*Implement Serialisation of HTMClassifier*

Amith Nair  
amith.nair@stud.fra-uas.de Aneeta Antony  
aneeta.antony@stud.fra-uas.de

Rohit Suresh  
rohit.suresh@stud.fra-uas.de

*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, Neocortex API focuses implementation of serialization Hierarchical Temporal Memory Classifier in C#/.NET Core. 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.

# 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]

# Methods

# 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. [4]

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.

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. [5]

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. [5]

## 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.

public void Serialize(object obj, string name, StreamWriter sw)

{

//Serialization code below.

.

.

ser.SerializeValue(maxRecordedElements, sw);

ser.SerializeDictionaryValue(m\_allinputs, sw);

.

.

}

An object to be serialized (obj), a name (name), and a StreamWriter object (sw) that writes a string of characters to a stream are the three input arguments for the method named Serialize in the supplied code snippet. The technique seems to be 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, maybe an integer or similar basic data type. a ser. To serialize this value and post it to the stream, the SerializeValue method is used. M\_allinputs, the second value, is a dictionary object. a ser. This dictionary object is serialized and written to the stream by using the SerializeDictionaryValue function. [6]

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 advantages of serialization include decreased network traffic, enabled application and system compatibility, and increased scalability of distributed systems. It does have some drawbacks, though, such greater complexity and certain security threats. [6]

## B) Deserilization

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 HtmClassifier<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 is a section of an HTMClassifier object's deserialization function. The StreamReader input argument is utilized by the Deserialize method to read the serialized data from a stream. [6]

The original object is recreated from the serialized data via the deserialization procedure. The HTMClassifier object is being rebuilt in this code. [6]

The first block of code determines if the serialized data has a key-value delimiter, which would indicate that it is a dictionary object. If so, the dictionary is deserialized and assigned to the m\_ AllInputs property of the HTMClassifier object by using the ReadDictSIarrayList function. [6]

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. [6]

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 HTMClassifier object's maxRecordedElements property. This characteristic seems to reflect the maximum amount of items the classifier is capable of storing. [6]

The code, taken collectively, defines a deserialization technique for an HTMClassifier object that may create the object from serialized data. 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. [6]

# Results

In this section, we evaluated the HTMClassifier class through several unit tests.

Firstly, we evaluated the HTMClassifier class's serialization and deserialization abilities. In order to perform the test, a class instance had to be serialized to a file, deserialized to produce a new instance, and then the two instances had to be compared using the Equals function to make sure they were equivalent. To confirm the correctness of the serialized files, we also compared them. [7]

Using a different method, we evaluated the classifier's capacity for serialization and deserialization. In order to verify consistency, we serialized the classifier to a memory stream in this test, read the stream to retrieve the serialized string, and then compared the obtained string to the string obtained using the original serialization method. [7]

The HTMClassifier class's tolerance for void spaces during deserialization was then evaluated. When serializing a class instance, we added the character "|" for an empty space to test this. Upon deserialization, we checked to make sure the object wasn't null. [7]

Overall, the unit tests showed that the HTMClassifier class can be serialized using a variety of ways, can accept empty spaces during deserialization, and has strong serialization and deserialization capabilities. These results provide additional support for the efficiency and dependability of the HTMClassifier class. [7]

# Discussion

By the use of multiple unit tests, we were able to effectively integrate serialization in the HTM classifier in this study. For the HTM classifier's practical use, being able to serialize and deserialize it offers a number of advantages, including the capacity to spread classifier instances across many computers or processes and the ability to save and restore classifier instances.

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.

The HTM classifier's capacity to serialize and deserialize using the standard serialization method was shown in the first unit test we ran. By contrasting the serialized instance with the original instance and the predicted output, we were able to confirm the integrity of the serialized instance. This test demonstrated the HTM classifier's dependability on the default serialization method.

The classifier was serialized to a memory stream and read to produce the serialized text in the second unit test, which served as a demonstration of an alternate serialization approach. To check for consistency, we compared the serialized string to the original serialized file and discovered that they were both the same. This test demonstrated that alternative serialization methods for the HTM classifier may be used without compromising the serialized instance's integrity.

The capacity of the HTM classifier to tolerate empty spaces during deserialization was the subject of the third unit test. In order to ensure that the deserialized object was not null, we inserted an empty space to the serialized file. This test proved how adaptable the HTM classifier's deserialization skills are and how well it can cope with unforeseen input.

Overall, the HTM classifier's effective serialization implementation offers important advantages for its practical use. It enables 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.

# References

|  |  |
| --- | --- |
| [1] | G. K. F. Z. Jahan Balasubramaniama, "Enhancement of Classifiers in HTM-CLA Using Similarity Evaluation Methods," *International Conference on Knowledge Based and Intelligent Information and Engineering Systems.* |
| [2] | J. &. G. D. Hawkins, "Hierarchical temporal memory including HTM cortical learning algorithms," *The handbook of brain theory and neural networks,* pp. 147-151, 2009. |
| [3] | G. K. F. Z. Jahan Balasubramaniama, "Enhancement of Classifiers in HTM-CLA Using Similarity Evaluation Methods," *19th International Conference on Knowledge Based and Intelligent Information and Engineering Systems .* |
| [4] | Hawkins and Ahmad, "provide a comprehensive overview of Hierarchical Temporal Memory (HTM) systems and their use of Sparse Distributed Representations (SDRs) as the fundamental data structure.," 2016. |
| [5] | Y. A. S. H. J. &. C. Y. Cui, "Hierarchical temporal memory for anomaly detection in crowded scenes," *IEEE Transactions on Circuits and Systems for Video Technology,* pp. 385-393, 2016. |
| [6] | [Online]. Available: https://github.com/antonyaneeta/neocortexapi/blob/team\_alpha/source/NeoCortexApi/Classifiers/HtmClassifier.cs. |
| [7] | [Online]. Available: https://github.com/antonyaneeta/neocortexapi/blob/team\_alpha/source/UnitTestsProject/Classifiers/HtmClassiferSerializationTests.cs. |