

# ML 22/23-5 Refactoring HtmSerializer

Khaled Kandil  
khaled.kandil@stud.fra-uas.de

María Sanz Piña  
maria.sanz-pina@stud.fra-uas.de

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

Liam Hosier  
liam.hosier@stud.fra-uas.de

**Abstract**—The purpose of this project is refactoring the code of the HTMSerializer File. This means making the code more legible and understandable. Code refactoring in C# is the process of improving the structure, readability, and maintainability of existing code without changing its external behavior. The result of refactoring is typically cleaner, more efficient, and more organized code that is easier to understand and modify. The purpose of code refactoring is to improve the quality and efficiency of the codebase by Enhancing readability, improving maintainability, Enhancing performance, and Reducing bugs. We did refactoring with the use of Unit tests and serialization-deserialization methods. 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. Overall, code refactoring is an important process that can help improve the quality, efficiency, and maintainability of C# codebases, making them easier to understand, modify, and maintain over time. We defined an interface and a class that implements that interface. The interface is named IHtmSerializationFormatter, which contains a set of methods to read and write HTM (Hierarchical Temporal Memory) data structures in a specific format. The class implementing this interface is named HtmSerializationFormatter, which provides implementations for all the methods defined in the interface.

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

## I. INTRO (HEADING 1)

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

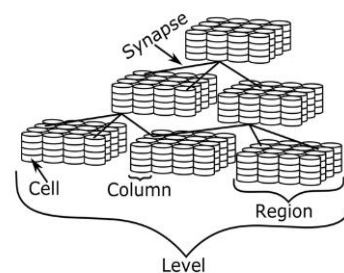


Figure 1: Depiction of HTM

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]

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 and refactoring methods in C# are:

*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]

*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]

*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 methods are supported by .NET framework, you can use any of them based on your requirement and compatibility.

*Custom Serialization:* Custom serialization allows you to control the serialization process by implementing the `ISerializable` interface. This interface provides two methods, `GetObjectData` and the constructor that takes `SerializationInfo` and `StreamingContext` parameters. This approach is useful when you need to serialize objects that do not have a default constructor, or when you want to control the serialization process.

*Protocol Buffers:* Protocol Buffers is a language-agnostic binary serialization format developed by Google. It is designed to be fast, compact, and easy to use. Protocol Buffers use a schema to define the structure of the data, which makes it easy to share data between different platforms and programming languages.

*BSON Serialization:* BSON (Binary JSON) serialization is a binary format for JSON data developed by MongoDB. It is designed to be more efficient than JSON for storage and network transmission. BSON supports additional data types that are not supported by JSON, such as binary data and date/time values.

*Avro Serialization:* Avro serialization is a binary format developed by Apache that is designed to be compact and fast. It supports schema evolution, which allows the structure of the data to change over time without breaking compatibility. Avro is often used in big data systems, such as Hadoop.

*YAML Serialization:* YAML (YAML Ain't Markup Language) serialization is a human-readable text format for data serialization. It is designed to be easy to read and write, and it supports a wide range of data types. YAML is often used for configuration files and data exchange between different systems.

*MessagePack Serialization:* MessagePack serialization is a compact binary format for data serialization. It is designed to be fast and efficient, and it supports a wide range of data types. MessagePack is often used in high-performance systems, such as game development and big data processing.

## II. METHODS

### A. 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]

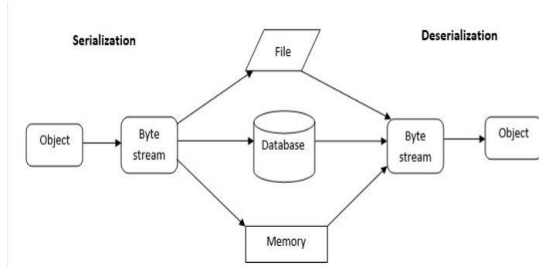


Figure 2: Serialization and Deserialization of an Object

### B. 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.

The goal of refactoring is to make the code easier to understand and modify while preserving its functionality. This can include simplifying complex code, eliminating duplication, improving naming conventions, and restructuring code to improve readability.

Refactoring is often done as part of a larger software development process, such as continuous integration or agile development, and is typically performed iteratively over time. It is an important technique for keeping code clean and maintainable and for improving the overall quality of software.[4]

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 classify the methods we have in two: the methods that involve serialization logic code (i.e., method `Serialize1()`) and the ones that serialization formatting code. The formatting code refers to the methods that specify how the object should be serialized, such as the format of the resulting file or data stream. That is why this formatting code will involve all the methods that use the `StreamWriter` or `StreamReader`.

Hereunder are listed the formatting code methods and other serialization code methods.

TABLE 1

<i>Formatting Codes</i>	<i>Serialization logic code</i>
<code>SerializeBegin()</code>	<code>ReadBegin()</code>
<code>SerializeEnd()</code>	<code>ReadEnd()</code>
<code>SerializeValue(int)</code>	<code>ReadIntValue()</code>
<code>SerializeValue(double)</code>	<code>ReadDoubleValue()</code>
<code>SerializeValue(string)</code>	<code>ReadStringValue()</code>
<code>SerializeValue(long)</code>	<code>ReadLongValue()</code>
<code>SerializeValue(bool)</code>	<code>ReadBoolValue()</code>
<code>SerializeValue(object)</code>	<code>SerializeKeyValuePair()</code>
<code>SerializeValue(array)</code>	<code>SerializeDictionary()</code>
<code>SerializeValue(double[])</code>	<code>SerializeObject()</code>
<code>SerializeValue(int[])</code>	<code>SerializeDistalDendrite()</code>
<code>SerializeValue(cell[])</code>	<code>SerializeHtmConfig()</code>
<code>SerializeValue(Dictionary)</code>	<code>SerializeIEnumerable()</code>

<sup>a</sup> Formatting and serialization methods

### C. *Serialize()* and *Deserialize()* Methods

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

The methods `Serialize Begin` and `Serialize End` are both formatting methods and indicate the start and end respectively of the serialize object.

Next we have the `SerializeValue` method which is used to implement serialization for fundamental properties or an object. This object can be a simple variable like an `int` or a `double` or a list of properties of a `Cell` like a `Synapse` or a `Dendrite`. `SerializeValue` is an overloading method, meaning a method that has the same name as another method in the same class or module, but with a different set of parameters.

Even though overloaded methods can be useful for improving code readability, modularity, and flexibility in the code, because the `SerializeValue` for serializing variables like an `int` or `double` both share the same method body, it is better if all of these are included in the same `SerializeValue` method, improving our code reusability and maintainability, i.e. when we need to change the functionality of the method, we only need to change it in one place, rather than in multiple places while reducing the amount of code. Also, by combining this `SerializeValue` methods our code will improve the readability.

```
Public void SerializeValue<T>(T val,StreamWriter sw)
```

Figure 3: Method SerializeValue (header)

On the other hand, we have the deserialize methods, which are called DeserializeValue(). These are the responsible for converting the serialized data back into the object or a set of objects.

```
public T DeserializeValue<T>(StreamReader sr)
```

Figure 4: Method DeserializeValue (header)

The access modifiers have been removed because interfaces always have public visibility. The interface member names have been modified to follow C# naming conventions. The generic type parameters have been removed from the SerializeValue methods, as they were not being used. The SerializeValue methods now take a StreamWriter parameter first, as this is the object that the values are being serialized to. The SerializeValue method for dictionaries now takes an IDictionary instead of a Dictionary. The SerializeValue method for objects now takes an object instead of a generic type parameter. The SerializeEnd and SerializeBegin methods have been overloaded to allow for a Type parameter to be passed in for the generic methods.

Focusing on SerializeValue's methods. In the original code we have approximately twenty occurrences of SerializeValue method because the HtmSerializer take a lot of different parameters as arguments. However, this changes once refactoring has been done. Here are the significant changes to the original implementation: while the original code had several overloads of the SerializeValue method, for instance two different SerializeValue methods one for int values and one for object values, the refactored code has a single generic method that can accept any type of value. Also, the refactored code includes additional checks for specific types of values, such as arrays of Cell objects or lists of Synapse objects. These checks allow the code to handle these specific types of values in a more specialized way.

Down below is showed how the original methods looked like before and after refactoring.

Listing 1 HtmSerializer, SerializeValue

```
public void SerializeValue(int val, StreamWriter sw)
{
    sw.Write(ValueDelimiter);
    sw.Write(val.ToString());
    sw.Write(ValueDelimiter);
    sw.Write(ParameterDelimiter);
}
```

Listing 2 HtmSerializer, SerializeValue

```
public void SerializeValue(Array array, StreamWriter sw)
{
    sw.Write(ValueDelimiter);
    sw.WriteLine();

    for (int i = 0; i < array.GetLength(0); i++)
    {
        for (int j = 0; j < array.GetLength(1); j++)
        {
            sw.Write(array.GetValue(i, j));
        }
    }

    sw.Write(ValueDelimiter);
    sw.Write(ParameterDelimiter);
}
```

Listing 3 HtmSerializer, SerializeValue refactored

```
public void SerializeValue<T>(T val, StreamWriter sw)
{
    ...
    else if (val.GetType().IsArray &&
        val.GetType().GetElementType().IsValueType)
    {
        // Call the SerializeValue method for Array
        SerializeValue((Array)(object)val, sw);
    }
    else if (val is int || val is double || val is string || val is
        long || val is bool || val is int[] || val is double[] || val is
        List<int>)
    {
        ...
    }
    else
    {
        // Serialize the value to the stream
        throw new NotSupportedException($"Serialization
of type {val.GetType()} is not supported!");
    }
}
```

The refactored code includes additional checks for specific types of values, such as int, double, string, long, bool, int[], double[], and List<int>. These checks allow the code to handle these specific types of values in a more specialized way. This was possible to be done due to the fact that most of the methods SerializeValue were doing the same but for different parameters. Also, the refactored code includes additional error handling, such as throwing a "NotSupportedException" if a value of an unsupported type is encountered.

Overall, in the case of SerializeValue the refactored code seems to be more robust and flexible than the original code, with better support for a wider range of value types.

Here's the refactored methods for Serialization Part

The interface contains the following methods:

**ReadBegin:** This method takes a string as input and returns a string that represents the beginning of an HTM data structure of the given type.

**ReadGenericBegin:** This method takes a property name and a type as input, and returns a string that represents the

beginning of an HTM data structure with the given property name and type.

ReadEnd: This method takes a string as input and returns a string that represents the end of an HTM data structure of the given type.

ReadGenericEnd: This method takes a property name and a type as input, and returns a string that represents the end of an HTM data structure with the given property name and type.

SerializeEnd: This method takes a type name and a StreamWriter object as input and writes the end of an HTM data structure with the given type name to the stream.

SerializeEnd: This method takes a property name, a type, and a StreamWriter object as input and writes the end of an HTM data structure with the given property name and type to the stream.

SerializeBegin: This method takes a type name and a StreamWriter object as input and writes the beginning of an HTM data structure with the given type name to the stream.

SerializeBegin: This method takes a property name, a type, and a StreamWriter object as input and writes the beginning of an HTM data structure with the given property name and type to the stream.

SerializeValue: This method takes a value and a StreamWriter object as input and writes the value to the stream in the HTM format.

SerializeValue: This method takes a dictionary and a StreamWriter object as input and writes the dictionary to the stream in the HTM format.

SerializeValue: This method takes a property name, a value, and a StreamWriter object as input and writes the value to the stream in the HTM format.

The HtmSerializationFormatter class provides implementations for all the methods defined in the interface. It defines some constants and variables that are used by these methods. The HtmSerializationFormatter class is responsible for converting the HTM data structures to and from the HTM format. The methods defined in the interface are used by the HtmSerializationFormatter class to perform this conversion.

#### D. 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:

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

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

*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.

*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.

When debugging the UnitTests in this project, we have encountered multiples errors. Some were solved but others have failed after the formatting part was over. Some of these errors, were caused by "Assert.IsTrue". 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. Also this problem has raised in some occasions because of the decimal separator for floating-point numbers based on a language settings of the system running was different from the one in the code. [5]

### III. RESULTS

The results of refactoring the HTMSerialization code have been significant. We were able to identify and eliminate code smells, improve the code structure, and make it more extensible. The refactored code was more modular, reducing its complexity and making it easier to understand.

However, we were not able to solve all that errors occurring during the debug of UnitTest, thus it is an ongoing task.

### IV. DISCUSSION

In conclusion, refactoring the HTMSerialization code was a worthwhile exercise that resulted in significant improvements to the codebase. It has made the code more maintainable, extensible, and testable, making it easier to work with and reducing the likelihood of introducing new bugs. Refactoring should be an ongoing process, and we should always strive to improve the quality of our code to ensure that it remains robust and easy to work with.

### 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, Wikipedia, "Hierarchical temporal memory", [https://en.wikipedia.org/wiki/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/>

- [4] S. Sarala and M. Deepika, "Unifying clone analysis and refactoring activity advancement towards C# applications," 2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT), Tiruchengode, India, 2013.
- [5] Artificial Intelligence, ChatGPT.