**Implement HTM Persistance**

Nabeela Maham  
line 2: email address

Kizito   
line 2: email address

Muhammad Jasim

line 2: email address

*Abstract*—

The Aim of this project is to create a system which trains the system and uses the data from the previously applied experiments. For the next time, the system will use the previous data. The persistence is designed as implementation of a custom serializer / desterilizer. The serializer saves the instance of some HTM module to the stream and deserializer is responsible to create the instance from the stream.

Keywords— - SparseObjectMatrix<T>, AbstractMatric, AbstractFlatMatrix, AsbstractBinaryMatrix, SpatialPooler, TemporalMemory, Column, Synapse, Topology, DistalDendriteSegment, ProximalDendriteSegment, Segment, SegmentActivity, Pool, Connections, InMemoryDistributedDictionaryspatial Pooler,

# **Intro (*Heading 1*)**

Hierarchical temporal memory (HTM) is a biologically constrained machine intelligence technology developed by Numenta. Originally described in the 2004 book. On Intelligence by Jeff Hawkins with Sandra Blakeslee, HTM is the distributed building and primarily used today for anomaly detection in streaming data. The technology is based on neuroscience and the physiology and interaction of pyramidal neurons in the neocortex of the mammalian (in particular, human) brain.

# persisting an object using C#

You can use serialization to persist an object's data between instances, which enables you to store values and retrieve them the next time that the object is instantiated.

## Persist the Object using serialization.

In order to persist the values for the class, you must first mark the class with the**Serializable** attribute. Add the following code above the class definition:

[Serializable()]

The SerializableAttribute tells the compiler that everything in the class can be persisted to a file. Because the PropertyChanged event does not represent part of the object graph that should be stored, it should not be serialized. Doing so would serialize all objects that are attached to that event. You can add the NonSerializedAttribute to the field declaration for the PropertyChanged event handler.

[field: NonSerialized()]

public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;

Beginning with C# 7.3, you can attach attributes to the backing field of an auto-implemented property using the field target value. The following code adds a TimeLastLoaded property and marks it as not serializable:

[field:NonSerialized()]

public DateTime TimeLastLoaded { get; set; }

The next step is to add the serialization code to the LoanApp application. In order to serialize the class and write it to a file, you use the [System.IO](https://docs.microsoft.com/en-us/dotnet/api/system.io) and [System.Runtime.Serialization.Formatters.Binary](https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatters.binary) namespaces. To avoid typing the fully qualified names, you can add references to the necessary namespaces as shown in the following code:

using System.IO;

using System.Runtime.Serialization.Formatters.Binary;

The next step is to add code to deserialize the object from the file when the object is created. Add a constant to the class for the serialized data's file name as shown in the following code:

const string FileName = @"../../../SavedLoan.bin";

Next, add the following code after the line that creates the TestLoan object:

if (File.Exists(FileName))

{

Console.WriteLine("Reading saved file");

Stream openFileStream = File.OpenRead(FileName);

BinaryFormatter deserializer = new BinaryFormatter();

TestLoan = (Loan)deserializer.Deserialize(openFileStream);

TestLoan.TimeLastLoaded = DateTime.Now;

openFileStream.Close();

}

You first must check that the file exists. If it exists, create a [Stream](https://docs.microsoft.com/en-us/dotnet/api/system.io.stream) class to read the binary file and a [BinaryFormatter](https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatters.binary.binaryformatter) class to translate the file. You also need to convert from the stream type to the Loan object type.

Next you must add code to serialize the class to a file. Add the following code after the existing code in the Main method:

Stream SaveFileStream = File.Create(FileName);

BinaryFormatter serializer = new BinaryFormatter();

serializer.Serialize(SaveFileStream, TestLoan);

SaveFileStream.Close();

At this point, you can again build and run the application. The first time it runs, notice that the interest rates starts at 7.5, and then changes to 7.1. Close the application and then run it again. Now, the application prints the message that it has read the saved file, and the interest rate is 7.1 even before the code that changes it.

SERIALIZATION

Serialization is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization.

## How serialization works

This illustration shows the overall process of serialization:



The object is serialized to a stream that carries the data. The stream may also have information about the object's type, such as its version, culture, and assembly name. From that stream, the object can be stored in a database, a file, or memory.

### Uses for serialization

Serialization allows the developer to save the state of an object and re-create it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform 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

## JSON serialization

The [System.Text.Json](https://docs.microsoft.com/en-us/dotnet/api/system.text.json) namespace contains classes for JavaScript Object Notation (JSON) serialization and deserialization. JSON is an open standard that is commonly used for sharing data across the web.

JSON serialization serializes the public properties of an object into a string, byte array, or stream that conforms to [the RFC 8259 JSON specification](https://tools.ietf.org/html/rfc8259). To control the way [JsonSerializer](https://docs.microsoft.com/en-us/dotnet/api/system.text.json.jsonserializer) serializes or deserializes an instance of the class:

* Use a [JsonSerializerOptions](https://docs.microsoft.com/en-us/dotnet/api/system.text.json.jsonserializeroptions) object
* Apply attributes from the [System.Text.Json.Serialization](https://docs.microsoft.com/en-us/dotnet/api/system.text.json.serialization) namespace to classes or properties
* [Implement custom converters](https://docs.microsoft.com/en-us/dotnet/standard/serialization/system-text-json-converters-how-to)

## Binary and XML serialization

The [System.Runtime.Serialization](https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization) namespace contains classes for binary and XML serialization and deserialization.

Binary serialization uses binary encoding to produce compact serialization for uses such as storage or socket-based network streams. In binary serialization, all members, even members that are read-only, are serialized, and performance is enhanced.

 WARNING: Binary serialization can be dangerous. For more information, see [BinaryFormatter security guide](https://docs.microsoft.com/en-us/dotnet/standard/serialization/binaryformatter-security-guide).

XML serialization serializes the public fields and properties of an object, or the parameters and return values of methods, into an XML stream that conforms to a specific XML Schema definition language (XSD) document. XML serialization results in strongly typed classes with public properties and fields that are converted to XML. [System.Xml.Serialization](https://docs.microsoft.com/en-us/dotnet/api/system.xml.serialization) contains classes for serializing and deserializing XML. You apply attributes to classes and class members to control the way the [XmlSerializer](https://docs.microsoft.com/en-us/dotnet/api/system.xml.serialization.xmlserializer) serializes or deserializes an instance of the class.

### Making an object serializable

For binary or XML serialization, you need:

* The object to be serialized
* A stream to contain the serialized object
* A [System.Runtime.Serialization.Formatter](https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatter) instance

Apply the [SerializableAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.serializableattribute) attribute to a type to indicate that instances of the type can be serialized. An exception is thrown if you attempt to serialize but the type doesn't have the [SerializableAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.serializableattribute) attribute.

To prevent a field from being serialized, apply the [NonSerializedAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.nonserializedattribute) attribute. If a field of a serializable type contains a pointer, a handle, or some other data structure that is specific to a particular environment, and the field cannot be meaningfully reconstituted in a different environment, then you may want to make it nonserializable.

If a serialized class contains references to objects of other classes that are marked [SerializableAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.serializableattribute), those objects will also be serialized.

### Basic and custom serialization

Binary and XML serialization can be performed in two ways, basic and custom.

Basic serialization uses .NET to automatically serialize the object. The only requirement is that the class has the [SerializableAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.serializableattribute) attribute applied. The [NonSerializedAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.nonserializedattribute) can be used to keep specific fields from being serialized.

When you use basic serialization, the versioning of objects may create problems. You would use custom serialization when versioning issues are important. Basic serialization is the easiest way to perform serialization, but it does not provide much control over the process.

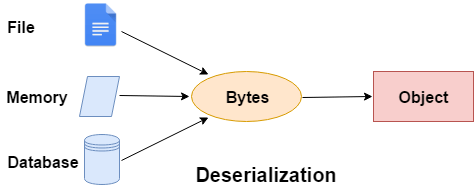
In custom serialization, you can specify exactly which objects will be serialized and how it will be done. The class must be marked [SerializableAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.serializableattribute) and implement the [ISerializable](https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.iserializable) interface. If you want your object to be deserialized in a custom manner as well, use a custom constructor.

Serialization in HTM Persistance:

# 

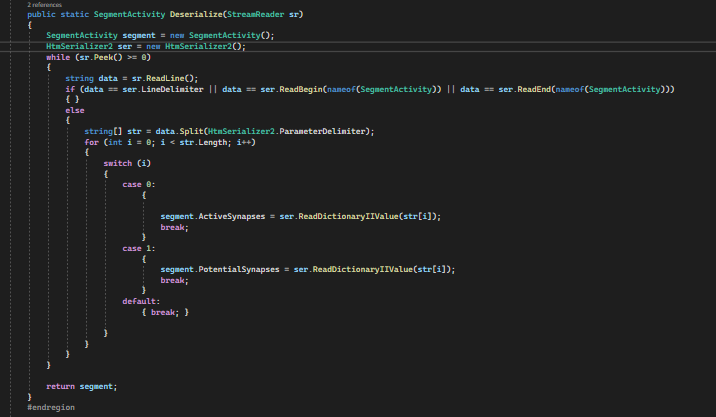
# Deserialization

Deserialization is the reverse process of serialization. It means you can read the object from byte stream. Here, we are going to use **BinaryFormatter.Deserialize(stream)** method to deserialize the stream.



### **C# Deserialization Example**

Derialization in HTM Persiatnce



The project in the folder HTMPersistanceUnitTest



For Instance:

A person is perform an activity and uring that activity the user is unable to do anything else such chatting with family and friends, watching videos and listening sound. He stopped the activity and start doing the chatting with friends and performed his office work as well. Now, he restarted the activity. HTM persistence helps to resume the activity from where it was stopped. Serializer saves the data into TXT file and deserialize get that TXT file and give it to the system. And continue the work.

Serialization: The serializer saves the instance of some HTM module to the stream.

Deserialization: deserializer is responsible to create the instance from the stream.

# **Methods**

Classes in this Project:

I have Tested three classes in this project. These classes are listed below.

1. SegmentActivity:

SegmentActivity.cs stores the calculus of a temporal cycle. It contains the index of segments with number of synapses with permanence higher than threshold. A

Dictionary, which holds the number of potential synapses of every segment.

Potential synspses are all established synapses between receptor cell and the segment's cell. Receprot cell was active cell in the previous cycle.

link to SegmentActivity

(https://github.com/nabeelamaham/neocortexapi/blob/Nabeela-HTMPersistance/source/NeoCortexEntities/Entities/SegmentActivity.cs)

~~~csharp

Dictionary[segment index, number of active synapses].

Dictionary [segment index, number of potential synapses].

~~~

Serialization and deserialization have been applied on SegmentActivity.

2. ProximalDendrite:

It defines the eproximal dentritte segment. Note the segment is used during SP compute operation.

TM does not use this segment.

It uses the pool of synapses in the receptive field.

link to ProximalDendrite class :

(https://github.com/nabeelamaham/neocortexapi/blob/Nabeela-HTMPersistance/source/NeoCortexEntities/Entities/ProximalDentrite.cs)

~~~csharp

public ProximalDendrite(int colIndx, double synapsePermConnected, int numInputs) : base(colIndx, synapsePermConnected, numInputs)

        {

        }

~~~

It creates and returns a newly created synapse with the specified source cell, permanence, and index.

This method is only called for Proximal Synapses. For ProximalDendrites, there are many synapses within a pool, and in that case, the index specifies the synapse's sequence order within the pool object, and may be referenced by that index.

It returns the instance of the new synapse.

~~~csharp

public Synapse CreateSynapse(int index, int inputIndex)

~~~

Indicies of Array of connected inputs defines RF(Potential Pool). It clear all the synapses from the segment. Sets the permanences for each linked Synapse specified by the indexes passed in which identify the input vector indexes associated with the permanences passed in are understood to be in "sparse" format and therefore require the int array identify their corresponding indexes.

This is the "sparse" version of this method.

Returns an array of synapse indexes as a dense binary array. Returns an array of indexes of input neurons connected to this pool. It returns the indexes of connected input neurons.

~~~csharp

   public override void Serialize(StreamWriter writer)

        public static ProximalDendrite Deserialize(StreamReader sr)

~~~

3. DistalDendrite:

Implements a distal dendritic segment that is used for learning sequences. Segments are owned by Cells and in turn own Cells which are obversely connected to by a "source cell", which is the Cell that will activate a given Synapse owned by this Segment.

link to DistalDendriteDendrite

(https://github.com/nabeelamaham/neocortexapi/blob/Nabeela-HTMPersistance/source/NeoCortexEntities/Entities/DistalDendrite.cs)

~~~csharp

  public class DistalDendrite : Segment, IComparable<DistalDendrite>, IEquatable<DistalDendrite>

    {

~~~

**Cell:** The cell that owns (parent) the segment.  the last iteration in which this segment was active.

The sequence number of the segment. Specifies the order of the segment of the Connections instance.

**ParentCell:**

 The cell, which owns the segment.

**flatIdx:**

 The flat index of the segment. If some segments are destroyed (synapses lost permanence)then the new segment will reuse the flat index. In contrast, the ordinal number will increas when new segments are created.

lastUsedIteration

**ordinal:**

The ordinal number of the segment. This number is incremented on each new segment.

If some segments are destroyed, this number is still increment.

**synapsePermConnected**

**numInputs**

~~~csharp

        public DistalDendrite(Cell parentCell, int flatIdx, long lastUsedIteration, int ordinal, double synapsePermConnected, int numInputs) : base(flatIdx, synapsePermConnected, numInputs)

~~~

It compares this segment with the given one and Compares by index as well.

It  Serialize method for DistalDendrite

~~~ csharp

internal void SerializeT(StreamWriter writer)

        {

~~~

Then apply deserialization.

~~~csharp

        public static DistalDendrite Deserialize(StreamReader sr)

        {

~~~

**Unit Test Classes for the project.**

the following code is applid for testing the classes(**SegmentActivity, DistelDendrite, pxorimalDendrite**)

Here is the link to the unit Test class.

link to project

(https://github.com/nabeelamaham/neocortexapi/blob/Nabeela-HTMPersistance/source/HTMPersistanceUnitTests/SerializeSegmentActivityTest.cs)

~~~csharp

     public void TestSegmentActivitySErialization()

        {

            SegmentActivity segment = new SegmentActivity();

            segment.ActiveSynapses = new Dictionary<int, int>();

            segment.ActiveSynapses.Add(23, 1);

            segment.PotentialSynapses = new Dictionary<int, int>();

            segment.PotentialSynapses.Add(2, 56);

            using (StreamWriter sw = new StreamWriter($"ser\_{nameof(SerializeSegmentActivityTest)}.txt"))

            {

                segment.Serialize(sw);

            }

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

            {

                SegmentActivity segment1 = SegmentActivity.Deserialize(sr);

                Assert.IsTrue(segment1.Equals(segment));

            }

        }

        [TestClass]

        public class DistalDendriteSerializationTest

        {

            [TestMethod]

            public void TestDistalDendriteSerialization()

            {

                Cell c1 = new Cell(1, 1, 10, 1, NeoCortexEntities.NeuroVisualizer.CellActivity.ActiveCell);

                DistalDendrite d1 = new DistalDendrite(c1, 1, 1, 1, 0.5, 10);

                using (StreamWriter sw = new StreamWriter("dist\_ser.txt"))

                {

                    d1.Serialize(sw);

                }

                DistalDendrite d2;

                using (StreamReader sr = new StreamReader("dist\_ser.txt"))

                {

                    d2 = DistalDendrite.Deserialize(sr);

                }

                var result = HtmSerializer2.IsEqual(d1, d2);

                Assert.IsTrue(result);

            }

        }

        [TestClass]

        public class ProximalDendriteSerializationTest

        {

            [TestMethod]

            public void TestProximalDendriteSerialization()

            {

                Cell c1 = new Cell(1, 1, 10, 1, NeoCortexEntities.NeuroVisualizer.CellActivity.ActiveCell);

                ProximalDendrite p1 = new ProximalDendrite(1, 1.2, 2);

                using (StreamWriter sw = new StreamWriter("prox\_ser.txt"))

                {

                    p1.Serialize(sw);

                }

                ProximalDendrite p2;

                using (StreamReader sr = new StreamReader("prox\_ser.txt"))

                {

                    p2 = ProximalDendrite.Deserialize(sr);

                }

                var result = HtmSerializer2.IsEqual(p1, p2);

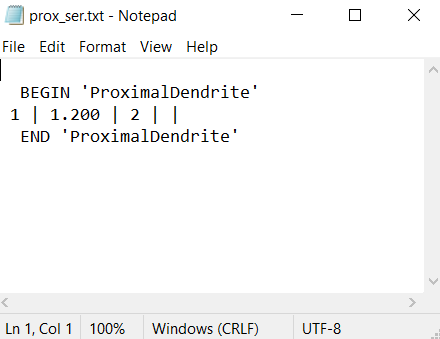
                Assert.IsTrue(result);

**Serialization logic**

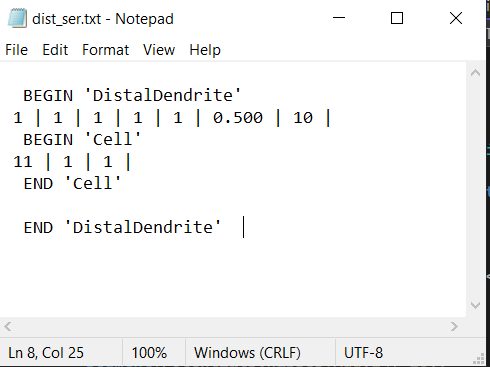
It is used for serialization and deserialization of primitive types. Such as Integer, Boolean, String, Array Int[], Double, Long. It work for non primitive type such as Synapses and cells.

 Serializes the begin and end marker of the type.

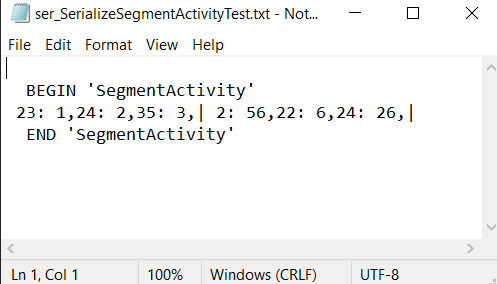
**ProximalDendrite.txt:**



**DistalDendrite.txt:**



**SegmentActivity.txt:**

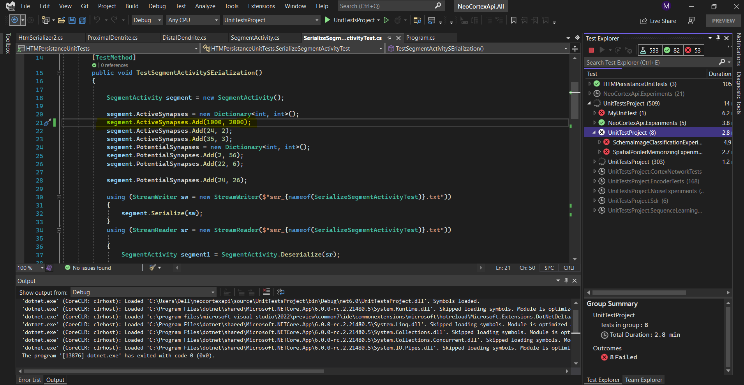


 Htm persistance serialize the property of type Int. Read the property of type Int and return that integer. Deserializes from text file to DistalDendrite and return DistalDendrite. Serialize the property of type Double, String, Long and Bool. Read the property of type Double, String, Long, Bool and return these values. Serialize the array of type Double. Read the array of type Double and return double. Serialize the array of type Int. Read the array of type Int returns Int[]. Serialize and Deserialize the array of cells. Deserializes from text file to Cell and return cells. Serialize the dictionary with key:string and value:int. Read the dictionary with key:string and value:int and return Dictionary<String, int>. Serialize the List of DistalDendrite, Synapses and Integer and Dictionary<Segment, List<Synapses>>. Read the List of DistalDendrite and returns distal dendrite. Serialize the dictionary and Concurrentdictionary with key:int and value:Synapse and DistalDendrite. Read the dictionary with key:int and value:Synapse return Dictionary<int, Synapse>

# **Results**

# Used different value and show the resuklt

# Conclusion of your i had used different values and the result is reached like this:



# we are three members in this project. Faizan has not been implemented his part because he is not present

# **Discussion**

In the conclusion, I could say that the implementation of HTM Persistance can save a lot of time. In addition, WE have implement it on visual studio using C sharp but we can work on Yaml, JSON and XML in future and perform serialization and deserialization and get better results as well.

##### **References**

1. <https://www.educba.com/deserialization-in-c-sharp/>
2. <https://www.guru99.com/c-sharp-serialization.html>
3. <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/serialization/>
4. <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/serialization/walkthrough-persisting-an-object-in-visual-studio>