Getting Started With NuPIC

Document Version 1.2.1
Setpember 2008

Contents

Figures

Preface

This document gets you started working with the Numenta Platform for Intelligent Computing (NuPIC in the rest of this document). This preface gives some introductory information.

Topics

Scope of Document

This document is meant for developers who want to get started using NuPIC.

The focus of the document is to make you productive as soon as possible. The document tells you everything you need to know to implement your first HTM network. The *Advanced NuPIC Programming* document discusses advanced topics such the NuPIC architecture, session management, parallel processing, and schedulers.

Document Overview

This document consists of the following chapters:

- ❍ Chapter 1, Bitworm: Getting Started Example, gives instructions for running a simple example program (Bitworm) and explains how the example works.
- ❍ Chapter 2, Understanding HTM Development: Waves Example, considers the tasks involved in HTM generation.
- ❍ Chapter 3, Constructing an HTM Network, briefly introduces the components of an HTM Network, then steps you through the process of constructing a network using the Python classes and explains how to save the network to a network file.

- Chapter 4, Running the HTM Network, illustrates how to train an HTM Network and use the trained network to perform inference.
- Chapter 5, Debugging Your HTM Application, gives some tips for what you can do when things aren't working and explains how to use visualization tools.

A Glossary and an Index complete the document.

Related Documentation

The companion volume to this document is *Advanced NuPIC Programming*, which explores many of the concepts and programming tasks in more depth.

A number of white papers explore in more detail the concepts introduced in this document:

- The white paper *The HTM Learning Algorithms* is an in-depth discussion of how HTM Networks work.
- The *Numenta Node Algorithms Guide* discusses the learning algorithm implemented by the NuPIC learning nodes.
- The white paper *Problems that Fit HTMs* explains which problems are well suited for HTM Networks and which problems are still difficult to solve with today's algorithms.
- The *Numenta Node Plugin Developer's Guide* explains how to develop custom nodes.

Conventions

This document uses the following conventions:

- All filenames, code examples, and names of code elements are displayed in `code` font.
- Document names are given in *italic* font.

This document uses the following icons:

Table 1: Icons used in this document

Icon	Description
⚠	**Note**. A noteworthy item. If you do not pay attention to a note, nothing bad is likely to happen.
⚡	**Warning**. If you do not pay attention to a warning, data loss or other problems may result.
💡	**Tip**. Paying attention to a tip may make it easier and faster to use Numenta software.

Document History

This section lists changes made in the specified version of the document. Note that document version numbers do not correspond to release numbers.

Document Release	Description
1.0 September 2007	First release.
1.0.1 January 2008	Miscellaneous bug fixes. Added overview of node types to NuPIC tour.
1.1.0 May 2008	Complete revision of first chapter (Helper functions). Revision of other chapters to correctly reflect Tools changes (new node names, two nodes per level, and so on). This version of the document was not posted on the Web site.
1.2.0 June 2008	More naming changes and cleanup. This version of the document is being prepared for the Numenta HTM Workshop and posted on the Web site.
1.2.1 September 2008	Minor bug fixes.

For More Information

The Numenta website includes a variety of educational materials and forums to help you find answers to your questions. See http://numenta.com/for-developers.php.

See Summary and Guide to Documentation on page 40 for an overview of the documentation.

See How to Access NuPIC Built-in Help on page 84 for information on using Pydoc and getting node help.

1 Bitworm: Getting Started Example

This chapter gets you started with the Numenta Platform for Intelligent Computing (NuPIC) by explaining how you can run a simple example HTM Network and by examining the example scripts.

See the Numenta website for hardware and software requirements and installation instructions.

Topics

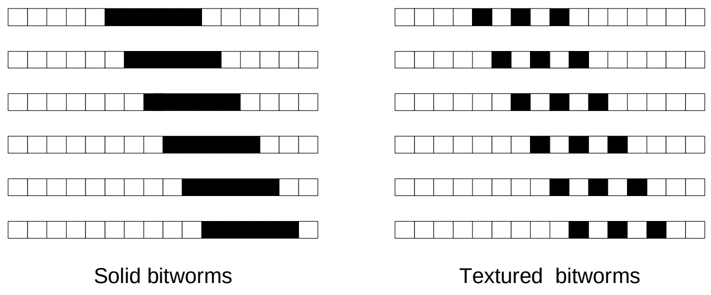
The Bitworm Example

This chapter introduces a simple example called Bitworm. The example illustrates how you might structure your input and category input, how to run your HTM Network, and how to interpret the results. Bitworm is not intended to be a realistic problem, instead, it's used as a Hello World example to get you up and running with NuPIC.

What are Bitworms?

Bitworms are 16-bit vectors. There are solid bitworms, which consist of consecutive on-bits, and textured bitworms, which consist of alternating on/off bits. In each case, the part of the vector that's not a bitworm consists of off bits. Here are some examples:

Figure 1 Solid and Textured Bitworms



The Bitworm example program trains an HTM Network to model the world of bitworms. After the HTM Network has been trained, you can submit new data and the HTM Network uses the model of the bitworm world to discriminate between solid and textured bitworms.

Bitworm Example Components

The Bitworm example consists of the following files, discussed in more detail below:

Script	Description	See
`RunOnce.py`	Creates the HTM structure, trains and saves the HTM Network, then runs the network with new data. You can edit `RunOnce.py` to experiment with different training settings.	Running the Example, page 15. RunOnce.py: Your Entry Point to Bitworms, page 20.
`GenerateData.py`	Generates training set data based on the settings in the `RunOnce.py` file.	GenerateData.py: An Example of Data Generation, page 20.
`GenerateReport.py`	For each group, prints the coincidences to a file. Grouping is an important part of the learning algorithm. You don't need to understand grouping or the learning algorithm for this simple example.	Examining the Report on page 16

Script	Description	See
`DisplayReport.py`	Displays the groups discovered by the training run. You must call this script explicitly, it's not called by `RunOnce.py`.	Displaying the Report on page 17
`ParameterExploration.py`	Illustrates how you can explore node parameters.	
`Cleanup.py`	Removes all generated files.	

Running the Example

This section explains how to run the example and how to explore what the HTM Network does by changing the example configuration. The example has been set up so you need to execute only one script.

To run the Bitworm example:

On Microsoft Windows, open a command prompt and type:

```
cd %NTA%\share\projects\bitworm
python RunOnce.py
```

On OS X and Linux — assuming `$HOME/nta` is the location where you installed the software — type the following at the command line:

```
cd $HOME/nta/current/share/projects/bitworm
python RunOnce.py
```

The example is set up so you always make modifications to the `RunOnce.py` script, then rerun `RunOnce.py`.

The script performs these tasks:

1. Calls `GenerateData.py` with the parameters set in `RunOnce.py` to generate a set of training data. The default is to generate temporally coherent data, that is, sequences of solid and textured bitworms of variable bitworm length. The minimum and maximum length are specified in the `trainingMinLength/testMinLength` and `trainingMaxLength/testMaxLength` parameters.
2. Creates the bitworm HTM Network, and calls helper functions to add nodes. The nodes are linked automatically.
3. Trains the HTM Network by calling the `TrainBasicNetwork()` function.

   During training, the nodes learn, that is, they construct a model of their world. Training proceeds level by level, starting at the bottom.
4. When the top-level node receives input during training, it also receives category information. It groups the data based on the categories.

5. `RunOnce.py` calls `RunBasicNetwork()` to explore how the trained HTM Network handles new data. The trained HTM Network looks at each input bitworm and determines the probability that the bitworm belongs to one or the other category. This process of categorizing data based on previous training is called inference.
6. Finally, `RunOnce.py` calls `GenerateReport`, which prints the coincidences for each group to a file called `report.txt`.

Examining the Report

The `GenerateReport.py` script that is run as part of `RunOnce` generates a report file named `report.txt` that includes the following information:

❍ General HTM Network statistics — Includes the number of nodes and the number of coincidences and groups the HTM Network found. For example:

```
General network statistics:
Network has  5 nodes.
Node names are:
    Sensor
    CategorySensor
    Level1
    Level2
    FileOutput


Node Level1 has 40 coincidences and 7 groups.
Node Level2 has 8 coincidences.
```

❍ Performance statistics — Information on how the HTM Network performed when the trained HTM Network was used to do inference with the training data. For example:

```
Performance statistics:
Comparing:  training_results.txt  with  training_categories.txt
Performance on training set:100.00%, 420 correct out of 420 vectors

Comparing:  test_results.txt  with  test_categories.txt
Performance on test set: 97.86%, 411 correct out of 420 vectors
```

Note that the Bitworm example gets very good results because this is a toy problem: The assumptions matched those of the current learning algorithm precisely. Achieving the same degree of success for more complex problems can be more challenging.

❍ A display of the groups and coincidences. Here's an example of the first two groups found by running Bitworm:

```
Getting groups and coincidences from the node Level1 in network '
trained_bitworm.xml

====> Group =  0
1 0 1 0 1 0 1 0 1 0 1 0 0 0 0 0
0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0
0 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0
0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 0
0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0
0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1
```

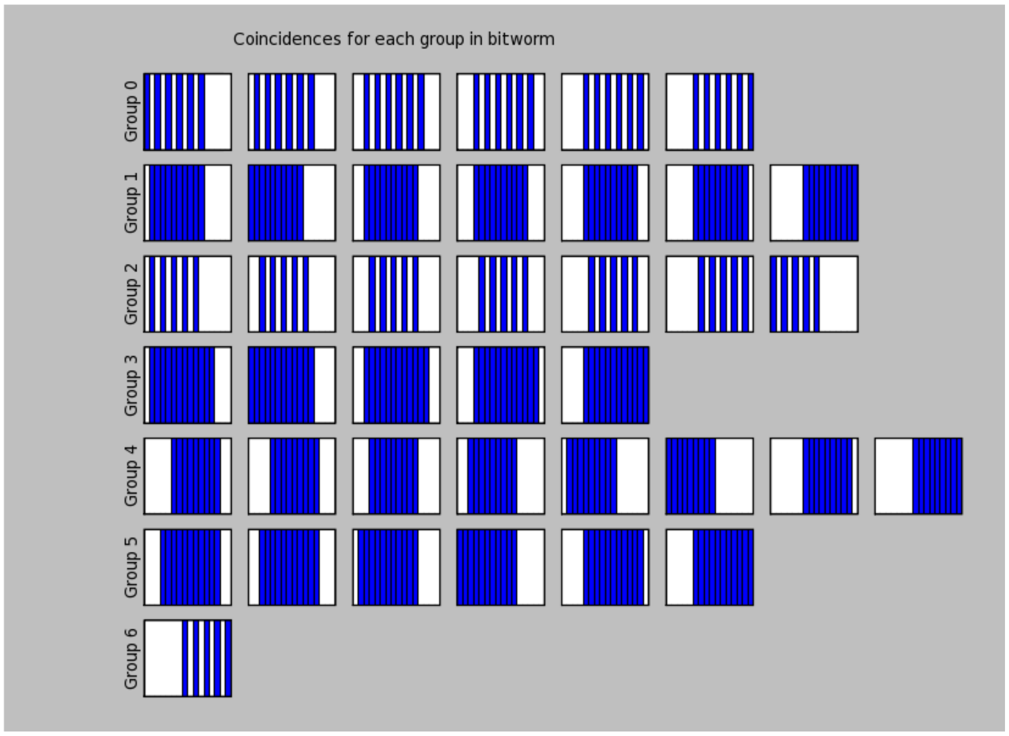
```
====> Group =  1
0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0
0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
```

This display can be helpful in a simple program, such as bitworms. It's easy to see how clean the groups are. Each group contains a different kind of bitworm. For other programs, using Numenta Visualizer or other tools to explore the grouping might be better than looking at a file. See Using HTM Network Visualizer on page 74 and Plotting and GUI Packages Bundled with NuPIC on page 78.

You can look at the `GenerateReport.py` file to see what Python calls you can use to retrieve information from the network. Comments in the file make it easy to understand your options.

Displaying the Report

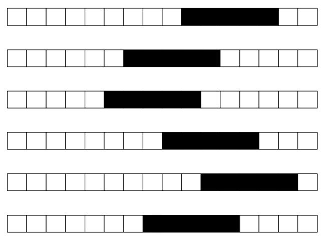
You can run `DisplayReport.py` to see a visual representation of the groups, for example:



Running the Example with Temporally Incoherent Data

You can change the `useCoherentData` parameter in `RunOnce.py` to generate solid and textured bitworms that are not presented in sequence, that is, that have no temporal relationship, as shown in Figure 2. Submitting those data to the trained HTM Network illustrates the importance of the temporal aspect of the training data.

Figure 2 Bitworms Presented Without Temporal Coherence



Solid bitworms, presented incoherently

To run the example with incoherent data:

1. In the `RunOnce.py` script, change the `useCoherentData` parameter to `False`.
2. Execute `RunOnce.py` again.

   The example runs with data that include both solid and textured bitworms but don't present sequences of solid bitworms followed by sequences of textured bitworms.
3. Examine the `report.txt` file this run generated. You should see that the HTM system found it difficult to find the groups and to categorize the data.

Running the Example with Noisy Data

The data generation script allows you to change the data by introducing some noise and to observe the results. There are two types of noise:

❍ Additive noise changes the bitworms by adding random noise to each input, for example:

0	0	0	0	1	1	1	1	1	0	0	0	0	0	without noise
-.1	0.1	0	1.01	1	1.1	.98	.98	1.05	0	0.05	0.09	0	0.07	with noise

❍ Bitflip noise causes the system to occasionally present a 0 instead of a 1 or a 1 instead of a 0. For example:

0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	without bitflip
0	0	0	1	0	1	1	1	0	0	0	0	0	0	1	0	with bitflip

In the bitworm example, you can introduce noise to the data and see how the noise affects recognition.

To run the example with noisy data:

1. In the `RunOnce.py` script, return the `useCoherentData` parameter to `True`.
2. Change `additiveNoiseTraining` to 0.1. This setting adds uniform random noise in the range [-0.1, 0.1] to the inputs. Once you start adding noise to the inputs, it becomes harder for the learning algorithm to detect temporal coherence.
3. Execute `RunOnce.py`.

   You get a Python exception with the message `The current parameters generated 400 groups, which exceeds the maximum of 25 groups`. This message means you didn't have enough outputs for the number of groups that were found. Although the number of underlying causes has not changed, the noise makes it harder for the algorithm to create a compact set of groups based on temporal coherence.

4. In `RunOnce.py` set `maxDistance` to 0.1 and save the revised file.

   The `maxDistance` parameter sets the maximum Euclidean distance at which two input vectors are considered the same during learning. The default for this parameter is 0, so a change usually means better performance if some noise is present. See Affecting Learning Node Behavior With Node Parameters on page 36 in *Advanced NuPIC Programming*.

5. Call `RunOnce.py` again. The script now runs without generating an exception.
6. Examine the `report.txt` file this run generated. You should see that the results are good; however, notice that the number of groups is fairly large compared to the original number. This is an indication that the learning algorithm found it more difficult to find the groups and categorize the data.
7. Set `maxDistance` to 0.2, rerun `RunOnce.py`, and examine `reports.txt`. This time the HTM gets the same number of groups as it did before you added noise.

This example illustrates how a combination of parameters (`maxDistance` and `maxGroups`) affects whether the HTM Network works well or does not work at all. If you wish, you can experiment with some of the other `RunOnce.py` parameters. See Table 2:, RunOnce.py parameters., on page 20.

Understanding the Example Scripts

This section briefly discusses the Bitworm example scripts.

RunOnce.py: Your Entry Point to Bitworms

The `RunOnce.py` script runs the component scripts of the example in sequence. As a rule, you should always call `RunOnce.py`, not one of the component scripts.

Like all NuPIC programs that use helper functions, `RunOnce.py` starts with these import statements:

```
from nupic.network import *
from nupic.network.helpers import AddSensor, AddClassifierNode, \
        AddZeta1Level, TrainBasicNetwork, RunBasicNetwork
```

`RunOnce` allows you to set the following parameters:

Table 2: RunOnce.py parameters.

`useCoherentData`	When set to true (the default), the `GenerateData.py` script creates sequences of solid bitworms followed by sequences of textured bitworms. When set to false, the `GenerateData.py` script mixes solid and textured bitworms randomly. In that case, the temporal element is missing from the data.
`numSequencesPerBitwormType`	Number of sequences for each bitworm type. For example, you could present ten sequences of textured bitworms and ten sequences of solid bitworms. The sequences are always separated by a row of zeros (0). `GenerateData.py` always generates the same number of sequences of each type.
`sequenceLength`	Length of each sequence (e.g 20 bitworm vectors, followed by one vector of zeros).

`trainingMinLength`
`trainingMaxLength`
`testMinLength`
`testMaxLength`
	Minimum and maximum length of the generated bitworms.

`inputSize`	Size of the input vector. Defaults to 16.
`bitFlipProbabilityTraining` `bitFlipProbabilityTesting`	Probability that a bit will be flipped from 0 to 1 or vice versa, that is, a 0 bit becomes 1 or a 1 bit becomes 0. Can be combined with additiveNoise. Default is 0. See Running the Example with Noisy Data on page 18.
`maxGroups`	Maximum number of groups that can be learned at level 1.
`maxGroupSize`	Specifies how large the groups in the temporal pooler can become.
`maxDistance`	Sets the maximum Euclidean distance at which two input vectors are considered the same during learning. See Affecting Learning Node Behavior With Node Parameters on page 36 in *Advanced NuPIC Programming*.

GenerateData.py: An Example of Data Generation

The `GenerateData.py` script generates a file with training data or testing data, plus an associated category file, using the parameter settings specified in `RunOnce.py`. Data are generated in sequences: For each sequence, the code generates a bitworm specified by `sequenceLength` using a random length and position (within the current parameter constraints), and then slides the bitworm left or right. At the end of each sequence, `GenerateData.py` inserts a line of zeros to reset the node so that the node does not attempt to learn temporal correlation between two bitworm sequences.

The script includes methods to generate data in which no temporal correlation exists. Those methods are called when the `useCoherentData` parameter is set to `False`.

Several aspects of this data setup are interesting:

❍ Each bitworm is not considered a sequence of bits but instead a single unit. The same way, a vision system processes input one "glance" at a time (not one pixel at a time) and compares "glances" over time.

❍ The data are presented as a set of logical sequences. Each sequence shows a single bitworm moving left or right. Within each sequence, each input pattern is correlated with the next pattern. This temporal coherence is critical for any HTM training data. HTMs (like people) expect a world that does not change drastically from one moment to the next. HTM learning algorithms exploit this property.

❍ In the data file, each sequence is followed by a row of zeros that separates it from the next sequence. At the start of each new sequence, the data generator decides which type of bitworm it wants to present next. This approach can be compared to viewing the frames of scenes in a movie. You need to see a set of frames to get a coherent picture, and then you can switch to the next scene.

RunOnce.py: Creating, Training, and Using the Trained Network

`RunOnce.py` performs the following tasks, discussed in this section:

❍ Creating the Untrained HTM Network File

❍ Training the HTM Network on page 23

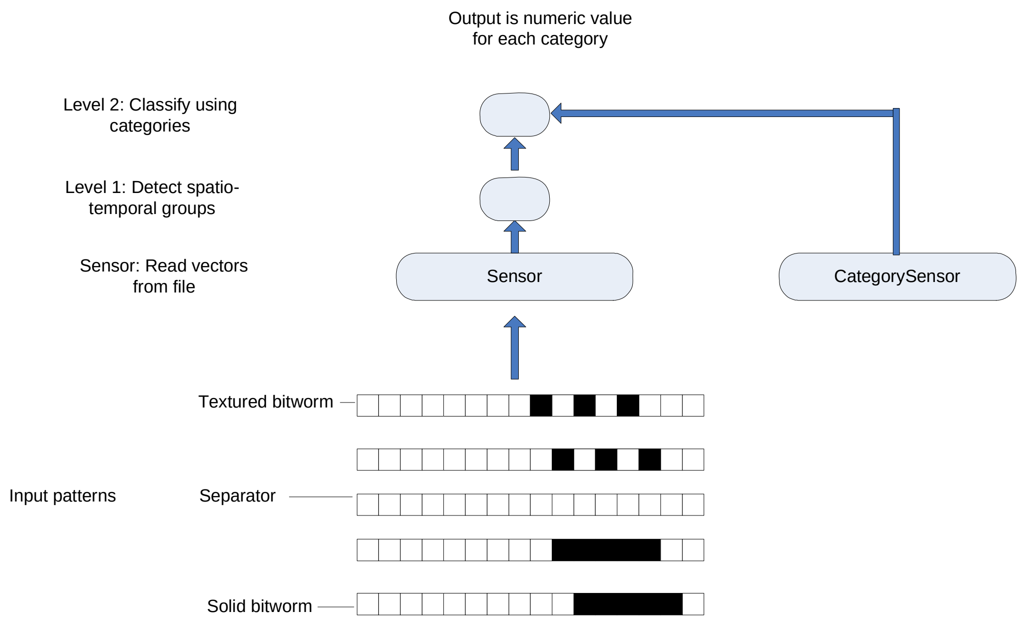
❍ Running the Trained Network with New Data on page 23

Creating the Untrained HTM Network File

`RunOnce.py` creates the `Network` and adds the nodes using helper functions. See Constructing an HTM Network on page 41.

Figure 3 shows the hierarchy of nodes in the bitworm example. This is the simplest possible HTM hierarchy.

Figure 3 Nodes in the Bitworm Example


Output is numeric value for each category
Level 2: Classify using categories
Level 1: Detect spatio-temporal groups
Sensor: Read vectors from file
Sensor
CategorySensor
Textured bitworm
Input patterns
Separator
Solid bitworm

To create this hierarchy, the script goes through these steps:

1. Creates the `Network` instance.

   ```
   bitNet = Network()
   ```

2. Uses the `AddSensor()`, `AddZeta1Level()`, and `AddClassifierNode()` function to specify each level in turn.

   — `AddSensor()` uses `featureVectorLength` to specify the length of a pattern.

   ```
   AddSensor(bitNet, featureVectorLength = inputSize)
   ```

   — `AddZeta1Level()` adds a `Zeta1Node` node, which uses the older learning algorithm. Other examples use `AddLevel()` instead to add spatial and temporal poolers that implement the new learning algorithm.

   ```
   AddZeta1Level(bitNet, numNodes = 1)
   ```

   — `AddClassifierNode()` adds a Zeta1TopNode by default and specifies the number of categories.

   ```
   AddClassifierNode(bitNet, numCategories = 2)
   ```

   The bitworm network has a data sensor and a category sensor, one bottom-level node, one top-level node, and one effector. In this example,

3. The `Network` accessor functions are used to set parameters, for example:

   ```
   bitNet['level1'].setParameter('maxDistance',maxDistance)
   bitNet['level1'].setParameter('transitionMemory', transitionMemory)
   ```

Training the HTM Network

During training, each node in the HTM Network builds a model of its world using the available input data.

`RunOnce.py` performs training using the `TrainBasicNetwork()` function. This function trains the network created earlier and returns a `RunTimeNetwork` object, which contains the trained network. `TrainBasicNetwork()` uses the training files generated by `GenerateData.py` (see GenerateData.py: An Example of Data Generation on page 20).

```
bitNet = TrainBasicNetwork(
      bitNet,
      dataFiles = [trainingFile],
      categoryFiles = [trainingCategories])
```

Running the Trained Network with New Data

`RunOnce.py` includes the `RunBasicNetwork()` function, which runs the trained HTM Network in inference mode using a given data file. You can invoke the function with the original data or with the new data.

`RunOnce.py` tests the network first with the training data.

```
accuracy = RunBasicNetwork(
      bitNet,
      dataFiles     = [trainingFile],
      categoryFiles = [trainingCategories],
      resultsFile   = trainingResults)
print "Training set accuracy with HTM = ", accuracy*100.0
```

`RunOnce.py` also submits new data to the trained HTM Network and judge how well the network learned the categories.

```
accuracy = RunBasicNetwork(
      bitNet,
      dataFiles     = [testFile],
      categoryFiles = [testCategories],
      resultsFile   = testResults)
print "Test set accuracy with HTM = ", accuracy*100.0
```

2 Understanding HTM Development: Waves Example

This chapter explores the sequence of tasks involved in developing an HTM application. The chapter uses the Waves example to illustrate two aspects of each task:

❍ How did the example developer perform that task? For example, what is the data representation?

❍ What are the most interesting aspects of this task? For example, what are some questions to consider when designing your own application.

The example in this chapter is slightly more complex than Bitworm. It uses the `RuntimeNetwork` API directly instead of just using the helper functions in Bitworm.

Topics

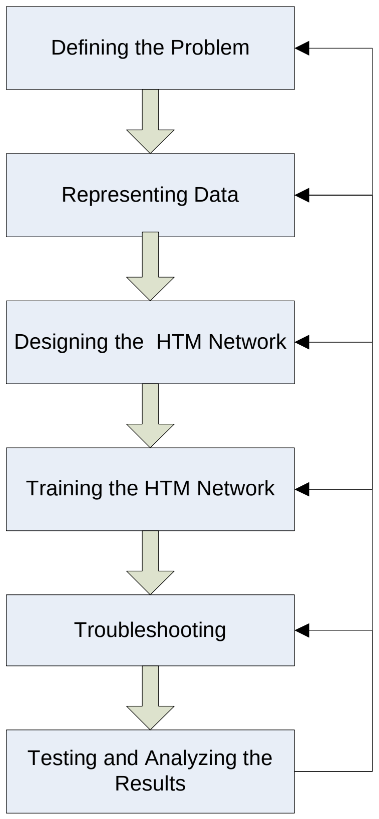
Development Overview

The process of creating an HTM application differs significantly from a traditional software engineering process. HTM Networks learn by building a statistical model based on a sequence of input data. The application developers task is not to specify an algorithm but to come up with a suitable representation of the data and to find the optimal parameter settings for the problem at hand. The quality of the final network depends on many factors, such as the network configuration (does the hierarchy have two levels or three?), node parameters (is `maxDistance` 0 or 0.2?), the training data (are there sufficient sequences?), and so on.

The HTM development process is iterative at a very high level. You don't just write the program, find problems, rewrite and rerun. Instead, each task in the process might influence other tasks earlier or later in the process. During HTM Network design, you might need to refine the problem definition. During testing and analysis, you might find that the data representation is less than optimal.

Pay particular attention to the early stages of the process. Unless your data are represented in a way that's easy to understand for the HTM (and often for humans as well), the results will most likely not be satisfactory.

Figure 4 Numenta Development Process



This chapter walks through the development process using the Waves example as the context.

Defining the Problem

Finding a problem that's well suited to an HTM Network and formulating that problem in a fashion that yields good results are important parts of the HTM development cycle. Many developers redefine the problem as they see the results of running early prototypes of their HTM Network.

See the white paper *Problems that Fit HTMs* for a discussion of problem definition. While the paper's focus is on what problems fit (and don't fit) HTMs, it implicitly discusses problem definition, for example, by contrasting problems that have temporal elements with problems that don't have them.

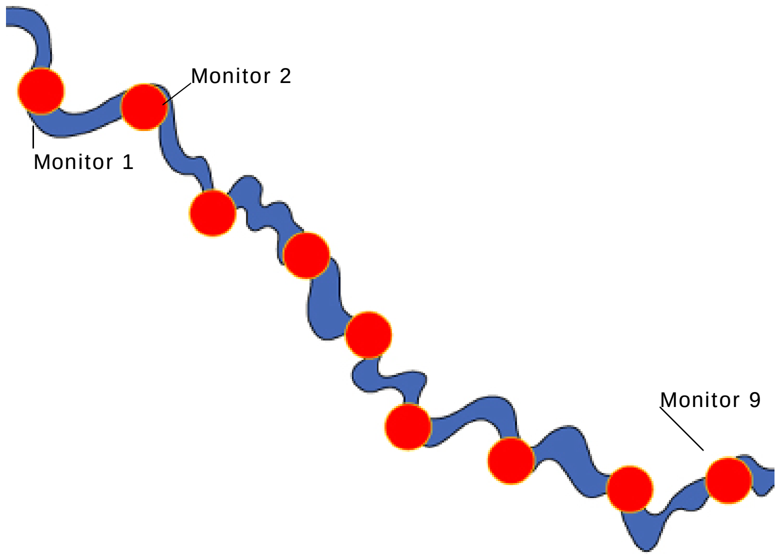
Problem Definition in the Waves Example

The Waves example generates data that represent temperature readings at fixed points in a moving stream. The example creates an HTM Network and trains the network with a set of input data. After training, the example submits new data to the HTM Network, and the network classifies those data.

This example assumes that 32 temperature monitors have been placed at fixed locations along the length of a stream. As time progresses, the monitors later in the stream see what the earlier monitors had seen.

This example assumes that the heat does not diffuse, and that hot and cold points move down the stream unaltered.

Figure 5 Waves Example



In the example, each hot spot is modeled as a Gaussian curve with positive amplitude; each cold spot is modeled as a Gaussian curve with negative amplitude. The state of the river is characterized as the combination of hot and cold spots.

The example uses these four state categories:

❍ One warm spot (category 0)

❍ Two warm spots (category 1)

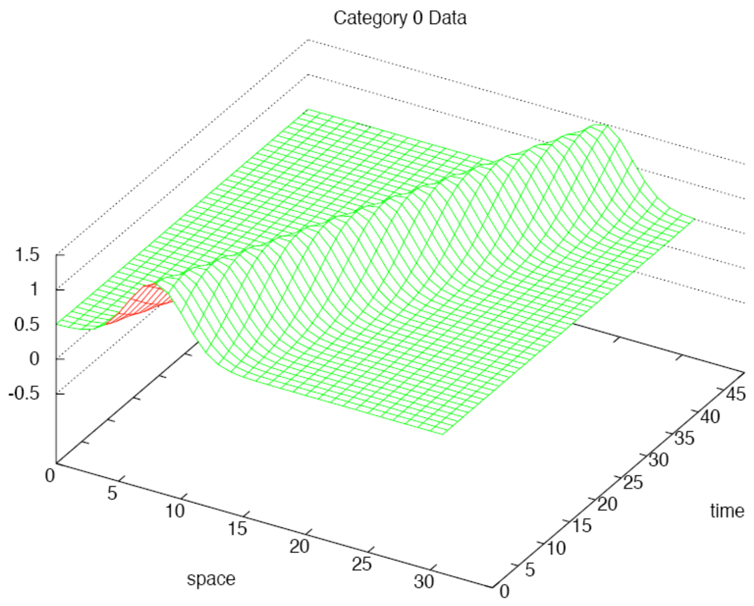
❍ One cold and one warm spot (category 2)

❍ Two warm spots with a cold spot in between (category 3)

The warm and cold spots could be anywhere in the river. The goal is to have an HTM application that examines incoming data and recognizes whether the river is in one of the four categories.

The following illustration shows Category 0 (one warm spot).

Figure 6 River in Category 0 State (One Warm Spot)



Note that the graph was generated as a by-product of calling `RunOnce.py`. After the `RunOnce.py` script completes, you can find the graphs in a subdirectory called `visuals`. As you change the data generation parameters, you can examine the corresponding updated graphs.

Input to the Temperature Monitors

The input vectors have both a spatial and a temporal element. Having both elements is critical for a problem well suited for an HTM Network.

❍ Spatial continuity — At each point in time, each monitor has a temperature reading. There is spatial continuity between monitor readings: At any point in time, neighboring monitors have a strong correlation while monitors far away from each other have no correlation.

❍ Temporal continuity — As time progresses, monitors later in the stream see what earlier monitors saw previously. In addition the reading at time T for each individual monitor is related to the reading at time T-1 and T+1 (there are no abrupt temperature changes). In contrast, readings far apart in time are not correlated.

Notes on Problem Definition

Take special care with problem definition. If problem definition was not thought out carefully, the HTM algorithms might be unable to work with your problem.

❍ Before you can start writing your code, you must determine the nature of the problem's spatial and temporal correlations, and you must consider how these correlations are structured hierarchically.

  To be successful, your HTM Network must build a statistical model of the underlying causes in your problem domain.

❍ Think carefully about the hierarchy of the domain's spatial and temporal correlations. These correlations suggest the best topological structure of your HTM Network.

❍ After you've developed an initial topology, that is, a hierarchy with a number of levels and a number of nodes per levels that works well, you can start to write code and think about tweaking node parameters such as `requestedGroupCount` or `maxDistance`. See Affecting Learning Node Behavior With Node Parameters on page 36 in *Advanced NuPIC Programming*.

❍ You might rethink your topology as you're working with parameter settings.

Representing Data

You must represent the data to match the sensor node, which receives input for the HTM Network. You need to prepare multiple datasets: training data, category data (if available), and testing data. Here are a few points to consider:

❍ Make sure the data match the `VectorFileSensor` precisely. The `VectorFileSensor` expects a file of vectors, either in `.txt` or `.csv` format.

  — The text file (`.txt`) has one vector per line, separated by spaces.

  — The csv file (`.csv`) has one vector per line, separated by commas.

  When you create a sensor, it expects a text file by default. To submit a csv file instead, set the `file_type` third argument, which defaults to text file, to 3, as follows:

```
sensor.execute('loadFile', trainingFile, 3)
```

  If your data source does not produce vectors as output, manipulating the data so that they fit the available sensors is usually the best solution. Developers with source licenses might consider creating a custom sensor using the node plug-in API.

❍ For the training dataset, create an input file and, if category information is available, a category file. For each line in the data file, you need a corresponding line in the category file. When both files are fed to the HTM Network, it matches data and categories.

  If no category information is available for a problem, the system groups data using the spatial and temporal information it can abstract from the data.

  In the Waves example, a category file that represents the four river state categories is presented to the classifier node to allow that node to discriminate between the different river state categories.

❍ Training data must match a real temporal sequence in the world. For example, the images sample does not look at images statically but scans each image. In contrast, if you feed in images of a cat, a dog, and a cow, there is no correlation between those images.

❍ For the testing dataset, make sure the data are new to the system in interesting ways. For example, when training a vision system using photos of objects, include a photo of the same type of object but a different specific object (for example a teapot that's shaped slightly differently from the teapot you used for training). You might also present pictures of the object from different angles.

  Testing data does not have to be temporally correlated; you can submit snapshots to the trained HTM Network for analysis.

  For the Waves example, you can introduce a different random seed, which means each monitor gets different noise samples. You can introduce different amounts of thermal noise and experiment with how much noise the system can tolerate before it can no longer assign the data to categories.

Data in the Waves Example Programs

The `WavesData.py` script generates data that are a sum of moving Gaussian curves plus a vertical offset. By default, each data time slice consists of 32 data points. There are four data categories; they differ in the numbers and signs of the curves. All curves have identical widths. Some curves have negative weight, some positive weight, and the number of curves ranges from 1 to 3. The centers of the curves are offset from each other by a little bit. The curves drift together at the same rate, which is 0.5 by default. The script saves the data in `.txt` format for easy submission to `VectorFileSensor`.

The goal is to have the HTM Network look at a set of data — representing the river in a certain state — and to determine whether the river is in one of the predefined four states. This categorization is trivial for a human to do, because it is easy to see how many humps are facing up and facing down, so it might also work well for the HTM system.

The Waves example data generator allows you to add two types of noise:

❍ Spatial noise — In each class, the center of each Gaussian curve is given a random horizontal offset. The generator chooses the offset based on the `spatialNoise` parameter. The offset is chosen uniformly in the range `[-spatialNoise/2, spatialNoise/2]`.

❍ Thermal noise — A small amount of noise is added to each data point. The amount is controlled by the `thermalNoise` parameter, and is chosen uniformly in the range `[0,thermalNoise]`.

The data generator also allows you to set the amplitudes of each Gaussian curve with an `amplitudes` parameter. Changing this parameter affects the height and sign of each Gaussian curve. If you change the `centers` parameter, the Gaussian curves have different locations. Other parameters affect data generation as well.

Here's the code fragment from the `RunOnce.py` script. The script creates one set of data for training the HTM Network and one set for testing. For each set, you specify the number of temperature monitors and the thermal noise (which differs between training and testing).

```
print "\n====== GENERATING DATA ==========="
trainData = WavesData()
trainData['prefix']='train_'
trainData['sensorDims'] = numTempMonitors
trainData['thermalNoise'] = trainThermalNoise
trainData['spatialNoise'] = trainSpatialNoise
trainData.createData()

testData = WavesData()
testData['prefix']='test_'
testData['randomSeed'] += 1
testData['numSeqPerCat'] = 4
testData['sensorDims'] = numTempMonitors
testData['thermalNoise'] = testThermalNoise
testData['spatialNoise'] = testSpatialNoise
testData.createData()
```

`RunOnce.py` includes a `visualize.Data()` method that allows you to view the data for any parameter combinations you choose.

Data Representation Questions

Assembling the data for training and testing the HTM requires organizing data correctly and ensuring that the data have the required spatial and temporal characteristics.

Consider the following questions when you prepare your data:

❍ How much data are appropriate? Too much data generally doesn't hurt, although it might slow down processing. Too little data might not be enough for learning. Experimentation is usually necessary but it's better to err on the side of having too much data.

❍ Is preprocessing required? You often have to preprocess source data before you can feed it to the HTM Network. In particular, the data need to be presented in temporal sequences. In the bitworm example, the bitworms are "walking" left to right and right to left over time. See The Bitworm Example on page 14 for a discussion of that problem. In the Waves example, the hot and cold spots are moving down the stream.

❍ Do I need category data? If you submit category data, the classifier can use that information to group the data. If the model domain includes clearly defined categories, you can feed a category file to the HTM Network as part of training. This allows the classifier node to categorize the groups that the rest of the network has found. If the data don't allow easy categorization, the HTM Network performs grouping but cannot assign the output to predefined categories.

❍ Should I consider data generation? If source data are incomplete, or if you want to start developing your HTM Network prototype with a small, easily controlled set of data, data generation makes sense. You might use only generated data, as in the bitworm and waves examples, or modify existing data in different ways for a more complete training set.

  If you already have a set of training data, you can generate a richer data set based on those data. Consider that humans learn to recognize objects at various distances, from different angles, and under different lighting conditions. Similarly, you can feed the HTM Network a comprehensive sample set for a vision problem by changing the lighting in the training images, blurring the images, and zooming in or out.

Designing and Creating the HTM Network Structure

Designing, configuring, and running the HTM Network are iterative steps. You usually start with a design that includes a sensor, a category sensor, and an effector and decide on the number of learning node levels. You then decide how many nodes you want at each level, and on any node parameters.

As part of analysis and testing, it might make sense to experiment with a different number of levels or to change other characteristics of the HTM Network.

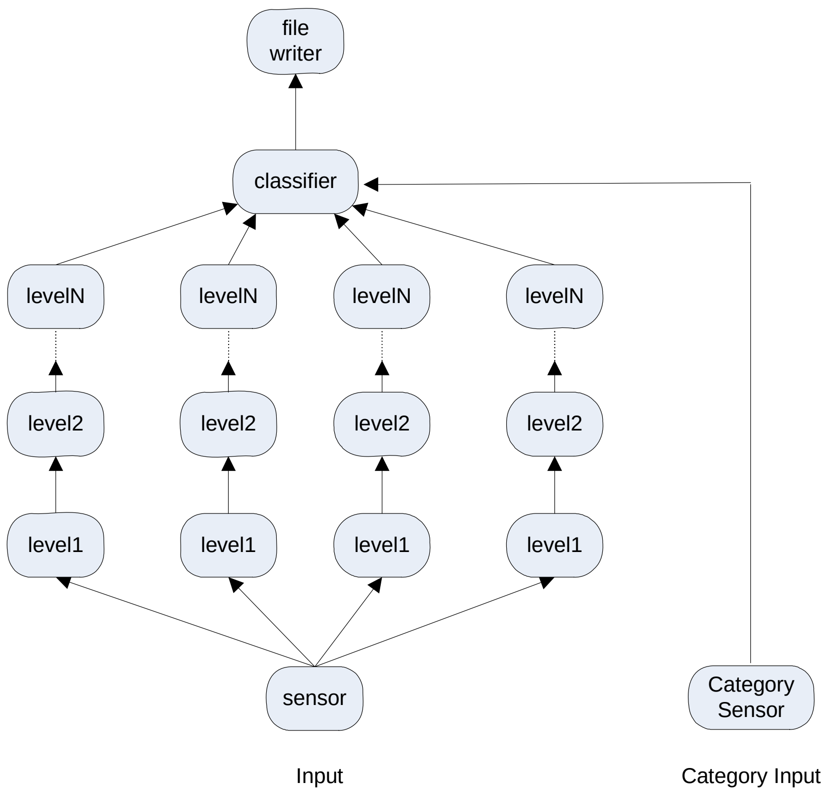
Network Creation with the Network API and Helper Functions

You can use the `AddSensor()`, `AddLevel()` and `AddClassifierNode()` helper functions to create most basic networks. The functions create an HTM Network with these elements:

If you call...	Helper functions use these nodes by default...
`AddSensor()`	`VectorFileSensor`
`AddLevel()`	`SpatialPoolerNode` / `TemporalPoolerNode` pairs
`AddClassifierNode()`	`Zeta1TopNode`

Using the helper function makes sense if the HTM Network structure is simple.

Figure 7 Structure of Network Created with Helper Functions



The helper functions support the following network structure:

❍ One sensor, with input going into each of the bottom-level nodes.

❍ One category sensor with category input going into the classifier node.

❍ One or more levels, linked directly. The levels can have different numbers of nodes.

❍ A classifier node that processes input from the previous level.

Network Creation with Node Constructors

When the default node type is not appropriate, for example, when you're using a non-default learning algorithm you can use node constructors. For example, if you want to use a non-default classifier node, you first create that node using a node constructor, then add the node to the network using `AddClassifierNode()`.

```
knnNode = CreateNode('py.KNNClassifierNode')
AddClassifierNode(myNet, numCategories = 4, nodeInstance = knnNode)
```

The `Waves` example does not use node constructors because the helper functions are appropriate for the desired structure.

The `net_construction` folder includes examples for many different types of HTM Network creation.

Network Creation in the Waves Example

The Waves example creates the HTM Network structure as follows:

1. Create the network.

   ```
   net = Network()
   ```

2. Add a sensor to the network, setting the vector length for the sensor to the number of temperature monitors.

   ```
   AddSensor(net, featureVectorLength = numTempMonitors)
   ```

3. Add each level and set appropriate parameter size, requested coincidences, and groups. The example adds levels using a loop.

   ```
   for level,size in enumerate(levelSize):
      AddLevel(net, numNodes = size,
      requestedCoincidences = coincidences[level],
      requestedGroups = groups[level])
   ```

4. Set parameters for experimentation. To have your HTM Network run well, you usually have to experiment with parameters.

   ```
   levelSpatialName, levelTemporalName = GetLevelNames(net, level+1)
   net[levelSpatialName].maxDistance = maxDistance[level]
   net[levelTemporalName].transitionMemory = transitionMemory[level]
   ```

   The example uses the parameters most likely to affect your HTM Network. See Affecting Learning Node Behavior With Node Parameters, page 36 in *Advanced NuPIC Programming* for a complete list.

When you add a level, you add two regions: one region of spatial pooler nodes and one region of temporal pooler nodes. Pairs of nodes are connected by `SimpleLink`.

5. Finally, add a classifier node to the HTM Network.

```
AddClassifierNode(net, numCategories = trainData['numCat'])
```

Network Design Questions

When designing your HTM application, consider the following questions.

❍ What are the best values for node parameters? Some experimentation is usually necessary. See Affecting Learning Node Behavior With Node Parameters, page 36 in *Advanced NuPIC Programming* for a description of each node parameter.

❍ What is the best way to arrange nodes (how many levels, how are nodes linked, etc.)?

❍ How many groups/coincidences do you expect for each node? You don't have direct control over the number of groups and coincidences, but HTM Network geometry affects those numbers. You might redesign parts of the HTM Network or manipulate the input data if the number of groups and coincidences is too small or too large.

❍ How much time is needed for training, and how can training time be optimized? Training time depends on a number of factors including HTM Network size, amount of data, training strategy, number of CPUs, speed of CPUs, and more.

There are no easy answers to these questions. In many cases, you need to experiment with the available options. For example, you might run your HTM with a different number of hierarchy levels, then compare the results. The Numenta website includes discussions of those topics.

Running the Network to Perform Learning and Inference

During training, an HTM Network performs learning and inference, one level at a time. The `RuntimeNetwork` API and the `RunBasicNetwork()` helper function turn learning and inference on and off automatically.

This section gives only a brief introduction to the topic.

❍ Inside a Learning Node: How Learning and Inference Happen on page 31 in *Advanced NuPIC Programming* gives some additional information

❍ The white paper *The HTM Learning Algorithms* discusses HTMs in general in depth.

❍ The document *Numenta Node Algorithms Guide* discusses the learning algorithms.

When you construct your HTM Network using `AddLevel`, the tool adds a region of `SpatialPoolerNode` instances and a region of `TemporalPoolerNode` instances, connected by single links. During learning, the learning algorithms work on that input on a per-level basis.

Here's a simple view of this process:

1. First, the system performs learning at the lowest level (Level 1). Level 1 nodes are in learning mode, and all nodes above Level 1 are disabled.
2. Next, the system performs inference at the lowest level (Level 1) and learning at the next level (Level 2).

   The Level 2 `SpatialPoolerNode` and `TemporalPoolerNode` are in learning mode. While Level 2 is being trained, Level 1 must be in inference mode, that is, its grouped data must be available to Level 2.
3. The system performs inference at Level 2 and learning at the next level (Level 3).

   — The Level 2 `SpatialPoolerNode` and `TemporalPoolerNode` are in inference mode

   — The Level 3 `SpatialPoolerNode` and `TemporalPoolerNode` are in learning mode.
4. The system continues performing first learning, then inference at each level until it reaches the classifier node. The classifier uses the data from the category sensor assign the data from the previous level to categories. If there is no category sensor, the classifier groups according to how the data are structured.
5. At the end of the run, each node is trained and in inference mode, ready to process more data.

Training in the Waves Example

The `Waves` example allows you to perform training and testing in one of two ways:

- **RunOnce.py —** The `RunOnce.py` script calls the `TrainBasicNetwork` function passing in the network structure and the names of the data and category files. After training, the script saves the HTM Network so the trained network can later be loaded in for experimentation.

  ```
  trainedNet = TrainBasicNetwork(net,
               dataFiles     = ["train_sensor.txt"],
               categoryFiles = ["train_category.txt"])
  trainedNet.save("trained_waves.xml")
  ```

  After the network has been trained, it is called with new data to test the HTM Network. `RunBasicNetwork()` returns the accuracy of the trained network. Accuracy is the percentage of input data that were classified correctly by the HTM Network. To receive accuracy information, you must supply both test data and category data (or accuracy is `None`).

  ```
  accuracy = RunBasicNetwork(trainedNet,
                          dataFiles     = ["train_sensor.txt"],
                          categoryFiles = ["train_category.txt"])
  print "Accuracy of trained network on original training data is ",
     accuracy
  ```

- **RunExperiment.py** — The `RunExperiment.py` script performs training, testing, and data collection using different parameter settings. For best performance, you must experiment with different parameter settings when you train your HTM network. The script illustrates how you can do so.

Training Questions

Before you start training runs, asking the following questions can be useful:

- What are the most effective parameter settings? You might have to experiment with these settings, but you may also know approximately the right values based on the amount of noise in the data. See Affecting Learning Node Behavior With Node Parameters, page 36 in *Advanced NuPIC Programming* for more information.
- What is the best sequence of training data to quickly train the HTM? If you want for the network to learn how to recognize things regardless of certain perturbations, for example, the noise or the position of the worm in the bitworm example, feed in sequences that demonstrate that perturbation.
- Can I reuse some trained nodes? For instance, the Pictures example trains one node, then make copies of that node for all other nodes at that level. See Training Your HTM Network: The Pictures Example on page 64.
- Can I speed up learning by supplying additional category information at certain levels?
- How many nodes per level, and how many levels give the best results? Consider starting with a one-level network and experiment with different parameter settings until the accuracy is optimal. Then train a two-level network using the same parameter settings.

Troubleshooting, Testing, and Analyzing Your HTM System

When you first attempt to run your HTM Network, you might have to do some troubleshooting if things don't work at all. Once the HTM Network runs satisfactorily on the training data set, you might wish to improve accuracy, speed, or both.

A number of tools and scripts allow you to view results and explore how the HTM Network arrived at the result.

❍ NuPIC allows you to access nodes and their parameters and change the node state. The following code fragment accesses the first node in level 1, retrieves its `requestedGroupCount` parameter, then sets the parameter to 3.

```
node = trainedNet(1_Temporal[0])
print node.requestedGroupCount
node.requestedGroupCount = 3
```

❍ Most settings are handled by parameters, but sometimes you need to call a command using the following syntax.

```
node.execute('<command>')
```

❍ Use `nodeHelp` for reference documentation for each node, for example:

```
nodeHelp('TemporalPoolerNode')
```

❍ Use the `Visualizer` tool (`nupic.analysis.visualizer`) for a graphic display of the results that includes the group structure and other information. See Using HTM Network Visualizer on page 74.

It is often useful to examine intermediate results or to experiment with different parameter settings to understand how your HTM application processed input data. Based on that information, you can change how you enter data, change nodes and their parameters, or modify other characteristics of your HTM Network. See Debugging Your HTM Application on page 67.

Testing and Analysis in the Waves Example

The Waves example generates visualizations of the coincidence matrix and group structure when it's invoked. The visualization code generates a file and is commented out by default. The code looks like this:

```
if length == 'normal':
print "Visualizing trained network..."
vis = Visualizer(networkFilename="trained_waves.xml",
                 dataType='bar_graph')
vis.visualizeNetwork(openBrowser=False)
```

To test Waves and view results:

1. First, run the trained HTM network using the existing input data. View the results by looking at the visualizations that are generated.
2. Modify the `RunOnce.py` file to change the input data or parameters. For example, try setting the `testThermalNoise` variable to a higher value. You can also change the `maxDistance` or `requestedGroupCount` parameters, which you can specify on a per-level basis in `RunOnce.py`.

3. Run the HTM network with the new configuration and examine the visualizations.

You can also use NodeInspector to interactively look at results.

Using RunExperiment.py to Explore Results of Changes

The `RunExperiment.py` script in the `Waves/simplehtm` directory illustrates how you can prepare different experiment runs to test the results of manipulating different aspects of the HIM's world. The available experiments include:

Experiment	Description
DifferentTestData	Varies test data offsets and Gaussian amplitudes while measuring accuracy.
MaxD	Varies the training data thermal noise and network `maxDistance` parameters while measuring accuracy.
NodeTypeTestNoise	Tests different inference modes (`dot`, `product`, `productNonUniform`, `productNonUniformNonViterbi`) against the thermal noise parameter in the testing data.
NodeTypeTrainNoise	Tests different inference modes (`dot`, `product`, `productNonUniform`, `productNonUniformNonViterbi`) against the thermal noise parameter in the training data.
TestingNoise	Varies the test data thermal noise while measuring accuracy.
TrainingData	Varies the amount of training data, Gaussian spatial noise of the training data, and Gaussian locations of the test data while measuring accuracy.
WhereItFails	Reports which test data points are correctly and incorrectly classified when using the default parameters.

Testing and Analysis Questions

Tuning your HTM Network is the biggest part of the development process.

❍ What are the outputs at the different levels of the HTM?

❍ How good is the recognition performance on the training data?

❍ How good is the recognition performance on new data?

❍ How do node parameter changes affect accuracy?

Remember that you might have to revisit data generation or submission, or even problem definition to arrive at a useful HTM Network design and implementation.

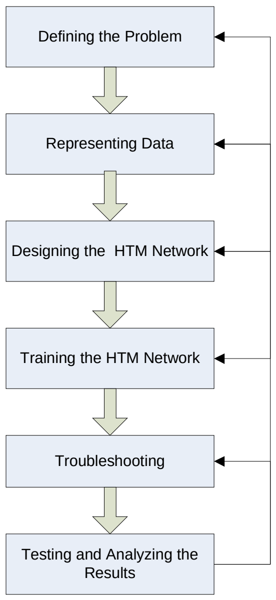
Summary and Guide to Documentation

The process from problem definition to HTM deployment is not linear. At each stage, you might need to return to an earlier stage for redefinition or refinement.

For example, when analyzing the data, you might discover the data gathering or data representation approach needs to change. When designing the HTM Network, you might find that the problem definition needs to be refined. Each stage is directly affected by other stages. You can deploy the HTM Network only when you're satisfied that it can generate satisfactory results for those new data; achieving good results for one set of data is not enough.

The following table summarizes design and implementation tasks and lists relevant documentation for each task. See How to Access NuPIC Built-in Help on page 84 for information on getting help while working with the APIs.

Table 3: Tasks Overview and Documentation

	Task	Documentation (Concepts)	Documentation (Task-based)
 Defining the Problem Representing Data Designing the HTM Network Training the HTM Network Troubleshooting Testing and Analyzing the Results	Defining the Problem	Numenta website.	Numenta website. Look at example code for examples of problem definition.
	Representing Data	Numenta website.	Numenta website. Look at example code for examples of data representation (Pictures example) and data generation (all examples).
	Designing and Creating the HTM Network Structure	Numenta website.	Constructing an HTM Network on page 41. Python online help.
	Running the Network to Perform Learning and Inference	Software Components on page 13 in *Advanced NuPIC Programming.* The *Numenta Node Algorithms Guide*	Running the HTM Network on page 55 Running HTM Networks With Sessions on page 41 in *Advanced NuPIC Programming.* Python online help. Numenta website.
	Troubleshooting, Testing, and Analyzing Your HTM System		Debugging Your HTM Application on page 67.

For information about the learning algorithm, see the *Numenta Nodes Algorithms Guide.* This white paper helps you understand how an HTM Network that uses Numenta learning nodes analyzes your data and how different node parameters affect the learning behavior. This, in turn, might help your understand how to improve your application's effectiveness.

3 Constructing an HTM Network

This chapter explains how you can use the Numenta Python APIs to create an untrained HTM Network.

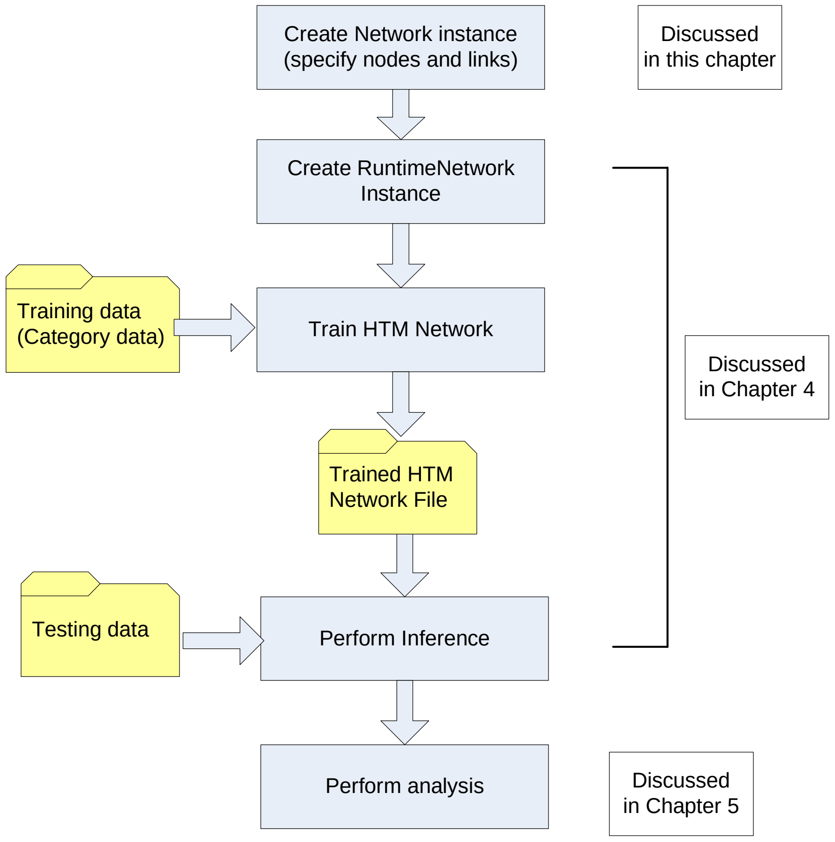
Topics

HTM Programming Steps Overview

This section briefly discusses the complete development process to help you understand the relationship between your HTM Network's structure (encapsulated by a `Network` instance) and the `RuntimeNetwork` that is used during training and inference.

For most basic HTM Networks, you actually don't need to create nodes and links explicitly but can instead using the helper functions to create the structure. Many of the examples, including Bitworm and Waves, use helper functions to add sensors, levels, and classifiers to the network and linking happens automatically.

Figure 8 HTM Development Process Overview


Create Network instance (specify nodes and links)
Discussed in this chapter
Create RuntimeNetwork Instance
Training data (Category data)
Train HTM Network
Discussed in Chapter 4
Trained HTM Network File
Testing data
Perform Inference
Perform analysis
Discussed in Chapter 5

Creating the HTM Network with Helper Functions

The helper module allows users to create, run, and experiment with basic HTM Networks The module is useful for creating and experimenting with initial prototypes. You can also look at the code for the helper module (`helpers.py`) as example code because it uses the explicit network and node creation functions to define the helper functions.

You cannot create image recognition networks with the helper functions. See the Vision Framework for those tasks.

Network Structure Supported by Helper Functions

A basic network is strictly defined to have the hierarchical structure shown in Figure 7 on page 33. Each level feeds into the next and each level can have a different number of nodes. There can be 0 or more levels, but there is always a single classifier node at the top. There can be overlapping connections between levels.

Nodes Supported by Helper Functions

The helper functions support the following nodes:

❍ Two sets of learning node types:

— `AddLevel()` adds a `SpatialPoolerNode` combined with `TemporalPoolerNode`

— `AddZeta1Level()` adds a `Zeta1Node`. This node type is used in the Bitworm example and includes both a spatial and a temporal pooler.

You cannot mix these two sets in the same network.

❍ A sensor node that reads input data from files containing space-separated or comma-separated numbers.

❍ A category sensor node that reads category data for each input vector from separate files.

Creating the HTM Network Components Explicitly

The rest of this chapter explains how you can create an untrained network (the Network object and all associated objects). In some cases, the examples use helper functions, but in many cases they don't. Here's an overview of the steps:

Table 4: Overview of Network Creation Tasks

Step	Task	Discussed in
1	Create the empty HTM Network	Creating the Network and Node Objects, page 47
2	Create and configure nodes, add them to the HTM Network OR Create a template nodes, then create regions, add them to the HTM Network	Creating the Network and Node Objects, page 47 Creating Regions on page 49
4	Link network elements	Linking the Elements of the Network on page 51

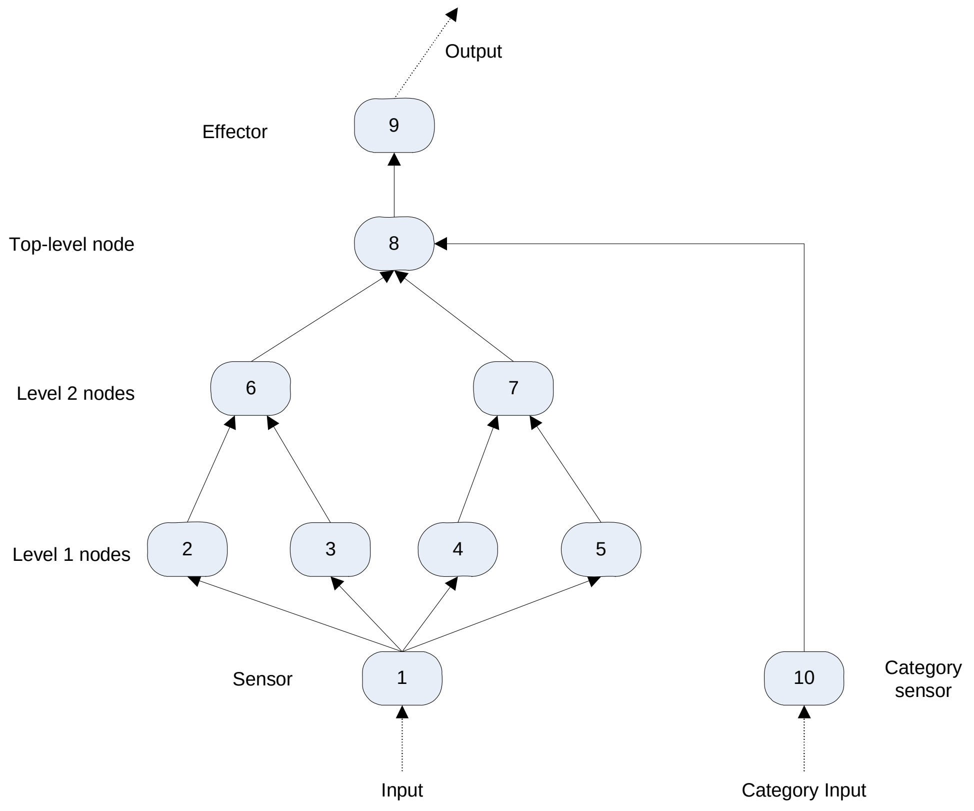
Understanding HTM Network Components

HTM Networks are data-driven learning systems.

The untrained network has no inherent capability and cannot be used to perform inference. The untrained network is like a scaffold that describes the flow of data and other aspects of analysis at runtime. This scaffold information includes, for example, the number of nodes and levels, characteristics of the nodes, and how the nodes' inputs and outputs are connected.

Figure 10 shows a sample HTM Network. While the sample network shown below is small and highly symmetrical, an HTM Network can actually have an arbitrary number of levels and connections, and it does not need to be symmetrical.

Figure 9 Sample HTM Network



Nodes

A node — sensor, effector, or learning node — is the basic unit of a Numenta HTM Network. Each node is implemented as a Python or C++ plugin to NuPIC. Numenta nodes are usually created and manipulated using classes in the NuPIC Python API, as discussed in Constructing an HTM Network on page 41.

Node Categories

Basic node categories are:

- ❍ Sensors — take input data or category data and make them available to learning nodes.
- ❍ Learning nodes — take data from another learning node or from a sensor and model the data based on the settings of certain parameters.
- ❍ Effectors — Take data from a learning node and send them to the outside world.

Node Attributes

Each node has a number of attributes.

Attribute	Description	See

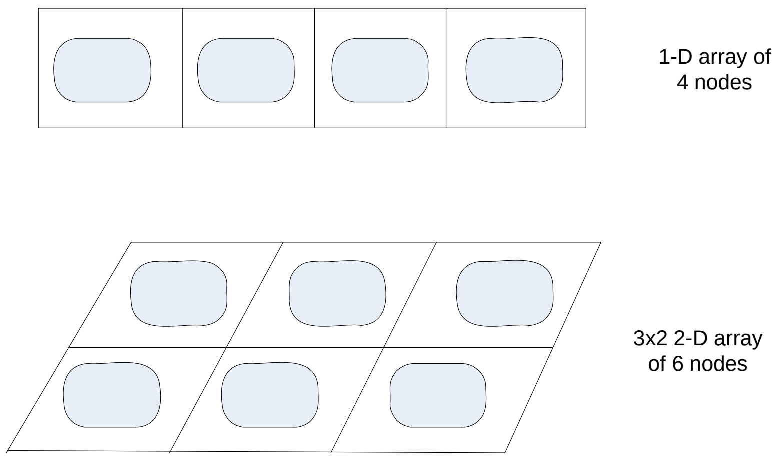
NodeHelp	Displays information about each node, for example:
	`from nupic.network import *`
	`nodeHelp('SpatialPoolerNode')`
		Getting Help for Nodes and Helper Functions on page 84

Inputs and outputs	Vectors of floating-point numbers in the nodes that are currently implemented. Sensors have no inputs but at least one output. Learning nodes have at least one input and output each. Effectors have no outputs but at least one input.	Node Inputs, Node Outputs and Links on page 26 in *Advanced NuPIC Programming*
Links	Connect nodes. A node can receive input from and send output to nodes to which it is linked. The helper functions link nodes automatically.	Linking the Elements of the Network on page 51
Parameters	Affect node behavior at runtime.	Affecting Learning Node Behavior With Node Parameters on page 36 in *Advanced NuPIC Programming* and the *Numenta Node Algorithms* white paper.
Scheduling (phase) information	determines the order in which the nodes are processed. The NRE processes each phase in sequence. You specify phase information when using node constructors or regions.	Scheduling Node Processing on page 57 in *Advanced NuPIC Programming*
Internal state	Most of the internal state is saved when the HTM Network is saved after training. Some node data, such as the current input and output vectors, are transient and are not saved.	

Regions

A region is an array of nodes that have precisely the same parameters, including the same phase. If you use the helper functions to create your HTM Network, they internally create a region for each set of nodes. You can also create regions explicitly. Many larger HTM Networks include regions. See Creating Regions on page 49.

Figure 10 1-D Region of Four Nodes (Top) and 2-D Region of 2x3 Nodes (Bottom).



Links

Links determine the flow of information in the HTM Network. You can link individual nodes, or link arrays of nodes at once using link types. See Link Types on page 51.

Creating the Network and Node Objects

You use the `Network` class to create the initial untrained HTM Network. The untrained network describes the network structure, similar to the way a database schema describes the structure of a database. Creating the HTM Network with Helper Functions on page 42 explains how to create most basic networks. This section discusses a network using node objects, which offers more flexibility. The process consists of these steps, discussed with code fragments below:

1. Import the nupic.network Python package
2. Create an Empty Network, page 47
3. Create and Configure the Nodes, page 48
4. Add the Nodes to the Network Object, page 48

After you've completed these steps, you can link the different nodes, as discussed in Linking the Elements of the Network on page 51.

1 Import the nupic.network Python package

The `nupic.network` package contains all the classes you need for creating your network. It usually makes sense to import all classes in the package as well as the helpers package, as follows:

```
from nupic.network import *
from nupic.network.helpers import *
```

Here are some of the classes and functions from `nupic.network`:

`Network`	Container class. You can create an instance of `Network`, and then add the network elements (nodes or regions).
`CreateNode`	Used to create nodes explicitly using the node class and its methods. See NuPIC Node Types on page 82 for an overview of node types. See Affecting Learning Node Behavior With Node Parameters on page 36 in *Advanced NuPIC Programming* for an introduction to node parameters.
`Region`	Use to create arrays of nodes based on a template. See Creating Regions on page 49.

For information about any of the classes, you can use pydoc, as discussed in Getting Help for Nodes and Helper Functions on page 84.

2 Create an Empty Network

After you've imported the appropriate classes, you can create an empty network to serve as container for network elements and links. For example:

```
myNet = Network()
```

3 Create and Configure the Nodes

The `CreateNode()` function allows you to create and configure a node with one call.

For each node, you can specify a node type and all node parameters. Some parameters are required, as indicated in the online help. If you make a mistake, `CreateNode()` warns you immediately by throwing a Python exception.

The following code fragment creates a node using `CreateNode()` and specifies its parameters.

```
CreateNode("SpatialPoolerNode",
   phase=1,
   bottomUpOut=200,
   maxDistance=0.05 ),
```

4 Add the Nodes to the Network Object

After you've created the node(s), you can add them to the network as follows:

```
myNet.add ("<unique_element_name>", myNode)
```

Here,

❍ `unique_element_name` is the name for the node or region,

❍ `myNode` is the variable to which you assigned the node earlier in the program.

For example:

```
myNet.add ("Level1Node", myNode)
```

You must remember the name you assign in this call and use it later when linking nodes or regions.

`Network.add()` adds an element reference to the network. Modifications to the original affect the element inside the HTM Network. If you prefer to add a copy, use `addElement()`.

Creating Regions

Creating a region involves a few simple steps.

1. Create a Node that Serves as the Template.

   Use `CreateNode()` to create a node template. To create an HTM Network that uses the Numenta learning algorithm, create a `SpatialPoolerNode` first and set all parameters as part of node creation.

   ```
   myTemplateNode = CreateNode(nodeType = "SpatialPoolerNode", ...)
   ```

2. Create a Region Using the Template and Specifying the Dimensions.

   You create a 1-D region that consists of four nodes as follows:

   ```
   myRegion = Region (4, myTemplateNode)
   ```

   You create a 2-D region of six nodes arranged in a grid with two rows and three columns (see Figure 10 on page 46) as follows:

   ```
   myRegion = Region ([2,3], myTemplateNode)
   ```

3. Add the Region to the Network.

   Adding a region to an HTM Network is identical to adding a node to a network.

   ```
   myNet.add("<unique_region_name>", myRegion)
   ```

After you've completed these steps, you can link the region to other regions or to nodes, as discussed in Linking the Elements of the Network on page 51. If you want to create an HTM Network that uses the Numenta learning algorithm, you can next create a `TemporalPoolerNode` template node, create a region, and link the spatial pooler and temporal pooler regions.

The `LogoExampleRegions` example in the `net_construction` example set illustrates creating regions.

Node Names in Regions

All the node names for a 1-D region are `<region_name>[<num>]`

where `<num>` is a number between 0 and the number of nodes in the region, minus 1.

All the node names for a 2-D region are `<region_name>[num,num]`

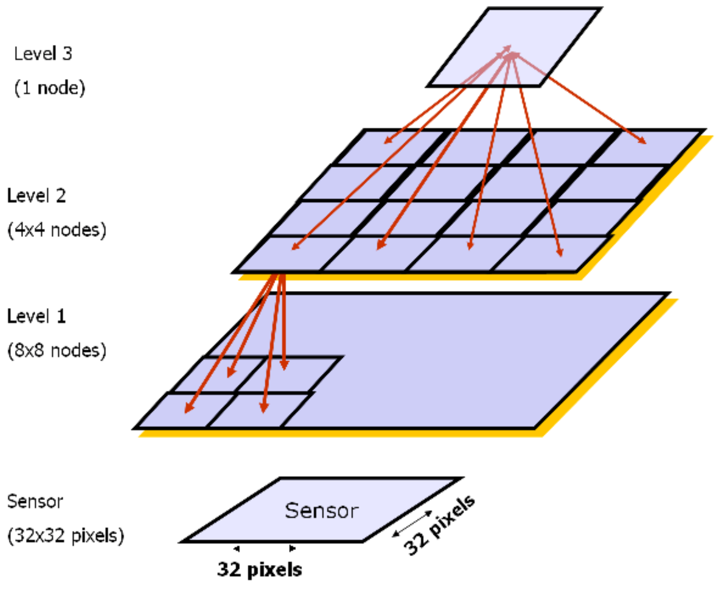
where the two numbers are the indices into the region. Note that there is no space between the two numbers.

You can use node names when setting up more advanced links, or when analyzing individual nodes in a trained network.

Region Example

Figure 11, Regions represented by the Pictures Example, shows an example of regions.

Figure 11 Regions represented by the Pictures Example



Linking the Elements of the Network

After you've added nodes to a `Network`, you must link the nodes (unless you use the helper functions, which link nodes automatically). In the simplest case, you can link the nodes as follows:

```
myNet.link(<source_node>, <dest_node>)
```

That call assumes a single link, and assumes a single source output and a single destination output.

Each node can have multiple named outputs and named inputs. You can therefore perform linking with the following arguments:

- ❍ Link without overlap:

```
myNet.link ("<source_node_name>",
         "<dest_node_name>",
         linkType= <link_type>,
         sourceOutputName= <source_output>,
         destinationInputName= <dest_input>,
         sourceDims= <dims>)
```

- ❍ Link with overlap:

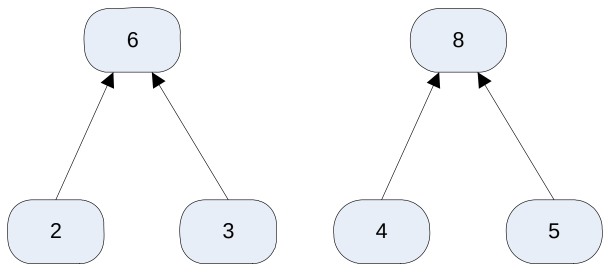
```
myNet.link ("<source_node_name>",
         "<dest_node_name>",
         linkType= <link_type>,
         sourceOutputName= <source_output>,
         destinationInputName= <dest_input>,
         sourceDims= <dims>,
         overlap= <overlap>)
```

The rest of this section discusses the different link types you can use, both with and without overlap.

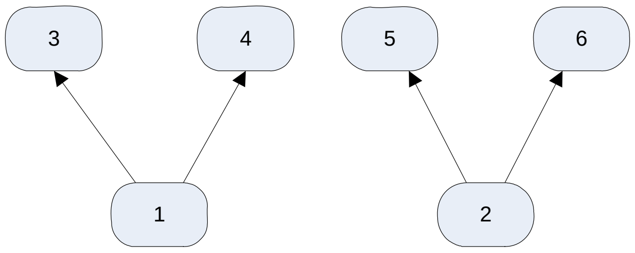
Link Types

Three link types are defined:

- ❍ **SingleLink** — default connection between two nodes.
- ❍ **FanIn** — For region to region linking that divides the nodes evenly. There are a number of options:
  - — Multiple bottom-level nodes send their output to a higher-level node, where the output is concatenated, as in the following illustration:



— A lower number of lower-level nodes sends their output to a higher number of higher-level nodes. In that case, each higher-level node receives the complete output from the lower-level node.

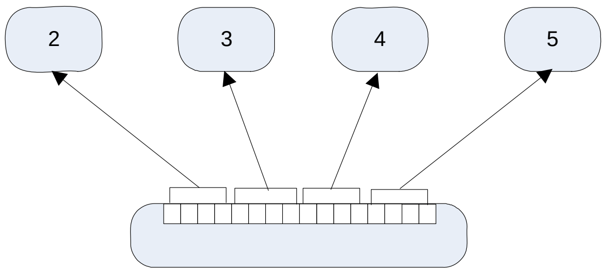


— 2-D Regions — When two 2-D regions are linked, the nodes at the bottom level are divided evenly to send their output to the next level. For example, if you have 8x8 (64) nodes at the bottom level, and you have 2x2 (4) nodes at the next level, each higher level node gets input from 16 lower level nodes arranged in a 4x4 grid.

To link two regions, use code like the following:

```
myNet.link ("<source_region_name", "<dest_region_name>", linkType = <link>)
```

❍ **SensorLink** — When you want to connect a sensor to a region, the goal is usually to divide the sensor output array evenly across all nodes in the region. When you use `SensorLink`, the system takes care of this automatically for 1-D, 2-D, or 3-D arrays.



For example, if the sensor output is an array of 64 elements, and the region consists of four nodes, the first 16 data values go to the first node, the next 16 data values go to the second node, and so on.

To link a sensor with a 1-D region, use code like the following:

```
myNet.link ("sensor",
            "level1",
            linkType="SensorLink",
            sourceOutputName= "dataOut",
            destinationInputName="bottomUpIn",
            sourceDims=[18])
```

To link a sensor with a 2-D region, use code like the following:

```
myNet.link ("sensor",
            "level1",
            linkType="SensorLink",
            sourceOutputName= "dataOut",
            destinationInputName="bottomUpIn",
            sourceDims=[18])
```

`sourceDims` is a list, e.g. [32, 32] if the sensor outputs 1024 elements that should be treated as a 32x32 array.

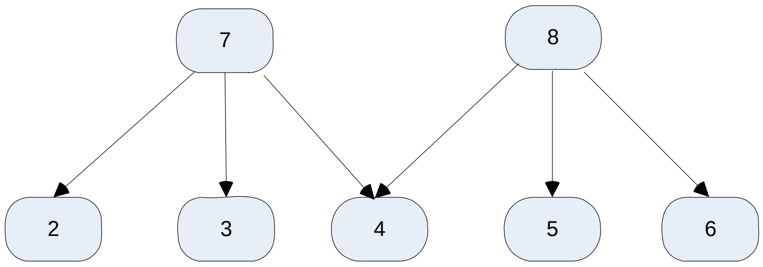
Overlapping Connections

To specify overlaps, use the `SimpleLink` link type or the `FanIn` link type with an overlap argument.

For example, you can link nodes at two levels using a specified overlap, as follows:

```
link ("level1", "level2", linkType="FanIn", sourceOutputName="bottomUpOut",
destinationInputName="bottomUpIn, overlap=[1])
```

Figure 12 FanIn Example with Overlap

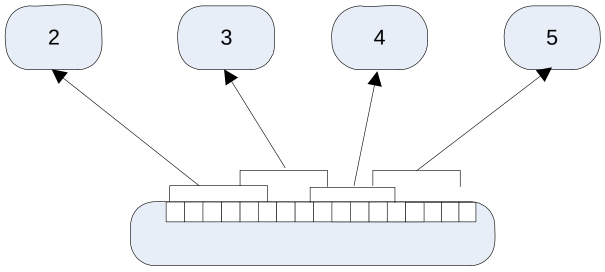


The system calculates the fan-in using the source element (or region) dimension, destination element (or region) dimension, and overlap. The overlap, which is a list to allow for multi-dimensional HTM Networks, specifies the number of source nodes shared by any two adjacent destination nodes.

You can use `link` with `SensorLink` link type and an `overlap` argument to divide a single sensor output array equally across the nodes of the destination region, for example:

```
link ("sensor", "level1", linkType="SensorLink", sourceOutputName= "dataOut",
destinationInputName="bottomUpIn", sourceDims=[18], overlap=[2])
```

Figure 13 FanIn Example with SensorLink



In this example, the sensor output array has 18 array elements, which are divided across four nodes. Each node receives the output of six array elements, with adjacent nodes getting the input from two shared elements each.

4 Running the HTM Network

This chapter discusses the tasks involved in running an HTM Network. It explains how you can use the `RuntimeNetwork` API to invoke the NRE and looks at some examples.

Simple HTM Networks such as Bitworm and Waves use the `TrainBasicNetwork()` and `RunBasicNetwork()` helper functions to run the network.

Topics

Understanding Learning and Inference

This section gives an overview of the training and inference processes.

What Happens During Training

During training, the NRE performs first learning and then inference on nodes at each level in the HTM Network. The training process consists of these steps:

1. The script creates a `RuntimeNetwork`. `CreateRuntimeNetwork()` expects an untrained HTM Network as the argument (usually a previously saved HTM Network file). `CreateRuntimeNetwork()` also expects training files that you must make available to the NRE.
2. The script initializes the sensor with input data from a file. In many cases, the script also initializes the `CategorySensor` with a file.
3. The script calls `RuntimeNetwork.run()` for each level of the HTM Network using a run policy. Here's an example from the `waves/runtimeNetwork` example.

   ```
   # Train Level 1
   runtimeNet.run(TrainPhase("level1", n), ["sensor", "level1"])

   #Train Level 2
   sensor.setParameter("position", 0)      # set to beginning of file
   runtimeNet.run(TrainPhase("level2", n), ["sensor", "level1", "level2"])
   ,...
   ```

   Training is done one level at a time:

   a. First, the NRE trains the bottom level using the input data.

   b. Next, the NRE performs inference on the bottom level, sends the resulting groups to Level 2 and performs training on Level 2.

   c. Next, the NRE performs inference on Level 2 and performs training on Level 3, and so on.

   The script uses a run policy which handles the details of setting each level's mode (learning or inference).

   For additional information, see the nodeHelp for `RuntimeNetwork`.
4. If a category file was submitted, the top-level node uses the category information during learning.

What Happens During Inference

During inference, you submit data to a trained HTM Network. All nodes are in inference mode, that is, each node has a model of its world and is ready to group or classify incoming data based on that model.

Options for Running the HTM Network

You can run your HTM Network in several ways.

Helper Functions

For simple networks, you can use the `TrainBasicNetwork()` and `RunBasicNetwork()` helper functions, as illustrated in the Bitworm example and discussed in Chapter 1.

RuntimeNetwork API

For more complex networks, you can use the `RuntimeNetwork` class directly, as discussed in this chapter.

❍ You can use the `RuntimeNetwork` class to submit training data and an untrained HTM Network to the NRE.

  After the training run, each node in the HTM Network knows about part of the domain and the trained HTM Network is a model of the domain world. You can save the trained HTM Network to a file.

❍ During inference, the HTM Network uses the learned model to process inputs one level at a time. You use the `RuntimeNetwork` class to load the trained network into the NRE, specify the input and output data, and run the network.

❍ During analysis and debugging, you use the `RuntimeNetwork` class to load the trained network into the NRE and then query the nodes for information about their trained state.

Session API

When running remotely or in a multiprocessing environment, advanced users might need to use the `Session` API discussed in Running HTM Networks With Sessions, page 41 in *Advanced NuPIC Programming*.

Understanding the RuntimeNetwork API

`RuntimeNetwork` allows you to train and test your HTM network and to perform additional analysis on individual nodes. You can instantiate `RuntimeNetwork` using its constructor, or call the `TrainBasicNetwork()` and `TestBasicNetwork()` helper functions as in the bitworm example. You have to instantiate `RuntimeNetwork` explicitly rather than using helper functions if you want to exclude levels or select levels to run.

`RuntimeNetwork` launches the NRE, loads an HTM Network, and provides an interface that allows you to manipulate the HTM Network at runtime. If you need more direct control over communication with the NRE you can use the `Session` API, discussed in Running HTM Networks With Sessions, page 41 in *Advanced NuPIC Programming*.

A `RuntimeNetwork` instance has these components.

❍ A `Network` object that contains the nodes, links, and regions that define the structure of the HTM Network (see Constructing an HTM Network on page 41). When you create a `RuntimeNetwork`, you specify the `Network` explicitly or specify an HTM Network XML file for loading the network.

❍ A `Session` object that interacts with the Numenta Runtime Engine. When you use `RuntimeNetwork`, you don't need to specify the `Session` explicitly. Instead, `RuntimeNetwork` creates a session and manages it for you. See Sessions and NRE Supervisor, page 19 in *Advanced NuPIC Programming* for more information.

❍ A temporary directory called a session bundle that stores files needed by the NRE.

The `RuntimeNetwork` object has the following associated function and methods:

Table 5: Using RuntimeNetwork

Function/Method	Description
`CreateRuntimeNetwork`	Creates a `RuntimeNetwork` instance. Takes as arguments the name of a network file, additional files required by the NRE, and any NRE configuration options. `runtimeNet = CreateRuntimeNetwork (<filename>, [files = ...)`
`RuntimeNetwork.run()`	Initiates the computation of selected HTM nodes in the NRE and waits for that computation to complete. See What RuntimeNetwork.run() Does, page 59.
`RuntimeNetwork.getElement()`	Accesses nodes or regions at runtime. See Accessing Individual Elements, page 59.

`RuntimeNetwork.getParameter()`
`RuntimeNetwork.setParameter()`
`or`
`node.<parameterName>`
	Gets or sets accessible parameters on HTM nodes in the NRE.

`RuntimeNetwork.execute()`	Sends node-specific commands to individual nodes or regions in the NRE.

For detailed information you can use help from the Python command prompt. See Getting Help for Nodes and Helper Functions on page 84.

Accessing Individual Elements

While the NRE processes your HTM Network, you can access individual elements by calling `RuntimeNetwork.getElement()` or `RuntimeNetwork.<elementName>()`.

For example, you can load and initialize the sensors that were defined in the HTM Network file.

1. First you access the element. You can use `getElement()` or address the element directly.

   — `getElement()` syntax:

   ```
   sensor = runtimeNet.getElement ("CategorySensor")
   categorySensor = runtimeNet.CategorySensor
   ```

   — Addressing the element directly:

   ```
   sensor = runtimeNet.Sensor
   categorySensor = runtimeNet.CategorySensor
   ```

2. Then you can work with the element:

   ```
   sensor.execute("loadFile", trainingFile, "0")
   categorySensor.execute("loadFile", trainingCategories, "0")
   ```

What RuntimeNetwork.run() Does

When you call `RuntimeNetwork.run()`, the NRE runs computation for the specified number of iterations using optional selection or exclusion arguments.

If no selection is specified, the entire network is enabled. You can disable individual elements using the `exclusion` parameter. Selection and exclusion arguments can refer to individual nodes or to regions.

NuPIC supports a number of node types and learning algorithms. Different learning nodes might require a `RunPolicy` instead of the number of iterations during training. For example, the basic learning nodes, `SpatialPoolerNode` and `TemporalPoolerNode` require level by level training.

The following table shows some examples:

Example	Description
`myNetwork.run(1000)`	Run the entire network for 1000 iterations
`myNetwork.run(1000, ["sensor", "level1"])`	Run the `sensor` node and the region `level1` for 1000 iterations
`myNetwork.run(1, exclusion=["effector"])`	Run the entire network, except for `effector`, for one iteration

Example	Description

`myNetwork.run(TrainPhase("level2", 1000),`
`            selection = ["sensor",`
`"level1", "level2"])`
	Train the `level2` region for 1000 iterations using the constructor for the `TrainPhase` run policy.
	Nodes with names "sensor" and all nodes in the `level1` and `level2` regions will be selected. Learning will be turned on for all nodes in `level2`.

Using the RuntimeNetwork Object to Perform Training

During training, each enabled node in the HTM Network reads its inputs and updates its internal model. Bottom-level and mid-level nodes perform pooling of patterns, but do not associate their groups with category labels. Over time, the HTM learns that certain patterns appear frequently and together in time.

If the learning nodes in your HTM Network are `SpatialPoolerNode` and `TemporalPoolerNode`, you can use the `LevelTrain` run policy during training. `LevelTrain` is a run policy specific to the basic learning algorithm, which requires that you train your network one level at a time. When training level L, the nodes at level L+1 and higher must be disabled. Nodes at levels L-1 and lower must be in inference mode. The `LevelTrain` policy handles this logic.

The following example uses `LevelTrain` to train an HTM Network.

1. Train your network one level at a time. For example, for a two-level network:
   a. Call `run()` to perform training on `Level1` nodes:

```
runtimeNet.run(LevelTrain("Level1", numVectors),
               selection=["Sensor", "Level1"])
```

   b. Reset all sensors to make sure they start at the beginning of the file.

```
sensor.setParameter("position", "0")
categorySensor.setParameter("position", "0")
```

   c. Call `run()` to perform training on `Level2` nodes. During this run, the `Level2` nodes take the information from the trained `Level1` nodes as input.

```
runtimeNet.run(LevelTrain("Level2", numVectors),
                           selection = ["Sensor", "Level1", "Level2"])
```

2. Save the trained network file and clean up.

```
runtimeNet.save(trainedNetworkFilename)
runtimeNet.cleanupBundleWhenDone()
```

Saving the Network to a File

NuPIC allows you to save untrained and trained HTM Networks. Because creating the HTM Network is just as fast as loading an untrained HTM Network, untrained networks are saved only rarely. However, because training can be time consuming, it makes a lot of sense to save a trained network.

When you export a network to an HTM Network File on disk, the system creates an XML file that contains information on all the nodes and links in the HTM Network.

```
myNet.save("<untrained_htm_network_filename>.xml")
```

When you call `save()`, the system saves the HTM Network to an XML file (extension `.xml`) using custom XML tags and attributes. HTM files can be quite large, so they can be zip-compressed by calling the command as follows:

```
myNet.save("<untrained_htm_network_filename>.xml.gz")
```

During training, testing, and debugging, the network file is loaded in the NRE, and a trained network file can be saved by the NRE. Trained network files use the same XML file format and can also be used with `RuntimeNetwork`. See Using the RuntimeNetwork Object to Perform Training on page 61.

Using the RuntimeNetwork Object to Perform Inference

After the HTM Network has been trained, you can perform inference, that is, use the network to process new input. During inference, you submit data and a trained HTM Network to the NRE, which processes and categorizes the data.

You use the `RuntimeNetwork` object for inference as follows:

1. Create the `RuntimeNetwork` loading in the trained HTM Network you saved at the end of the training process and a set of test data.

   ```
   runtimeNet = CreateRuntimeNetwork(trainedNetwork, files=[testFile])
   ```

   In most cases, you perform inference first with the training data to verify that the training data are categorized correctly. You then perform inference with test data.

2. Retrieve the runtime object corresponding to the sensor and load the `testFile` file into that sensor runtime node.

   ```
   sensor = runtimeNet.Sensor
   sensor.execute('loadFile', testFile)
   ```

   Optionally, if the sensor is a vector file sensor, you can retrieve the number of vectors loaded in the previous command:

   ```
   numVectors = sensor.numVectors
   ```

3. Set the filename so the effector outputs are stored:

   ```
   fileOutputEffector = runtimeNet.FileOutput
   fileOutputEffector.execute('setFile', resultsFile)
   ```

4. Run the HTM Network. During inference, we exclude the category sensor because we want for the runtime engine to determine the categories.

   ```
   runtimeNet.run(numVectors, exclusion=["CategorySensor"])
   ```

5. Retrieve the results file and clean up.

   ```
   fileOutputEffector.execute("flushFile")
   runtimeNet.getFiles(resultsFile)
   runtimeNet.cleanupBundleWhenDone()
   ```

Training Your HTM Network: The Pictures Example

The basic pattern of creating the HTM Network structure, performing training using RuntimeNetwork, and performing inference using RuntimeNetwork is the same for all HTM Networks. The details might differ depending on the problem you're working with. Numenta includes a number of example programs to illustrate different approaches. This section briefly discusses the Pictures example.

The Pictures example performs training in a slightly unusual fashion. This section gives an overview of the training stages in the Pictures example. See the Pictures example itself, which is included in the Vision framework, for more details.

Stage 1: Training at the Bottom Level

The Pictures example uses an advanced learning technique of training a single node at each level and replicating its state to all the other nodes at that level. This approach is possible because the example sweeps across each training image. As a result, all bottom-level nodes would receive the same windowed images.

At the bottom level, training proceeds as follows:

1. Enable computation of a single bottom-level node and the Sensor.
2. Enable learning for that node.
3. Limit the window size of the Sensor (`ImageSensor`) to the receptive field of that node.
4. Calculate the number of iterations necessary to sweep the appropriately-sized window over all the training images. This number depends on the size of the image and the size of the window.
5. Compute for the full number of training iterations.
6. Disable learning and enable inference for that node.
7. Clone that node's state to all the other bottom-level nodes.

Stage 2: Training for Mid-Level Nodes

At the next level, training proceeds as follows:

1. Enable computation on a single mid-level node and on exactly the set of bottom-level nodes that provide inputs to that mid-level node (plus the sensor that feeds those nodes).
2. Enable learning for this single mid-level node.
3. Expand the sensor window as necessary to provide valid inputs to all of the enabled bottom-level nodes.
4. Reset the sensor to the first image and recompute the number of training iterations necessary to sweep over all the images with the new window size.
5. Compute for the full number of training iterations.
6. Disable learning for that node.
7. Enable inference for that node.

8. Clone the trained mid-level node's state to all the other mid-level nodes.

Stage 3: Training for Top-Level Nodes

For the top-level node, input from the preceding mid-level nodes and category information are used for training.

1. Enable computation of **all** nodes.
2. Enable learning for the top-level node only.
3. Expand the sensor window to provide valid inputs (that is, the entire image) to all bottom-level nodes.
4. Reset the sensor to the first image and recompute the number of training iterations necessary.
5. Compute for the full number of training iterations.
6. Disable learning for the top-level node.
7. Enable inference for the top-level node.

5 Debugging Your HTM Application

This chapter gives some introductory information on troubleshooting your HTM application.

For additional up-to-date troubleshooting information, see the Numenta website's Education section.

Topics

Making Sure The Data Are Valid

When the results don't make sense, check first whether the data you're using are fed in appropriately and picked up by the HTM Network as expected. A number of things can go wrong with data preparation and data submission. Ask yourself these questions:

❍ Are the data accurate? Look at the data. In many cases, humans can see the patterns that the HTM System can recognize later.

Some patterns are difficult to recognize by the human eye.

❍ Are your sensors picking up the correct training data files? If you're not sure, move or rename the training data files and confirm that an error results.

❍ Are you using the right number of training files? In some examples with multiple sensors, the system expects a corresponding number of training files. If you're using both a data sensor and a category sensor, you must submit two files.

❍ Is your category data file in sync with the data file? It's easy to make a mistake, which results in wrong information for each array in the training data.

❍ Are your data formatted correctly? Different sensors might have different formatting requirements.

Per-Level Training: Troubleshooting Efficiently

If you train your HTM Network completely and then run inference, the results are not always satisfactory. One reason is that if nodes at one level in your hierarchy are not performing well, nodes at a higher levels can't perform well either because their input is difficult to process. Because of the dependency of each node on all lower levels, it makes sense to train and analyze your HTM Network one level at a time while working on performance improvements.

1. Train the bottom level of your HTM Network and stop the training before training higher levels.
2. Examine the bottom-level output. You can either use the Visualizer tool (see Using HTM Network Visualizer on page 74) or extract data structures from the nodes directly.
   - You can use `RuntimeNetwork.getElement()` or the corresponding shortcut in most cases, for example:

   ```
   categorySensor = runtimeNet.getElement("CategorySensor")
   categorySensor = runtimeNet.CategorySensor
   ```

   - You can also use `Session` methods (see Using Session.execute() to Access Nodes or Execute Commands at Runtime, page 47 in *Advanced NuPIC Programming*.
3. See where potential problems lie, then change one or more parameters of the bottom-level nodes and repeat steps 1 and 2. For example:
   - If too many items are lumped in the same group, reduce the `maxDistance` parameter.
   - If your data have a lot of noise and you get too many groups with very few members each, increase the `maxDistance` parameter.
4. Repeat this process level by level until you have trained and analyzed your HTM Network completely.

Experimenting With Parameters and Input Data

At times, certain settings — either learning node parameters or application-specific parameter settings — might make it impossible for your HTM Network to complete a run. An example is shown in Running the Example with Noisy Data, page 18, where the `maxDistance` parameter becomes too small for a run with noise and an error results. In addition, issues with your input data might make it impossible for the HTM Network to recognize the patterns.

Experiment With Key Parameters

Each node performs spatial learning (coincidence detection) and temporal learning (grouping).

- For spatial learning, a key node parameter is `maxDistance`. Use larger `maxDistance` values for higher noise, smaller `maxDistance` values if things that don't belong together are lumped in the same group.
- For temporal grouping, a key parameter is `topNeighbors`. If you want fewer groups, increase `topNeighbors`. If you want more groups, decrease `topNeighbors`. Note that `topNeighbors` is needed only by `Zeta1Node`, not by the newer `SpatialPoolerNode` and `TemporalPoolerNode`.

See Affecting Learning Node Behavior With Node Parameters, page 36 in *Advanced NuPIC Programming* for a discussion of each parameter.

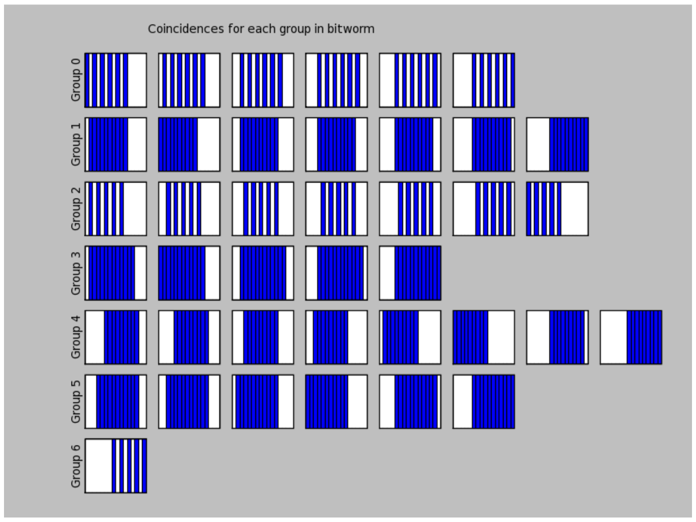
If your files are set up appropriately, experimenting with key parameters is fairly straightforward. For example, in Bitworm, you can make changes to many of the parameters by editing `RunOnce.py`.

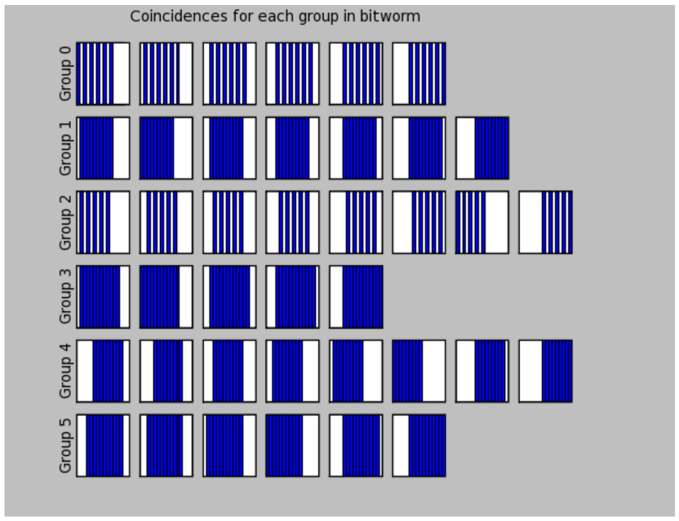
A second approach is to experiment with key parameters interactively. The Bitworm example includes a script, `ParameterExploration.py` that allows you to do so. Here is the core part of this script:

```
# Now instantiate the RuntimeNetwork
net = CreateRuntimeNetwork("trained_bitworm.xml")

# Get the level 1 node
node = net.Level1

# Run through all reasonable values for the topNeighbors parameter
# and see the resulting number of groups.
for t in range(1,10):
   node.setParameter("topNeighbors",t)
   node.execute("computeGroups")
   print "With topNeighbors set to",t,"the number of groups = ", \
         node.getParameter("groupCount")
```

When you run the script, you will see the network get trained and then see the effect of various `topNeighbors` values on the group structure of the `Level1` node. Output is sent to `stdout`.

For example, you might find that a different `topNeighbors` value than the default value results in a cleaner group structure. With the default value, one of the groups has only one element, and it's not clear why that bitworm was not assigned to one of the groups. With `topNeighbors` value of 4, the Group 6 bitworm is assigned to the appropriate textured bitworms group.





Experiment With Input Data

There are a number of things to look for in your input data:

❍ Do the input data have temporal sequences? — Temporal sequences are essential because they allow the HTM to learn that large sets of input patterns are related to each other because all patterns are generated by a single high-level cause. For example, a sequence of images of a cat moving, sitting, yawning, etc. is good input because even though the actual pixels are different, the underlying cause (the cat) is the same.

  HTM Networks depend on temporal continuity: Based on the frequency of transitions between naturally occurring sequences of patterns, nodes learn invariances and form a statistical model of their domain.

❍ Are sequences long enough? — Short sequences are jarring. Like a movie with several cuts a second, short sequences result in many random transitions, which become noise in the transition matrix. Nodes can't tell whether a transition is part of a moving object or a move to a next object unless instructed explicitly.

❍ Consider blank space for differentiating sequences — Insert blank space as an input vector with all zeros. These blanks tell the learning algorithm that the last pattern of sequence N has no temporal association with sequence N+1.

❍ Do you have enough training data? — Increasing the number of examples might noticeably improve performance. While more data are especially helpful if you wish to improve recognition accuracy, it might also be necessary if you're experimenting with a sample set that's too small.

Consider Preprocessing Data

In some cases, preprocessing is necessary because your data don't match the sensor you're using. You might also consider preprocessing your raw data if you believe that the raw data are difficult for the HTM Network to process. For example, if you're working with an HTM vision system, consider running the images through an edge detector and feeding the edge output instead of the images themselves to the sensor. However, be careful: Whenever you replace the raw data with preprocessor output you choose, you run the risk of discarding useful correlations hidden in the raw data before the HTM Network even sees the data.

Experiment with Network Topology

As a rule, network topology, such as fan-in at each node and the number of levels in the hierarchy should be based on design decisions that stem from the hierarchical spatial and temporal correlations inherent in the problem. However, if you believe after experimentation that a different topology might yield better results, reconsider your design and try a different topology.

Adding levels and changing the number of nodes per level is easy with the helper functions.

Improving Recognition Accuracy

After your HTM Network has successfully completed a training run, you are ready to test your HTM Network's performance. Follow these steps:

- Test the network's recognition accuracy using the training dataset — At a minimum, your HTM Network should be able to categorize the data on which it's been trained with very high accuracy (though not at 100%).
- Test recognition accuracy on an independent dataset — As a second step, test the HTM Network using new data. You could start with data that are relatively easy to categorize, then move on to data that are more difficult to categorize. If you only test on the original training data, you won't know whether the network can generalize to recognize new data, or whether it simply has memorized what it has seen.

  When you test with new data, make sure that the first set of new data is drawn from the same statistical model as your training data. When your HTM Network can correctly categorize that set of test data, you can move on to noisy data or to other data that are more difficult to test.
- Try changing other settings to improve performance (see Experiment With Key Parameters on page 70).

  For each case, change the configuration settings and save a new HTM Network, then train and test the newly-configured network.

Using HTM Network Visualizer

The Numenta Visualizer tool allows you to examine a trained HTM Network and see what the network has learned. Visualizer generates an HTML page for each node in the network. Each page displays the node's groups and coincidences as well as general statistics.

During development, it is usually best to train your network one level at a time, checking each level to make sure the nodes at that level have learned well before moving on. Training an entire network at once and then looking at end-to-end accuracy is not recommended, because it is difficult to determine exactly where a problem originates. If a certain level is not performing well, all higher levels are impacted because the outputs of one node are the inputs to its parent. You can use Visualizer to examine performance at one level before moving on to the next.

To illustrate the drawbacks of training the entire network at once, imagine that one of your bottom-level nodes is grouping all the patterns it sees into a single group. In this case, the nodes at the next level receive the exact same input no matter what data you feed to the hierarchy. The overall performance of your HTM Network is poor. It therefore makes sense to proceed as follows:

1. Run your training script, but stop after the Level 1 nodes have been trained.
2. Use Visualizer to examine the Level 1 nodes. If the nodes have poor coincidences and groups, change node parameters, and then train and look at the results again.
3. When you're satisfied with the bottom-level nodes, move on to successively higher levels of the hierarchy, making changes one level at a time.

Invoking Visualizer

Visualizer is installed as part of NuPIC and is located in the `nupic.analysis` package.

To invoke Visualizer

1. Start the Python interpreter.

   ```
   python -i
   ```

2. Import the Visualizer module.

   ```
   from nupic.analysis import Visualizer
   ```

3. Run the Visualizer on your trained network

   ```
   v = Visualizer(<your_network.xml>)
   v.visualizeNetwork()
   ```

where `<your_network.xml>` points to your trained HTM Network file

You can also run Visualizer automatically from another Python script by including the code from steps 2 and 3 above in your `.py` file. If you include a call to Visualizer at the end, the Visualizer HTML pages are generated automatically after training.

When you run `Visualizer.visualizeNetwork()`, Visualizer takes the following steps:

1. Starts a `RuntimeNetwork` with your trained HTM Network file.
2. Extracts data for each node in the HTM network.
3. Creates an HTML page for each node in the network, as well as a main page linking to all nodes.

   If you wish to examine only a particular node, pass a node name or regular expression to `Visualizer.visualizeNetwork()`. For example, `visualizeNetwork("level1.*")` prints all nodes that begin with `level1`.

   Visualizer deletes its temporary files when it is done.

Visualizer Output

The Visualizer output is organized as follows:

❍ Visualizer places the output HTML pages into a folder named after your network. For example, if you call Visualizer with a network file named `mynetwork.xml`, Visualizer creates a folder named `mynetwork`.

❍ Visualizer generates a main page called `index.html` within the main folder. Open this file to see an overview of the network with links to the pages for each of the nodes in the network.

❍ Each node has its own subfolder within the top-level folder. The HTML page for a given node is stored as `index.html` in that node's subfolder.

Supported Types of Data

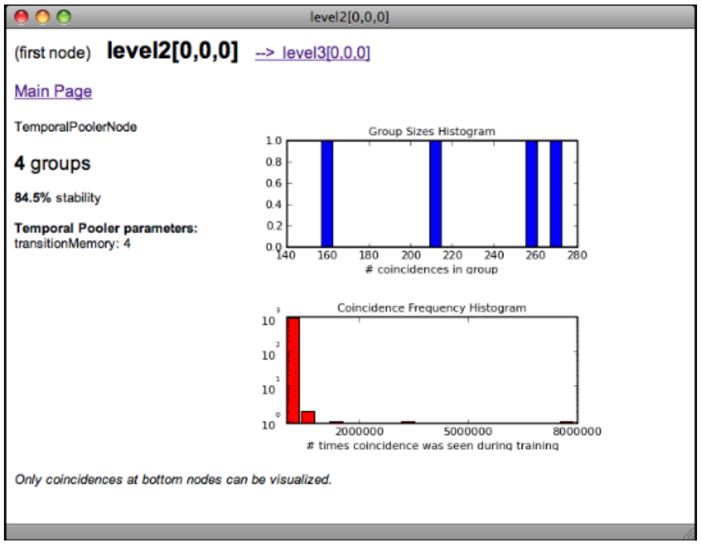
Once your data is fed from the sensor into the bottom-level nodes of your HTM Network, the input to any node is a vector of floating-point numbers. The network does not necessarily know whether these vectors represent pixels of an image, letters of text, or something else entirely.

Based on the type of sensor in your network, Visualizer can determine how to turn each of the node's coincidences into its original form. At higher levels, the coincidences do not correspond to raw input, so Visualizer does not show them.

Interpreting Visualizer Output

Visualizer creates a folder named after your network, located in the same directory. Open the `index.html` page within it to see an overview page for the network. This overview page lists the total number of nodes in the network — including sensors, effectors, and other nodes that are not visualized — as well as the number of nodes visualized. For each visualized node, the main page contains a link to the node's page and a count of the number of coincidences and groups in that node. Here's a main page from the Waves example:

Figure 14 Visualizer Main Page (Waves Example)


level2[0,0,0]

(first node) **level2[0,0,0]** --> level3[0,0,0]

Main Page

TemporalPoolerNode

4 groups

84.5% stability

Temporal Pooler parameters:
transitionMemory: 4

Group Sizes Histogram
# coincidences in group
Coincidence Frequency Histogram
# times coincidence was seen during training

Only coincidences at bottom nodes can be visualized.

Each node page displays the following information (See Figure 15):

❍ Node name

❍ Links to neighboring nodes

❍ Link back to the main page

❍ Name of the node class (such as `SpatialPoolerNode`)

❍ Number of coincidences in the node (for nodes that do spatial pooling)

❍ Number of groups in the node (for nodes that do temporal pooling)

❍ Node parameters such as `maxDistance`

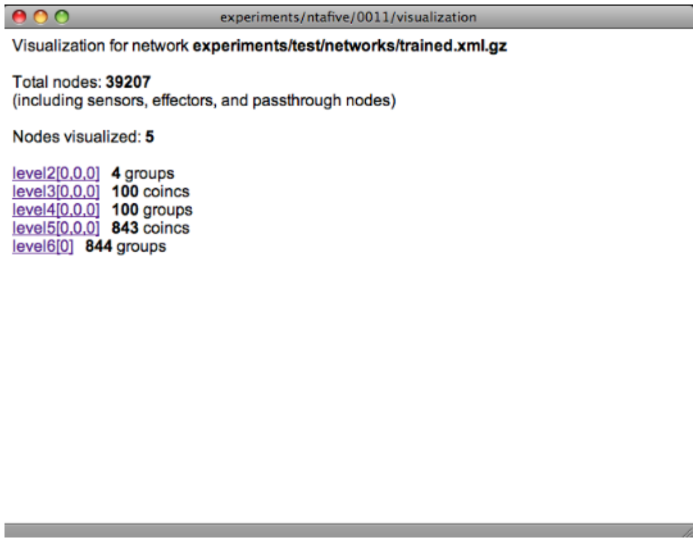
❍ A histogram of group sizes, where the size of a group is the number of coincidences it contains (for nodes that do temporal pooling)

❍ Stability ratings, computed for each coincidence and group, and for the node as a whole. Stability is related to the percentage of transitions to one coincidence that came from another member of the same group. A high stability value indicates that most transitions occur within the group. A low stability value indicates that many transitions occur between groups, suggesting the groups are not well formed (a node with random groups would have very low stability numbers). If you see low stability numbers, change the parameters of the temporal pooler, or examine the way you're feeding data into the network.

❍ For the bottom nodes, the page also displays the node's groups, and the coincidences within them

Here's a node page from the Waves example:

Figure 15 Visualizer Node Page (Waves Example)


experiments/ntafive/0011/visualization

Visualization for network **experiments/test/networks/trained.xml.gz**

Total nodes: **39207**
(including sensors, effectors, and passthrough nodes)

Nodes visualized: **5**

level2[0,0,0] **4** groups
level3[0,0,0] **100** coincs
level4[0,0,0] **100** groups
level5[0,0,0] **843** coincs
level6[0] **844** groups

Using Visualizer to examine the nodes of your HTM Network allows you to see how well your network is performing. If you spot problems with nodes at a particular level, you can adjust the node's parameters to fix them. If you spot serious problems in the level 1 nodes — for example, the groups seem nonsensical — you might have to reexamine how you feed your data into the network.

Plotting and GUI Packages Bundled with NuPIC

In addition to Numenta Visualizer, a number of plotting and GUI packages are bundled with NuPIC:

- matplotlib
- wxPython
- Traits GUI

A NuPIC Directories, Elements, and Help

This appendix gives an overview of the NuPIC components and explains how to get online help.

Topics

NuPIC Directory Structure

This section gives an overview of the NuPIC directory structure.

Top level		
README.TXT	contains release notes for this version of the software.	
<license file>	The license file named LICENSE-NuPIC-<xyz>.pdf differs for different situations. The file contains a copy of the actual license terms you agreed to when you downloaded the software.	
/bin	The /bin directory contains NuPIC executable programs.	
	numenta_runtime	The Numenta Runtime Engine. This application can act as either a Supervisor or a Node Processor.
	launcher	Numenta tool for starting up the runtime in different configurations.
	orted (UNIX only)	The Open Run Time Environment Daemon from Open MPI (www.open-mpi.org), which Numenta uses for underlying process management and interprocess communication
	orterun (UNIX only)	a program for starting Open MPI programs.
/lib	The /lib directory contains NuPIC plugins and python modules used with NuPIC.	
	libLearningPlugin.so /dll /dylib	Contains the learning node types provided by Numenta, including the `TemporalPoolerNode`, `SpatialPoolerNode`, and `Zeta1TopNode`.
	libBasicPlugin.so /dll /dylib	Contains all non-learning node types (except for Python nodes) provided by Numenta, including `VectorFileEffector`, `VectorFileSensor`, and `PassThroughNode`.
	python2.5/site-packages	Contains all Python packages provided by Numenta. The nupic python module contains NuPIC-specific functionality and are described in more detail below. NuPIC also includes the Python Imaging Library (PIL), the Numerical Python Library (numpy) the matplotlib plotting package, iPython, xPython, opencv, and the Traits GUI.
/share		
	doc	The /doc directory contains documentation plus licenses for 3rd party software used by NuPIC.
	projects	The /projects directory contains example HTM applications such as bitworm, waves, flu, and pictures.
	openmpi (UNIX only)	The openmpi directory is used by Open MPI and can be ignored.
	vision	Vision Framework

NuPIC Python Modules

The `nupic` package contains all Numenta Python support. It includes utility routines for configuring and saving HTM networks, as well as routines for launching the Numenta Runtime Engine and using the Runtime Engine for training and testing HTM networks. The most important packages in NuPIC Tools are:

❍ `nupic.network` — Contains modules for configuring, saving and running HTM networks. Includes helper functions and APIs for launching and managing the Numenta Runtime Engine.

❍ `nupic.analysis` — Contains modules for analyzing and visualizing HTM networks and their inputs and outputs. Also contains utilities for constructing large-scale experiments for HTM networks and problem parameterizations.

For information on each package, call `help(<package>)`, for example:

```
help(nupic.network)
```

NuPIC Node Types

This section gives an overview of the available nodes. Extensive online help for each node is available. See How to Access NuPIC Built-in Help on page 84.

❍ **VectorFileSensor** is a sensor that reads in text or csv (comma-separated values) files containing lists of vectors and outputs these vectors in sequence. The output is updated each time the sensor's `compute()` method is called. If `repeatCount` is greater than 1, each vector is repeated that many times before moving to the next one. The sensor loops when the end of the vector list is reached.

  If the file contains an incorrect number of floats, the sensor has no way of checking assignments. Currently it silently ignore the last vector of floats if there is an incorrect number.

❍ **py.ImageSensor** is a custom sensor created as a Python plug-in. This sensor was originally designed for the Pictures example but has been expanded to handle grayscale images. Because this node has been implemented in Python (not C++), getting help for it differs from getting help for other nodes. Use one of the following:

```
nodeHelp("py.ImageSensor")
nodeHelp("py.ImageSensor")
```

  See the *Vision Framework Guide* for more information.

❍ **SpatialPoolerNode** and **TemporalPoolerNode** are the learning nodes that are usually used for all levels except the top level. The learning nodes implements the algorithm described in the *Numenta Node Algorithms Guide*. See Affecting Learning Node Behavior With Node Parameters, page 36 in *Advanced NuPIC Programming* for an introduction to node parameters.

❍ **py.GaborNode** — Performs Gabor filtering (spatial pooling). The algorithm of this in this node is customized to work with image applications. Node input must come from an `ImageSensor` or from a node with identical output format. See the `Images` example.

❍ **Zeta1TopNode** — In learning mode, `Zeta1TopNode` learns coincidences and the mapping of those coincidences to categories. In inference mode, `Zeta1TopNode` computes to which category the input vector belongs. The output of `Zeta1TopNode` is thus a distribution over the categories of `Zeta1TopNode`, representing how likely the input vector is to belong to each of those categories.

  See Affecting Learning Node Behavior With Node Parameters, page 36 in *Advanced NuPIC Programming* for a discussion of `Zeta1TopNode` parameters and the *Numenta Node Algorithms Guide*.

❍ **py.SVMClassifierNode** — Implements the SVM (support vector machines) algorithm.

❍ **py.KNNClassifierNode** — Implements the KNN (k-nearest neighbor) algorithm. The node can perform learning and inference simultaneously.

❍ **VectorFileEffector** is an effector that writes its inputs to a text file. You can specify the target filename using the `setFile` execute command at runtime.

❍ **PassThroughNode** copies its input to its output. Both input and output must have 4-byte elements. `PassThroughNodes` might be useful if a sensor needs to connect to a node that's not at phase 1.

For an in-depth discussion of the NuPIC learning algorithm, see the *Numenta Node Algorithms Guide*. For an exploration of HTM behavior in general, which includes the discussion of an example, see the white paper *The HTM Learning Algorithms*.

How to Access NuPIC Built-in Help

The NuPIC APIs contain extensive built-in help. You can use Pydoc and Python's built-in help to get a complete online reference manual for the APIs. This section explains how you can access that help:

Using Pydoc

❍ To get Unix manpage-style help on the `nupic.network` API, type the following into the operating system command prompt:

```
pydoc nupic.network
```

❍ To generate and view HTML help, follow these steps:

a. Type `pydoc -g`.

b. Select **Open Browser** in the window that is displayed.

c. Click **nupic** for help on NuPIC.

❍ To get help from the Python interactive prompt, type

```
import nupic.network
help(nupic.network)
```

Getting Help for Nodes and Helper Functions

In NuPIC, node types are implemented as plugins, so pydoc cannot generate help on node-specific parameters and commands. NuPIC includes `nodeHelp`, which reads in each plugin's specification and displays help for the corresponding node. For example, the following command generates extensive online help for `SpatialPoolerNode`:

```
from nupic.network import *
nodeHelp('SpatialPoolerNode')
```

If the node is implemented as a Python plug-in, you can choose one of the following ways to invoke node help:

```
nodeHelp("nupic.pynodes.ImageSensor.ImageSensor")
nodeHelp("py.ImageSensor")
```

For the helper functions, you can display help as follows:

```
from nupic.network.helpers import *
help ("nupic.network.helpers")
```

You can also display help for individual helpers:

```
from nupic.network.helpers import *
help ("AddLevel")
```

Glossary

B

Belief
Within the context of an HTM, a belief is the probability distribution on a **cause** or set of causes. Specifically, belief refers to the distribution over a set of potential causes once all top-down, bottom-up and lateral evidence has been considered.

Bindings
Exposure to a target language of APIs originally written and implemented in a (different) source programming language. For example, Numenta Tools has Python bindings to a C++ library. In this case, Python is considered the target language, and C++ is the source.

Bundle
Collection of files stored in a single directory hierarchy. On some operating systems, the bundle can be made to appear as a single file. In Numenta Tools, a session bundle holds all files associated with a single conceptual NRE session.

C

Category
The top-level, distinct class to which entities or concepts belong.

Category file
A category file classifies training data.

Cause
An object in the world. From the HTM perspective, what's important about the objects in the world is that they have persistence, that is, they exist over time. A cause is not necessarily a physical object.

Classification
Classification is first performed during training: the HTM system is presented with a category file, which maps training data to categories. After that, the HTM system is presented with new data and can decide on the closest category match for each pattern.

Client code
Code that is accessing an API. Client code is not part of the API or its implementation. Client code includes software developed internally by Numenta engineering or QA departments, or may be developed externally by customers.

Cluster
Set of servers networked together using Ethernet or other networking protocols.

Coincidence
A coincidence is the noteworthy alignment of two or more events or circumstances without obvious causal connection. In the context of an HTM Network, a specific combination of patterns that are likely to occur together at one point in time.

Coincidence Detection
The process of detecting frequently occurring coincidences among input patterns.

Coincidence Matrix
A matrix of the coincidences the HTM system found after performing learning at one level.

Confusion matrix
The confusion matrix allows you to see how many items were assigned to which category. You run the `NetConfusion.py` script to get a confusion matrix.

CPU
One-processor core. A server can have multiple CPUs per server or per chip. For example, a server with two dual-core chips has a total of four CPUs.

E

Effector
Effectors are nodes that receive the output of the top-level node as input. The effector might send the output to a file or hardware device.

F

Fan-in
Fan-in refers to the number of outputs leading to one input of a node.

Fan-out
Fan-out refers to the number of outputs going from a node to the inputs of other nodes.

G

Geometry
The network geometry specifies the number of levels and for each level the node parameters such as fan-in that determine how nodes are linked.

Group
A set of coincidence patterns that are likely to occur close together in time.

Grouping
Process of creating groups.

H

HTM
Hierarchical Temporal Memory. Theory describing the structural and computational properties of the neocortex.

HTM Network
A set of nodes, sensors, and effectors connected to perform a specific function. Serve as the HTM structure that is being computed by the NRE.

HTM System
A complete system for running HTM Networks consisting of software and hardware components.

I

Inference
Inference is the act or process of deriving a conclusion based solely on what one already knows. In the context of the Numenta platform, it can mean that during training, nodes can infer for example, the likelihood that a certain item is the next item in a sequence based on other sequences it has seen. After the HTM Network has been trained, you can feed it new data and the HTM can infer the corresponding category (as a statistical pattern).

Input
Any node can receive input from all nodes to which it is linked. A node can have multiple inputs.

Invariance
Occurs when a belief is unchanged by a wide range of real-world transformations, often those which cannot be specified in concrete mathematical terms.

L

Launcher
The Launcher process is part of the NRE. The process runs only briefly as it launches the NRE. As a rule, users don’t interact with the launcher directly.

Learning
A node is in the learning state when it is receiving inputs, measuring the statistics of the inputs, and making modifications to its internal structures to represent the statistics of the inputs.

Learned State
Portion of a node's static state that is updated when learning occurs.

Link
Connection between nodes in an HTM Network.

M

maxDistance
The `maxDistance` parameter sets the maximum Euclidean distance at which two input vectors are considered the same during learning. When you set `maxDistance` to a higher number, you're more likely to get matches even if the noise-level is high. However, if `maxDistance` is too high, items that actually belong to different groups can end up in the same group.

N

NetExplorer tool
Numenta tool that allows you to test your HTM Network with different parameters and data and to see the results using gnuplot.

Network
The Python `Network` class implements an HTM Network.

Node
A node is the basic computational unit of an HTM Network. Node types include sensor, effector, and learning node. A learning node learns and represents the spatial and temporal statistics of the inputs to which it is exposed.

Node Input
See Input

Node Output
See Output

NP (Node Processor)
Software component that is responsible for running and scheduling a portion of an HTM Network.

NuPIC
Acronym for Numenta Platform for Intelligent Computing.

NRE
Numenta Runtime Engine. Software executables required for running HTM Networks. The NRE consists of the NP and the Supervisor.

NSAP (Numenta Supervisor Access Protocol)
A sockets-based protocol for communicating with the Numenta Supervisor. NSAP is a component of the Runtime API. Most developers don't use this protocol directly.

Numenta Network File Format
When you save a network after constructing it, or when you save a trained network, it is saved in Numenta Network File Format. Files in this format are in XML. If you modify an NFF file explicitly, it might no longer load. Use the tools for modification instead.

Numenta Platform for Intelligent Computing
Full name for the Numenta software platform. Includes the runtime engine and Numenta tools. Abbreviated NuPIC.

O

Output
The node output is the part of the node's state that's accessible by other nodes. Outputs can be arbitrary data types. Each node can have multiple named outputs. When a node is in inference node, it makes outputs available to other nodes.

P

Phase
A node's phase determines when the NRE executes it. You can specify the phase for each level during node creation.

Pipeline Scheduler
The pipeline scheduler is a high-performance node scheduler that can be used with feed-forward networks. The pipeline scheduler double-buffers node outputs and pipelines computation so that all nodes can be computed concurrently, making the pipeline scheduler ideal for multiprocessing.

Plugin
A plugin (or plug-in) is a computer program that interacts with a main application (a web browser or an email program, for example) to provide a certain, usually very specific, function. Numenta supports a node plug-in API that allows licensed users to create custom nodes.

Process
Single instance of a running program. Occupies system memory for program code, variables and objects.

R

Region
Regions are groups of nodes that all have the same configuration. Regions simplify common network topologies. Within a region, parameters cannot vary.

Runtime Engine
See NRE

S

Scheduler
The scheduler you choose determines the order in which nodes are executed. The basic scheduler uses phases. Advanced users can work with the pipeline scheduler in a multiprocessing environment.

Sensor
Input to HTM Networks. Sensors interface to external files, hardware devices, etc. and format data for input to other nodes.

Server
Physical computer containing one or more CPUs, hard drive, and power supply.

Session
A session includes all input data, output data, and interaction involved in a single use of the NRE. You can create and modify a session using the Python `Session` interface.

Session Bundle
See Bundle.

Static State
Portion of the node state that is independent of the runtime state of the system.

Supervised Learning
During supervised learning, the HTM Network is fed data and corresponding category information to learn the mapping between data and categories. After that, the HTM Network can perform inference on new data.

Supervisor
Portion of the NRE responsible for coordinating one HTM Network, and for communicating with external applications (e.g. the tools).

Supervisor command set
Portion of the NRE responsible for coordinating one HTM Network and for communicating with external applications (e.g. the tools).

Supervisor Command
Text commands for controlling the Supervisor. The command set is a subset of the API.

T

Time Adjacency Matrix
The system forms a time adjacency matrix by observing coincidences over time. The system uses that matrix to group the coincidences into temporal groups.

Tools (Numenta Tools)
Collection of software libraries, language bindings, and applications. Numenta tools provide access to the NRE, offer additional HTM related features, and are used by HTM applications.

Training
To train your HTM Network, you invoke the NRE with the network configuration and the training data. During training, the nodes in the HTM Network perform learning and inference.

U

Unsupervised Learning
During unsupervised learning, you feed data to the system without providing category information.

Visualizer Tool
The Numenta Visualizer tool allows you to examine a node to analyze its performance. It generates an HTML page for each node in the network, displaying groups and coincidences as well as general statistics.

Index