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

Improve UnitTests for Temporal Memory Algorithm

Name: Syed Mostain Ahmed
Email: syed.ahmed@stud.fra-uas.de

Name: Farjana Akter
Email: farjana.akter@stud.fra-uas.de

Name: Md. Shamsir Doha
Email: mohammad.doha@stud.fra-uas.de

***Abstract*—Temporal memory algorithms have gained popularity as a promising approach for modeling temporal sequences in machine learning. This project aims to improve the unit test for the given temporal memory algorithm, which is based on the principles of the cortical column and the neocortex. The algorithm uses a sparse distributed representation of data and incorporates temporal context to predict future values in a sequence. We implemented improvements to the existing unit test, including the addition of more test cases with varying complexity and the implementation of cross-validation techniques for better evaluation of the algorithm's performance. We also optimized the implementation of the algorithm for improved efficiency and scalability.**

Keywords— Temporal memory algorithm, Cortical column, Neocortex, Sparse distributed representation

I. Introduction

Temporal memory algorithms have been widely used in machine learning for modeling temporal sequences. These algorithms are inspired by the principles of the cortical column and the neocortex [1][2], which are responsible for processing sensory information and storing long-term memories in the brain. The Temporal Memory algorithm is a well-known implementation of these principles, which has been used in various applications such as natural language processing, anomaly detection, and stock price prediction.

However, accurately evaluating the performance of the Temporal Memory algorithm can be challenging, especially when dealing with complex and noisy data. Therefore, improving the unit test for this algorithm is crucial for ensuring its accuracy and reliability. In this project, we propose several improvements to the existing unit test for the Temporal Memory algorithm. These improvements include the creation, removal, and update of synapses in distal segments, growth of new dendrite segments, activation of cells in columns, and detection/handling of duplicate active columns.

We also implemented learning and recalling patterns of sequences with different sparsity rates and the ability to initialize Temporal Memory with custom parameters such as the number of cells per column and number of column dimensions. Additionally, we adapted segments and increased the permanence of active synapses, limited the number of active cells per column, and retrieved winner cells from Temporal Memory Compute. Furthermore, we implemented least used cell selection and correct initialization of Connections object and used different parameters for existing unit tests to reinforce testing.

Overall, our project aims to enhance the reliability and accuracy of the unit test for the Temporal Memory algorithm, enabling more accurate evaluation of its performance. This improvement can help advance the development of more robust and reliable algorithms for modeling temporal sequences, benefiting various domains and applications.

II. Methods

In this section, we describe the approach we followed to evaluate the performance of our Temporal Memory algorithm. Our objective was to test the algorithm's ability to learn and predict spatio-temporal patterns in a dataset. To achieve this, we performed a series of experiments using unit tests, which allowed us to systematically evaluate the algorithm's behavior under different conditions. We first describe the setup of our experiments, followed by a detailed explanation of the unit tests we designed and the results obtained.

- *Testing New Segment Growth when multiple matching segments found* [3]: The purpose of this test is to verify the growth of a new dendrite segment when multiple matching segments are found. The TemporalMemory object is initialized, and a set of default parameters are applied to create a Connections object. The TemporalMemory is then initialized with the Connections object. Next, a set of active columns and cells are created, and multiple matching segments are created in the Connections object for the first active cell. The TemporalMemory is then instructed to compute based on the active columns, and the test asserts that a new segment is grown for the active cell with two synapses. This test is important because it verifies the ability of the TemporalMemory to dynamically adapt to changing input patterns and grow new dendrite segments when necessary.

- *Testing value of Synapse Permanence Updating When Matching Segments are Found [4]*: The purpose of the Test Synapse Permanence Update When Matching Segments Found unit test is to verify that the permanence of synapses in a matching dendritic segment are updated when the segment is activated. Firstly, we initialize the TemporalMemory and Connections object. The Parameters object is used to set the permanence decrement parameter, which is applied to the Connections object. In the test, two sets of columns and cells are used: the previous active columns and cells, and the current active columns and cells. Two matching segments are created in the Connections object, each containing synapses to the previous active cells and a synapse to a different cell. The previous active columns are then computed using the TemporalMemory object, followed by the current active columns. The test then verifies that the synapse permanence of the matching synapses is updated correctly. In a nutshell, this unit test verifies that the TemporalMemory algorithm can update the permanence of synapses in matching dendritic segments correctly.

- *Testing TemporalMemory Algorithms initialization*: TestCellsPerColumn [5], TestCustomDimensionsAndCells [6], and TestColumnDimensions [7] are unit tests written to test the initialization of the TemporalMemory class with custom parameters. The first test checks if the class initializes correctly with a custom number of cells per column, while the second test checks if it initializes correctly with custom column dimensions and cells per column. The third test checks if the class initializes correctly with a custom number of column dimensions. To perform the tests, the TemporalMemory class is initialized with a Connections object and the Parameters object, which contains the default parameters for the temporal memory algorithm. The custom parameters are then set using the Set() method of the Parameters object and applied to the Connections object using the apply() method. The TemporalMemory class is then initialized using the initialized Connections object. To check if the initialization was successful, the number of cells in each column is counted and compared to the expected value using the Assert.AreEqual() method. If the number of cells is equal to the expected value, the initialization is considered successful. These tests help to ensure that the TemporalMemory class is properly initialized with the desired custom parameters, which is necessary for the proper functioning of the temporal memory algorithm. Note that these are extensions of original existing test methods [8] [9].

- *Testing Recycle Least Recently Active Segment To Make Room For New Segment* [10]: We enhanced this existing test [11] by giving different various data by DataRow attribute. This tests the behavior of the Temporal Memory algorithm in recycling the least recently active segment to make room for a new segment. The test method is parameterized with different sets of previously active and currently active columns to test the algorithm's behavior in different scenarios. The Temporal Memory object is initialized with a set of default parameters, and the connections are set up with a fixed number of cells per column and a maximum number of segments per cell. The test method then simulates previous and current sets of active columns using the Temporal Memory object's Compute method. The test asserts that the least recently active segment is replaced by the new segment after exceeding the maximum number of segments per cell. The test verifies that the new segment's synapses are disjoint with the replaced segment's synapses by checking that the presynaptic cells of the synapses in the old and new segments are not shared. Overall, this test method ensures that the Temporal Memory algorithm's behavior in segment recycling is consistent and operates correctly in different scenarios.

- *Testing New Segment Add Synapses To All Winner Cells* [12]: This is also an existing test [13] enhanced by us to test it against different dataset. This test verifies the behavior of the algorithm when new segments are added to winner cells. The test is parameterized with the number of previously active columns and the number of currently active columns. The method is initialized as usual. The method creates two arrays, previousActiveColumns and activeColumns, which represent the columns that were previously active and the currently active columns, respectively. The method then computes the TM algorithm on the previousActiveColumns and activeColumns arrays and retrieves the winner cells from the compute cycle. The method asserts that the number of previous winner cells matches the number of previously active columns and that the number of current winner cells matches the number of currently active columns. The method then retrieves the distal dendrites and synapses for the first winner cell in the current winner cells list. The method asserts that the number of segments is 1 and that the number of synapses is 4, which is the default value for the maximum number of new synapses per segment. Finally, the method retrieves the presynaptic cells for each synapse and sorts them in ascending order. The method asserts that the presynaptic cells match the previous winner cells in the same order, which verifies that the new segment has added synapses to all of the previous winner cells.

- *Testing Destroy Weak Synapse On Wrong Prediction* [14]: This is one of the existing tests [15] we modified with different datasets to ensure the tests true potential in testing the feature of Temporal Memory algorithms

behaviour in destroying weak synapses. Our modified test checks if weak synapses are correctly destroyed when an incorrect prediction is made with different datasets. The method takes in a double value for the permanence of the weak synapse and creates a TemporalMemory object along with its required connections and parameters. It then sets up a previous active column, a set of previous active cells, and an active column with an expected active cell. A distal dendrite segment is created with synapses connected to the previous active cells and a weak synapse connected to the fourth previous active cell with the specified permanence. The method then calls the Compute method of the TemporalMemory object with the previous active column and the active column as input. After the computation, the method asserts that the distal dendrite segment has correctly destroyed the weak synapse, leaving only the strong synapses. The unit test method is executed with various values of weakSynapsePermanence using the [DataRow] attribute. This allows for multiple test cases to be executed in a single test method.

- *Test Adding Segment To Cell With Fewest Segments* [16]*: This* existing test [17] was previously implemented with only one set of data but we now modified it to run with three different datasets. The purpose of the test is to verify the behavior of the Temporal Memory algorithm under various conditions. The test checks if adding a new distal segment to the cell with the fewest segments grows the segment on the correct cell. The test is repeated 100 times with different random seeds to ensure the behavior is consistent across multiple runs. The test uses a mock Connections object and sets up a Temporal Memory object with default parameters. It then creates an array of previous active columns and cells, an array of currently active columns, and an array of non-matching cells. Two new distal segments are created on non-matching cells and connected to two of the previous active cells. The Temporal Memory algorithm is then run, and the number of segments, number of segments on specific cells, and number of synapses on each segment are checked for correctness. The test then checks if the segment was grown on the cell with the fewest segments and ensures that the correct columns are activated. The test utilizes the DataRow attribute to test the method with different input data, in the form of weak synapse permanence values or random seeds. Finally, the test checks whether the distal segment grew on both cell 1 and cell 2 at least once during the 100 runs.

- *Test Adapt Segment To Max* [18]*:* We modified the existing TestAdaptSegmentToMax [19] to test the ability of the TemporalMemory class to adapt the permanence value of a synapse in a distal dendrite segment to the maximum value specified in the HTM configuration parameters with different values with the help of [DataRow] attribute. The test is performed with different initial permanence values of the synapse, and the expected permanence value after adaptation is also specified in the test. The test method creates a new instance of the TemporalMemory and Connections classes, initializes them with default parameters, and creates a new distal dendrite segment with a synapse connecting it to a cell. The AdaptSegment method of the TemporalMemory class is then called with the specified parameters to adapt the permanence value of the synapse to the maximum value. The test method asserts that the permanence value of the synapse is equal to the expected value within a tolerance of 0.1. The test is repeated with the same segment and cell, and the AdaptSegment method is called again to ensure that the permanence value remains at the maximum value. The test method again asserts that the permanence value of the synapse is equal to the expected value within a tolerance of 0.1. So, this test method verifies that the TemporalMemory class is able to correctly adapt the permanence value of a synapse to the maximum value specified in the HTM configuration parameters.

- *Testing Destroy Segments With Too Few Synapses To Be Matching* [20]: We have modified this existing method [21]. The method tests the ability of the TemporalMemory class to destroy a segment with too few synapses to be considered a match. The modified test method is parameterized using the DataRow attribute to retest the method with different sets of input parameters. The test method sets up a new instance of the TemporalMemory class and its related objects such as Connections and Parameters. The test method then creates a new DistalDendrite object with a matching segment and synapses using the Connection object. The test method then computes the active columns and the expected active cell using the TemporalMemory object. Finally, the test method asserts that the number of segments associated with the expected active cell matches the expected number of segments. The purpose of this test method is to ensure that the TemporalMemory class is correctly identifying segments with too few synapses and destroying them. By parameterizing the test method with different input values, the test method verifies that the TemporalMemory class can correctly handle a variety of different scenarios.

- *Test Punish Matching Segments In Inactive Columns* [22]: This existing test method [23] is modified by us to take 13 input parameters dynamically. The purpose of the unit test is to verify the behavior of a Temporal Memory algorithm under various input conditions. It takes various permanence values and expected permanence values for a set of synapses. The unit test creates an instance of the TemporalMemory class and initializes it with default parameters. It then creates a set of active

and inactive cells, as well as two distal dendrites with multiple synapses. The algorithm is run on these inputs and the permanence values of the synapses are compared against the expected values. The [DataRow] attribute is used to pass in multiple sets of input data to the same unit test, allowing it to be retested with different data.

- *Testing High Sparsity Sequence Learning And Recall* [24]: This test is designed to verify if the Temporal Memory algorithm can successfully learn and recall patterns of sequences with a high sparsity rate. The test first initializes a Temporal Memory object and a Connections object, and applies default parameters to the Connections object. Then, it sets the column dimensions to be 64 and initializes the Temporal Memory object with the Connections object. The test creates two sequences, each with a different set of active columns. It then computes the Temporal Memory algorithm for each of these sequences to train the model. Next, the test recalls the first sequence and the second sequence using the Temporal Memory algorithm with the "compute" method, and checks if all the active cells in the second sequence are also active in the first sequence. This test is important because it validates the ability of the Temporal Memory algorithm to learn and recall patterns of sequences with a high sparsity rate, which is a crucial capability for machine learning algorithms that are designed to process high-dimensional data, such as images or speech signals.

- *Testing Low Sparsity Sequence Learning and Recall* [25]: The purpose of the test is to verify if the model can learn and recall patterns of sequences with a low sparsity rate, which refers to the proportion of active columns in a given input. The test initializes a Temporal Memory model and sets up connections with default parameters, including column dimensions of 64. The test then generates two sequences of active columns, one with 7 active columns and another with 5 active columns, and feeds them to the model. The test then attempts to recall both sequences and checks if the model can accurately recall the desired result, which is a set of 3 specific active columns that should be present in both sequences. As those columns are re-enforced in both cycle, the Compute() method should match our given desired output.

- *Testing Create Synapse in Distal Segment* [26]: The test ensures that the new synapse is created with the correct parameters and is added to the distal segment's list of synapses. The test initializes a new Temporal Memory and Connections object with default parameters, and then creates a distal segment for a specified cell. It then creates a new synapse in the distal segment with a specified presynaptic cell index and permanence value. The test checks that the distal segment contains only one synapse, which is the one that was just created. It also checks that the created synapse has the correct presynaptic cell index and permanence value. This test is important to ensure that the creation of synapses in a distal segment works correctly, which is essential for the learning and recall of temporal sequences in a Temporal Memory.

- *Testing New Segment Growth When No Matching Segment Found* [27]: This test method first initializes the TM object, connections, and parameters with default values. It then creates a distal dendrite segment and adds two synapses to it. The TM is then fed with a single active column and four active cells. Since there are no existing segments that match the active cells' pattern, a new dendrite segment should grow. The assertions made in the code check whether the new segment has indeed been created and has the expected number of synapses. The test passes if all the assertions are true. This test case is crucial in verifying the HTM algorithm's ability to learn and recognize previously unseen patterns by growing new dendrite segments when no matching segments are found. It ensures that the algorithm can continue learning and adapting to new data inputs without being limited by its existing knowledge.

- *Testing No Overlap in Active Cells* [28]: This unit test verifies that the output of the Temporal Memory algorithm does not contain any active cell that is present in more than one column. The test initializes a Temporal Memory object, sets up connections and parameters, and then computes active cells for two columns. The test then separates the active cells for each column and checks that no cell is active in both columns. The purpose of this test is to ensure that the Temporal Memory algorithm is correctly processing the input and producing an output that conforms to the constraints of the algorithm. The constraint that no active cell should be present in more than one column is a key aspect of the algorithm and must be strictly enforced for the algorithm to work correctly. This test is useful for detecting errors in the implementation of the Temporal Memory algorithm that might cause cells to be active in more than one column. If such errors are present, they can lead to incorrect predictions and reduce the accuracy of the algorithm. In conclusion, the above code implements a unit test that verifies the correct behavior of the Temporal Memory algorithm with respect to the constraint that no active cell should be present in more than one column.

- *Testing Temporal Memory Compute Returns Winner Cells* [29]: This unit test method is designed to verify that the Temporal Memory algorithm returns the correct winner cells. The Temporal Memory object is initialized and the connections and parameters are set. The method then creates an array of active columns and calls the Compute method of the Temporal Memory

object with these columns as input. The Compute method returns a ComputeCycle object, which contains the winner cells for each column. The method verifies that the number of winner cells is correct and that their parent column indices are as expected. Specifically, the method tests that the first active column has the first winner cell, and the second active column has the second winner cell. If the test passes, it indicates that the Temporal Memory algorithm is functioning correctly in identifying the winner cells for the given input columns.

- *Testing Temporal Memory Compute Returns Winner Cells* [30]: This is an existing tests which we modified to take multiple data in order to re-enforce the test and verify how the algorithm was supposed to work. Each set of test data consists of an array of active column indices, the expected number of winner cells, and an array of expected winner cell indices. The method initializes the Temporal Memory object, creates connections, applies parameters, and then computes the winner cells using the input active column indices. The method then checks whether the number of winner cells and their parent column indices match the expected values for each set of test data. The test passes if all checks are successful. This unit test method helps to ensure that the Temporal Memory algorithm's Compute method correctly identifies the winner cells based on the input active column indices.

- *Test Getting Least Used Cell* [31]: This test method verifies the functionality of the GetLeastUsedCell method of the Temporal Memory class. It creates a Connections object with some cells and segments, and then calls the GetLeastUsedCell method with a list of cells and a Random object. The test then asserts that the cell returned by the method is the one that is expected, which in this case is the cell with the lowest number of active synapses, i.e., c3. The GetLeastUsedCell method returns the least used cell from a given list of cells by randomly selecting a cell from the list that has the fewest number of active synapses. In this test method, the least used cell is verified by checking that it has the same ParentColumnIndex and Index as c3. Overall, this test method verifies that the GetLeastUsedCell method is working as expected and returns the correct cell with the fewest number of active synapses.

- *Testing Which Cells Become Active* [32]: This tests whether the correct cells become active in the Temporal Memory (TM) when certain columns are activated in the input space. The test method initializes the TM object and activates some columns in the input space. It then computes the next state of the TM and checks which cells become active. The expected set of active cells is obtained by getting the cells corresponding to the activated columns, and the actual set of active cells is obtained from the ComputeCycle object returned by the Compute method of the TM. These sets are then compared to ensure that they are equal using the SetEquals method of the HashSet class. If the sets are not equal, the test fails. The purpose of this test is to verify that the TM is correctly computing the next state based on the input and that the expected cells are becoming active.

- *Test Active Segment Grow Synapses According to Potential Overlap* [33]: The method tests the behavior of the TemporalMemory class's ability to grow synapses on active segments based on their potential overlap with previous active segments. This is a modified version of the existing test [34] method. Our modified version uses the DataRow attribute to specify multiple test cases with different input parameters and expected outputs. For each test case, the method creates a new instance of the TemporalMemory class and initializes it with default parameters. It then creates a list of winner cells based on the previous active columns and computes the next cycle of the temporal memory with the active columns. The method then creates a distal dendrite and adds synapses to it based on the potential overlap with the active columns. It computes the next cycle of the temporal memory with the new active columns and checks the number of presynaptic cells on the active segment against the expected output. The method uses assertions to check that the expected output matches the actual output. If the assertion fails, the test case fails, and the test output will show an error message with the reason for the failure. Overall, the test method checks that the TemporalMemory class can grow synapses on active segments according to their potential overlap with previous active segments, which is an important function of the Temporal Memory algorithm.

- *Testing Destroy Weak Synapse On Active Reinforce* [35]: We improved this test [36] by providing several alternative data using the DataRow attribute. The function being tested is responsible for destroying weak synapses on a segment that is being actively reinforced. Each DataRow attribute specifies the values of three integer parameters: prevActive, active, and expectedSynapseCount. The test method initializes a new instance of the Temporal Memory class, applies default parameters to a new instance of the Connections class, and initializes the temporal memory with the connections. It then sets up the input data for the function being tested, by creating a segment with four synapses, one of which is intentionally set to a low permanence value to represent a weak synapse. The temporal memory is then computed with the previous and current active columns to simulate the activation of the segment. Finally, the expected synapse count is compared against the actual synapse count on the active segment to determine if the weak synapse was destroyed.

- *Testing AdaptSegment Increase Permanence* [37]: This method is responsible for updating the permanence values of synapses in a given distal dendrite segment based on the activity of a set of previous active cells. The test ensures that the method correctly increases the permanence of synapses in the segment by a given increment value. To achieve this, the test sets up a Temporal Memory instance and creates a distal dendrite segment with a synapse to a presynaptic cell and an initial permanence value of 0.5. The test then creates a collection of previously active cells and defines permanence increment and decrement values. The "AdaptSegment" method is called on the segment with these inputs, and the test asserts that the permanence value of the synapse has been correctly increased by the specified increment value to 0.45. The purpose of this test is to verify that the "AdaptSegment" method correctly updates synapse permanence values based on previous cell activity, as expected by the temporal memory algorithm. This is an important aspect of the algorithm, as it enables the memory to learn and adapt to recurring patterns of input over time.

- *Testing AdaptSegment Previous Active Cells Contain Presynaptic Cell Increase Permanence* [38]: The purpose of this test is to verify the behavior of the AdaptSegment method of the TemporalMemory class under a specific scenario. The scenario being tested involves creating a new distal dendrite segment in a temporal memory and adding a synapse to it. The AdaptSegment method is then called with a list of active cells that includes the presynaptic cell connected to the synapse, a permanence increment value, and a permanence decrement value. The expected behavior is that the permanence of the synapse should increase by the specified increment value, since the presynaptic cell is included in the list of active cells. The test sets up the scenario by initializing a TemporalMemory object, creating a Connections object with a specified number of cells per column, and creating a new DistalDendrite object and a synapse with a specified permanence value, using the CreateSynapse method of the Connections object. The AdaptSegment method is then called with the Connections object, the DistalDendrite object, a list of active cells that includes the presynaptic cell and another cell, and the specified permanence increment and decrement values. Finally, the test asserts that the permanence of the synapse has been updated as expected. This test verifies that the AdaptSegment method correctly updates the permanence of a synapse in a distal dendrite segment in response to a list of active cells that includes the presynaptic cell connected to the synapse.

- *Testing Adding New Synapse To DistalSegment* [39]: This test initializes a temporal memory object and a connections object, applies default parameters to the connections object, and initializes the temporal memory with the connections object. The test creates a new distal dendrite segment associated with a cell object retrieved from the connections object. A new synapse is then created on the distal dendrite segment with a given permanence value and associated with another cell object retrieved from the connections object. The test then asserts that the synapse is present in the distal dendrite segment's synapses collection and that its permanence value matches the value set during creation. This test ensures that the functionality of creating a new synapse on a distal dendrite segment in a temporal memory implementation is working correctly. It can be used in further testing and development of the temporal memory implementation.

- *Testing Removing Synapse From Distal Segment* [40]: In this test we initialized a TemporalMemory object, a Connections object, and applied default parameters to the Connections object. Then, it creates a DistalDendrite object with a cell from the Connections object as its source and adds two Synapse objects to the DistalDendrite. The test checks that the number of Synapses in the DistalDendrite is 2, and then removes one of the Synapse objects using the KillSynapse method. The test checks that the number of Synapses in the DistalDendrite is now 1, and that the removed Synapse object is no longer in the DistalDendrite's Synapses collection, while the remaining Synapse object is still present. The purpose of this test is to ensure that the KillSynapse method correctly removes a Synapse object from a DistalDendrite object's Synapses collection. This functionality is important for the overall functioning of the TemporalMemory object, which relies on the ability to add and remove Synapse objects from DistalDendrite objects in order to learn and recognize patterns.

- *Testing Updating Permanence Of Synapse* [41]: This tests the ability of the algorithm to update the permanence value of a synapse in a distal dendrite. The test initializes a temporal memory and connections object with default parameters, and creates a distal segment and synapse with an initial permanence of 0.5. The code then increments and decrements the permanence value of the synapse by the permanence increment and decrement values specified in the HTM configuration. Finally, it asserts that the permanence value is updated correctly. This test is useful in verifying that the algorithm can adjust the permanence values of synapses.

- *Testing AdaptSegment to Centre Synapse Already at Centre* [42]: The purpose of the TestAdaptSegmentToCentre_SynapseAlreadyAtCentre is to ensure that the AdaptSegment method can properly adapt the permanence

values of synapses in a distal dendrite segment when a synapse is already connected to the center cell. The test case is arranged by creating a temporal memory object, initializing a connection object, and applying default parameters. Then a distal dendrite segment is created, and a synapse with a permanence value of 0.6 is added to the center cell. The AdaptSegment method is called with the central cell and the synapse as arguments, along with permanence increment and decrement values from the HTM configuration. The assertion checks that the permanence of the synapse has increased by the permanence increment value.

- *Testing Increase Permanence of ActiveSynapses* [43]: TestIncreasePermanenceOfActiveSynapses unit test method tests the ability of a temporal memory algorithm to increase the permanence of active synapses in response to a set of active columns. The test initializes a temporal memory object and applies the default parameters to it. It then activates a set of cells in the temporal memory and stores them in a list. Next, the algorithm is called again with a different set of active cells, and the permanence of the synapses in those active cells is increased. Finally, the test checks that the permanence of the synapses in the active cells has increased above a certain threshold. The test is designed to ensure that the temporal memory algorithm can learn and adapt to changing input patterns by increasing the permanence of the synapses in active cells. This is an important feature of a successful temporal memory algorithm, as it allows the system to recognize and respond to patterns in a dynamic and adaptive way. The test also verifies that the algorithm is correctly updating the permanence values of the synapses, which is critical for accurate prediction and classification of input patterns. Overall, this unit test serves as a validation of the functionality and effectiveness of the temporal memory algorithm.

- *Testing Get Least Used Cell* [44]: The purpose of TestGetLeastUsedCell1 unit test is to verify the correct functionality of the "GetLeastUsedCell" method. The unit test initializes a TemporalMemory object with a Connections object and applies default parameters. Then, a DistalDendrite object is created with two Synapse objects. The "GetLeastUsedCell" method is called with the Cells of a specified Column as a parameter to find the least used Cell in that Column. The returned Cell is then verified to ensure that it is not equal to the Cell with the lowest usage count. The usage count of the returned Cell is then incremented, and the "GetLeastUsedCell" method is called again to ensure that a different Cell is returned. This unit test verifies the functionality of the "GetLeastUsedCell" method and ensures that the method returns the expected result.

- *Test Active Cell Count* [45]: TestActiveCellCount verifies the functionality of the active cell count feature of the Temporal Memory algorithm. The purpose of this test is to ensure that the number of active cells in the output of the algorithm does not exceed the maximum number of active cells allowed per column. The test initializes the Temporal Memory algorithm and the Connections object using default parameters. It then sets the active columns to a single column (column 0), and computes the output of the algorithm with the "learn" parameter set to true. The number of active cells in the output is then checked to ensure that it is less than or equal to the maximum number of active cells allowed per column, which in this case is 5.

- *Test AdaptSegment to Centre* [46]: By providing varied data using DataRow attribute, we improved this test [47] that was already in place. The purpose of this test is to ensure that the AdaptSegment method correctly adapts the permanence of a synapse in a distal dendrite segment based on the activity of its connected cells. In this test, a new distal dendrite segment is created, and a synapse is added to it with an initial permanence value. The AdaptSegment method is then called with a set of active cells, and the expected permanence value of the synapse is provided as a test data row. The test asserts that the actual permanence value of the synapse after the AdaptSegment method call is equal to the expected value, within a tolerance of 0.1. This test uses the DataRow attribute to allow multiple sets of test data to be provided and tested in a single method. The test covers a range of initial and expected permanence values, testing the robustness of the AdaptSegment method across different input values.

- *Test Array Not Containing Cells* [48]: By providing diverse data via the DataRow attribute, we improved this test [49]. TestAdaptSegmentToCentre test case aims to verify that the output of the Temporal Memory algorithm does not contain certain excluded cells. The test is parameterized using the DataRow attribute, which allows for multiple sets of test inputs to be run in a single test method. The test method first initializes a TemporalMemory instance and a Connections instance using default parameters. It then calls the Compute method of the TemporalMemory instance with the specified activeColumns and retrieves the output in the form of a ComputeCycle instance. The test then checks whether the output ActiveCells collection does not contain the excluded cells specified in the input arguments using the CollectionAssert.DoesNotContain method. The purpose of this test method is to ensure that the Temporal Memory algorithm correctly identifies and excludes cells that should not be activated based on the input data.

- *Test Burst Not predicted Columns* [50]: We improved this test [51] by providing several alternative data using the DataRow attribute. TestBurstNotpredictedColumns checks the functionality of the Temporal Memory algorithm. The method takes two integer arrays as input: the "activeColumns" array, which specifies the columns that are active in the current iteration of the algorithm, and the "expectedBurstingCellIndexes" array, which contains the indices of the cells that are expected to be bursting in the current iteration. The purpose of the method is to test whether the Temporal Memory algorithm can correctly identify the bursting cells in the active columns, given the current input. To perform the test, the method creates a new instance of the Temporal Memory class and initializes it with a set of default parameters. It then computes the active cells for the specified set of active columns and checks whether the resulting set of active cells matches the expected set of bursting cells. The method uses the "ComputeCycle" class to perform the computation and compare the two sets of cells. The test is designed to be repeatable and reusable with different input data, as indicated by the annotation.

- *Testing No Change To Not Selected Matching Segments in Bursting Column* [52]: By providing diverse data via the DataRow attribute, we improved this test [53]. The purpose of the test is to verify that the permanence values of synapses in selected and non-selected distal segments remain unchanged after the bursting of a column, provided that no cells were previously active in the current time step. The test takes in several parameters such as previous active columns, current active columns, previous active cell indexes, bursting cell indexes, and permanence values. These parameters are used to set up a temporal memory and connections between cells. The method then creates distal dendrite segments and synapses and simulates two-time steps in the temporal memory by computing the active cells for both the previous and current time steps. Finally, the method asserts that the permanence values of synapses in the selected and non-selected distal segments remain unchanged after the bursting of a column. This is achieved by comparing the expected permanence values with the actual permanence values of the synapses.

III. Results

We conducted a series of Unit Tests to improve Temporal Memory algorithm functionality. A total of 38 Unit Tests were executed, which includes 36 passed tests and 2 skipped tests. The skipped tests do not have usual behavior and therefore excluded from the analysis.

If we consider the DataRow attributes, we applied on existing tests to modify them in order to strengthen those test cases then number of execution of tests will be 57 including two skipped tests.

The total duration of the tests is around 443~455 milliseconds, with an average test duration of 7.87 milliseconds. The longest test took around 122 milliseconds to complete which has DataRow attribute, while the shortest test took only 1 millisecond.

We also used the DataRow attribute in existing tests to test with different input data. A total of 28 DataRow tests were executed, covering a wide range of input values and edge cases. All DataRow tests passed successfully, indicating that the algorithm is robust and reliable under various input conditions.

We found some anomalies when tried to test the GetLeastUsedCell() method [54] and tried to pinpoint the problem. We found out that every time after calling the method and passing the random value, it again initializes the Random object value within the method itself, so we modified the method [55] and tried to test it. It successfully passed the test. It can be an improvement in the Algorithm.

IV. Discussion

In conclusion, our project aimed to write unit tests for the TemporalMemory algorithm, which required a steep learning curve due to the complexity of the algorithm. However, we were able to produce 38 unit test cases, including modifications to existing test cases, that were executed with numerous datasets. All test cases passed except for two, which were skipped due to unexpected behavior. Overall, our project was successful in providing a robust set of tests for the TemporalMemory algorithm, ensuring its correctness and reliability in various scenarios.

References

[1] *Temporal memory algorithm - numenta* (no date) *www.numenta.com*. Available at: https://www.numenta.com/assets/pdf/temporal-memory-algorithm/Temporal-Memory-Algorithm-Details.pdf (Accessed: March 28, 2023).

[2] Taylor, M. (2016) *HTM School, YouTube*. YouTube. Available at: https://www.youtube.com/playlist?list=PL3yXMgtrZmDqhsFQzwUC9V8MeeVOQ7eZ9 (Accessed: March 28, 2023).

[3] S. M. Ahmed, F. Akter, M.S. Doha, *Testing New Segment Growth when multiple matching segments found, 2023.* Source Code

[4] S. M. Ahmed, F. Akter, M.S. Doha, *Testing value of Synapse Permanence Updating When Matching Segments are Found, 2023.* Source Code

[5] S. M. Ahmed, F. Akter, M.S. Doha, *Testing Cells Per Column, 2023.* Source Code

[6] S. M. Ahmed, F. Akter, M.S. Doha, *Testing Custom Dimensions And Cells, 2023.* Source Code

[7] S. M. Ahmed, F. Akter, M.S. Doha, *Testing Column Dimensions, 2023.* Source Code

[8] Dobric, *et al*, *testNumberOfColumns, 2022,* Source code

[9] Dobric, *et al*, *testNumberOfCells. 2022,* Source code

[10] S. M. Ahmed, F. Akter, M.S. Doha, *Testing Recycle Least Recently Active Segment To Make Room For New Segment 2023,* Source code

[11] Dobric, *et al*, *Testing Recycle Least Recently Active Segment To Make Room For New Segment,2022,* Source code

[12] S. M. Ahmed, F. Akter, M.S. Doha, *Testing New Segment Add Synapses To All Winner Cells, 2023,* Souce code

[13] Dobric, *et al.* TestNewSegmentAddSynapsesToAllWinnerCells, *2022,* Source code

[14] S. M. Ahmed, F. Akter, M.S. Doha, *TestDestroyWeakSynapseOnWrongPrediction, 2023.* Source Code

[15] D. Dobric, *et al*, TestDestroyWeakSynapseOnWrongPrediction, *2022*, Source code

[16] S. M. Ahmed, F. Akter, M.S. Doha, *Test Adding Segment To Cell With Fewest Segments, 2023.* Source Code

[17] D. Dobric, *et al*, TestAddSegmentToCellWithFewestSegments, *2022*, Source code

[18] S. M. Ahmed, F. Akter, M.S. Doha, *Test Adapt Segment To Max, 2023.* Source Code

[19] D. Dobric, *et al*, TestAdaptSegmentToMax, *2022*, Source code

[20] S. M. Ahmed, F. Akter, M.S. Doha, *Testing Destroy Segments With Too Few Synapses To Be Matching, 2023.* Source Code

[21] D. Dobric, *et al*, TestDestroySegmentsWithTooFewSynapsesToBeMatching, *2022*, Source code

[22] S. M. Ahmed, F. Akter, M.S. Doha, *Test Punish Matching Segments In Inactive Columns, 2023.* Source Code

[23] D. Dobric, *et al*, TestPunishMatchingSegmentsInInactiveColumns, *2022*, Source code

[24] S. M. Ahmed, F. Akter, M.S. Doha, *Testing High Sparsity Sequence Learning And Recall, 2023.* Source Code

[25] S. M. Ahmed, F. Akter, M.S. Doha, *Testing Low Sparsity Sequence Learning and Recall, 2023.* Source Code

[26] S. M. Ahmed, F. Akter, M.S. Doha, *Testing Create Synapse in Distal Segment, 2023.* Source Code

[27] S. M. Ahmed, F. Akter, M.S. Doha, *Testing New Segment Growth When No Matching Segment Found, 2023.* Source Code

[28] S. M. Ahmed, F. Akter, M.S. Doha, *Testing No Overlap in Active Cells, 2023.* Source Code

[29] S. M. Ahmed, F. Akter, M.S. Doha, *Testing Temporal Memory Compute Returns Winner Cells, 2023.* Source Code

[30] S. M. Ahmed, F. Akter, M.S. Doha, *Testing Temporal Memory Compute Returns Winner Cells, 2023.* Source Code

[31] S. M. Ahmed, F. Akter, M.S. Doha, *Test Getting Least Used Cell, 2023.* Source Code

[32] S. M. Ahmed, F. Akter, M.S. Doha, *Testing Which Cells Become Active, 2023.* Source Code

[33] S. M. Ahmed, F. Akter, M.S. Doha, *Test Active Segment Grow Synapses According to Potential Overlap, 2023.* Source Code

[34] D. Dobric, *et al*, TestActiveSegmentGrowSynapsesAccordingToPotentialOverlap, *2022*, Source code

[35] S. M. Ahmed, F. Akter, M.S. Doha, *Testing Destroy Weak Synapse On Active Reinforce, 2023.* Source Code

[36] D. Dobric, *et al*, TestDestroyWeakSynapseOnActiveReinforce, *2022*, Source code

[37] S. M. Ahmed, F. Akter, M.S. Doha, *Testing AdaptSegment Increase Permanence, 2023.* Source Code

[38] S. M. Ahmed, F. Akter, M.S. Doha *Testing AdaptSegment Previous Active Cells Contain Presynaptic Cell Increase Permanence, 2023.* Source Code

[39] S. M. Ahmed, F. Akter, M.S. Doha *Testing Adding New Synapse To DistalSegment, 2023.* Source Code

[40] S. M. Ahmed, F. Akter, M.S. Doha *Testing Removing Synapse From Distal Segment, 2023.* Source Code

[41] S. M. Ahmed, F. Akter, M.S. Doha *Testing Updating Permanence Of Synapse, 2023.* Source Code

[42] S. M. Ahmed, F. Akter, M.S. Doha *Testing AdaptSegment to Centre Synapse Already at Centre, 2023.* Source Code

[43] S. M. Ahmed, F. Akter, M.S. Doha *Testing Increase Permanence of ActiveSynapses, 2023.* Source Code

[44] S. M. Ahmed, F. Akter, M.S. Doha *Testing Get Least Used Cell, 2023.* Source Code

[45] S. M. Ahmed, F. Akter, M.S. Doha *Test Active Cell Count, 2023.* Source Code

[46] S. M. Ahmed, F. Akter, M.S. Doha, *Test AdaptSegment to Centre, 2023.* Source Code,

[47] D. Dobric, *et al*, TestAdaptSegmentToCentre, *2022*, Source code

[48] S. M. Ahmed, F. Akter, M.S. Doha, *Test Array Not Containing Cells, 2023.* Source Code,

[49] D. Dobric, *et al*, TestAdaptSegmentToCentre, *2022*, Source code

[50] S. M. Ahmed, F. Akter, M.S. Doha, *Test Burst Not predicted Columns, 2023.* Source Code,

[51] D. Dobric, *et al*, TestBurstNotpredictedColumns, *2022*, Source code

[52] S. M. Ahmed, F. Akter, M.S. Doha, *Testing No Change To Not Selected Matching Segments in Bursting Column, 2023.* Source Code,

[53] D. Dobric, *et al*, *Testing No Change To NoT Selected Matching Segments In Bursting Column, 2022*, Source code

[54] D. Dobric, *et al*, *GetLeastUsedCell,, 2022*, Source Code

[55] S. M. Ahmed, F. Akter, M.S. Doha, *GetLeastUsedCell, 2023*, Source Code