Information Technology Course
Module: Individual Project
By Damir Dobric / Andreas Pech

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

HTM Serialization and Deserialization

***Abstract*—Hierarchical temporal memory (HTM) provides a theoretical framework that models several fundamental computational principles of the Neocortex. The Spatial Pooler (SP), an essential component of HTM, models how neurons generate effective representations of information by learning feedforward connections. It uses a mix of competitive Hebbian learning principles and homeostatic excitability control to transform arbitrary binary input patterns into sparse distributed representations (SDRs). Serialization and Deserialization of Spatial Pooler are to store the Spatial Pooler state into a format and load the Spatial Pooler back with the original saved object. Storing an object helps to reuse it in every application. The purpose of this study analyses few Serialization techniques for Spatial Pooler and attempts to provide a best fit Serialization Model.**

Keywords: **Hierarchical temporal memory, Neocortex, spatiotemporal, Spatial Pooler, Sparse Distributed Representations, Serialization, Deserialization.**

I. Introduction

Our brain gets massive amounts of information concerning the outside environment on a continual basis via peripheral sensors, which convert changes in light intensity, sound pressure, and skin deformations into millions of spike trains. Each cortical neuron must realize the sensation of a tidal input flood by establishing synaptic connections to a presynaptic neuron subset. Our perception and behaviour are influenced by the overall activity pattern of groups of neurons. Neuroscience recognizes how individual cortical neurons can react to distinct pitch patterns of input and how a group of neurons overall represents characteristics of the inputs in a flexible, dynamic, yet robust way [1].

HTM is mostly applied in streaming data to detect anomalies today. The approach focuses on neurology and the anatomy, and engagement of the pyramid-shaped neurons in the Neocortex of the mammalian anatomy (in particular, the human brain).

HTM learning algorithms are also known as cortical learning algorithms (CLA). It implements a dataformat called sparse distributed representations (a data format whose elements are binary, 1 or 0, and whose number of 1 bits is small compared to the number of 0 bits) to represent the brain activity and a more biologically-realistic neuron model (often also referred to as a cell, in the context of HTM). Two key components of this HTM generation are a spatial pooling algorithm and a sequence memory algorithm. The spatial pooling algorithm generates sparse distributed representations (SDR) as output, and the sequence memory algorithm adapts to depict and predict complex sequences.

The cerebral cortex is partially represented and handled by the layers and minicolumns. Each HTM layer is made up of a series of interconnected minicolumns. An HTM layer produces a sparse distributed representation from its input, such that a fixed proportion of minicolumns are active at any instant. A minicolumn is considered as a group of cells that have the same receptive field. There are numerous cells in every minicolumn, which can recall certain previous states. Three different cell states can be defined, which are active, inactive and predictive state.

At any given time, a cell (or a neuron) in a minicolumn can be found in active, inactive, or predictive. At first, cells are inactive. If a cell (one or more than one) in the active minicolumn is found to be in the predictive state, they will be the only cells to become active in the current time step. If none of the cells in the active minicolumn are found to be in the predictive state, all cells are made active. This usually happens during the initial time step or when the activation of this minicolumn was not expected. Whenever a cell becomes active, connections to neighbouring cells (which tend to be active during several previous time steps) are gradually created. Thus, a cell discovers to identify a known sequence by determining whether the cells that are attached to it are active. If a significant number of connected cells are found to be active, this cell flips to the predictive state in expectation of one of the few next inputs of the sequence. A layer's output contains minicolumns in both active and predictive states. As a result, minicolumns are active for extended periods of time, resulting in better temporal stability observed by the parent layer.

In HTM theory, different cells within a minicolumn represent this feedforward input in different temporal contexts. The SP models synaptic growth in the proximal dendritic segments. Since cells in a minicolumn share the same feedforward classical receptive field, the SP models how this common receptive field is learned from the input. The SP output represents the activation of minicolumns in response to feedforward inputs.

The SP models local inhibition among neighbouring minicolumns. This inhibition implements a k-winners-take-all computation. At any time, only a small fraction of the minicolumns with the most active inputs become active. Feedforward connections onto active cells are modified according to Hebbian learning rules at each time step. A homeostatic excitatory control mechanism operates on a slower time scale. The mechanism is called “boosting” because it increases the relative excitability of mini-columns that are not active enough. Boosting encourages neurons with insufficient connections to become active and participate in representing the input.

The minicolumn of each space pooler establishes synaptic links with the collection of input neurons. And the state of the SP can be stored using the serialization techniques which are discussed [2].

II. Serialization and Deserialization

The process of turning an item into a stream of bytes in order to store or transfer it to memory, a database, or a file is known as Serialization. Its major function is to store the state of an item so that it may be recreated later. Deserialization is the opposite of Serialization.

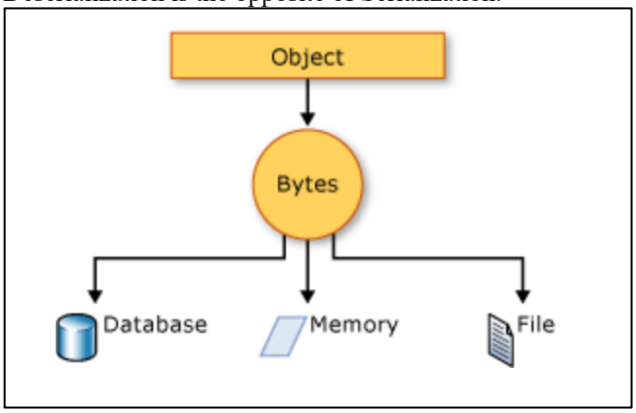

Figure 1 Serialisation of an Object

The data is serialized from the object into a stream. Information regarding the object's kind, such as version, culture, and assembly name, may also be included in the stream. The item can be saved in a database, a file, or memory using that stream.

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

- Sending the object to a remote application by using a web service
- Passing an object from one domain to another
- Passing an object through a firewall as a JSON or XML string
- Maintaining security or user-specific information across applications

Serialization can be done in many ways as follows:

1) XML Serialization:

XML serialization serializes an object's public fields and properties, or method parameters and return values, into an XML stream that adheres to a certain XML Schema defining language (XSD) specification. XML serialization leads to highly typed classes with public properties and XML-converted fields. System.Xml. Serialization includes XML serialization and deserialization classes. To manage how an XmlSerializer serializes or deserializes a class instance, you use the attributes of classes and class members.

To specify instances of the type, apply the SerializableAttribute property to a type. If you try to serialize a type that does not have the SerializableAttribute attribute, an error is thrown. Use the NonSerializedAttribute attribute to prohibit a field from being serialized. If a field of a serialized type contains a pointer, handle or some other specialized data structure, and the field is not significantly reconstituted in a new environment, then you may choose to make it non-serializable. When a serialized class contains references to the SerializableAttribute objects of other classes, these objects will likewise be serialized. However, XML employs more words to express the purpose. It is sometimes more than necessary. Parsing XML software is a slow and tedious job. This costs in terms of memory consumption. Whereas JSON is less verbose and faster. Moreover, JSON document is more readable compared to XML [3].

2) JSON Serialization:

The Text.Json namespace offers classes for serializing and deserializing JavaScript Object Notation (JSON). JSON is an open standard extensively used for web-based data exchange.

JSON serialization serializes an object's public properties into a string, byte array, or stream. To control the serialization or deserialization of JsonSerializer in a class instance:

- Use a JsonSerializerOptions object
- Apply attributes from the System.Text.Json.Serialization namespace to classes or properties.

The JSON Serialization does not serialize all private properties. The data is also more compact and legible to store in a text file. This can be done by using Reflection

3) Reflection:

Reflection is any programming system functionality that allows a programmer to observe and alter code elements without knowing their identity or formal structure ahead of time. Inspection is the process of studying things and types in order to collect structured information about their definition and behavior. Apart from some essential supplies, this is usually done with little or no prior awareness of them.

Custom Attributes can be accessed using Reflection, as well as dynamically loading an assembly and creating instances, invoking methods, and learning about private/public methods.

III. Implementation of Reflection Methodology

As already mentioned, in any serialization mechanism the public and private fields of an object, the assembly containing the class, are converted to a stream of bytes which is then written to a data stream. Through this process the object is stored in a storage medium. When deserialized it restores the exact state of the object. In this project, Spatial Pooler is a class in the HTM software model that encapsulates various data properties and data methods. The idea is to implement the reflection technique to serialize all the properties of Spatial pooler class. Additionally, Spatial pooler contains complex properties which need some special attention which will be discussed further. The

whole structure of the spatial pooler class is given in the UML diagram below.

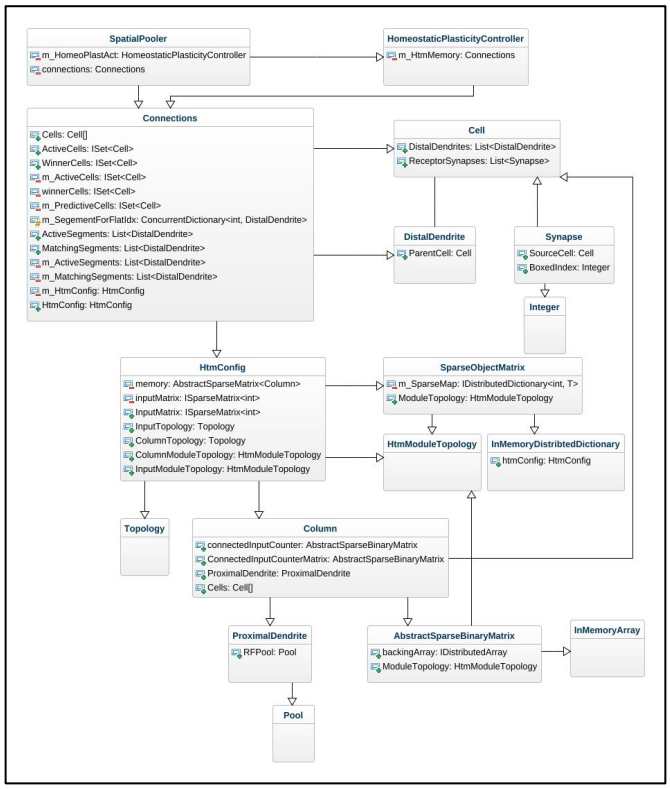

Figure 2 Complex Properties of Spatial Pooler

This diagram gives the brief idea about the complex properties of spatial pooler. Reflection technique involves creation of a serialization method, deserialization and an equals method in the spatial pooler class. The methods for serialization, deserialization and equals are named as Serialize(), Deserialize() and Equals(). The graphical representation of the steps involved in the serializing/deserializing a class are given in the flowchart diagram below.

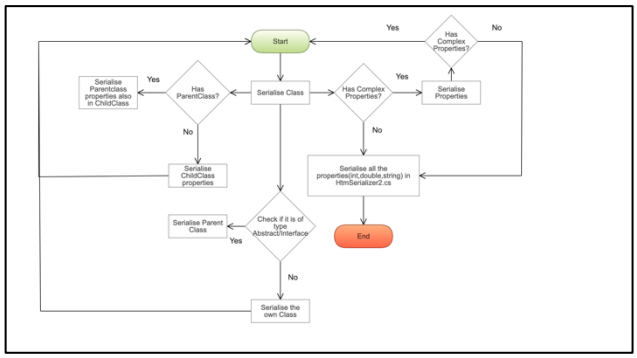

Figure 3 Serializing and Deserializing a class

The outline of this project can mainly be descried in three parts as following:

a) Implementing Serialize(), Deserialize() and Equals() methods in complex classes.

b) Identifying complex properties and implementing the above three methods for relevant classes.

c) Performing Unit Test to validate Serialization and Deserialization functionality

Serialize() and Deserialize() Method in SpatialPooler class:

Two methods named Serialize() and Deserialize() have been implemented inside the class SpatialPooler.cs. SpatialPooler.cs is under NeoCortexApi directory in HTM.

```
public void Serialize(StreamWriter writer)
{
    HtmSerializer2 ser = new HtmSerializer2();

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

    ser.SerializeValue(this.MaxInibitionDensity, writer);
    ser.SerializeValue(this.Name, writer);

    if (this.m_HomeoPlastAct != null)
    {
        this.m_HomeoPlastAct.Serialize(writer);
    }

    if (this.connections != null)
    {
        this.connections.Serialize(writer);
    }

    ser.SerializeEnd(nameof(SpatialPooler), writer);
}
```

Figure 4 Serialization of Spatial Pooler

This Serialize function creates an instance of the HtmSerializer2 class and initiates a function called SerializeBegin with the created instance. This SerializeBegin function has been implemented in HtmSerializer2 class. SerializeBegin and SerializedEnd are used to identify the class start and end in text file for a user. The Serialization for the basic properties such as int, double, bool, string, array, list, dictionary are implemented using SerializeValue method. All the properties serialized in HtmSerializer2 are listed below.

Property	Serialization Method	Deserialization Method
Int	SerializeValue()	ReadIntValue()
Double	SerializeValue()	ReadDoubleValue()
String	SerializeValue()	ReadStringValue()
Long	SerializeValue()	ReadLongValue()
Bool	SerializeValue()	ReadBoolValue()
Random	-	ReadRandomValue()
Double[]	SerializeValue()	ReadArrayDouble()
Int[]	SerializeValue()	ReadArrayInt()
Cell[]	SerializeValue()	DeserializeCellArray()
List<DistalDendrite>	SerializeValue()	-
List<Synapse>	SerializeValue()	-

Dictionary<String, int>	SerializeValue()	ReadDictSIValue()
Dictionary<int, int>	SerializeValue()	ReadDictionaryIIValue
Dictionary<String, int[]>	SerializeValue()	ReadDictSIarray()
List<int>	SerializeValue()	ReadListInt()
Dictionary<Segment, List<Synapse>>	SerializeValue()	-
Dictionary<Cell, List<DistalDendrite>>	SerializeValue()	-
Dictionary<int, Synapse>	SerializeValue()	ReadKeyISValue()
ConcurrentDictionary<int, DistalDendrite>	SerializeValue()	ReadCKeyIDValue()

Table 1 Methods in Htmserializer2

These are placed in the .txt file in such a way that the properties are clearly well formatted after serializing. After SerializeBegin a function named SerializeValue is called from the HtmSerializer2 class. This method is called with different arguments. These arguments are the properties which a class holds. This technique is called method overloading and can be performed by changing the number of arguments and the data type of the arguments . It provides flexibility so that we can call the same method for different types of properties.

```
public static SpatialPooler Deserialize(StreamReader sr)
{
    SpatialPooler sp = new SpatialPooler();

    HtmSerializer2 ser = new HtmSerializer2();

    while (sr.Peek() >= 0)
    {
        string data = sr.ReadLine();
        if (data == String.Empty || data == ser.ReadBegin(nameof(SpatialPooler)))
        {
            continue;
        }
        else if (data == ser.ReadBegin(nameof(HomeostaticPlasticityController)))
        {
            sp.m_HomeoPlastAct = HomeostaticPlasticityController.Deserialize(sr);
        }
        else if (data == ser.ReadBegin(nameof(Connections)))
        {
            sp.connections = Connections.Deserialize(sr);
        }
        else if (data == ser.ReadEnd(nameof(SpatialPooler)))
        {
            break;
        }
        else
        {
            string[] str = data.Split(HtmSerializer2.ParameterDelimiter);
            for (int i = 0; i < str.Length; i++)
            {
                switch (i)
                {
                    case 0:
                        {
                            sp.MaxInibitionDensity = ser.ReadDoubleValue(str[i]);
                            break;
                        }
                    case 1:
                        {
                            sp.Name = ser.ReadStringValue(str[i]);
                            break;
                        }
                    default:
                        { break; }

                }
            }
        }
    }

    return sp;
}
```

Figure 5 Deserialization of Spatial Pooler

Finally, the SerializeEnd method will be called by passing the required arguments, so that the serialization of that particular class is completed.

Serialization of abstract and interfaces are carried out in the parent class. Now the serialisation of parentclass will include all the properties of childclass. Therefore, serializing the abstract/interface class is fulfilled.

As discussed, the Deserialize method is implemented in the SpatialPooler class and it returns a spatial pooler object. The code snippet is given in the below figure. It initializes two instances each from the spatial pooler and Htmserializer classes respectively. SpatialPooler instance is initialized with either 0 or null according to the property. The function tries to read the .txt file using StreamReader content line by line by checking the provided condition. After validation the function reads each line in the .txt file and append all the read information into a spatial pooler object. The function returns this spatial pooler object after deserializing each property.

The ReadBegin(), ReadEnd() and all the methods to deserialize the basic properties are implemented in HtmSerializer2. The deserialization method for complex properties are implemented in their respective classes. All the properties serialized using methods either in HtmSerializer2/Serialize() are also deserialized in HtmSerialiser2/Deserialize() methods. The properties deserialized are also listed above.

Equals() Method in class:

The Equals method is to validate the serialized and deserialized properties. This method also follows the same procedure as the serialize and deserialize methods.

Basic Properties (int, bool, string, double, long) are validated using != , whereas properties such as (array, list and dictionary) are checked for equality using SequenceEqual and all the remaining complex properties are implemented with Equals() method prior to checking if either of them is null or not equal to null.

```
public bool Equals(SpatialPooler obj)
{
    if (this == obj)
        return true;

    if (obj == null)
        return false;

    if (MaxInibitionDensity != obj.MaxInibitionDensity)
        return false;
    else if (Name != obj.Name)
        return false;

    SpatialPooler other = (SpatialPooler)obj;

    if (m_HomeoPlastAct == null)
    {
        if (other.m_HomeoPlastAct != null)
            return false;
    }
    else if (!m_HomeoPlastAct.Equals(other.m_HomeoPlastAct))
        return false;

    if (connections == null)
    {
        if (other.connections != null)
            return false;
    }
    else if (!connections.Equals(other.connections))
        return false;

    return true;

}
```

Figure 6 Equals methods in Spatial Pooler

The Equals method in SpatialPooler is as below and the same way is used to check the similarity between properties.

Serialize() and Deserialize() Method for complex properties:

Spatial Pooler contains complex properties such as connections, HomeoStaticPlasticityController etc. These properties require some special attention to serialize and deserialize. The complex properties are serialized similarly as described above with SerializeBegin, SerializeValue(for Basic Properties) and SerializeEnd methods. These properties need to be checked whether they are of null value or not. If not, we have to serialize the current instance of the property by Serialize methods which has to be implemented in the relevant class. As the complex properties sometimes refer to be null, check for it being null before serializing the property. The parameters for the proper format in the .txt file are also to be taken care of.

```
if (this.m_HomeoPlastAct != null)
{
   this.m_HomeoPlastAct.Serialize(writer);
}
```

If there are any other complex properties present inside the HomeoStaticPlasticityController class again the same procedure should be implemented which is to implement the serialize and Deserialize methods in the relevant classes. This process goes on by looping into the classes as shown in the class diagram in the above section.

Unit Tests:

Unit testing is an important part of your software development workflow, it has the biggest impact on the quality of your code. Unit tests evaluate the behaviour of the code in response to standard, boundary, and wrong cases of input data, as well as any explicit or implicit assumptions made by the code [4].

A popular technique of designing unit tests for a method under test is to use the AAA (Arrange, Act, Assert) pattern.

- The **Arrange** part of a unit test method establishes the value of the data provided to the method under test and initializes objects.
- The **Act** section runs the test procedure with the specified parameters.
- The **Assert** section ensures that the method under test's operation is as expected.

Unit Tests for our project are carried out in HTMSerializationTests class.

Initially, UnitTest for all the complex properties are performed. For an instance, consider Column (which is a complex property). Column is initialised using Arrange section described above. As the constructor of Column has parameters int, int, double and int (numCells, colIndx, synapsePermConnected and numInputs respectively), Column is initialized with int, int, double and int which is named as matrix. After the computation made using the constructor, all the properties are initialized. The calculated properties are written into ser_SerializeColumnTest.txt file in the form of a stream. Using StreamReader the stream/String present in the text file is read line by line and retrieved back into the Column object, column. The state of the object is stored in the test file, which can also be used back at any point of time. Basically, Serializing and Deserializing comes under the Act Section which is a test to analyze the behaviour of the serialization and deserialization methods implemented in the Column class. Another test is using Assert to check the equality of the objects, matrix and matrix. The UnitTest for Column is as follows:

```
[TestMethod]
[TestCategory("Serialization")]
[DataRow(2, 5, 8.3, 2)]
public void SerializeColumnTest(int numCells, int colIndx, double synapsePermConnected, int numInputs)
{
    Column matrix = new Column(numCells, colIndx, synapsePermConnected, numInputs);

    using (StreamWriter sw = new StreamWriter($"ser_{nameof(SerializeColumnTest)}.txt"))
    {
        matrix.Serialize(sw);
    }
    using (StreamReader sr = new StreamReader($"ser_{nameof(SerializeColumnTest)}.txt"))
    {
        Column column = Column.Deserialize(sr);

        Assert.IsTrue(matrix.Equals(column));
    }
}
```

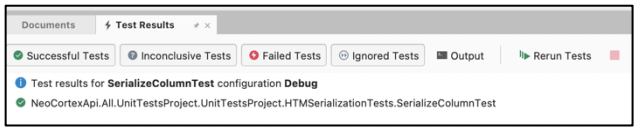

Figure 7 UnitTest

The Serialization and Deserialization of Column is successful which is shown above. The successful UnitTest in HTMSerializationTests are HtmConfig, Cell, DistalDendrite, Synapse, Pool, ProximalDendrite, HtmModuleTopology, Integer and SegmentActivity.

```
 BEGIN 'Column'
0 | 5 |
 BEGIN 'SparseBinaryMatrix'
0,|
 BEGIN 'HtmModuleTopology'
1,2,| False | 2,1,|
 END 'HtmModuleTopology'

 BEGIN 'InMemoryArray'

00 | 1,2,| 1 |
 END 'InMemoryArray'

 END 'SparseBinaryMatrix'

 BEGIN 'SparseBinaryMatrix'
0,|
 BEGIN 'HtmModuleTopology'
1,2,| False | 2,1,|
 END 'HtmModuleTopology'

 BEGIN 'InMemoryArray'

00 | 1,2,| 1 |
 END 'InMemoryArray'

 END 'SparseBinaryMatrix'

 BEGIN 'ProximalDendrite'
5 | 8.300 | 2 |
 BEGIN 'Integer'
5 |
 END 'Integer'
|
 END 'ProximalDendrite'

 BEGIN 'Cell'
10 | 0 | 5 |
 END 'Cell'

 BEGIN 'Cell'
11 | 0 | 5 |
 END 'Cell'
|
 END 'Column'
```

Figure 8 State of a class

The properties present in Column are CellId(int), connectedInputCounter(AbstractSparseBinaryMatrix), ConnectedInputCounterMatrix(AbstractSparseBinaryMatrix), ProximalDendrite(ProximalDendrite) and Index(int).

The ser_SerializeColumnTest.txt text file starts and ends with BEGIN 'Column' and END 'Column'. All the properties are written with their values which are obtained after initializing the Class separated by ParameterDelimiter ('|'). Complex Properties are started and ended in the same way along with their respective class name expect for the abstract and interface. For the abstract and interface types of classes begin and end starts with their Parentclass name.

Please refer to the above image to get clear understanding. In the example Column, the complex properties are connectedInputCounter, ConnectedInputCounterMatrix and ProximalDendrite serialized in the similar way.

IV. Conclusion

The Serialization and Deserialization techniques that are proposed in this project is able to successfully Serialize the simple and complex properties of Spatial Pooler in a standard format in the .NET environment. Hence it can be integrated with the existing HTM module. However, the Serialization approach that has been attempted in this project appears to be promising since it can replicate a few of the complex properties in the destination Object but it is realized that the implementation is not complete. Further research is continued to identify some of the issues with this approach for ISet<T> which is used as a property type in Connections class.

References

[1] A. S. a. J. Y. Cui, "The HTM Spatial Pooler Distributed Coding," Comput. Neurosci, 2017.

[2] "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Hierarchial_temporal_memory.

[3] "Microsoft," [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/serialization.

[4] "Microsoft," [Online]. Available: https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2019.