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

Implement Serialization of HTM Classifier

Amith Nair
amith.nair@stud.fra-uas.de

Aneeta Antony
aneeta.antony@stud.fra-uas.de

Rohit Suresh
rohit.suresh@stud.fra-uas

***Abstract*— Serialization is the act of transforming an object or data structure into a format that is simple to store, send, or exchange with other systems or applications. A text-based format, a binary stream, or any other structured file format can be used for the serialized representation. This paper focuses on the implementation of serialization in Hierarchical Temporal Memory (HTM) classifier and demonstrates how the parameters of HTM classifier can be serialized. A machine learning method called the HTM classifier learns and anticipates patterns in time-series data using the neocortex's structural layout. In this project, serialization functionality of HTM Classifier in C#/.NET Core is implemented for Neocortex API. Serialization in the HTM classifier enables the classifier to preserve its state and structure as a file, enabling it to maintain its state over different runs or transfer its state to a different machine. It is helpful for picking up where training stopped off and incorporating the classifier into different programs or frameworks. We have carried out several unit tests by comparing the serialization and deserialization results in order to validate our methods.**

Keywords—: Classifier, Hierarchical Temporal Memory, Sparse Distributed Representation, neocortex.

I. INTRODUCTION

A new paradigm in the study of artificial intelligence has emerged as a result of the idea of Hierarchical Temporal Memory (HTM) and Cortical Learning Algorithms (CLA) recently developing. A biomimetic model called HTM-CLA is based on Jeff Hawkins' memory-prediction hypothesis of brain activity. It is a technique for identifying and extrapolating the root causes of observable input patterns and sequences in order to construct an ever-more sophisticated model of the real world. The structure and operation of the human neocortex are the foundation of HTM-CLA. It lays the foundation for creating robots that do numerous cognitive activities at levels close to or higher than those of humans. It is utilized by the NuPIC, a Numenta-led open-source project. Fundamentally, HTM-CLA is a memory-based prediction system. The networks hold a substantial number of patterns and sequences and are trained on time-varying input. It is intrinsically time based and organized hierarchically. HTM-CLA acquires and saves the hierarchy of regions' structure and sequences of data in a special representation known as Sparse Distributed Representation (SDR). [1]

This paper focuses on the serialization functionality in an HTM classifier with the goal of outlining the procedure in great detail and emphasizing the advantages of this method. The purpose of serialization is to enable the trained model to be stored to a file or database for subsequent use in an HTM (Hierarchical Temporal Memory) classifier. A stream of bytes that may be

saved on storage or sent over a network is created by serializing an object that is now in memory.

Deserialization is the process of building an object from stored bytes in the opposite direction. When an HTM classifier has been trained using a set of data, its state may be saved by serializing the classifier. As a result, we can reuse the model instead of always having to train it from scratch. This is especially helpful in situations when the training procedure is computationally expensive or the training data is huge. Moreover, serialization makes it possible to share the model across many contexts or apps. For usage in production settings, a trained HTM classifier, for instance, might be serialized and disseminated to other systems. Overall, trained models become more portable and effective because to serialization, which offers a mechanism to preserve and reuse them. [2]

A) HTM CLASSIFIER

The biological processes of the brain and its learning mechanism are the foundation of HTM. The findings are highly pertinent and demonstrate a small percentage of incorrect predictions over time. The goal of the theory and machine learning technology known as the Hierarchical Temporal Memory Cortical Learning Algorithm (HTM CLA) is to model the cortical algorithm of the neocortex. [3]

To represent patterns in the incoming data, HTM systems employ sparse distributed representations (SDRs), which serve as its core data structure. SDRs are binary vectors with a disproportionately high proportion of zeros and a disproportionately low proportion of ones, with the ones being scattered randomly across the vector. [3]

The classifier, one of the essential parts of an HTM system, is in charge of identifying and categorizing patterns in the input data. The classifier learns and recognizes patterns over time by combining spatial pooling with temporal memory. SDRs are generated from the input data using spatial pooling, and SDR sequences are stored and recognized using temporal memory. [4]

SDRs are particularly well-suited for usage in the HTM classifier due to their mathematical characteristics. For instance, the classifier can effectively represent a large number of patterns using a relatively small number of bits because to the SDRs' sparsity. The distributed representation of patterns also enables the classifier to identify patterns in the presence of missing or truncated bits in the SDR. [4]

Also, the classifier may use similarity-based categorization because to the mathematical characteristics of SDRs. The classifier can assess how similar two patterns are by comparing the overlap between their SDRs. As a result, even if new patterns are not precise matches to those it has already encountered, the classifier may still detect them. [4]

II. METHODS

A) Serilization

An object or data structure is serialized when it is put into a format that makes it simple to store or send across a network. It is frequently employed in programming to allow data flow across various applications and systems. Many methods and formats, including JSON, XML, Protocol Buffers, and others, can be used to perform serialization. [5]

```
public void Serialize (object obj, string name,
 StreamWriter sw)
{.
.
HtmSerializer ser = new HtmSerializer();
ser. SerializeValue(maxRecordedElements, sw);
ser.SerializeDictionaryValue(m_allinputs, sw);
.
.
 }
```

An object to be serialized (obj), is the HTM Classifier object, and a StreamWriter object (which enables the functionality of writing the values to a

stream, are the 2 main input arguments for the new method named Serialize () in the above code snippet. The technique is a component of a bigger codebase that offers serialization and deserialization capabilities. [6]

The serialization of the object supplied as an input parameter and writing of the serialized data to a stream are the two functions of the Serialize method. The data is serialized using a serialization library in the code above, which is represented by the ser object. The code specifically serializes the variables m_allinputs and maxRecordedElements. [6]

The first parameter, maxRecordedElements, is a straightforward number, is of datatype integer. To serialize this value and convert it to the stream, the already existing SerializeValue() method in the HtmSerialize library is reused. M_allInputs, the second value, is a KeyValuePair type dictionary object. This dictionary object is serialized and written to the stream by using the SerializeDictionaryValue function. [6]

We here have used binarization or writing to text files following the same approach used in other Spatial pooler or encoder method of the neocortex API or other serialization types available like JSON or XML. Depending on the needs of the application, such as data size, performance, and compatibility with various programming languages, a serialization format should be chosen. Many of serialization include decreased network traffic, enabled application and system compatibility, and increased scalability of distributed systems. [7]

B) Deserialization

The process of transforming a stream of bytes back into an object is called deserialization. Deserialization is frequently used in the setting of artificial intelligence to recover previously serialized models that have been stored to a file or database. Without having to start from scratch, the model may be deserialized and then loaded back into memory to be utilized for prediction or other activities.

```
public HTM Classifier<TIN, TOUT> Deserialize
(StreamReader sr)
{
.
.
if (data.Contains(HtmSerializer.KeyValueDelimiter))
{

var kvp = ser.ReadDictSIarrayList<TIN>(cls.m_allinputs,
data);
cls.m_allinputs = kvp;

}
.
.
if (int.TryParse(str[0], out int maxRecordedElements))
cls.maxRecordedElements = maxRecordedElements;
}
```

The code snippet describes the deserialization functionality newly added to HTM Classifier class. The StreamReader input argument is utilized by the Deserialize method to read the serialized data from a file. [8]

The original object is recreated from the serialized data via the deserialization procedure. The HTM Classifier object is being rebuilt in this code.

Firstly, from the StreamRead the stored SDR dictionary values are identified, and the dictionary is deserialized and assigned to the m_AllInputs property of the HTM Classifier object by using the ReadDictSIarrayList function. [8]

The m_AllInputs field appears to be a dictionary that converts a list of Sparse Distributed Representation (SDR) arrays to input values of type TIN. The HTM classifier uses SDRs, a mathematical idea, to express patterns in a dispersed and sparse way. With the use of this representation, the classifier can identify patterns even when they don't exactly match previously observed patterns. [8]

An integer value from the serialized data is attempted to be parsed in the second block of code. If successful, the value is set for the HTM Classifier object's maxRecordedElements property. This characteristic is used to denote the maximum allowed values of SDRs the classifier is capable of storing. [8]

The code, taken collectively, defines a deserialization technique for an HTM Classifier object. More details beyond the supplied code snippet would be necessary to comprehend the nuances of how the object is being used and how the serialization and deserialization functions integrate into the wider system.

C) Equals Method

In order to check the equality of two instances of the Classifier we need to override the object Equals() method.

The Object.Equals() in C# determines whether the specified object is equal to the current object and returns Boolean true if the specified object is equal to the current object; otherwise, false. The override added in here checks the two parameters in the HTM Classifier is checked individually. Various conditional checks are added in order to verify the Equality Criteria.

Firstly, the specified object is check for null or empty. By this any null specified object instance wouldn't be equal with current object

Secondly, the type of the objects in comparison is checked. If both the type doesn't match, then equality check fails as expected.

Next, the first parameter integer param, maxRecordedElements of the two compared instances is checked. For every equal object this should be same, or else unequal.

Finally, the second parameter m_AllInputs is taken for equality check. This being a Dictionary complex object ,of Key value Pair type , careful checks are required for Equality.

At first again null check is done for this parameter of both the specified object and the second compared object. If any of them alone is null this Equality check will return Boolean false.

Next in loops we evaluate the keys and values equality of both the Classifier instances which are passed for comparison. If and only if all keys and values are matching, then the Equals methods completes its check and return Boolean true. Else the object comparison fails.

Hence Equals method override in the HTM Classifier conditionally checked with enough scrutiny.

D) GetHashCode Method

A hash code is a numeric value that is used to insert and identify an object in a hash-based collection such as the Dictionary <TKey,TValue> class, the Hashtable class, or a type derived from the DictionaryBase class. The GetHashCode method provides this hash code for algorithms that need quick checks of object equality. Here as part of implementation in code, overriding Equals requires that you also override the GetHashCode method, otherwise, hash tables might not work correctly.

The Prime number hashing is used. The cumulative hash value of each parameter in the HTM Classifier need to be calculated and returned when GetHashCode is checked.

Firstly, a prime odd prime number is chosen to calculate hash. We choose 31. The first parameter, maxRecordedElements being an int value we could directly get the hash. Now we need to aggregate the hash value of the second m_ AllInputs too.

Next, the complex dictionary Key Value parameter needs to be taken care separately as it contains a List<int[]>. For each value in the Dictionary collection, in consecutive steps we carefully consider the hash value of Keys along

with the int array values in the List to get its hash code. The implementation can be found in the code.

During serialization, an instance of Classifier converted a text file using the serialization method. Then, the same file is used for deserialization, this creates the HTM Classifier instance with the previous saved values. We can verify the values are same in program. To ensure that the above functionality is successful, the original object and the deserialized object are compared for equality using the customized equals override added in the HTM Classifier. This ensures that and no data was lost or corrupted during the process. [9]

III. RESULTS

Successfully implemented the Serialization function for HTM Classifier. The Serialize() method newly added is called and along a StreamWriter, which helps to save data stream to a text file and is created in a required format implemented in code. This file is saved accordingly in the \bin folder when we test it. Consequently, this same file is used to regenerate the HTM Classifier instance with the Deserialize() implementation, and we found that the exact same parameters were set and the new instance. This is tested to be equal with actual Classifier object. This enables saving the state of the object whenever required.

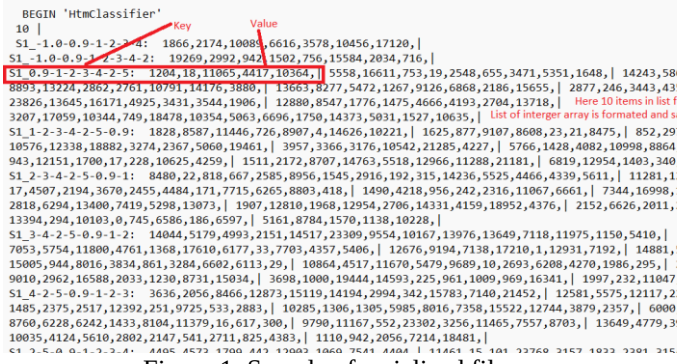

Figure 1: Sample of serialized file.

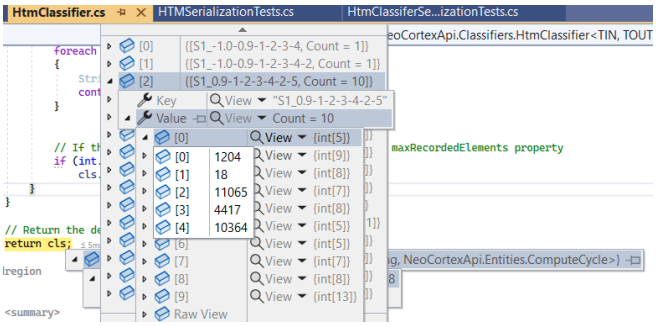

Figure 2: Classifier instance obtained after Deserialization.

Above you can find snippet form the saved serialized file in Figure 1. The Figure 2 shows the Deserialized and recreated Classifier parameters form the same file.

This is verified also with the help of unit testing briefly explained below. Optimum number of unit test cases are added to check all the scenarios.

a) *Test Category :- "Test-Serialization".*

In this test category, we have few tests to demonstrate that the serialize() is working as expected.

Test namely, *TestSerializationHtmClassifier* is testing with the help of serialization with help of Equals method. The serialized instance is written to a file using a StreamWriter, and the serialized data is retrieved from the same file and deserialized using a StreamReader. The two instances are then compared to make sure they are equal using the Equals function. [10]

Test namely, *TestSerializeDeserializeHtmClassifierFailure* checks failure of obtaining same object after deserialization by using a different mock file other than the one which was serialized. Therefore, the newly generated output params will differ for the original expected object. Therefore, this approach examines the HTM classifier's resilience to unanticipated mistakes. [10]

Test viz, *TestSerializeHtmClassifierByFileComparison* use the serialized files of two classifier objects to verify correctness. Similarly another test *TestSerializeHtmClassifierByUsingStreamCompar*

ison demonstrates the serialize() method credibility by stream comparison.

b) *Test category :- "Equals-Override-UnitTests"*

Few test methods both positive and negative tests are used to check each and every condition added with in the Equals() override we created. IT starts with checking object null check, followed by object type check. Then each parameter inside the classifier class is checked if equal or not and accordingly return true or false Boolean value . So, this will help make sure the correct behavior of Equals comparisons wherever the Classifier class is asserted for equality. [10]

c) *Test method :- "GetHashCode-Testing"*

The objective is to test the GetHashCode() override.The tests verifies if proper hash is returned when method called. Also 2 testcases in this category, checks hash code generated for two equal HTM Classifier instance to be equal or hash code generated for two different HTM Classifier instance will be different

Test Category	**Test Name**	**Result / Assertion.Equal**
GetHashCode-Testing	1) HashcodeSameForEqualObjectsTest 2) HashcodeDifferentWhenUnequalObjectsTest 3) HashCodeValueTestForAnHtmClassifierObject	1. Same hash 2. Different 3. Hash value properly returned
Test-Serialization	1) TestSerializationHtmClassifier 2) TestSerializeDeserializeHtmClassifierFailure 3) TestSerializeHtmClassifierByFileComparison 4) TestSerializeHtmClassifierByUsingStreamComparison	Serialized and saved to .txt file ,Serialized file read and Deserialized.
"SerializeDictionaryValue-SequenceCheck"	1) SerializeDictionaryStringListIntArray	Expected output format of Dictinary parameter obtained in test.
Equals-Override-UnitTests	1) TestHtmClassifierEqualsReturnFalseWhenTwoDifferentInstances 2) TestHtmClassifierEqualsReturnFalseWhenTwoDifferentObjectTypePassed 3) TestHtmClassifierEqualsReturnFalseWhenComparedObjectIsNull 4) TestHtmClassifierEqualsReturnFalseWhenParametersAreDifferent 5) TestHtmClassifierEqualsReturnFalseWhenParametersMissingValuesInside 6) TestHtmClassifierEqualsReturnFalseWhenKeyValueParametersDifferent 7) TestEqualsReturnFalseWhenInstancesHasNullParameters	1) Unequal instance, False 2) Different object type, False 3) Null, False 4) Unequal params, Flase 5) False 6) False 7) Not Equal objects, False

Table 1: Unit Test Outputs

d) *Unit test coverage*

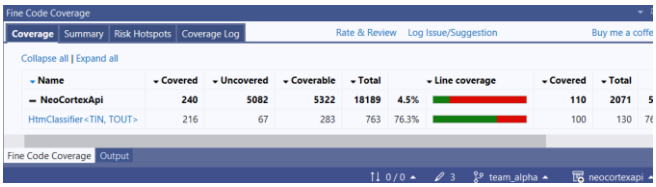

Figure 3: Test coverage with Fine code Coverage (FCC)

As a result, 76 percentage of unit test coverage was achieved which indicates that a significant part of the code has been fully tested according to the figure 3. All the newly added serialization and deserialization methods and lines are having full unit test coverage. This, in turn, can provide a higher level of confidence in the reliability of the HTM classifier.

IV. Discussion

Serialization was successfully implemented in the HTM classifier in this work, and various unit tests were used to show how effective it was. The HTM classifier's ability to be serialized and deserialized offers a number of advantages for its practical application, including the capacity to save

and restore classifier instances as well as the distribution of classifiers across many computers or processes.

As extension or next step to improve the efficiency of the serialization, either by using different file format or by improving the time taken to serialize huge data.

Overall, the HTM classifier's effective serialization implementation offers effective classifier instance distribution and storage, which can enhance scalability and performance in practical applications. Also, the HTM classifier can manage unexpected input thanks to the flexible deserialization procedure, which can increase its dependability and robustness.

V. References

[1] J. K. C. G. &. Z. F. Balasubramaniam, "Enhancement of classifiers in HTM-CLA using similarity evaluation methods.," *Procedia Computer Science*, pp. 1516-1523, 2015.

[2] X. W. W. a. W. L. Chen, "An overview of Hierarchical Temporal Memory: A new neocortex algorithm.," *Proceedings of International Conference on Modelling, Identification and Control*, 2012.

[3] [Online]. Available: https://www.numenta.com/assets/pdf/whitepapers/hierarchical-temporal-memory-cortical-learning-algorithm-0.2.1-en.pdf.

[4] [Online]. Available: https://www.numenta.com/assets/pdf/biological-and-machine-intelligence/BaMI-SDR.pdf.

[5] [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/serialization/.

[6] [Online]. Available: https://github.com/antonyaneeta/neocortexapi/blob/cec4993b41577690613e4b24bf1510471527e7f4/source/NeoCortexApi/Classifiers/HtmClassifier.cs#L501.

[7] [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/serialization/binary-serialization.

[8] [Online]. Available: https://github.com/antonyaneeta/neocortexapi/blob/cec4993b41577690613e4b24bf1510471527e7f4/source/NeoCortexApi/Classifiers/HtmClassifier.cs#L529.

[9] [Online]. Available: https://github.com/antonyaneeta/neocortexapi/blob/cec4993b41577690613e4b24bf1510471527e7f4/source/NeoCortexApi/Classifiers/HtmClassifier.cs#L586.

[10] [Online]. Available: https://github.com/antonyaneeta/neocortexapi/blob/cec4993b41577690613e4b24bf1510471527e7f4/source/UnitTestsProject/Classifiers/HtmClassiferSerializationTests.cs.