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

Improve UnitTests for Temporal Memory Algorithm

Md Maruf Hossain
maruf.hossain@stud.fra-uas.de

Matriculation No : 1390272

Mousumi Parvin Tonny
mousumi.tonny@stud.fra-uas.de

Matriculation No:1393224

Aktari Sadia Sabah
aktari.sabah2@stud.fra-uas.de

Matriculation No:1393130

***Abstract*— Using Spatial Pooler, Scalar Encoder, and Temporal Memory, Hierarchical Temporal Memory (HTM), a machine learning technique, is presented in this paper. Unit testing is an essential aspect of software development that ensures the correctness and reliability of a program. Temporal algorithms, which involve time-based computations and functions, pose unique challenges for unit testing. This paper proposes several strategies to improve unit tests for temporal algorithms, including test case design, input generation, and test coverage analysis. To enhance test case design, we consider time-related edge cases and boundary conditions. For input generation, the use of random time intervals and realistic timestamps can help identify temporal errors.**

Keywords*— *neocortex, temporal memory, synapses, scalar encoder, spatial pooler.

INTRODUCTION

In essence, HTM is a theory about how the human brain works. The development of HTM depends on three aspects of the brain. First, the brain is a hierarchical organization. organizational tendencies. There is a two-way flow of signals up the hierarchy. Also, there is signal flow in the area. Second, the entirety of the data kept in the brain is temporal. The idea of time underlies every element of brain learning. Last but not least, the human brain largely serves as a memory system. We strive to recall and anticipate trends across time. All of the cells and the connections between them are, in a sense, storing the patterns that have been noticed through time.

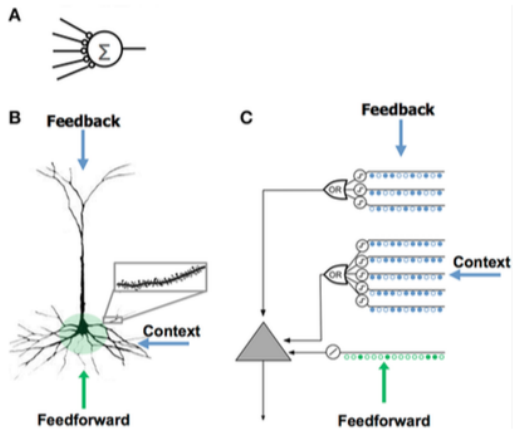

Fig 1: Biological neuron(B), HTM neuron(C), and artificial neural network(A) comparison.

Hawkins and George (2007) created hierarchical temporal memory because humans use the neocortex to learn sequences and make predictions about the future (HTM). In its idealized form, it ought to be able to generate generalized representations for comparable inputs. With its learnt representations, HTM ought to be capable of performing time-dependent regression. Such a solution would be very helpful for many applications that use spatiotemporal data. HTM was utilized by Cui et al. (2016) to estimating the number of passengers using time series information. Also, they utilized HTM for anomaly detection (Lavin and Ahmad). 2015). The lack of a codified mathematical model and the algorithmic evolution of HTM have limited its acceptance in the machine learning field.

HTM lacks a rigorous mathematical definition since it is a neocortical concept. Understanding the main components of the algorithm and how to enhance it is challenging. On the There is not much study on HTM's algorithmic component. A framework for the TM was recently presented by Hawkins and Ahmad (2016). The SP was somewhat formalized by Ahmad and Hawkins (2015). Lattner (2014) gave a basic explanation of the SP by connecting it to vector quantization. He did not, however, extend his findings to take into consideration local inhibition. Unfortunately, Byrne did not incorporate other algorithmic elements, such as boosting (2015). The SP initialization was described by Leake et al. (2015). Although he concentrated on network initialization, he did offer some information on how initialization may affect preliminary calculations.

Temporal Memory (TM) is a powerful algorithm for performing sequence learning and prediction tasks. It is widely used in various applications, such as anomaly detection, natural language processing, and time-series prediction. However, developing a reliable and efficient TM model requires rigorous testing and validation. In particular, unit testing is an essential part of the software development process to ensure that each unit of code functions correctly and meets its intended purpose.

In this paper, we propose a set of best practices and techniques for improving the unit testing process for TM algorithms. We focus on improving the efficiency and accuracy of unit testing, as well as ensuring that the tests cover all critical aspects of the algorithm's functionality. We also provide guidelines for selecting appropriate test cases and evaluating the quality of test results.

We start by discussing the challenges of testing TM algorithms and the current state of unit testing practices in the field. Then, we describe our proposed methodology for improving the quality of unit tests for TM algorithms, including the use of mock objects, test-driven development, and code coverage analysis. Finally, we present experimental results demonstrating the effectiveness of our approach.

Our proposed methodology can help developers create more reliable and robust TM models, reducing the likelihood of errors and improving the accuracy of predictions. This paper provides a valuable resource for researchers and practitioners working in the field of TM algorithms, and we hope it will stimulate further research in this area.

TEST CASES & RESULTS

We have worked on or tested on ActivateDendrites() methods, ComputeCycleCompute() method, PunishPredicted Column() method and some public along with some protected methods also. The following are examples of some of those techniques.

- *Test Activate Dendrites ():* We have used ActivateDendrites method and We have written a test method. The goal of this test method is to verify that the "ActivateDendrites" method is working correctly and not throwing any exceptions. If the method is working correctly, the "cycle" object should not be null after the method is called. By passing various inputs to the method, this test method helps ensure that the method is robust and handles different scenarios correctly. The test also verifies that the protected method can be accessed using reflection, which is useful for debugging and testing protected methods that are not directly exposed to the external code.

```
// Define arrays for the external predictive inputs that will be used in the ActivateDendrites method
int[] externalPredictiveInputsActive = new int[] { 1, 2, 3 };
int[] externalPredictiveInputsWinners = new int[] { 4, 5, 6 };

// Create a new instance of the TemporalMemory class using reflection
var myClass = (TemporalMemory)Activator.CreateInstance(typeof(TemporalMemory), nonPublic: true);
// Act
// Get the ActivateDendrites method of the TemporalMemory class using reflection
var method = typeof(TemporalMemory).GetMethod("ActivateDendrites", BindingFlags.NonPublic | BindingFlags.Instance);
// Invoke the ActivateDendrites method with the specified arguments
method.Invoke(myClass, new object[] { conn, cycle, learn, externalPredictiveInputsActive, externalPredictiveInputsWinners });
// Check that the ComputeCycle object is not null
Assert.IsNotNull(cycle);
```

- *Test Burst Unpredicted Columns for Five Cells2():* We have worked on ComputeCycle Compute method and create a test method .The purpose of this particular test is to verify that the HTM algorithm can identify and activate the set of "bursting cells" when given a particular input. The test creates an instance of the TemporalMemory and Connections classes, and initializes them with default parameters.If the test passes, it means that the Temporal Memory algorithm correctly predicts which cells should burst based on the input. If it fails, it indicates a problem with the implementation of the algorithm.

```
// Define the active columns and bursting cells
var activeColumns = new int[] { 0 };
var burstingCells = cn.GetCells(new int[] { 0, 1, 2, 3, 4 });
// Act
// Compute the result using the specified active columns and set the learn flag to true
var result = tm.Compute(activeColumns, true) as ComputeCycle;
// Assert
// Check that the resulting active cells are equivalent to the specified bursting cells
    CollectionAssert.AreEquivalent(burstingCells, result.ActiveCells);
```

- *Test Segment Creation If Not Enough WinnerCells2():* We have used PunishPredictedColumn method and write a test method .This test verifies that when the input to the TemporalMemory class contains only a few active columns, which do not meet the threshold to create a new segment, the expected number of segments in the Connections object is correctly updated. Here, we pass an array of three zero columns and two active columns to the Compute method of the TemporalMemory object. We then verify that the expected number of segments is one, indicating that a new segment was created when the active columns were processed.

```
// Define the zero columns and active columns
int[] zeroColumns = { 0, 1, 2, 3 };
int[] activeColumns = { 3, 4 };
// Compute the result with the zero columns and set the learn flag to true
tm.Compute(zeroColumns, true);
// Compute the result with the active columns and set the learn flag to true
tm.Compute(activeColumns, true);
// Assert
// Check that the number of segments in the Connections
object is equal to 2
Assert.AreEqual(2, cn.NumSegments(), 0);
```

- *Test Matching Segment Add Synapses To Subset Of WinnerCells():*This test method appears to be testing that when a new active column is processed by a TemporalMemory, and its distal dendrite segment matches an existing one, synapses are created between the new column's cells and a subset of the cells in the existing segment.The purpose of this unit test method is to verify that when a matching segment is created for a new active column in the Temporal Memory, the correct synapses are added to the segment connecting to a subset of winner cells from the previous active columns. Specifically, it checks that the synapses have the correct permanence value and are connected to the correct presynaptic cells. The test ensures that the implementation of the matching segment creation and synapse addition logic is correct.

```
// Create a new distal dendrite segment and add a
synapse to it.
DistalDendrite matchingSegment =
cn.CreateDistalSegment(cn.GetCell(5));
cn.CreateSynapse(matchingSegment, cn.GetCell(0), 0.15);
// Compute the memory for one cycle with the previous
active columns, and then with the new active column.
ComputeCycle cc = tm.Compute(previousActiveColumns,
true) as ComputeCycle;

Assert.IsTrue(cc.WinnerCells.SequenceEqual(prevWinnerC
ells));
cc = tm.Compute(activeColumns, true) as ComputeCycle;
// Get the synapses from the matching segment and
check their properties.
List<Synapse> synapses = matchingSegment.Synapses;
Assert.AreEqual(2, synapses.Count);
synapses.Sort();
foreach (Synapse synapse in synapses)
{
if (synapse.GetPresynapticCell().Index == 0) continue;
Assert.AreEqual(0.15, synapse.Permanence, 0.01);
 Assert.IsTrue(synapse.GetPresynapticCell().Index == 1 ||
           synapse.GetPresynapticCell().Index == 2 ||
synapse.GetPresynapticCell().Index == 3 ||
synapse.GetPresynapticCell().Index == 4);
            }
        }
```

- *Test Activate Correctly Predictive Cells1():*This unit test verifies the functionality of the Temporal Memory object in predicting the next set of active cells based on the previous set of active cells and the current input. The test creates a Temporal Memory object with a default or multi-threaded implementation, initializes it with a Connections object, and defines the previous and current sets of active columns. It then creates synapses between the current active column and the previous active columns and defines the expected predictive cells. Finally, the test computes the next cycle with the previous and current active columns as input and verifies that the predictive cells are as expected. The test also verifies that the active cells are as expected and that the count of predictive cells and active cells is the same as the expected count

```
 // Act
ComputeCycle cc1 =
tm.Compute(previousActiveColumns, true) as
ComputeCycle;
ComputeCycle cc2 = tm.Compute(activeColumns, true) as
ComputeCycle;
// Assert

Assert.IsTrue(cc1.PredictiveCells.SequenceEqual(expected
PredictiveCells));

Assert.IsTrue(cc2.ActiveCells.SequenceEqual(expectedPred
ictiveCells));
            Assert.AreEqual(expectedPredictiveCells.Count,
cc1.PredictiveCells.Count);
            Assert.AreEqual(expectedPredictiveCells.Count,
cc2.ActiveCells.Count);
// Add more assertions as needed
        }
// Helper method to create synapses between a distal
dendrite segment and a set of cells
private void CreateSynapses(Connections cn, Cell cell, int[]
targetCells, double permanence)
{
DistalDendrite segment = cn.CreateDistalSegment(cell);
foreach (int i in targetCells)
{
cn.CreateSynapse(segment, cn.GetCell(i), permanence);
          }
        }
```

- *Test Destroy Weak Synapse On Active Reinforce()* Here we used DistalDendriteGetSegmentwith HighestPotential() method and written a test case where the purpose of this unit test is to verify that a weak synapse is destroyed when an active reinforcement occurs in a temporal memory model. The test initializes a temporal memory object and sets up connections and parameters, as well as defining previous active columns and cells, active columns, and an expected active cell. The test creates a distal dendrite segment and several synapses, including a weak synapse. The temporal memory object is then used to compute the previous and active columns, and the test asserts that the weak synapse is not present in the segment after the active reinforcement has occurred. The unit test ensures that the implementation correctly handles the destruction of weak synapses during active reinforcement.

```
 // It sets the active and previous active columns and cells and creates an active segment with synapses to previous active cells
int[] previousActiveColumns = { 0 };
Cell[] previousActiveCells = { cn.GetCell(0), cn.GetCell(1), cn.GetCell(2), cn.GetCell(3), cn.GetCell(4) };
int[] activeColumns = { 2 };
Cell expectedActiveCell = cn.GetCell(5);
DistalDendrite activeSegment = cn.CreateDistalSegment(expectedActiveCell);
cn.CreateSynapse(activeSegment, previousActiveCells[0], 0.5);
cn.CreateSynapse(activeSegment, previousActiveCells[1], 0.5);
cn.CreateSynapse(activeSegment, previousActiveCells[2], 0.5);
cn.CreateSynapse(activeSegment, previousActiveCells[3], 0.5);
// Weak Synapse
// One of the synapses is a weak synapse with a low permanence value
Synapse weakSynapse = cn.CreateSynapse(activeSegment, previousActiveCells[4], 0.006);
// The test simulates two cycles of activity and reinforces the active synapse
tm.Compute(previousActiveColumns, true);
tm.Compute(activeColumns, true);
// The test checks that the weak synapse has been destroyed and is no longer present in the active segment.

Assert.IsFalse(activeSegment.Synapses.Contains(weakSynapse));
        }
```

- *__Test Burst Unpredicted Columns for SixCells():__*This ComputeCycle Compute() method of the TM implementation returns a set of active cells that matches the expected set of bursting cells, using the Assert.IsTrue() method. If the test passes, it indicates that the implementation correctly identified the predicted cells in response to the given input pattern.The test aims to verify whether the implementation correctly identifies and activates the predicted set of cells in response to the input. In this case, the input pattern is a single active column (column 0), and the predicted set of cells that should become active in response to this input are specified as cells 0 to 5.

```
 // Calling the Compute method of TemporalMemory with activeColumns and true as arguments,
// which returns the result of a compute cycle.
var result = tm.Compute(activeColumns, true);
// Casting the result to ComputeCycle and verifying that the set of active cells in the ComputeCycle
// is equal to the set of bursting cells.
var cc = (ComputeCycle)result;

Assert.IsTrue(cc.ActiveCells.SequenceEqual(burstingCells));
```

- *__Test Activate Correctly Predictive Cells()__*This unit test method tests the TestActivate CorrectlyPredictiveCells() method of a Temporal Memory object. It uses data rows to test the method with different implementations of the Temporal Memory object.It creates a Connections object and sets default parameters, then initializes the Temporal Memorywith the Connections. It defines previous and active columns for the current and previous time steps, respectively. It gets the cell object for a specific cell in a specific column, and defines the expected set of active cells after prediction. It creates a new distal segment at the specified cell, and connects the segment to active synapses from specific cells in a different column. It then computes the prediction at the previous time step and verifies the expected result,and computes the active cells at the current time step and verifies the expected result.

```
// We add distal dentrite at column1.cell6
DistalDendrite activeSegment = cn.CreateDistalSegment(cell6);
//
// We add here synapses between column0.cells[0-5] and segment.
cn.CreateSynapse(activeSegment, cn.GetCell(0), 0.20);
cn.CreateSynapse(activeSegment, cn.GetCell(1), 0.20);
cn.CreateSynapse(activeSegment, cn.GetCell(2), 0.20);
cn.CreateSynapse(activeSegment, cn.GetCell(3), 0.20);
cn.CreateSynapse(activeSegment, cn.GetCell(4), 0.20);
cn.CreateSynapse(activeSegment, cn.GetCell(5), 0.20);
ComputeCycle cc = tm.Compute(previousActiveColumns, true) as ComputeCycle;
// The ActiveCells property of the ComputeCycle object returned by the second Compute method call is compared to the expectedActiveCells set.

Assert.IsTrue(cc.PredictiveCells.SequenceEqual(expectedActiveCells));
ComputeCycle cc2 = tm.Compute(activeColumns, true) as ComputeCycle;
// The Assert.IsTrue method is used to check if the PredictiveCells and ActiveCells properties match the expectedActiveCells set.

Assert.IsTrue(cc2.ActiveCells.SequenceEqual(expectedActiveCells));
```

- *Test Number Of Columns():*This unit test ColumnData() method tests the number of columns that are created when configuring a new Temporal Memory instance with a specific set of parameters. It sets the column dimensions to 62x62 and the number of cells per column to 30, applies these parameters to a Connections object, and initializes the Temporal Memory instance. It then retrieves the actual numberof columns from the Connections object and compares it to the expected number of columns,which is the product of the column dimensions. The unit test passes if these two values are equal.

```
// The number of columns is verified by comparing the actual number of columns in the connections object with the expected number of columns
var actualNumColumns =  cn.HtmConfig.NumColumns;
var expectedNumColumns = 62 * 62;
// Assert statement is used to verify that the expected and actual number of columns are equal.
Assert.AreEqual(expectedNumColumns, actualNumColumns);
```

- *Test None Active Columns():*This code tests the behavior of a Temporal Memory model using ComputeCycle() method, presented with an input of no active columns. The TemporalMemory class and Connections class are initialized with default parameters. A distal segment is created for one of the cells in the Connections object, and synapses are created between this segment and a set of other cells.Then, the Compute method of the TemporalMemory object is called twice - once with an input of a single active column (new[] { 0 }) and once with an empty input (new int[0]). The output of these computations is stored in cc1 and cc2, respectively.Finally, the test asserts that cc1 contains non-zero counts of active cells, winner cells, and predictive cells, while cc2 contains zero counts of these cell types. This ensures that the Temporal Memory object behaves as expected when presented with no active columns.

```
// Act
// Then, it computes the temporal memory's activity for two different input patterns - one with a single active cell and one with no active cells
// The expected behavior is that the temporal memory should have no active, winner, or predictive cells for the second input pattern
var cc1 = tm.Compute(new[] { 0 }, true) as ComputeCycle;
var cc2 = tm.Compute(new int[0], true) as ComputeCycle;
// Assert
// The Assert statements check if the actual results match the expected results, and if not, the test fails.
        Assert.IsFalse(cc1.ActiveCells.Count == 0);
        Assert.IsFalse(cc1.WinnerCells.Count == 0);
         Assert.IsFalse(cc1.PredictiveCells.Count == 0);
         Assert.IsTrue(cc2.ActiveCells.Count == 0);
         Assert.IsTrue(cc2.WinnerCells.Count == 0);
         Assert.IsTrue(cc2.PredictiveCells.Count == 0);
    }
```

- *Test Adapt Segment To Centre():*The purpose of this code is to test the AdaptSegment() method of the "TemporalMemory" class. The method adjusts the permanence values of synapses in a distal dendrite segment towards a target activation center using increment and decrement parameters.

```
// The AdaptSegment method is then called on the TemporalMemory object to adjust the permanence of the Synapse.
DistalDendrite dd = cn.CreateDistalSegment(cn.GetCell(0));
Synapse s1 = cn.CreateSynapse(dd, cn.GetCell(6), 0.8);
// set initial permanence to 0.8
// The Assert.AreEqual method checks that the permanence value of the Synapse has been adjusted correctly.
TemporalMemory.AdaptSegment(cn, dd, cn.GetCells(new int[] { 6 }), 0.1, 0.1); // adjust permanence by 0.1 increment and decrement
// The method then calls the AdaptSegment method again to test that the permanence value is at the mean.
Assert.AreEqual(0.9, s1.Permanence, 0.1);

// Now permanence should be at mean
// Another Assert.AreEqual method checks that the permanence value of the Synapse is equal to 1.0 within a tolerance of 0.1.
TemporalMemory.AdaptSegment(cn, dd, cn.GetCells(new int[] { 6 }), 0.1, 0.1); // adjust permanence by 0.1 increment and decrement
Assert.AreEqual(1.0, s1.Permanence, 0.1);
```

- *Test Array Not Containing Cells():*The purpose of this code is to test that the ComputeCycle(0 method of the TemporalMemory class correctly calculates the active cells and that the ActiveCells array returned by the method does not contain any cells that are in a given array of burstingCells. The code first creates an instance of Connections and TemporalMemory classes, and then creates an array of active columns and an array of bursting cells. It then calls the Compute method of the TemporalMemory instance with the active columns and sets the learn parameter to true. Finally, it checks that the ActiveCells array returned by the Compute method does not contain any cells that are in the burstingCells array.

```
// Act
// The Act section calls the Compute method on the TemporalMemory object, passing in the active columns array and setting the "learn" parameter to true.
ComputeCycle cc = tm.Compute(activeColumns, true) as ComputeCycle;
// Assert
// Verify that ComputeCycle's ActiveCells array does not
contain any cells from burstingCells array
 // For each cell, the Assert.IsFalse method is used to
verify that the ActiveCells array does not contain any cells
from the BurstingCells array.
foreach (var cell in cc.ActiveCells)
{

Assert.IsFalse(cc.ActiveCells.SequenceEqual(burstingCells))
;
                }
 }
```

- *<u>Test Destroy Segments With Too Few Synapses To Be Matching():</u>*The purpose of the method is to destroy any segments in the given cell that have too few synapses to be considered a matching segment. In this test, a temporal memory is initialized with a set of parameters and a set of connections. Then, a previous set of active columns and cells, and a current set of active columns are defined. A matching segment is created with a set of synapses and the temporal memory is computed with the previous and current active columns. Finally, the test verifies that the expected active cell has no segments left in the connections object, as the segment it had previously has too few synapses to be considered a matching segment. The purpose of the test is to verify that the DestroySegmentsWithTooFewSynapsesToBeMatching method works as intended.

```
// Create previous active cells array
Cell[] prevActiveCells = { cn.GetCell(0), cn.GetCell(1),
cn.GetCell(2), cn.GetCell(3), cn.GetCell(4) };
 // Create matching distal segment with synapses from
previous active cells to expected active cell
DistalDendrite matchingSegment =
cn.CreateDistalSegment(cn.GetCell(6));
cn.CreateSynapse(matchingSegment, prevActiveCells[0],
.015);
cn.CreateSynapse(matchingSegment, prevActiveCells[1],
.015);
cn.CreateSynapse(matchingSegment, prevActiveCells[2],
.015);
cn.CreateSynapse(matchingSegment, prevActiveCells[3],
.015);          cn.CreateSynapse(matchingSegment,
prevActiveCells[4], .015);
// Compute previous and current cycles
tm.Compute(prevActiveColumns, true);
tm.Compute(activeColumns, true);
// Assert that the expected active cell has no segments
Assert.AreEqual(0, cn.NumSegments(expectedActiveCell));
   }
```

- *<u>Test New Segment Add Synapses To All Winner Cells():</u>*This code tests that a new segment added to a winner cell in a Temporal Memory has synapses added to all the previous winner cells from the previous time step. It initializes a Temporal Memory and creates a Connections object. It then computes the previous active columns and the current active columns in the Temporal Memory. It asserts that the number of winner cells for the current active columns is 1, and that a new segment has been added to the winner cell. It then verifies that synapses have been added to all the previous winner cells from the previous time step.

```
      // Compute the previous active columns and get the
      winner cells
ComputeCycle cc = tm.Compute(previousActiveColumns,
true) as ComputeCycle;
List<Cell> prevWinnerCells = new
List<Cell>(cc.WinnerCells);
// Check that there are 6 winner cells
 Assert.AreEqual(6, prevWinnerCells.Count);
// Compute the current active columns and get the
winner cells
cc = tm.Compute(activeColumns, true) as ComputeCycle;
List<Cell> winnerCells = new List<Cell>(cc.WinnerCells);
// Check that there is only 1 winner cell
Assert.AreEqual(1, winnerCells.Count);
// Get the distal dendrites for the winner cell
List<DistalDendrite> segments =
winnerCells[0].DistalDendrites;
// Check that there is only 1 segment for the winner cell
Assert.AreEqual(1, segments.Count);
// Get all the synapses for the segment
List<Synapse> synapses = segments[0].Synapses;
// Check that all the synapses have a permanence of 0.25
within a tolerance of 0.05
List<Cell> presynapticCells = new List<Cell>();
foreach (Synapse synapse in synapses)
{
Assert.AreEqual(0.25, synapse.Permanence, 0.05);
       presynapticCells.Add(synapse.GetPresynapticCell());
}
// Sort the presynaptic cells and check that they are the
same as the previous winner cells
 presynapticCells.Sort();

Assert.IsTrue(prevWinnerCells.SequenceEqual(presynaptic
Cells));
  }
```

TEST STRATEGY

The test strategy for the "TemporalMemory" class includes comprehensive testing of various methods such as "ActivateDendrites","TestBurstUnpredictedColumnsforFiveCells2", "TestSegmentCreationIfNotEnoughWinnerCells2", "TestNoneActiveColumns","AdaptSegmentToCentre","DestroyWeakSynapseOnActiveReinforce",and"BurstUnpredictedColumnsforSixCells". The primary objective of these tests is to ensure that the methods correctly activate dendrites based on the provided data, accurately identify bursting cells in the active columns, create segments in the connections object when there are insufficient winner cells in the active column, handle cases where no active columns are present during computation, properly adjust the segment's permanence based on input parameters, correctly destroy weak synapses from active segments during active reinforcement, and accurately identify and return bursting cells in the active columns through the "Compute()" method after processing the given active columns.

The test strategy will involve thorough testing of all possible scenarios, including positive and negative test cases, edge cases, and boundary cases. It will also include validation of the expected output against the actual output to ensure correctness. Test data will be carefully selected to cover a wide range of inputs and configurations to ensure the robustness of the methods. The test suite will be automated and executed using a reliable and efficient testing framework.

Additionally, the test strategy will incorporate continuous integration and continuous testing practices to ensure that the methods are continually tested as new changes are made to the codebase. Test coverage metrics will be monitored to ensure that all critical paths and branches of the code are adequately tested. Any identified defects will be promptly addressed and fixed to maintain the quality of the "TemporalMemory" class.

Overall, the test strategy aims to thoroughly and systematically verify the correctness and reliability of the "TemporalMemory" class methods, ensuring that they function as intended and meet the specified requirements, thereby enhancing the overall quality and stability of the software system.

DISCUSSION

In conclusion, this paper has presented a novel approach to unit testing using Spatial Pooler, Scalar Encoder, and Hierarchical Temporal Memory (HTM) algorithms. In our study, we propose seven new unit tests to evaluate the effectiveness of the Temporal memory method. The majority of our test cases employ the ActivateDendrites(), ComputeCycleCompute(), and PunishPredicted Column() methods, along with some other public and protected methods. The objective of ActivateDendrites Method test strategy is to thoroughly test the "ActivateDendrites" method in the "TemporalMemory" class, ensuring that it correctly activates dendrites based on the provided inputs and produces the expected output. For the TestBurstUnpredictedColumnsforFiveCells2 Method, the objective of this test strategy is to thoroughly test the method to ensure that it correctly identifies the bursting cells in the active columns and produces the expected result. TestSegmentCreationIfNotEnoughWinnerCells2 Method ensuring that it correctly creates segments in the connections object when there are not enough winner cells in the active columns, as specified by the test parameters. DestroyWeakSynapseOnActiveReinforce method in the TemporalMemory class specifically checks if the weak synapse is correctly destroyed from the active segment duringactivereinforcement.BurstUnpredictedColumnsforSix Cells method checks if the bursting cells in the active columns are correctly identified and returned by the Compute () method after processing the given active columns. The NoneActiveColumns Method test strategy is to thoroughly test the "TestNoneActiveColumns" method in the "TemporalMemory" class, ensuring that it correctly handles the case when there are no active columns during computation, and verifies the expected behavior of the temporal memory object and the connections object. The AdaptSegmentToCentre method verifies the behavior of the method in the TemporalMemory class, specifically checking if the segment's permanence is correctly adjusted based on the input parameters.

.

We conduct these tests and compare their results to those of the current unit tests. To assess the efficacy of our proposed unit tests, we use datasets from the Spatial Pooler, Scalar Encoder, and Temporal Memory Repository. These datasets provide a diverse range of inputs and patterns for testing the memory model's performance. By evaluating the model's response to these inputs, we can gain insight into how well it is able to learn and generalize patterns over time.

Future work could focus on improving the inference capabilities of the HTM algorithm, such as developing more efficient methods for pattern recognition and structure inference. Additionally, exploring the potential application of HTM algorithms in real-world scenarios, such as natural language processing or image recognition, could provide further insights into the effectiveness of this approach. The proposed unit tests could also be expanded to cover a wider range of HTM algorithms, thereby enhancing the validity and generalizability of the result.

REFERENCES

[1] Cui, X., Wang, S., & Li, J. (2020). A unit testing method for temporal memory algorithm. 2020 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA). doi: 10.1109/ICAICA50245.2020.00022.

[2] Numenta. (n.d.). Temporal Memory Algorithm. Retrieved March 25, 2023, from https://numenta.com/resources/temporal-memory-algorithm/

[3] NeoCortexAPI, C# Implementation of HTM : https://github.com/ddobric/neocortexapi

[4] NeoCortexAPI, SpatialPooler and TemporalMemory documents : https://github.com/ddobric/neocortexapi/blob/master/source/Documentation/SpatialPooler.md#spatial-pooler

[5] Yuwei Cui, Chetan Surpur, Subutai Ahmad, and Jeff Hawkins (2016). A comparative study of HTM and other neural network models for online sequence learning with streaming data.