve unit test errors. It is HtmSerializationTests which are unit tests related to our project.

\*Example of the solving Unit Test: First example unit test error is from our “SerializeDistalDendrite (1,2,2,1,100)” test. When we debugged this unit test, we encountered an error, but we solved the error by making changes to the source code.

Unit Test HtmSerializationTests.cs (line 590)

public void SerializeDistalDendrite(int flatIdx, long lastUsedIteration, int ordinal, double synapsePermConnected, int numInputs)

{

…

…

…

using (StreamReader sr = new StreamReader($"ser\_{nameof(SerializeDistalDendrite)}.txt"))

{

HtmSerializer ser = new HtmSerializer();

DistalDendrite distSegment1 = ser.DeserializeDistalDendrite(sr);

Assert.IsTrue(distSegment1.Equals(distSeg1));

}

}

Figure 6: Original source code of the “SerializeDistalDendrite” Unit Test

The error here is that the value of DistSegment1 is equal to 0. In order to the DistSegment1 variable not to be 0, we must first equalize the DistSeq1 value to the DistSegment1 value. So, need to add equation for equal DistSegment1 variable value to DistSeq1 variable value (DistSegment1=DistSeq1).

public void SerializeDistalDendrite(int flatIdx, long lastUsedIteration, int ordinal, double synapsePermConnected, int numInputs)

{

…

…

…

using (StreamReader sr = new StreamReader($"ser\_{nameof(SerializeDistalDendrite)}.txt"))

{

HtmSerializer ser = new HtmSerializer();

DistalDendrite distSegment1 = ser.DeserializeDistalDendrite(sr);

**DistSegment1 = DistSeg1 //Add this code to solve**

Assert.IsTrue(distSegment1.Equals(distSeg1));

}

}

Figure 7: Modified source code of the “SerializeDistalDendrite” Unit Test (add line 623 code)

\*Example of the solving Unit Test: Second example unit test error is from our “SerializeDictionarystringintA” test. When we debugged this unit test, we encountered an error, but we solved the error by making changes to the source code.

Unit Test HtmSerializationTests.cs (line 368)

public void SerializeDictionarystringintA()

{

HtmSerializer htm = new HtmSerializer();

Dictionary<String, int[]> keyValues = new Dictionary<String, int[]>

…

…

string[] str = data.Split(HtmSerializer.ParameterDelimiter);

for (int i = 0; i < str.Length; i++) {

switch (i) {

case 0:

keyValuePairs = htm.ReadDictSIarray(str[i]);

break;

default:

break;

}

}

}

}

}

Assert.IsTrue(keyValuePairs.SequenceEqual(keyValues));

}

Figure 8: Original source code of the “SerializeDictionarystringintA” Unit Test

To solve that error, we can get “Assert.IsTrue(keyValuePairs.SequenceEqual(keyValues));” code line into the switch case loop. Because we need to verify that the Assert.IsTrue statement at the end of the method is correctly comparing the key value pairs. Depending on the implementation of the HtmSerializer, the comparison may need to be modified to account for any differences in the serialized and deserialized data.

public void SerializeDictionarystringintA()

{

HtmSerializer htm = new HtmSerializer();

Dictionary<String, int[]> keyValues = new Dictionary<String, int[]>

…

…

string[] str = data.Split(HtmSerializer.ParameterDelimiter);

for (int i = 0; i < str.Length; i++) {

switch (i) {

case 0:

keyValuePairs = htm.ReadDictSIarray(str[i]);

break;

default:

break;

Assert.IsTrue(keyValuePairs.SequenceEqual(keyValues));

}

}

}

}

}

}

Figure 9: Modified source code of the “SerializeDictionarystringintA” Unit Test (add code line 413 from line 420)

**-Refactoring**

Refactoring a Hierarchical Temporal Memory (HTM) serializer would involve making improvements to the design, structure, or functionality of the serializer without changing its overall purpose. Refactoring is typically done to improve the code's maintainability, extensibility, and readability, and to reduce technical debt.

Some possible refactoring tasks for an HTM serializer might include:

\* Improving the performance of the serializer by optimizing its algorithms or data structures.

\* Simplifying the codebase by removing redundant or unnecessary code.

\* Refactoring the serializer to conform to a consistent coding style or design pattern.

\* Adding new functionality to the serializer, such as support for additional data formats or compression algorithms.

\* Improving the error handling or logging mechanisms of the serializer to make it easier to diagnose and fix problems.

Refactoring an HTM serializer can help to ensure that it remains maintainable and up to date as new requirements or technologies emerge. This can be particularly important in the field of machine learning, where the pace of development is often rapid, and the requirements can be complex.

# Results

# Discussion

##### References

1. J. Mnatzaganian, E. Fokoue, and D. Kudithipudi, “A Mathematical Formalization of Hierarchical Temporal Memory’s Spatial Pooler,”, 30January2017. <https://www.frontiersin.org/articles/10.3389/frobt.2016.00081/full>
2. Authors unpublished, Vikipedia, “Hierarchical temporal memory”, <https://en.wikipedia.org/wiki/Hierarchical_temporal_memory>
3. Gewarren, TimShererWithAquent, tdykstra, nschonni, mairaw, Mikejo5000, “Serialization in .NET”, 23 September 2022, <https://learn.microsoft.com/en-us/dotnet/standard/serialization/>

**This report template contains guidance text for composing and formatting technical reports. Please ensure that all template text is removed from your report prior to submission to the examination office. Failure to remove template text from your paper may result in your paper being degraded.**