Prog Notes – openHTM

# Contents

[1 Contents 1](#_Toc437001726)

[2 Brain size 2](#_Toc437001727)

[3 Windows 2](#_Toc437001728)

[3.1 StartForm.cs 2](#_Toc437001729)

[3.2 MainForm.cs 2](#_Toc437001730)

[3.3 Simulation3DForm.cs 4](#_Toc437001731)

[4 Events 5](#_Toc437001732)

[5 Network Structure – Regions, columns, Cells, Synapses 6](#_Toc437001733)

[6 Event flow - Simulation3DForm <==> WatchForm 6](#_Toc437001734)

[7 CLA Class Hierarchy 8](#_Toc437001735)

[7.1 Diagram 8](#_Toc437001736)

[8 Step/Initialize 9](#_Toc437001737)

[9 HTM basics 10](#_Toc437001738)

[9.1 Wikipedia 10](#_Toc437001739)

[9.2 Numenta 11](#_Toc437001740)

[9.3 Region.cs Summary 14](#_Toc437001741)

[10 OpenHTM Network Structure 15](#_Toc437001742)

[10.1 Project.Open() 15](#_Toc437001743)

[10.2 Initialize HTM 16](#_Toc437001744)

[10.2.1 buttonInitHTM\_Click() 16](#_Toc437001745)

[10.2.2 NetControllerForm.InitializeNetwork() 17](#_Toc437001746)

[10.2.3 .InitializeRegion() 17](#_Toc437001747)

[10.2.4 FileSensor.FileSensor() 18](#_Toc437001748)

[10.3 NextStep 18](#_Toc437001749)

[10.3.1 buttonStepHTM\_Click() 18](#_Toc437001750)

[10.3.2 Region.NextTimeStep() 18](#_Toc437001751)

# Brain size

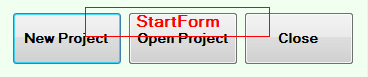
100 Billion cells

100 000 000 000

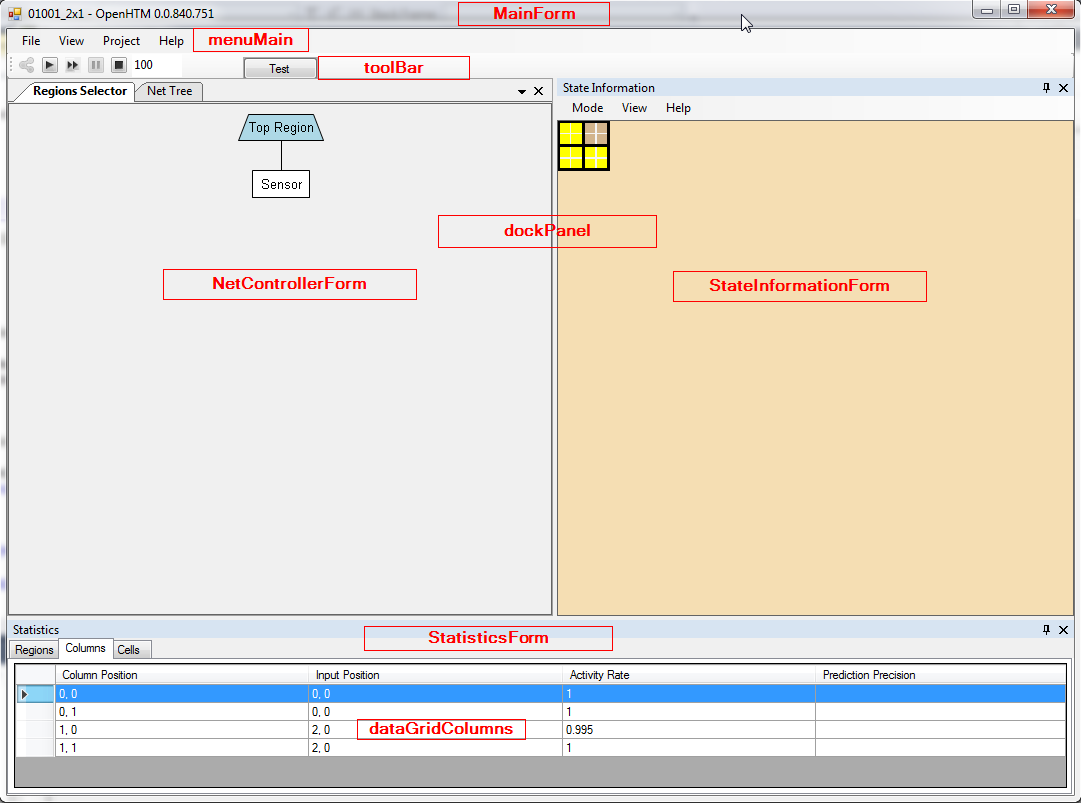
100 x 109

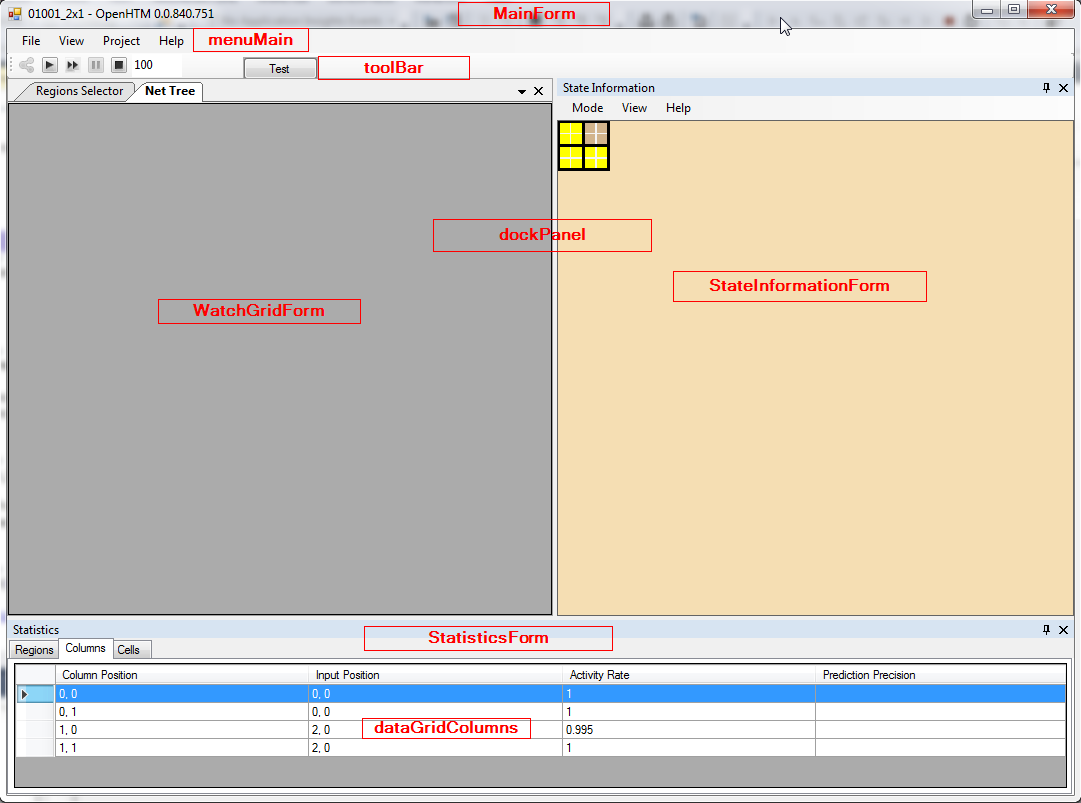
# Windows

## StartForm.cs



## MainForm.cs



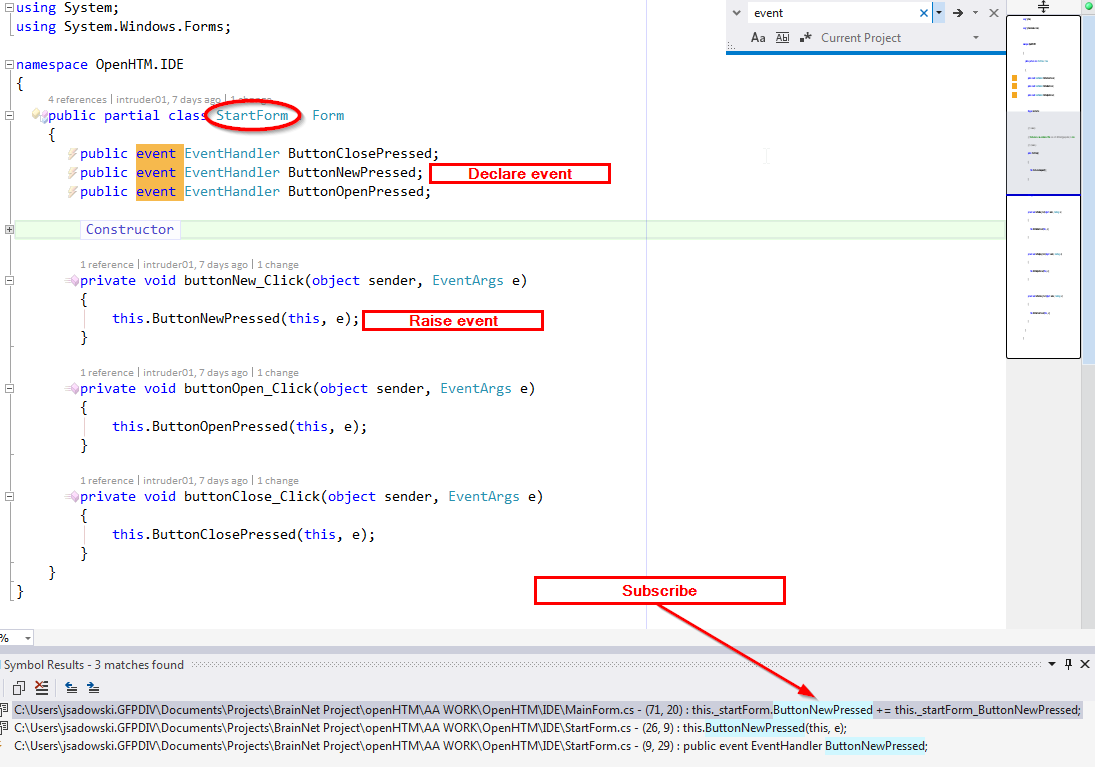


## Simulation3DForm.cs

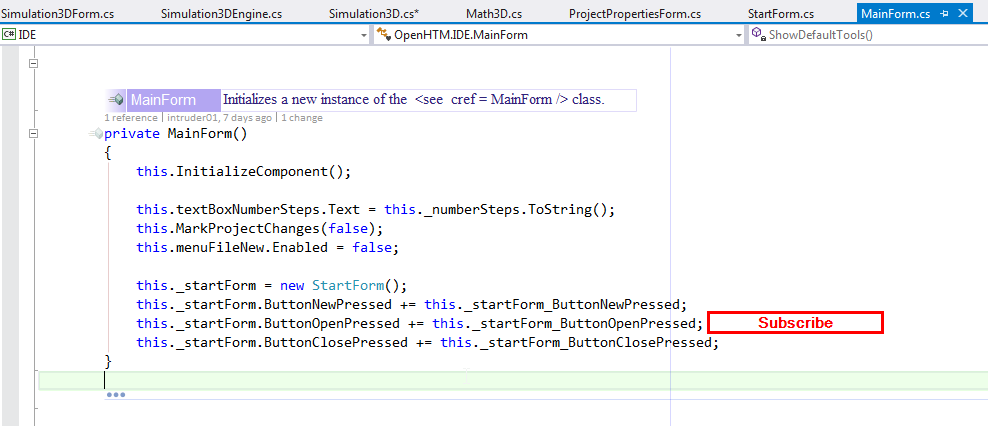


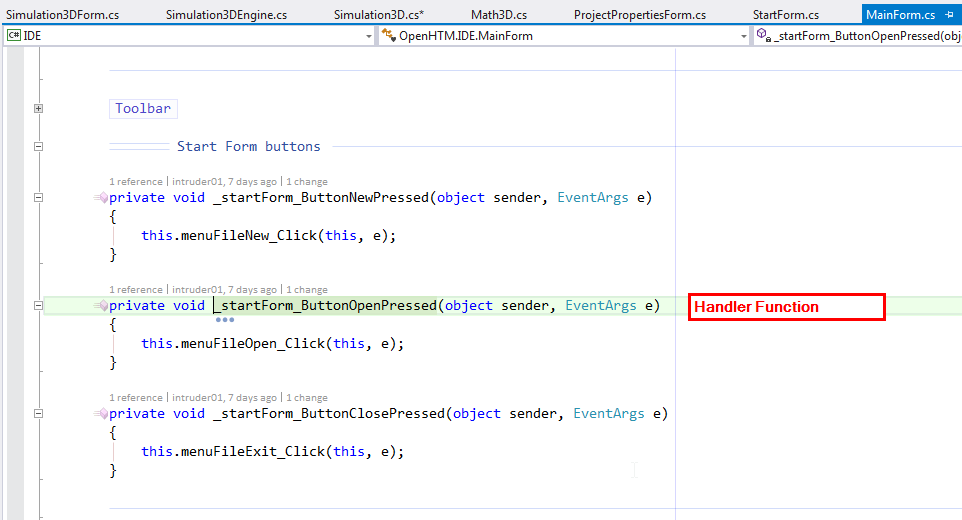
# Events

Provider: Raise event:



Subscriber:





# Event flow - Simulation3DForm <==> WatchForm

Simulation3DForm 🡪 WatchGridForm

* Selection changed

Simulation3DForm - declares delegate type (ChangedEventHandler Changed)

* OnSelectionChanged() method

WatchGridForm - subscribes +=

* ListChanged()

WatchGridForm 🡪 Simulation3DForm

* Value changed

To establish SelectionChanged updates, first engine is created, then proper updates are wired from engine to WatchForm.

1. When mouse clicked to start Sim3D, listener is wired to receive reference to the newly created engine

MainForm.cs

menuView3DSimulation\_Click()

{

Simulation3D.Start();

//this doesnt work - Engine is null at this point

//Simulation3D.Engine.SelectionChangedEvent += WatchGridForm.SimSelectionChanged\_Handler;

//wire event waiting for Engine created

Simulation3D.EngineStarted += WatchForm.Instance.Handler\_SimEngineStarted;

}

1. When engine created, event is called in Sim3D

Simulation3D.cs

global:

public delegate void SimEngineStarted (object sender, EventArgs e);

class:

public static event SimEngineStarted EngineStarted = delegate { };

private static void StartMethod()

{

//Create and display Engine…

EngineStarted ( Engine, new EventArgs () );

}

1. And received in WathForm. Handler is wired to the engine received, to start receiving SelectionChanged events.

WathForm.cs

//wait for Engine created

public void Handler\_SimEngineStarted ( object sender, EventArgs e )

{

Simulation3DEngine engine = (Simulation3DEngine)sender;

//subscribe to Engine objectSelectionChanged event

engine.SelectionChangedEvent += Handler\_SimSelectionChanged;

}

public void Handler\_SimSelectionChanged ( object sender, EventArgs e )

{

//List<Selectable3DObject> WatchList = (List<Selectable3DObject>) sender;

//Instance.RebuildWatchLists ( WatchList );

//ds = Instance.WatchListCellsToDataSet ();

SetPropertyGridDataSource ( (Region)sender );

}

1. So now, when selection changes in Sim3D, list of selected objects can be updated (not implemented now, displaying whole network) and event triggered to notify listeners.

Simulation3DEngine.cs

public delegate void SimEngineEvent\_ObjectSelectionChanged ( object sender, EventArgs e );

public event SimEngineEvent\_ObjectSelectionChanged SelectionChangedEvent = delegate { };

public void UpdateSelectedObjectList(object obj, bool add)

{

SelectionChangedEvent ( NetControllerForm.Instance.TopNode.Region, EventArgs.Empty );

}

1. Listener controls displaying selected item in WatchForm. (whole network is currently displayed)

WathForm.cs

public void Handler\_SimSelectionChanged ( object sender, EventArgs e )

{

//List<Selectable3DObject> WatchList = (List<Selectable3DObject>) sender;

//Instance.RebuildWatchLists ( WatchList );

//ds = Instance.WatchListCellsToDataSet ();

SetPropertyGridDataSource ( (Region)sender );

}

Change this to enable WatchForm when network is created (without starting Sim3D)

Will need to handle separate mouse selections from 3DSim and AdvancedVisualizer. Most likely, they will end up doing the same thing, but need to ensure display consistency. When selection is changed on one visualizer, watch data is updated, and the other visualizer reflects the new selection.

* Add network created event after FileOpen to populate WatchForm
* Add selection avents from StateInformationForm to WatchForm
* Add events to notify Sim3D when selection changed and vice-versa.

WatchForm is shown in MainForm.cs in ShowDefaultTools(), which is called on FileNew, FileOpen, FileOpenPrevious.

# CLA Class Hierarchy

## Diagram

Region : INode

Public class

Column

Public class

Cell : Selectable3DObject

Public class

ProximalSeg : Segment

Public class

DistalSeg : Segment

Public class

Segment : Selectable3DObject

public

ProxSynapse : Synapse

Public class

DistalSynapse : Synapse

Public class

Synapse : Selectable3DObject

public

InputCell

Public class

Cell : Selectable3DObject

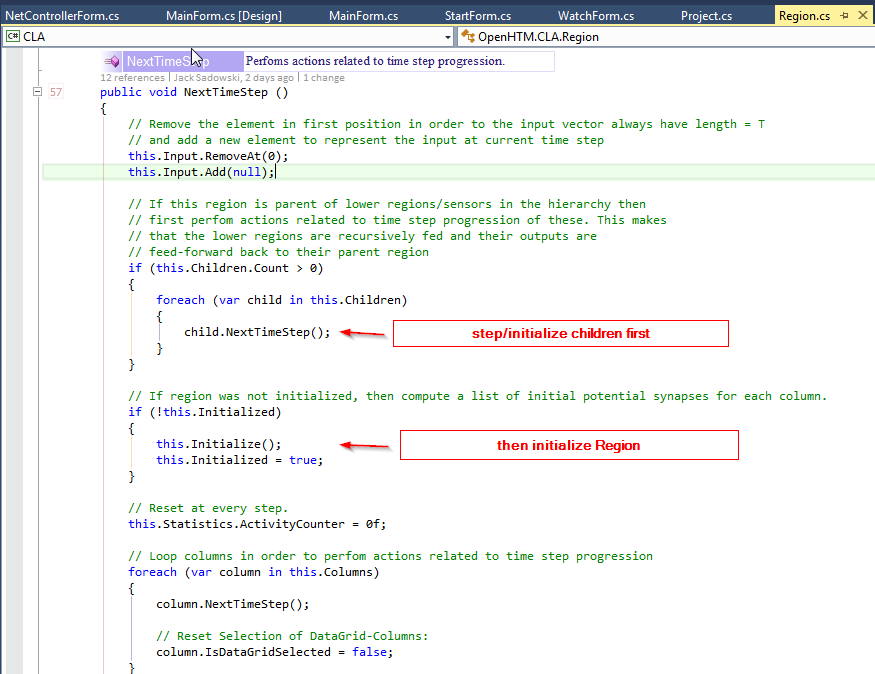
Public class

Selectable3DObject

Public class

# Step/Initialize

In Region.NextTimeStep():



* Initialization is performed while stepping
* Shoud this be separated and initialization performed during Project.LoadFromFile()?? – No, network can be reconfigured dynamically so it needs to initialize new areas while stepping.

# HTM basics

## Wikipedia

**Cortical learning algorithms**[[edit](https://en.wikipedia.org/w/index.php?title=Hierarchical_temporal_memory&action=edit&section=3" \o "Edit section: Cortical learning algorithms)]

The new generation of HTM learning algorithms relies on fixed-[sparsity](https://en.wikipedia.org/wiki/Sparse_coding) distributed representations.[[2]](https://en.wikipedia.org/wiki/Hierarchical_temporal_memory#cite_note-2)[[3]](https://en.wikipedia.org/wiki/Hierarchical_temporal_memory#cite_note-SDR-3) It models cortical columns that tend to inhibit neighboring columns in the neocortex thus creating a sparse activation of columns. A region creates a sparse representation from its input, so that a fixed percentage of columns are active at any one time.

Each HTM region consists of a number of highly interconnected [cortical columns](https://en.wikipedia.org/wiki/Cortical_column). A region is similar to layer III of the [neocortex](https://en.wikipedia.org/wiki/Neocortex). A cortical column is understood as a group of cells that have the same receptive field. Each column has a number of cells that are able to remember several previous states. A cell can be in one of three states: active, inactive and predictive state.

**Spatial pooling:** The receptive field of each column is a fixed number of inputs that are randomly selected from a much larger number of node inputs. Based on the input pattern, some columns will receive more active input values. Spatial pooling selects a relatively constant number of the most active columns and inactivates (inhibits) other columns in the vicinity of the active ones. Similar input patterns tend to activate a stable set of columns. The amount of memory used by each region can be increased to learn more complex spatial patterns or decreased to learn simpler patterns.

**Representing the input in the context of previous inputs:** If one or more cells in the active column are in the predictive state (see below), they will be the only cells to become active in the current time step. If none of the cells in the active column are in the predictive state (during the initial time step or when the activation of this column was not expected), all cells are made active.

**Predicting future inputs and temporal pooling:** When a cell becomes active, it gradually forms connections to nearby cells that tend to be active during several previous time steps. Thus a cell learns to recognize a known sequence by checking whether the connected cells are active. If a large number of connected cells are active, this cell switches to the predictive state in anticipation of one of the few next inputs of the sequence. The output of a region includes columns in both active and predictive states. Thus columns are active over longer periods of time, which leads to greater temporal stability seen by the parent region.

Cortical learning algorithms are able to learn continuously from each new input pattern, therefore no separate inference mode is necessary. During inference, HTM tries to match the stream of inputs to fragments of previously learned sequences. This allows each HTM region to be constantly predicting the likely continuation of the recognized sequences. The index of the predicted sequence is the output of the region. Since predictions tend to change less frequently than the input patterns, this leads to increasing temporal stability of the output in higher hierarchy levels. Prediction also helps to fill in missing patterns in the sequence and to interpret ambiguous data by biasing the system to infer what it predicted.

Cortical learning algorithms are currently being offered as commercial [SaaS](https://en.wikipedia.org/wiki/SaaS) by Numenta (such as Grok[[4]](https://en.wikipedia.org/wiki/Hierarchical_temporal_memory" \l "cite_note-4)).

The following question was posed to Jeff Hawkins September 2011 with regard to Cortical learning algorithms: "How do you know if the changes you are making to the model are good or not?" To which Jeff's response was "There are two categories for the answer: one is to look at neuroscience, and the other is methods for machine intelligence. In the neuroscience realm there are many predictions that we can make, and those can be tested. If our theories explain a vast array of neuroscience observations then it tells us that we’re on the right track. In the machine learning world they don’t care about that, only how well it works on practical problems. In our case that remains to be seen. To the extent you can solve a problem that no one was able to solve before, people will take notice."[[5]](https://en.wikipedia.org/wiki/Hierarchical_temporal_memory#cite_note-ai.stanford.edu-5)

## Numenta

Hierarchical Temporal Memory: A Summary

Hierarchical Temporal Memory (HTM) is a new computing paradigm that replicates the structure and function of the human neocortex. HTMs can perform tasks which have been easy for people but hard for computers - for example, recognizing objects, making predictions, understanding speech, or navigating in a complex environment. HTMs promise to approach or exceed human performance in many cognitive tasks.

HTM Networks Mirror the Hierarchical Structure of Real-World Objects

The structure of an HTM Network mirrors the nested hierarchical structure of objects in the world. Most larger objects are made up of smaller objects. A neighborhood consists of houses, roads, and schools, etc.

The hierarchical structure of the world is temporal as well as spatial. Language, for example, is a hierarchical temporal sequence. A speaker expresses an idea over time by combining consonants and vowels to make syllables, syllables to make words, etc.

Most environments such as markets, traffic, and human interactions have both spatial and temporal hierarchical structure. Because HTM Networks have a hierarchical structure, they could potentially model such environments.

The Structure of an HTM

HTMs consist of a multi-level hierarchy of nodes in an inverted tree structure (see Figure 1, where each rectangle represents a memory node).   
  
  *Figure 1 - Symbolic Representation of an HTM:*

Information flows up and down. Nodes in Level 4 receive sensory data from the environment. Nodes in Levels 2 and 3 exchange information with a parent node above and with child nodes below.

How an HTM Works

An HTM receives direct sensory input from its world. It builds a model of its environment by discovering the causes of sensory data and observing how those causes behave over time. To build the model, it looks for correlations between data points that are adjacent in space and time.

Each node in the lowest level of the hierarchy receives raw sensory data related to some quantity in a limited region of the environment. For example, in a machine vision application, a node in the lowest level might receive signals from a few camera pixels within a large CCD array. If the purpose of the HTM is to predict earthquakes, a node in the lowest level might receive data from adjacent pressure sensors along a fault line.

An HTM Network can receive raw input data for any physical quantity that sensors can measure. It can also receive data from a computer file. For example, in an HTM designed to predict behavior of the stock market, a node in the lowest level might receive economic data about companies in a particular industry.

In most applications, the sensory data entering each lowest-level memory node changes rapidly. Each node receives updated information at regular intervals; the interval period depends on the application.

*Discovering Spatial and Temporal Patterns*

Suppose each node in Level 4 receives as its input the state of 16 (bright or dark) binary pixels in a 4 x 4 array within a larger array. The number of possible patterns of bright and dark pixels in a 4 x 4 array is about 64K. Some patterns tend to recur, such as those in Figure 2, which look like edges and corners.

  *Figure 2 - Commonly Occurring Spatial Patterns:*

The node identifies each commonly occurring pattern as a "cause;" i.e., it infers that an external event must be causing the pattern to occur. The node gives each cause a name and stores the names in memory. The node ignores non-repeating patterns.

Each node in Level 4 also searches its input data for recurring temporal patterns or sequences, such as those in Figure 3. Figure 3a shows an edge moving through space horizontally; 3b shows a corner moving diagonally from upper left to lower right. The node associates each temporal sequence with a cause, assigns a number identification (ID) to each cause, and stores the ID in memory.

  *Figure 3 - Commonly Occurring Temporal Sequences:*

*Establishing and Reporting Beliefs*

Every node establishes beliefs about what happens when the HTM Network receives new input data. For example, every node might create a two-column table in which the left column lists the recurring spatial patterns it has identified and the right column lists the probability that each spatial pattern is present. We call such table of spatial patterns a belief.

Each node also creates a second table in which the left column lists recurring temporal sequences and the right column lists the probability that each sequence is occurring. This table is also called a belief.

During each cycle, each child node reports its belief about temporal sequences in the environment to its parent node. Depending on the application, each parent node can also send downward to its child nodes its belief about temporal sequences in the environment.

*Nodes at Higher and Lower Levels of the Hierarchy*

All nodes at all levels in the hierarchy run the same learning algorithm: they search input data for recurring temporal sequences and spatial patterns. Nodes at higher levels do not receive sensory data, but rather the beliefs of their child nodes. For nodes at higher levels, spatial patterns consist of commonly occurring combinations of beliefs that their child nodes simultaneously report. The temporal sequences consist of recurring changes in those beliefs.

As a result, nodes at different levels deal with different phenomena. At lower levels of the HTM, nodes deal with simple events occurring in lower levels of the hierarchy of its environment. These phenomena change quickly and occur in small regions of space. At higher levels of HTM, these higher level nodes oversee the lower level nodes and so receive influences from a broader range of sensory data. They deal with more complex phenomena (sequences of sequences) that occur in higher levels of the hierarchy of its environment. These phenomena tend to change more slowly and to occur over larger regions of space.

*Basic Functions of an HTM*

The HTM Network as a whole performs two key functions:

* *Discover causes.* Causes are persistent and repeating structures in the world. An HTM discovers causes by examining its input data to identify recurring spatial and temporal patterns.
* *Infer novel causes in its environment.* An HTM classifies its input data relative to known causes that it has identified. When an HTM sees a novel input, it determines not only the most likely high-level cause, but also the hierarchy of sub-causes.

Depending on the application, an HTM could also perform these additional functions:

* *Make predictions.* An HTM compares changes in the incoming data with known sequences, looks for similarities, and predicts events based on the way sequences commonly evolve.
* *Direct behavior.* An HTM could also operate a broad range of systems.

An HTM Network resembles a Bayesian Probability Network, in which the nodes constantly share information. HTMs exploit a variation of Belief Propagation, an iterative mathematical technique, to force the entire network to settle quickly on a set of mutually consistent beliefs.

Types of Problems Which HTMs Can Solve

A suitable application for HTMs would satisfy these conditions:

* The domain you want to model has an inherent hierarchy.
* The goal of the HTM is to discover causes in the real world, infer causes of novel input, make predictions, or direct behavior.
* The problem domain generates data related to the causes you want to investigate. The data flow and patterns change continuously through time, while its underlying causes remain relatively stable.
* The data have some spatial and temporal correlation.
* The problem domain provides a large set of training examples organized in temporal sequences.

To read the complete version of this whitepaper, please see [HTM: Concepts, Theory and Terminology](http://web.archive.org/web/20090604083735/http:/numenta.com/Numenta_HTM_Concepts.pdf)

## Region.cs Summary

/// <summary>

/// Represents an entire region of columns for the CLA.

/// </summary>

/// <remarks>

/// Code to represent an entire Hierarchical Temporal Memory (HTM) Region of

/// <see cref="Column"/>s that implement Numenta's new Cortical Learning Algorithms

/// (CLA).

/// The Region is defined by a matrix of columns, each of which contains several cells.

/// The main idea is that given a matrix of input bits, the Region will first sparsify

/// the input such that only a few Columns will become 'active'. As the input matrix

/// changes over time, different sets of Columns will become active in sequence.

/// The Cells inside the Columns will attempt to learn these temporal transitions and

/// eventually the Region will be able to make predictions about what may happen next

/// given what has happened in the past.

/// SpatialPooling snippet from the Numenta docs:

/// The code computes activeColumns(t) = the list of columns that win due to

/// the bottom-up input at time t. This list is then sent as input to the

/// temporal pooler routine.

/// Phase 1: compute the overlap with the current input for each column

/// Phase 2: compute the winning columns after inhibition

/// Phase 3: update synapse permanence and internal variables

///

/// 1) Start with an input consisting of a fixed number of bits. These bits might

/// represent sensory data or they might come from another region lower in the

/// hierarchy.

/// 2) Assign a fixed number of columns to the region receiving this input. Each

/// column has an associated dendrite segmentUpdateList. Each dendrite

/// segmentUpdateList has a set of potential synapses representing a subset of the

/// input bits. Each potential synapse has a permanence value.

/// Based on their permanence values, some of the potential synapses will be valid.

/// 3) For any given input, determine how many valid synapses on each column are

/// connected to active input bits.

/// 4) The number of active synapses is multiplied by a 'boosting' factor which is

/// dynamically determined by how often a column is active relative to its neighbors.

/// 5) The columns with the highest activations after boosting disable all but a fixed

/// percentage of the columns within an inhibition radius. The inhibition radius is

/// itself dynamically determined by the spread (or 'fan-out') of input bits.

/// There is now a sparse set of active columns.

/// 6) For each of the active columns, we adjust the permanence values of all the

/// potential synapses. The permanence values of synapses aligned with active input

/// bits are increased. The permanence values of synapses aligned with inactive

/// input bits are decreased. The changes made to permanence values may change

/// some synapses from being valid to not valid, and vice-versa.

# OpenHTM Network Structure

Region

Parent

Children

Region

Region

Parents

Sensor

Sensor

Children

Children nodes feedforward Parent nodes.

## Project.Open()

private void menuFileOpen\_Click(object sender, EventArgs e)

{

// Check if the current project has changed before continue operation

this.CheckCurrentConfigChanges();

this.CleanUp();

// Ask user for an existing repository

var folderBrowserDialog = new FolderBrowserDialog();

folderBrowserDialog.SelectedPath = Application.StartupPath +

Path.DirectorySeparatorChar + "Data";

//default to last project path

if (Properties.Settings.Default.LastProjectPath.Length > 0)

{

if (Directory.Exists ( Properties.Settings.Default.LastProjectPath ))

{

folderBrowserDialog.SelectedPath = Properties.Settings.Default.LastProjectPath;

}

}

// If repository exists, continue operation

DialogResult result = folderBrowserDialog.ShowDialog();

if (result == DialogResult.OK)

{

if (File.Exists(folderBrowserDialog.SelectedPath +

Path.DirectorySeparatorChar + Project.ProjectPropertiesFile))

{

// Initialize project state

this.MarkProjectChanges(false);

this.Text = Path.GetFileName(folderBrowserDialog.SelectedPath) + " - "

+ Application.ProductName + " " + Application.ProductVersion;

// Open project

Project.ProjectFolderPath = folderBrowserDialog.SelectedPath;

try

{

Project.Open();

this.\_startForm.Close();

this.menuFileNew.Enabled = true;

// Show tools like network selector, statistics, etc

this.ShowDefaultTools();

// Bind the UI controls

this.ReBind();

Properties.Settings.Default.LastProjectPath = Project.ProjectFolderPath;

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

else

{

MessageBox.Show(

"The current repository do not have any project files.",

"Error", MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

}

Project.cs:

public static void Open()

{

// TODO: Read from xml

// Load project properties from file

ProjectProperties.LoadFromFile(ProjectFolderPath +

Path.DirectorySeparatorChar + ProjectPropertiesFile);

// Load neural network configuration from file

NetConfig.LoadFromFile(ProjectFolderPath + Path.DirectorySeparatorChar

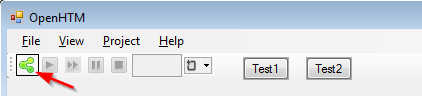
+ NetConfigFile);

}

Project.Open() only loads ProjectProperties and NetConfig classes from XML files. Both are just classes containing some parameters relating to the network. They don’t store current state of the network.

## Initialize HTM

### buttonInitHTM\_Click()



/// <summary>

/// Initialzes the HTM-Network by creating the htm-controller to connect to events database

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void buttonInitHTM\_Click(object sender, EventArgs e)

{

// Set flag initialization on

this.SimulationInitialized = true;

// Disable relevant buttons:

this.buttonInitHTM.Enabled = false;

this.EnableSteeringButtons(true);

// There was no simulation yet.

this.buttonPauseHTM.Enabled = false;

this.buttonStopHTM.Enabled = false;

// Initialize HTM Processor

NetControllerForm.Instance.InitializeNetwork();

}

### NetControllerForm.InitializeNetwork()

/// <summary>

/// Initialize the network starting from top region.

/// </summary>

public void InitializeNetwork()

{

// Initialize project parameters

Global.T = 5;

Global.SpatialLearning = ProjectProperties.Instance.SpatialLearning;

Global.TemporalLearning = ProjectProperties.Instance.TemporalLearning;

// Initalize synapses parameters

Synapse.ConnectedPermanence = NetConfig.Instance.SynapseParams.ConnectedPermanence;

Synapse.InitialPermanence = NetConfig.Instance.SynapseParams.InitialPermanence;

Synapse.PermanenceDecrement = NetConfig.Instance.SynapseParams.PermanenceDecrease;

Synapse.PermanenceIncrement = NetConfig.Instance.SynapseParams.PermanenceIncrease;

// Initalize nodes including region/sensors and their parameters

this.InitializeRegion(null, this.TopNode);

}

### .InitializeRegion()

public void InitializeRegion(TreeNode parentNode, TreeNode node)

{

switch (node.Params.Type)

{

case NetConfig.Type.Region:

{

var regionParams = (NetConfig.RegionParams) node.Params;

// Percentage calculations:

var percentageInput = (float) (regionParams.PercentageInputCol \* 0.01);

var percentageLocalActivity = (float) (regionParams.PercentageLocalActivity \* 0.01);

var percentageMinOverlap = (float) (regionParams.PercentageMinOverlap \* 0.01);

// Initalize region with params

Region parentRegion = null;

if (parentNode != null)

{

parentRegion = parentNode.Region;

}

var newRegion = new Region(0, parentRegion, regionParams.Size, percentageInput, percentageMinOverlap, regionParams.LocalityRadius, percentageLocalActivity, regionParams.CellsPerColumn, regionParams.SegmentActivateThreshold, regionParams.NewNumberSynapses);

node.Region = newRegion;

// If this region is parent of lower regions/sensors in the hierarchy then initialize them too.

foreach (var childNode in node.Children)

{

this.InitializeRegion(node, childNode);

}

}

break;

case NetConfig.Type.FileSensor:

{

var fileSensorParams = (NetConfig.FileSensorParams) node.Params;

// Initalize file sensor with params

var newFileSensor = new FileSensor(parentNode.Region, fileSensorParams.GetInputFilePath());

node.FileSensor = newFileSensor;

}

break;

}

}

### FileSensor.FileSensor()

Just adds itself to ParentRegion’s Children:

public FileSensor(Region parentRegion, string file)

{

// Set fields

this.ParentRegion = parentRegion;

this.\_file = file;

// Since this sensor is child of an higher region in the hierarchy then updates the set of children of the latter

if (this.ParentRegion != null)

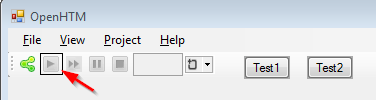
{

this.ParentRegion.Children.Add(this);

}

}

## Region.NextTimeStep()



### buttonStepHTM\_Click()

private void buttonStepHTM\_Click(object sender, EventArgs e)

{

//try

{

NetControllerForm.Instance.TopNode.Region.NextTimeStep();

}

//catch (Exception ex)

{

//MessageBox.Show(ex.Message);

}

// Refresh controls

this.RefreshControls();

this.menuView3DSimulation.Enabled = true;

this.buttonStopHTM.Enabled = true;

}

Just initiates NextTimeStep() for TopNode.

### Region.NextTimeStep()

1. Step all Children first
2. Initialize() itself (if necc.)
3. Step all Columns
4. SpatialPooling
5. TemporalPooling
6. Set input in Parent

public void NextTimeStep ()

{

// Remove the element in first position in order to the input vector always have length = T

// and add a new element to represent the input at current time step

this.Input.RemoveAt(0);

this.Input.Add(null);

// If this region is parent of lower regions/sensors in the hierarchy then

// first perfom actions related to time step progression of these. This makes

// that the lower regions are recursively fed and their outputs are

// feed-forward back to their parent region

if (this.Children.Count > 0)

{

foreach (var child in this.Children)

{

child.NextTimeStep();

}

}

// If region was not initialized, then compute a list of initial potential synapses for each column.

if (!this.Initialized)

{

this.Initialize();

this.Initialized = true;

}

// Reset at every step.

this.Statistics.ActivityCounter = 0f;

// Loop columns in order to perfom actions related to time step progression

foreach (var column in this.Columns)

{

column.NextTimeStep();

// Reset Selection of DataGrid-Columns:

column.IsDataGridSelected = false;

}

// Compute Region statistics

this.ComputeBasicStatistics();

// Perform pooling

this.PerformSpatialPooling();

this.PerformTemporalPooling();

// Column accuracies must be run after the processing finishes

// TODO: should all statistics be run after region processing?

this.ComputeColumnAccuracy();

this.ComputeNumberActiveColumns();

// If this region is child of an higher region in the hierarchy then

// feedforward input to the latter

if (this.ParentRegion != null)

{

this.ParentRegion.SetInput(this.GetOutput());

}

}

### Children.NextTimeStep()

Perform Next Step on all children if they exist.

// If this region is parent of lower regions/sensors in the hierarchy then

// first perfom actions related to time step progression of these. This makes

// that the lower regions are recursively fed and their outputs are

// feed-forward back to their parent region

if (this.Children.Count > 0)

{

foreach (var child in this.Children)

{

child.NextTimeStep();

}

}

### Region.Initialize()

1. this.inputSize – calculate by adding sizes of all children
   1. If HardCodedSpatial – use this as size
   2. Else – calculate Proportion factors for X, Y
2. Add Columns (this.Size)
3. Columns CreateProximalSegments()
4. Calc InhibitionRadius
5. Calc LocalityRadius (num of columns activated within InhibitionRadius)
6. Calc DesiredLocalActivity

/// <summary>

/// Prior to receiving any inputs, the region is initialized by computing a list of initial potential synapses for each column.

/// This consists of a random set of inputs selected from the input space. Each input is represented by a synapse and assigned a random permanence value.

/// The random permanence values are chosen with two criteria.

/// First, the values are chosen to be in a small range around connectedPerm (the minimum permanence value at which a synapse is considered "connected").

/// This enables potential synapses to become connected (or disconnected) after a small number of training iterations.

/// Second, each column has a natural center over the input region, and the permanence values have a bias towards this center (they have higher values near the center).

/// </summary>

/// <remarks>

/// In addition to this Uwe added a concept of Locality Radius, which is an

/// additional parameter to control how far away synapse connections can be made

/// instead of allowing connections anywhere. The reason for this is that in the

/// case of video images I wanted to experiment with forcing each Column to only

/// learn on a small section of the total input to more effectively learn lines or

/// corners in a small section without being 'distracted' by learning larger patterns

/// in the overall input space (which hopefully higher hierarchical Regions would

/// handle more successfully). Passing in 0 for locality radius will mean no

/// restriction which will more closely follow the Numenta doc if desired.

/// </remarks>

public void Initialize()

{

Segment.ActivationThreshold = SegmentActiveThreshold;

// Loop over all lower regions and sensors to calculate the total size of the input

// All regions/sensors should the same dimensions

var inputSize = new Size();

foreach (var child in this.Children)

{

inputSize.Width += child.Size.Width;

inputSize.Height += child.Size.Height;

}

this.InputSize = inputSize;

// Hard-coded means that input bits are mapped directly to columns. In other words the

// normal spatial pooler is disabled and we instead assume the input sparsification

// has already been decided by some preprocessing code outside the Region.

// It is then assumed (though not checked) that the input array will have

// only a sparse number of "1" values that represent the active columns

// for each time step.

if (this.HardcodedSpatial)

{

this.Size = this.InputSize;

}

// Calculate the conversion factor to get from region's column grid back to original input grid

this.InputProportionX = (this.InputSize.Width - 1) / ((float) this.Size.Width - 1);

this.InputProportionY = (this.InputSize.Height - 1) / ((float) this.Size.Height - 1);

// Create the columns based on the size of the region and input data to connect to

for (int x = 0; x < this.Size.Width; x++)

{

for (int y = 0; y < this.Size.Height; y++)

{

// Position of this column in the region grid

var positionInRegion = new Point(x, y);

// Calculate the conversion factor to get from region's column grid back to original input grid

var inputPositionX = (int) Math.Round(positionInRegion.X \* this.InputProportionX);

var inputPositionY = (int) Math.Round(positionInRegion.Y \* this.InputProportionY);

// Set 'center' position of columns in the original input grid

var centralPositionInInput = new Point(inputPositionX, inputPositionY);

// Create new column and add to region's list

var column = new Column(this, centralPositionInInput, positionInRegion);

this.Columns.Add(column);

//if (diag)

//{

// file.WriteLine ( positionInRegion.X + "," + positionInRegion.Y + ",," + centralPositionInInput.X + "," + centralPositionInInput.Y );

//}

}

}

// With hardcoded the Region will create a matching number of Columns to

// mirror the size of the input array. Locality radius may still be

// defined as it is still used by the temporal pooler. If non-zero it will

// restrict temporal segments from connecting further than r number of

// columns away.

if (this.HardcodedSpatial)

{

this.PercentageInputPerColumn = 1.0f / this.Columns.Count;

this.PercentageMinOverlap = 1.0f;

this.PercentageLocalActivity = 1.0f;

this.DesiredLocalActivity = 1;

}

else

{

// Create Segments with potential synapses for columns

foreach (var column in this.Columns)

{

column.CreateProximalSegments();

}

// Inhibition radius is recomputed

this.InhibitionRadius = this.AverageReceptiveFieldSize();

// Set the desired local activity, ie the number of columns that will be

// activated within a given spatial pooling inhibition radius

if (this.LocalityRadius == 0)

{

this.DesiredLocalActivity = (int) Math.Round(this.InhibitionRadius \* this.PercentageLocalActivity);

}

else

{

this.DesiredLocalActivity = (int) (Math.Pow(this.LocalityRadius, 2) \* this.PercentageLocalActivity);

}

this.DesiredLocalActivity = Math.Max(2, this.DesiredLocalActivity);

}

Write (0);

}

### Column.CreateProximalSegments()

/// <summary>

/// For each (position in inputSpaceRandomPositions):

/// 1. Create a new InputCell with input bit = position

/// 2. Attach a new ProximalSynapse

/// 3. Add the synapse to the synapse update list.

/// </summary>

/// <remarks>

/// Prior to receiving any inputs, the region is initialized by computing a list of

/// initial potential synapses for each column. This consists of a random set of inputs

/// selected from the input space. Each input is represented by a synapse and assigned

/// a random permanence value. The random permanence values are chosen with two

/// criteria. First, the values are chosen to be in a small range around connectedPerm

/// (the minimum permanence value at which a synapse is considered "connected"). This

/// enables potential synapses to become connected (or disconnected) after a small

/// number of training iterations. Second, each column has a natural center over the

/// input region, and the permanence values have a bias towards this center (they

/// have higher values near the center).

///

/// The concept of Locality Radius is an additional parameter to control how

/// far away synapse connections can be made instead of allowing connections anywhere.

/// The reason for this is that in the case of video images I wanted to experiment

/// with forcing each Column to only learn on a small section of the total input to

/// more effectively learn lines or corners in a small section.

/// </remarks>

internal void CreateProximalSegments()

{

// Calculates inputRadius for Columns from localityRadius

var inputRadius = (int) Math.Round(this.Region.LocalityRadius \* this.Region.InputProportionX);

// The coordinates of the input space for the Column

// Think of input space like a 'imaginary' square below the column center.

int minY, maxY, minX, maxX;

if (this.Region.LocalityRadius > 0)

{

// Compute values of input square and cut radius on edges

minX = Math.Max(0, this.CentralPositionInInput.X - inputRadius);

minY = Math.Max(0, this.CentralPositionInInput.Y - inputRadius);

maxX = Math.Min(this.Region.InputSize.Width - 1,

this.CentralPositionInInput.X + inputRadius);

maxY = Math.Min(this.Region.InputSize.Height - 1,

this.CentralPositionInInput.Y + inputRadius);

}

else

{

minX = 0;

minY = 0;

maxX = this.Region.InputSize.Width - 1;

maxY = this.Region.InputSize.Height - 1;

}

// Compute input area

int inputArea = (maxX - minX + 1) \* (maxY - minY + 1);

// Proximal synapses per Column (input segment)

// TODO: give user some control over the number of synapses per segment

var synapsesPerSegment =

(int) (inputArea \* this.Region.PercentageInputPerColumn);

//debug js

if (synapsesPerSegment <= 0)

{

synapsesPerSegment = 2;

}

// Updates minimum overlap value, i.e. the minimum number of inputs that must

// be active for a column to be considered during the inhibition step.

this.MinOverlap =

(int) Math.Round(synapsesPerSegment \* this.Region.PercentageMinOverlap);

// Overlap must be at least 1

if (this.MinOverlap <= 0)

{

this.MinOverlap = 1;

}

// Create all possible x,y positions for this column input space

var inputPositions = new List<Point>();

for (int y = minY; y <= maxY; y++)

{

for (int x = minX; x <= maxX; x++)

{

var inputPosition = new Point(x, y);

inputPositions.Add(inputPosition);

}

}

// Random sample of unique input positions (no duplicates).

// Tie the random seed to this Column's position for reproducibility

int randomSeed = (this.PositionInRegion.Y \* this.Region.Size.Width) + this.PositionInRegion.X;

IEnumerable<Point> inputRandomPositions =

inputPositions.RandomSample(synapsesPerSegment, randomSeed, false);

// Initialize the gaussian normal distribution

// The values are chosen to be in a small range around connectedPerm

// (the minimum permanence value at which a synapse is considered "connected")

var gausianNormalDistribution =

new Normal(Synapse.InitialPermanence, Synapse.PermanenceIncrement);

gausianNormalDistribution.RandomSource = new Random(randomSeed);

// Create proximal synapses to ramdom positions in input

int longerSide = Math.Max(this.Region.InputSize.Width, this.Region.InputSize.Height);

foreach (var position in inputRandomPositions)

{

var newInputCell = new InputCell(this.Region, position.X, position.Y);

if (this.Region.FullDefaultSpatialPermanence)

{

// Create new synapse and add it to segment

this.ProximalSegment.CreateSynapse(newInputCell, 1.0f);

}

else

{

// Get new value for permanence from distribution

double permanence = gausianNormalDistribution.Sample();

// Distance from 'center' of this column to the current position in the input space

var distanceInputFromColumn = new Point();

distanceInputFromColumn.X = this.CentralPositionInInput.X - position.X;

distanceInputFromColumn.Y = this.CentralPositionInInput.Y - position.Y;

double distanceToInput = Math.Sqrt(

(distanceInputFromColumn.X \* distanceInputFromColumn.X) +

(distanceInputFromColumn.Y \* distanceInputFromColumn.Y));

// Each column has a natural center over the input region, and the

// permanence values have a bias towards this center (they have higher values near

// the center)

int radiusBiasPeak = 2;

float radiusBiasStandardDeviation = 0.25f;

double localityBias = radiusBiasPeak / 2.0f \*

Math.Exp(Math.Pow(distanceToInput /

(longerSide \* radiusBiasStandardDeviation), 2) / -2);

double permanenceBias = Math.Min(1.0f, permanence \* localityBias);

// Create new synapse and add it to segment

this.ProximalSegment.CreateSynapse(newInputCell, permanenceBias);

}

}

}

### Columns.NextTimeStep()

Run NextTimeStep for each Columns

// Reset at every step.

this.Statistics.ActivityCounter = 0f;

// Loop columns in order to perfom actions related to time step progression

foreach (var column in this.Columns)

{

column.NextTimeStep();

// Reset Selection of DataGrid-Columns:

column.IsDataGridSelected = false;

}

### Region.SpatialPooling

// Compute Region statistics

this.ComputeBasicStatistics();

// Perform pooling

this.PerformSpatialPooling();

this.PerformTemporalPooling();

### Region.TemporalPooling

// Compute Region statistics

this.ComputeBasicStatistics();

// Perform pooling

this.PerformSpatialPooling();

this.PerformTemporalPooling();

### Region.ComputeColumnAccuracy()

// Column accuracies must be run after the processing finishes

// TODO: should all statistics be run after region processing?

this.ComputeColumnAccuracy();

this.ComputeNumberActiveColumns();

### Region.ComputeNumberActiveColumns()

// Column accuracies must be run after the processing finishes

// TODO: should all statistics be run after region processing?

this.ComputeColumnAccuracy();

this.ComputeNumberActiveColumns();

### Region propagate output

Propagate output up to parent region by setting parent’s input:

// If this region is child of an higher region in the hierarchy then

// feedforward input to the latter

if (this.ParentRegion != null)

{

this.ParentRegion.SetInput(this.GetOutput());

}

## Column.NextTimeStep()

Region calls NextTimeStep for every Column:

// Loop columns in order to perfom actions related to time step progression

foreach (var column in this.Columns)

{

column.NextTimeStep();

// Reset Selection of DataGrid-Columns:

column.IsDataGridSelected = false;

}