Implement Temporal Memory Learning sample with Serialization

Hung Bui Tran Hai
hung.buitranhai@stud.fra-uas.de

Tanu Agarwal
tanu.agarwal@stud.fra-uas.de

***Abstract*—This paper gives an example of how Serialization with Temporal Memory can be used in the "MultiSequence Learning (MSL)" project. In this project, the serialization method is implemented to save the trained model of "MSL" and store it locally. The de-serialization method is also implemented to rebuild the objects from the file to be used as input for the next training. The project aims to make the "Predictor" object, which is the output of the "MSL" serializable. To make the serialization of the Predictor instance possible, serialization method for classes of Predictor properties such as Connections, CortexLayer (contains Temporal memory, Spatial Pooler, and encoder), and HtmClassifier are also implemented, which is our prime approach in the project. The paper demonstrates how we could implement Serialize() and Deserialize() methods in the Predictor class. The example made in this project will show how the methods for serialization and deserialization are used to save and load the trained model, and how the last trained model can be used for the next training.**

Keywords— Hierarchical Temporal Memory (HTM), Spatial Pooler (SP), Multisequence Learning (MSL), Serialization, Deserialization, Sparse Distributed Representations (SDRs), Synapses, CortexLayer, Encoder, Classifier, Connections, Predictor, Cells.

I. INTRODUCTION

Temporal memory constitutes a key element within the Hierarchical Temporal Memory Cortical Learning Algorithm (HTM CLA). This algorithm adeptly learns sequences of Sparse Distributed Representations (SDRs) generated by the Spatial Pooling algorithm, enabling predictive capabilities. The HTM comprises two main components: the Spatial Pooler and Temporal Memory. In the HTM algorithm, multiple mini-columns function as synapses, mimicking the synaptic activity of the human brain. To operate, HTM relies on numerical inputs, necessitating an encoder to convert real-world concepts into a binary format of '0's and '1's. The encoded output, known as a Binary Vector, feeds into the HTM Spatial Pooler (SP) algorithm as shown in the flow diagram of the HTM pipeline (see **Fig.1**). The Spatial Pooler transforms arbitrary binary input vectors into Sparse Distributed Representations (SDRs). These SDRs serve as input for HTM Temporal Memory, which learns patterns and predicts subsequent values in sequences based on its accumulated knowledge. The SDR Classifier, then taking active cell vectors from Temporal Memory, engages in cognitive functions such as prediction, inference, and learning as shown in Fig. 1. [1]

After initializing components, they are chained together in the Cortex Layer as HtmModule pipelines in the following order:

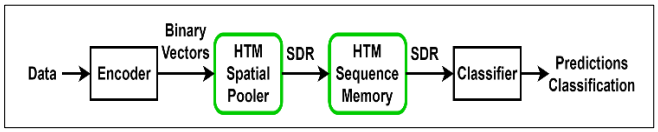

Figure 1 HTM Pipeline.[2]

Specifically, the Temporal Memory algorithm:

1) Forms a representation of the sparse input that captures the temporal context of previous inputs.
2) Forms a prediction based on the current input in the context of previous inputs.

In the Temporal Memory algorithm, when a cell activates, it establishes connections with other cells that were priorly active. This allows cells to anticipate their future activation by examining their connections. Through collective participation, all cells can store and retrieve sequences, making predictions about upcoming events. Unlike having centralized storage for a sequence of patterns, memory is decentralized across individual cells.[3]

Each cell within the system exhibits three states: active, predictive, and inactive. Cells are equipped with one proximal segment connecting the column to various bits in the input space, and multiple distal dendrite segments linking the cell to neighboring cells. During the initiation of the HTM system with a new input or in the absence of prior input context, all cells in the active column become active, a phenomenon known as bursting (refer to Fig. 2). In the presence of a previous input context, the algorithm selects a winning cell for each column based on the context of the prior input. Subsequently, other cells assume the predictive state when the connections to the currently active cells in their distal segments surpass a specified ACTIVATION_THRESHOLD.

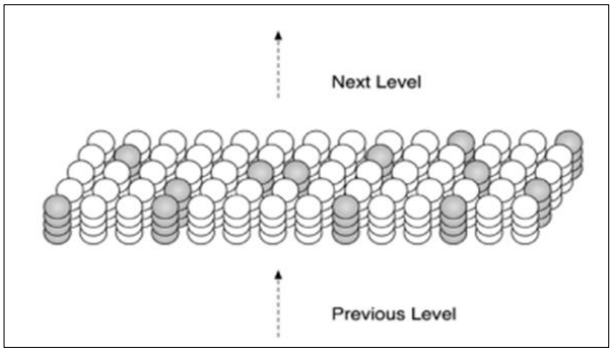

Figure 2 SDR bursting.[4]

With learning enabled, when predictive cells become active in the subsequent input, the permanence values of their active synapses increase, while those of inactive synapses decrease. However, if the system predicts incorrect cells, the active synapses of those cells are penalized by reducing their synaptic permanence. In the absence of learning, there is no alteration in the synapses of the segments. To interpret the output from Temporal Memory, a classifier, the final component in the HTM system, is employed to match the input with the predicted SDR result. This classifier encompasses an algorithm for comparing the SDR output of different input sequences and storing them as key-value pairs.

The state after prediction can be stored using the serialization technique. To make the serialization of the Predictor instance possible, serialization for classes of Predictor properties such as Connections, CortexLayer, HtmClassifier, ... should also be implemented, which is our prime approach in the project. Later, the deserialization technique is also implemented to load the last model which will be the input for the next upcoming training. The use of TM is demonstrated in the MultiSequence Learning (MSL) Experiment. The predictor which is the result of MultiSequenceLearning undergoes serialization and deserialization using the designated methods within the Predictor class. To validate the project implementation, predictions were made using both the standard predictor and the serialized predictor. The “standard predictor” acts as a normal predictor which predicts the next element without being serialized and deserialized. However, "serialized predictor" is the instance of class Predictor which is the result after serialization and deserialization of Predictor. The outcomes of predictions from both predictors were examined and compared, revealing identical results. [5]

Therefore, the outline of this project can mainly be described in two parts as following:

- Implementing Serialize(), and Deserialize() methods in the predictor class for serialization and deserialization of Predictor objects containing Connections, CortexLayer, and HtmClassifier as properties of the Predictor class.
- Comparing predictions made using the standard predictor and the serialized predictor to validate Serialization and Deserialization functionality.

Serialization: It involves transforming an object into a format suitable for streaming, enabling it to be stored in a file, database, or memory, and facilitating network transfer, as depicted in **Fig 3**. The primary objective is to capture the object's state for potential recreation when required.

Serialization involves converting the object into a stream that includes not only its data but also details about its type, such as version, culture, and assembly name. This stream can then be stored in a database, a file, or memory. Serialization enables the preservation of an object's state in a persistent store, allowing the object to be reconstructed from the saved information when needed in the future. It provides the capability to serialize (persist) any object that is not immediately required, has been utilized in another application, or may be employed later. [6] [7]

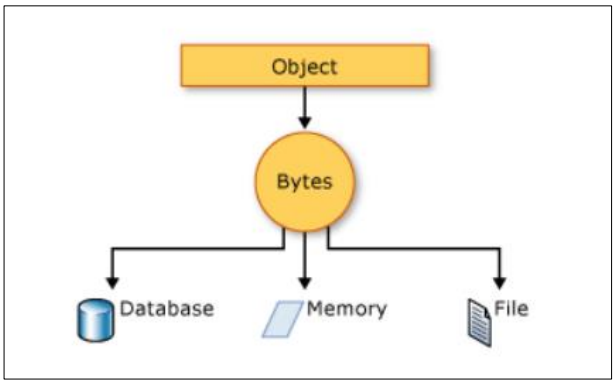

Figure 3 Serialization of an object.[7]

Deserialization: It is the counterpart of Serialization, and involves retrieving a serialized object for future use. Essentially, it restores the object's state by configuring properties, fields, and other relevant attributes.[8]

II. Methods

To implement serialization with TM for the MSL project, we need to first make the “Predictor” which is the output of MSL become serializable. Hence, the methods used for serialization and deserialization of the Predictor need to be implemented. After that, a comparison of predictions was made using the standard predictor and the serialized predictor to validate Serialization and Deserialization functionality.

1. Serialization

In this project, the Predictor class inherits the interface ISerializable which defines serialization methods. As you can see from the code in **Listing 1**.

```
public class Predictor : ISerializable
// Predictor inherits the interface Iserializable

public interface ISerializable
{
        // Serialization Methods
        void Serialize(object obj, string name,
        StreamWriter sw);

        static object Deserialize<T>(StreamReader sr,
        string name) => throw new
        NotImplementedException();
 }
```

Listing 1: Interface ISerializable.

Now our work is to implement Serialize() and Deserialize() methods in the Predictor class. The Predictor class contains three objects of class Connections, CortexLayer, and HtmClassifier (**Listing 2**). We are supposed to serialize every object within the predictor. The code below shows the objects in the predictor:

```
public class Predictor : ISerializable
{
    private Connections connections { get; set;
    }

    private CortexLayer<object, object> layer {
    get; set; }

    private HtmClassifier<string, ComputeCycle>
    classifier { get; set; }
```

Listing 2: Objects in Predictor.

The Predictor.Serialize() method (**Listing 3**) will serialize all the objects in the predictor. It will call the Connections.Serialize(), CortexLayer.Serialize(), and HtmClassifier.Serialize() methods to serialize the connections, layer, and classifier in the Predictor instance respectively. That means we have to implement the Serialize() methods for these objects first to make the Predictor.Serialize() method work. You can see the implementation of Predictor.Serialize() method below:

```
public void Serialize(object obj, string name,
StreamWriter sw)
 {
        if (obj is Predictor predictor)
        {

        // Serialize the Connections in Predictor
        instance
        var connections = predictor.connections;

        connections.Serialize(connections, null, sw);

        // Serialize the CortexLayer in Predictor
        instance
        var layer = predictor.layer;
        layer.Serialize(layer, null, sw);

        // Serialize the HtmClassifier object in
        Predictor instance
        var classifier = predictor.classifier;
        classifier.Serialize(classifier, null, sw);

        }
}
```

Listing 3: Predictor.Serialize() Code.

The Predictor.Serialize() needs three input arguments, the most important two inputs are the Predictor instance that we want to serialize and the stream writer needed for serialization. In order to make the program cleaner and easier to use, we implemented a Predictor.Save() method (**Listing 4**). The save method will take the name of the file where you want to save the Predictor instance to and the Predictor instance as the input arguments. When somebody invokes the Predictor.Save() method and provides a file name. The method will create a stream writer from the file name and call the Predictor.Serialize() method to serialize the Predictor instance. You can see how the Predictor.Save() method implemented below:

```
    public static void Save(object obj, string
fileName)
    {
        if (obj is Predictor predictor)
        {
         HtmSerializer.Reset();
         using (StreamWriter sw = new
         StreamWriter(fileName))
            {
             predictor.Serialize(obj, null, sw);
             //predictor.Serialize(sw);
             }
        }
    }
```

Listing 4: Predictor.Save() Code.

We have mentioned that we need to implement Serialize() methods for Connections, CortexLayer, and HtmClassifier to make the Predictor.Serialize() method work. Hence, the Serialize() methods for those classes are implemented. You can check the references for Serialize() methods of those classes (see Connection.Serialize(), layer.Serialize(), classifier.Serialize()).[9][10] [11]

2. *Deserialization*

After serialization of the Predictor instance, we have to then implement the deserialize() method that can retrieve the objects back from the file and return the Predictor instance. The Predictor.Deserialize() method will deserialize the properties of the Predictor instance which are Connections, CortexLayer, and HtmClassifier objects. The Predictor.Deserialize() method is implemented in the reference [12]. The Deserialize() method will deserialize the Predictor's properties and finally return a Predictor instance. However, the Deserialize() methods for the properties of the Predictor instance need to be implemented in advance. Hence, the Deserialize() methods for Connections, HtmClassifier, Spatial Pooler, Encoder, and Temporal Memory instances are implemented (see Connections.Deserialize(), Encoder.Deserialize(), HtmClassifier.Deserialize(), SpatialPooler.Deserialize(), TemporalMemory.Deserialize()). [13] [14] [15] [16] [17]

On the other hand, we also implemented the Predictor.Load() method which is used to create a stream reader for the Predictor.Deserialize() method. The input argument for the Load() method is the file name where the predictor was saved to. The **Listing 5** shows how the Predictor.Load() method is implemented.

```
    public static T Load<T>(string fileName)
    {
        HtmSerializer.Reset();
        using StreamReader sr = new
        StreamReader(fileName);
        return (T)Deserialize<T>(sr, null);
    }
```

Listing 5: Predictor.Load().

3. *Experiments*

To know whether the serialization is implemented properly, we used the debugger to see how the objects are serialized and deserialized.

As shown in **Fig. 5**, and **Fig. 6**, the Connections object is serialized and deserialized successfully. We received the object after deserialization.

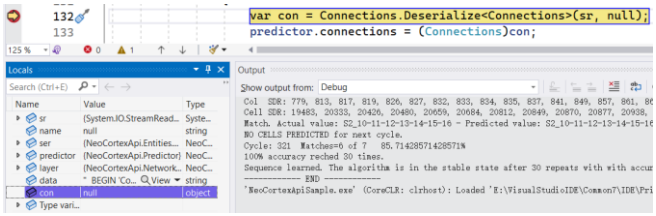

Figure 5 Before Deserialization of Connections

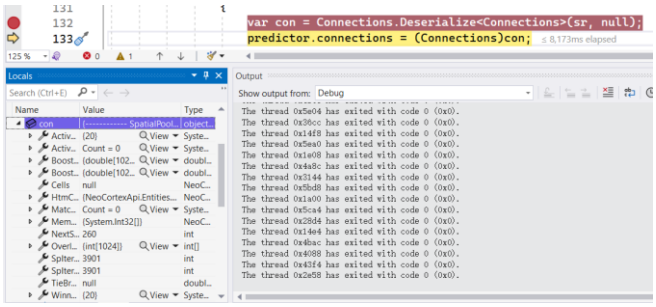

Figure 6 After Deserialization of Connections

Obviously, **Fig. 7** and **Fig. 8** show that serialization of Spatial Pooler is implemented properly since we get the Spatial Pooler object after deserialization.

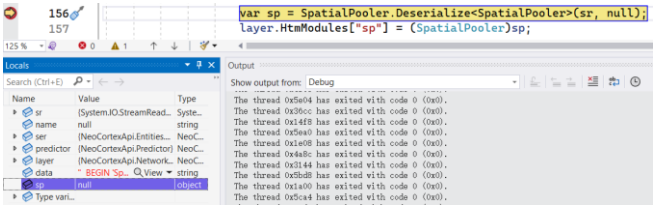

Figure 7 Before Deserialization of Spatial Pooler

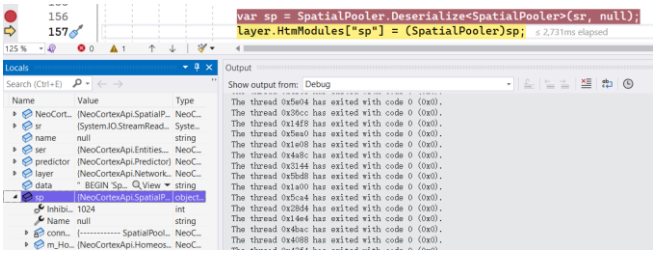

Figure 8 After Deserialization of Spatial Pooler

Fig. 9 and **Fig. 10** show that the serialization method is also working properly for Temporal Memory by receiving the object after deserialization.

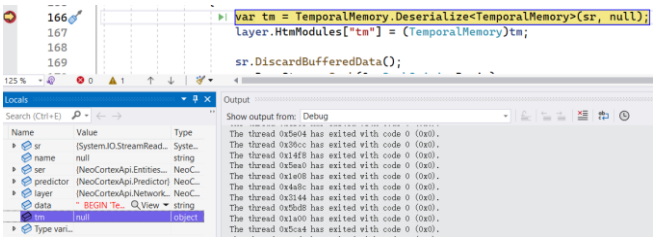

Figure 9 Before Deserialization of Temporal Memory

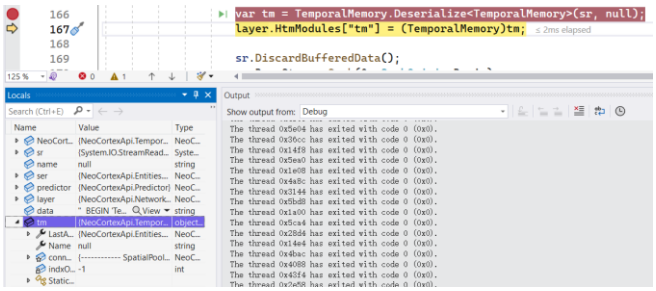

Figure 10 After Deserialization of Temporal Memory

Fig. 11 and **Fig. 12** show that the HtmClassifier object is serialized and deserialized properly by receiving the object after deserialization.

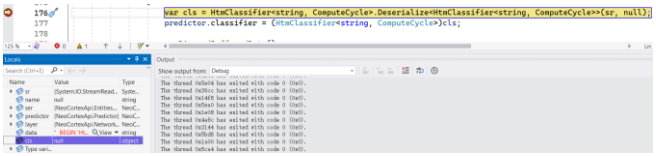

Figure 11 Before Deserialization of HtmClassifier

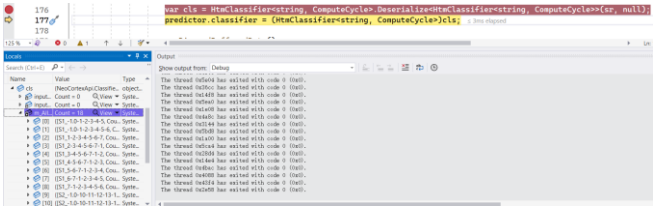

Figure 12 After Deserialization of HtmClassifier

After the Predictor.Save() method is called, the Predictor instance is serialized to a file and saved locally (see **Fig. 13**, **Fig. 14**).

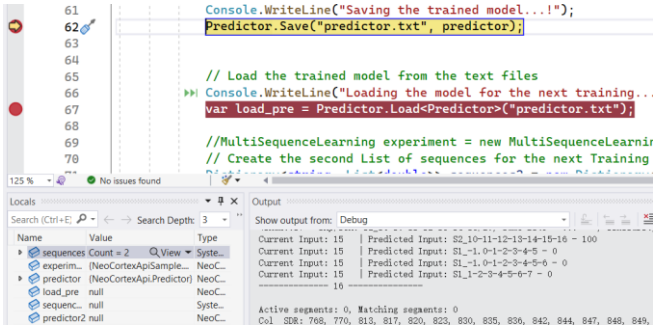

Figure 13 Before Serialization of Predictor

Figure 14 After Serialization of Predictor

As shown from **Fig. 15** that the object load_pre is null before deserialization. However, it becomes a Predictor object after deserialization, and contains three objects of class HtmClassifier, Connections, and CortexLayer.

From the above experiments, we know that serialization methods of classes such as Predictor, Connections, Spatial

Pooler, Temporal Memory and HtmClassifier are implemented properly.

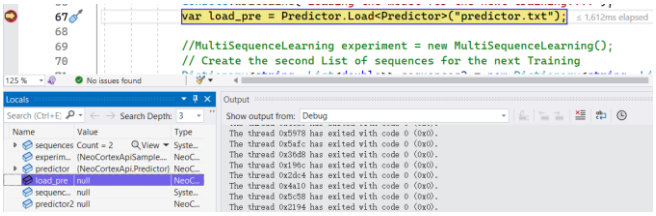

Figure 15 Before Deserialization of Predictor

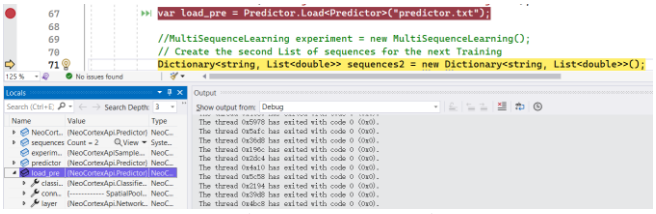

Figure 16 After Deserialization of Predictor

However, to validate the project implementation, predictions were made using both the standard predictor and the serialized predictor. The outcomes of predictions from both predictors were examined and compared, verifying the result is identical or not.

4. MSL experiment and comparison between normal Predictor and Serialized Predictor

Multi Sequence Learning (MSL) experiment demonstrates how to learn two sequences and how to use the prediction mechanism through RunMultiSequenceSerializationExperiment() method. [18] For the training, two sequences S1 and S2, containing scaler values are defined and learned and further used to predict the next element from the sequence as shown in **Listing 6**.

```
private static void
RunMultiSequenceSerializationExperiment()
    {
        Dictionary<string, List<double>> sequences =
        new Dictionary<string, List<double>>();

        sequences.Add("S1", new List<double>(new
        double[] { 0.0, 1.0, 2.0, 3.0, 4.0, 2.0, 5.0,
        }));

        sequences.Add("S2", new List<double>(new
        double[] { 8.0, 1.0, 2.0, 9.0, 10.0, 7.0,
        11.00 }));
```

Listing 6: Input Sequences for MSL Experiment.

For learning of sequences, MultiSequenceLearning.Run() method is called from the MultiSequenceLearning class in Program.cs, which returns the object of Predictor class. This Run() method will returns both the normal predictor and the serialized predictor as shown in **Listing 7**.

```
//The serializedPredictor is the predictor after
serialization and deserialization.

Predictor serializedPredictor;

//The predictor is the normal result of
MultiSequenceLearning. The Run() method will return
not only the normal predictor but also the
serializedPredictor.

var predictor = experiment.Run(sequences, out
serializedPredictor, "predictor");
```

Listing 7: Calling of Run() Method in Program.cs.

The Run() method returns the MultiSequenceLearning.RunExperiment() method which finally returns the instances of Predictor class as shown in **Listing 8**. For testing purposes, we defined two instances of class Predictor i.e., "predictor" and "serializedPredictor". The "predictor" instance acts as a normal predictor which predicts the next element without being serialized and deserialized. However, "serializedPredictor" is the instance of class Predictor which is the result after serialization and deserialization of Predictor.

```
public Predictor Run(Dictionary<string, List<double>>
sequences, out Predictor serializedPredictor, string
fileName)
    {
        .......

        return RunExperiment(inputBits, cfg, encoder,
        sequences, out serializedPredictor,
        fileName);
    }

private Predictor RunExperiment(int inputBits,
HtmConfig cfg, EncoderBase encoder, Dictionary<string,
List<double>> sequences, out Predictor
serializedPredictor, string fileName)
    {
        .......

        // The "predictor" is the instance of class
        Predictor which is result after learning.
        This "predictor" object later on put in the
        argument of Save() method for serialization.

        // The "serializedPredictor" is the instance
        of Predictor class which is the result after
        serialization and deserialization of
        Predictor.

        var predictor = new Predictor(layer1, mem,
        cls);

        //Save() method is callled from Predictor
        Class, which serialize the instance of
        Predictor Class.

        Predictor.Save(predictor, fileName);

        //Load() method is callled from Predictor
        Class, which deserialize the instance of
        Predictor Class.

        serializedPredictor =
        Predictor.Load<Predictor>(fileName);

        return predictor;
    }
```

Listing 8: MultiSequenceLearning.RunExperiment() method.

After learning these two sequences, three sequence lists are defined to check how the prediction works as shown in **Listing 9**.

```
    // These list are used to see how the prediction
works.
    // Predictor is traversing the list element by
element
    // By providing more elements to the prediction,
the predictor delivers more precise result.
    var list1 = new double[] { 1.0, 2.0, 3.0, 4.0,
2.0, 5.0 };
    var list2 = new double[] { 2.0, 3.0, 4.0 };
    var list3 = new double[] { 8.0, 1.0, 2.0 };
```

Listing 9: Sequences to check for prediction.

The Program.PredictNextElement() method uses instance of class Predictor to predict the next element by transversing the given list element by element. The PredictNextElement() method is defined in **Listing 10**.

```
    private static void PredictNextElement(Predictor
predictor, double[] list)
    {
        Debug.WriteLine("----------------------------
        --");

        foreach (var item in list)
        {
            var res = predictor.Predict(item);

            if (res.Count > 0)
            {
                foreach (var pred in res)
                {

                 Debug.WriteLine($"{pred.PredictedInpu
                 t} - {pred.Similarity}");
                 Console.WriteLine($"{pred.PredictedIn
                 put} - {pred.Similarity}");
                }

             var tokens =
             res.First().PredictedInput.Split('_');

             var tokens2 =
             res.First().PredictedInput.Split('-');

             Debug.WriteLine($"Predicted
             Sequence:{tokens[0]}, predicted next
             element {tokens2.Last()}");

            Console.WriteLine($"Predicted Sequence:
            {tokens[0]}, predicted next element
            {tokens2.Last()}\n");

            }

            else
             Debug.WriteLine("Nothing predicted :(");
        }

        Debug.WriteLine("----------------------------
        -");
    }
```

Listing 10: Program.PredictNextElement() method

Now, we compare the prediction output from "predictor" and "serializedPredictor" instance and the result must be same to verify that serialization and deserialization for instance of Predictor class were correct.

As described below in the code of **Listing 11**, the "predictor" and "serializedPredictor were used as inputs for PredictNextElement() method to predict the next elements respectively. Here, the next elements for list2 ({2.0, 3.0, 4.0}) were predicted and the results from both the normal predictor and the serialized predictor were checked and compared.

```
    //The PredictNextElement() method will predict the
next element using the normal predictor and the
serialized predictor.
    //The prediction of both normal predictor and
serialized predictor are checked, and compared.
    predictor.Reset();
    Console.WriteLine("\n\n\t\tPrediction next
elements with normal predictor: \n\n");
    //Prediction with normal predictor
    PredictNextElement(predictor, list2);

    serializedPredictor.Reset();
    Console.WriteLine("\n\n\t\tPrediction next
elements with serialized predictor: \n\n");
    //Prediction with serialized predictor
    PredictNextElement(serializedPredictor, list2);
```

Listing 11: Prediction of next elements for list 2 sequence.

Testing Output: The Listing 12 shows the predicted output, which we got by using the normal predictor and the serialized predictor respectively. As shown in **Listing 12**, the same prediction (5, 4, 2) was made by both the normal predictor and the serialized predictor with the same accuracy.

```
Hello NeocortexApi! Experiment MultiSequenceLearning


Prediction next elements with normal predictor:


S1_0-1-2-3-4-2-5 - 33.33
S2_10-7-11-8-1-2-9 - 33.33
S1_-1.0-0-1-2-3-4 - 0
Predicted Sequence: S1, predicted next element 5

S1_-1.0-0-1-2-3-4 - 100
S1_2-5-0-1-2-3-4 - 100
S1_-1.0-0-1-2-3-4-2 - 0
S1_0-1-2-3-4-2-5 - 0
S1_1-2-3-4-2-5-0 - 0
Predicted Sequence: S1, predicted next element 4

S1_-1.0-0-1-2-3-4-2 - 100
S1_5-0-1-2-3-4-2 - 100
S1_-1.0-0-1-2-3-4 - 0
S1_0-1-2-3-4-2-5 - 0
S1_1-2-3-4-2-5-0 - 0
Predicted Sequence: S1, predicted next element 2



Prediction next elements with serialized predictor:


S1_0-1-2-3-4-2-5 - 33.33
S2_10-7-11-8-1-2-9 - 33.33
S1_-1.0-0-1-2-3-4 - 0
Predicted Sequence: S1, predicted next element 5

S1_-1.0-0-1-2-3-4 - 100
S1_2-5-0-1-2-3-4 - 100
S1_-1.0-0-1-2-3-4-2 - 0
S1_0-1-2-3-4-2-5 - 0
S1_1-2-3-4-2-5-0 - 0
Predicted Sequence: S1, predicted next element 4

S1_-1.0-0-1-2-3-4-2 - 100
S1_5-0-1-2-3-4-2 - 100
S1_-1.0-0-1-2-3-4 - 0
S1_0-1-2-3-4-2-5 - 0
S1_1-2-3-4-2-5-0 - 0
Predicted Sequence: S1, predicted next element 2


C:\SE\source\MySeProject\bin\Debug\net6.0\MySeProject.
exe (process 9356) exited with code 0.
To automatically close the console when debugging
stops, enable Tools->Options->Debugging->Automatically
close the console when debugging stops.
Press any key to close this window . . .
```

Listing 12: Testing Output

III. Results

We have successfully implemented serialization with Temporal Memory for the MultiSequenceLearning project in which the Serialize(), Deserialize(), Save(), and Load() methods in Predictor class are used. In order to make the serialization of the Predictor instance possible, methods for serialization and deserialization of Predictor properties are also implemented which are Connections.Serialize(), Connections.Deserialize(), HtmClassifer.Serialize(), HtmClassifier.Deserialize(), CortexLayer.Serialize(), SpatialPooler.Serialize(), SpatialPooler.Deserialize(), TemporalMemory.Serialize(), TemporalMemory.Deserialize(). By using the debugger, we know that the objects are serialized and deserialized successfully. On the other hand, an MSL example [19] has been made to demonstrate the use of Serialization in the MSL project. In the example, the predictor is trained with the first input sequences and then saved to a file locally using the Predictor.Save() method. After that, it is loaded from the file using the Predictor.Load() method, and then used as input argument for the next training. Moreover, to confirm the project's implementation, predictions were executed using both the standard predictor and the serialized predictor. The results of predictions from both predictors were scrutinized and found to be identical."

IV. Discussion

The Serialization and Deserialization techniques that are used in this project have been presented clearly throughout the paper. The properties of the Predictor class, and the Predictor instance are successfully serialized. Predictor serialization is important for implementation of serialization with Temporal Memory in MSL projects. Example [18] shows that the project is now serializable, and we can store the trained model and use it for the upcoming training. By comparing predictions made using the standard predictor and the serialized predictor, this project proves that the serialization approach appears to be promising and reliable since the objects are serialized and deserialized accurately. The serialization method can replicate the properties in the destination Predictor object. However, it is realized that the implementation is not totally complete as serialization for scalar encoder is still not done yet. Further, research is necessary to identify how to do serialization of scalar encoder with this approach.

V. Acknowledgment

Our group would like to express my deepest thanks to Prof. D. Dobric, my project supervisor, for his continuous support and motivation. We are also thankful for your insightful feedback, and intellectual guidance. We are also grateful to my classmate Aneeta, and her group for sharing their research, honest feedback, and encouragement. Finally, we want to thank the tutors Sabin Bir, Sahith Kumar Singari and Paween for their help and advice.

References

[1] D. Dobric. [Online]. Available: https://github.com/ddobric/neocortexapi/blob/master/source/Documentation/TemporalMemory.md [accessed 05 January 2024].

[2] D. Dobric. [Online]. Available: https://github.com/ddobric/neocortexapi/blob/master/source/Documentation/images/HtmPipeline.png [accessed 05 January 2024].

[3] D. Dobric. [Online]. Available: https://www.numenta.com/assets/pdf/temporal-memory-algorithm/Temporal-Memory-Algorithm-Details.pdf [accessed 05 January 2024].

[4] D. Dobric. [Online]. Available: https://github.com/ddobric/neocortexapi/blob/master/source/Documentation/images/SDR_bursting.png [accessed 05 January 2024].

[5] D. Dobric. [Online]. Available: https://github.com/ddobric/neocortexapi/blob/master/source/Documentation/TemporalMemory.md [accessed 05 January 2024].

[6] [Online]. Available: https://www.developer.com/java/java-serialization-persist-your-objects/ [accessed 05 January 2024].

[7] [Online]. Available: https://www.c-sharpcorner.com/article/serialization-and-deserialization-in-c-sharp/ [accessed 05 January 2024].

[8] [Online]. Available: https://www.section.io/engineering-education/deserialization-in-csharp/#:~:text=Serialization%20in%20C%23%20is%20the,can%20be%20stored%20in%20memory [accessed 25 March 2023].

[9] [Online]. Available: https://github.com/Hungbth2000/tm_msl_serialization/blob/master/source/NeoCortexEntities/Entities/Connections.cs#L1621 [accessed 05 January 2024].

[10] [Online]. Available: https://github.com/Hungbth2000/tm_msl_serialization/blob/master/source/NeoCortexApi/Network/CortexLayer.cs#L166 [accessed 05 January 2024].

[11] [Online]. Available: https://github.com/Hungbth2000/tm_msl_serialization/blob/master/source/NeoCortexApi/Classifiers/HtmClassifier.cs#L734 [accessed 05 January 2024].

[12] [Online]. Available: https://github.com/Hungbth2000/tm_msl_serialization/blob/master/source/NeoCortexApi/Predictor.cs#L100 [accessed 05 January 2024].

[13] [Online]. Available: https://github.com/Hungbth2000/tm_msl_serialization/blob/master/source/NeoCortexEntities/Entities/Connections.cs#L1672 [accessed 05 January 2024].

[14] [Online]. Available: https://github.com/Hungbth2000/tm_msl_serialization/blob/master/source/NeoCortexApi/Encoders/EncoderBase.cs#L342 [accessed 05 January 2024].

[15] [Online]. Available: https://github.com/Hungbth2000/tm_msl_serialization/blob/master/source/NeoCortexApi/Classifiers/HtmClassifier.cs#L754 [accessed 05 January 2024].

[16] [Online]. Available: https://github.com/Hungbth2000/tm_msl_serialization/blob/master/source/NeoCortexApi/SpatialPooler.cs#L1451 [accessed 05 January 2024].

[17] [Online]. Available: https://github.com/Hungbth2000/tm_msl_serialization/blob/master/source/NeoCortexApi/TemporalMemory.cs#L907 [accessed 05 January 2024].

[18] [Online]. Available: https://github.com/Hungbth2000/tm_msl_serialization/blob/master/source/MySeProject/Program.cs#L26 [accessed 05 January 2024].

[19] [Online]. Available: https://github.com/Hungbth2000/tm_msl_serialization/blob/master/source/MySeProject/Program.cs [accessed 05 January 2024].