///<summary>

/// Test adapt segment from syapse to centre when synapse is already at the center

/// <Summary>

[TestClass]

public class TemporalMemoryTests

{

[TestMethod]

public void TestAdaptSegmentToCentre\_SynapseAlreadyAtCentre()

TestTemporalMemoryComputeReturnsWinnerCells\

TestIncreasePermanenceOfActiveSynapses()

TestGetLeastUsedCell(

TestColumnDimensions

{

//Arrange

TemporalMemory tm = new TemporalMemory();

Connections cn = new Connections();

Parameters p = Parameters.getAllDefaultParameters();

p.apply(cn);

tm.Init(cn);

DistalDendrite dd = cn.CreateDistalSegment(cn.GetCell(0));

Synapse s1 = cn.CreateSynapse(dd, cn.GetCell(23), 0.6); // central

//Act

TemporalMemory.AdaptSegment(cn, dd, cn.GetCells(new int[] { 23 }), cn.HtmConfig.PermanenceIncrement, cn.HtmConfig.PermanenceDecrement);

//Assert

Assert.AreEqual(0.7, s1.Permanence, 0.1);

}

}

The code snippet provided is a unit test for a method called "AdaptSegmentToCentre" in a Temporal Memory class. This method is used to adjust the permanence value of synapses in a distal dendrite segment to enhance the prediction accuracy of the temporal memory algorithm. The unit test aims to verify that the method works correctly when a synapse is already located at the center of the segment. The unit test method is named "TestAdaptSegmentToCentre\_SynapseAlreadyAtCentre". It is decorated with the "TestMethod" attribute to indicate that it is a unit test. The method first initializes a temporal memory instance and sets up the necessary parameters for the algorithm. Then, it creates a distal dendrite segment and adds a synapse with a permanence value of 0.6 to the center of the segment. After setting up the initial state, the unit test calls the "AdaptSegment" method with the created segment, the cell located at the center, and the permanence increment and decrement values from the configuration. The purpose of this method is to adjust the permanence values of the synapses in the segment to strengthen the connections between the cells that are active in the current input pattern.Finally, the unit test checks if the permanence value of the synapse created in the setup stage is increased by 0.1, which is the permanence increment value in the current configuration. The expected value is 0.7, and the test verifies that the actual value of the synapse's permanence is within a 0.1 tolerance of the expected value using the "Assert.AreEqual" method. The unit test verifies that the "AdaptSegment" method works correctly when a synapse is already located at the center of the distal dendrite segment. It ensures that the permanence value of the synapse is increased by the configured amount, which indicates that the method correctly identifies and strengthens the existing connections between the cells. The unit test for the "AdaptSegmentToCentre" method with a synapse already at the center of the segment verifies that the method works correctly and produces the expected results. This is an important test case, as it ensures that the algorithm can accurately adapt the segment to improve prediction accuracy even when there is an existing connection at the center of the segment.

[TestMethod]

public void TestTemporalMemoryComputeReturnsWinnerCells()

{

TemporalMemory tm = new TemporalMemory();

Connections cn = new Connections();

Parameters p = getDefaultParameters(null, KEY.CELLS\_PER\_COLUMN, 2);

p = getDefaultParameters(p, KEY.MIN\_THRESHOLD, 2);

p.apply(cn);

tm.Init(cn);

int[] activeColumns = { 0, 1, 2, 3 };

ComputeCycle cc = tm.Compute(activeColumns, true) as ComputeCycle;

List<Cell> winnerCells = new List<Cell>(cc.WinnerCells);

Assert.AreEqual(4, winnerCells.Count);

Assert.AreEqual(0, winnerCells[0].Index);

Assert.AreEqual(2, winnerCells[1].Index);

}

------------------------------------------------------------------------------------------

The code snippet provided is a unit test for a method called "Compute" in a Temporal Memory class. This method computes the active and winner cells in a temporal memory based on the input pattern and the current state of the memory. The unit test aims to verify that the "Compute" method returns the correct set of winner cells given a specific input pattern and the current state of the temporal memory. The unit test method is named "TestTemporalMemoryComputeReturnsWinnerCells". It is decorated with the "TestMethod" attribute to indicate that it is a unit test. The method first initializes a temporal memory instance and sets up the necessary parameters for the algorithm. The parameters are set to have two cells per column and a minimum threshold of two. Then, it creates a connections instance based on the parameters and initializes the temporal memory instance with the connections instance. After setting up the initial state, the unit test calls the "Compute" method with an array of active columns representing the input pattern and the "true" value for the learn parameter, which indicates that the algorithm should adjust its permanence values during the computation. The method returns a ComputeCycle object that contains the winner cells and other computed values. Finally, the unit test checks that the number of winner cells is four, which is the number of active columns. It then verifies that the first winner cell has an index of 0, and the second winner cell has an index of 2. These expected values are based on the input pattern and the configured parameters. The unit test verifies that the "Compute" method correctly identifies the winner cells given a specific input pattern and the current state of the temporal memory. It ensures that the algorithm correctly adjusts the permanence values of the synapses and determines the cells that have the strongest connections with the active columns.The unit test for the "Compute" method in the Temporal Memory class verifies that the method returns the correct set of winner cells given a specific input pattern and the current state of the temporal memory. This is an important test case, as it ensures that the algorithm can accurately identify the cells that are most likely to be activated in the next time step based on the current input pattern and the history of the input patterns.

[TestMethod]

public void TestIncreasePermanenceOfActiveSynapses()

{

// Arrange

TemporalMemory tm = new TemporalMemory();

Connections cn = new Connections();

Parameters p = getDefaultParameters(null, KEY.MIN\_THRESHOLD, 2);

p.apply(cn);

tm.Init(cn);

int[] previousActiveColumns = { 0, 1, 2 };

int[] activeColumns = { 1, 2, 3 };

// Activate some cells

ComputeCycle cc = tm.Compute(previousActiveColumns, true) as ComputeCycle;

List<Cell> prevActiveCells = new List<Cell>(cc.ActiveCells);

Assert.AreEqual(12, prevActiveCells.Count);

// Increase permanence of synapses for active cells

cc = tm.Compute(activeColumns, true) as ComputeCycle;

List<Cell> activeCells = new List<Cell>(cc.ActiveCells);

Assert.AreEqual(12, activeCells.Count);

// Assert that the permanence of synapses has increased

List<Synapse> activeSynapses = new List<Synapse>();

//foreach (Cell cell in activeCells)

//{

//foreach (DistalDendrite segment in cell.DistalDendrites)

//{

// activeSynapses.AddRange(segment.Synapses.FindAll(synapse => synapse.IsDefined()));

//}

//}

foreach (Synapse synapse in activeSynapses)

{

Assert.IsTrue(synapse.Permanence > 0.5);

}

}

----------------------------------

This document presents a technical description of the test case named "TestIncreasePermanenceOfActiveSynapses" in the TemporalMemoryTests class. This test case is a part of the NuPIC (Numenta Platform for Intelligent Computing) library, which is an open-source project that aims to develop machine learning algorithms inspired by the human brain's neocortex.The objective of this test case is to verify that the permanence of synapses in a temporal memory network increases when some cells are activated, which is an essential feature of the Temporal Memory algorithm. The test case is implemented in C# language using the MSTest framework. The test case creates a new Temporal Memory object and a Connections object, which represents the connections between the cells in the network. The default parameters of the network are applied to the Connections object, and the Temporal Memory object is initialized with this connection object. The test case then activates some cells by computing the network with the given activeColumns. The previous active cells are verified, and then the permanence of the synapses in the active cells' distal dendrites is increased by calling the "Compute" method again with different activeColumns. The test case finally verifies that the permanence of synapses in the active cells' distal dendrites has increased by checking that all synapses' permanence is greater than 0.5. The test case is passed if all the synapses' permanence is greater than 0.5 after increasing the permanence of synapses. The "TestIncreasePermanenceOfActiveSynapses" test case ensures that the Temporal Memory algorithm's critical feature, i.e., the permanence of synapses, is working correctly. This test case verifies that the algorithm is capable of adapting to the input data by increasing the synapses' permanence in active cells, thus making them more likely to activate in the future.

[TestMethod]

public void TestGetLeastUsedCell()

{

Connections cn = new Connections();

Parameters p = getDefaultParameters(null, KEY.COLUMN\_DIMENSIONS, new int[] { 4 });

p = getDefaultParameters(p, KEY.CELLS\_PER\_COLUMN, 3);

p.apply(cn);

TemporalMemory tm = new TemporalMemory();

tm.Init(cn);

// Create a distal segment and synapses

DistalDendrite dd = cn.CreateDistalSegment(cn.GetCell(1));

cn.CreateSynapse(dd, cn.GetCell(0), 0.30);

cn.CreateSynapse(dd, cn.GetCell(2), 0.50);

// Get the least used cell in column 1

Cell leastUsedCell = TemporalMemory.GetLeastUsedCell(cn, cn.GetColumn(1).Cells, cn.HtmConfig.Random);

// Verify that the least used cell is correct

Assert.AreNotEqual(leastUsedCell, cn.GetCell(0));

// Increment the usage count of the least used cell

leastUsedCell.ParentColumnIndex++;

// Get the least used cell in column 1 again

Cell newLeastUsedCell = TemporalMemory.GetLeastUsedCell(cn, cn.GetColumn(1).Cells, cn.HtmConfig.Random);

// Verify that the new least used cell is not the same as the original least used cell

Assert.AreNotEqual(newLeastUsedCell, leastUsedCell);

}

The TestGetLeastUsedCell method is a unit test for the TemporalMemory class in an HTM (Hierarchical Temporal Memory) implementation. This method tests the functionality of the GetLeastUsedCell method which selects the least used cell from a given list of cells in a column. The test scenario is as follows: Create a connections object and set the parameters for the HTM implementation. Initialize a TemporalMemory object with the connections object. Create a distal segment and synapses on a specific cell. Call the GetLeastUsedCell method and verify that the least used cell is not the same as the cell with the lowest usage count. Increment the usage count of the least used cell. Call the GetLeastUsedCell method again and verify that the new least used cell is not the same as the original least used cell. Create a connections object and set the parameters for the HTM implementation using the getDefaultParameters method. Initialize a TemporalMemory object with the connections object. Create a distal segment and synapses on a specific cell using the CreateDistalSegment and CreateSynapse methods of the Connections object. Call the GetLeastUsedCell method with the appropriate parameters and verify that the least used cell is not the same as the cell with the lowest usage count using the Assert.AreNotEqual method. Increment the usage count of the least used cell by incrementing its ParentColumnIndex property. Call the GetLeastUsedCell method again with the appropriate parameters and verify that the new least used cell is not the same as the original least used cell using the Assert.AreNotEqual method. The TestGetLeastUsedCell method tests the functionality of the GetLeastUsedCell method and verifies that it selects the least used cell from a given list of cells in a column. The test passes if the selected cell is not the same as the cell with the lowest usage count and if the new least used cell is not the same as the original least used cell after the usage count is incremented.

/// <summary>

/// Testing if the TemporalMemory class initializes correctly with a custom number of column dimensions

/// </summary>

[TestMethod]

public void TestColumnDimensions()

{

// Initialize

TemporalMemory tm = new TemporalMemory();

Connections cn = new Connections();

Parameters p = Parameters.getAllDefaultParameters();

p.Set(KEY.COLUMN\_DIMENSIONS, new int[] { 32, 64 }); // Set custom column dimensions

p.Set(KEY.CELLS\_PER\_COLUMN, 32);

p.apply(cn);

tm.Init(cn);

int cnt = 0;

foreach (var item in cn.GetColumns())

{

cnt += item.Cells.Length;

}

Assert.AreEqual(32 \* 64 \* 32, cnt);

}

This technical documentation describes a test case that verifies the correct initialization of the TemporalMemory class with a custom number of column dimensions. The test ensures that the correct number of cells is created based on the specified custom column dimensions. The test case is implemented as a unit test method named "TestColumnDimensions". The method uses the Microsoft Visual Studio testing framework and is decorated with the [TestMethod] attribute. The test method starts by creating an instance of the TemporalMemory class and an instance of the Connections class. The getDefaultParameters() method is used to set default parameters, and then the custom column dimensions are set using the Set() method. The cells per column parameter is also set to 32 using the same method. The apply() method is called on the Connections instance to apply the parameters. Then the Init() method is called on the TemporalMemory instance to initialize the TemporalMemory with the specified Connections. The test method then calculates the total number of cells by iterating through all the columns and counting the number of cells. Finally, an assertion is made that the total number of cells is equal to the product of the custom column dimensions and the cells per column parameter. This test case verifies that the TemporalMemory class initializes correctly with a custom number of column dimensions. By specifying custom column dimensions, the number of cells per column can be increased or decreased to fit the requirements of the application. This test ensures that the correct number of cells is created based on the specified custom column dimensions.

/// <summary>

/// TestActiveCellCount: Verify that the number of active cells in the

/// output of Temporal Memory Algorithm is less than or equal to the maximum

/// number of active cells allowed per column.

/// </summary>

[TestMethod]

public void TestActiveCellCount()

{

// Initialize

TemporalMemory tm = new TemporalMemory();

Connections cn = new Connections();

Parameters p = getDefaultParameters(null, KEY.CELLS\_PER\_COLUMN, 5);

p.apply(cn);

tm.Init(cn);

int[] activeColumns = { 0 };

ComputeCycle cc = tm.Compute(activeColumns, true) as ComputeCycle;

var activeCells = cc.ActiveCells;

Assert.IsTrue(activeCells.Count <= 5);

}

Temporal Memory Algorithm is a machine learning algorithm based on the principles of neuroscience. It is used to analyze and learn patterns from input data streams. This algorithm uses a concept called sparse distributed representations (SDRs) to represent the input data. In this document, we will discuss the TestActiveCellCount method which is used to verify that the number of active cells in the output of the Temporal Memory Algorithm is less than or equal to the maximum number of active cells allowed per column. The purpose of this test is to ensure that the Temporal Memory Algorithm is functioning correctly and that the number of active cells generated by the algorithm does not exceed the maximum number of active cells allowed per column. The TestActiveCellCount method initializes the Temporal Memory Algorithm and sets the number of cells per column to 5. It then runs a compute cycle with one active column and checks the number of active cells generated by the algorithm. The test passes if the number of active cells is less than or equal to 5. If the number of active cells generated by the algorithm is less than or equal to 5, the test will pass. If the number of active cells is greater than 5, the test will fail. The TestActiveCellCount method is an important test for ensuring the correct functioning of the Temporal Memory Algorithm. By verifying that the number of active cells does not exceed the maximum number of active cells allowed per column, we can be confident that the algorithm is functioning correctly and learning patterns from the input data streams.