



# **Advanced NuPIC Programming**

*Document Version 1.8.1*  
*September 2008*

**Important Information**

This document contains proprietary information of Numenta Inc, and its receipt or possession does not convey any rights to reproduce, disclose its contents, or to manufacture, use or sell anything it may describe. It may not be reproduced, disclosed, or used without specific written authorization of Numenta Inc.

Numenta Inc, reserves the right to make changes in its products without notice in order to improve design or performance characteristics.

This document neither states nor implies any warranty of any kind, including but not limited to implied warrants of merchantability or fitness for a particular application.

The information in this document is believed to be accurate in all respects at the time of document, but is subject to change without notice. Numenta Inc. assumes no responsibility for any error or omissions, and disclaims responsibility for any consequences resulting from the use of the information included herein.

Copyright © 2006-2008 Numenta Inc. ALL RIGHTS RESERVED.

# Contents

<b>Figures</b> .....	<b>7</b>
<b>Preface</b> .....	<b>9</b>
Scope of Document .....	10
Document Overview .....	10
Related Documentation .....	10
Conventions .....	11
Document History .....	11
For More Information .....	12
<b>1 Software Components</b> .....	<b>13</b>
Introduction .....	14
Numenta Tools .....	14
Numenta Runtime Engine (NRE) .....	14
Runtime API .....	15
NRE Supervisor Process .....	16
NRE Node Processors (NPs) .....	17
Startup Sequence .....	17
Loading an HTM Network .....	17
Running the HTM Network .....	17
Sessions and NRE Supervisor .....	19
What is a Session? .....	19
Session Startup .....	19
Supervisor/Session Interaction .....	20
Interaction Example .....	20
Understanding Numenta APIs .....	22
<b>2 Developing HTM Networks: Advanced Topics</b> .....	<b>23</b>
NuPIC Node Types .....	24
Getting Node Help .....	24
Available Node Types .....	24
Node Inputs, Node Outputs and Links .....	26
Node Inputs and Output .....	27
Links .....	28
Link Types .....	28
Regions .....	30
Inside a Learning Node: How Learning and Inference Happen .....	31
Related Documentation .....	31
Learning and Inference During Training .....	31
Supervised and Unsupervised Learning .....	32
What Nodes Do During Learning .....	33
What Nodes Do During Inference .....	34
Affecting Learning Node Behavior With Node Parameters .....	36
Parameters in Both Learning Nodes .....	36
Parameters in SpatialPoolerNode .....	36
Parameters in TemporalPoolerNode .....	37
Working with HTM Network Files .....	39

Numenta .xml Files (Numenta Network File Format) . . . . .	39
Manipulating Trained Network Files . . . . .	39
Compression Support for HTM Network Files . . . . .	40
<b>3 Running HTM Networks With Sessions . . . . .</b>	<b>41</b>
Running HTM Networks: Options . . . . .	42
Understanding the Training Process . . . . .	43
Using the Session API to Run Your HTM Network . . . . .	44
Starting the Session . . . . .	44
Running the HTM Network . . . . .	45
Sessions and Session Bundles . . . . .	49
What RuntimeNetwork.run() Does . . . . .	51
Accessing Session Information at Runtime . . . . .	53
Interacting with Sessions . . . . .	53
Examining Node Content . . . . .	53
Look At Output Information. . . . .	55
Examining Scripting/Session Commands . . . . .	55
Log Files . . . . .	55
The launch.py File . . . . .	56
<b>4 Scheduling Node Processing . . . . .</b>	<b>57</b>
Understanding Scheduling. . . . .	58
Scheduler Overview . . . . .	58
Supported Schedulers. . . . .	58
Different Schedulers with Multiple NPs . . . . .	59
Using the Basic Scheduler with Multiple NPs . . . . .	59
Using the Pipeline Scheduler With More Than One NP . . . . .	61
Profiling and Load Balancing . . . . .	63
<b>5 Using the Numenta Runtime Engine: Advanced Topics . . . . .</b>	<b>65</b>
Introduction and Terminology . . . . .	66
Terminology. . . . .	66
Single-NP Process and Multiple NPs. . . . .	66
NRE Process Structure with Multiple NPs. . . . .	67
Hardware Configurations. . . . .	68
Single-CPU Machine . . . . .	68
Multi-CPU Machine . . . . .	68
Cluster (Unix-like Systems Only) . . . . .	69
Running in Parallel: Experiment Mode. . . . .	70
Running in Parallel: Large Problem Mode . . . . .	71
Using RuntimeNetwork in Large Problem Mode . . . . .	71
Using Sessions in Large Problem Mode . . . . .	71
Using TrainBasicNetwork() in Large Problem Mode . . . . .	72
Setting up a Cluster to Run NuPIC . . . . .	73
Introduction to Cluster Setup. . . . .	73
Requirements . . . . .	74
Cluster Performance Bottlenecks and Host Hardware . . . . .	75
How to Use NuPIC in Complex Configurations . . . . .	76
Using Multiple NPs . . . . .	76
Starting a RuntimeNetwork or a Session that Runs on a Cluster. . . . .	76
Launching on a Remote Host . . . . .	78
SessionConfiguration Object Methods. . . . .	81

<b>A Examples</b>	<b>83</b>
Bitworm Example	84
Problem Definition	84
Implementation	84
Exploration and Verification	84
Notes	84
See Also	84
Waves Example	85
Problem Definition	85
Implementation	86
See Also	86
Net_Construction Examples	87
Example Scripts	87
Flu Example	89
Problem Definition	89
Implementation	89
Learning from the Example	90
Pictures Example	92
Problem Definition	92
Implementation	92
Exploration and Verification	94
Experimenting Using the Pictures Demo GUI	95
<b>B Numenta NetExplorer</b>	<b>99</b>
Using Numenta NetExplorer	100
NetExplorer Basics	100
TestCrossParameters Class	101
TestCrossParameters Options	102
Classes Overview	102
Using Your Own DataInterface	103
Using Your Own NetInterface	104
Parameterized Tools	105
Advanced Exploration	105
Running NetExplorer Tests in Parallel	108
Parallel HTM Networks	108
<b>Glossary</b>	<b>111</b>
<b>Index</b>	<b>117</b>



# Figures

Figure 1 Runtime Components . . . . .	14
Figure 2 Node Inputs and Outputs . . . . .	27
Figure 3 Node Outputs, Inputs, and Links . . . . .	27
Figure 4 Common Link Types . . . . .	29
Figure 5 1-D Region of Four Nodes (Top) and 2-D Region of 2x3 Nodes (Bottom). . . . .	30
Figure 6 Three-layer HTM Network Example . . . . .	33
Figure 7 MaxDistance Example . . . . .	37
Figure 8 Session Bundle Example . . . . .	49
Figure 9 Example Network for Basic Scheduler, Multiple NPs . . . . .	59
Figure 10 Level-skipping HTM Network (Left) and HTM Network Using Pass-through Nodes (Right). . . . .	62
Figure 11 Different Configurations on Multi-CPU Machines . . . . .	69
Figure 12 Running in a Clustered Environment . . . . .	69
Figure 13 Typical Cluster Configuration . . . . .	73
Figure 14 Processes After Launching Remotely on a Single Host . . . . .	78
Figure 15 Processes After Launching 3 NPs on a Remote Host . . . . .	78
Figure 16 Processes After Launching the NRE on a Remote Cluster . . . . .	78
Figure 17 Waves Example . . . . .	85
Figure 18 Waves State Example: Category 3 Data . . . . .	85
Figure 19 Example of Rake in Pictures Example Set . . . . .	92





# Preface

This document is the companion volume to *Getting Started With NuPIC*. This document discusses advanced topics for developers who implement an HTM Network with the Numenta Platform for Intelligent Computing (NuPIC in the rest of this document). This preface gives some introductory information.

## Topics

- [Scope of Document, page 9](#)
- [Document Overview, page 9](#)
- [Related Documentation, page 10](#)
- [Conventions, page 10](#)
- [Document History, page 11](#)
- [For More Information, page 12](#)

## Scope of Document

---

This document is meant for developers who want to create HTM systems using NuPIC and who have already explored the materials in *Getting Started with NuPIC*.

The document includes discussions of the concepts behind the technology and practical, step-by-step instructions for the tasks involved in designing and implementing your HTM Network.

## Document Overview

---

This document consists of the following chapters and appendixes:

- Chapter 1, [Software Components](#), explains how the different NuPIC components interact at runtime.
- Chapter 2, [Developing HTM Networks: Advanced Topics](#), explores what it takes to build an HTM Network. This chapter includes background information and practical advice, and also explains how each task was performed for the Bitworm example program.

- Chapter 3, [Running HTM Networks With Sessions](#), illustrates how to use the `Session` API to invoke the Numenta Runtime Engine and explains what happens at runtime using some examples.
  - Chapter 4, [Scheduling Node Processing](#) discusses how the NRE schedules node processing using the basic scheduler or the pipeline scheduler.
  - Chapter 5, [Using the Numenta Runtime Engine: Advanced Topics](#), explores the concepts behind running in multiprocessor mode and running remotely, and explains how to run in these advanced configurations.
  - Appendix A, [Examples](#), gives overview information for some the examples included with NuPIC.
  - Appendix B, [Numenta NetExplorer](#), explains how you can test your HTM Network with different parameter settings using the NetExplorer tool.
- A [Glossary](#) and an Index complete the document.

## Related Documentation

---

*Getting Started With NuPIC*, the companion volume to this document, explains the HTM Development process using a simple example.

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

- 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 algorithms implemented by the NuPIC learning nodes in detail.
- The white paper *Problems that Fit HTMs* explains which problems are well suited for HTM Networks and which problems are better solved by more traditional technology.

Developers with the appropriate license can use the *Numenta Node Plugin Developer's Guide* to learn about developing 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
	<b>Note.</b> A noteworthy item. If you do not pay attention to a note, nothing bad is likely to happen.
	<b>Warning.</b> If you do not pay attention to a warning, data loss or other problems may result.
	<b>Tip.</b> 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 March 2007	First release.
1.0.1 March 15 2007	Minor bug fixes including update of white paper name, <i>Node Plugin Developer's Guide</i> document name, script name, and similar issues. Added Document History.
1.0.2 April 2007	Minor bug fixes. Added information on different ways of saving the HTM Network.
1.0.3 May 2007	Replaced node tools with <code>CreateNode()</code> function. Numerous name changes including pooler to spatial pooler and grouper to temporal pooler. Number of node processors for SimpleHTM is now specified in call to <code>train()</code> or <code>test()</code> , not as part of the constructor. Added <code>ImageSensor</code> appendix.
1.0.4 August 2007	Minor fixes/updates: Pipeline scheduler fill/drain is now done by NuPIC. <code>ImageSensor</code> appendix updated significantly. Miscellaneous updates to NetExplorer section.
1.5 September 2007	Completely restructured this guide. All basic information was moved to <i>Getting Started With NuPIC</i> . The focus of this document is now on more advanced information such as the architecture, using sessions, running in different configurations, and so on.
1.6 January 2008	Added information on time-based inference. Miscellaneous minor bug fixes.
1.7 May 2008	Replaced discussion of SimpleHTM with discussion of revised Network class. Changed document to reflect two nodes per level change. Replaced Zeta1Node with other node types. Migrated chapter 1 to <i>Getting Started With NuPIC</i> . Revamped architecture chapter.

Document Release	Description
1.8 June 2008	Updated document to reflect new node types and new linking behavior. Added some information on Helper functions. Added information on new examples. Removed tasks overview chapter. Migrated ImageSensor appendix to image framework documentation.
1.8.1 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.

For additional information, see <http://numenta.com/for-developers.php>.



# 1 **Software Components**

---

This chapter explains how the different NuPIC components work together to create and run an HTM Network.

This chapter is for developers who are interested in the NuPIC architecture. Skip this chapter if your main interest is using the advanced APIs.

See the *Getting Started with NuPIC* manual if you're interested in learning about developing an HTM Network.

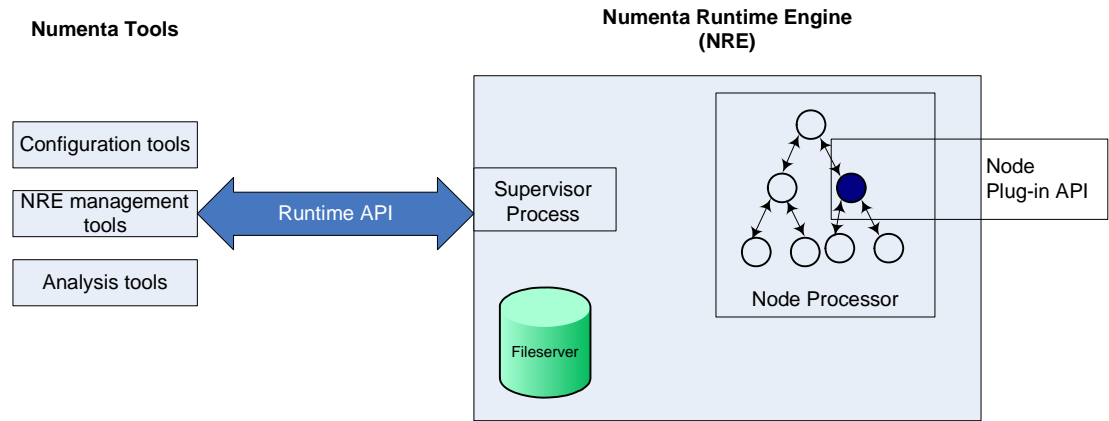
## **Topics**

- [Introduction on page 14](#)
- [NRE Supervisor Process on page 16](#)
- [NRE Node Processors \(NPs\) on page 17](#)
- [Sessions and NRE Supervisor on page 19](#)
- [Understanding Numenta APIs on page 22](#)

## Introduction

NuPIC has the following components, shown in Figure 1. The rest of this section discusses each component in more detail.

Figure 1 Runtime Components



### Numenta Tools

Numenta Tools allow you to create the HTM Network structure, run the HTM Network, and analyze the results of the run. See [Understanding Numenta APIs on page 22](#) for an overview of the different available APIs.

### Numenta Runtime Engine (NRE)

The NRE processes the HTM Network. You can run the NRE in different configurations, from a simple HTM on the single processor of your local machine all the way up to a complex HTM on a cluster in a remote location.

Parallel computing with the NRE is an advanced topic. While it is easy to turn on parallelism in the NRE, you are unlikely to see better performance unless you understand how multiprocessing works, have an HTM Network that is suitable for multiprocessing, and take into account the factors discussed in the relevant chapters of this document (see [Using the Numenta Runtime Engine: Advanced Topics on page 65](#)).

The NRE consists of these components:

- The **Supervisor** process manages node processors (NPs) and communicates with the `Session API`. The Supervisor starts and stops the NRE and is active while the NRE is running. It is responsible for:
    - performing global HTM Network operations
    - distributing nodes across NPs and coordinating the NPs
    - acting as the central point of communication between external tools and the NPs
- Numenta Python APIs such as `RuntimeNetwork` and the `TrainBasicNetwork()` helper function call `Session` to interact with the NRE.

- The **node processors** perform the actual node computations. See [NRE Node Processors \(NPs\) on page 17](#).

You can use Numenta Tools to create nodes that implement the Numenta algorithms. For custom algorithms, you can use the Node Plugin API discussed in the *Node Plugin Developers Guide*.

- Launcher — The launcher is of interest only to developers with multi-CPU versions of NuPIC, discussed in [Using the Numenta Runtime Engine: Advanced Topics on page 65](#).

## Runtime API

The **runtime API** handles all communications between the Supervisor process and the Session. It is the primary means of communication with the NRE. The Session API is the lowest-level client interface to the runtime API. RuntimeNetwork is a higher-level API used by most example applications, often in conjunction with the RunBasicNetwork() and RunBasicNetwork() helper function.

Internally, the runtime API consists of these components, which are only relevant under special circumstances:

- Supervisor Commands — Text-based requests that tools can send to the Supervisor to control the Supervisor's runtime behavior.

Applications can use Session methods that wrap Supervisor commands to interact with the Supervisor if that interaction seems necessary. Each request consists of the command name and optional arguments. For example, run 1000 runs the HTM Network for 1000 iterations.

For help for all Supervisor commands, type the following at the Python command line:

```
from nupic.network import Session
mySession = Session('test')
mySession.start()
mySession.sendRequest('help')
```

To get help on a specific command, call mySession.sendRequest('help <cmdName>').

- Numenta Supervisor Access Protocol (NSAP) — The NSAP sockets-based protocol is the basic mechanism used to connect with the NRE.



Most developers do not need to be familiar with NSAP because they use the Session API or the RuntimeNetwork API to communicate with the NRE.

At runtime, an instance of the NRE listens for NSAP requests.

NSAP is used to establish a direct socket connection between the client tool application and the NRE. NSAP includes protocols for connection, authentication, and sending and receiving commands. The basic mechanism is relatively high bandwidth. External tools can repeatedly connect to and disconnect from the NRE using NSAP. Once an NSAP connection is created, it establishes the basic communication channel through which Supervisor commands can be sent to the Supervisor over the network.

## NRE Supervisor Process

---

The Supervisor's primary functions are to support the tools interface and to manage the NPs. When launched, the Supervisor initiates a startup sequence to set up communication with the tools, set up communication with each of the NPs, and validate the license file. Once the initialization has completed, the Supervisor logs success, and then enters an event loop and waits for and processes events from the tools and the NPs.

During normal operation, the Supervisor must listen for and respond to:

- Commands coming in via the `Session` API through a connection.
- Messages received from the NPs. Because the Supervisor is a single-threaded design, these operations are done in the context of an event loop.

Once startup has completed, the Supervisor enters its main event loop, which goes through these steps:

1. Checks for a connection request from an external tool. If there is a request, opens a new connection to handle the connection and starts the NSAP connection negotiations.
2. Checks for incoming data on any established NSAP tool connections. Collects all incoming data. If a complete request has arrived, what happens next depends on whether the connection has been established:
  - If still in the connection negotiation phase, the event loop processes the next stage of connection negotiation.
  - If the connection has been established, the Supervisor:
    - a. Extracts the command line from the NSAP request
    - b. Passes the command line to the command-processing logic for parsing and execution
    - c. Collects the results of the command from the command processor and sends it back to the tools over the connection.



While processing the current request, the Supervisor is unresponsive to additional incoming requests. In steps 1 and 2b above, a command is passed to the command processor for execution. If a command takes a significant amount of time to process, the tools might notice a significant response time lag if they issue another request during this time.



## NRE Node Processors (NPs)

---

A Node Processor (NP) process runs a portion of an HTM Network on a CPU. There can be one or more active NPs for any instance of the NRE. These NPs can run on the same host as the Supervisor or be distributed across a number of hosts. As part of running the network, each NP communicates with the Supervisor and with other NPs on the network using an internal API.

The NP instantiates nodes, which are implemented as plug-ins. You must therefore make sure that each NP has access to the plug-in Python file or C++ object file. The NP is also responsible for managing a local scheduler, see [Understanding Scheduling on page 58](#).

### Startup Sequence

When launched by the Supervisor, an NP performs the following startup operations.

- Performs basic initialization.
- Establishes communication with the Supervisor.
- Logs a message stating that the startup sequence completed successfully. The log message includes the host name and the process ID of the NP.
- Starts listening for private commands from the Supervisor.

### Loading an HTM Network

The HTM Network load process is centrally managed by the Supervisor. The NP performs the following operations:

1. If an HTM Network is already loaded, it is removed from memory.
2. The NP loads the network as follows:
  - a. Receives information from the Supervisor describing nodes to be instantiated.
  - b. Instantiates each node in its sub-network, loading plug-ins as required. The NP logs any error detected during node instantiation.
  - c. Initializes the currently selected scheduler with the list of nodes.
  - d. Initializes some default state for each node output.
3. After initialization is complete, the NP waits for further commands from the Supervisor for running the HTM Network.

### Running the HTM Network

Once the HTM Network has been instantiated, the NP is in a paused state.

The Supervisor initiates the start of computation. Running the network involves cycling through the nodes in an order determined by the active scheduler. Each node is given a chance to perform its `compute()` operation.

The Supervisor can continue to send requests to the NP at any time. The NP processes these requests in between individual node computations. It does not need to be responsive while a node computation is in process.

In a setup with multiple NPs, each NP transmits its output to any other NPs that require it as each node updates its output. The NP has full control to determine the frequency of these broadcasts and any optimizations that might be performed. Likewise, each NP receives node outputs from other NPs.

## Sessions and NRE Supervisor

The `Session` API allows you to manage an NRE session from Python, to communicate with the NRE Supervisor, and to collect any runtime results produced during execution. See [Using the Session API to Run Your HTM Network on page 44](#).



In many cases, you don't use sessions explicitly but work instead with the `RuntimeNetwork` API, which encapsulates an HTM Network structure and a session, or the corresponding `RunBasicNetwork` helper function. See *Running the HTM Network*, page 55 in *Getting Started With NuPIC*.

### What is a Session?

The Python `Session` class allows users to interact with the NRE. You can use a `Session` to launch the NRE, load HTM Network files, control execution, and shut down the NRE on completion. A `Session` instance encapsulates the inputs, outputs, and interaction involved in a single use of the NRE.

Internally, the session communicates with the NRE using the Numenta Supervisor Access Protocol.

A typical session makes use of several files. To organize these files for easy transfer, analysis, and cleanup, the session manages a session bundle: a single filesystem directory created by the `Session` instance. The bundle is created immediately on session instantiation. `Session` methods allow you to add files to the bundle, extract files from the bundle, copy the bundle to and from a remote host, and clean up the bundle when it is no longer needed. See [Sessions and Session Bundles on page 49](#) for more information.

Here's how the different components interact:

- A session has full control over its files and over the NRE it launched. Different sessions write to different output files and launch and communicate with different Supervisor processes.
- A session can request that the NRE load or unload an HTM Network file and can control how the NRE processes that network file. The NRE manages only one network at a time.
- After a session has been shut down, you can still access session outputs for offline analysis.

### Session Startup

A `Session` instance can launch the NRE on a local or a remote host. You can launch the NRE using a number of parameters which determine, for example, the set of remote hosts, distribution of processes, executable, logging and instrumentation configuration, and communication options. All parameters have defaults that you can override for custom `Session` configurations.

During launching of the NRE, a session goes through these steps:

1. Creates a launch script called `launch.py` in the local session bundle and saves it to the `resources` directory. This script is ultimately executed on the NRE host.

2. If running in a cluster environment, copies the local session bundle to the host that will be used for launching the NRE.
3. Runs the launch script on the launch host. The working directory for the launcher is the session bundle.
4. Establishes a network connection with the Supervisor that was launched by the launch script.
5. Authenticates the connection by responding to a Supervisor challenge prompt.

Failures in session launch and communication leave the session in an unconnected state. In that case, an exception results and no attempt to connect or communicate succeeds.

If communication fails after a successful connection (for example, due to a loss of network access), the `Session` instance is put in an unconnected state and further communication fails.

## Supervisor/Session Interaction

The `Session` API allows you to affect NRE computation and retrieve state from the running HTM Network. Any command sequence that can be executed by the Supervisor can be run through a `Session` instance. For example, the session can adjust the set of nodes to be scheduled, can compute all scheduled nodes repeatedly for a number of iterations, and can pause computation at any time. The session can also tell the Supervisor to wait, which means to block all communication and ignore all requests until all running computations complete.

- When the Supervisor is not in a wait state, every request from a session to the Supervisor generates an immediate response.
- The response is sent back from the Supervisor over the current communication channel, is interpreted by the Session instance, and is returned as a return value from the `Session.sendRequest()` call.

## Interaction Example

If your program calls the `Session.getNetworkDescription()` method, the following events occur:

1. The `Session` sends the `netPrint` command to the Supervisor.
2. The Supervisor outputs descriptions of a set of HTM nodes to a string.
3. The Supervisor returns that text string as a normal response.
4. The `Session` waits for the response, receives the text string (marked as normal) and returns a `RuntimeResponse` data structure.

The `RuntimeResponse` data structure is a class in the tools library. `RuntimeResponse` has a type (normal, exception, timeout, unconnected), and a response message. The caller receives the response and can act on the included message.

If a request that arrives in the Supervisor generates an error exception upon execution, the Supervisor catches this exception and returns an `exception` response. The session recognizes this response and in turn throws a Python exception. When possible, the exception includes the exception message as the response message. The calling code in the local application should check whether the response is an exception and consider the exception message an error.

If communication between the Supervisor and the session fails — either during the original request or while retrieving the response — an `unconnected` response is returned from the `session` API. This response signals that the connection between the session and the Supervisor is in failure mode and additional communication should not be attempted.

In interactive applications, blocking session calls could hurt responsiveness. Some operations, like `session.loadNetwork()` and `session.saveRemoteNetwork()` may take a long time to complete, and give no feedback while they wait. The `session` API allows you to specify a timeout to prevent calls from blocking. If the session has not received a response within a specified amount of time, the `session` API generates and returns a timeout response, which has no associated response message. The timeout response indicates to the caller that session believes it is missing a response, and the session will check for that response before receiving any others (as all communication is received in order). The `session.setTimeout(number_of_seconds)` and `waitForResponses()` methods allow you to manipulate the timeout time.

The Supervisor supports certain requests that query the state of the HTM Network. These queries often result in responses that contain a text description of the state requested (assuming a normal response). The application can retrieve either the raw text response or both the raw text and the parsed data of the response from the `RuntimeResponse` object.

## Understanding Numenta APIs

Table 2: considers for different application types the main tasks the application needs to perform and the API the applications can use to perform them.

For any of the application types, you can use the Numenta Visualizer and Numenta NodeInspector tools to examine, debug, and improve the HTM network.

Table 2: Numenta APIs for Creating and Running HTM Networks

Application	Creating HTM	Running HTM
Basic, for example Bitworm	Helper functions: AddSensor() AddLevel() AddClassifier()	Helper function: RunBasicNetwork() Allows you to run the whole network.
Complex	Network API Link API Region API	RuntimeNetwork with run policy. Allows you to select levels to run. (TrainPhase for default nodes)
Custom learning algorithm	Node constructor See Network Creation with Node Constructors, page 34 in <i>Getting Started With NuPIC</i> .	.
Run NRE remotely or run clustered NREs		Use Session API for setup, RuntimeNetwork for running the network. See <a href="#">Running HTM Networks With Sessions</a> , page 41.
Vision networks	Vision framework	



## 2 ***Developing HTM Networks: Advanced Topics***

---

This chapter discusses advanced HTM development topics. It is a companion chapter to Constructing an HTM Network, page 41 in *Getting Started With NuPIC*, which discusses HTM Development fundamentals and explores them using example programs.



In most cases, you can use the helper functions to perform the tasks discussed in this chapter. If you need to customize the default behavior, this chapter can give guidance.

A good example for potential customization is the source code for the helper functions in `$NTA/lib/python2.5/site-packages/nupic/network/helpers.py`

### **Topics**

- [NuPIC Node Types, page 24](#)
- [Node Inputs, Node Outputs and Links on page 26](#)
- [Inside a Learning Node: How Learning and Inference Happen on page 31](#)
- [Affecting Learning Node Behavior With Node Parameters on page 36](#)
- [Working with HTM Network Files on page 39](#)

## NuPIC Node Types

---

Numenta ships a number of nodes, including learning nodes and sensor and effector nodes. You construct your HTM Network using these node types.



Because most node types are implemented as C++ plugins to the NRE, you cannot create node subclasses for them in Python.

Numenta Tools perform validation as you create and save nodes. During node creation the tools check whether each parameter is of the right type. When you link nodes or link regions, tools check whether the link you use is supported. Numenta Tools also check for incompatible combinations of parameters.

### Getting Node Help

In NuPIC, most node types are implemented as plugins, so pydoc cannot generate help on node-specific parameters and commands. NuPIC includes `nodeHelp`, which reads in each plug-in'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')
```

Note that getting node help is slightly different for nodes that were implemented in Python, not in C++, such as `ImageSensor`:

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

### Available Node Types

This section gives an overview of available node types.

#### Sensors

- **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. The size of pattern set is specified by `featureVectorLength()` when using the `AddSensor` helper function or the output size in a `CreateNode()` call.

Currently it silently ignores the last vector of floats if there is an incorrect number. The file to be read is specified using the `loadFile` execute command at runtime.

The following examples show how you can load a text file or a csv file:

```
sensor.execute('loadFile', trainingFile)           //text file
sensor.execute('loadFile', trainingFile, 3)        //csv file
```

You can use `VectorFileSensor` to submit data files or category files to your HTM Network.



- **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 C++ nodes.

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

## Learning Nodes

The following node types encapsulate the learning algorithms. Spatial and temporal pooling is in separate nodes. Usually, each spatial pooler node feeds directly into one temporal pooler node, but you can experiment with different fan-ins from spatial to temporal pooling. You can experiment with learning node behavior by using different parameters, see [Affecting Learning Node Behavior With Node Parameters on page 36](#).

- **SpatialPoolerNode** — Performs spatial pooling operations based on parameter settings.
- **TemporalPoolerNode** — Performs temporal pooling operations based on parameter settings.
- **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.

## Classifier Nodes

Classifier nodes learn coincidences and map those coincidences to categories. In inference mode, classifier nodes compute for each input vector the category to which that input vector belongs. The output of a classifier node is a distribution over the categories which represents how likely the input vector is to belong to each of those categories.

See NodeHelp for each node for more information.

- **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.
- **Zeta1TopNode** — Implements a Naive Bayes algorithm. This node, which was also available in earlier releases of NuPIC, can perform learning and inference simultaneously. This node is added by default if you use the AddClassifierNode() helper function.

## Effector Nodes

Effector nodes communicate with the outside world. VectorFileEffector receives input and writes it to a text file. You can specify the target file name using the setFile execute command at runtime.

```
effector.execute('setfile', 'myeffector42.txt')
```

## Other Nodes

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

See the `PicturesNetworkPassThrough.py` script in the `net_construction` example set.

## Node Inputs, Node Outputs and Links

---

When the NRE runs your HTM Network, data are passed from one node to another as part of learning and inference. This section discusses node inputs, node outputs, and links.

- Node inputs — Each node can have multiple named inputs, through which it receives data from other nodes. Sensor nodes don't have inputs because they receive data from outside the HTM Network, not from other nodes.

Each node has a predefined list of available named inputs, similar to arguments in a function signature. If one named input receives data from multiple nodes, the data are concatenated, as in [Figure 3, Node Outputs, Inputs, and Links](#).

- Node outputs — Each node can have multiple outputs, through which it makes data available to other nodes. Effector nodes don't have outputs because they send data outside the HTM Network not to other nodes.

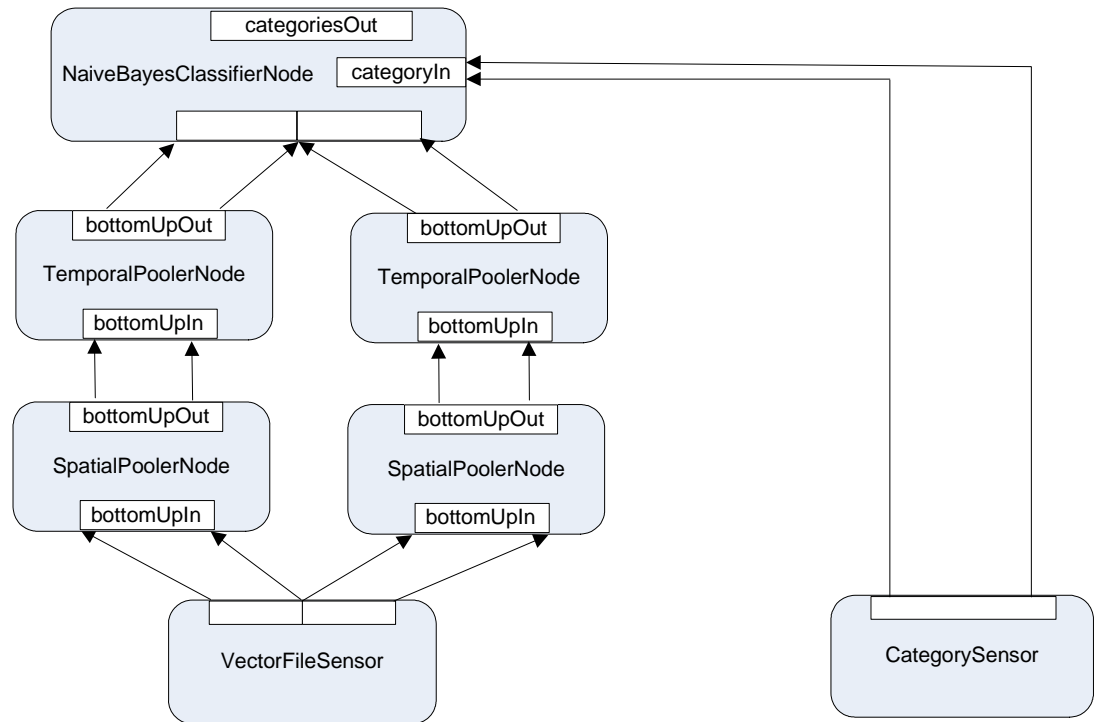
Each node has a predefined list of available named outputs, similar to return arguments in a function signature. Node output names and output element type are specified as part of the node class definition by default, your program can't change them. See [Node Inputs and Output, page 27](#).

A node output can be available to more than one node at the next level.

- Links — Links between nodes map outputs to inputs. While output from a node can be made available to any other node in the HTM Network, you must create a link to connect the node's output to another node's input. See [Links, page 28](#).

The following diagram illustrates node inputs, node outputs, and links using the input and output names of the node types included with this release. See [NuPIC Node Types, page 24](#), for more information.

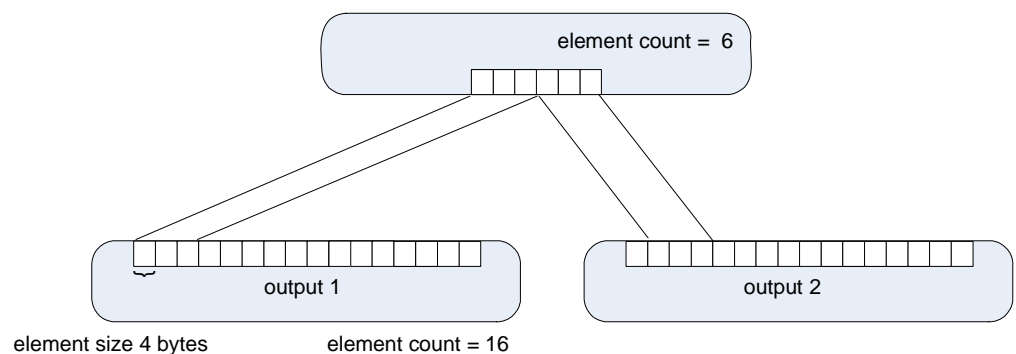
Figure 2 Node Inputs and Outputs



## Node Inputs and Output

Each node input or output is a contiguous array of elements of the same size. For nodes currently included with NuPIC, it's an array of floating-point numbers. You can link any output with any compatible input (that is, input with the same element size and type). Multiple links can be attached to the same input, as shown in [Figure 3](#). Inputs are concatenated in the order in which the links are created.

Figure 3 Node Outputs, Inputs, and Links



Node inputs and outputs have the following parameters:

- **Element size** — Size of the individual entries in the output vector. In [Figure 3](#) above, element size is 4 bytes. Element size for output and input must be the same. All nodes included with the Numenta Tools framework are already the same size, but if you create a custom node, you must take care of matches.
- **Output element count** — Each output has an output element count (the number of entries in the vector) which you can specify when you create the node. Node outputs are allocated statically. The number of elements for the output is set at startup and does not change during HTM Network operation.
- **Data type** — All nodes included with NuPIC have a vector of floating-point numbers as output; however, outputs can be any data type. If you create or use a custom node, that node might use other data types.
- **Name** — Each output has a name, which is predefined for that type of node. For example, both `SpatialPoolerNode` and `TemporalPoolerNode` use `bottomUpOut` for its bottom up outputs. The `VectorFileSensor` has a single output named `dataOut`. See [NuPIC Node Types, page 24](#) for available output names.



If you're developing custom nodes using the plug-in API, you must name inputs and outputs as part of the node definition. Nodes included with NuPIC have predefined output names.

- **Host information** — In multi-core systems, node outputs form messages that are transmitted to destination hosts. See [Using the Numenta Runtime Engine: Advanced Topics, page 65](#).

## Links

Each link connects one source node output with one destination node input. Links project part of a source node's output vector into the destination node's input field. In the simplest case, when the source node has one output and the destination node has one input, you can specify the link by specifying source and destination node names.

To fully specify a link, you need the name and output name of the source node, and the name and input name of the destination node. Optionally, you might specify that only a portion of a particular output will be communicated. This portion must be contiguous, and can be offset from the beginning.

## Link Types

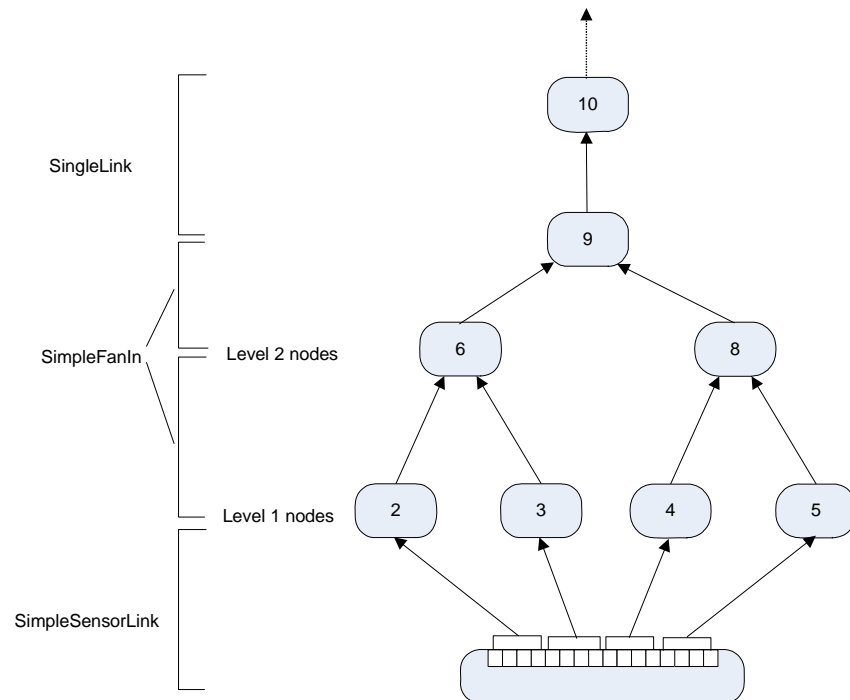
Certain nodes are often linked to certain other nodes in a particular way. For example, sensor output usually goes to multiple Level 1 nodes, learning nodes might use single links and fan-ins or fan-outs, and classifier nodes link to effectors using a single link.

You can use link types to efficiently connect the nodes at two levels (instead of linking each node explicitly to other nodes). Three link policies are defined:

- `SimpleSensorLink` divides the sensor output array evenly across all nodes at the next level (this link policy works for both 1-D and 2-D arrays).

- SimpleFanIn divides node output evenly, as shown in [Figure 4](#).
- SingleLink connects one node to another node; useful for connecting a classifier node to an effector.

Figure 4 Common Link Types



See Link Types, page 51 in *Getting Started With NuPIC*.

## Customizing Link Policies

Customizing a link policy allows you, for example, to specify an offset.

You customize the link policy by customizing the link policy, used as the third argument in the call to `Network.link()`.

```
myNet.link("<source_region_name>", "<dest_region_name>", <link_policy> )
myNet.link("<source_node_name>", "<dest_node_name>", <link_policy> )
```

For `<link_policy>`, you can specify either a link policy by name, or a link policy constructor.

For example, the `singleLink` policy constructor has the following prototypes:

```
SingleLink("<sourceOutputName>", "<destinationInputName>")
SingleLink("<sourceOutputName>", int <offset>, "<destinationInputName>")
SingleLink("<sourceOutputName>", int <offset>, int <elementCount>,
"<destinationInputName>")
```

Use `NodeHelp` for information about each link policy.

For example, to select output elements 3-7 (inclusive) of output `dataOut` of node `sensor`, and send those elements to the default input of node `level1`, you can call:

```
net.link("sensor", "level1", SingleLink("data", 3, 5, ""))
```

## Regions

A region is a collection of nodes that have precisely the same parameters, including the same phase. The sample network in Figure 4 does not include regions, but most larger HTM Networks do.

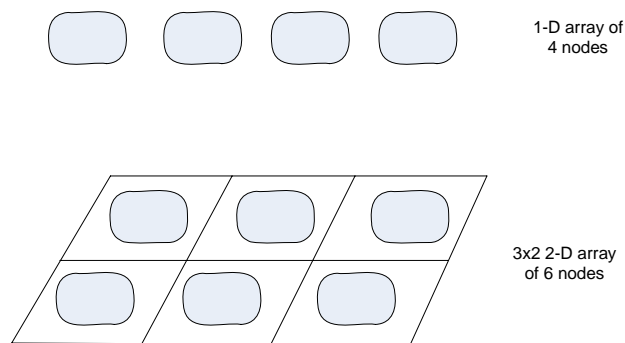
Regions are a useful mechanism for creating your HTM Network because you can create the most common types of networks with very little code. This includes being able to specify region to region links and links between a region and a sensor or effector, with one call using link policies. Regions also handle multi-dimension input connections.

Regions are not a good choice if most of the nodes in the HTM Network have different parameters. There is no implied communication between the nodes of a region.

You can create 1-D and 2-D regions:

- A 1-D region is a simple sequence of nodes.
- A 2-D region is a group of nodes that are logically arranged in a 2-D plane. Each node has the same phase, but there are multiple nodes in both dimensions. For example, a 2-D region can be used for vision applications where you want the nodes to be arranged in a plane.

Figure 5 1-D Region of Four Nodes (Top) and 2-D Region of 2x3 Nodes (Bottom).



See Creating Regions, page 49 in *Getting Started With NuPIC* for additional information.

The `NetConstruction` example illustrates the different kinds of networks you might wish to create. Most of the examples use regions of spatial and temporal poolers at each level.



The `Images` example uses multi-dimensional regions, as discussed in the corresponding documentation.

## Inside a Learning Node: How Learning and Inference Happen

---

At runtime, Numenta learning nodes perform learning and inference.

- Learning — Nodes perform spatial learning and temporal learning to build a model of their world. During spatial learning, the node learns frequent spatial patterns in the input vector. During temporal learning, the node learns temporal groups of frequently adjacent spatial patterns. See [What Nodes Do During Learning](#), page 33.
- Inference — During inference, the node receives inputs and uses the information it gathered during learning to produce an output for each input.

This section explores the concepts behind learning and inference for the default learning nodes (`SpatialPoolerNode` and `TemporalPoolerNode`) and `Zeta1TopNode`.

If you're using the helper functions (`TrainBasicNetwork()` etc) or `RuntimeNetwork`, learning and inference at the different levels happen automatically.

### Related Documentation

This section gives only an introduction to the learning process. To fully understand how Numenta learning nodes perform learning and inference, read the *Numenta Node Algorithms Guide*. See the white paper *The HTM Learning Algorithms* for an in-depth discussion of how an HTM works.

### Learning and Inference During Training

During training, each node needs to perform both learning and inference. This can happen implicitly or explicitly depending on the API you are using.

1. First, the program enables sensor and bottom-level (Level 1) nodes. The Level 1 nodes are set to learning mode and create a model of their world. All other nodes are disabled. This setup is held for a number of Level 1 learning iterations until Level 1 is fully trained. This number must be determined by the HTM developer and usually requires some experimentation. You specify the number of iterations when calling `RuntimeNetwork.run()`, usually in conjunction with the run policy.
2. The program sets Level 1 nodes to inference mode and the next level to learning mode. That means the output from the computation performed at Level 1 becomes available to Level 2. Level 2 nodes build a model of their world based on that output. This setup is used for a number of Level 2 learning iterations until Level 2 is fully trained.
3. The program sets the Level 2 nodes to inference mode (making their output available) and the next level to learning mode.

4. The program progresses until all nodes have completed learning and have been set to inference mode. The program can then save the fully trained HTM Network and reload for testing with the training data or with new data. Because all nodes are in inference mode at the end of the training run, incoming data can be processed right away.



If you're using `RuntimeNetwork`, you don't need to enable and disable modes explicitly. You just call `RuntimeNetwork.run()`.

For an example, see [Running the Network to Perform Learning and Inference, page 36](#).

## Supervised and Unsupervised Learning

When you train your HTM Network, you can submit category information to the classifier node so classifier nodes can group according to category (supervised learning). If you don't include category information (unsupervised learning), the classifier creates groups based on the characteristics of the input data.



## What Nodes Do During Learning

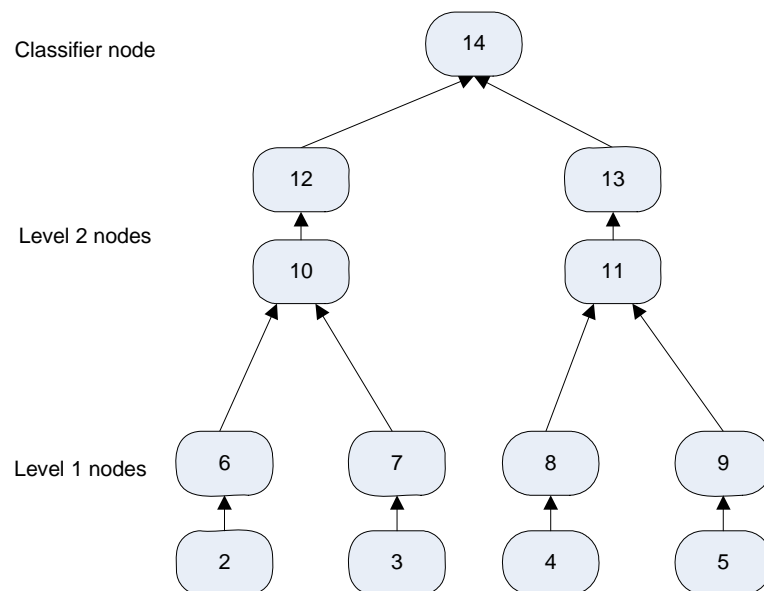
This section explores how nodes perform learning. The current learning algorithm, implemented by `SpatialPoolerNode` and `TemporalPoolerNode`, consists of two major operations:

- Learning frequent spatial patterns.
- Learning the temporal relations between the learned spatial patterns and forming temporal groups based on these temporal relations.

When a node is in learning mode, it is receiving inputs, measuring the statistics of the inputs, and making modifications to its internal structures to represent the statistics of the inputs. During this stage, the node does not currently produce any output.

Consider nodes 10 and 12 in the three-layer HTM Network shown in [Figure 6](#):

Figure 6 Three-layer HTM Network Example



5. Node 10 receives input data from nodes 6 and 7. Each input is a vector.
6. Each time Node 10 receives data, it performs spatial pooling and makes the spatially pooled data available to Node 12.
7. Node 12 learns the temporal relationships between the spatial patterns in Node 10. For each input pattern, Node 12 records the pattern(s) that precede(s) the input pattern in a transition matrix.
8. Each time Node 10 receives a new input pattern, the two nodes repeat the steps of storing the pattern in the matrix and performing temporal pooling again.
9. As the last step of learning — after processing all new input — the node partitions the set of spatial patterns into temporally coherent subgroups. To do so, the node forms groups so that all the spatial patterns within a group are highly likely to follow each other in time (and the spatial patterns within different groups are less likely to follow each other in time). The motivation for this is that spatial patterns that are highly likely to follow one another in time are likely to be linked to the same cause in the world.

### Structure of a Fully Trained Level

Each level consists of a spatial and a temporal pooler node. Once an HTM level finishes all stages of learning, it contains the following information in the nodes at each level:

- A set of spatial patterns in the spatial pooler.
- A temporal transition matrix that stores the first-order temporal relations between the spatial patterns in the temporal pooler.
- A set of temporal groups. Each temporal group is a subset of the set of spatial patterns in the spatial pooler.

At this point, the level has built a model of part of the world. During inference, the NRE will use this model to process incoming data.

### What Nodes Do During Inference

Learning nodes are trained on a per-level basis. Each level contains sets of spatial pooler and temporal pooler nodes. Classifier nodes encapsulate both the spatial and the temporal pooler. This section discusses inference for both learning nodes and classifier nodes.

#### Learning Nodes

Once a learning node level has been trained, it can be switched to inference mode. During inference, the level already has a model of the world (stored in the spatial and temporal pooler nodes). When the level receives an input from its children, it uses its internal model of the world to create an output to send to its parent(s).

A learning node level creates its output by taking the following steps:

1. The level receives as input a single vector of floating-point numbers, which represents the concatenation of the output vectors from its children. Each of these child outputs is a vector containing a distribution over the groups of that child.
2. The input first arrives at the spatial pooler node, which compares this input to all previously-learned coincidences, and then outputs a distribution over the coincidences.
3. The distribution over coincidences is sent from the spatial pooler node to the temporal pooler node. The temporal pooler knows which coincidences belong to which groups. It outputs a distribution over groups. This vector becomes the output of the level, and becomes available to the node's parent at the next level.

#### Classifier Node

The behavior of the classifier node is very similar to that of the learning node, except for the last step.

1. The classifier receives a single vector of floating-point numbers as input, which represents the concatenation of the output vectors from its children. Each of these child outputs is a vector containing a distribution over the groups of that child.
2. The input first arrives at the spatial pooler inside the classifier, which compares this input to all previously-learned coincidences, and then outputs a distribution over the coincidences.

3. The distribution over coincidences is sent from the spatial pooler node to the temporal pooler node. The temporal pooler knows which coincidences belong to which groups.
4. Finally the classifier node, it receives outputs a distribution over its learned categories based on the distribution over groups. This vector becomes the output of the node. Often, an HTM network is configured so that the output of a top node is sent to an effector, which then writes this distribution over categories to a file.

If you would like to learn more about the inner workings of the spatial pooler, temporal pooler, and supervised mapper, see the *Numenta Node Algorithms Guide*, which describes these algorithms in detail.

## Affecting Learning Node Behavior With Node Parameters

---

When you create a learning node, the node's parameters affect the node's behavior.

Note that Numenta Tools check for incompatible combination of parameters when you write to the HTM Network file.



This section gives only overview information of the most frequently used parameters of `SpatialPoolerNode` and `TemporalPoolerNode`.

For an in-depth discussion of the concepts behind the algorithms see the *Numenta Node Algorithms Guide*. Use `NodeHelp` for reference information for individual node parameters (see [Getting Node Help, page 24](#)).

### Parameters in Both Learning Nodes

The following parameters of `SpatialPoolerNode` and `TemporalPoolerNode` are frequently changed to affect node behavior.

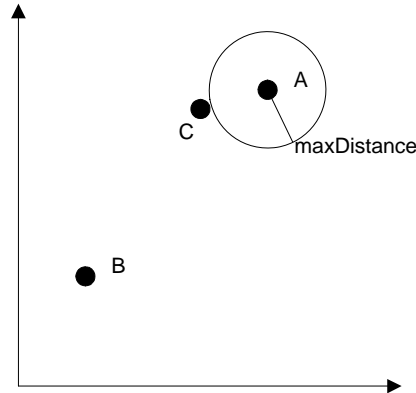
- `clonedNodes` — Applies when nodes are in regions. If true all nodes in the region share state. Default is false
- `inferenceMode` — Allows you to determine whether a node is in inference mode (`True`) or not (`False`). You can also explicitly set a node to inference mode using this parameter.
- `learningMode` — Allows you to determine whether a node is in learning mode (`True`) or not (`False`). You can also explicitly set a node to learning mode using this parameter.

### Parameters in `SpatialPoolerNode`

The following `SpatialPoolerNode` parameters are frequently changed.

- `maxDistance` — Sets the maximum Euclidean distance at which two input vectors are considered the same during learning. For example, in the figure below, results are in a 2-dimensional space. Point A and B are clearly different points; however, point C is close to point A. If the value of `maxDistance` is large enough to have the circle include point C, then the two are considered two occurrences of the same point. If `maxDistance` is small and the circle does not enclose C, then the two are considered two different points.

Figure 7 MaxDistance Example



If a lot of noise is present in the data, consider setting `maxDistance` to a higher value.

- `sigma` — Sigma to be used in the rbf (radial basis function) in Gaussian inference mode.
- `sparsify` — If true, the system stores a sparse version of the input vector. Default is false.

The following parameters are available from the node after learning:

- `coincidenceCount` — Number of learned coincidences as an integer.
- `coincidenceMatrix` — The coincidence matrix as a sparse matrix.

## Parameters in TemporalPoolerNode

The following `TemporalPoolerNode` parameters are frequently changed to affect node behavior.

- `requestedGroupCount` — Number of groups requested.
- `transitionMemory` — Specifies how many true steps of history to keep in the temporal pooler to track the time structure of coincidences while learning the time adjacency matrix. Default is 1.
- `temporalPoolerAlgorithm` (required for `TemporalPoolerNode`) — A string that selects the method of computing output probabilities. Note that this parameter only affects the inference behavior of the node, and has no impact on what is learned.
  - When set to `maxProp`, computes a more peaked score for the group based on the current input only.
  - When set to `sumProp`, computes a smoother score for the group based on the current input only.
  - When set to `tbi`, computes a score using Time-Based Inference (TBI) which uses the current as well as past inputs. See [Understanding Time-Based Inference](#).

The following parameters are available from the node after learning:

- `coincidenceVectorCounts` — Array of integers that shows how often each coincidence was encountered during learning.

- `groupCount` — Number of groups actually generated.
- `groups` — Set of coincidences belonging to each group, returned as a list of coincidence indices.
- `TAM` — Time adjacency matrix, returned as a sparse matrix.

### Understanding Time-Based Inference

When you specify `tbi` as the value of `temporalPoolerAlgorithm` of a `TemporalPoolerNode`, the system computes a score using Time-Based Inference (TBI), which takes into account the current input as well as past inputs.

Using time-based inference can help your network produce better inference results in applications where each successive input is temporally related to the previous one. For example, if you have an image application and you are feeding in successive frames of a movie, TBI mode will produce better inference results than either `maxProp` or `sumProp`. The `maxProp` and `sumProp` modes are best suited for applications where each successive inference input is completely independent from the previous one (i.e. flash inference).

In TBI mode, the output is computed based on the current and previous inputs. The system takes into account the likelihood of each coincidence to follow another (based on the order of coincidences seen during training) and uses this information to compute the output probabilities after each input. Consider this example:

- The temporal pooler has formed a group consisting of coincidences C1 and C3, and during training C3 preceded C1 quite often.
- During inference, C3 is fed in on time step 1 and C1 on time step 2.
- If the node is in TBI inference mode, the output for the group will be higher in time step 2 than it is in time step 1.
- If the node is in `sumProp` or `maxProp` inference mode, the output for the group will be the same in both time steps.



The current implementation of TBI mode works well only at level 1 of the network. This is because the current TBI algorithm does not take into account the different relative time scales of each level of the network that result from temporal pooling.

Note also that you can change the `temporalPoolerAlgorithm` parameter at inference time. For example, you can take any previously trained network and enable TBI inference by setting the `temporalPoolerAlgorithm` to `tbi` for all level 1 nodes or use flash inference by setting it to `maxProp` or `sumProp`. The network does not have to be trained in TBI mode in order to use TBI during inference.

## Working with HTM Network Files

You can save your HTM Network at various stages of the development process:

- Save your file after you've completed configuration. When you save the untrained HTM Network, you save only its structure. There is as yet no learned node state. Developers rarely save untrained HTM Networks because recreating the network is just as fast as loading the file.
- Save during or after training. At any point, you can save the current state of your trained network. This might be after you've trained one level, or after the HTM Network is fully trained. When you save the trained HTM Network, the learned node state is included and will be available when you later load the file into the NRE. Developers often save trained networks because training the network can take a long time.

In most cases, you save the file using the `save()` method. The method writes the file in Numenta Network File Format, the only format the NRE can understand.

```
myNetwork.save("<filename>.xml")
```



End-to-end support for compression is included with NuPIC. See [Compression Support for HTM Network Files](#), page 40.

### Numenta .xml Files (Numenta Network File Format)

The system saves your file in the HTM Network XML file format.



This file format is likely to change in the future. Loading the files using Numenta tools and APIs instead of editing the XML file is highly recommend.

An HTM Network File (called HTM file in the rest of this document) contains all the information required to instantiate an HTM Network, initialize its static state to some known value, and run it. If training scripts have been executed, the learned state of each node is included in the file. That means you can train an HTM Network on one system, and run that same network on any other system with the same or greater capabilities.

An HTM file specifies an HTM Network completely, including all the nodes, their types, the node topology, the node allocation across CPUs, and each node's state parameters and learned state.



The HTM file does not include dynamic network state such as the next node to be scheduled, the current outputs of each node, previously queued outputs, or each node's internal dynamic state.

### Manipulating Trained Network Files

In most cases, you load the trained HTM file into the NRE. The NRE uses the information in the file to create the runtime network hierarchy.

You can also load the HTM file directly with NuPIC Tools to allow applications to manipulate trained HTM Networks. For example, you could copy trained nodes from one network and insert them into a different network.



You can read and copy network elements but can't modify the structure of the network.

For example, to create an HTM Network from an existing file call:

```
myNetwork = Network("<filename>.xml")
```



Use this feature only if you don't need a precise match. It is possible that the constructed HTM Network does not exactly match the untrained network that was originally written to disk as an XML file.

When you load the HTM Network file explicitly, you cannot query individual node parameters, though you can still use the NRE to query and change the parameters. In addition, links that were stored as link policies become point to point links, which makes them less flexible.

## Compression Support for HTM Network Files

HTM Network files provide end-to-end compression support. The following methods automatically compress and decompress appropriately when the specified file extension is .gz:

```
Network::Network("myFilename.xml.gz")
Network::save("myFilename.xml.gz")
Network::write("myFilename.dot.gz", "dot")
Session::loadNetwork("myFilename.xml.gz")
Session::loadRemoteNetwork("myFilename.xml.gz")
Session::saveRemoteNetwork("myFilename.xml.gz")
```





## 3 **Running HTM Networks With Sessions**

---

Running the HTM Network, page 55 in *Getting Started With NuPIC* explains how you can run an HTM Network using either `RuntimeNetwork.run()` or the `RunBasicNetwork()` helper function.

This chapter explains how you can use the `Session` API to interact with the NRE and how you can explicitly turn on learning or inference for certain levels. Using sessions is required only if you want to access some of the advanced features of NuPIC. For example, using sessions makes sense when running on clusters. This chapter lays the groundwork to the topics discussed in [Using the Numenta Runtime Engine: Advanced Topics](#), page 65.

In the last section, the chapter gives supplementary information about `RuntimeNetwork.run()`.

### Topics

- [Running HTM Networks: Options on page 42](#)
- [Understanding the Training Process on page 43](#)
- [Using the Session API to Run Your HTM Network on page 44](#)
- [What `RuntimeNetwork.run\(\)` Does, page 51](#)
- [Accessing Session Information at Runtime on page 53](#)

## Running HTM Networks: Options

---

You perform training and inference by running the HTM Network. There are a number of options:

- Use the `RunBasicNetwork()` and `RunBasicNetwork()` helper functions used by the Bitworm example.
- Create a `RuntimeNetwork` instance from the `Network` you created, then call `RuntimeNetwork.run()`. See Running the HTM Network on page 55 in *Getting Started With NuPIC* for a discussion and examples.
- Use a session to run the HTM Network. See [Using the Session API to Run Your HTM Network on page 44](#).
- For large HTM Networks, advanced users can use multi-CPU machines or clusters. See [Using the Numenta Runtime Engine: Advanced Topics on page 65](#).

After you have defined the structure of the HTM Network, you perform training and inference. During training, there are two choices:

- **Supervised learning** — If you know which of your data points belong to which category, you can train the HTM Network using a category file. After training is complete, the HTM Network has assigned each input data point to a category.
- **Unsupervised learning** — If you don't know the categories in your problem, the NRE still groups the data in your training data set. After training, the NRE will then assign each data point to one of the groups it decided on during training.

After the HTM Network has been trained, you can submit the trained network and new data to the NRE for inference.

## Understanding the Training Process

---

During training, the nodes in the HTM Network perform learning and inference. This section gives an overview of the process, which is illustrated in the `bitworm_session` example. In contrast to *Getting Started With NuPIC*, which uses `RuntimeNetwork`, the focus of this section is on sessions.

From the NRE's point of view, the training process consists of these steps:

1. The user loads an untrained HTM Network file. The file contains the classes, links, and other information that the NRE needs to load the network.
2. The user initializes the sensor with input data from a file. In many cases, the user also initializes the category input data.
3. The script that runs the HTM calls the `Session` constructor, then calls `Session.start()`. In response to the call, the NRE creates a session and creates the HTM Network, the specifies node instances, and the links.
4. The session feeds the sensor data file to the HTM Network.
5. The NRE trains the HTM Network, one level at a time:
  - a. Trains the bottom level using the input data.
  - b. Performs inference on the bottom level, sends the resulting groups to the next level and performs training on that level.
  - c. Performs inference on Level 2 and performs training on Level 3, and so on.
6. If a category file was submitted, the classifier node uses the category information during learning. Otherwise, the classifier node groups the data based on the input data and the parameter settings.

## Using the Session API to Run Your HTM Network

This section explains how you can use the `Session` API to run your HTM Network on a single CPU, using fragments from the `bitworm_session` example.

Starting the session and running the HTM Network are two separate tasks. You must perform all session startup tasks before you can run the network.



Before you can run the HTM Network, you must configure its structure. See [Constructing an HTM Network](#) on page 41 in *Getting Started With NuPIC* for information.

### Starting the Session

Starting the session includes creating the `Session` object, adding the required files to the session, and calling the `start()` method.

#### To start the session:

1. Import the session-related classes:

```
from nupic.network import Session, SessionConfiguration
```

2. Create a session instance:

```
mySession = Session("dir/subdir/mysession")
```

This call returns a `Session` object; however, it does not start the NRE yet. Upon creating the session, the system creates a directory called `mySession.<num>.local_bundle` where it stores files related to the runtime engine. For the example above, a directory called `dir/subdir/mysession.1.local_bundle` is created. See [Sessions and Session Bundles on page 49](#) for background information.



You can customize the session by setting `SessionConfiguration` parameters before you actually start the session. See [SessionConfiguration Object Methods on page 81](#).

3. Add the files the session needs, such as training data and category files, using the `addFiles` method, which has the following prototype:

```
mySession.addFiles (<origin>,<destination> )
```

The `addFiles()` method copies the file(s) specified by `<origin>` to `<destination>`. If no destination is specified, it copies the files into the main session bundle directory, and preserves the original file names. Here's a code fragment from the Bitworm example:

```
session.addFiles("training_data.txt")
session.addFiles("train_catsensor.txt")
```

4. Start the session.

```
mySession.start()
```

This command launches the runtime engine with the default configuration inside the bundle. The bundle's top level is now considered the working directory. The log files

are placed in a subdirectory called `session_log`. See [Table 3; Session log files, on page 55](#).

There is a log file for each process. The Supervisor and each NP have a separate log file. These files are named `logS0.txt`, `logN1.txt`, and so on. A file called `stdout.txt` contains the combined information from all processes.

The call to `start()` places a single script file called `launch.py` into the bundle's `session_resources` subdirectory. This script is complex, and is kept in the bundle to facilitate troubleshooting.

## Running the HTM Network

This section first gives an overview of low-level details of running an HTM Network. It then illustrates how the `bitworm_session` example actually runs the network.

### Low-level Session Interaction

A program that does not want to take advantage of the higher-level APIs can run the HTM Network as follows:

1. Load the HTM Network file:

```
mySession.loadNetwork(<filename>)
```

Here, `<filename>` refers to an HTM Network on your local file system that you've already created. See [Constructing an HTM Network on page 41 in \*Getting Started With NuPIC\*](#).

If you'd like to avoid copying the network into the session bundle, you can use the optional `copyHint` parameter to `loadNetwork()`, which defaults to `true`.

```
mySession.loadNetwork(self, path="net.xml", copyHint)
```

If you're running your HTM Network remotely, the network must be copied.

2. Interact with the nodes in the HTM Network. You can call the following methods:

---

`Session.execute`  
(`node_regx`, `command`)

Call `Session.execute()` to send an execute command directly to a set of nodes. For example:

```
session.execute('level1', ('setInference', '1'))
session.execute('level1', 'resetHistory')
```

For help for all Supervisor commands, type the following at the Python command line:

```
from nupic.network import Session
mySession = Session('test')
mySession.start()
mySession.sendRequest('help')
```

To get help on a specific command, call `mySession.sendRequest('help <cmdName>')`.

---

---

<code>Session.disableNodes</code> ( <code>node_regx</code> )	When the NRE runs, it executes each enabled node's <code>compute()</code> method. By default, all nodes are enabled. The <code>disableNodes()</code> / <code>enableNodes()</code> methods disable or re-enable the specified nodes. <code>node_regx</code> is a regular expression that selects nodes.
<code>Session.enableNodes</code> ( <code>node_regx</code> )	You can disable/enable individual nodes within levels. For example, during training, the Pictures example program enables one node at a level and disables the rest for faster performance. Usually, you enable individual nodes by first disabling all nodes, and then individually re-enabling desired nodes, but that approach is not required.

---

<code>Session.compute</code> ( <code>num_iter</code> , <code>wait</code> )	Cycles through enabled nodes the specified number of times. Does not return until all <code>compute()</code> methods finish.  The <code>wait</code> parameter is advanced. If set to <code>True</code> (the default), the call does not return until all computation completes. If <code>False</code> , the call returns immediately, even though computation has only been started. Changing this parameter is not recommended.
---	--

---

3. After computation is complete, you can save the trained HTM Network to a file.

```
Session.saveRemoteNetwork(<filename>)
```

This method tells the NRE to save the current state of the network to an HTM Network file. The file is stored inside the bundle. Specify a filename that is relative to the bundle directory.

4. Call `stop()` to stop the NRE.

```
Session.stop()
```

5. Copy the HTM Network file to a location outside the bundle.

```
Session.getLocalBundleFiles(<origin>, <destination>)
```

Extracts the specified file(s) from the local bundle and puts it in the specified directory (local directory by default). Identical to the Unix `cp` command.

6. If you wish, you can tell the session to delete the local bundle upon completion:

```
Session.setCleanupLocal()
```

By default, the session does not clean up the bundle directory, so that users can examine its contents for debugging purposes. But if you plan to copy any files you need using `Session.getLocalBundleFiles()`, you can use `Session.setCleanupLocal()` to tell the session to remove the bundle when it is done. The bundle is removed only when the `session` object is deleted (which occurs if your script finishes, or the `session` instance goes out of scope, or you delete the instance manually). The bundle is not deleted immediately upon calling `Session.setCleanupLocal()`, nor upon calling `Session.stop()`.



See [How to Use NuPIC in Complex Configurations on page 76](#) for information on running in a multiprocessing environment.

### Performing Learning in the Bitworm Example

In the Bitworm example, the training script `TrainNetwork.py` performs the following tasks:

1. Disables all nodes.

```
session.disableNodes()
```

By default, all nodes are enabled after the HTM Network has been loaded. Bitworm then enables nodes selectively.

2. Enables learning for the Sensor and the Level1 nodes.

```
session.enableNodes("Sensor")
session.enableNodes("Level1")
```

3. Runs the `compute()` method of all enabled nodes, once for every input vector.

```
session.compute(numVectors)
```

4. Disables learning and enables inference for the Level1 node.

```
session.execute('Level1', ['setLearning', '0'])
session.execute('Level1', ['setInference', '1'])
```

### Performing Learning and Inference in the Bitworm Example

As the second part of training, you turn on inference for Level 1 and learning for Level 2. While Level 1 is in inference mode, it passes the information it deduced to the next level. Level 2 can then perform learning on those outputs.



Switching between learning and inference mode for the different levels is the appropriate way to train your network. Each Level has to complete learning before it can go into inference mode and hand off its data to the next level. Each level is trained once.

Higher-level tools (`RuntimeNetwork`, `RunBasicNetwork()`) enable and disable nodes automatically. The `Session` API requires you enable and disable explicitly.

The program implements this as follows:

1. Returns the sensor and category sensor to the beginning of the training data.

```
session.execute("Sensor", ("seek", "0"))
session.execute('CategorySensor', ("seek", "0"))
```

2. Enables the Level 2 node (the Level 1 node is already enabled). With inference enabled earlier for Level 1, turn on learning for Level 2.

```
session.enableNodes("Level2")
session.execute('Level2', ['setLearning', '1'])
```

3. Runs the training cycles for Level 2 and finishes learning. Because the Level 2 node in Bitworm is a `Zeta1TopNode`, it performs classification as part of learning, that is, it attempts to map the input from the previous level to the categories in the category file.

```
session.compute(numVectors)
```

4. Turns off learning and turns on inference for the Level 2 node.

```
session.execute('Level2', ['setLearning', '0'])
```

### Using `Session.execute()` to Access Nodes or Execute Commands at Runtime

You can retrieve and set node parameters at runtime using `Session.execute()`, for example:

```
mySession.execute('myNode', ['setParameter', 'transitionMemory', '4'])
mySession.execute('myNode', ['getParameter', 'coincidenceCount'])
```

You can also execute certain commands, for example:

```
mySession.execute('myNode', 'pruneCoincidences')
```

Use `nodeHelp` for the individual nodes for detailed information.



## Sessions and Session Bundles

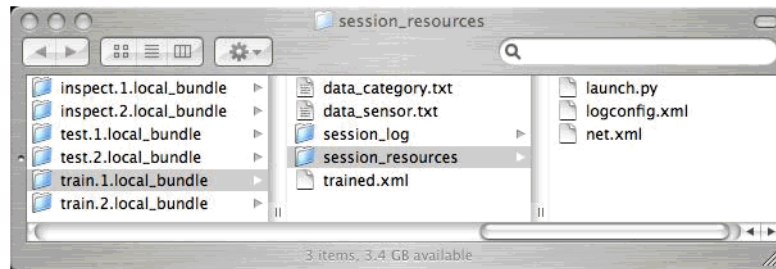
A typical session makes use of several files. To organize these files for easy transfer, analysis, and cleanup, the session manages a session bundle: a single file system directory created by the `Session` instance. The bundle is created immediately on session instantiation. `Session` methods allow you to add files to the bundle, extract files from the bundle, copy the bundle to and from a remote host, and clean up the bundle when it is no longer needed.

The `Session` instance creates a local bundle. The bundle holds all files associated with a single conceptual NRE session. The bundle includes configuration scripts, working inputs and outputs, and log files on the local file system. You must specify the location of the bundle on the file system when you create the `Session` instance using the constructor.

Because the runtime engine can be launched on a remote host, the local bundle might not be accessible to the remote NRE processes. Instead, the session instance copies the local bundle to the remote host with a file synchronization utility (`rsync` on Unix-like systems). This works as long as the local machine has secure-shell (`ssh`) access to the remote host.

The following screen shot shows the bundle layout for a typical session:

Figure 8 Session Bundle Example



The bundle directory functions as a working directory for the NRE. All relative paths used by the NRE refer to paths starting in the bundle directory. Files written to or read from relative paths refer to files stored in this directory. This includes paths used to load and save network and data files.

By default, the bundle directory is populated with a number of files and directories when you create the session:

- **The `session_resources` subdirectory** holds configuration files and other immutable files that the NRE needs, but that do not change during the lifetime of the session. The launch script (`launch.py`) is saved into this directory.

When there is a remote bundle, this directory does not need to be synchronized after startup because the NRE does not modify files in the `session_resources` directory.

- **The `session_log` subdirectory** holds log outputs from the NRE processes. The log subdirectory is always synchronized in the opposite direction, from the remote bundle to the local bundle. See [Table 3; Session log files, on page 55](#).

When there is a remote bundle, the NRE always copies this directory back to the local bundle upon NRE shutdown.

### Bundles and Remote Hosts

If the NRE is launched on a remote host, the main bundle directory is synchronized just before launch from the local bundle to the remote bundle, in case input files are stored in the local bundle. At any time during the session lifetime, you can add new files to the local bundle's main directory and synchronize to the remote side using the `Session` API. Similarly, if periodic updates of the remote outputs are needed locally, you can explicitly synchronize the local bundle with the remote bundle. When the remote NRE is shut down, the main directory and the log directory are synchronized back from the remote bundle to the local bundle. In this way, the local application can access all NRE output files.

### Loading Networks Without Copying them into Bundles

You can load an HTM Network without copying it into a bundle by using the `copyHint` flag to `Session.loadNetwork`:

```
mySession.loadNetwork(self, path="net.xml", copyHint=True)
```

When `copyHint` is true, (the default), the HTM Network file is copied into the bundle. When `copyHint` is false, the network is not copied into the bundle. This parameter is only a hint. If there is a remote bundle, the HTM Network File must be copied regardless of the value of the flag.

## What RuntimeNetwork.run() Does

While all other sections in this chapter dealt with sessions, this section gives some advanced information on `RuntimeNetwork`, which is discussed in *Running the HTM Network*, page 55 in *Getting Started With NuPIC*.

When you call `RuntimeNetwork.run()`, the NRE runs computation on a selected subset of the network for the specified number of iterations.

### Simple Scenarios

In the simple case the user specifies an iteration number and optional selection or exclusion arguments. The names specified in the selection and exclusion arguments can refer to individual nodes or regions. The following table shows some examples:

Example	Description
<code>myNetwork.run(1000)</code>	Run the entire network for 1000 iterations
<code>myNetwork.run(1000, ["sensor", "level1"])</code>	Run the sensor node and the region level1 for 1000 iterations
<code>myNetwork.getElement("sensor").run(1)</code>	Run the sensor node for one iteration
<code>myNetwork.run(1, exclusion=["effector"])</code>	Run the entire network, except for effector, for one iteration
<code>myNetwork.run(10, ["sensor", "level1", "level2[0]"], exclusion=["level1[1,0]"])</code>	Run sensor, the region level1 with the exception of node level1[1,0], and the node level2[0], for ten iterations.

### Scenarios With Run Policies

NuPIC supports a wide variety of node types and learning algorithms. Different learning nodes might require complex run schemes during training. To support this, the run method can accept a `RunPolicy`.

The following table shows some examples with `zeta1Node` an older node that is still supported:

Example	Description
<code>myNetwork.run(TrainPhase("level2", 1000), selection = ["sensor", "level1", "level2"])</code>	Train the level2 region for 1000 iterations using the default algorithm. 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.

Example	Description
<code>myNetwork.run(TrainPhase("topLevel", 1000), exclusion=["effector"])</code>	Train the top level for 1000 iterations using the default algorithm. All nodes in the network will be selected, with the exception of <code>effector</code> .
<code>myNetwork.run(UntilException())</code>	Run all nodes in the network continuously until an exception occurs. Common run policies include <code>TrainPhase</code> , <code>SimpleCompute</code> , and <code>UntilException</code> . See the Python documentation for these policies for more details.

### **RuntimeNetwork.run() Parameters**

You can call `RuntimeNetwork.run()` with the following parameters:

Parameter	Type	Description
<code>runPolicy</code>	integer or <code>RunPolicy</code>	If an integer, specifies the number of iterations to run on the selected nodes. If a run policy, hands off the <code>RuntimeElement</code> , its attached <code>Session</code> , and the selection information to the run policy, which determines how many iterations to run and what nodes to enable.
<code>selection</code>	list of strings (optional)	Relative names of the elements to enable. If <code>None</code> , the entire <code>NetworkElement</code> will be enabled.
<code>exclusion</code>	list of strings (optional)	Relative names of the elements to disable. Only necessary to disable elements that would be enabled through the current <code>NetworkElement</code> or the selection parameter.

## Accessing Session Information at Runtime

---

You can interact with sessions and examine node content at runtime. You can also later access the logs generated by the NRE.

### Interacting with Sessions

You can interact with sessions as follows:

- Retrieve individual outputs, using the `nodeOPrint` Supervisor command. You can call Supervisor commands using the `Session.sendRequest()` method, for example:

```
mySession.sendRequest('nodeOPrint <nodeName>')
```

For help for all Supervisor commands, type the following at the Python command line:

```
from nupic.network import Session
mySession = Session('test')
mySession.start()
mySession.sendRequest('help')
```

To get help on a specific Supervisor command, call `mySession.sendRequest('help <cmdName>')`.

- Call `Session.execute()` to send a command directly to a set of nodes. For example:

```
mySession.execute('level1', ('setInference', '1'))
mySession.execute('level1', 'resetHistory')
```

You can call `mySession.execute('<nodeName>', 'getCoincidences')` to retrieve the coincidence matrix from the node.

- Instead of issuing session commands repeatedly, you can iterate through them as follows:

```
for k in range(0, numIterations):
    compute(1)
    self.session.sendRequest('nodeOPrint sensor')
    self.session.sendRequest('nodeOPrint level1\[0,0\]')
```

Then review the `session_log/stdout.txt` file to verify that you are getting reasonable results.

- Run `mySession.compute(1)` to perform a single computation on all nodes, then retrieve information from a node to see how it changed during that one computation.

### Examining Node Content

You can check the number of coincidences and groups at each level and after each chunk of training data.

```
self.session.execute('level1\[0,0\]', 'getNGroups')
self.session.execute('level1\[0,0\]', 'getNCoincidences')
```

Check the ratio of coincidences and groups at each level.

Using Visualizer, plot coincidences at each level and plot groups (see Using HTM Network Visualizer, page 74 in *Getting Started With NuPIC*). In an HTM that works well, you should expect a balanced histogram of group sizes. You usually don't want one or two huge groups and dozens of twin or singleton groups.

## Look At Output Information

This section explores a few things you can check after you have run your application.

### Examining Scripting/Session Commands

When you're not sure what's happening, you can look at the log files and any print statements you included in your program. For example, you can check the `session_resources/launch.py` script and the corresponding `session_log/launch.txt` file to see whether startup proceeded as expected.

To see whether things are scheduled appropriately, see the session bundle's `/log` directory for Supervisor commands.

Check whether each level is being trained appropriately. Each level should go through learning, then inference. If you're using `RuntimeNetwork`, these steps are followed automatically. If you're performing training and inference explicitly, see the `bitworm_session` example's `TrainNetwork` script.

Examine numbers at each level to see whether a level or a node is not getting scheduled.

### Log Files

Each time you use the NRE, information is added to the logs. This information can be useful, for example, if you don't get any results.

The logs are especially useful to plug-in developers. Because plug-in developers create their own nodes, they might experience NRE crashes if errors exist in their nodes.

After a session has stopped, the following log files are available in the local session bundle directory (`session_log`). Currently, the `stdout.txt` log is the most useful log file. The `rrlog.txt` log is also useful, it records node requests and responses.

Table 3: Session log files

File	Description
<code>rrlog.txt</code>	Transcript of all requests sent to the Supervisor and all responses received back. Also useful for troubleshooting and for confirming your HTM Network is operating as expected.
<code>launch.txt</code>	Transcript of the commands issued to launch the Supervisor.
<code>stdout.txt</code>	Integrated “standard” output from the launcher, MPI, the Supervisor and all Node Processors. This file is especially useful if you are developing your own sensor, effector, or learning node plugin.
<code>log.S0.txt</code>	Log file generated by the Supervisor while it was running.
<code>log.N1.txt</code>	Log files generated by the first node processor while it was running.
<code>log.N2.txt, ...</code>	Log file generated by additional node processors if they existed.

## The launch.py File

The call to `Session.start()` places a single script file called `launch.py` into the bundle's `session_resources` subdirectory. This script is complex, and is kept in the bundle to facilitate troubleshooting. The script might have useful information if your session does not seem to start at all, or if it starts but does not seem to be processing the input.

If you're using `RuntimeNetwork`, the session created by `RuntimeNetwork` creates the script.





## 4 ***Scheduling Node Processing***

---

This chapter first explains how a scheduler manages the flow of data through an HTM Network. The second section discusses using the basic scheduler or the pipeline scheduler with multiple NPs.

[Using the Numenta Runtime Engine: Advanced Topics, page 65](#) explains how to use multiple NPs.

### **Topics**

- [Understanding Scheduling, page 58](#)
- [Different Schedulers with Multiple NPs on page 59](#)
- [Profiling and Load Balancing on page 63](#)

## Understanding Scheduling

---

A scheduler manages the flow of data through an HTM Network (see [Inside a Learning Node: How Learning and Inference Happen on page 31](#)). The scheduler, which is part of the NRE, determines the order in which the nodes' `compute()` methods are called. This section discusses how schedulers control execution and introduces the available schedulers.

### Scheduler Overview

In a hierarchical HTM Network, there are data update dependencies. For example, mid-level nodes might require data calculated by bottom-level nodes. The NRE scheduler

- Is responsible for coordinating node computations so that data dependencies are satisfied.
- Is responsible for coordinating computations across NPs when there is more than one NP.
- Keeps track of nodes that have been explicitly enabled or disabled, and schedules only enabled nodes.

When you schedule a complete scheduling sweep through a simple HTM Network, the `compute()` method on each enabled node is invoked once.

### Supported Schedulers

NuPIC includes two schedulers, the basic scheduler and the pipeline scheduler.

- **Basic Scheduler** — The basic scheduler schedules computation in order of a non-negative integer phase assigned to each node. The node creation methods assign a default phase (see *Constructing an HTM Network* on page 41 in *Advanced NuPIC Programming*). In a simple HTM, the sensor is typically assigned to phase 0 (zero), bottom-level nodes are assigned to phase 1, and so on. Nodes in the same phase might be computed in any order.

The basic scheduler can schedule nodes on one or more node processors, and ensures that the computation result is the same as if all nodes were on a single NP.

If you want to assign a non-default phase, you can use the `CreateNode()` API.

- **Pipeline Scheduler** — If you're running your HTM Network on more than one NP, you can take advantage of the pipeline scheduler. Using the pipeline scheduler makes your HTM application run faster in a multi-CPU setting but requires some additional code. See [Using the Pipeline Scheduler With More Than One NP on page 61](#).

## Different Schedulers with Multiple NPs

This section discusses how different schedulers interact with multiple NPs. Using multiple NPs is discussed in detail in [Using the Numenta Runtime Engine: Advanced Topics](#), page 65.



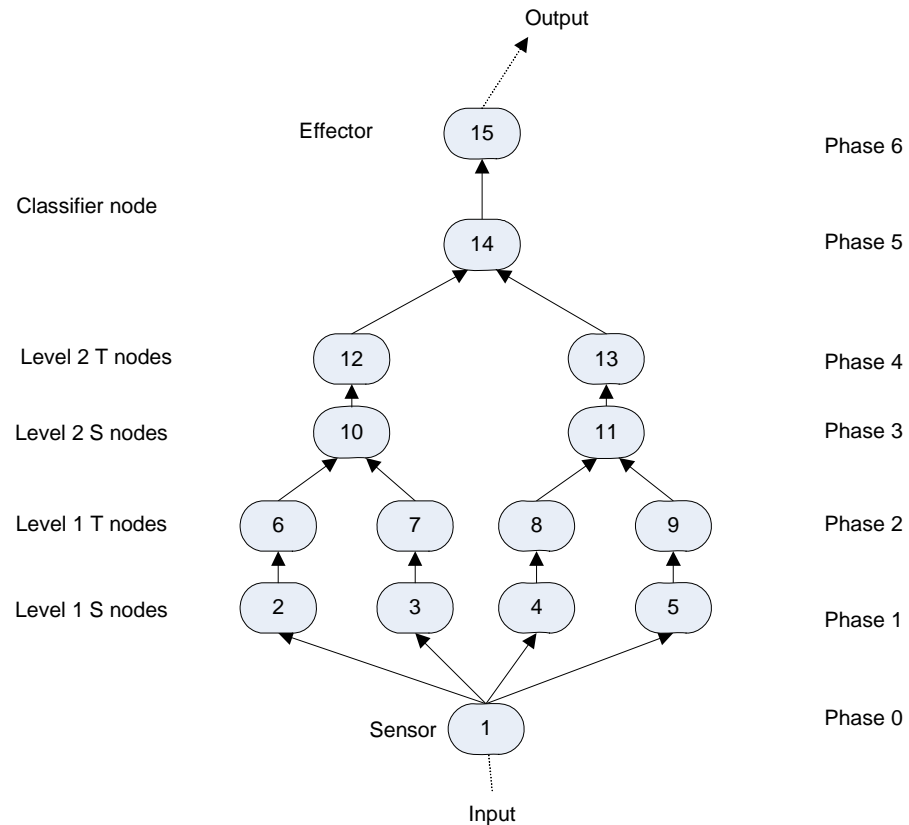
In most cases, the potential performance improvement of using multiple processors is much greater with the pipeline scheduler than with the basic scheduler.

### Using the Basic Scheduler with Multiple NPs

By default, the basic scheduler is enabled. This section explains how to use the basic scheduler when there is more than one NP. For an introduction to scheduling, see [Understanding Scheduling](#) on page 58.

The phase assigned to each node determines the order in which the basic scheduler updates the nodes. Nodes in phase  $N+1$  are always computed after nodes in phase  $N$ . With the basic scheduler, two nodes in the same phase can be computed in parallel if they are on different NPs, but nodes in different phases cannot be computed in parallel.

Figure 9 Example Network for Basic Scheduler, Multiple NPs



In the example in [Figure 9](#), assume that phases have been assigned as follows:

- sensor = phase 0
- level 1 nodes = phase 1 and 2

- level 2 nodes = phase 3 and 4
- classifier node = phase 5
- effector node = phase 6

The phase assignments capture the data dependencies that are implicit in the network. For example, node 10, in phase 3, is not computed until new data are available from its upstream nodes 5 and 6, in phase 2.



Advanced note: You might sometimes need to propagate signals back down the hierarchy. You can do so by assigning more than one phase to some nodes. For example, you might assign sensor = phase 0; level 1S = phases 1, 11; level1T = 2, 10, and so on. This assignment results in computation of sensor/level1/level2/level3.../classifier/...level3/level2/level1. You can specify multiple phases as a space-separated list of integers in a node's phase property.

Suppose you have two NPs and an HTM Network with phases 0-6 as shown in Figure 9 on page 59. Nodes 1, 2, 3, 6, 7, 10, 12, 15 are on NP 0, and nodes 4, 5, 8, 9, 11, 13, 14 are on NP 1. Then the computation proceeds as follows:

Phase	NP0	NP1
0	Compute 1	Idle
1	Compute 2, 3	Compute 4, 5
2	Compute 6,7	Compute 8,9
3	Compute 10	Compute 11
5	Compute 12	Compute 13
6	Idle	Compute 14
7	Compute 15	Idle

One might hope for a performance improvement of nearly  $N$  when using  $N$  NPs. However, for this small HTM Network, one NP is idle in three out of seven phases, so the potential performance improvement is much less than  $N$ .

Several factors limit the actual performance improvement you can get using multiple NPs with the basic scheduler. These factors include:

- Load balance at each phase — Load balance is a measure of the relative computational load on two or more processors. The more nodes the better. Larger HTM Networks are better able to utilize multiple processors because they can be better load balanced.
- Relative amount of computation between phases.
- Synchronization and communication overhead — Whenever there is more than one NP, the NRE must synchronize and exchange node outputs after every phase. Communication between NPs is efficient, but can hurt performance if the amount of computation is not significantly larger than the communication overhead. For small HTM Networks with relatively simple computations (including most of the Numenta

example programs) the amount of computation is so small that there is little benefit, or even a negative impact, from parallelization.

Fatter nodes (nodes with more computation) reduce communication overhead, resulting in improved performance.

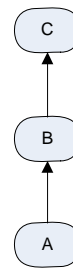
For the Learning Nodes distributed with NuPIC, the amount of computation grows with the number of coincidences learned by a node, so fully trained nodes are fatter than untrained nodes.

## Using the Pipeline Scheduler With More Than One NP

The pipeline scheduler is an advanced scheduler that makes load balancing easier and enables greater parallelism. Other than computation time, an HTM Network using the pipeline scheduler performs identically to the basic scheduler.

### Pipeline Scheduler Example

Consider a simple 3-level network with nodes A, B, C. A is a sensor, B is a single Level 1 spatial pooler node, and C is a single Level 1 temporal pooler node.



At each iteration, the sensor computes a single image. The B computation uses the output of A, and the C computation uses the output of B. With the basic scheduler, this computation must be performed serially. The pipeline scheduler allows this computation to be parallelized through a technique known as pipelining. Consider the following series of computations:

	A	B	C
1	Computes, produces image 1	Idle	Idle
2	Produces image 2.	Processes image 1.	Idle
3	Produces image 3.	Processes image 2.	Processes B's output from image 1
4	Produces image 4.	Processes image 3.	Processes B's output from image 2
5	Produces image 5.	Processes image 4.	Processes B's output from image 3
6	Idle	Processes image 5.	Processes B's output from image 4
7	Idle	Idle	Processes B's output from image 5

In steps 3-5, all nodes can be computed in parallel. Steps 1 and 2 are known as “pipeline fill” and steps 6 and 7 are known as “pipeline drain”.

To make pipelining possible, the pipeline scheduler uses a technique called double buffering. In the example above, when B is processing image 2, A might overwrite the input with image 3. To avoid overwrites, each node has two outputs called x and y. During one iteration, nodes write to the x buffer and read from the y buffer. During the next iteration, nodes write to the y buffer and read from the x buffer.

The additional complexity of the pipeline scheduler is hidden from you: NuPIC automatically fills and drains the pipeline.

### Pipeline Scheduler Considerations

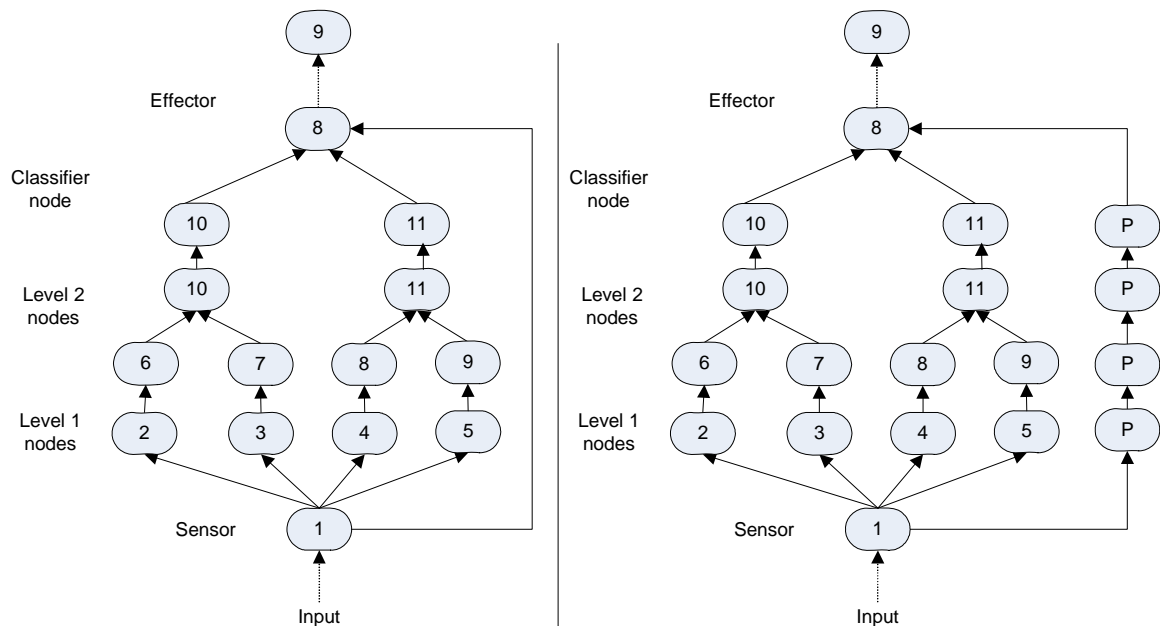
When using the pipeline scheduler, be sure to consider these points:

- 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.
- The pipeline scheduler can be used with feed-forward networks.
- The available concurrency with the pipeline scheduler is the total number of nodes.
- The pipeline scheduler requires less inter-process synchronization per iteration than the basic scheduler.
- Pipeline throughput is limited by the slowest NP in the pipeline. One NP might be slower than the others because of load imbalance.

Most HTM Networks behave identically with the basic and the pipeline schedulers.

Networks with level-skipping connections might behave differently. In a level-skipping connection, the output from a low-level node skips levels to go to a high-level node. The Pictures example avoids level skipping by using two pass-through nodes that copy their input to their output. In most cases, using pass-through nodes is recommended.

Figure 10 Level-skipping HTM Network (Left) and HTM Network Using Pass-through Nodes (Right).



## Profiling and Load Balancing

---

Load balancing appropriate for the basic scheduler is fairly straightforward. Assuming each node at a given level needs to do approximately the same amount of computation, dividing the nodes at a given level among NPs is a reasonable load-balancing strategy.

For the Pipeline scheduler, it might not be evident how much time is spent in a level 1 node vs. a level 2 node vs. a level 3 node. The NRE provides a simple way to profile your HTM Network for basic load balancing.

### To profile your HTM Network:

1. Invoke `mySession.sendRequest("profile -enableProfiling")` to turn on profiling.
2. Run your network for some number of iterations.
3. Invoke `mySession.sendRequest("profile -disableProfiling")` to turn off profiling.
4. Invoke `mySession.sendRequest("profile -report")` to generate a report of the cumulative time spent in the `compute()` method of each node.



NuPIC currently does not include tools for automatically rebalancing a network.

The images framework also includes some profiling. See the images documentation for more information.







# 5

## ***Using the Numenta Runtime Engine: Advanced Topics***

---

This chapter discusses using the NRE in advanced configurations: remotely, in multiprocessing mode, and on a cluster.

### **Topics**

- [Introduction and Terminology on page 66](#)
- [NRE Process Structure with Multiple NPs on page 67](#)
- [Hardware Configurations on page 68](#)
- [Running in Parallel: Experiment Mode on page 70](#)
- [Running in Parallel: Large Problem Mode on page 71](#)
- [Setting up a Cluster to Run NuPIC on page 73](#)
- [How to Use NuPIC in Complex Configurations on page 76](#)
- [SessionConfiguration Object Methods on page 81](#)

## Introduction and Terminology

---

Many computers manufactured today have more than one processor. Some systems have more than one single-core processor; others have one or more multi-core processors. Clusters of computers extend the potential size of a system to tens of thousands of processors. The NRE can use more than one processor effectively, and can take advantage of almost any type of multi-processor system, from a single dual-core chip to a large cluster.



Parallel computing with the NRE is an advanced topic. While it is easy to turn on parallelism in the NRE, you are unlikely to see better performance unless you understand how multiprocessing works, have an HTM Network that is suitable for multiprocessing, and take into account the factors discussed in this and the previous chapter.

### Terminology

This chapter uses the following terms:

- **CPU** — A CPU is a single-processor core. A dual-core processor has two CPUs, two single-core processors are two CPUs, and a cluster of five systems, each with two dual-core CPUs, has 20 CPUs.
- **Process** — A process is an operating system process on Mac OS, Linux, or Microsoft Windows. The NRE does not use threads, so speedup from parallel execution is always achieved through the use of multiple processes.
- **Run in parallel** — The term "run in parallel" means "execute concurrently on more than one CPU" with the goal of reducing overall execution time.
- **Experiment mode and large problem mode** — There are two general classes of parallel operation with NuPIC. Running multiple HTMs concurrently (experiment mode) and running a single HTM (large problem mode).

### Singe-NP Process and Multiple NPs

The NRE can run in SP-NuPIC or MP-NuPIC mode.

- **SP-NuPIC (single-process)** — The default and most common use of the NRE is to have a single NP. For that case, computations are performed serially, and additional CPUs do not improve performance. When there is a single NP, everything runs inside a single operating system process. Python tools, the supervisor, and node processor are contained within this process.

This mode of running NuPIC is called SP-NuPIC. Most of the examples run in SP-NuPIC mode.

- **MP-NuPIC (multiple processes)** — Running with more than one NP makes sense if you want to take advantage of more than one CPU, or if you want to run remotely. The focus of this chapter is MP-NuPIC.

## NRE Process Structure with Multiple NPs

To take advantage of more than one CPU, you must run with more than one NP. NuPIC automatically starts several operating system processes when you request more than one NP. This mode of running NuPIC is called MP-NuPIC.

The details presented here are specific to Unix-like (OS X, Linux) systems. On Microsoft Windows, some of the details are different.

When you use the NRE with two NPs (e.g. by running the images example on multiple NPs) and display the processes, for example, using the `ps` command, you see the following processes.

Table 4: NRE Processes

Process	Description
The Python program that controls the NRE session.	
Three processes named <code>numenta_runtime</code>	One Supervisor process and two NP process.
launcher process	Numenta process responsible for starting the NRE.
orterun and orted processes	Processes associated with the underlying process management and inter-process communication interface provided by Open MPI (see below).

Of these, the NP processes do the main computational work. None of the other processes are CPU intensive. In some cases the Supervisor might use a large fraction of a CPU, but it gives priority to any other process that needs to run, so that the Supervisor does not significantly impact performance in most cases.

The Supervisor and NPs communicate with each other using the Message Passing Interface (MPI) API. On Unix-like systems, Numenta uses the Open MPI implementation of MPI. See <http://www.open-mpi.org> for more information. Open MPI, which is bundled with NuPIC, provides high-performance inter-processor communication. The Open MPI capability to use high-performance cluster interconnects such as InfiniBand and Myrinet is not yet enabled in Numenta software.

The focus of the rest of this chapter is on the Supervisor and NPs — the other processes are not usually relevant to NuPIC users.



You can force NuPIC to run in MP-NuPIC mode even if you're using just one NP by calling:

```
SessionConfiguration.setMultiProcessMode()
```

## Hardware Configurations

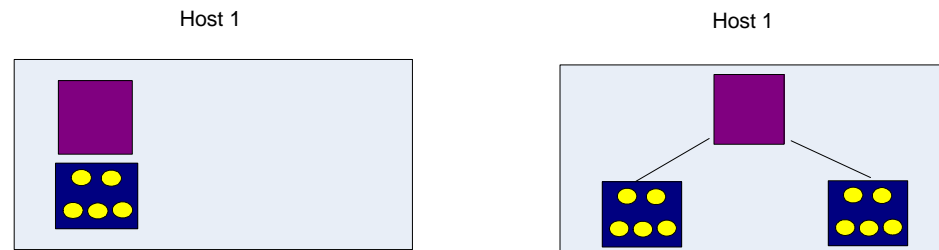
This section describes different hardware configurations and considers multiprocessing issues in context.

### Single-CPU Machine

In the simplest case, the NRE runs on a single-CPU machine. There are two ways of running the NRE in that case:

- SP-NuPIC mode — The Supervisor, one NP, and the control program all run on the same CPU in a single process. This is the default mode.
- MP-NuPIC mode — If you request more than one NP, the NRE automatically runs in MP-NuPIC mode. In that case, you see the Supervisor process, the two NP processes, and the other processes discussed in [NRE Process Structure with Multiple NPs on page 67](#).

While it is possible to run more than one NP on a single CPU, performance suffers because the CPU has to be shared between the NPs. Nevertheless, multiple NP instances might be useful during prototyping.



One Supervisor, one NP in one process (SP-NuPIC)

On Supervisor process, two NP processes (MP-NuPIC)

### Multi-CPU Machine

You should have only one NP per CPU. For example, on a system with a dual-core chip, you could invoke the NRE with two NPs to run one HTM Network. On the same system, you could invoke two instances of the NRE, each with one NP, to run two HTM Networks. In both cases the Supervisor and Tools instances do not consume significant CPU resources and HTM Network computation can efficiently use the available CPUs.

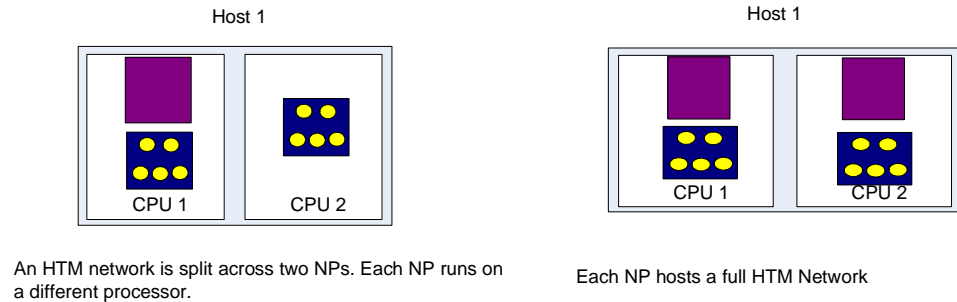
The performance you can obtain with multiple CPUs is limited by the load balancing and other issues discussed in [Using the Basic Scheduler with Multiple NPs on page 59](#). Performance might also be limited by hardware and operating system constraints.

- Processes running on different CPUs might compete for limited memory system resources.
- The operating system might move a process from one CPU to another, losing cached data and (on some architectures) increasing memory latency for the process to access its memory.

A complete discussion is beyond the scope of this guide.

The Supervisor processes, while using limited CPU resources, still use some resources, and the amount of resources used increases with the number of Supervisor processes on a system.

Figure 11 Different Configurations on Multi-CPU Machines

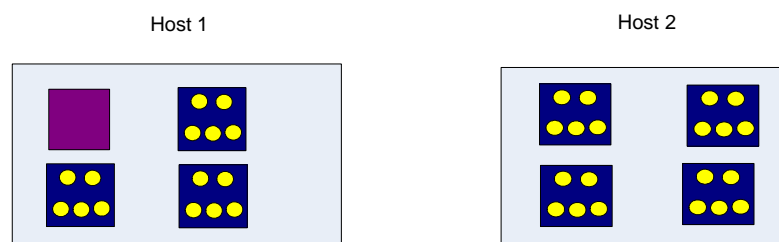


## Cluster (Unix-like Systems Only)

NuPIC supports a cluster of networked hosts. The main differences between using a cluster and using a multi-CPU machine are:

- You have to explicitly specify the names of the hosts that you want to use in the cluster.
- Communication overhead between different hosts in the cluster is much higher than communication overhead between two processes on the same host.
- If a single NRE is distributed across several hosts, only one of those hosts has a Supervisor process. Because the Supervisor takes some CPU resources, it might compete with NPs running on the same host, so that NPs on that host run slightly slower than NPs on other hosts. To avoid load balance problems, consider allocating a whole CPU to the Supervisor.

Figure 12 Running in a Clustered Environment



Running in a clustered environment; the Supervisor runs on one of the machines in the cluster.

See [Setting up a Cluster to Run NuPIC](#) on page 73 and [How to Use NuPIC in Complex Configurations](#) on page 76.

## Running in Parallel: Experiment Mode

---

The simplest way to take advantage of multiple CPUs is to run multiple instances of the NRE, each with a separate HTM Network.



Multiple NREs make sense only if you have multiple CPUs.

While this approach is simple (so simple that it is called “embarrassingly parallel” in the high performance computing literature) it can be effective, and avoids many of the potential pitfalls of parallel processing.

This approach is particularly useful in experiment mode. For example, you might want to train an HTM Network using different learning parameters, and see what values of the parameters have the best results.

You can use multiple NRE instances by launching multiple copies of your controlling Python script. To avoid possible problems, you must structure your script as follows:

- Each script must use a separate session bundle directory. Because the system creates new bundle directories (with .1, .2, etc. suffixes) automatically, this is not normally an issue.
- If a session writes to a file outside the session bundle, you must ensure that it uses a unique output filename.

While it is possible to start and control multiple sessions from within a single Python program, it is difficult to get good performance with this approach. Python itself is single-threaded (even when using Python “threads”), so Numenta does not recommend performing simultaneous computation from one Python script.

Multiple NRE instances are supported automatically by the NetExplorer framework, which can explore the parameter space by running several experiments in parallel. See [Using Numenta NetExplorer on page 100](#).

## Running in Parallel: Large Problem Mode

In some situations, especially when you work with a large problem, using a single NRE with more than one NP can improve execution speed.



The degree of improvement (if any) depends on the structure of your HTM, the scheduler you choose, the hardware on which you're running, and some other factors. See [Different Schedulers with Multiple NPs on page 59](#) for a discussion of potential issues.

You can start the NRE using `TrainBasicNetwork()`, or the `RuntimeNetwork` or `Session` interface.

In all cases, the network is automatically allocated to NPs in a round-robin fashion. If the HTM Network sizes can easily be divided across a small number of NPs, the default scheme also provides good default load balancing if you are using the basic scheduler.

### Using RuntimeNetwork in Large Problem Mode

You can create a runtime network that uses more than one NP by using the optional `numNodeProcessors` argument to `CreateRuntimeNetwork`. The following example creates a `RuntimeNetwork` with four NPs.

```
myRuntimeNetwork = CreateRuntimeNetwork (
                                <netfilename>.xml,
                                files = [<dataFiles>],
                                numNodeProcessors = 4)
```

### Using Sessions in Large Problem Mode

You can use multiple NPs using a `SessionConfiguration` object. This section uses the `Session` interface as an example.

1. Import the `Session` and `SessionConfiguration` classes:

```
from nupic.network import Session, SessionConfiguration
```

2. Create a `Session` object:

```
mySession = Session("dir/subdir/mysession")
```

This step is identical to the single-process session creation.

3. Add the files the session needs.

This step is identical to the single-process session bundle management, discussed in [Sessions and Session Bundles on page 49](#).

4. Build a `SessionConfiguration` object for storing advanced session parameters:

```
myConfig = SessionConfiguration()
```

5. In the simplest case, you specify only the number of node processors. This example starts four NPs:

```
myConfig.setNumNodeProcessors(4)
```

6. Start the session using the custom-configured `SessionConfiguration` as an argument:

```
mySession.start(myConfig)
```

After you've started the multi-process session, all training, shutdown, and bundle management methods are identical to their single-process counterparts.



NuPIC decides on which CPUs to run which NP. You don't currently have control over NP assignment.

More complete information about the `SessionConfiguration` can be found in [How to Use NuPIC in Complex Configurations on page 76](#).

## Using `TrainBasicNetwork()` in Large Problem Mode

If you want to use the helper functions in a multi-NP environment:

1. Create a `RuntimeNetwork` that includes node processor information.
2. Pass in that network when you call `TrainBasicNetwork()`.



## Setting up a Cluster to Run NuPIC

This section discusses setting up a cluster to run NuPIC.



Cluster setup is an advanced topic. You should have experience running NuPIC on a single host and working with HTMs before you try to use NuPIC on a cluster. You should also be proficient in Linux systems administration. (If you don't know the acronyms used in this section, you probably don't know enough to set up a cluster).

### Introduction to Cluster Setup

A cluster is a collection of hosts that are designed to operate as a single computational resource. NuPIC can be run on a cluster either in experiment mode (running multiple HTMs concurrently) or large problem mode (running a single HTM).

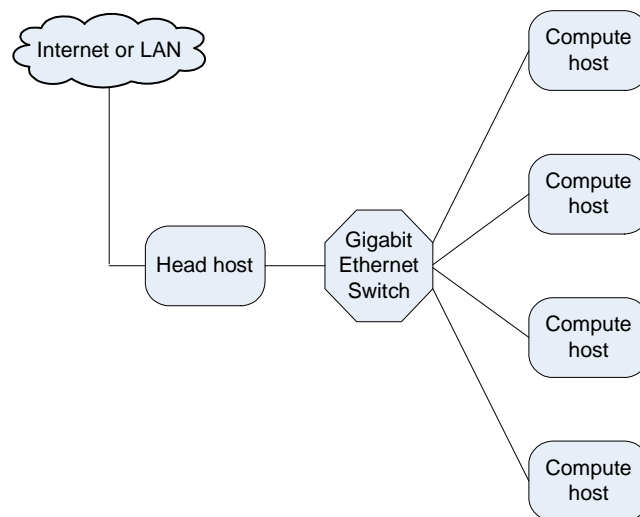
This section defines some basic terminology, describe the minimum requirements for running NuPIC, and make some suggestions for cluster setup.

#### Terminology

- **Cluster** — A cluster is a collection of *hosts*, networked together. In the field of cluster computing, these hosts are usually known as *nodes*, but to avoid confusion with an HTM Node, this document uses the term *host* instead.
- **Head host and compute host** — A cluster often contains a *head host* (also called *head node*) where you log in and work interactively, and a number of *compute hosts* whose role is computation (in this case, running the NRE).

A typical cluster configuration is shown in [Figure 13](#).

Figure 13 Typical Cluster Configuration



In the configuration above, the head host has two network interfaces. One interface connects to the Internet or local area network, and the other to the cluster. The only way to access the cluster is through the head host.

Many other configurations are possible, but we suggest this one for people who are new to clusters, because it has fewer security issues than a cluster in which every host is accessible from the Internet.

## Requirements

The following software configuration is required to run NuPIC in large problem mode and is strongly recommended for all uses of NuPIC on a cluster.

- NuPIC is enabled for clusters for all supported operating systems except Microsoft Windows. NuPIC has not been tested on darwin or darwin86 clusters. All hosts in the cluster must use the same operating system.
- A shared file system must be mounted by all hosts in the cluster at the same mount point. Typically this shared file system is an NFS file system exported by the head host, but does not have to be.
- UIDs must be common across the cluster. Usually cluster-wide UIDs are provided by LDAP or NIS.
- Passwordless `ssh` must be enabled between any two nodes in the cluster. There are many ways to configure `ssh` so that no password is required (see `ssh` documentation).
- NuPIC must be installed in the same location on all hosts (installing it in the shared file system makes this easier).

Additionally, we recommend the following:

- The network should be switched Gigabit Ethernet or faster. The network is probably not a bottleneck when running in Experiment Mode, but could be a bottleneck when running in large problem mode, depending on a number of factors as discussed elsewhere in this chapter.
- All compute hosts should be identical (same processor) when running in large problem mode. Using hosts with processors running at different speeds greatly complicates load balancing.
- All unnecessary services should be turned off on the cluster. Daemons for services (even unused services) compete with NuPIC for CPU time and degrade NuPIC performance.

For more information, consider reading the archives of the beowulf mailing list ([www.beowulf.org](http://www.beowulf.org)). There are also a number of books on setting up clusters — search for “beowulf cluster” at an online bookseller.

NuPIC includes an MPI implementation. You do not need to (and probably should not) install MPI on your cluster. Note that NuPIC uses TCP for inter-host messages, and does not take advantage of high-performance cluster interconnects.

## Cluster Performance Bottlenecks and Host Hardware

Users often ask what type of processor they should use, and how many cores should be in each host. These questions have no universal answers, and we can offer only general advice. We also have no recommendation for cluster vendor.

A single machine is limited in

- Memory bandwidth (constrains total performance, depending on algorithm details)
- Number of cores (constrains degree of parallelism)
- Amount of memory (constrains problem size)

When one of these factors is your bottleneck, running on a cluster might improve performance.

We strongly recommend the following:

- Do not run more `numenta_runtime` processes than cores. The number of cores is therefore a hard limit.
- Never run more than one instance of the NRE on a single host, even if there appear to be enough cores. Two two-process runtime engines use the same two cores, even in a quad-core machine, because NuPIC binds its processes to specific cores to prevent unnecessary context switching, cache invalidation, and other problems.

Because memory contention is difficult to measure, and because no simple specification can help you resolve memory contention, such bottlenecks are hard to resolve. Memory bandwidth is usually a problem before the number of cores becomes a problem. You might therefore achieve better performance running on two dual-core processors than on a single quad-core processor. Intel and AMD memory systems are completely different, and results for one system do not apply to the other. NuPIC is memory bandwidth-intensive, and not cache friendly.

When you move to a cluster, interconnect bandwidth and latency become constraints. Technically these constraints also apply on a single machine, but you run into other limits first.

For experiment mode, cluster interconnect bandwidth and latency is not a limiter, so it makes sense to move to a cluster as soon as memory bandwidth contention starts to limit performance.

For large problem mode, there hasn't been much experimentation. Networks with many fat nodes (tens of thousands of coincidences) are likely to run well on a cluster. Other information is still to be discovered.

## How to Use NuPIC in Complex Configurations

---

You can use the `SessionConfiguration` object for advanced session configuration. By default, the `Session` interface starts an NRE with one NP on the same machine on which the Python program from which you started the session is running.

Using a `SessionConfiguration` object allows you to do one or more of the following:

- [Using Multiple NPs on page 76](#)
- [Starting a RuntimeNetwork or a Session that Runs on a Cluster on page 76](#)
- [Launching on a Remote Host on page 78.](#)



Be sure to read the earlier sections of this chapter before you start multiple NPs.

### Using Multiple NPs

How to run your HTM Network using multiple NPs is discussed in some detail in [Running in Parallel: Large Problem Mode on page 71](#).



The assignment of NPs to CPUs is currently handled by the NRE. You have no control over the assignment.

### Starting a RuntimeNetwork or a Session that Runs on a Cluster

A cluster consists of several computers, each with one or more CPUs. Clusters have additional software requirements beyond what is needed for a single host.



Running on a cluster is not supported for Microsoft Windows.

For the following examples, assume you have a cluster with four hosts: SnowWhite is the master host and Green, Brown, and Gray are the other hosts in the cluster.

#### To create and start a RuntimeNetwork to run on a cluster:

1. Import the `CreateRuntimeNetwork`, `SessionConfiguration`, and `SessionServerDistribution` classes:
 

```
from nupic.network import (CreateRuntimeNetwork, SessionConfiguration,
                           SessionServerDistribution)
```
2. Build a `SessionConfiguration` object, which stores all advanced session parameters:
 

```
myConfig = SessionConfiguration()
```
3. Change the `numNodeProcessors` parameter for the session configuration:
 

```
myConfig.setNumNodeProcessors(5)
```
4. Create a `SessionServerDistribution` object:
 

```
myServerDist = SessionServerDistribution()
```
5. Add the hosts in the cluster to `myServerDist`. Don't add the head host.

```
myServerDist.addServer ("Green" )
myServerDist.addServer ("Brown" )
myServerDist.addServer ("Gray" )
```

6. Add myServerDist to the session configuration object:

```
myConfig.setServerDistribution (myServerDist)
```

7. Create a RuntimeNetwork with the custom session configuration and any files the NRE needs to access.

```
myNetwork = CreateRuntimeNetwork( <netfilename>.xml,
                                   files = [<myfiles>],
                                   config = myConfig)
```

#### To create and start a session to run on a cluster:

1. Import the Session, SessionConfiguration, and ServerDistribution classes:

```
from nupic.network import Session, SessionConfiguration,
                        SessionServerDistribution
```

2. Create a Session object:

```
mySession = Session("dir/subdir/mysession")
```

This step is identical to the single-process session creation.

3. Add files the session needs.

This step is identical to the single-process session bundle management, discussed in [Sessions and Session Bundles on page 49](#).

4. Build a SessionConfiguration object, which stores all advanced session parameters:

```
myConfig = SessionConfiguration()
```

5. Change the numNodeProcessors parameter for the session configuration:

```
myConfig.setNumNodeProcessors(5)
```

6. Create a SessionServerDistribution object:

```
myServerDist = SessionServerDistribution()
```

7. Add the hosts in the cluster to myServerDist. Don't add the master server.

```
myServerDist.addServer ("Green" )
myServerDist.addServer ("Brown" )
myServerDist.addServer ("Gray" )
```

8. Add myServerDist to the session configuration object:

```
myConfig.setServerDistribution (myServerDist)
```

9. Start the session with the custom session configuration:

```
mySession.start (myConfig)
```

Launching on a Remote Host

At times, you might want to start the NRE from one host but run it on a remote host. For example, you might want to use a remote cluster that has more processing power than your local system.



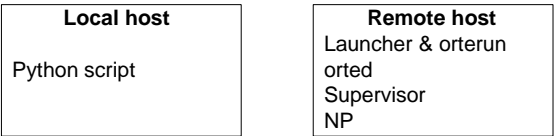
Launching on a remote host is not supported on Microsoft Windows.

To understand what happens when you launch remotely, consider the processes that run as part of NuPIC (see also [NRE Process Structure with Multiple NPs on page 67](#)).

- Python script
- launcher & orterun
- orted
- Supervisor
- NP (one or more)

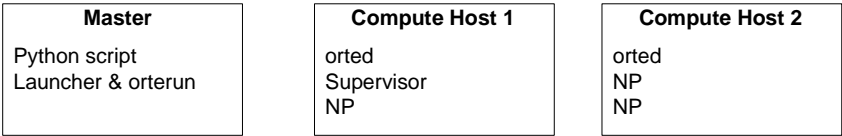
If you’re launching NuPIC remotely on a single host, only the Python script runs locally. The processes are distributed as follows:

Figure 14 Processes After Launching Remotely on a Single Host



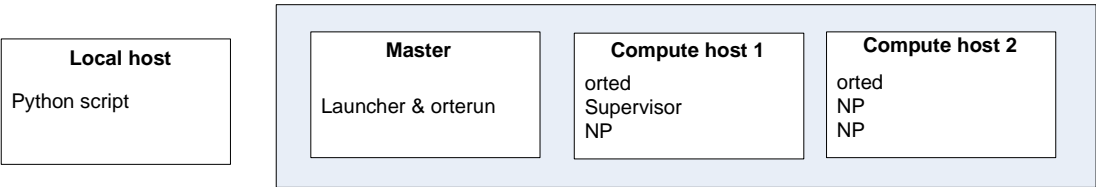
If you’re running on 2 remote hosts and use 3 NPs, the processes are allocated as follows:

Figure 15 Processes After Launching 3 NPs on a Remote Host



If you’re launching the NRE on a remote cluster, the processes are allocated as follows:

Figure 16 Processes After Launching the NRE on a Remote Cluster



This section steps you through creating and starting a session on a remote host.

**To create and start a session to run on a remote host:**

1. Import the `Session`, `SessionConfiguration`, and `ServerDistribution` classes:

```
from nupic.network import Session, SessionConfiguration,
    SessionServerDistribution
```

2. Create a `Session` object:

```
mySession = Session("dir/subdir/mysession")
```

This step is identical to the single-process session creation.

3. Add files the session needs:

This step is identical to the single-process session bundle management, discussed in [Sessions and Session Bundles on page 49](#).

4. Build a `SessionConfiguration` object and specify its parameters (see [Customizing a SessionConfiguration object to run on a remote host on page 79](#)).

5. Start the session with the custom session configuration:

```
mySession.start (myConfig)
```

**To set up a RuntimeNetwork to run on a remote host:**

1. Import the `CreateRuntimeNetwork`, `SessionConfiguration`, and `SessionServerDistribution` classes:

```
from nupic.network import (CreateRuntimeNetwork, SessionConfiguration,
    SessionServerDistribution)
```

2. Build a `SessionConfiguration` object and specify its parameters (see [Customizing a SessionConfiguration object to run on a remote host](#)).

3. Create a `RuntimeNetwork` with the custom session configuration and any files the NRE needs to access.

```
myNetwork = CreateRuntimeNetwork( <netfilename>.xml,
    files = [<myfiles>],
    config = myConfig)
```

**Customizing a SessionConfiguration object to run on a remote host**

1. Build a `SessionConfiguration` object, which stores all advanced session parameters:

```
from nupic.network import SessionConfiguration
myConfig = SessionConfiguration()
```

2. Change the parameter for the session configuration. You can change the following parameters:

---

<code>setLaunchHostname</code>	Host name for the remote host.
--------------------------------	--------------------------------

---

<code>setTunnel</code>	Turns on ssh tunneling. The number specifies a TCP/IP port on the tools host that accepts socket communications, routes it through encrypted ssh, and establishes a connection to the NRE on the launch host.  Pick a number greater than 1025. Do not run two session instances using the same tunnelport number at the same time.
<code>setUsername</code>	Needed only if your user name on the remote host differs from the user name on the local host.
<code>addToEnv</code>	Allows you to specify any environment variables you might need to set on the remote host.
<code>setPython</code>	Allows you to specify the path to the Python executable to use, for example, if the remote host uses Python 2.3 by default but you need to use Python 2.4 to work with NuPIC.

The following example illustrates this:

```
from nupic.network import SessionConfiguration
myConfig = SessionConfiguration()

launchHostname = "SnowWhite"
myConfig.setLaunchHostname(launchHostname)

tunnelport_number = 2600
myConfig.setTunnel(tunnelport_number)

username = "rblack"
myConfig.setUsername(username)

myConfig.addToEnv("PATH", "/opt/nta/current/bin")
myConfig.addToEnv("PYTHONPATH",
                  "/opt/nta/current/lib/python2.4/site-packages")

myConfig.addToEnv("NTA_LICENSECONFIG", "/home/rblack/.nta/license.cfg")
myConfig.setPython("/usr/local/bin/python")
```



## SessionConfiguration Object Methods

This section summarizes all `SessionConfiguration` methods. It includes a brief description or points to the section where the method is discussed.

Method	Description	Default
<code>setLaunchHostname</code> (string hostname)	Sets the name of the host (e.g. <code>cluster1.numenta.com</code> ) where the launcher will be run to launch the runtime engine. "Password-less ssh" is required to use a non-default launch host.	localhost
<code>setTunnel</code> (integer port)	Sets the TCP/IP port to use for tunneling through firewalls to reach the host the Supervisor will run on. The port number is used to open a tunnel that receives data on that port on the local host and forwards the data to the Supervisor host by tunneling (using ssh) through the launch host. Contact Numenta Support if there's a firewall between the local and the remote host.	None. ssh tunneling is not used.
<code>setUsername</code> (string username)	The user name that must be specified to log into the launch host via ssh. "Password-less ssh" is required to use a non-default user name.	None. Uses the current login user name.
<code>setNumNodeProcessors</code> (integer numNodeProcessors)	Sets the number of NPs to launch on the remote host(s). This number is incremented by 1 internally to account for the Supervisor process to calculate the total number of processes to launch. To launch a single Supervisor and 2 NPs, call <code>setNumNodeProcessors(2)</code> .  See <a href="#">Using Multiple NPs on page 76</a> for an example.	1  (A single Supervisor and a single NP is launched.)
<code>addToEnv</code> (string key, string value)	Specifies an additional environment variable that should be set on the launch host. Can be called repeatedly to build up a list of environment variables to set. Certain environment variables receive special treatment within this call. For example, if <code>PATH</code> is set this way, the specified path is prepended to the default <code>PATH</code> available on the launch host.	Empty list. (No additional environment variables are set.)
<code>setPython</code> (string path)	Set the full path to the Python executable on the remote host. If the remote host has a non-standard installation of Python 2.4 (not named "python" accessible from <code>PATH</code> ), the custom path can be specified here. This might be necessary if the user's environment has not been fully configured.	python
<code>setServerDistribution</code> (serverDistribution)	Specifies the set of hosts across which the Supervisor and NPs are distributed. <code>SessionServerDistribution</code> is an object that holds a list of host descriptions. This data structure is analogous to an MPI host file.  To use this method, you must first build a <code>SessionServerDistribution</code> , as follows:  <pre>serverDistribution = SessionServerDistribution() serverDistribution.addServer(string hostname)</pre> See <a href="#">Starting a RuntimeNetwork or a Session that Runs on a Cluster on page 76</a> for a complete example.	A single host (the launch host).





# A *Examples*

---

This appendix gives information about the examples available for Numenta developers. For each example, it discusses the problem it solves and explains the implementation. It also lists the script, explains what each script does, and discusses how you can experiment and view results using the example.



The Images example is discussed in a separate document.

## Topics

- [Bitworm Example, page 84](#)
- [Waves Example, page 85](#)
- [Net\\_Construction Examples, page 87](#)
- [Flu Example, page 89](#)
- [Speech Example, page 92](#)
- [Pictures Example, page 93](#)

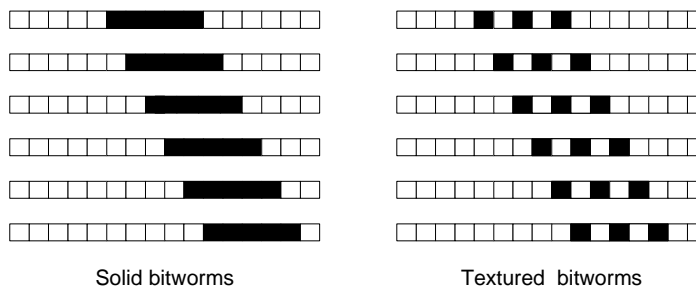
## Bitworm Example

---

The Bitworm example is a very simple program that is meant to illustrate how to build and run an HTM Network using node tools.

### Problem Definition

In the Bitworm world, there are two types of bitworm, solid and textured. Each bitworm is a 16-bit vector. Solid bitworms include a sequence of on bits, textured bitworms include a sequence of on-off-on-off etc. bits.



### Implementation

The Bitworm scripts use the most basic way of creating an HTM Network: node tools, explicit links, and sessions that are created and invoked explicitly. See Understanding the Example Scripts, page 20 in *Getting Started With NuPIC*.

The `RunOnce.py` script is the entry point; it allows you to change parameters. For example, you can add noise to the training data or the testing data.

### Exploration and Verification

By default, `RunOnce.py` trains with data that don't contain additive or bitflip noise, then tests first with the same data, then with a second set with a different seed.

You can then make changes to the `RunOnce.py` script, for example, to add noise, then rerun the example. After each run, you can examine the `results.txt` file to see the statistics and a display of the groups.

### Notes

There are two versions of Bitworm. One is a simple getting started example, the other (`bitworm_sessions`) illustrates using sessions and enabling and disabling nodes for learning and inference.

### See Also

Bitworm: Getting Started Example, page 13 in *Getting Started With NuPIC*.

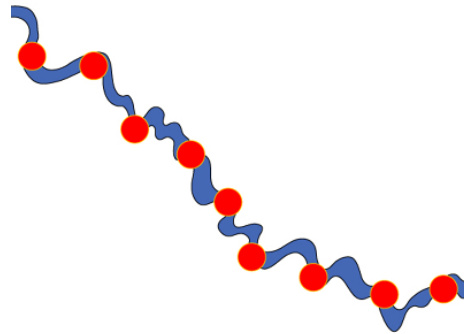
## Waves Example

The Waves example implements an HTM Network that categorizes the state of a river in which sensors have been placed.

### Problem Definition

The Waves example assumes there's a set of temperature monitors in a river. Hot and cold spots migrate down the river. If there's a hot spot upstream, it appears downstream after a short time.

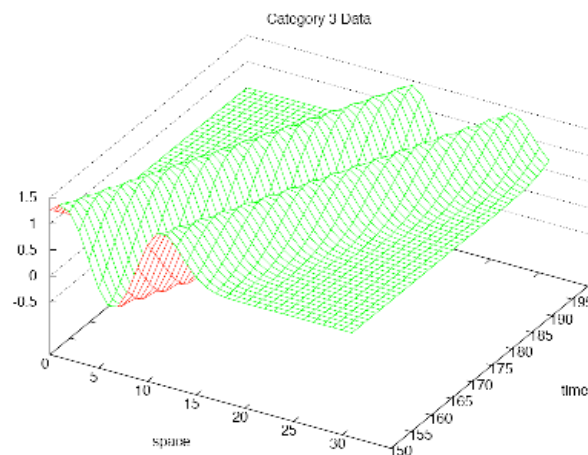
Figure 17 Waves Example



A given pattern of temperatures is called a state. Put slightly differently, the patterns of temperatures observed at the sensors are manifestations of an underlying "cause", unspecified in this example, but that called the state of the river.

The goal of the program is to identify the river as being in one of four states. Based on the observed temperature patterns, we want the HTM network to infer which state the river is in. Each state is represented by a Gaussian curve that shows, for example, one hot spot (category 0) two hot spots (category 1), one hot and one cold spot (category 2), or two hot spots with a cold spot between (category 3):

Figure 18 Waves State Example: Category 3 Data



## Implementation

The waves example is slightly more complex than Bitworms. There are several versions of the program:

- The top-level example uses the helper functions.
- `runtimeNetwork` uses the `RuntimeNetwork` interface.
- `simplehtmWaves` uses the `SimpleHTM` interface. This interface is no longer recommended.

### Experimenting with Waves

In each example, you can add two types of noise.

- Spatial noise — The first is spatial noise. In each class, the center of each Gaussian is given a random horizontal offset. This offset is chosen based on the `spatialNoise` parameter; the offset is chosen uniformly in the range  $[-\text{spatialNoise}/2, \text{spatialNoise}/2]$ .
- Thermal noise — Each data point has a small amount of noise added. The amount is controlled by the `thermalNoise` parameter, and is chosen uniformly in the range  $[0, \text{thermalNoise}]$ .

Noise defaults to 0. You can modify it by changing the following parameters in the `RunOnce.py` script.

```
# Training Data Parameters:
trainThermalNoise = 0.0
trainSpatialNoise = 0.0

# Testing Data Parameters
testThermalNoise  = 0.0
testSpatialNoise  = 0.0
```

## See Also

Understanding HTM Development: Waves Example, page 25 in *Getting Started With NuPIC* uses the Waves example to illustrate HTM Development tasks.

## Net\_Construction Examples

The scripts in the `net_construction` folder illustrate how to create different network topologies using the NuPIC network construction APIs. Each script creates a single network structure.



NO attempt has been made to validate that these networks actually learn anything useful! The scripts are merely meant to illustrate the mechanics of using the network construction APIs. For useful networks, please see the other examples, such as `bitworm`, `waves`, `wallstreet`, `flu`, and `pictures`.

Most of the scripts use the region and link policy concepts.

- A region is a set of nodes that share the same node type and settings. For example, each level in an HTM network is typically a single region. A region can have an implicit multi-dimensional structure. As an example, for image recognition, it helps to have the early levels in your network arranged in a two-dimensional fashion. See [Regions, page 30](#).
- A link policy specifies how two regions can connect. A link policy makes it easy to connect two regions, and automatically takes into account fan-in, overlapping connections, and multi-dimensional structure. See [Link Types, page 28](#).

Regions and link policies make it easy to create common network structures with a few API calls.

### Example Scripts

The following scripts are included:

Table 5: Scripts in the `net_construction` example set

Script	Description
<code>LogoNetwork.py</code>	Creates a simple 7 node network by creating individual nodes and links.
<code>LogoNetworkRegions.py</code>	As above but uses regions and link policies.
<code>OneDimensional.py</code>	Creates a one-dimensional network, that takes 4096 inputs and consisting of 137 nodes.
<code>PicturesNetwork.py</code>	Creates a two dimensional network similar to Pictures.
<code>PicturesNetworkPassThrough.py</code>	Creates a two dimensional network similar to Pictures but adds <code>PassThroughNodes</code> so the network can be pipelined.
<code>ImageNetwork.py</code>	Create a large network to process one megapixel images.
<code>ImageNetworkOverlapping.py</code>	s above, but with overlapping receptive fields at each level.

Table 5: Scripts in the `net_construction` example set (Cont'd)

Script	Description
<code>MultiNetwork.py</code>	Creates a large HTM network that combines two sensory modalities. The first modality (say audio) is processed using a one-dimensional network. The second modality (say vision) is processed using a two dimensional network. The two modalities are combined at a higher level.
<code>MergedNetwork.py</code>	Creates a new merged network from two existing network files. Specifically, this example takes the <code>Pictures</code> and <code>OneDimensional</code> networks created above, loads them, and creates a new network that is the equivalent of the <code>MultiNetwork</code> . If the existing networks had undergone training, the trained states would be preserved in this new network.



## Flu Example

The flu example demonstrates how to explore a trained HTM network using `RuntimeNetwork` and `Zeta1` analysis functionality.

Detailed API documentation on the classes used can be found using `NodeHelp`.

### Problem Definition

The HTM Network is meant to predict when there will be a flu epidemic. The data are very noisy, but the HTM Network performs some preprocessing that allows it to determine the peak months for flu.

The data used in this example is weekly influenza mortality data from nine reporting regions around the US. The numbers in the data files represent weekly log influenza mortality risk.

For more information about the reporting system and available data, see: <http://wonder.cdc.gov/mmwr/mmwmort.asp>

### Implementation

The flu example uses `SimpleHTM`, an older API that allows you to quickly create nodes by specifying parameters.

The example includes the following files.

Table 6: Flu example files

File	Description
<code>learn.py</code>	Creates and trains an HTM network from the data.
<code>explore.py</code>	Explores the contents of the trained HTM network.
<code>utils.py</code>	A collection of utility functions supporting <code>explore.py</code> .
<code>runinference.py</code>	Runs trained HTM inference on test data.
<code>data/data.txt</code>	The original data used to test the inference of the HTM.
<code>data/train.txt</code>	Smoothed training data generated from <code>data.txt</code> .
<code>visuals/cities.pdf</code>	Visualization of the spatial relationship between the data.
<code>visuals/risk.pdf</code>	Visualization of the test data.
<code>visuals/smooth.pdf</code>	Visualization of the smoothed training data.
<code>cleanup.py</code>	Cleans up files created when running.
<code>README.txt</code>	Description of the example and how to use it.

## Learning from the Example

The example focus is on exploring a trained HTM network. This section explains the output of `explore.py`.

1. First, the script loads the trained HTM network file in `trained.xml`, and enumerates the elements of the network. The network has a sensor, a bottom level with 3 nodes, a top level with a single node, and an output node.

```
Loading a network...
```

```
Elements in the network:
```

```
-----
Node sensor ()
Region level1 (3,)
Region level2 (1,)
Node output ()
```

2. Next, the script chooses a single bottom-level node and prints its learned data structures. You can find similar information in the Visualizer output at `trained/level1[0]/index.html`.

```
Exploring node level1[0]:
```

```
-----
Node 0 coincidences:
[[ 0.102307    0.28016099 -0.0273473 ]
 [-0.40698501 -0.0392707  -0.23639201]]
Node 0 counts:
[261, 195]
Node 0 TAM:
[[ 241.    8.]
 [   9. 180.]]
Node 0 groups:
[set([0]), set([1])]
```

3. The script then analyzes the learned data structures of the top node.

```
Exploring the top node:
```

```
-----
Top coincidences: 6
[[1, 1, 2], [1, 1, 1], [1, 2, 1], [2, 2, 1], [2, 1, 1], [2, 1, 2]]
Top groups: 3
[set([1, 2, 3, 4]), set([0]), set([5])]
Top TAM (sorted): [4, 1, 1]
[[ 822.    77.    54.    72.   148.    7.]
 [  41.    66.    80.     0.     3.     0.]
 [  60.    38. 1162.   104.     2.     4.]
 [ 110.     0.    70.   424.    11.     5.]
 [ 137.     9.     0.    14.  1180.     0.]
 [   0.     0.     0.     0.    16.     4.]]
```

4. The groups and coincidences of the top node are difficult to interpret, as they refer to distributions over the outputs of the bottom nodes. To help interpret the groups in the top node, the script then uses utility functions in `utils.py` to print all sensor-level inputs that correspond to each of the three top node groups.

```
Playing down top groups:
```

```
Group: 0
```

```
Sample: 0
```

```
Child: 0 [ 0.102307    0.28016099 -0.0273473 ]
```

```
Child: 1 [ 0.0361848  0.174444  0.0524514]
```

```
Child: 2 [-0.21359199 -0.140718  -0.0217913 ]
```

```
Sample: 1
```

```

Child: 0 [ 0.102307    0.28016099 -0.0273473 ]
Child: 1 [-0.27483699 -0.28606001 -0.23867799]
Child: 2 [-0.21359199 -0.140718   -0.0217913 ]
Sample: 2
Child: 0 [-0.40698501 -0.0392707  -0.23639201]
Child: 1 [-0.27483699 -0.28606001 -0.23867799]
Child: 2 [-0.21359199 -0.140718   -0.0217913 ]
Sample: 3
Child: 0 [-0.40698501 -0.0392707  -0.23639201]
Child: 1 [ 0.0361848   0.174444   0.0524514]
Child: 2 [-0.21359199 -0.140718   -0.0217913 ]
Group: 1
Child: 0 [ 0.102307    0.28016099 -0.0273473 ]
Child: 1 [ 0.0361848   0.174444   0.0524514]
Child: 2 [ 0.108758    0.28517699  0.302306 ]
Group: 2
Child: 0 [-0.40698501 -0.0392707  -0.23639201]
Child: 1 [ 0.0361848   0.174444   0.0524514]
Child: 2 [ 0.108758    0.28517699  0.302306 ]

```

5. Finally, the script loads the test data file and watches the state of the top node change as sensor inputs are processed in inference mode.

Watching an internal parameter:

-----

```

Iteration: 1 level2[0] spatialPoolerOutput
1.0 0.0 0.0 0.0 0.0 0.0
Iteration: 1 level2[0] bottomUpOut
0.0 0.0 1.0 0.0 0.0

```

```

Iteration: 2 level2[0] spatialPoolerOutput
1.0 0.0 0.0 0.0 0.0 0.0
Iteration: 2 level2[0] bottomUpOut
0.0 0.0 1.0 0.0 0.0

```

```

Iteration: 3 level2[0] spatialPoolerOutput
0.4 0.0 0.0 0.0 0.0 1.0
Iteration: 3 level2[0] bottomUpOut
0.0 0.0 0.4 1.0 0.0

```

```

Iteration: 4 level2[0] spatialPoolerOutput
0.7 1.0 0.0 0.0 0.0 0.0
Iteration: 4 level2[0] bottomUpOut
0.0 0.4 0.7 0.0 0.0

```

```

Iteration: 5 level2[0] spatialPoolerOutput
0.4 1.0 0.0 0.0 0.2 0.1
Iteration: 5 level2[0] bottomUpOut
0.0 0.4 0.4 0.1 0.0

```

```

Iteration: 6 level2[0] spatialPoolerOutput
0.0 0.0 0.0 1.0 0.0 0.0
Iteration: 6 level2[0] bottomUpOut
0.0 0.4 0.0 0.0 0.0

```

```

Iteration: 7 level2[0] spatialPoolerOutput
0.0 0.0 1.0 0.2 0.0 0.0
Iteration: 7 level2[0] bottomUpOut
0.0 0.2 0.0 0.0 0.0

```

## Speech Example

---

The speech example includes a set of audio files and experiments for classifying human speech using HTM. The example trains HTMs to classify data by gender and by speaker. A PDF included with the example explains the data collection, file structure, and training process and also suggests how you can experiment with the data.

### Problem Definition

The speech HTM experiments perform two tasks:

- For a new audio file with speech data, determine whether the speaker is female or male.
- For a new audio file with speech data, determine the speaker from the set of speakers already known to the HTM Network.

### Speech Data

Speech data are based on recordings of text read by four individuals. Each reading was divided into a 20-second training sequence and six 5-second test sequences. Each experiment processes the audio into 100 "frames" per second. The HTM is trained and tested using these import vectors.

### HTM Network Structure

The examples use simple HTM Network structures. One example uses the simplest possible structure for training: a sensor, spatial pooler region and temporal pooler region. After the HTM Network has been trained, a classifier node is added. The classifier builds a mapping between the HTM's training data output and a classification file. During subsequent runs, the classifier evaluates new data submitted through the trained HTM and computes an appropriate classification.

### Running the HTM Network

The PDF included with the example discusses each of the included experiments, the audio preprocessing, and how to evaluate speech HTMs.

## Pictures Example

This section introduces the Pictures example. For more detailed information on some of the topics, see the `README` file included with the program. For a discussion on what happens at runtime, see *Training Your HTM Network: The Pictures Example*, page 64 in *Getting Started With NuPIC*.



This example is being replaced by the images example. The discussion is left in this document because the example includes some interesting implementation specifics.

### Problem Definition

The Pictures HTM Network recognizes pictures. The current dataset consists of 453 hand-drawn black and white pictures divided into 48 categories such as cat, dog, or rake. Each picture is 32x32 pixels. There are 5-20 examples of each category. Here are some examples of a rake (which actually looks more like a pitchfork). You can see how the rake has different sizes and different distances between prongs.

Figure 19 Example of Rake in Pictures Example Set



After the HTM Network has been trained, it can recognize the pictures in the training set but can also recognize new pictures in the same category. For example, it might recognize a pitchfork with a long handle and short prongs. The Pictures retrain functionality makes it possible to add new categories to a previously trained HTM Network quickly.

### Implementation

You can run the example either embedded in the NetExplorer framework (`RunExperiments.py`) or standalone (`RunOnce.py`). Both scripts call the other scripts that are part of the example as needed. A number of other scripts are also included, as follows:

Script	Description	Output
<code>RunOnce.py</code>	Simple command-line script that illustrates most of the basic aspects of the Pictures example. The script runs through several steps, including creating, training, and testing a network.  This script is a basic introduction, like a HelloWorld program.	Several files. The most important ones are: <code>pic.trained.xml</code> <code>pic.retrained.xml</code> <code>pic.report.txt</code>  <code>pic.test&lt;n&gt;.results</code> files are also available after the run.

Script	Description	Output
<code>RunExperiments.py</code>	<p>Framework for systematically running experiments to test the effects of varying different network, training, and/or testing parameters.</p> <p>This script is a large-scale framework for running the example. It allows you to run various prepared experiments and can be extended to run custom experiments.</p>	<p><code>results.pkl</code> <code>plot.ps</code></p> <p>These are created inside the <code>experiments/&lt;experiment_name&gt;/</code> directories.</p>
<code>PicturesDemo.py</code>	A set of scripts that support a graphical user interface (GUI). The GUI allows you to experiment with the application and understand all basic concepts.	No output.
<code>BuildPicturesNet.py</code>	<p>Command-line tool for quickly building and testing a Pictures network.</p> <p>This is a utility tool for building new networks from different training sets.</p>	Depends on the command-line arguments.
<code>PicturesVisualizer.py</code>	A command-line tool for invoking the Visualizer tool on a Pictures network.	Depends on the command-line arguments.

### Node Hierarchy

The Pictures HTM Network uses the following hierarchy of nodes:

- At the bottom level, each node is divided using an 8x8 grid and sees a 4x4 pixel viewing window of the picture at any time. That means there are 64 viewing windows at that level. During training, the node looks through sequential viewing windows, first left to right, then top to bottom.
- The next level uses by default a 2x2 grid; each viewing window is 16x16 pixels. The grid can be changed, in fact, the Topology experiment tests a 4x4 middle level.
- The classifier node gets input from all the viewing windows at the previous level.

Pictures has been designed to test different topologies. You could, for example, try a different grid for level 2 or add another level of learning level.

### Training Process

During training, each picture is submitted to the HTM Network as if it were moving: The simulated movement goes both horizontally and vertically. The process is repeated for each picture in the category. This provides the temporal aspect. The HTM Network looks through a window and sees a cat or dog go by. For each time instance, something different is visible.

Because the program fully scans each picture horizontally and vertically, one node has all the information a node can have. The training process takes advantage of that: It completely trains one node, then copies the state of that node to all the other nodes at the same level.



Training one node, then copying its state dramatically speeds up training time.

### Data Files

The Pictures example ships with two sets of picture data.

- The first data set, called `clean`, consists of non-noisy drawings of 48 different object categories, such as `cat`, `dog`, and `helicopter`. There are a total of 453 pictures in this data set. The pictures are in 1-bit/pixel `.bmp` format, and are organized into 48 different sub-directories, with each sub-directory bearing the name of a particular object category. The `clean` data set is intended for training.
- The second data set, called `distorted`, is organized in the same hierarchical directory structure as `clean`, and consists of pictures in the same 1-bit/pixel `.bmp` format. However, the `distorted` data set contains pictures that have been warped, corrupted with noise, and otherwise distorted relative to the `clean` set. The pictures in this set provide a large corpus of testing data for evaluating the generalization capability of an HTM Network.

Both the `clean` and the `distorted` sets are in the `data.d` directory. As shipped, the data sets are in the form of compressed tar files. They must be decompressed and untarred before any training or testing of HTMs can be performed.

To decompress and untar the data files, issue the following commands:

```
python Unpack Pictures.py
```

The `RunOnce.py` script automatically decompresses and untars these data sets the first time it is run. Most of the other scripts perform decompression as well.

## Exploration and Verification

The Pictures `RunOnce.py` script demonstrates most of the Pictures functionality. It performs a total of twelve steps, including the creation, training, and testing of a new network, followed by a retraining stage. During retraining, additional categories are presented and the classifier node is trained on these new categories.

The `RunOnce.py` script also illustrates how to perform runtime inference, as opposed to batch testing.

Batch testing is for systematically presenting a large set of test data to the HTM to measure its performance. Runtime inference presents a single pattern at a time to the HTM in the context of a real, deployed application.

The `RunOnce.py` script generates a report that summarizes the results of the twelve steps. The report exemplifies the mechanism by which application code can gain access to internal node details.

**To run the RunOnce.py script:**

```
cd <NuPIC location>/share/projects/pictures
python RunOnce.py
```

RunOnce.py takes four optional switches:

<code>--first=N, --last=M</code>	Force the script to start on the Nth step (where N=1 by default) and finish on the Mth step (where M=12 by default). These switches are useful when you wish to perform individual steps without re-running the entire script.
<code>--short</code>	Run a shorter version of the sequence. Allows you to see results sooner.
<code>--deterministic</code>	Always seed the pseudo-random number generator to the same value. Allows you to compare results with results of previous runs to see how things changed.

During a run, RunOnce.py prints output to the console that describes the training and testing process.

**Experimenting Using the Pictures Demo GUI**

The Pictures example includes an interactive, GUI-based demonstration program called Pictures. Pictures Demo is built from a collection of classes called Phoenix.

Pictures allows you to sketch a hand-drawn picture using the mouse, and immediately present that drawing to a pre-trained HTM. The HTM runs inference on the drawing and display the best matching object categories.

Pictures Demo consists of a core application class, called PhoenixCore, and a GUI layer that implements a particular skin. The Pictures demo ships with two skins:

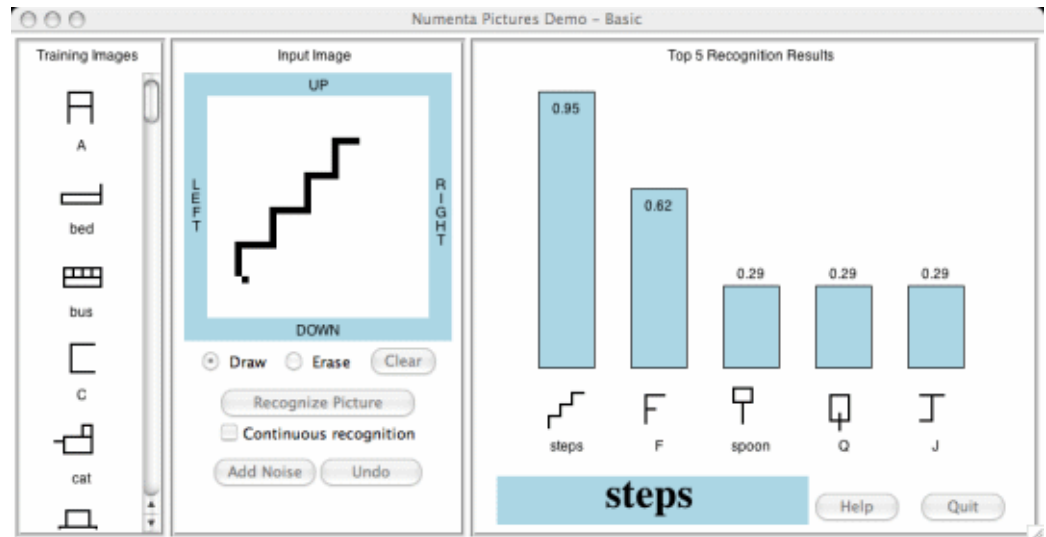
- Basic skin, implemented by the PhoenixGui class. The Basic skin allows you to draw sketches, shift them horizontally and vertically, and add noise to the pictures by clicking a button.
- Advanced skin, implemented by the PhoenixGuiAdvanced class derived from PhoenixGui. The Advanced skin adds the ability to retrain the network with new custom object categories, and to perform picture transformations, such as rotations, scaling effects, and spatial distortions.

**Phoenix Basic**

To launch Pictures Demo with the Basic skin, type: `python PicturesDemo.py`



The following GUI appears:

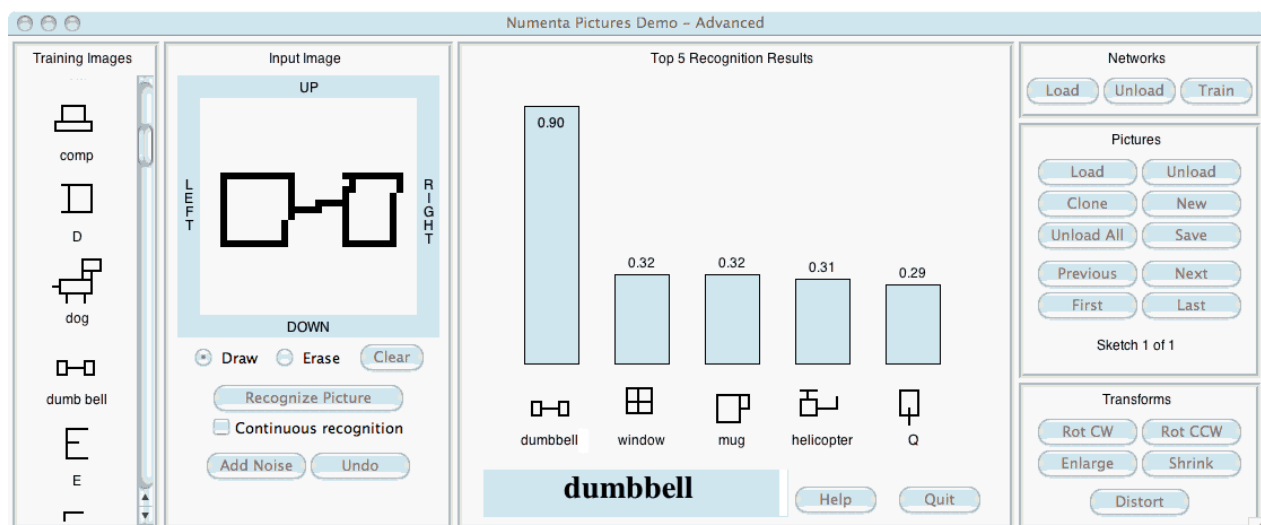


You can perform the following tasks:

- Select a picture.
- Modify a picture using your cursor. By default, **Draw** is selected and you can click or drag to add black pixels. Select **Erase** to change black pixels to white.
- Add noise to an picture by clicking the **Add Noise** button, remove noise you've added by clicking **Undo**.
- Click the **Continuous recognition** check box to see how changes you make to an picture affect recognition accuracy.

### Pictures Demo Advanced

To launch the Pictures Demo with the Advanced skin, type: `python PicturesDemo.py --skin=advanced`. The following GUI appears:



The Advanced skin offers the same functionality as the basic skin. In addition, you can train new categories that were not part of the original training set.

You can use the following buttons to interact with the HTM Network:

### Networks panel

Button	Description
Load	Loads a complete HTM Network.
Unload	Unloads the current HTM Network.
Train	Retrains the current HTM Network.

### Pictures panel

Button	Description
Load	Prompts you for a picture to load.
Unload	Unloads the currently selected picture.
Clone	Clones the currently selected picture. You can create a new picture by modifying the clone.
New	Creates a new blank picture.
Unload All	Unloads all currently loaded pictures.
Save	Saves the currently displayed picture to a disk file.
Previous/Next	Displays the previous or next picture in the set of pictures currently loaded.
First/Last	Displays the first or last picture in the set of pictures currently loaded.

### Transforms panel

Button	Description
Rot CW	Rotates the current picture clockwise.
Rot CCW	Rotates the current picture counter-clockwise.
Enlarge	Enlarges the picture.
Shrink	Shrinks the picture
Distort	Distorts the picture by moving its pixels. The result is a picture that more closely resembles the original than if you'd added noise.



# B

## ***Numenta NetExplorer***

---

Numenta NetExplorer is an older API for testing HTM Networks and data with different settings. This appendix discusses using NetExplorer.

### **Topics**

- [Using Numenta NetExplorer on page 100](#)
- [Running NetExplorer Tests in Parallel on page 108](#)

## Using Numenta NetExplorer

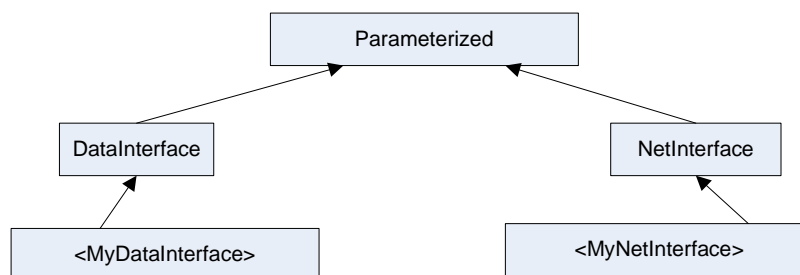
Numenta NetExplorer makes it easy to test HTM Networks and data with different settings. NetExplorer is a framework: You instantiate the framework and embed your HTM Network. You can then run the network with those different settings.

If you decide to use NetExplorer, you spend a little bit more time fitting your network and data classes into the framework. Once that is done, using the network and testing results over any ranges of settings is easy.

### NetExplorer Basics

To fit your HTM Network into the NetExplorer framework, you must subclass `NetInterface` to represent your HTM Network and `DataInterface` to represent your data. You also need to represent the values that parameterize your HTM Network as `Parameter` objects.

NetExplorer includes the following main classes (additional classes are discussed throughout this section):



- The `DataInterface` class exports a `createData()` method for generating data. Your subclass should override this stub method so that NetExplorer correctly interfaces with your class.



Note that the NetExplorer `Data.visualizeData()` method is completely different from the `Visualizer.visualizeData()` method.

- The `NetInterface` class exports the `createNetwork()`, `train(data, nprocs, hosts)`, and `test(data, nprocs, hosts)` methods for training and evaluating the network. Your subclass should override this stub method so that NetExplorer correctly interfaces with your class.

Both `DataInterface` and `NetInterface` are subclasses of `Parameterized`, a class that has a collection of `Parameters`, which are easy to inspect and modify.

Because the NetExplorer classes define a standardized interface, you can design tests that run on a variety of different HTM Networks. For example, suppose you start with an HTM Network, `Example2Network`, and a data set, `Example2Data`. Then you can train and test the network like this:

```
# Prepare the data and network
```

```

trainData = Example2Data()
testData = Example2Data()
network = Example2Network()

# Give them all prefixes so they don't write over each other's files
trainData['prefix'] = 'train_'
testData['prefix'] = 'test_'
network['prefix'] = 'network_'

# Create some training data
trainData.createData()

# Train the network
network.createNetwork()
network.train(trainData)

# Test the network
testData.createData()
result = network.test(testData)

```

## TestCrossParameters Class

The `TestCrossParameters` class offers a simpler way of running tests. `TestCrossParameters` lets you describe a set of tests and then runs them for you. `TestCrossParameters` tests an HTM Network by computing performance measurements such as classification accuracy as one or more network or data set parameters change.

`TestCrossParameters` takes advantage of the `NetInterface` and `DataInterface` APIs. It allows you to run tests on the parameters of your network and data classes. If you instantiate `TestCrossParameters` with your data and network objects as parameters, `NetExplorer` tests all combinations of these parameters in an efficient way. Optionally, if you have multiple CPUs or computers available, `NetExplorer` can also run these tests in parallel. Instead of having to write new procedural code for each test you want to run, you can simply describe the test.

---

```

# Prepare the data and network
trainData = Example2Data()
testData = Example2Data()
network = Example2Network()

# Give them all prefixes so they don't write over each other's files
trainData['prefix'] = 'train_'
testData['prefix'] = 'test_'
network['prefix'] = 'network_'

# Describe the parameters we want to test
testParams = []
# We want to test the spatial noise in the test data
testParams.append( Parameter('spatialNoise', owner='testData', minval=0.0,
maxval=0.3, numSteps=10) )
# We want to test the thermal noise in the training data
testParams.append( Parameter('thermalNoise', owner='trainData', minval=0.0,
maxval=1.0, numSteps=10) )

# Optionally, use a selection method to avoid testing certain parameter
# combinations.
def mySelectionMethod(trainData, network, testData):

```

```

"""For no good reason (just an example), tell the TestCrossParameters not
to test if the thermal noise and the spatial noise are the same."""

if trainData['thermalNoise']==testData['spatialNoise']:
    # don't test this case
    return False
# do test this case
return True

# Run the experiment
test = TestCrossParameters(trainData, testData, network, params,
                           constraintFunction=mySelectionMethod) filename='results.pkl')
test.runTest()
test.plotResults()

# We can easily load the results and examine them more closely
test2 = Test()
test2.loadResults('results.pkl')
test2.plotResults()
firstResult = test2.results[0]
print firstResult.results['accuracy']
testNetwork = firstResult.configuration['network']

```

You can see from this example that you only need to provide some setup code and the parameters you want to test. You don't need to write any iteration or plotting code yourself. Instead, you write declarative code that describes the parameter values to test. In addition, `TestCrossParameters` automatically takes care of optimizing the iteration so that it runs multiple test sets without needlessly retraining the HTM Network.



See the Waves sample program for examples of `TestCrossParameter` use.

## TestCrossParameters Options

The NetExplorer framework supports constraints that you can add to the testing regime. A constraint acts as a filter that discards certain combinations of parameters from the set of tests that `TestCrossParameters` runs.

When you use `TestCrossParameters`, you can vary parameters in two ways:

- Specify minimum and maximum values and the number of steps, as in the example above.
- Explicitly specify a set of sample points over which to iterate.

## Classes Overview

The following classes are included with NetExplorer:

Class	Description
<code>TestCrossParameters</code>	Tests every value in the cross product of parameters.
<code>DataInterface</code>	Base class that defines methods useful for a dataset.

Class	Description
<code>NetInterface</code>	Base class that defines methods useful for an HTM Network, such as training and testing.
<code>Parameter</code>	Encapsulates parameters with a value, type, range, owner, and other useful parameters.
<code>Parameterized</code>	A parameterized object, the base class for both <code>DataInterface</code> and <code>NetInterface</code> .
<code>TestResult</code>	A data structure encapsulating the results of a single test. <code>TestResult</code> has a configuration (the <code>Parameterized</code> objects used to create it) and a set of results.
<code>Test</code>	A generalized test. If you want to run a test more general than <code>TestCrossParameters</code> , you can subclass <code>Test</code> . <code>Test</code> has some useful helper methods for working with intermediate and final results.

## Using Your Own DataInterface

If you want to analyze your own data set:

1. Define a data class that inherits from `netexplorer.DataInterface`
2. Override the following methods:
  - `createData()` — Method that creates the data, for example, reads the source data files or calls scripts that generate the data.
  - `visualizeData()` — Method that plots the data. You can use `Data.easyPlot2D` or `Data.easyPlot3D` inside this method.
  - `getData()` — It might be helpful to write a `getData()` method as well. Having a `getData()` method makes it easy to plot data along with other information, such as test results.



The `DataInterface` implementations of these methods are simply empty placeholders.

3. Optionally override `cleanup()`, a method that allows your derived class to clean up any data files upon completion of the test.

If you already have data, you just wrap it up in a class that derives from `DataInterface`. You might want to add a parameter in the new class describing where on disk to find the data, so you can easily switch data sets. For example, the following code fragment deals with data on disk:

```
class MyData(netexplorer.DataInterface):
    def __init__(self):
        netexplorer.DataInterface.__init__(self)
        self.addParam('dataLocation', default='standard data location')
    def createData(self):
        # nothing to do here; we already have our data
    def getData(self):
        # return our data in a form conducive to plotting

class MyNetwork(netexplorer.NetInterface):
    def train(self, data, ...):
        # read and use the data we already have, from data['dataLocation']
    def test(self, data, ...):
```

```
# read and use the data we already have, from data['dataLocation']
```

---

## Using Your Own NetInterface

To use an HTM Network of your own design, define a `NetInterface` class that inherits from `netexplorer.NetInterface` and overrides the following methods:

- `createNetwork()` — Creates the network.
- `train(data, np, hosts)` — Trains the created network on the specified data set. If you want to run tests in parallel (see below for advanced use of `TestCrossParameters`), use `np` (the number of node processors) and `hosts` (the computer hostnames to run on). If you are not running tests in parallel, you can ignore these parameters.
- `test(data, uniqueID, np, hosts)` — Tests the trained network on the specified data set. Just like in `train()`, you should use the `np` and `hosts` parameters when creating a session so that NetExplorer's parallel tools can function correctly on this HTM Network. If you are planning to run parallel tests, incorporate the value of `uniqueID` into the names of any files this method creates. That way, NetExplorer will not overwrite files when several tests are run in parallel.

This method should return a Python dictionary of the test results. Normally, including accuracy in this dictionary is recommended. You can also include other measurements of interest, such as `trainingTime`, `networkSize`, and so on. Because `TestCrossParameters` saves only the results listed in its `resultsToSave` parameter, you can later choose which results to save when running `TestCrossParameters`, so returning unnecessary results here does not use up disk space. Instead, results are saved only if you find them useful to save.



Because NetExplorer relies on Python's `pickle` library to transfer and store objects, the `createNetwork()`, `train()`, and `test()` methods must return only pickleable objects. If these methods return any objects that cannot be pickled, using `TestCrossParameters` signals an exception and fail

The saved results file (`results.pkl`) stores the accuracy for each test, as well as the parameter values in that test. By default, the network measurements computed are just classification accuracy, but the framework supports any type of measurement, such as training time.

When it has run a test and gotten a result, `TestCrossParameters` saves not only the result, but also a copy of the `netInterface` that was involved in the test. (That way, you can easily recreate the test.) However, because the `netInterface` may well have lots of stored data, which you likely do not want to save, `TestCrossParameters` calls `netInterface.savable()`, which is inherited from class `Parameterized`, to get a copy of the object with everything but the parameters stripped out. Then `netInterface.savable()` calls your class's constructor with no arguments to get a clean copy of the object.

You must either override the `netInterface.savable()` method or override the constructor to succeed with no arguments. Otherwise, `TestCrossParameters` fails, when it tries to save the results.



## Parameterized Tools

You can use the Parameterized tools when you create your data and network. For any configuration parameter, add a `Parameter` object by calling `addParam()`. You can then access (and set) the parameter value by calling `self['parameter_name']`. Declaring your `Parameter` instances in this way allows you to run tests over their values by using a `TestCrossParameters` test.

You can use the Parameterized tools when you create your data and network. For any configuration parameter, add a `Parameter` object by calling `addParam`. Its value can be accessed via `self['parameter_name']`. You use `Parameters` in the same way in your `DataInterface` class.

---

```
from nupic.analysis.netexplorer import NetInterface, DataInterface

class MyNetwork(NetInterface):
    def __init__(self):
        NetInterface.__init__(self)
        self.addParam('maxDistance', default=0.05)
        self.addParam('nodesPerLevel', default=[16, 4, 1])

    def createNetwork(self, data):
        # Our configuration depends on these parameters, and
        # might also depend on the data's parameters
        self.buildNetwork_(self['maxDistance'], self['nodesPerLevel'],
                           data['vectorLength'])

    def train(self, data, np=1, hosts=[]):
        # Use a nupic session to train the network
        doTrain_(data, np, hosts)

    def test(self, data, np=1, hosts=[]):
        # Use a nupic session to test the network
        (acc, beliefVector) = doTest_(data, np, hosts)
        return {'accuracy': acc, 'beliefs': beliefVector}

class MyData(DataInterface):
    def __init__(self):
        self.addParam('vectorLength', default=32)
        self.addParam('noiseLevel', default=0.25)

    def createData(self):
        self.genDataFromParams_(self['vectorLength'],
                                self['noiseLevel'])
        self.writeDataToFile_()

    def visualizeData(self):
        self.genDataFromParams_(self['vectorLength'],
                                self['noiseLevel'])
        self.easyPlot3D(self.getData(), title='My Data')
```

---

## Advanced Exploration

Numenta NetExplorer also has some more advanced features. Here are some tips:

- Long tests might be interrupted, for example by a power outage.  
`TestCrossParameters` automatically performs checkpointing so that intermediate

results will not be lost. If you run a test, a checkpoint file with the extension `.checkpoint` will be created. If for some reason your test is interrupted, you can restore that checkpoint file by setting the `restoreCheckpoint` parameter to `true`, as follows:

---

```
test = TestCrossParameters( trainData, testData, network, testParameters,
                           filename='results.pkl', restoreCheckpoint=True )
```

---

- If you are working with a cluster, or even just a dual-core CPU, you might benefit from NetExplorer's built-in parallelization features. You can use the `hosts` parameter of `TestCrossParameters` for this purpose. See [Running NetExplorer Tests in Parallel on page 108](#)
- `NetInterface.test()` can produce lots of output, which might clog your disk or network if you saved it all. Therefore, `TestCrossParameters` saves only selected results, so that `NetInterface.test()` can be written generally to output any results that might be remotely interesting, but you can still get only the results you need for any particular test. By default, `TestCrossParameters` saves only the accuracy. However, you can change the saved results by specifying an array of results to save. For example:

---

```
test = TestCrossParameters(..., resultsToSave=['accuracy',
        'other_result1', 'other_result2'])
```

---

When you pass in the string `'all'`, instead of an array of strings, `TestCrossParameters` saves all parameters.

- Using `Test.advancedPlot`, you can plot any function of your test results, not just the accuracy versus the parameter values. You can write a function that takes as input a `TestResult` object and returns a list of data points, as shown below. The resulting data points are then plotted as usual.

---

```
def errorDataFunction(result):
    accuracy = result.results['accuracy']
    noise1 = result.configuration['testData']['spatialNoise']
    noise2 = result.configuration['testData']['thermalNoise']
    return [1 - accuracy, # plot error, not accuracy
            noise1+noise2] # versus a measure of noise
test.advancedPlot(myDataFunction)
```

---

- You might want to configure an HTM network from the command line, either by hand or in a script. The built-in `getopt` module is the standard Python tool for this, and the `Parameterized` class (the parent of both `DataInterface` and `NetInterface`) has methods for easily working with `getopt`. In particular, `getoptOptions` and `loadGetoptOptions` can be useful. They allow the `Parameterized` object to have its parameters set via `Getopt` options. For example:

---

```
try:
    # tell Getopt to use the data's parameter names as options
```

```

options,args = getopt.getopt(sys.argv[1:],'', data.getoptOptions())
except getopt.GetoptError, (msg,opt):
    print 'Error:',msg
    sys.exit(2)
    # Use Parameterized.loadGetoptOptions to get parameters from getopt
    #data structure
data.loadGetoptOptions(options)

```

---

- Suppose that some of your parameters need to change together. For instance, suppose that you want to test a parameter called `param1` that always needs to be the same in both the training and test data. The way to do this with `TestCrossParameters` is to use the `constraintFunction` parameter. You pass in a function that takes training data, a network, and test data, and returns 'True' only for those combinations that you want to test. In the example with 'param1' given above, you could do it this way:

---

```

testParameters = [Parameter('param1', owner='trainData', samplePoints=[1,2,3]),
                  Parameter('param1', owner='testData', samplePoints=[1,2,3])]
def myConstraintFunction(trainData, network, testData):
    return trainData['param1']==testData['param1']
test = TestCrossParameters( trainData, testData, network, testParameters,
                           filename='results.pkl', constraintFunction=myConstraintFunction)

```

---

## Running NetExplorer Tests in Parallel

Suppose that you have many tests to run (e.g. testing two parameters at 10 points each: that's 100 tests), and more than one CPU available, for example a dual-core processor, or even a cluster of computers. NetExplorer includes tools for running tests on as many cores as you have available, so that you finish the tests much faster.

In particular, you can use the `TestCrossParameters` `hosts` parameter to specify the CPUs you have available. This parameter lists the different host computers to run tests on. You pass in a list of the available computers and the number of node processors to run, as follows:

```
test = TestCrossParameters(...,
                           hosts=[{'np':1, 'hosts': ['computer1']},
                                   {'np':1, 'hosts': ['computer2']}])
```

For a dual-core computer, you would use a list like this:

```
test = TestCrossParameters(...,
                           hosts=[{'np':1, 'hosts': ['localhost']},
                                   {'np':1, 'hosts': ['localhost']}])
```

That list of dictionaries contains some extra complexity for a reason. Suppose that you want to run each test on more than one processor. Each of the inner lists describes the set of hosts for a single test. So, to run two tests, each with 8 node processors on two machines, make this call:

```
test = TestCrossParameters(...,
                           hosts=[{'np':8, 'hosts': ['computer1', 'computer2']},
                                   {'np':8, 'hosts': ['computer3', 'computer4']}])
```

## Parallel HTM Networks

If you wish to perform training or testing on multiple hosts, both `train()` and `test()` must receive a `host` parameters. Each time the function is called, will get the contents of one of the above dictionaries.

There are two ways to get `train()` and `test()` to use this host information correctly.

- The first, and easiest way, is to use `nupic.network.SimpleHTM`. HTM Networks created that way automatically take care of these parameters.
- If your network is more complicated than a `SimpleHTM` allows, and you explicitly create nodes and sessions, you need to specify the `hosts` parameter explicitly in a `SessionServerDistribution`. Look at the code for `SimpleHTM` for an example.



When writing your `NetInterface` and `DataInterface` classes, be sure to use the value of the parameter `prefix` in the names of any files they create. That way, when NetExplorer creates multiple copies of networks and data sets, they will not overwrite each other's files.

Similarly when writing your `NetInterface.test()` function, you should use the `uniqueID` parameter in the names of any files you create. That way, if multiple tests are run at the same time, they won't overwrite each other.

When you run tests in parallel, `NetInterface.train()` and `NetInterface.test()` will be run on *copies* of the objects rather than on the objects themselves. In fact, they run in separate processes, so that all the objects they can see are copies. As a result, you cannot change global state from these methods. Instead, override `NetInterface.postTrainUpdate()` and `NetInterface.postTestUpdate()` to change global state.



# 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 hosts 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 host can have multiple CPUs per host or per chip. For example, a host with two dual-core chips has a total of four CPUs.

**E****Effector**

Effectors are nodes that receive the output of the classifier 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****Host**

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

**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 mode, 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**

See Host.

**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

## Numerics

1-d region 30  
2-d region 30

## B

basic scheduler  
  example 59  
  multiple NPs 59  
bitworm  
  learning 46  
bitworm example 84  
bottomUpOut node output 28  
bundles 49, 50  
  example 49

## C

cluster 69  
clustered environment 69  
clusters  
  run on cluster 77  
clusters 76  
common link types 29  
customer support 12

## D

DataInterface class 103  
different configurations on multi-CPU machines 69  
different schedulers with multiple NPs 59

## E

examples  
  bitworm 84  
  pictures 92  
exception response 21

## F

fully trained node 34

## H

hardware configurations 68  
HTM Network  
  profiling 63  
  running 17, 45  
HTM Network example 33  
HTM Network files  
  format 39  
HTM Networks  
  level skipping 62  
  loading 17  
  parallel 108

## I

implementation 84, 86  
inference 31  
  introduction 34  
inputs 26, 27  
interacting with sessions 53  
interaction example 20  
introduction 14, 66

## J

jobs  
  definition 113

## L

launch.py file 56  
launching on a remote host 78  
learning 31  
  bitworm example 46  
  introduction 33  
  supervised 32  
  unsupervised 32  
learning nodes  
  parameters 36  
level-skipping HTM Network 62  
link policies 29  
links 26, 27, 28  
  common types 29  
  commonly used links 28  
loading an HTM Network 17  
log files 55

**M**

- maxDistance parameter
  - example 37
- MPI 67
- multi-CPU machines 68
- multiple NPs 59, 71, 76
  - basic scheduler 59
  - pipeline scheduler 61
- multiple NRE instances 70

**N**

- NetExplorer 100
  - classes 102
    - DataInterface class 103
    - NetInterface class 104
  - parameterized tools 105
  - running tests in parallel 108
- NetExplorer basics 100
- NetExplorer tests 108
- NetInterface class 104
- network file format 39
- node content 53
- node processors 17
- nodes
  - bottomUpOut 28
  - fully trained 34
  - inputs 26, 27
  - links 26, 27
  - outputs 26, 27
  - parameters 36
  - pass-through 62
- NPs 17
  - multiple 59, 71, 76
  - multiple NPs 76
- NRE
  - multiple instances 70
  - Supervisor 16
- Numenta network file format 39

**O**

- official results
  - definition 114
- offset 29
- open MPI 67
- output information 55
- outputs 26, 27

**P**

- parallel HTM Networks 108
- parameters 36
- pass-through nodes 62
- performing inference at the lower level and learning at the higher level 47

- Pictures
  - Demo 96
  - GUI 96
- Pictures Demo 96
- pictures example 92
- pipeline scheduler 62
  - example 61
  - multiple NPs 61
- problem definition 84, 85

**R**

- regions
  - 1-d 30
  - 2-d 30
- remote host 79
- remote hosts 78
- remote servers 50
- running multiple NRE instances 70
- running one NRE 71
- running the HTM Network 17, 45
- RuntimeResponse structure 20

**S**

- scheduler
  - basic 59
- schedulers
  - overview 58
  - pipeline 62
  - supported schedulers 58
- scripting commands 55
- server cluster 69
- session
  - on remote host 79
  - starting 44
- Session API 44
- session bundle example 49
- session bundles 49
  - example 49
- session commands
  - troubleshooting 55
- session startup 19
- session\_log files 55
- SessionConfiguration object
  - methods 81
- sessions 19
  - interactions 53
  - introduction 19
  - run on cluster 77
  - starting on cluster 76
- SimpleFanin 29
- SimpleHTM
  - multiple NPs 76
- SimpleSensorLink 29
- single-CPU machine 68
- SingleLink 29
- startup 19
- startup sequence 17
- structure of a trained node 34
- supervised learning 32
- Supervisor 16

Supervisor/session interaction 20  
support, contacting 12

## T

technical support 12  
TestCrossParameters class 101  
TestCrossParameters options 102  
timeout response 21  
training 31  
training process 43  
troubleshooting 55  
    node content examination 53  
    output information 55  
    session commands 55

## U

unconnected response 21  
unsupervised learning 32

## W

waves example 85

