# Emergent

## Neural Networks Simulation System

UNIVERSITY OF COLORADO

Emergent v5.3.2 – April 2012

# - CONTENTS -

# About Emergent

The official *emergent* logo, featuring "emer".

See the email Announcement for more specific contrasts with PDP++ (http://psych.colorado.edu/~oreilly/PDP++/) , and Authors for who wrote it, and Grant Support that funded its development. The ChangeLog contains a list of changes for various releases.

**Emergent** (formerly PDP++ (http://psych.colorado.edu/~oreilly/PDP++/) ) is a comprehensive simulation environment for creating complex, sophisticated models of the brain and cognitive processes using neural network models. These networks can also be used for all kinds of other more pragmatic tasks, like predicting the stock market or analyzing data. Emergent includes a full GUI environment for constructing networks and the input/output patterns for the networks to process, and many different analysis tools for understanding what the networks are doing. It has a new tabbed-browser style interface (in Qt 4), with full 3D graphics (via Open Inventor/Coin3D), and powerful new GUI programming tools and data processing and analysis capabilities. It supports the same algorithms as PDP++: Backpropagation (feedforward and recurrent), Self-Organizing (e.g., Hebbian, Kohonen, Competitive Learning), Constraint Satisfaction (e.g., Boltzmann, Hopfield), and the Leabra algorithm that integrates elements of all of the above in one coherent, biologically-plausible framework.

PDP++ (http://psych.colorado.edu/~oreilly/PDP++/) was under constant development since 1993, and over the past several years has been completely overhauled and released with a new name: *emergent* (first released in August of 2007). The GUI was completely rewritten using Qt from Trolltech (the basis of the popular KDE environment) and Coin3d (an implementation of the Open Inventor framework for 3D scenegraph rendering). Everything has been boiled down to the most essential elements, which are now much more powerful and flexible. It is now much easier to understand and modify the flow of processing, and to create programs that automate various tasks, including the creation of complex training environments. Important new visualization tools have been added, and many optimizations have been implemented in terms of the memory footprint and script execution speed. There is also a convenient plugin architecture that makes extending the software much easier than before.

**Contents**

## Reasons to Use Emergent and Motivations for its Development

1. Provides many *powerful visualization and infrastructure tools* tailored to the neural network domain that would otherwise take a lot of individual effort to write.

2. Provides a structured environment and GUI support for making it *as easy as possible for others to use and modify models* (e.g., an automatic GUI interface for operating the models, a GUI programming environment with high-level constructs that are easily understood and modified, builtin documentation system, etc).

3. Completely *open source* -- this makes it much easier to use *emergent* in educational or other cost-sensitive situations, in addition to all of the other major advantages of the open source development model.

4. Highly *optimized runtime performance* and extensive builtin object-oriented support for 'complex network structures -- *supports multiple forms of parallel processing including distributed memory (clusters) and shared memory (threads).*

Although tools like MATLAB satisfy point number 1 above, they do not satisfy the other important points. Put simply, if you are primarily developing new algorithms, use MATLAB. If you are primarily using existing algorithms and developing complex high-performance models, use *emergent* (if you're just using Backprop and relatively simple network architectures, LENS may be faster, but the GUI is not as powerful). If you are making detailed multi-compartment models of individual neurons, use NEURON or GENESIS.

### General Characteristics of Emergent

- Written in C++
- Gui-based, cross platform (Windows, MacX, Linux/Unix)
- Uses the Trolltech Qt graphical and foundation class library see: http://www.trolltech.com
- Use the Coin "Inventor" library for Open-GL-based 3D visualizations see: http://www.coin3d.org
- Supports the multiple different neural network learning and processing algorithms, from backpropagation to more biologically-based algorithms including Leabra.

- Provides for dynamic "plugins" which can extend the main program capabilities in many ways (new network algorithms, new visualizers, new data sources, network interfaces, etc.)
- Provides low-level multi-dimensional matrices, and higher level "tables" (similar to an in-memory database) for data logging and manipulation
- Provides visual graphing and logging
- Networks can be entirely internal (data input/output driven inside the program) or can be hooked to external input/output (ex. a simulation environment, ACT-R, etc.)
- Provides a data analysis framework, including multi-dimensional matrices, statistical processing, data summarization, etc.
- Linked against the Gnu Scientific Library -- our matrix class is compatible with the matrix classes defined in that library see http://www.gnu.org/software/gsl/
- Can be run on clusters under MPI (Message Passing Interface)
- Supports multi-CPU/core computers
- Includes TA and CSS (described below)

## Libraries and Components of Emergent

### The Emergent Toolkit

This is the foundation on which *emergent* is built (see TEMT wiki (http://grey.colorado.edu/temt/index.php/Main_Page) for more details). It includes TA and CSS capabilities described below, basic application framework classes (String, Variant, lists, arrays, matrices, tables), a general-purpose application browser, a tabbed property editor (extensible to provide custom visualization and editing views for various types of application objects), a full 3D visualization/rendering library, low-level text-based CSS scripting and higher-level gui-based Programs (which render CSS scripts to execute.)

### TA (TypeAccess)

TypeAccess (TA) is a sophisticated meta-type information system for C++ programs. The header files of the application are scanned by a program called maketa which builds a comprehensive description of all the classes (`TypeDef`) and their methods (`MethodDef`), members (`MemberDef`), etc. This information is then available at runtime, which enables things like:

- Programmatic access to members and methods
- Automatic streaming of complex object hierarchies
- Automated building of property edit dialogs
- Dynamic creation of object instances based on dynamic runtime typing
- Dynamic generic runtime manipulation of objects, such as querying its members, methods, class type, whether it inherits from a particular class, etc.

TA basically provides the same kinds of services as "reflection (http://java.sun.com/docs/books/tutorial/reflect/index.html) " in systems like Java and C#. A set of directives can be applied to comments in the header files to control runtime behavior, such as visibility of member variables, etc.

### CSS (C-Super Script)

CSS is a C++-like scripting language built in to *emergent*. It is used as the basis of network simulation control, and can also be used to customize data input/output to networks, and to facilitate interactions with external components.

## Third-Party Software Emergent Interfaces with

### Qt

Qt is a comprehensive application framework and gui library. It is C++-based, cross-platform, and runs on virtually every modern system. It also has Python and (new) Java bindings. It has basic classes such as String, Variant, etc., container classes (lists, etc.), a complete graphical framework (windows, widgets, etc.), and libraries for network/socket access, and XML. Qt includes an elegant but sophisticated mechanism "signals and slots" for dynamically connecting objects together at runtime. It also includes a meta-typing system, which provides somewhat of a subset of the TA system.

### Coin / Inventor

"Inventor" is a 3D API for describing collections of 3D objects for visualization. It renders these objects using the OpenGL framework. "Coin" is an open-source implementation of the Inventor API.

# Using emergent

| | |
|---|---|
| **Using emergent** | **Developing emergent** |

## Documentation

This is the place to come if you need help using *emergent*!

The wiki is organized into several major sections, with links to the various subtopics. The main sections of the wiki are:

**Emergent Support**

*emergent*-users mailing list
How to submit a bug report

**Wiki References**

What is a wiki? (http://en.wikipedia.org/wiki/Wiki)
How to edit a page

### ***Must-Read*** **Introductory**

Getting Started -- introductory manual, including how to Build your own network
Concepts -- brief overview of key concepts (especially relative to PDP++)
Tips and Tricks -- quick "top 10" overviews of major operations

### Tutorials and Demos

HowTos -- an alphabetical list of brief recipes on all manner of things, and pointers to many useful things
Tutorials -- detailed step-by-step examples
Demos -- overview of demo projects

### Thorough, reference:

User Guide -- comprehensive reference information on each software element
Network Algorithm Guide -- detailed information about each of the network algorithms
ChangeLog -- brief overview of new features by release -- *always look here if you are updating your version of the program!*

## *Where Do I Start?*

If you haven't installed *emergent* yet, see the *emergent* Installation Guide.

Getting Started provides a quick introduction to the *emergent* application.

Build Your Own Network provides a beginner tutorial on building a very simple network simulation.

The Tutorials section covers more advanced topics. After you have built your own network for the first time, the **AX Tutorial** provides an excellent tutorial which teaches the fundamental tools used in the *emergent* application.

The User Guide provides detailed information on all the major elements used in building *emergent* simulations.

After you have run through the basic tutorials, there are a number of projects available which demonstrate various principles in *emergent*. The *emergent* software is installed with several 'demo' projects which can be used help understand the various objects in the software and how those objects are used in actual simulations. Also, the CCN wiki textbook projects (http://grey.colorado.edu/CompCogNeuro/index.php/CCNBook/Sims/All) referenced below, used in conjunction with the new CCN wiki textbook (http://grey.colorado.edu/CompCogNeuro/index.php/CCNBook/Main) provide an excellent introduction to the principles of computational cognitive neuroscience.

The Network Algorithm Guide provides detailed theory and instructions on the specific network algorithms that can be used to build *emergent* simulations.

The *emergent* Reference has information on the internal and technical aspects of *emergent*, including information required for advanced script programming, and for extending *emergent*.

## CCN Textbook Projects

The wiki textbook on *Computational Cognitive Neuroscience* (http://grey.colorado.edu/CompCogNeuro/index.php/CCNBook/Main) , includes a number of *emergent* projects (http://grey.colorado.edu/CompCogNeuro/index.php/CCNBook/Sims/All) demonstrating various principles of computational cognitive neuroscience. Each of these projects has detailed wiki documentation, and the projects are a great way to learn CCN and *emergent*! The CCN Book (http://grey.colorado.edu/CompCogNeuro/index.php/CCNBook/Main) provides a *complete rewrite* of the first edition CECN textbook *Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain* (O'Reilly & Munakata, MIT Press, 2000), and also replaces the original CECN Projects (http://grey.colorado.edu/CompCogNeuro/index.php/CECN1_Projects) .

- CCN Book (http://grey.colorado.edu/CompCogNeuro/index.php/CCNBook/Main) -- *Computational Cognitive Neuroscience* wiki textbook
- CCN Book Projects (http://grey.colorado.edu/CompCogNeuro/index.php/CCNBook/Sims/All) -- *emergent* projects which complement the *Computational Cognitive Neuroscience* wiki textbook

## External Links

- Teaching Cognitive Modeling Using PDP++ by David C. Noelle: http://www.brains-minds-media.org/archive/1406 -- excellent overview of use of PDP++ for teaching, much of which applies to *emergent*.
- An overview paper on *emergent* by Aisa, Mingus & O'Reilly: The *emergent* neural modeling system. Neural Networks (2008), doi: 10.1016/j.neunet.2008.06.016 (http://dx.doi.org/10.1016/j.neunet.2008.06.016) -- **please reference this paper for any published work you do using *emergent* -- thank you!**

# Getting Started

**Using emergent**

This page presents an overview of *emergent*, starting up, opening and running an existing simulation, and building your first network!

**Contents**

## Running Emergent

The following are the instructions for launching *emergent* in GUI mode, without any command line options. For advanced launch scenarios, such as options, batch (no GUI) operation, or clusters, see the *emergent* Reference.

### Mac

Emergent.app is installed in /Applications along with all the other applications -- you can drag it from there into your Dock if you want. The GUI can also be launched from a terminal shell by typing `emergent`.

### Linux

*emergent* is installed in the Edutainment/Misc/Emergent or Education/Emergent application start menu, depending on KDE or Gnome. The GUI can also be launched from the console by typing `emergent`.

**Microsoft Windows**

*emergent* is installed in C:\Program Files. The GUI can be launched by selecting the *emergent* icon from the Windows Start Menu or from the Windows Desktop. NOTE: The GUI can also be launched by double clicking the Emergent\bin\emergent.exe icon in Explorer, or by typing `emergent` from a Windows Command Prompt.

# Interface basics

This section provides a brief description of the main interface elements of the *emergent* application.

## Menu/Toolbars/Status bar

The following menus are used in the *emergent* user interface:

1. Main Menu — the Main Menu appears on the *top* of each window, or in the menu bar on the Mac
2. Application Toolbar — the Application Toolbar appears directly below the Main Menu
3. Tools — by default the Tools appear on the *left-hand side* of each window
4. Status Bar — the status information appears at the *bottom* of each window
5. Context Menus — the context menus appear when you right click (Mac: ctrl-click) on an object
6. Property Page Menus — the property page menu can be found at the top of the center Panel View for each object when that object is selected, and refers only to that object

### Main Menu

This section describes all the items in the main menu bar of the application.

#### File Menu

New Project - this creates a new project

Open Project - enables you to select a project to open

Open Recent - enables you to quickly open a recent project

Save Project (project window only) - save the current project, prompt for a filename if the project is new

Save Project As (project window only) - save the project under a new name

Save Note Changes (project window only) - prefix new message to the project Change Log

Updt Change Log (project window only) - prefix new message to the project Change Log

Save All Projects - save all currently open projects, prompt for a filename if any project is new

Close Project (project window only) - close the current project (prompts for save if changed)

Options (Mac: 'Preferences' on App menu) - opens the *emergent* application options dialog

Close Window (project window only) - close the current browser window - **note** the project remains open - to reopen the viewer, navigate to the root/projects/YourProject, right click, and choose Open View

Quit (main app window only) - exit *emergent*

#### Edit Menu

Undo -- undo last edit or action -- see css console for information on what was undone -- see Undo

Redo -- redo last action that was Undone -- see Undo

Cut - puts the selected object(s) onto the clipboard; it will be moved if you Paste it somewhere -- see Cut and Paste

Copy - copies the selected object(s) onto the clipboard -- see Cut and Paste

Duplicate (project window only) - makes a duplicate of the selected object(s) at the current location -- see object copy

Paste - inserts the object(s) from the clipboard to the current location -- see Cut and Paste

> 'Paste Into' - the destination for the 'Paste' operation is a list or group - this term helps disambiguate situations where the item could either be placed as a peer of other items at its level, or into a list or group
>
> 'Paste Assign' - the destination for the 'Paste' operation object is a matching object - this term helps disambiguate situations where the item could either be placed as a peer of other items at its level, or assigned to completely replace an existing item

Delete - deletes the selected object(s) -- see object delete

Find - enables you to find an object(s) by specifying many different kinds of criteria

Find Next - not implemented (plan to implement to enable quickly finding the next instance of an object in the using the previous Find criteria)

#### View Menu

Refresh - refreshes the gui so it corresponds to the underlying project items - note that *emergent* usually does a good job of keeping the gui refreshed, but a manual refresh could be needed in some obscure scenarios

Back - browser-style navigation of the project [#Tree View|tree] which allows you to go **back** to the previous object selected in the project tree, and displays the corresponding view in the center editor panel

Forward - browser-style navigation of the project [#Tree View|tree] which allows you to go **forward** to the next object selected in the project tree, and displays the corresponding view in the center editor panel

Frames - enables you to show or hide any of the main frames currently being used, such as the Tree View, Panel View, or T3Frames (3D View). *Note:* You can also dynamically shrink these frames to nothing by using the grab bar that divides frames.

Toolbars - enables you to show or hide any of the current toolbars (only the *emergent* application toolbar is currently used)

Dock Windows - enables you to show or hide any of the current dock windows, such as the programming *toolbox*

Save View - saves the state of the current view - it will be restored to this state next time it is opened (*emergent* currently saves only a few aspects of the view state, such as window size/position, and frame sizes)

### Show Menu

Some objects and properties in *emergent* are marked as being Expert or Hidden.

**Expert** items are typically for advanced usage scenarios, and so are normally not shown.

**Hidden** items are rarely used, except by *emergent* developers.

You can turn on showing of these items globally using this menu. Note that individual property pages enable you to selectively show Expert and Hidden items. If you select Expert or Hidden globally, then the corresponding items are automatically shown in all property panels.

Normal Only - only show normal items, not Expert or Hidden

All - show all items

Normal/Expert/Hidden - individually choose which items to show

You would usually want to leave the 'Show' option set to 'Normal', except possibly to highlight what are Expert and/or Hidden. The 'Normal' items are all that is required for standard usage.

### Tools

Help Browser - opens a browser that enables you to see all the objects and their methods and members, with interactive search and hyperlinks -- extremely helpful!

### Window

This menu provides a list of all browser windows open in the application, so you can easily switch between windows.

### Help

Help - invokes the Help Browser

About - shows information about the version of the application -- useful when reporting bugs

## Application Toolbar

The *emergent* application toolbar has icons that correspond to some frequently used Main Menu items.

## Toolbox

When inside a project, the programming 'Tools' menu (docked by default on the left-hand side of the window) provides access to a toolbox containing clip/drag/droppable basic and advanced *emergent* Program elements.

## Status Bar

The status bar (at the bottom of the window) dynamically displays the help text associated with a objects, menu items, or other selected items in the GUI. For a full listing of the help text, see the Help Browser.

## Context Menus

Most objects in *emergent* feature rich context menus that are invoked by right clicking on the object (Mac: ctrl-click). The context menus allow you to quickly access the large range of menu options which apply to that object (such as 'Find from here' and catagories such as 'Object', 'SelectEdit', and 'Edit')

## Property Page Menus

When an object is selected in the [#Tree View|tree] the *top* of the tabbed object in the center Panel View displays the 'Property Page Menu' for that object. The 'Property Page Menu' is a subset of the full 'Context Menus' (above), and contains relevant menu catagories such as 'Object' and 'SelectEdit'.

# Clipboard / Drag-and-Drop

Emergent makes extensive use of the clipboard, and of Drag and Drop. Most objects in the tree view have clipboard commands associated with them, such as Cut and Paste.

**Clipboard** The clipboard can be used throughout *emergent* to Cut and Paste objects.

Note that when you Cut something, the item is not immediately deleted in the GUI - it is only moved when you perform the corresponding Paste. If you choose Cut and then Paste into an external application (including another instance of *emergent*) then the operation is identical to a Copy/Paste, i.e. the source is not deleted.

Many contexts will accept data from an external application. For example, you can copy data from many common spreadsheet programs (Excel, Open Office Calc, etc.) and then paste it into an *emergent* datatable.

**Drag and Drop** You can Drag and Drop both within an application, and between the application and other applications. For example, you can select a series of cells in a data table, and drag and drop them into a spreadsheet application.

When you complete a drag operation, a context menu will appear at the destination, with the available options, including a Cancel. You will probably want to use this menu until you become familiar with the program; to skip the menu, you can hold down the key indicated next to the operation in the menu.

Many operations such as assignment of an object to a variable can be achieved using drag and drop - you will want to experiment with dragging objects to destinations to discover the operations that can be performed (you can always hit Esc or click Cancel to do nothing.)

## Console

The console is a "command line" inside the *emergent* application. It displays diagnostic messages, and can be used to print object values, list programs, and other operations. The console appearance will differ, depending on your platform. Some platforms provide the option of showing a free-floating console that tracks underneath your project, or embedding it in the main application window. See the Options panel, and experiment.

## Viewers

The main type of window in *emergent* is called a Project viewer -- it is a modular window that can contain any of the following:

- toolboxes (see Tools above)
- toolbars (see Property Page Menus above)
- Tree View on the left side of the window, to the right of the 'Tools' -- with a tree browser view of all the objects in the project -- provides the primary means of accessing objects for editing and further manipulation.
- Panel View tabbed frame in the center -- items selected in the left tree browser will have an edit dialog presented here. also contains view control panels for the 3D viewer objects (see next).
- T3Frames (3D View) tabbed frame on the right -- shows three-dimensional views of Network, DataTable and Virtual Environment objects

### Tree View

The Tree View provides a familiar "explorer"-style browser view of objects, starting at some base object. For the application, this is called **root**; for projects, the root object is the project itself. You can also open browsers that are rooted on any specific object - just right click and select **Browse From Here**. The Tree View contains the following major catagories for each project: docs, wizards, edits, data, data_proc, programs, viewers, and networks.

- The tree will automatically update if objects are added or removed.
- You can view the properties of a given item in the tree by clicking on it with the mouse.
- You can select several items from the tree by holding down the **Ctrl** key while clicking each successive item.

### Error and Enable Highlighting

Many items that can be shown in the Tree View can have an error condition associated with them. The main ones are parameter errors in network simulation objects, and program errors in Programs. When an error is detected, the offending item is highlighted in red - to help you easily find these items (which may be buried inside other items, and not even visible yet in the gui), higher-level items containing error items are highlighted in yellow.

Some items can be disabled (switched off) - for items that support this capability, they will be highlighted in grey

### Color Hints

To help differentiate the various categories of objects in the project, colors have been arbitrarily assigned to categories. These colors are used in the text of items in trees, and as background in edit panels. You can enable or disable the use of color in the browser tree or in the edit panel, by selecting the corresponding **color hints** option from the Preferences.

### Convenience Features

The Tree View provides the following convenience features - most are accessed from the right mouse-click context menu:

- Find from here - you can search for other objects from this point
- clipboard operations (ex Cut, Paste, etc.) where applicable - also generally available on main menu, or via keyboard shortcuts
- Browse from here - you can open a new browser window (tree + panels, no 3D) rooted at the item selected
- Edit Dialog/Panel - you can open an edit dialog on the item, or insure the panel shows the item to edit
- item-specific commands - the most frequently needed commands for this object are available either directly on the right context menu - the remaining commands for the object are in a submenu of it, such as Object/

Expand/Collapse - you can expand or collapse the children, or all children - double click at item to (alternately) expand/collapse all

## Panel View

The Panel View provides panels for editing properties, as well as control panels for objects in the T3Frames (3D View). A Panel provides a viewer for properties of objects, and also provides specialized views, such as list views of lists, tabular views of data, 3D Viewer options, and so on. When more then one sub-panel type is available, the view for a particular panel can be switched by clicking on the view button near the top of the panel (e.g. 'List View' and 'Properties') below the Property Page Menu.

#### tree-to-panel

In general, when you left-click to select an object in the left hand Tree View, then that object will be displayed in the left-most tab in the Panel View. When a new object is selected from the tree view, it will replace the currently displayed object in the **unpinned left-most edit tab**. If you would like a specific object to stay visible in the center panel, you can **pin** the tab by right clicking on the tab and selecting Pin - this is useful so you can return easily to the properties without having to navigate to the object in the tree. Once you pin a tab, a new panel tab is opened for any new objects that you select in the tree view.

#### frame-to-panel

Similarily, when you left-click to select a particular 'frame' tab in the right hand T3Frame (3D View), then that frame's control options will be displayed in the right-most tab in the Panel View. When a new frame tab is selected from the 3D View, it will replace the currently displayed frame options in the **right-most edit tab**. There is only one frame tab automatically provided in the center panel, and it cannot be **pinned**.

### Edit Panels

The most common type of panel is an Edit Panel -- all objects support this view, which enables the properties ("members") of the object to be edited. Edits generally show a list of name/value pairs, with an appropriate control for each value type. A tooltip and statusbar tip generally accompany each control, to provide more detailed information on the item.

When you view an item for the first time, the Apply and Revert buttons at the bottom are both disabled - this means the object has not been changed. As soon as you make a change to any field (except as noted next), the Apply and Revert buttons will be activated. You can continue changing fields, until you are ready to commit your changes by pressing Apply. Once you Apply, the object is updated, and the buttons again are disabled. If you change your mind, and would like to cancel making your changes, press Revert - this will restore the object to the state it had before you started editing it.

All simple controls like check boxes, selectors, etc will be automatically saved/updated if you change that value. In other words, it is the same as editing the item and hitting Apply in one step. Any other pending changes will also be applied at the same time. You cannot Revert from *auto-Apply*.

### List Panels

Lists and Groups ("collections") also have a List Panel that shows the items in the list. You can perform operations on these items, such as moving them via drag and drop, or invoking context menu operations.

### Other Edit Panel Types

Various other objects have specialized additional sub-panels to help view and edit the item. For example, datatables have a built-in editor, programs have a built-in program editor, and so on.

### Control Panels

Many T3Frame (3D View) objects, such as networks and graphs, provide a control panel for easily changing the view parameters. When you have a 3D View frame visible, any associated control panels will be automatically shown in the right hand tab of the Panel View. Only the control panels for the currently selected 3D View frame are shown.

Unlike the Object Panels, Control Panels are always shown, and are automatically pinned in place.

Note that there is often some overlap between the values that can be changed in an Edit Panel for an object, and those that can be changed in a Control Panel. The difference, is that the Control Panel is a customized GUI interface that has been optimized for controlling the item, whereas the Edit Panel is a generic, machine-generated interface.

## 3D View

Emergent enables you to visualize many types of objects in a true 3D Viewer. The 3D Viewer provides a space for one or more 3D Frames (**T3Frames**), each frame can hold a view containing several objects. You can have as many frames as you want, and can visualize an object any number of times in any number of frames. For example, you can show a tabular 3D view of a data table in one frame, and a graph of that data in another frame, or even several graphs in one frame.

Each object type that supports 3D views has an item on its context menu that enables you to make a new view.

To make a new frame, you can either make a view of an object, and leave the frame name set to 'New Frame', or you can make a new blank frame by right clicking any frame tab and selecting 'Add Frame'.

## Application Basics

When you start *emergent* it opens a **root** window with a Tree View Panel and an Edit Panel. The items in this first window are not usually that important when you are using *emergent*. Instead, you usually interact with a Project window, which is visible after you open a project from the *emergent* **root** window.

### Application Objects

All objects in an *emergent* application live in lists (or groups) or other objects. The base of this tree is called **root** and in programs is referred to by a single period **.** . Along with **docs** and **wizards**, the root object contains these important collections:

**projects** - a list of all projects open in the application - *you will rarely if ever need to use this*
**viewers** - this is a list of all non-project viewers open - *you will rarely if ever need to use this*
**plugins** - a list of all plugins that were found on startup - *note that plugins are not loaded automatically, you must enable desired plugins and then restart the application*
**colorspecs** - named color specifications - *you can edit these if the colors are not to your liking*

## Project Basics

An *emergent* project is an object that contains all the other types of objects that make up a network simulation. A project has the following main collections (follow the links for detailed descriptions of the object types):

**docs** - a Doc object is a web-like document that can be used to document a model, provide instructions on operating it, and so on
**wizards** - a Wizard provides automated assistance in creating other objects, such as complete network simulations (note: you will never create these)
**edits** - a SelectEdit provides a way to gather properties from many other objects, to set them in one place - for example, you can collect various network parameters all in one place, to easily set them
**data** - holds datatables - the system automatically provides several subgroups to help organize your data, you can also add new subgroups of your own
**data proc** - holds pre-built DataProc objects - similar to Wizards, these provide various data services that operate on data (note: you will never create these)
**programs** - a Program controls network simulations, data processing, and other automated procedures - when you use a wizard to create a simulation, it automatically imports the required standard programs from the program library
**viewers** - holds 3D Viewer objects - every project gets a Default Viewer, which is created when the project is created, and opened automatically when the project is opened - you can also create additional viewers - you might want to do this for larger projects, to include lesser-used graphs or other 3D views - to open auxiliary views, right click the view and select View Window
**networks** - most network simulations use a single network, but you can create more than one, such as for example to experiment with different configurations

## Running a Simulation

This section provides a simple tutorial on running an existing simulation. Since the ultimate goal of brains is moving the organism around, we will use the **virt_env** simulation to show a simple brain controlling the arm of a simulon.

1. select File/Open Project - navigate to `Emergent/demo/virt_env`
2. select **ve_arm.proj**
   The project will open.
3. in the *left* Tree View, browse to and left-click on: programs/LeabraAll_Std/LeabraBatch
   The **LeabraBatch** edit panel should be showing in the *middle* Edit Panel View.
4. click Init - if it asks to "Initialize network weights?" click "yes".
5. click Run

The simulation should start running, and you will see the simulon attempting to catch the ball in the the *right* 3D Viewer.

## Building your first network!

Ok, we are ready to build your own first network!

## Next steps...

After you have built a simple network, the AX Tutorial provides a comprehensive introduction to many of the more detailed aspects of the *emergent* program. → next AX Tutorial

# Build your own network

**Using emergent**

*This tutorial shows you how to quickly setup your own standard Leabra network, and will introduce you to the key elements of an* emergent *project.*
*This tutorial covers using the new project templates, which were introduced in* emergent *5.3.0. While creating your first project, if you get lost at some point, don't worry. You can optionally close the project without saving, and start all over again. It won't take long to understand the basics of project creation, and you will soon be able to quickly create a new Leabra network.*

**Contents**

## Create A New Project

1. Start by opening *emergent* and creating a new project: File/New Project -- the **Choose Template** dialog will appear.
2. Select *LeabraStd* to create a standard Leabra project, then click **Ok**.
3. The new project will be created, and an *emergent* project window is opened. The new project will have the following structure:

    [default] LeabraProject name = `Project_0`

    [default] LeabraWizard name = `LeabraWizard_0`

    [default] Input DataTable name = `StdInputData`

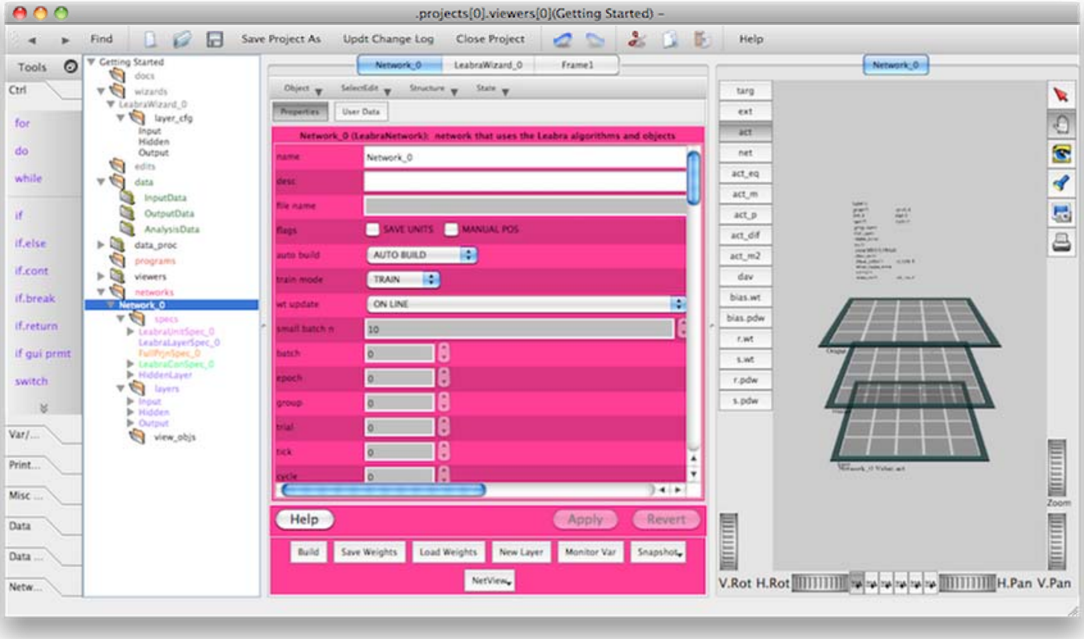
    [default] Output DataTable names = `TrialOutputData` and `EpochOutputData`

    [default] standard programs = include `LeabraBatch`, `LeabraTrain` ... `SaveWeights`

    [default] network name = `Network_0`

        [default] specifications = layer spec, unit spec, bias spec, projection spec, connection spec

        [default] layers = Input, Hidden, and Output



Network_0 (LeabraNetwork): network that uses the Leabra algorithms and objects

name: Network_0
desc:
file name:
flags: □ SAVE UNITS □ MANUAL POS
auto build: AUTO BUILD
train mode: TRAIN
wt update: ON LINE
small batch n: 10
batch: 0
epoch: 0
group: 0
trial: 0
tick: 0
cycle: 0

## Touring the Project

Your new project will open with 4 main panels (from left to right): tools menu, tree view, editor view, and 3D view. The **tools** menu can be turned off/on from the main menu, under **View** → **Dock Windows**. The **tree**, **editor**, and **3D** views can be turned off/on from the main menu under **View** → **Frames**. For now, simply leave all 4 panels visible.

Let's start by browsing through the project structure (you may need to expand the tree to see some of these objects). The tree view contains the following major catagories for each project: docs, wizards, edits, data, data_proc, programs, viewers, and networks. We will be looking at a few of these catagories in more detail:

1. From the top of the **tree** view, select `Project_0` and the main project properties will appear in the **editor** view. The project can be renamed here if you wish.
2. In the **tree** view, under **wizards**, select `LeabraWizard_0` and the *emergent* wizard will appear in the **editor** view. If you had created your new project using the *LeabraBlank* template rather than the *LeabraStd* template, you would likely use this *emergent* wizard to create your network. We will not be using this wizard now, but it is worth reading through. Your new project was created as a **Standard Everything**. Also, take note of the **Network**, **Data**, and **Programs** buttons at the bottom of the **LeabraWizard_0** control panel. These buttons can be used to manually setup a project.
3. In the **tree** view, under **data**, click through the default datatables: `StdInputData`, `TrialOutputData`, `EpochOutputData`. Presently, these datatables are empty. In a few moments we will be updating StdInputData manually to provide network training data. The `TrialOutputData` and `EpochOutputData` will contain the network performance monitoring data that will be recorded when the network is trained or tested.
4. In the **tree** view, under **programs**, check out each program under the **LeabraAll_Std** folder. This group of default programs will be used to train/test the network. As you click on each program, you can read a brief program description in the **editor** view.
5. In the **tree** view, under **networks**, select `Network_0` and the network properties will appear in the **editor** view. The network properties can be customized, but for now we will leave them in their default state.

   In the **tree** view, under the networks → `Network_0`, select `specs` and you will see a summary list of the specifications utilized by the network. If you click through each specification in the tree view, then you can see a full listing of the default parameters for each specification type (layer spec, unit spec, bias spec, projection spec, and connection spec).

   In the **tree** view, under the networks → `Network_0`, select `layers` and you will see a summary list of the layers within the network. The defaults are set to generate a standard three layer network, with one Input layer, one Hidden layer, and one Output layer. The default geometry for each layer is 5 units x 5 units, and corresponds to the size of the data contained in the `StdInputData` datatable.

   *NOTE:* Check the **tree**; you should only have one network. If you accidentally created any extra networks you can right-click on the name for the extra network in the tree frame, and select the delete option to remove the extra networks.

***editor view tabs control TIP*** : If you observe the left-most tab in the **editor** view, you will note that the focus of the tab will automatically shift to whichever object is selected in the **tree** view. You can return to the **LeabraWizard_0** control panel in the **editor** view by selecting the object in the **tree** view, or by selecting the tab in the main panel frame.

***pinned tabs TIP*** : The **LeabraWizard_0** tab should remain available as it was automatically 'pinned', which means fixed in place. You can pin or unpin any tab by right-clicking on the tab itself and selecting 'Pin in Place' or 'Unpin'. If a tab is pinned it will not disappear from the tabs in the center panel when a new focus object is selected from the project tree. Remember that you can re-open any tab by simply selecting the corresponding object in the project tree.

## Create Network Input and Output Patterns

The `StdInputData` must be updated with training data in order for the network to learn anything. This data will be presented on the network's Input and Output layers to train the network to give particular output patterns in response to particular input patterns. For this demo, we will simply make a vertical line, a horizontal line, and a diagonal line as our inputs; with matching output patterns.

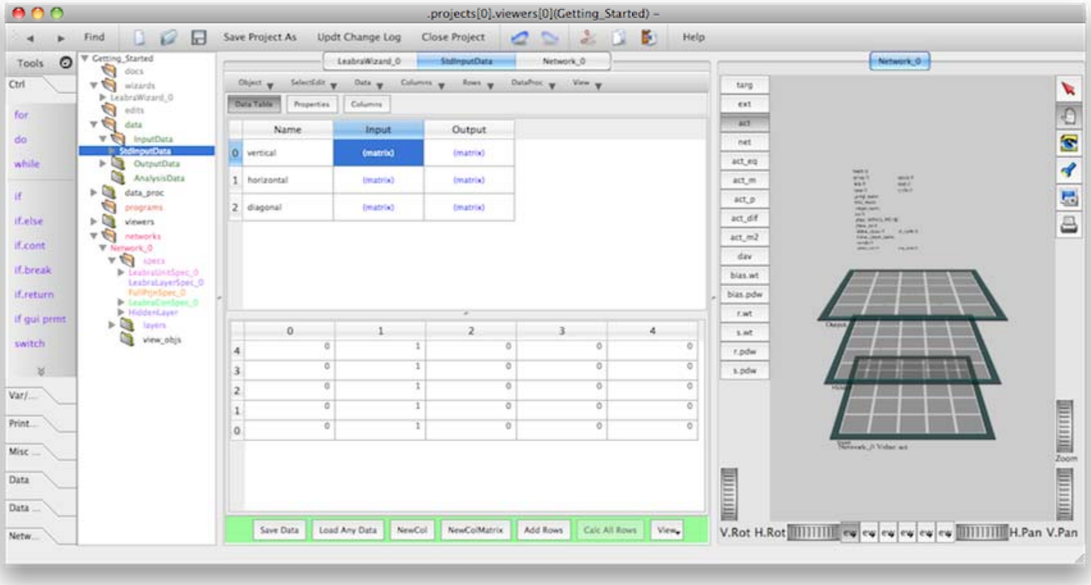
1. In the **tree** view, under the **data** section, click on `StdInputData`. In the `StdInputData` **editor** view, the **Data Table** button should be selected. The table does not contain any rows, but does contain the following columns:

    **Name** is a place to put a distinct name for each pattern you have created. The pattern name can be used in log files, or in programs.

    **Input** is a matrix-type column -- each Input matrix cell corresponds in dimension to the Input layer of your network. Each cell in the Input matrix will contain the value (from 0.0 to 1.0) that will get presented to the network during training and testing, in the Leabra 'plus' phase.

    **Output** is also a matrix-type column -- each Output matrix cell corresponds in dimensions to the Output layer of your network. Each cell will contain the value that you want your network to learn when it is presented with the Input pattern for that table entry (row).
2. We will need to first add three rows to the datatable. In the `StdInputData` **editor** view, select the **Add Rows** button. This will bring up an **AddRows** dialog box. Set `n=3`, then hit **Ok**.
3. Now we can enter our data. In the `StdInputData` **editor** view, select the first matrix cell under the **Input** column. A matrix (table) will appear in the lower half of the panel, and you can manually type in values into each cell:
    - For a **vertical** line: type a '1' into each cell down a single *column* in the Input (matrix) table
    - For a **horizontal** line: type a '1' into each cell across a single *row* in the Input (matrix) table
    - For a **diagonal** line: type a '1' into each cell diagonally across the Input (matrix) table

    Because in this case we are going to train the network to output the same thing as the input, just enter the same values you entered for the **Input** matrix into the corresponding **Output** matrix. Alternatively, you can simply copy the matrix values by selecting a completed **Input** (matrix) by right-clicking on the (matrix) and choosing *Copy*, then right-click on the corresponding **Output** (matrix) and choosing *Paste*.
4. Finally, you can also enter descriptive names for the rows in the 'Name' column (eg. vertical, horizontal, and diagonal). You should now have a completed StdInputData table, with names for each data row, and values entered for both the Input and Output data matrices in each data row. For example, your vertical data should look something like this:



| | Name | Input | Output |
|---|---|---|---|
| 0 | vertical | (matrix) | (matrix) |
| 1 | horizontal | (matrix) | (matrix) |
| 2 | diagonal | (matrix) | (matrix) |

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 4 | 0 | 1 | 0 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 |

*NOTE:* Check to make sure that both your input pattern and output pattern match for each pattern set which will be presented to the network (eg. vertical 'line' for vertical Input and Output patterns, horizontal 'line' for horizontal Input and Output patterns, diagonal 'line' for diagonal Input and Output patterns).

## Run the Network

To see your network in action you will first need to train the network on the data you created, and then you can test the network to see how well it has learned your patterns.

### Training

1. In the **tree**, under **programs**, click on `LeabraTrain` (you may need to first expand the tree by clicking on **programs**). This will cause a `LeabraTrain` control panel to appear in the **editor**. The `LeabraTrain` program will train your network given the `StdInputData` you created. Take note of the **Init**, **Run**, **Step**, **Stop**, and **Abort** and other buttons at the bottom of the `LeabraTrain` control panel.
2. Click **Init**.
3. Click **Run** -- your network should start running, and you should see it operating in the **3D Viewer**.

During training the `StdInputData` you created is presented to the network and the network weights are trained based on the Leabra algorithm. Depending on how fast your computer is, you should observe that each pattern is presented twice in succession:

1. the first time is the **"minus"** phase, in which the network is allowed to run free -- since we started with initially random weights, the output will be random
2. the second phase is the **"plus"** phase, in this phase, the network output is clamped to the desired value; the Leabra algorithm then compares the plus and minus phases, and applies a correction to the weights.

The **Train** program calls **Epoch**, **Epoch** calls **Trial**, **Trial** calls **Settle**, **Settle** calls **Cycle**. As each plus and minus phase has a preset cycle count, if you simply initialize the network (**Init**), set **Step: 10**, and then press the **Cycle** button, the program will execute 10 cycles and will tend to transition into and output the plus and minus phases so you can observe the network learning. Watch the **phase** in the network text display in the **3D Viewer** network view (right-hand panel). Over time, you should see the minus phase patterns starting to approximate the hard clamped output values (the pattern you created in your Output matrix in your **StdInputData**).

### Testing

When the network has finished training, you can manually step through the patterns to see what the output is. To do this, you will change the default stepping program used by the network:

1. You should still have the `LeabraTrain` program selected in the tree and visible in the panel; if not, select it in the **tree**, under **programs**, by clicking on `LeabraTrain`
2. In the `LeabraTrain` **editor**, press the **Program Ctrl** button to show the control panel, and then set the **train_mode=TEST**
3. At the bottom of the **editor** panel, set **Step: 1**, and then press the **Settle** button. The program will execute 1 settling cycle.

Now you can press the **Settle** button again to advance the network through one phase of settling -- the network will alternate between plus and minus phases with each step; in the plus phase you will see the exact output pattern displayed in the Output layer; in the minus phase, you should see a pattern of activation that is at least somewhat close to the output pattern (depending on how long your trained your network, and its batch termination criteria. (See Leabra for more information on the Leabra algorithm.)

# AX Tutorial

**AX Tutorial** → next BuildNet

This tutorial shows you how to construct a simple model of an actual psychological task (the CPT-AX task), from start to finish. This one tutorial touches on all major aspects of the system. The tutorial can be read from a web browser, or accessed as a **`Wikidoc`** from within the *emergent* project. Simply launch *emergent*, open the project (it can be found in the 'Emergent' folder under `demo/leabra/ax_tutorial.proj`), select the **`Wikidoc`** tab which is pinned to the top of the middle edit panel (or accessed from the `docs → Wikidoc` section of the left browser panel).

Many links in within the tutorial refer to the presence of the documentation or objects within the *emergent* project itself, and thus it makes the most sense to follow these directions after loading the `ax_tutorial.proj` project in *emergent*, and opening the tutorials WikiDoc within the project, rather than reading along outside the project.

**Contents**

## Building a Complete Model

This project provides step-by-step directions for constructing a working neural network simulation from the ground up, including programming a simple psychological task (target detection), which we extend through several stages to ultimately simulate the more complex CPT-AX task used in working memory studies.

Some basic terminology:

- **Left *Browser* panel** is the left portion of the window with a "tree" of objects in the simulation (including the network, and the input/output data, etc).
- **Middle *Editor* panel** is where you are currently reading -- it can display different things depending on the selected tabs at the top, and what is currently selected in the left browser panel. The left-most tab in the middle edit panel usually shows what is selected in the left browser panel, and the other tabs with "pins" down are locked in place and contain this document and the Wizard, which we will be making heavy use of. The right-most tab in the middle edit panel represents the configuration information for the current 3D display shown in the right-most view panel. The current "Frame1" tab is empty, because the corresponding "Frame1" view panel is empty.
- **Right *Viewer* panel** shows 3D displays of various simulation objects, including the network, input/output patterns, and graphs of results, etc. The current view displayed is called "Frame1" and is empty.

## Basic Task: Target Detection

The basic task we'll simulate involves presenting letters one at at a time to the network, and having it identify "targets" from "non-targets". The targets are the letters 'A' and 'X', and the non-targets are 'B', 'C', 'Y', and 'Z'.

The network will have six input units representing each of these letters, and two output units, one for "target" and the other for "non-target". It will have a hidden layer to learn the mapping (though this task is initially so trivial that it doesn't even require a hidden layer -- we'll make it harder later).

## Chapters

Here are the steps we'll go through, organized as separate document chapters (which live under the `docs` section of the browser, as does this document):

1. AXTut BuildNet -- build the network
2. AXTut InputData -- make basic patterns (data) to present to the network
3. AXTut Programs -- create and control the programs that perform the simulation
4. AXTut OutputData -- monitor and analyze the performance of the model
5. AXTut TaskProgram -- write a program to construct the task input patterns, including more complex tasks
6. Extras: elaborations that go all the way to the full CPT-AX task
    1. AXTut CPTAX_Program -- extend our basic program to the full CPT-AX task
    2. AXTut PfcBg -- add a prefrontal cortex/basal ganglia to the model to handle the full CPT-AX task

## If You Get Off Track...

In the same directory where you loaded this project is a `ax_tutorial_final.proj` project file, which has the full project that will result from following these directions (not the extras). You can load this project and refer to it to see what things are supposed to look like.

Let's begin...

→ Go to BuildNet

## AX Tutorial Archive

AX_Tutorial_v5.0.2.0

# AXTut BuildNet

← back to AX Tutorial main page | **AX Tutorial** | → next InputData

***NOTE:*** Within *emergent*, you can return to this document by selecting the **`Wikidoc`** tab which is pinned to the top of the middle edit panel (or accessed from the `docs → Wikidoc` section of the left browser panel).

**Contents**

## Building a Network

The wizard makes it easy to create a network. It is located in the `wizards` section of the left browser panel, and can always be found in the `LeabraWizard_0` tab which is pinned to the top of the middle edit panel.

Click the `Network` menu button at the bottom of the panel, and choose the Network/StdNetwork option. This will pop up a dialog box showing the network name and asking for the number of layers you want your network to have. We are going to create a simple 3 layer network so leave it at the default value of 3 and hit ok.

The next screen shows a table with the names of the layers, the type of layer, the layer sizes and their dimensions (Size_X and Size_Y, along with the other layers from which each one will receive connections (RecvPrjns). To get further information about each column you can mouse over the headings. For this network, you should enter dimensions as follows:

| Layer | Size_X | Size_Y |
|---|---|---|
| Input | 3 | 2 |
| Hidden | 4 | 4 |
| Output | 2 | 1 |

This will create a network with 6 input units, 16 hidden units and 2 output units. Since we are making a bidirectionally connected network the input layer should not receive any connections, the hidden layer should receive from both the input and the output and the output layer should receive only from the hidden layer. When you can set everything up hit OK.

You will see a network appear in the right view panel (replacing Frame1), and the left browser will expand to reveal all of the objects created (e.g., Network_0 containing the layers and specifications for the new network). Feel free to click around on these objects now to see what they have in them -- we will just use the defaults so there is no need to change anything.

Now we'll move on to making the **InputData** for the simple target detection task.

Below are a few optional topics that you can explore if you wish (or come back to later at any time).

→ skip optional topics and continue to InputData

## Manipulating the 3D View

The controls for the 3D view panel are located on the extreme right-hand side of the view panel. Drag the mouse over them to see what they do (a "tool tip" should pop up when the mouse hovers over the button).

To begin, you can experiment with the ***hand*** tool -- if you click the mouse and move it around, you'll see that you can manipulate the position of the "camera view" for the 3D view of the network.

Two key tips:

- Holding down the SHIFT key (while moving the mouse) will **pan the view** instead of rotating the view
- Pressing the ***eye*** icon **restores the initial view** (you can also save alternate views by assigning new views to the `View` buttons at the bottom of the view panel.)

At some point you'll discover that if you don't completely stop before lifting the mouse button, the view continues to rotate -- kind of mezmerizing -- apologies if you spend too much time doing this (we certainly have.. :)

If you have a scroll wheel on your mouse, you'll see that it acts like a zoom. The same effect can be had by manually scrolling the **Zoom** wheel.

The **V.Rot** and **H.Rot** wheels rotate precisely around the x and y axes -- these are often more useful than the mouse-based rotation using the "hand" tool because they don't introduce off-angles.

The **Flashlight** button is very useful for zooming in on something of interest (especially for large complicated displays) -- after clicking on it, then click on an object in the view (NOTE: text doesn't work as a target for this purpose). This button stays on until unclicked or another button is clicked, so you can do repeated exploration.

Finally, for extra thrills, you can click the right mouse button (or CTRL+mouse on a Mac) and configure many interesting display options -- check out the different draw styles, and the stereo mode options -- dig out those old red/green stereo glasses!

## Configuring the Network View

Once the network has been created, the right-most tab in the middle edit panel, labeled Network_0 provides various parameters for controlling the network display.

There are 3 main segments in the editor panel for the network view (see the wiki Network View page for more info):

- Primary display parameters at the top (font sizes, display style etc) (mouse over to get more info, and explore!)
- **Unit Display Values** Selector (tab) -- which network variable value to display in the units in the network view (activations vs. weights vs. netinputs etc). If you select one of the weight variables (e.g., `r.wt` for receiving weights into a selected unit; `s.wt` is for sending weights out), you then need to use the red arrow tool in the viewer to select a unit to view -- it will turn green, and you should be able to see its weights. By default, short-cut buttons to select the most common network variables are available in the view panel on the left-hand side of the view.
- **Spec Explorer** (tab) -- this is very handy for seeing where your specs are used in the network -- try clicking on the `HiddenLayer` and then `Input_Output` -- note that the layer border changes color indicating which layers are using these layer specs. You can also use the context menu to edit the network specs right there.

## Changing the Network Layout

You can also configure the network layout interactively in the viewer, including repositioning the layers, and orienting the network display relative to other objects in the view (which we postpone for later, when this arises).

To do this, select the **red arrow** tool, and you'll see that transparent purple arrow objects now appear on each layer, and a box appears on the network text box. These are the manipulation controls. Try clicking on one of the horizontal arrows for the Output layer, and moving it around. This moves within the "horizontal" plane (the X-Y plane for the network). The vertical arrows not surprisingly move in the vertical dimension (the Z axis for the network).

→ next InputData

# AXTut InputData

← previous BuildNet **AX Tutorial** → next Programs

***NOTE:*** Within *emergent*, you can return to this document by selecting the **`Wikidoc`** tab which is pinned to the top of the middle edit panel (or accessed from the `docs` section of the left browser panel).

## Input Data (Patterns to Present to the Network)

We return to the LeabraWizard_0 wizard tab, and at the bottom of the wizard move across to the next menu button labeled `Data`, and select the Data/StdData option. This will bring up a dialog with mostly default information already filled in (some parameters cannot be modified), but there is one parameter we need to specify: `n_patterns`. Enter 6 -- one for each of the different input letters. Press OK.

A "data table" object (much like a spreadsheet or simple data base) is constructed that has columns automatically corresponding to the Input and Output layers of the network, with 6 rows where we can specify the different input patterns to the network, which define our simple target detection task. The `Name` column is useful for labeling our patterns. You will see `(matrix)` in the Input and Output columns, and if you click on `(matrix)`, an extra editor shows up at the bottom of the window to allow you to enter values for each "matrix" of input and output units. The `Name` column, in contrast, has a simple text value for each row (i.e., a "string"), so it can be edited directly in the main table view.

This data table object is called StdInputData and it lives in the `data → InputData → StdInputData` subgroup in the left browser, in case you need to get back to it.

Within the StdInputData table, for the `Output (matrix)` we are calling the first output unit the "non-target" and the second output unit the "target". The active units (set to '1') within the `Input (matrix)` are ordered left-to-right, bottom-to-top. Here is what you should enter for the StdInputData table:

| Name | Input | Output |
|---|---|---|
| A | 000<br>100 | 01 |
| B | 000<br>010 | 10 |
| C | 000<br>001 | 10 |
| X | 100<br>000 | 01 |
| Y | 010<br>000 | 10 |
| Z | 001<br>000 | 10 |

*NOTE:* If you're really lazy, you can just load in these data patterns from the file `sim_tutorial_input_data.dtbl` by doing Object/Load on the StdInputData object. This reloads the entire StdInputData datatable. (For more on datatables, see the wiki Datatable page):

## Visualizing the Data Patterns

The best way to make sure you've entered the right patterns is to create a "Grid View" of your input data -- do this by selecting the View/New Grid View option from the menu at the top of the data table editor that you've been using to enter input patterns with. Go ahead and keep the "New Frame" default for the dialog that pops up (you can also add multiple view elements together in a single 'frame' in the 3D view -- we'll do that later).

This will create a StdInputData view tab in the right view panel, and display your input patterns, which hopefully match those shown in the above table. If not, you can correct them by clicking back on StdInputData data table object in the left browser.

There is also a new StdInputData edit tab in the middle panel, which corresponds to the new grid view and contains various parameters for controlling the configuration of the grid view display. You can mouse-over the fields to get more information. Many of these require you to hit the `Apply` button at the bottom before they take effect in the view.

*NOTE:* By default, in the `Grid Table View` edit options for the StdInputData view, the `Click Vals` option is turned off. If you select the `Click Vals` option, you can now use the viewer's **red arrow** to edit unit values in the grid view display.

The next step is to create programs to control the presentation of these input patterns to the network.

→ next Programs

# AXTut Programs

← previous InputData **AX Tutorial** → next OutputData

***NOTE:*** Within *emergent*, you can return to this document by selecting the **Wikidoc** tab which is pinned to the top of the middle edit panel (or accessed from the docs section of the left browser panel).

## Programs for Controlling the Simulation

Return to the LeabraWizard_0 wizard panel, and at the bottom, click the Programs menu button, and choose the Programs/Std Progs option. This pops up a confirmation dialog explaining that it will create a new set of standard programs based on the project type (this project is a **LeabraProject**). Hit OK.

This created a set of standard Leabra programs used to train the network, which organize the presentation of input patterns to the network into a hierarchy of time scales:

- LeabraBatch -- iterates over multiple simulated *network "subjects"* -- each having their own different random initial weights (we won't use this initially).
- LeabraTrain -- a complete training of the network from random initial weights to final correct performance, by iterating over *multiple "epochs"*
- LeabraEpoch -- one full pass through all of the different task input patterns, by iterating over *multiple "trials"*
- LeabraTrial -- processes one input pattern, using two *phases of "settling"* -- the *minus phase* presents the input stimulus, and allows the network to come up with its own best guess as to the correct response, and the *plus phase* presents the correct answer to allow the network to learn to perform the task correctly.
- LeabraSettle -- multiple updates of neural unit activations to process a given input/output pattern, interated over *multiple "cycles"*
- LeabraCycle -- a single cycle of updating of neural unit activation states (roughly 5-10msec of simulated real time)

There are also some other supporting programs that we'll discuss later.

*NOTE:* You might notice that the ApplyInputs program is opened up to show the LayerWriter_0 object -- this was auto-configured to apply the input data values to the appropriate (same name) layers in the network. If you change the layer names or add additional layers, etc, you may need to go back to this LayerWriter_0 object and hit the Auto Config button to reconfigure it. We'll do this later in the tutorial.

## Running the Simulation

First, make sure you're viewing the Network_0 network in the right side view panel. Then click on the LeabraTrain program, which can be selected from the browser under programs → LeabraAll_Std → LeabraTrain. At the bottom of the LeabraTrain program window, press the Init button, followed by the Run button (these links will actually initialize and run the program for you in *emergent*!).

You should then see the network processing each of the input patterns for the task multiple times, as the program iterates over *epochs* of *trials* of *settles* of *cycles* of processing. Depending on your hardware, this may wiz by in quite a blur.

Once it finishes, you can see more clearly what it is doing by hitting the Step button, which will perform one phase of settling at a time. You should observe that the network gets the correct output unit active in the *minus phase*; look for MINUS_PHASE or PLUS_PHASE in the text displayed in the Network_0 view as you step through the training program. It has successfully learned the task!

We'll learn a lot more about how programs work when we write one from scratch to generate our input data for training the network. If you're adventurous, you can click on the programs and hit the Edit Program button near the top of the program editor panel to see the underlying "guts" that make the programs do what they do. Everything that happens in running the simulation is explicitly listed out, and can be modified in any way that you might want -- this is very powerful and probably a bit dangerous too.. :) *We recommend that you don't do anything to modify the programs at this point.*

The next step is to more clearly monitor the performance of the network as it learns, by recording OutputData from the network.

→ next OutputData.

# AXTut OutputData

← previous Programs **AX Tutorial** → next TaskProgram

***NOTE:*** Within *emergent*, you can return to this document by selecting the **`Wikidoc`** tab which is pinned to the top of the middle edit panel (or accessed from the `docs → Wikidoc` section of the left browser panel).

In this Output Data section we explore various ways to monitor, analyze, and display network output data in order to understand better how the network is performing...

**Contents**

## Graphing Learning Performance over Epochs

To see how your network is learning a given task, the first step is to generate a graph of the learning performance (sum squared error on the training patterns) over epochs. Fortunately, the default programs are already collecting this data for us, so we just need to display the graph.

The data is being recorded in the EpochOutputData datatable object in the `data → OutputData` data group. You should see a few rows of data from the previous run, and you should notice that the `avg_sse` column shows a general decrease in average sum-squared-error on the training patterns, ending in a 0, which is what stopped the training.

To graph this data, you just generate a graph view of this datatable. As a general rule in the software, all view information is always generated by a given main object like a datatable or a network -- **you don't create a graph and then associate data with it** -- it goes the other way. The menu selection to create the graph view from the datatable is View/New Graph View. This time, let's be adventurous and instead of putting this graph in a separate view frame, put it in the `Network_0` frame.

If you happened to have put it in the wrong frame initially, don't worry -- just do the context menu over the frame view tab on the right (the tab should be called `EpochOutputData`), and select `Delete Frame`. Note that there can be multiple views of the same underlying data.

You should see a graph appear in the upper right of your Network_0 display, showing a decreasing `avg_sse` line from left to right. By default the line is (redundantly) color coded for the plot value. You can control this and many other features of the graph display in the graph control panel.

*But wait, where is that panel?* You should see a Network_0 tab in the middle panel tabs. If you click on that tab, you will see the display options for **Network_0**. Look at the bottom of this edit panel, and you will see *tab selectors* for the different view objects within this one view frame: the `Net View` tab for the **Network_0** network view and the `Graph Table View` tab for the **EpochOutputData** graph view. Select the `Graph Table View` tab. Now you can mouse over the various controls and play around with them. As you can see, there are many different ways of configuring a graph -- feel free to explore. Note that there are several other variables that you could plot, including average cycles to settle (`avg_cycles`), and a count of the number of errors made across trials (`cnt_err`). Also see the wiki Graph View page for more details.

To see your graph updating in real-time, you can re-initialize and run the LeabraTrain program: Init and Run.

### Arranging the 3D View

Although the current Network_0 view is sufficient, it could be configured to look better. We could shrink the graph view a bit, and orient it better. To do this (optional), click on the **red arrow** button to the right of the view, and then grab the upper horizontal bar of the small purple box in the lower-left hand corner of the graph view display. Drag this slowly down -- you'll see the green frame rotating as you do. Do this to the point where graph is angled more "head on". Similarly, you can grab the left vertical bar and rotate the graph to the left a bit to make it more face on. Next, grab any corner of the box, and shrink the view a bit (maybe to half or so of its original size). Finally, you can move the view down and to the right a bit, by clicking on the face of the cube (not on any of the purple elements), to fit in between the Hidden and Output layers.

When you've got it the way you want, you can press the **eye** button to reset the display to fit, and then maybe Zoom in a bit. You could perhaps pan using SHIFT-mouse drag to center the view (or use **H.Pan** and **V.Pan**). When it looks good, *click and hold* the **View_0** button at the bottom of the view to save your new view. You can also rename the view if you like.

*NOTE:* The position, scale, and orientation of each object in the viewer can also be set manually in the viewers object properties, but we will not cover that here.

## Recording Network Activations for a Grid View

Another common analysis task is to look at the pattern of activations across trials. To do this, we need to record activation values to our trial-level output data table (which was automatically created by the wizard). One way to do this is to select the network object in the Network_0 view by clicking on the thin green frame surrounding the network, and then using the right mouse button to get the context menu, and select [[.networks[0].MonitorVar()|Monitor Var]]. For the `net_mon` field, click and select the `trial_netmon`, which is for monitoring information at the trial level (the other one is for the epoch level). For the `variable` field, type in `act_m`, to record *minus-phase* activations (the monitor runs at the end of the trial, after the *plus-phase*). This will record activations for all three layers in the network in one easy step.

*NOTE:* You could alternatively select `Monitor Var` from the `Network_0` object in the left browser tree, on each of the layers individually, or on any other object in the network for that matter, and record any variable. In the left browser tree, the `Monitor Var` operation can be selected by right clicking on the target object and selecting `Monitor Var` from the context menu. See the wiki Monitor Data for more information on using the **NetMonitor** object in *emergent*.

Next, we need to make a Grid View of the resulting activation data, which will be recorded in the .data.gp.OutputData.TrialOutputData object -- do a View/New Grid View, and again let's put this in the **Network_0** frame. Follow the same general steps as before (see "Arranging the 3D View" above) to position this new grid view into the bottom right hand region of the view.

The grid view will not contain new information until the LeabraTrain program is Init and Run again. After doing that, you need to scroll the grid view display all the way over to the right -- there are too many columns to fit within the 5 columns that the standard grid view is configured to display. To do this, select the **red arrow** view tool and drag the purple bar at the bottom of the `TrialOutputData` grid view all the way to the right. You should see some colored squares with the Input, Hidden, and Output column headers.

To really make things clean, you can **hide** the column headers of the columns you don't want to display (e.g., ext_rew, minus cycles). Select the **red arrow** view tool, left-click to highlight the column header you wish to remove, then right-click to bring up the context menu for the column header. Select `GridColView` and then `Hide`. Ideally, you'd just want to see the trial name, sse, and the different layer activation columns.

*NOTE:* If you make a mistake, **Undo** (**Ctrl-Z** key board shortcut) can be used to undo the last action. If you wish to restore all hidden columns in the grid view object, first select the **red arrow** view tool. Next, left-click to highlight the grid view object, and then right-click to bring up the context menu. Select `GridTableView` and then `ShowAllCols`. This will restore *all hidden columns*, so you can start over and **hide** any of the columns you do not wish to display.

As there are only 6 events in this training program, we can set the rows to display to 6 in the grid view panel, to make the display fit just right.

Init and Run the LeabraTrain program again to update the display.

## Analyzing the Hidden Layer Representations

Now that we have some data from the network, we can perform some powerful analysis techniques on that data.

First, we can make a cluster plot of the Hidden Layer activations, to see if we can understand better what is being encoded. To do this, find the data_proc → data_anal object in the left browser. This contains many useful analysis tools, organized by different topics in the buttons at the bottom. Select HighDim/Cluster, and set the following parameters (leave the rest at their defaults):

- `view` = **on** (generate a graph of the cluster data)
- `src_data` = **TrialOutputData**
- `data_col_nm` = **Hidden_act_m** (specifies the column of data that we want to cluster)
- `name_col_name` = **trial_name** (specifies the column with labels for the cluster nodes)

You should see a new graph show up, with the A,B,C,X,Y,Z labels grouped together into different clusters. Most likely, you should observe in a trained network that the A and X are grouped together, separate from the other items. Can you figure out why this would be the case?

Another way to process this high-dimensional activation pattern data is to project it down into 2 dimensions. Two techniques for this are principal components analysis (PCA) and multidimensiona scaling (MDS). To try PCA, select HighDim/PCA2dPrjn -- fill in the same info you did for the Cluster plot. You should see that the labels are distributed as points in a two-dimensional space, with the X-axis being the dimension (principal component) of the hidden layer activation patterns that captures the greatest amount of variance across patterns, and the Y-axis being the second strongest component. Accordingly, you should see A and X on the left or the right side of the graph, and the others on the other side. It is not clear what the vertical axis represents..

There are many more things one could do, but hopefully this gives a flavor. The next step:TaskProgram is to write a program to automatically generate our input patterns for training the network -- initially we'll start out with the simple task we ran already, but then we'll progressively expand to more complex tasks.

→ next TaskProgram

# AXTut TaskProgram

← previous OutputData **AX Tutorial** → next CPTAX_Program

***NOTE:*** Within *emergent*, you can return to this document by selecting the **`Wikidoc`** tab which is pinned to the top of the middle edit panel (or accessed from the `docs` section of the left browser panel).

**Programming the Task Environment**

The goal here is to write a program that will automatically generate a set of input/output patterns to train the network. Because the task is so easy (at least to start), this will not represent a savings in time, but will hopefully generate understanding of how programs work, and will also provide a basis for making more complex programs.

The major steps involved are:

1. Create the new program object
2. Initialize "enums" based on unit names -- allows us to refer to units by name (an enum is geek-speak for an enumerated set of labeled values -- more later).
3. Iterate over the input units and generate the appropriate output response.
4. Write the appropriate information into the input datatable.

**Contents**

## Create the Program

In the context menu (right mouse or Ctrl+mouse on mac) on the `programs` item in the left browser, select New. The default parameters are fine, so hit `OK`.

This should have created a Program_11 program, which you should now click on, and change the name to: **`AXTaskGen`** (the rest of the links here will assume this name, so enter exactly that name).

It is good to get in the habit of entering descriptions of various objects in your simulation, especially programs, so enter something like **"generates the simple A-X target detection task"** in the `desc` field of the AXTaskGen program.

Note that there are three sub-tabs or sub-panels for a program in *emergent*:

- `Program Ctrl`: For the "end user" to control the running and key parameters of the program
- `Edit Program`: For writing the program.
- `Properties`: For setting overall parameters of the program object, including `tags` which help people find this program if it is uploaded to a common repository, and `flags` that determine various advanced properties.

Select `Edit Program` and we can get started doing that!

### Overview of Programing Process

Programming in this system mostly consists of dragging program elements from the **Tools** at the very left edge of the display into your program, and then configuring their properties (drag-and-drop and duplicate are also very handy here).

The **`program`** object has several different locations for different types of program elements:

- **`objs`** -- place to put misc objects that contain local data for the program (e.g., a local DataTable that might hold some intermediate processing data for the program).
- **`types`** -- special user-defined types that define properties of corresponding variables (e.g., the *enums* we'll be using)
- **`args`** -- (short for arguments) this is where you put variables that other programs will set when they run this program
- **`vars`** -- place for other non-argument variables that are used in the program
- **`functions`** -- you can define subroutines (functions) that can be called within a program to perform a given task that needs to be done repeatedly. These functions are only accessible from *within* this given program
- **`init_code`** -- actions to be performed when the user presses the **Init** button -- to initialize the program and other objects that it operates on (e.g., initializing the network weights, as the LeabraTrain initialization process does).
- **`prog_code`** -- finally, this is where the main code for your program goes! Because the program code can depend on any of the preceding elements, it logically comes last (and it is typically the largest).

In the **Tools**, the program elements are organized into various sub-categories (Network, Ctrl, Var/Fun, etc). Take a look through these categories and use the mouse-over to see what kinds of things are available.

## Initialize Unit Names and Enums

To begin your program, locate the `Network` category in **Tools**, and drag (click and hold and move the mouse) the `init nm units` element into the `init_code` section of your program.

This `init nm units` program element is a very powerful, and automatically runs through a number of configuration steps when it is first dropped into place. You'll see various things being created in your project, and you should get an error message indicating that it could not find the `input_data` table. Just hit `OK` in response to the error message, and let's take stock of what just happened (and fix the error).

You should see that a variable named **`input_data`** was created in the `args` section, and **`unit_names`** was created in the `vars` section. These are both "pointer" variables that provide a local "handle" within the program to refer to objects that actually live outside of the program, in the `data` section of the overall project.

Click on the input_data object, and take some time to mouse over the various fields and read the tooltips. As our first change, we want to set the `object_val` field to point to our **StdInputData** datatable -- so click on `object_val` field and select **StdInputData** from the list.

Now go down to the unit_names variable, and note that it is already set to point to the UnitNames datatable, which was automatically created under the `data → InputData` data subgroup in the project. The UnitNames datatable will eventually contain a single row of data, with labels for each of the units in the StdInputData datatable. However, right now it is empty, because we have not yet initialized the **UnitNames** datatable from the **StdInputData** datatable (we only set it up).

### Entering UnitNames

Now that we have set `input_data`, we can go back to select the `Init Named Units` in the `init_code`, and hit the `Init Names Table` button ([[.programs.AXTaskGen.init_code[0].InitNamesTable()|Init Names Table]]). This will pull up an informational dialog -- hit `OK`. Now go back up to the UnitNames datatable, and you should see two columns: `Input` and `Output`, with a single row of data.

Click on the `Input (matrix)` and enter text labels for each of the units, as follows:

X Y Z

A B C

Click on the `Output (matrix)` and enter N and T (for non-target and target, respectively), as follows:

N T

### View Data Legend

Next, go back to select `Init Named Units` object again ([[.programs.AXTaskGen.init_code[0]|Init Named Units]]), and select the `View Data Legend` button ([[.programs.AXTaskGen.init_code[0].ViewDataLegend()|View Data Legend]]). This will configure a new view frame which will include the input data patterns, plus a legend from the UnitNames datatable showing what each of the unit names are. These same names can also be applied directly to the network to label the units -- we'll do that later. You might want to remove your other view frame for **StdInputData** (without the legend) -- right click on the old **StdInputData** view tab to bring up the context menu and select `Delete Frame`. Now you should see only one StdInputData tab in the view panel.

### Creating Enums

Next, we'll create those *enums* mentioned previously. Select the `Init Named Units` object again ([[.programs.AXTaskGen.init_code[0]|Init Named Units object]]), and select the `Init Dyn Enums` button ([[.programs.AXTaskGen.init_code[0].InitDynEnums()|Init Dyn Enums]]). You will see two new entries in the `types` section of your program -- `Input` and `Output`.

Under the `Input (DynEnumType)`, you should see six items with names like *I_A*, *I_B*, etc., plus a final *NInputs* that indicates the total number of input units. Under the `Output (DynEnumType)`, you should see three items with names *O_N*, *O_T*, and *NOutputs* (the last item indicates the total number of output units). The first letter is taken from the first letter of the layer (I = Input, O = Output), and the remainder after the underbar is the name you entered in the UnitNames datatable.

The purpose of these *enum* types is to allow you to use a symbol to refer to a unit. If you want to activate the X input unit, you can use the I_X enum value to do that. It represents the *index* of the X unit within the input layer -- when you click on *I_X*, you can see that it has a value of **3**. *Enums* have both numeric and symbolic (name-like) properties, and can be converted to and from names and numbers. You'll understand more about why they are so useful as we go along.

We are done with the unit names for now, and can move on to writing our program.

## Iterating over the Inputs

The core of our program will be to *loop* or *iterate* over each of the possible input units, and then generate an appropriate output for each. We can use a **for loop** for this purpose.

In the left-side **Tools**, click on the `Ctrl` category (for "control"), and drag the `for` element into your program code (`prog_code` section within `programs → AXTaskGen`).

You should see three main fields in the for loop object in the program code: init, test, iter -- the default values produce a for loop that loops ten times (counts from **0** to **9**):

- init is for initializing your looping variable (i = 0), where i is the integer variable that keeps track of where we are in the loop -- it was automatically created by the for loop element.
- test is for testing when to terminate the loop (i < 10) -- as long as the i variable remains less than **10**, we continue looping.
- iter is what to do on each iteration prior to the test (i++) -- i.e., increment the loop variable.

To see this in action, let's drag print_var from the Print.. category of program elements (in **Tools**) into the loop_code of the for loop. This is where we put the program elements ("code") that we want to run during each iteration of the loop. Select the i variable for the print_var field.

Now we can Init and Run our simple **AXTaskGen** program. You should see a sequence of "i=0, i=1...i=9" in the console window (typically located below the main project window). It is a very good idea to keep that window visible during when creating and running programs, as various informative messages may show up there.

Although perhaps fascinating for new programmers, this for loop is not exactly what we want. We want to iterate over the input units, not just over the numbers from 0-9. To do that, we need to click on the i variable in the vars section of the program. Change the var_type to **DynEnum** instead of **Int**. Then, click on the enum_type field and select the **enum Input** type (which is what we created earlier). You can also change the name of this i variable to something more expressive, like input_unit. Note that when you apply this name change, the for loop code automatically updates to use this new name, as does the print code.

Let's go back to that for loop ([[.programs.AXTaskGen.prog_code[0]|for loop]]), and change the test field to: input_unit <= I_Z. Note how you can just type in **I_Z** and this is automatically treated as a number -- this is what enums do.

*Note:* On **Lookup** functionality... You can auto-complete many fields without much typing by selecting from a list of known variables and enums by using the **Ctrl-L** keyboard shortcut (lookup_var and lookup_enum buttons are available in some cases).

Init and Run the **AXTaskGen** program again. You should now see the following sequence appear in the console window:

```
input_unit = I_A
input_unit = I_B
input_unit = I_C
input_unit = I_X
input_unit = I_Y
input_unit = I_Z
```

## Generating the Correct Output

Next, we need to generate the correct output (target or non-target) corresponding to each input value. To do this, we first need to create a variable to hold the output value. Goto Var/Fun in the **Tools**, and drag the first Var item into your program vars (select "Copy Here", not "Add Var To", when you drop -- we'll explain later). Set the variable's name to **output_unit**, change the var_type to **DynEnum**, and then click on the enum_type field and select the **enum Output** type.

Now we just need to set this variable inside our for loop code, depending on the value of the input_unit variable. In the left-side **Tools**, click on the Ctrl category, and drag the if.else element into the for loop code, and drop it on top of the PrintVar (it will become the first element in the loop). In the cond *conditional* expression, enter: input_unit == I_A || input_unit == I_X (again note that you can use the var lookup and enum lookup to save some typing and make sure you've spelled them correctly). The == is the logical test for equality, and the || is the logical OR operator (this is standard C++ syntax -- any valid C++ expression can be entered here). This is the condition for a target.

We need to assign our output_unit variable to the target value when this "if condition" is true, and to the non-target case otherwise. To assign a variable value, drag var= from the Var/Fun category to the true_code section of your if statement. For the result_var, select **output_unit**, and for the expr expression, type in **O_T** (or choose using the enum_lookup). This sets **output_unit** to **O_T** (target) when the condition is true. As a timesaver, drag this var=AssignExpr code from true_code into false_code, use Copy Here -- then you can just change **O_T** to **O_N** very quickly. This sets **output_unit** to **O_N** (non-target) when the condition is false.

*NOTE:* On **drag-n-drop**... When using drag-n-drop to add elements to your code, if you drop the new element **onto** a "code" folder, the Copy Into option is used and the new element will be appended (added as the last item) in the "code" folder. If you drop the new element **onto** an existing element, the Copy Here option is used and the new element will be added in front of the existing element. Dropping **onto** is indicated by a box around the folder or element you are dragging over. You can also slowly drag-n-drop a new element, waiting for a *line* to appear to indicate the exact position where the new element will be placed using Copy Here (rather than a box indicating drop **onto**).

Finally, go to your earlier PrintVar statement and select **output_unit** for print_var1 (or print_var2 if that is easier to see). Init and Run your program, and you should see the correct values for input_unit and output_unit being displayed in your console! It should look something like this:

```
input_unit = I_A output_unit = O_T
input_unit = I_B output_unit = O_N
input_unit = I_C output_unit = O_N
input_unit = I_X output_unit = O_T
input_unit = I_Y output_unit = O_N
input_unit = I_Z output_unit = O_N
```

## Writing the Data to the Input Data Table

Now that you have all the key logic of your task, you just need to write the results to the input data table. There are three main steps for this:

1. Erase any existing data at the start
2. In the `for` loop, add a new row
3. In the `for` loop, write the data for each input/output pattern to the new row
4. In the `for` loop, tell the system that we're done writing to each row so it can update the view

The first step is achieved by dragging a `reset_rows` element from the `Data` **Tools** to the first line of the program code (drop right on top of the `for` loop). Then select the **input_data** variable for the `data_var` field. This `ResetDataRows` element will erase any existing data in the StdInputData datatable when the program is run.

Next, drag and drop the `new_row` element from the `Data` **Tools** onto the `loop code` of the `for` loop. The `AddNewDataRow` element will appear at the end of the `loop code` after the `if` statement. Again select **input_data** for the `data_var` field. `AddNewDataRow` will add a new blank row to the StdInputData datatable.

Then, drag and drop the `set units var` element from the `Network` **Tools** onto the `loop code` of the `for` loop. The `Set Unit Vars` element should appear directly after `AddNewDataRow` element in your program code. `Set Unit Vars` is a magic little program element that uses the name of the *DynEnum* type of a variable, plus its value, to determine which unit to activate in the input datatable. All you have to do is select **input_unit** for `unit 1`, and **output_unit** for `unit 2`, and you're done -- it automatically located the `input_data` datatable (based on its name). Note that you want to make sure you don't put one of the vars as the `offset` variable, which allows you to have, for example, multiple slots of the same units repeated in an input layer -- the offset determines which slot to put things in. `Set Unit Vars` will write data to the blank row you just created.

Finally, go back to the `Data` **Tools**, and drag and drop the `row_done` element onto the `loop code` of the `for` loop. `DoneWritingDataRow` simply lets the system know that you're done writing to the current row of data, and that it can update any relevant displays.

Congratulations -- you're done!!! Select the StdInputData tab in the 3D view area, and then Init and Run your program -- you should see the display update with the correct patterns freshly generated by your program!

You have now done a little bit of each of the critical main elements of simulating using this system. The next few steps in the tutorial take this simple starting point and go all the way to a more scientifically interesting model of an actual psychological task, the CPT-AX task.

→ next CPTAX_Program

# AXTut CPTAX Program

← previous TaskProgram **AX Tutorial** → next PfcBg

The next challenge is to write a program that will generate the CPT-AX task (CPT stands for continuous performance task), which is the logical extension of our simple AX task to the *sequential* domain. Instead of A and X each being targets, the target is now an A *followed by* an X in sequence. Other sequences such as A followed by Y or B followed by X are non-target sequences, which nevertheless overlap with the target sequence. In our simplified version of this task (and in several of the actual experiments on people), we restrict the sequences to cue-probe pairs, where cues are A,B,C and probes are X,Y,Z.

See e.g., Braver, T.S., Barch, D.M. & Cohen, J.D. (1999). Cognition and control in schizophrenia: A computational model of dopamine and prefrontal function. *Biological Psychiatry, 46,* 312-328, for an application of this task and further discussion and references.

One implementational detail in how this task is run is key for generating interesting behavioral and neural data: the frequency of the A-X target sequence is set to be relatively high (typically 70%), so that it becomes the default expectation. Then, the two related non-target sequences become much more interesting. For A-Y, there should be a strong expectation of getting an X, which will be influenced by the extent to which the A cue is well remembered. Errors on this trial type, where people might press "target" at the Y, would actually suggest strong maintenance of the A cue. A similar argument applies to B-X, where the X is typically a target, but if you remember the B cue, you should not press the target key. The C,Z items serve as baseline controls, as does the B-Y sequence.

**Contents**

## Plan for the Program

With the above in mind, we can sketch out the logic of our overall program:

- Flip a weighted coin to determine whether we want to generate a target sequence or not. 70% of the time we generate a target sequence, in which case we just produce A followed by X and that is easy.
- Generating a non-target sequence is harder. We need to randomly select from the cues (A,B,C) and the probes (X,Y,Z), while ensuring that we don't randomly pick A-X. We'll discuss a couple of different strategies for this.
- We can do the above cue-probe generation process multiple times to generate a larger set of trials that we will run on a given epoch worth of network training. That is just a simple for loop around the above code.

### List of Variables

Once we have this plan in place, we can create a set of variables that we'll need -- the general flow of programming in this system involves creating variables and then opearating on them, so getting the variables down is the key first step:

- `pct_target` -- how frequent should the target sequence be? this is actually a proportion, but pct is a much simpler label -- for the default case it should be .7
- `rnd_number` -- a random number between 0 and 1 (floating point or Real) that we'll generate to simulate the flipping of a weighted coin.
- `cue` -- the identity of the cue input (A,B, or C) represented as a DynEnum of type Input, taking on values I_A, I_B, or I_C.
- `prob` -- the identity of the probe input (X,Y, or Z), represented by an Input DynEnum as well.
- `output_unit` -- correct answer for the output layer (DynEnum of type Output) -- we'll have the model respond "non-target" for all the cue items, and "target" for the targets.

## Getting Started: Copy and Modify

The easiest way to get started is to duplicate and modify the existing AXTaskGen program (this is a general rule -- if there is a program that has several elements that you want, just copy and modify instead of starting from scratch). To do this, click on the AXTaskGen program in the left browser, and use the context menu to select Duplicate. In the new program, enter the name as CPTAXGen, and update the description to reflect what we're doing.

Now go to EditProgram, and click on the ForLoop object in the prog_code, and use the context menu to Delete that object -- this will also delete everything within it, which is almost all of the code from the previous program. All that should remain is the ResetDataRows at the start (which we can use in any case).

We can now setup our variables as indicated above. Just rename input_unit to cue, then duplicate it and call it probe. Then drag a new var (from Var/Fun toolbox) into vars and call it "pct_target", and set the type to Real, and enter a value of .7. Duplicate it, and call it "rnd_number". It would be a good idea to enter the descriptions for each variable in their desc fields (you can copy and paste from the above text if you want).

## Flipping a Weighted Coin For the Target

The first step in our actual program code is to flip a weighted coin to decide if it is a target sequence or not. We do this by generating a random number (rnd_number) which is uniformly distributed between 0 and 1. We then see if this number is *less than* our target percent value -- this will be true 70% of the time for a value of .7, and that is what we want!

- in the `Misc Fun` **Tools**, there is a `random()` element -- grab that and put it at the end of your program (drop on prog_code or after the reset data rows). Set the result_var to rnd_number, and click on the method -- in the browser window that comes up, you can select different categories (in the menu at the top) of random numbers to generate -- select "Float", and then pick ZeroOne. This will set rnd_number = a random number between 0 and 1.
- Go to `Ctrl` **Tools** and drag `if` to the end of your program. In the cond expression, type/select: `rnd_number < pct_target`.

The true_code for this if is now the target case, and the false_code is the non-target. To set the target values, we just need to assign cue and probe to A and X respectively. Drag var= from the Var/Fun toolbox into true_code, and set the result_var to cue, and choose the I_A enum from the enum_lookup button. Drag var= again and set probe = I_X. Finally, drag var= and set output_unit = O_T.

## Generating the Non-Target

There are two strategies for generating the non-target sequence that excludes A-X:

- Brute force: randomly generate a cue and a probe and check that they aren't A-X -- if they are, then repeat the process until they aren't. This is not particularly efficient, but it is easy to code.
- Choose from a list: generate a list of all possible cue-probe combinations, remove A-X from this list, and then randomly select an item from this list. This is more efficient overall, but harder to code. It is left as an excercise for later, as it demonstrates some important techniques.

To do the brute-force method, you need to enclose the random generation code in a "do" loop, which does some things (generates the random cue/probe) and then tests whether it should loop again (if it is A-X) or not.

Drag the do element from `Ctrl` **Tools** into your false_code of the target if test. Enter/lookup `cue == I_A && probe == I_X` as the test for continuing to loop ( == is the equality operator, and && is logical AND).

Now inside the loop_code, we need to randomly generate a cue and a probe. Drag the random() guy from Misc Fun in there, and set the result_var to cue. For the method, select the Int category, and choose IntMinMax -- we'll specify a minimum and maximum value to generate random numbers between. Notice that the min and max arguments open up below this element -- these are the values that will be passed to the IntMinMax function. Click on min, and enter/lookup I_A. For the max, enter I_C + 1, because this IntMinMax function generates values *exclusive* of max (this is consistent with the C programming language convention, where values go between 0 and n-1 instead of 1 to n).

Just duplicate this RandomCall element, and change cue -> probe and min = I_X, max = I_Z+1. Finally, drag var= and set output_unit = O_N to show that this is a non-target case (quicker to drag AssignExpr guy from true_code and change O_T to O_N).

To test the program at this point, you can drag a print var object to the end of the code, and select cue, probe, and output_unit for the vars to print, and do Init and Run and see that it tends to produce a predominance of A-X and O_T. To really test it, set pct_target = 0, and Run some more. You should never see an A-X, and only O_N.

## Generating the Input Data Patterns

The last step is to produce the input data patterns for the values we have generated. This is just a matter of adding a couple of `new row` guys from the Data toolbox and `set units var` from the Network toolbox to set the units.

- drag the `new row` from Data Toolbox to the end of the program (drop on prog_code) and set data_var to input_data.
- drag the `set units var` from Network toobox to the end, and set unit 1 to the cue variable.
- Because the output is a literal in this case (O_N or non-target), we cannot set it using this element, which requires a variable. So, you need to drag the `set units lit` to the end, and set the enum_type to Output, and the value to O_N.
- drag and drop the AddNewDataRow and SetUnitsVar guys on top of prog_code and select Copy Into to duplicate them at the end of the program, and change cue to probe in the set units var, and select output_unit for unit 2 to also set that guy.
- finally, drag a Data/row done guy to the end to tell it to update the view (select input_data as the data_var).

Init and Run the program while looking at the StdInputData view tab. You should see it generate a valid cue-probe input that matches what is printed out on the console. Keep running to see a range of inputs.

## Generating Multiple Cue-Probe Trials

The last bit of programming needed is to simply loop over the existing set of code multiple times to create several cue-probe sets per epoch for the network to train on. Drag a for loop from Ctrl on top of the 2nd line of the program (RandomCall). Then, multi-select the rest of the program code (only the guys at the main level, not the true_code or false_code within the if statement (this is done with alt-click on linux or mac-command-click on the mac -- note that order matters so select down in order!), and then drag the whole thing into the loop_code of the for guy. Pretty slick.

If you just Run it like this, you'll get 10 trials. It would be good to make the number of trials a variable that can be set. Drag a var into vars and call it "n_trials", and set it to 50 (Int = integer type). Then, click on the for loop and replace the 10 in the test expression with n_trials.

You can probably turn off the console printout by now -- just click on the PrintVar guy and click the OFF flag -- this keeps it around in case you want to do some debugging or something later, but it is not actually used in the code.

## Updating the Control Panel

If you go back to the Program Ctrl tab for the CPTAXGen program (instead of Edit Program where we've been), you'll see that all of the args and vars are present there. However, some of those vars are actually more internal variables that the end-user doesn't need to set or configure. So, we should remove those from the control panel, leaving only the pct_target and n_trials variables. To do this, go back to Edit Program and click on each of the vars, and turn off the CTRL_PANEL flag for all but pct_target and n_trials. Then go back and marvel at the clean interface you've provided for your grateful user! The mouse-over tooltip even shows whatever comment you entered in the desc field for that variable. This can provide a quick but quite usable interface for many different programs.

## Calling from the Epoch Program

The final final step is to call the CPTAXGen program every epoch, so that we get a new random selection of trials every epoch (keeps the network from simply memorizing the particular sample we happen to have generated). To do this, we go to the LeabraEpoch program in the LeabraAll_Std subgroup of programs, and do Edit Program. Then drag the prog() guy from the Var/Fun toolbox just after the 3rd line of the prog_code, which starts the timer recording how long the epoch takes to process (it is a MethodCall, in blue). Then select CPTAXGen for the target. Note that this automatically brings up the input_data arg, and it even automatically fills this in with the input_data variable in the LeabraEpoch program -- if an arg has the same name as a variable in the calling program, it is used automatically (of course you can always change it if that isn't right).

There is one additional and ***very critical*** final step with the LeabraEpoch program. Go to the Program Ctrl tab, and observe the set of program vars available for you to set. The first one, called `data_loop_order` is set to `PERMUTED` by default -- this means that the trials (rows of the input data table) are presented in a shuffled random order (without replacment, so each trial only appears once). Clearly this is not going to be good for our *sequential* input data. So, change it to `SEQUENTIAL`, which will present the trials in sequential order. Given that we're doing the randomization within our program, this should be just fine. There is also another way of doing this that involves creating grouped trials (i.e., cue-probe), where you can randomize the order of the groups, but present the trials within the group in sequential order. The LeabraEpochGpData program available in the standard program library does this, but that is beyond the scope of this project.

## Running the Network

Now you're finally ready to run the network on this CPT-AX task! Go back to the LeabraTrain process, do Init and Run on it (initialize the weights) and see what happens!?

You should see that it will run and run and never fully learn the task (it will stop training if the error goes to zero). There is some chance that it might get there just by virtue of a lucky set of trials -- try hitting Run again -- it should not stay at zero, and will keep running.

You probably want to turn off the network and trial output data views at some point -- just click off the display button on their respective control panels under the Network_0 view tab.

It shouldn't come as that much of a surprise that the network doesn't fully learn the task -- this task requires working memory, and this network hasn't got any! In fact, it is quite surprising that it is able to learn as well as it can. Turns out it can learn to use weight changes as a kind of fairly unreliable form of working memory. Plus, the default target of AX is highly frequent, so it can get pretty far by focusing a lot on that.

In the next and final segment, we give this network some working memory, and see if that helps:

→ next PfcBg: Adding a Prefrontal Cortex, Basal Ganglia Working Memory System

# AXTut PfcBg

← previous CPTAX_Program **AX Tutorial**

**Contents**

## Adding a Prefrontal Cortex, Basal Ganglia Working Memory System

There are many different ways of giving a neural network some amount of working or active memory, to hold on to prior events. Perhaps the simplest is to add a "simple recurrent network" (SRN) context layer that holds on to the prior time step's hidden layer activations, and then feeds back into the hidden layer to provide context for the current inputs.

However, there are various limitations of this simple SRN memory, which can be overcome by having an active gating mechanism that determines when to hold onto information and when to forget it. One scientific theory is that the basal ganglia provide this function, by interacting with the prefrontal cortex, which is widely regarded as the brain area responsible for holding onto working memory. The specific implementation of this idea, called PBWM (prefrontal-cortex basal-ganglia working memory; O'Reilly & Frank, 2006, Neural Computation) is available through the Leabra wizard, and we'll use that.

First, to prepare the model for the PBWM components, we need to move the Input layer up to the same level as the output layer. For anatomically-inspired reasons, PBWM locates various brain-stem dopamine systems in the lower level of the model. To do this, click on the red arrow in the .T3Tab.Network_0 panel, and click on the virtical arrow poking through the green Input layer border, and drag it up to the level of the Output layer. The Output layer should move out of the way, and that is all you need to do, but if things don't look right, you can drag layers around with the horizontal arrows too.

Next, go to the .PanelTab.LeabraWizard_0, and select Network/PBWM. A dialog with several options and lots of information comes up. Turn off `out_gate`, and turn on `nolrn_pfc`. This makes the PFC working memory layer activated directly from the input layer, and not the hidden layer, and it makes it a direct copy of the input layer, instead of having it learn new representations. These are "hacks" that simplfy the model and make it easier to understand -- performance is generally the same without them. When you hit OK, you'll get a series of dialogs with information -- just keep hitting OK until it is done. You should see a rather more elaborate network now, with many more layers.

For complete details about these layers, see the [and Frank, 2006] paper (O'Reilly, R.C. & Frank, M.J. (2006). Making Working Memory Work: A Computational Model of Learning in the Frontal Cortex and Basal Ganglia. *Neural Computation, 18,* 283-328.) Here is a very brief overview:

- First, note that there are four separate **stripes** (groups of units) in the PFC and Matrix layers -- this was determined by the `n_stripes` parameter in the wizard. Each stripe can be independently updated, such that this system can remember up to 4 different things at the same time, each with a different "updating policy" of when memories are updated and maintained. The active maintenance of the memory is in PFC, and the updating signals (and updating policy more generally) come from the Matrix units (a subset of basal ganglia units).
- PV* and LV* and friends at the very bottom layer of the network: these represent the dopaminergic system, which provides reinforcement learning signals to train up the dynamic gating system in the basal ganglia. The PV layers represent primary values of reward (i.e., actual externally-delivered reward values), while the LV layers represent learned ("anticipated") values -- together, they account for Pavlovian conditioning phenomena and associated dopaminergic firing data.
- Matrix: this is the dynamic gating system representing the matrix units of the basal ganglia. Every even-index unit within a stripe represents "Go", while the odd-index units represent "NoGo." The Go units cause updating of the PFC, while the NoGo units cause the PFC to maintain its existing memory representation.
- SNrThal: represents the substantia nigra pars reticulata (SNr) and the associated area of the thalamus, which produce a competition among the Go/NoGo units within a given stripe. If there is more overall Go activity in a given stripe, then the associated SNrThal unit gets activated, and it drives updating in PFC.
- PFC: has 4 different stripes each of which has a localist one-to-one representation of the input units (due to the nolrn_pfc flag). Thus, you can look at these PFC representations and see directly what the network is maintaining.

### Setting the RewTarg Input

Before we can run the model, we need to do one extra bit of configuration. The PBWM model learns from rewards and punishments generated based on how it is performing the task. Only the reward values generated on the probe trials are relevant, however, so we need to tell the model when the relevant trials are. This is done using the RewTarg layer (in the bottom layer), which is a new input layer that was added by the wizard. When we set this unit activation to 1, then that tells the network that this is a trial when reward should be computed based on the difference between the network's output and the correct answer. Note that this is not the direct value of the reward itself, just the indicator of when reward should be computed.

The procedural steps for making this RewTarg work are mostly the same for any kind of change in the input data table structure (e.g., adding more input units), so these steps are generally useful:

- First, go to the .PanelTab.LeabraWizard_0 and

select Data/UpdateInputDataFmNet -- this will automatically reconfigure your StdInputData table to include the RewTarg input (and it will adjust it to any other changes you might make in your network -- a very useful function!).

- Next, we need to update the program that applies the input data to the network, so that it will appropriately apply the new RewTarg input to the network. This is the .programs.gp.LeabraAll_Std.ApplyInputs program in LeabraAll_Std subgroup.

In its objs section, there is an object called .programs.gp.LeabraAll_Std.ApplyInputs.LayerWriter_0, which provides the info for mapping input data the network layers. Hit AutoConfig on this object, and it will automatically update based on the new input data and network configuration.

- Now we need to modify our .programs.CPTAXGen program to set this RewTarg input value correctly. This requires several steps:
  - Update the unit names so we can refer to the rew targ input using an enum: click on the InitNamedUnits object in the init_code section of the program (under Edit Program tab) and hit the [[.programs.CPTAXGen.init_code [0].InitNamesTable()|InitNamesTable]] button -- this will update the UnitNames data table to match the updates in the input data table.
  - Go to .data.gp.InputData.UnitNames and enter the name "rew_targ" for the single RewTarg unit.
  - Go back to InitNamedUnits and do [[.programs.CPTAXGen.init_code[0].InitDynEnums()|InitDynEnums]] -- this will add a RewTarg DynEnum in the types section (it would also update the enums based on any other changes you might have made in the UnitNames table -- again a very useful function to remember)
  - Now we are finally ready to add the code to set the rew_targ input for the probe trial. Just drag a `set units lit` from the Network toolbox to end of the prog_code before the DoneWritingRowData guy, and set the enum_type to RewTarg, and the value should be R_rew_targ.

Finally, you can hit Init and Run on the LeabraTrain program to run your new network (select Yes to Initialize the weights).

## Increasing the Hidden Layer Size

It may or may not learn the task very quickly (depends on your random initial weights, etc). It turns out from playing with this model a bit that the initial 16 unit hidden layer is just a bit too small to handle all the new information being represented in the PFC layer. So, click on the Hidden layer's green border in the Network_0 3d view, and you should see the edit panel for the network in the middle panel. Locate the un_geom line, and change it to 5 x 5 (25 units) instead of 4 x 4. When you hit apply, the layer will change size in the display, but the extra units will not be filled in. You need to click back on the Network_0 tab in the middle panel, and hit the Build button at the bottom to rebuild the network based on the changes you made.

Now Init and Run on the LeabraTrain program again, select Yes to initialize the weights, and you should see the network learning within 10-20 epochs or so (again, toggle off the display button on the Network view control panel to speed things up -- same with the TrialOutputData Grid display).

## Displaying Unit Names

Once the network has learned, we can use the Step button on the Train program to see how it operates step by step (turn the network and trial grid log display's back on). By default, the step goes one settle at a time -- you can change this to LeabraTrial to get one trial at a time.

To better visualize what is happening, you can change the input, output, and PFC layers to display the name of the most active unit, rather than just the raw unit activations. This makes it just that much easier to figure out what the network is doing.

There are two steps to this. First, we need to get the unit name labels from the UnitNames data table into our network. Then, we need to configure the network display to show the labels instead of the units.

1. Go to our good friend the [[.programs.CPTAXGen.init_code[0]|InitNamedUnits]] object in the

init_code of the .programs.CPTAXGen program. There is a LabelNetwork button there, but if you press it, you'll get an error about not finding a network variable to apply the names to. To create this variable, you can just copy the network variable from the args section of any of the programs in the LeabraAll_Std subgroup, and put it in the args of the CPTAXGen program (do context menu copy on the network variable and then context menu paste on args, or you can actually open up the args section of a program in the left browser and drag the network directly into your CPTAXGen args in the middle browser -- that is a convenient way to do various copies). Then do [[.programs.CPTAXGen.init_code[0].LabelNetwork()|LabelNetwork]] again, and this time it should work.

1. In the .T3Tab.Network_0 view, select the red arrow and then click on the Input layer (green border) to select it, and then use the context menu (right mouse button or mac-command-mouse) to select `Disp Output Name`. Repeat this for the Output and PFC layers.

Now Step your LeabraTrain program, and you should see the names of the active units.

In the network that we trained (which you can load into Network_0 by clicking on that guy and doing Object/Load and selecting `ax_tutorial_cptax.net`), it is very clear that the middle two stripes update for an A or a C, while the first and third stripe update for B. No stripes update for any of the probe stimuli. This encoding of the cue but not the probe is just what you'd expect the network to learn.

It might be easier to see what the network is doing if you change the pct_target to .25 or something instead of .7 -- you don't have to spend so much time clicking through AX trials.

That is all we have for now. You might notice a project called `12ax4s.proj` in this same directory -- that is an even more complex version of the CPT-AX task involving an outer-loop of 1 or 2 stimuli that determines what the inner-loop target sequence is (AX or BY). These same mechanisms can learn that more difficult task, though it takes longer. It is described in detail in the O'Reilly & Frank 2006 paper referenced above.

**Using emergent**

# Concepts

**Contents**

**Using emergent**

## Overview

Key concepts for *emergent*, especially relative to PDP++. See also Changes from PDP++ for a more detailed list of such changes.

- **There are only 3 top-level object types: DataTable, Program, and Network** (plus associated 3d Graphical View objects). You should be familiar with what these do, and organize your thinking around them.
- **Always try the context menu on various objects** (right mouse button, or Ctrl-mouse on Mac) -- lots of good stuff is available there! There are so many different objects in a given simulation, each with different functions, that instead of trying to have one master menu, each object has its own special menu!
- **The gui is now browser-based**. Instead of popping up a million different edit dialogs, you interact by browsing and clicking on objects, with the edit dialog appearing in the central panel. The GUI views of objects (e.g., the NetView) is in the right-hand panel, sporting fancy, full, 3D graphics.
- **Use the mouse-over to get help**: just hold the mouse over various items to get helpful hints.

## Objects

### DataTables

DataTable replaces Environments and Logs from PDP++ v3.2, and also has many other new uses:

- There are sets of taDataXXX objects for doing XXX = "Gen" (generation), "Proc" (data processing/database ops), and "Anal" (data analysis) on data tables.
- Use data tables for organizing the structure of training of your network: create lists (of lists) of conditions or specifications of events.
- Record network data into data tables (using NetMonitor); then use DataProc or DataAnal routines to aggregate and analyze the data. Gone are the rigid "Stat aggregation" mechanisms from PDP++ v3.2.

### Programs

Programs replace Scripts, Processes and Stats from PDP++ v3.2. They provide a full programming language via the GUI. A Program generates a Css script, which is then executed to actually run the program. Many common operations can be done using Program elements through the GUI, but you can always write Css code directly using a UserScript. The GUI also allows you to lookup functions and variables on objects using a powerful GUI chooser with search abilities.

- Programs can call other Programs, and the standard "[[SchedProcess]" hierarchy of scheduling processes has been largely replicated with Programs. This makes it easy to see exactly what these Programs are doing, and *insert your own specialized functionality wherever you want* in the program flow. Gone are the rigid and mysterious "init, loop and final" slots.
- Programs are fully encapsulated and self-contained, and can (optionally) be passed arguments. Arg management is much improved over the s_args of PDP++ v3.2. Programs can contain their own functions, types, and variables.

If you want to implement some substantial new chunk of functionality, just create a new Program, then call it from one of the standard existing Programs at the appropriate place(s).

## Networks

Networks are not much changed from PDP++ v3.2, except that specs are now contained within a Network. This makes it fully encapsulated, such that you can drag and drop or copy and paste Networks from one project to another, and they will be fully functional because all their specs come along with them. Also, all the control parameters that used to be distributed throughout the Process objects (learning modes, cycles to settle, etc) are now all consolidated on the Network object.

## Views

Views of objects (NetView, GridTableView, GraphTableView) can be combined together in one integrated 3D display, or in separate frames, each with their own tabs. Each View has an associated control panel that shows up in the middle edit panel area -- use this to configure the View to your liking. You can find additional configuration options by clicking on the green frame of the viewed object. This pulls up an edit dialog of the underlying view object, where all the settable parameters are listed.

# Tips and Tricks

**Using emergent**

## Browser and other Core GUI Tools

**NUMBER 1 TIP:** Use the *context menu* on individual objects!! This is activated by the right mouse button (ctrl + mouse on Mac). It contains many useful actions.

Drag and Drop -- very very powerful!

Cut and Paste -- also handy

Edit Dialog -- things you can do with the edit dialog/panels

3d Graphical View -- how to navigate

Project Window Tips -- special functions of the main project window

Keyboard Shortcuts -- you can do a lot without leaving the keyboard!

## Wizard, SelectEdit etc

WizardTips -- its pretty obvious, but..

SelectEditTips -- this is a very powerful tool for organizing parameters and comparing values, etc.

## Core Objects: Programs, DataTables, Networks

ProgramTips -- a program automates operations, including training networks, generating input data for networks, and analyzing the performance of networks.

DataTableTips -- DataTables are used everywhere for containing rows and columns of data -- e.g., input data for training a network, output data (logs) of how the network performed, statistics analzing aspects of performance.

NetworkTips -- a quick short list of main things to do on a network.

# HowTos

These are quick pointers to various functionality that may not otherwise be obvious. Please add things you discover so others may benefit!

**Using emergent**

**Contents:** A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

## A

Activation Based Receptive Field -- how to create and use this useful statistic, which can determine selectivity of hidden units relative to input/output patterns (or even other hidden units).

Algorithm information: See Network Algorithm Guides for descriptions, pseudocode, and parameters for the various algorithms available in *emergent*, including Leabra.

Attaching a .proj file to a Bugzilla bug can be accomplished by first filing the bug report, then viewing the bug and clicking "Create new attachment." For the content type, select the "Manual" radio dial and enter `model/proj`. Click submit and you're done.

3d Animation of your model's behavior.

## B

"background run" or "batch run" -- how to run a project without any gui (nogui run) -- requires a startup program or startup script that controls the entire run.

## C

Column widths in datatable: adjust by dragging on right edge of header; save using View/Save View (saves all view parameters)

Compare variables within the Network View -- use one of the Snapshot functions on the Network and turn on the Snap Bord display in the network view.

CSS console: The console typically floats below the main Project window, or it can be docked into the root window via a Preferences selection. The console is where warning, info, and error messages are sent, and is particularly important when running Programs.

copy data from one DataTable to another: CopyCell method, or sel rows (DataSelectRowsProg), sel cols (DataSelectColsProg) in data proc toolbox.

## D

Document editing formatting (wiki text and ta: urls)

## E

## F

## G

if your Graph variable On button is getting turned OFF mysteriously: often this is due to removing all the columns of a data table and then rebuilding that table with new columns, which often are the same ones it had before. The problem is that the graph view structure gets rebuilt when things get reset, so it turns off the display because there is no column there anymore. The solution is to prevent the view rebuild during the transitional state: bracket the whole operation of removing and rebuilding in StructUpdate calls.

Grid View of Weights -- make a "receptive field" plot of the weights for a layer of units.

Grouping: all the grouped items in a group spec must be listed at the start together. Remember that GROUP for the agg op (aggregation operator) means that all the rows for each different value that shows up in the given column of the source data table will be grouped together in the resulting output data table.

## H

## I

Image Data -- how to deal with images in data tables

Importing Data -- how to import data into a data table e.g., for use in training a network.

InputData -- everything you need to know about how input patterns from the input data table are applied to the network.

## J

## K

Keyboard shortcuts -- become more efficient at using *emergent* by studying these

## L

Lesion - remove individual cell or a group of cells (specified by percentage of cell numbers in a layer) and monitor the performance degradation of the network

Local variables in Programs -- you aren't restricted to the global vars and args -- the `loc vars` in the `Var/Fun` toolbox can be put anywhere in a program (especially useful for Functions) to define new variables.

Logging Output layer in OutputData can be done by right clicking on TrialMonitorTest-objs-trial_netmon and selecting Layer from the drop down menu. FYI, going to the network layer and clicking Monitor Var will not make the layer monitored\logged in the OutputData.

## M

Math on Data -- how to perform math operations on data in a datatable (very useful for data analysis)

Matrix from DataTable -- how to access/modify Matrix objects within DataTable objects

Monitor Data -- how to record or monitor data from the network *or any other kind of object* when it is being trained -- also look here if you get config errors in NetMonitor, TrialMonitor, or EpochMonitor

Movies -- how to make a movie of your network in action

Multiple Input Data Tables -- how to train or test a network on multiple different input data tables.

## N

nogui run -- how to run a project without any gui in the background -- requires a startup program or startup script that controls the entire run.

## O

Oscillations

## P

Param Search -- support for automatic parameter searching.

PDP++ Project Conversion -- short version: you must *remove units* from PDP++ Networks before loading them into *emergent*, then hit Convert button at bottom of Project editor panel.

Plugins -- how to make fast C++ code compiled extensions to the software that build on the existing codebase and are easy and quick to build..

Programs replace Scripts, Processes and Stats from PDP++. See Programs for general information on creating, initializing, and running your programs. Program Tips provide solutions to common programming problems.

Projection selecting/viewing in Network View -- increase the Prjn Width from default of .002 to .005 or even .01 -- this makes the lines much easier to see.

## Q

## R

## S

Save View -- save the overall "view" configuration of your project window.

SSH -- tips for using *emergent* over SSH with X-forwarding.

Signals -- "kill" signals recognized by the program (when running in the nogui background mode and otherwise unreachable) -- basically just USR1 or USR2 or ALARM, all of which will cause a project recover file to be saved, which can then be loaded to check out what is going on with the project.

Small screen -- tips for making the most of limited screen real estate.

Startup program -- how to make an automated program that will auto-run when the project is loaded. This includes info on how to set command-line arguments that can be passed to the project to set various parameters -- very convenient for exploring parameter space. The GUI has a 'run on STARTUP' checkbox available for each program in the ‘program control’ panel. Most projects have a “startup” program that runs automatically when *emergent* is run with the -nogui switch from the command line, but are disabled when the gui is active.
Stop program code from running. In some situations pressing Stop/Abort will not stop a program from running, even if it is generating warning/error messages. In order to stop it you will need to add a Stop/Step program element at some point in your program.

## T

Making Test Programs for testing your network during or after training.
Thread Params -- how to configure parallel threading on multi-CPU machines.

## U

URL (uniform resource locator) syntax for Documents

## V

Virtual Environment construction and manipulation.

## W

Weights, accessing in Programs / Css: `network.layers.LayerName.units[un_no].recv [prjn_no].Cn(cn_no)` -- the sending unit is Un(cn_no). The implementation has been highly optimized so these functions (Cn, Un) are the way to access. See Unit for more info.
Wiki style syntax for Document editing formatting

***emergent* won't start**. One possible reason *emergent* won't start is corrupted preferences. On Mac OSX, this is indicated by the *emergent* icon bouncing in the dock several times and then stopping without actually starting. There are two possible files that might be the problem, one is 'options' and the other is 'root.' Start with 'options' first. Delete the options file and then restart *emergent*. Emergent should create a new 'options' file. If that doesn't work try 'root.'

On Mac OSX, there are at least two ways to do this. One way is to use the Terminal program to go to the directory ~/.*emergent*, where ~ indicates that this is in your user directory. To do this type 'cd ~/.*emergent* ***to change to the correct directory. Then remove the 'options' file. The other procedure is to use the Go To Folder command in the Go menu in the Finder. Enter '~/.*****emergent** in the Folder name box. You can then move the 'options' or 'root' file to the trash. Note that Finder hides these hidden folders that start with a period, so you will either need to use the Go To Folder command or else use the Terminal program.

The procedure should be comparable for Linux and Windows, but I don't know for sure.

## Y

## Z

About Emergent Powered by MediaWiki Designed by Paul Gu

# Tutorials

**Contents**

**Using emergent**

## Tutorials for Emergent

**IMPORTANT NOTE:** many of these tutorials do not yet exist unfortunately -- that is not a bug in your browser. Contributions welcome!

### Video tutorials

- ▶ How to build your own network --*Needs to be updated!*
- ▶ How to run *emergent* from the command line
- ▶ Visualization features --*Needs to be updated!*
- ▶ Test Screencast --*Sandbox for uploading tutorials - for testing only!*

### Core Stuff: Networks, DataTables and Programs

Build Your Own Network -- constructing a basic 3-layer network using the wizard, and training it on a pre-defined (small) set of training patterns.

AX Tutorial -- construct a simple model of an actual psychological task (the CPT-AX task), from start to finish. This one tutorial touches on all major aspects of the system. It is available as an interactive *emergent* project in `demo/leabra/ax_tutorial.proj` (just run *emergent*, open the project and start following the instructions that appear), and in the wiki.

Data Processing Tutorial -- data processing tutorial for DataTables -- covers the basic operations that you can do with data tables, which are central to much of *emergent*

Backpropagation Tutorial (**incomplete!**)-- construct a feed-forward backpropagation (backward propagation of errors, Bp in *emergent* shorthand) neural net model on a sample data set. This tutorial covers the following topics: an outline of a Bp network and its learning algorithms, setting up a Bp project, constructing a multi-layer Bp network with a single output variable, setting up data training and testing data tables, populating data tables by importing data from a file, setting up data tables and variables to contain train and test output data, setting up two sets of control programs (one to train the network and one to test additional data on the trained network)

Network Data Tutorial -- How to gather data on various aspects of network function (success, activity levels, etc) and process these appropriately for different conditions. Expands on the basic data gathering in the AX tutorial, but starts from the basics.

### Multi-Media Input/Outputs

ImageProcTutorial -- image processing tutorial -- how to configure a network and Programs to operate on bitmap images.

AudioProcTutorial -- audio processing tutorial -- how to configure a network and Programs to operate on audio inputs.

### Complex Network Structures

SRNTutorial -- how to construct a simple recurrent network.

TDTutorial -- constructing a Temporal Differences model in Leabra.

PVLVTutorial -- constructing a PVLV (Primary Value, Learned Value Pavlovian conditioning) model in Leabra.

PBWMTutorial -- constructing a Prefrontal-cortex Basal-ganglia Working Memory (PBWM) model in Leabra.

# Demos

Using emergent

There are a number of demo projects included with the software, which are typically installed in `/usr/local/share/Emergent/demo` (Linux, Mac) and live in the Emergent\demo application directory under Windows.

These can be useful for providing working examples of various types of models and procedures. Each contains its own more detailed documentation that should auto-open when you open the project, so this is just a brief listing of what is available.

**Contents**

## demo/bp (Backpropagation)

**bp_xor.proj** -- classic XOR problem -- also includes separate testing and training programs, and a grid view display of the network's performance.

## demo/cs (Constraint Satisfaction)

**424_cs.proj** -- classic 4-2-4 auto-encoder using stochastic units, learning via the Boltzmann machine learning rule

**424_cs_det.proj** -- deterministic version of the above, learning using CHL (deterministic Boltzmann machine learning rule)

**cs_ra.proj** -- random associator (deterministic) -- learns to associate random bit patterns and is a good test of general learning abilities.

## demo/data_proc (DataProcessing)

**ca_eq_sim.pro**j -- calcium equation simulation -- shows how to use a DataTable to graph equations (e.g., like gnuplot or similar), using a DataCalcLoop object and various other important data processing techniques.

**canvas_drawing.proj** -- example of using the taCanvas object for using basic graphic rendering operations on an image

**data_calc_loop.proj** -- focused simple example of the powerful DataCalcLoop program element.

**epoch_log_analysis.proj** -- program that performs various standard analysis operations on epoch data logs generated as a network is trained -- tells you how many epochs it too to reach various criteria, etc (associated shell script makes a simple executable system for analyzing your data!)

**matrix_demo.proj** -- how to use Matrix objects in Programs -- creates matrices from scratch, sets values, runs Math functions, etc.

## demo/leabra (Leabra)

**12ax4s.proj** -- the 1-2 CPT-AX task using the PBWM model of PFC and BG in Leabra -- see AX_Tutorial for more info on related models.

**ax_tutorial*.proj** -- see AX_Tutorial -- does the simpler CPT-AX task

**model_and_task.proj** -- an early version of the model from chapter 6 of the CECN book (see link below), demonstrating combined model (hebbian) and task (error-driven) learning.

**param_search.proj** -- simple demo of Param Search using SelectEdit infrastructure.

**simple_vis_search.proj** -- a very simple visual search model.

**scalar_val_test.proj** -- a test of the ScalarValLayerSpec, trained with different probabilities of a 0 or 1, showing that they accurately represent these probabilities as a graded scalar value.

## demo/network_misc

This has misc projects for exploring various general neural network functionality:

**projection_sampler.proj** -- demonstrates how to use various ProjectionSpec's to connect units according to various patterns.

**std_wizards.proj** -- standard wizards constructed in Programs using taGui programmable gui -- good starting point for making your own custom wizards.

## demo/so Self Organizing

**som.proj** -- self-organizing map model -- learns topographic representations of input data in a self-organizing manner

## demo/virt_env Virtual Environment

**ve_arm.proj** -- a simple simulated robot that can move a 2 jointed arm to reach a target, all under control of the network.

## Computational Explorations in Cognitive Neuroscience Projects

http://grey.colorado.edu/CompCogNeuro/index.php/CECN1_Projects -- large collection of well-documented projects built in emergent, exploring a wide range of cognitive phenomena.

Back to Using emergent

# User Guide

**Using emergent**

This user guide is just a master index into a set of individual pages describing all of the key elements of the software. The software is object oriented, and this documentation is organized around the objects involved. See the guide to using *emergent* for other ways of accessing this information.

## Major Objects

These are the main entities you will interact with in constructing a simulation:

- Project -- overall container for everything associated with a given simulation, including the Project window graphical interface.
- DataTable -- multi-purpose container of all manner of data, including input patterns to present to a network, output recorded from the network, analysis, statistics, etc.
  - Grid View -- graphical view of data using text and colored squares in rows and columns
  - Graph View -- graphical view of data as lines or bars
- Network -- the neural network itself
  - Network View -- 3D graphical display of the network
  - Specs -- parameters for controlling network function
  - Layer -- contains Unit objects
  - Projection -- specifies connectivity between layers]
  - NetMonitor and Monitor Data -- record data from the network to a DataTable
- Program -- controls all aspects of the simulation
- Containers -- List, Group and Matrix objects that are used everywhere to contain and manage other objects -- they form the skeleton of the project. See css list comprehension and Matrix css for powerful syntax within Programs to access these containers. in the Gui, they are accessed through the left browser and other similar browsers.

## Other Objects and Functions

- Preferences -- configure things just the way you want.
- SelectEdit -- create a custom control panel for your simulation including all the main parameters and functions. Also can show a "diff" comparison of different objects.
- Docs -- create project-specific documentation so others can take advantage of all your hard work!
- DataProc -- specialized data processing functions, including data base, data analysis & statistics, data generation, image processing, and basic math operations.
- Keyboard shortcuts -- become more efficient at using *emergent* by studying these
- Virtual Environment -- uses ODE library http://ode.org/ to provide realistic physics simulation of a virtual world with which your network can interact.
- Nogui run -- running your model in the background without the GUI active (e.g., for clusters or other "batch jobs")
- Command line switches -- control the program from the command line
- Environment variables -- environment variables that change the behavior of *emergent*
- Network Algorithm Guide -- detailed info on the neural network algorithms
- CSS -- the "C Super Script" scripting language that runs Programs and can be used directly at the Console or in user scripts.
- UserData -- find out how to add your own persistent named data items to *emergent* objects
- Server Protocol -- describes the TCP/IP interface for remotely running *emergent* programs, setting/fetching variables, and sending/receiving data
- Plugins -- easily extend the software with new hard-coded C++ features (learning algorithms, data table functions, anything)

# Network Algorithm Guide

**Using emergent**

The system supports the following neural network algorithms -- follow the links for more details. To create a project that uses a given algorithm, you typically create a project of the given type (e.g., a BpProject for Backprop). This sets all the appropriate defaults (no more .def defaults files as used in PDP++). However, you can easily just mix and match within the same project -- all of the relevant default information is contained within the type-specific Network object (e.g., BpNetwork).

Backpropagation (Bp) -- feedforward and recurrent networks learning from backpropagated error signals.

Leabra -- Local, error-driven and associative, biologically realistic algorithm -- combines Bp-like error driven learning with self organizing learning plus inhibitory competition and constraint satisfaction processing in a biologically plausible manner. Used to simulate many different cognitive neuroscience phenomena. See Leabra Params for hints on parameter setting.

Constraint Satisfaction (Cs) -- symmetric, bidirectionally connected networks that settle into a state that maximizes the various internal and external constraints (e.g., Boltzmann machines, Hopfield networks).

Self Organizing (So) -- learn without an explicit teacher, based on Hebbian learning, includes Kohonen networks and various flavors of Competitive Learning.

# Backpropagation

**Contents**

## Introduction

Backpropagation is perhaps the most commonly used neural network learning algorithm. Several different "flavors" of backpropagation have been developed over the years, several of which have been implemented in the software, including the use of different error functions such as cross-entropy, and recurrent backprop, from the simple recurrent network to the Almeida-Pineda algorithm up to the real-time continuous recurrent backprop. The implementation allows the user to extend the unit types to use different activation and error functions in a straightforward manner.

Note that the simple recurrent networks (SRN, a.k.a. Elman networks) are described in the feedforward backprop section, as they are more like feedforward networks than the fully recurrent ones.

The basic structure of the backpropagation algorithm consists of two phases, an activation propagation phase, and an error backpropagation phase. In the simplest version of Bp, both of these phases are strictly feed-forward and feed-back, and are computed sequentially layer-by-layer. Thus, the implementation assumes that the layers are organized sequentially in the order that activation flows.

In the recurrent versions, both the activation and the error propagation are computed in two steps so that each unit is effectively being updated simultaneously with the other units. This is done in the activation phase by first computing the net input to each unit based on the other units current activation values, and then updating the activation values based on this net input. Similarly, in the error phase, first the derivative of the error with respect to the activation (dEdA) of each unit is computed based on current dEdNet values, and then the dEdNet values are updated based on the new dEdNet.

## Feedforward Bp Reference

The classes defined in the basic feedforward Bp implementation include:

Reference info for type **BpConSpec**: Wiki | Emergent Help Browser

,

Reference info for type **BpCon**: Wiki | Emergent Help Browser

Reference info for type **BpUnit**: Wiki | Emergent Help Browser

,

Reference info for type **BpUnitSpec**: Wiki | Emergent Help Browser

Reference info for type **BpLayer**: Wiki | Emergent Help Browser

Reference info for type **BpNetwork**: Wiki | Emergent Help Browser

Bias weights are implemented by adding a BpCon object to the BpUnit directly, and not by trying to allocate some kind of self projection or some other scheme like that. In addition, the BpUnitSpec has a pointer to a BpConSpec to control the updating etc of the bias weight. Thus, while some code was written to support the special bias weights on units, it amounts to simply calling the appropriate function on the BpConSpec.

## Variations on the Standard

Reference info for type **LinearBpUnitSpec**: Wiki | Emergent Help Browser

implements a linear activation function

Reference info for type **ThreshLinBpUnitSpec**: Wiki | Emergent Help Browser

implements a threshold linear activation function with the threshold set by the parameter @code{threshold}. Activation is zero when net is below threshold, net-threshold above that.

Reference info for type **NoisyBpUnitSpec**: Wiki | Emergent Help Browser

adds noise to the activations of units. The noise is specified by the noise member.

Reference info for type **StochasticBpUnitSpec**: Wiki | Emergent Help Browser

computes a binary activation, with the probability of being active a sigmoidal function of the net input (e.g., like a Boltzmann Machine unit).

Reference info for type **RBFBpUnitSpec**: Wiki | Emergent Help Browser

computes activation as a Gaussian function of the distance between the weights and the activations. The variance of the Gaussian is spherical (the same for all weights), and is given by the parameter var.

Reference info for type **BumpBpUnitSpec**: Wiki | Emergent Help Browser

computes activation as a Gaussian function of the standard dot-product net input (not the distance, as in the RBF). The mean of the effectively uni-dimensional Gaussian is specified by the mean parameter, with a standard deviation of std_dev.

Reference info for type **ExpBpUnitSpec**: Wiki | Emergent Help Browser

computes activation as an exponential function of the net input (e^net). This is useful for implementing SoftMax units, among other things.

Reference info for type **SoftMaxBpUnitSpec**: Wiki | Emergent Help Browser

takes one-to-one input from a corresponding exponential unit, and another input from a LinearBpUnitSpec unit that computes the sum over all the exponential units, and computes the division between these two. This results in a SoftMax unit. Note that the LinearBpUnitSpec must have fixed weights all of value 1, and that the SoftMaxUnit's must have the one-to-one projection from exp units first, followed by the projection from the sum units. See `demo/bp_misc/bp_softmax.proj` for a demonstration of how to configure a SoftMax network.

Reference info for type **HebbBpConSpec**: Wiki | Emergent Help Browser

computes very simple Hebbian learning instead of

backpropagation. It is useful for making comparisons between delta-rule and Hebbian leanring. The rule is simply `dwt = ru->act * su->act`, where `ru->act` is the target value if present.

Reference info for type **ErrScaleBpConSpec**: Wiki | Emergent Help Browser

scales the error sent back to the sending units by

the factor @code{err_scale}. This can be used in cases where there are multiple output layers, some of which are not supposed to influence learning in the hidden layer, for example.

Reference info for type **DeltaBarDeltaBpConSpec**: Wiki | Emergent Help Browser

implements the delta-bar-delta learning rate

adaptation scheme (Jacobs, 1988). It should only be used in batch mode weight updating. The connection type must be

Reference info for type **DeltaBarDeltaBpCon**: Wiki | Emergent Help Browser

, which contains a connection-wise learning rate

parameter. This learning rate is additively incremented by lrate_incr when the sign of the current and previous weight changes are in agreement, and decrements it multiplicatively by lrate_decr when they are not. The demo project `demo/bp_misc/bp_ft_dbd.proj` provides an example of how to set up delta-bar-delta learning.

## Simple Recurrent Networks (SRN's)

Simple recurrent networks (SRN) (Elman, 1988) involve the use of a special set of context units which copy their values from the hidden units, and from which the hidden units receive inputs. Thus, it provides a simple form of recurrence that can be used to train networks to perform sequential tasks over time. The

Reference info for type **BpWizard**: Wiki | Emergent Help Browser

has a `Network/SRNContext` function that will

automatically build an SRN context layer as described below.

The implementation of SRN's uses a special version of the BpUnitSpec called the

Reference info for type **BpContextSpec**: Wiki | Emergent Help Browser

. This spec overloads

the activation function to simply copy from a corresponding hidden unit. The correspondence between hidden and context units is established by creating a single one-to-one projection into the context units from the hidden units. The context units look for the sending unit on the other side of their first connection in their first connection group for the activation to copy. This kind of connection should be created with a

Reference info for type **OneToOnePrjnSpec**: Wiki | Emergent Help Browser

.

**Important:** The context units should be in a layer that *follows* the hidden units they copy from. This is because the context units should provide input to the hidden units before copying their activation values. This means that the hidden units should update themselves first.

The context units do not have to simply copy the activations directly from the hidden units. Instead, they can perform a time-averaging of information through the use of an updating equation as described below. The parameters of the context spec are as follows:

The demo project <file>demo/bp_srn/srn_fsa.proj</file> is an example of a SRN network that uses the sequence processes.

## Recurrent Backprop--NYI!

**NOTE: Recurrent backpropagation (RBp) is not currently supported (as of *emergent* 5.0.2)**

Check the Comparison of Neural Network Simulators for existing simulators that support RBp.

Recurrent backpropagation (RBp) extends the basic functionality of feed-forward backprop to networks with recurrently interconnected units, which can exhibit interesting dynamical properties as activation propagates through the network over time.

The recurrent backprop implementation (RBp) defines a new set of types that are derived from the corresponding Bp versions:

- Reference info for type **RBpConSpec**: Wiki | Emergent Help Browser
- Reference info for type **RBpUnit**: Wiki | Emergent Help Browser
- Reference info for type **RBpUnitSpec**: Wiki | Emergent Help Browser

Note that RBp uses the same Connection type as Bp. In addition, support for the Almeida-Pineda algorithm is made possible by special ApBp Programs.

There are a couple of general features of the version of recurrent backprop that the user should be aware of. First of all, the model used is that of a discrete approximation to a continuous dynamic system, which is defined by the sigmoidal activation of the net input to the units. The granularity of the discrete approximation is controlled by the dt parameter, which should be in the range between 0 and 1, with smaller values corresponding to a finer, closer to continuous approximation. Thus, the behavior of the network should be roughly similar for different dt values, with the main effect of dt being to make updating smoother or rougher.

Also, there are two ways in which the units can settle, one involves making incremental changes to the activation values of units, and the other involves making incremental changes to the net inputs. The latter is generally preferred since it allows networks with large weights to update activations quickly compared to activation-based updates, which have a strict ceiling on the update rate since the maximum activation value is 1, while the maximum net input value is unbounded.

As in standard backpropagation, recurrent backprop operates in two phases: activation propagation and error backpropagation. The difference in recurrent backprop is that both of these phases extend over time. Thus, the network is run for some number of activation update cycles, during which a record of the activation states is kept by each unit, and then a backpropagation is performed that goes all the way back in time through the record of these activation states. The backpropagation happens between the receiving units at time t and the sending units at the previous time step, time t-1. Another way of thinking about this process is to unfold the network in time, which would result in a large network with a new set of layers for each time step, but with the same set of weights used repeatedly for each time step unfolding. Doing this, it is clear that the sending units are in the previous time step relative to the receiving units.

The exact length of the activation propagation phase and the timing and frequency of the backpropagation phases can be controlled in different ways that are appropriate for different kinds of tasks. In cases where there is a clearly-defined notion of a set of distinct temporal sequences, one can propagate activation all the way through each sequence, and then backpropagate at the end of the sequence. This is the default mode of operation for the processes.

There are other kinds of environments where there is no clear boundary between one sequence and the next. This is known as "real time" mode, and it works by periodically performing a backpropagation operation after some number of activation updates have been performed. Thus, there is a notion of a "time window" over which the network will be sensitive to temporal contingencies through the weight updates driven by a single backpropagation operation. In addition, these backpropagations can occur with a period that is less than the length of the time window, so that there is some overlap in the events covered by successive backpropagation operations. This can enable longer-range temporal contingencies to be bootstrapped from a series of overlapping backpropagations, each with a smaller time window.

There is a simpler variation of a recurrent backpropagation algorithm that was invented by Almeida and Pineda, and is named after them. In this algorithm, the activation updating phase proceeds iteratively until the maximum change between the previous and the current activation values over all units is below some criterion. Thus, the network settles into a stable attractor state. Then, the backpropagation phase is performed repeatedly until it too settles on a stable set of error derivative terms (i.e., the maximum difference between the derivative of the error for each unit and the previously computed such derivative is below some threshold). These asymptotic error derivatives are then used to update the weights. Note that the backpropagation operates repeatedly on the asymptotic or stable activation values computed during the first stage of settling, and not on the trajectory of these activation states as in the "standard" version of RBp. The Almeida-Pineda form of the algorithm is enabled by using the APBp Programs, which compute the two phases of settling over cycles of either activation propagation or error backpropagation.

# Leabra

**Contents**

## Introduction

Leabra stands for *Local, Error-driven and Associative, Biologically Realistic Algorithm*, and it implements a balance between Hebbian and error-driven learning on top of a biologically-based point-neuron activation function with inhibitory competition dynamics (either via inhibitory interneurons or a fast k-Winners-Take-All approximation thereof). Extensive documentation is available from the book: *Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain*, O'Reilly and Munakata, 2000, Cambridge, MA: MIT Press. For more information, see the website: Computational Explorations..

Hebbian learning is performed using conditional principal components analysis (CPCA) algorithm with correction factor for sparse expected activity levels.

Error driven learning is performed using GeneRec, which is a generalization of the Recirculation algorithm, and approximates Almeida-Pineda recurrent backprop. The symmetric, midpoint version of GeneRec is used, which is equivalent to the contrastive Hebbian learning algorithm (CHL). See O'Reilly (1996; Neural Computation) for more details.

The activation function is a point-neuron approximation with both discrete spiking and continuous rate-code output.

Layer or unit-group level inhibition can be computed directly using a k-winners-take-all (KWTA) function, producing sparse distributed representations, or via inihibitory interneurons.

The net input is computed as an average, not a sum, over connections, based on normalized, sigmoidaly transformed weight values, which are subject to scaling on a connection-group level to alter relative contributions. Automatic scaling is performed to compensate for differences in expected activity level in the different projections. See Leabra Netin Scaling for details.

Weights are subject to a contrast enhancement function, which compensates for the soft (exponential) weight bounding that keeps weights within the normalized 0-1 range. Contrast enhancement is important for enhancing the selectivity of self-organizing learning, and generally results in faster learning with better overall results. Learning operates on the underlying internal linear weight value, which is computed from the nonlinear (sigmoidal) weight value prior to making weight changes, and is then converted back. The linear weight is always stored as a negative value, so that shared weights or multiple weight updates do not try to linearize the already-linear value. The learning rules have been updated to assume that wt is negative (and linear).

There are various extensions to the algorithm that implement things like reinforcement learning (temporal differences), simple recurrent network (SRN) context layers, and combinations thereof (including an experimental versions of a complex temporal learning mechanism based on the prefrontal cortex and basal ganglia). Other extensions include a variety of options for the activation and inhibition functions, self regulation (accommodation and hysteresis, and activity regulation for preventing overactive and underactive units), synaptic depression, and various optional learning mechanisms and means of adapting parameters. These features are off by default but do appear in some of the edit dialogs --- any change from default parameters should be evident in the edit dialogs.

## Overview of the Leabra Algorithm

The pseudocode for Leabra is given here, showing exactly how the pieces of the algorithm described in more detail in the subsequent sections fit together.

```
Iterate over minus and plus phases of settling for each event.
  o At start of settling, for all units:
    - Initialize all state variables (activation, v_m, etc)
    - Apply external patterns (clamp input in minus, input & output in
      plus).
    - Compute net input scaling terms (constants, computed
      here so network can be dynamically altered).
    - Optimization: compute net input once from all static activations
      (e.g., hard-clamped external inputs).
  o During each cycle of settling, for all non-clamped units:
    - Compute excitatory netinput (g_e(t), aka eta_j or net)
       -- sender-based optimization by ignoring inactives.
    - Compute kWTA inhibition for each layer, based on g_i^Q:
      * Sort units into two groups based on g_i^Q: top k and
        remaining k+1 -> n.
      * If basic, find k and k+1th highest
        If avg-based, compute avg of 1 -> k & k+1 -> n.
      * Set inhibitory conductance g_i from g^Q_k and g^Q_k+1
    - Compute point-neuron activation combining excitatory input and
      inhibition
  o After settling, for all units, record final settling activations
    as either minus or plus phase (y^-_j or y^+_j).
After both phases update the weights (based on linear current
    weight values), for all connections:
  o Compute error-driven weight changes with soft weight bounding
  o Compute Hebbian weight changes from plus-phase activations
  o Compute net weight change as weighted sum of error-driven and Hebbian
  o Increment the weights according to net weight change.
```

## Point Neuron Activation Function

```
Default parameter values:

Parameter | Value | Parameter | Value
-------------------------------------------
E_l       | 0.15  | gbar_l     | 0.10
E_i       | 0.15  | gbar_i     | 1.0
E_e       | 1.00  | gbar_e     | 1.0
V_rest    | 0.15  | Theta      | 0.25
tau       | .02   | gamma      | 600
k_hebb    | .02   | epsilon    | .01
```

Leabra uses a *point neuron* activation function that models the electrophysiological properties of real neurons, while simplifying their geometry to a single point. This function is nearly as simple computationally as the standard sigmoidal activation function, but the more biologically-based implementation makes it considerably easier to model inhibitory competition, as described below. Further, using this function enables cognitive models to be more easily related to more physiologically detailed simulations, thereby facilitating bridge-building between biology and cognition.

The membrane potential V_m is updated as a function of ionic conductances g with reversal (driving) potentials E as follows:

$$\Delta V_m(t) = \tau \sum_c g_c(t) \overline{g_c} (E_c - V_m(t))$$

with 3 channels (c) corresponding to: e excitatory input; l leak current; and i inhibitory input. Following electrophysiological convention, the overall conductance is decomposed into a time-varying component g_c(t) computed as a function of the dynamic state of the network, and a constant gbar_c that controls the relative influence of the different conductances. The equilibrium potential can be written in a simplified form by setting the excitatory driving potential (E_e) to 1 and the leak and inhibitory driving potentials (E_l and E_i) of 0:

$$V_m^\infty = \frac{g_e \overline{g_e}}{g_e \overline{g_e} + g_l \overline{g_l} + g_i \overline{g_i}}$$

which shows that the neuron is computing a balance between excitation and the opposing forces of leak and inhibition. This equilibrium form of the equation can be understood in terms of a Bayesian decision making framework @cite{(O'Reilly & Munakata, 2000)}.

The excitatory net input/conductance g_e(t) or eta_j is computed as the proportion of open excitatory channels as a function of sending activations times the weight values:

$$\eta_j = g_e(t) = \langle x_i w_{ij} \rangle = \frac{1}{n}\sum_i x_i w_{ij}$$

See Leabra Netin Scaling for details on rescaling of these values across projections to compensate for activity levels etc.

The inhibitory conductance is computed via the kWTA function described in the next section, and leak is a constant.

Activation communicated to other cells (y_j) is a thresholded (Theta) sigmoidal function of the membrane potential with gain parameter gamma:

$$y_j(t) = \frac{1}{\left(1 + \frac{1}{\gamma[V_m(t) - \Theta]_+}\right)}$$

where [x]_+ is a threshold function that returns 0 if x<0 and x if X>0. Note that if it returns 0, we assume y_j(t) = 0, to avoid dividing by 0. As it is, this function has a very sharp threshold, which interferes with graded learning learning mechanisms (e.g., gradient descent). To produce a less discontinuous deterministic function with a softer threshold, the function is convolved with a Gaussian noise kernel (\mu=0, \sigma=.005), which reflects the intrinsic processing noise of biological neurons:

$$y_j^*(x) = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{-z^2/(2\sigma^2)} y_j(z - x) dz$$

where x represents the [V_m(t) - \Theta]_+ value, and y^*_j(x) is the noise-convolved activation for that value. In the simulation, this function is implemented using a numerical lookup table.

## k-Winners-Take-All Inhibition

Leabra uses a kWTA (k-Winners-Take-All) function to achieve inhibitory competition among units within a layer (area). The kWTA function computes a uniform level of inhibitory current for all units in the layer, such that the k+1th most excited unit within a layer is generally below its firing threshold, while the kth is typically above threshold. Activation dynamics similar to those produced by the kWTA function have been shown to result from simulated inhibitory interneurons that project both feedforward and feedback inhibition (OReilly & Munakata, 2000). Thus, although the kWTA function is somewhat biologically implausible in its implementation (e.g., requiring global information about activation states and using sorting mechanisms), it provides a computationally effective approximation to biologically plausible inhibitory dynamics.

kWTA is computed via a uniform level of inhibitory current for all units in the layer as follows:

$$g_i = g^{\Theta}_{k+1} + q(g^{\Theta}_k - g^{\Theta}_{k+1})$$

where 0<q<1 (.25 default used here) is a parameter for setting the inhibition between the upper bound of g^Theta_k and the lower bound of g^Theta_k+1. These boundary inhibition values are computed as a function of the level of inhibition necessary to keep a unit right at threshold:

$$g_i^{\Theta} = \frac{g_e^* \bar{g}_e (E_e - \Theta) + g_l \bar{g}_l (E_l - \Theta)}{\Theta - E_i}$$

where g^*_e is the excitatory net input without the bias weight contribution --- this allows the bias weights to override the kWTA constraint.

In the basic version of the kWTA function, which is relatively rigid about the kWTA constraint and is therefore used for output layers, g^Theta_k and g^Theta_k+1 are set to the threshold inhibition value for the kth and k+1th most excited units, respectively. Thus, the inhibition is placed exactly to allow k units to be above threshold, and the remainder below threshold. For this version, the q parameter is almost always .25, allowing the kth unit to be sufficiently above the inhibitory threshold.

In the average-based kWTA version, g^Theta_k is the average g_i^Theta value for the top k most excited units, and g^Theta_k+1 is the average of g_i^Theta for the remaining n-k units. This version allows for more flexibility in the actual number of units active depending on the nature of the activation distribution in the layer and the value of the q parameter (which is typically .6), and is therefore used for hidden layers.

## Hebbian and Error-Driven Learning

For learning, Leabra uses a combination of error-driven and Hebbian learning. The error-driven component is the symmetric midpoint version of the GeneRec algorithm @cite{(O'Reilly, 1996)}, which is functionally equivalent to the deterministic Boltzmann machine and contrastive Hebbian learning (CHL). The network settles in two phases, an expectation (minus) phase where the network's actual output is produced, and an outcome (plus) phase where the target output is experienced, and then computes a simple difference of a pre and postsynaptic activation product across these two phases. For Hebbian learning, Leabra uses essentially the same learning rule used in competitive learning or mixtures-of-Gaussians which can be seen as a variant of the Oja normalization @cite{(Oja, 1982)}. The error-driven and Hebbian learning components are combined additively at each connection to produce a net weight change.

The equation for the Hebbian weight change is:

$$\Delta_{hebb} w_{ij} = x_i^+ y_j^+ - y_j^+ w_{ij} = y_j^+ (x_i^+ - w_{ij})$$

and for error-driven learning using CHL:

$$\Delta_{err} w_{ij} = (x_i^+ y_j^+) - (x_i^- y_j^-)$$

which is subject to a soft-weight bounding to keep within the 0-1 range:

$$\Delta_{sberr} w_{ij} = [\Delta_{err}]_+ (1 - w_{ij}) + [\Delta_{err}]_- w_{ij}$$

The two terms are then combined additively with a normalized mixing constant k_hebb:

$$\Delta w_{ij} = \epsilon[k_{hebb}(\Delta_{hebb}) + (1 - k_{hebb})(\Delta_{sberr})]$$

## Specific Leabra Object Information

The following provide more details for the main class objects used in Leabra.

### Network structure

- LeabraNetwork
- LeabraLayer
- LeabraPrjn
- LeabraUnit
- LeabraCon

### Specs

As discussed in Specs, the specs contain all the parameters and algorithm-specific code, while the above objects contain the dynamic state information.

- LeabraLayerSpec
- LeabraUnitSpec
- LeabraConSpec

### Special Algorithms

See LeabraWizard for special functions for configuring several of these special architectural features.

#### Simple Recurrent Network Context Layer

- LeabraContextLayerSpec -- copies activation state of hidden layer

#### Scalar, Graded Value Representation

- ScalarValLayerSpec -- encodes and decodes scalar, real-numbered values based on a coarse coded distributed representation (e.g., a Gaussian bump) across multiple units. This provides a very efficient and effective way of representing scalar values -- individual Leabra units do not do a very good job of that, as they have a strong binary bias.

- TwoDValLayerSpec -- two-dimensional version of scalar val
- MotorForceLayerSpec -- represents motor output and input forces in a distributed manner across unit groups representing position and velocity.

### Temporal Differences and General Da (dopamine) Modulation

Temporal differences (TD) is widely used as a model of midbrain dopaminergic firing. Also included are Leabra Units that respond to simulated dopaminergic modulation (DaMod)

See Leabra TD for details.

### PVLV -- Pavlovian Conditioning

Simulates behavioral and neural data on Pavlovian conditioning and the midbrain dopaminergic neurons that fire in proportion to unexpected rewards (an alternative to TD). It is described in these papers: O'Reilly, Frank, Hazy & Watz, 2007 Hazy, Frank & O'Reilly, 2010. The current version (described here) is as described in the 2010 paper. A PVLV model can be made through the LeabraWizard -- under Networks menu.

See Leabra PVLV for full details.

### PBWM -- Prefrontal Cortex Basal Ganglia Working Memory

Uses PVLV to train PFC working memory updating system, based on the biology of the prefrontal cortex and basal ganglia. For complete details, see O'Reilly and Frank, 2006.

Described in Leabra PBWM.

### Other Misc Classes

- MarkerConSpec -- a "null" connection that doesn't do anything, but serves as a marker for special computations (e.g., the temporal derivative computation instead of standard net input).
- LeabraLinUnitSpec -- linear activation function
- LeabraNegBiasSpec -- negative bias learning (for refractory neurons)
- TrialSynDepConSpec, TrialSynDepCon -- synaptic depression on a trial-wise time scale
- FastWtConSpec, FastWtCon -- transient fast weights in addition to standard slowly adapting weights
- ActAvgHebbConSpec -- hebbian learning includes proportion of time-averaged activation (as in the "trace" rule)

## Parameter Setting Tips and Tricks

See Leabra Params for detailed discussion about adjusting parameters.

## Software Version Update Issues

- Leabra 4.0.15 Notes -- Version 4.0.15 release notes for Leabra

# Leabra Params

This page contains various user's experiences for setting parameters in the Leabra algorithm.

See Modeling Principles for some more general tips on how to go about modeling.

## Randy's General Tips for Basic Networks

In a basic 3 layer Leabra network, the key parameters that make the biggest difference are the inhibition parameters and Hebbian learning:

- inhibition: Hidden layers should generally use either KWTA_AVG_INHIB or KWTA_KV2K with kwta_pt set to .6 in the former and .25 in the latter. Output layers often benefit somewhat by using KWTA_INHIB with kwta_pt set to .25, because they don't need the extra flexibility that these others provide. The main thing to manipulate is the kwta.pct parameter in the hidden layer(s), which generally can be between .10 and .25. With more systematic mappings where similarity-based generalization is beneficial, higher values are better. When the mappings are more random, sparser values are better, and sometimes even lower than .1 can work well (e.g., in the hippocampus model, for an extreme example).
- Hebbian: try the lmix.hebb value in .01, .001, .0001, 0.0, kinds of values, and then hone in on more detailed values around the one that works the best. Also the savg_cor.cor value can be important -- it defaults to .4, but .8 and 1.0 can also work better in some circumstances.

## Tips for Updating Projects (and XCAL parameters)

*If* you have any projects that have set the following parameters on the LeabraNetwork to non-standard values, you should reset them and save with latest version:

**sse_unit_avg** *un-check*<br>
**sse_sqrt** *un-check*<br>
**cycle_max** = 60 <-- XCAL uses ct_time parameters, which then (re)set cycle_max, so setting this directly has no effect!<br>
**maxda_stopcrit** = -1 <-- XCAL automatically (should) set maxda_stopcrit to -1 -- it doesn't use it -- every trial is same length.<br>
**trg_max_act_stopcrit** = 1<br>
**norew_val** = 0.5 <-- this is only in in PFC_BG_Layers/PVeLayer<br>
**off_errs** *check*<br>
**on_errs** *check*

The new XCAL default parameters are shown in the defaults for the LeabraUnitSpec **act_avg** field -- because both the old and new versions are supported, it might be a bit ambiguous, but here's the story for the new **l_sq** mode:

**l_sq** = true -- this is what puts it into the new squared mode..<br>
**l_gain** = 60<br>
**thr_min** = 0.01<br>
**thr_max** = 0.9<br>
**l_dt** = 0.05<br>
**ml_dt** = 1.0

*NOTE:* The remaining parameters in the LeabraUnitSpec **act_avg** field have not changed (not relevant for standard trial-based XCAL anyway -- only for fully continuous XCAL_C)

*NOTE:* For XCAL, if the connection specification (in the LeabraConSpec) is set to **wt_sig.off >= 1.1** the network may train poorly, and is never really any better, so use **wt_sig.off = 1.0** as the standard offset for the sigmoidal weight function. The auto default will be updated but existing projects will have whatever they were originally set to.

*NOTE:* Experimental results from MH, for self-organizing (Hebbian) set xcal.thr_l_mix=1, and lower the lrate by maybe half.

For general information about the **XCAL** learning model, please see the CECN textbook chapter on

Learning.

## BioParams

void LeabraUnitSpec::BioParams ( bool gelin = true, float norm_sec = 0.001f, float norm_volt = 0.1f, float volt_off = -0.1f, float norm_amp = 1.0e-8f, float C_pF = 281.0f, float gbar_l_nS = 10.0f, float gbar_e_nS = 100.0f, float gbar_i_nS = 100.0f, float erev_l_mV = -70.0f, float erev_e_mV = 0.0f, float erev_i_mV = -75.0f, float act_thr_mV = -50.0f, float spk_thr_mV = 20.0f, float exp_slope_mV = 2.0f, float adapt_dt_time_ms = 144.0f, float adapt_vm_gain_nS = 4.0f, float adapt_spk_gain_nA = 0.0805 )

set parameters based on biologically-based values, using normalization scaling to convert into typical Leabra standard parameters. gelin = configure for gelin rate-code activations instead of discrete spiking (dt = 0.3, gain = 80, gelin flags on), norm_x are normalization values to convert from SI units to normalized values (defaults are 1ms = .001 s, 100mV with -100 mV offset to bring into 0-1 range between -100..0 mV, 1e-8 amps (makes g_bar, C, etc params nice). other defaults are based on the AdEx model of Brette & Gurstner (2005), which the SPIKE mode implements exactly with these default parameters -- last bit of name indicates the units in which this value must be provided (mV = millivolts, ms = milliseconds, pF = picofarads, nS = nanosiemens, nA = nanoamps)

# Constraint Satisfaction

## Introduction

Constraint satisfaction is an *emergent* computational property of neural networks that have recurrent connectivity, where activation states are mutually influenced by each other, and settling over time leads to states that satisfy the constraints built into the weights of the network, and those that impinge through external inputs.

Constraint satisfaction can solve complicated computational problems where the interdependencies among different possible solutions and high-dimensional state spaces make searching or other techniques computationally intractable. The extent to which a network is satisfying its constraints can be measured by a global energy or "goodness" function. Proofs regarding the stability of equilibrium states of these networks and derivations of learning rules have been made based on these energy functions.

In the software, a collection of constraint satisfaction style algorithms have been implemented under a common framework. These algorithms include the binary and continuous Hopfield style networks (Hopfield, 1982, 1984), the closely related Boltzmann Machine networks (Ackley, Hinton and Sejnowski, 1985), the interactive activation and competition (IAC) algorithm (McClelland and Rumelhart, 1981), and GRAIN networks (Movellan and McClelland, 1994).

In addition to recurrent activation propagation and settling over time, these algorithms feature the important role that noise can play in avoiding sub-optimal activation states. The work with the GRAIN algorithm extends the role of noise by showing that the network can learn to settle into different distributions of activation states in a probabilistic manner. Thus, one can teach a network to go to one state roughly 70 percent of the time, and another state roughly 30 percent of the time. These distributions of possible target states can be specified by using a probability environment, which is described in a subsequent section.

Also, learning takes place in these networks through a more local form of learning rule than error backpropagation. This learning rule, developed for the Boltzmann machine, has been shown to work in a wide variety of activation frameworks, including deterministic networks. This rule can be described as a "contrastive Hebbian learning" (CHL) function, since it involves the subtraction of two simple Hebbian terms computed when the network is in two different "phases" of settling.

The two phases of settling required for learning are known as the *minus* (negative) and *plus* (positive) phase. The minus phase is the state of the network when only inputs are presented to the network. The plus phase is the state of the network when both inputs and desired outputs are presented. The CHL function states that the weights should be updated in proportion to the difference of the coproduct of the activations in the plus and minus phases:

```
cn->dwt = lrate * (ru->act_p * su->act_p - ru->act_m * su->act_m)
```

where ru is the receiving unit and su is the sending unit across the connection cn, and act_m is the activation in the minus phase, and act_p is the activation in the plus phase.

It turns out that in order to learn distributions of activation states, one needs to collect many samples of activation states in a stochastic network, and update the weights with the expected values of the coproducts of the activations, but the general idea is the same. This learning rule can be shown to be minimizing the cross-entropy between the distributions of the activations in the minus and plus phases, which is the basis of the Boltzmann machine derivation of the learning rule.

The *emergent* implementation allows you to perform learning in both the stochastic mode, and with deterministic networks using the same basic code. Also, there is support for *annealing* and *sharpening* schedules, which adapt the noise and gain parameters (respectively) over the settling trajectory. Using these schedules can result in better avoidance of sub-optimal activation states.

## Reference Implementation Details

There are Cs specific versions of all the standard network classes:

▸

Reference info for type **CsNetwork**: Wiki | Emergent Help Browser

▸

Reference info for type **CsLayer**: Wiki | Emergent Help Browser

▸

Reference info for type **CsUnit**: Wiki | Emergent Help Browser

,

Reference info for type **CsUnitSpec**: Wiki | Emergent Help Browser

▸

Reference info for type **CsCon**: Wiki | Emergent Help Browser

,

Reference info for type **CsConSpec**: Wiki | Emergent Help Browser

# Self Organizing

**Contents**

## Introduction

The defining property of self-organizing learning is that it operates without requiring an explicit training signal from the environment. This can be contrasted with error backpropagation, which requires target patterns to compare against the output states in order to generate an error signal. Thus, many people regard self-organizing learning as more biologically or psychologically plausible, since it is often difficult to imagine where the explicit training signals necessary for error-driven learning come from. Further, there is some evidence that neurons actually use something like a Hebbian learning rule, which is commonly used in self-organizing learning algorithms.

There are many different flavors of self-organizing learning. Indeed, one of the main differences between self-organizing algorithms and error-driven learning is that they need to make more assumptions about what good representations should be like, since they do not have explicit error signals telling them what to do. Thus, different self-organizing learning algorithms make different assumptions about the environment and how best to represent it.

One assumption that is common to many self-organizing learning algorithms is that events in the environment can be *clustered* together according to their "similarity." Thus, learning amounts to trying to find the right cluster in which to represent a given event. this is often done by enforcing a competition between a set of units, each of which represents a different cluster. The *competitive* learning *algorithm (CL) of Rumelhart and Zipser, 1985 is a* classic example of this form of learning, where the single unit which is most activated by the current input is chosen as the "winner" and therefore gets to adapt its weights in response to this input pattern.

The current implementation of self-organizing learning, called So, includes competitive learning and several variations of it, including "soft" competitive learning (Nowlan, 1990), which replaces the "hard" competition of standard competitive learning with a more graded activation function. Also included are a couple of different types of modified Hebbian learning rules that can be used with either hard or soft activation functions.

An additional assumption that can be made about the environment is that there is some kind of *topology* or ordered relationship among the different clusters. This notion is captured in the *self-organizing map* (SOM) algorithm of Kohonen (1989, 1990, 1995). This algorithm adds to the basic idea of competition among the units that represent a cluster the additional assumption that units which are nearby in 2-D space should represent clusters that are somehow related. This spatial-relatedness constraint is imposed by allowing nearby units to learn a little bit when one of their neighbors wins the competition. This algorithm is also implemented in the So package.

The directory <file>demo/so</file> contains two projects which demonstrate the use of both the competitive-learning style algorithms, and the self-organizing maps.

# Implementation Reference

The So implementation is designed to be used in a mix-and-match fashion. Thus, there are a number of different learning algorithms, and several different activation functions, each of which can be used with the other. The learning algorithms are implemented as different connection specs derived from a basic

Reference info for type **SoConSpec**: Wiki | Emergent Help Browser

type. They all use the same

Reference info for type **SoCon**: Wiki | Emergent Help Browser

connection type object.

Reference info for type **SoNetwork**: Wiki | Emergent Help Browser

,

Reference info for type **SoLayer**: Wiki | Emergent Help Browser

## Learning Functions (ConSpecs)

Reference info for type **SoConSpec**: Wiki | Emergent Help Browser

(base spec): has a learning rate parameter lrate, and

a range to keep the weight values in: wt_range. Unlike error-driven learning, many self-organizing learning algorithms require the weights to be forcibly bounded, since the positive-feedback loop phenomenon of associative learning can lead to infinite weight growth otherwise. Finally, there is a variable which determines how to compute the average and summed activation of the input layer(s), which is needed for some of the learning rules. If the network is fully connected, then one can set avg_act_source to compute from the LAYER_AVG_ACT, which does not require any further computation. However, if the units receive connections from only a sub-sample of the input layer, then the layer average might not correspond to that which is actually seen by individual units, so you might want to use COMPUTE_AVG_ACT, even though it is more computationally expensive.

Reference info for type **HebbConSpec**: Wiki | Emergent Help Browser

This computes the most basic Hebbian learning rule, which is just the

coproduct of the sending and receiving unit activations:

```
  cn->dwt += ru->act * su->act;
```

Though it is perhaps the simplest and clearest associative learning rule, its limitations are many, including the fact that the weights will typically grow without bound. Also, for any weight decrease to take place, it is essential that activations be able to take on negative values. Keep this in mind when using this form of learning. One application of this con spec is for simple pattern association, where both the input and output patterns are determined by the environment, and learning occurs between these patterns.

Reference info for type **ClConSpec**: Wiki | Emergent Help Browser

This implements the standard competitive learning algorithm as described in (Rumelhart & Zipser, 1985). This rule can be seen as attempting to align the weight vector of a given unit with the center of the cluster of input activation vectors that the unit responds to. Thus, each learning trial simply moves the weights some amount towards the input activations. In standard competitive learning, the vector of input activations is normalized by dividing by the sum of the input activations for the input layer, sum_in_act (see avg_act_source above for details on how this is computed).

```
cn->dwt += ru->act * ((su->act / cg->sum_in_act) - cn->wt);
```

The amount of learning is "gated" by the receiving unit's activation, which is determined by the competitive learning function. In the winner-take-all "hard" competition used in standard competitive learning, this means that only the winning unit gets to learn. Note that if you multiply through in the above equation, it is equivalent to a Hebbian-like term minus something that looks like weight decay:

```
cn->dwt += (ru->act * (su->act / cg->sum_in_act)) - (ru->act * cn->wt);
```

This solves both the weight bounding and the weight decrease problems with pure Hebbian learning as implemented in the HebbConSpec described above.

Reference info for type **SoftClConSpec**: Wiki | Emergent Help Browser

This implements the "soft" version of the competitive learning learning rule (Nowlan, 1990). This is essentially the same as the "hard" version, except that it does not normalize the input activations. Thus, the weights move towards the center of the actual activation vector. This can be thought of in terms of maximizing the value of a multi-dimensional Gaussian function of the distance between the weight vector and the activation vector, which is the form of the learning rule used in soft competitive learning. The smaller the distance between the weight and activation vectors, the greater the activation value.

```
cn->dwt += ru->act * (su->act - cn->wt);
```

This is also the form of learning used in the self-organizing map algorithm, which also seeks to minimize the distance between the weight and activation vectors. The receiving activation value again gates the weight change. In soft competitive learning, this activation is determined by a soft competition among the units. In the SOM, the activation is a function of the activation kernel centered around the unit with the smallest distance between the weight and activation vectors.

Reference info for type **ZshConSpec**: Wiki | Emergent Help Browser

This implements the "zero-sum" Hebbian learning algorithm (ZSH) (O'Reilly & McClelland, 1992), which implements a form of subtractive weight constraints, as opposed to the multiplicative constraints used in competitive learning. Multiplicative constraints work to keep the weight vector from growing without bound by maintaining the length of the weight vector normalized to that of the activation vector. This normalization preserves the ratios of the relative correlations of the input units with the cluster represented by a given unit. In contrast, the subtractive weight constraints in ZSH exaggerate the weights to those inputs which are greater than the average input activation level, and diminish those to inputs which are below average:

```
cn->dwt += ru->act * (su->act - cg->avg_in_act);
```

where avg_in_act is the average input activation level. Thus, those inputs which are above average have their weights increased, and those which are below average have them decreased. This causes the weights to go into a corner of the hypercube of weight values (i.e., weights tend to be either 0 or 1). Because weights are going towards the extremes in ZSH, it is useful to introduce a "soft" weight bounding which causes the weights to approach the bounds set by @code{wt_range} in an exponential-approach fashion. If the weight change is greater than zero, then it is multiplied by wt_range.max - cn->wt, and if it is less than zero, it is multiplied by cn->wt - wt_range.min. This is selected by using the soft_wt_bound option.

Reference info for type **MaxInConSpec**: Wiki | Emergent Help Browser

This learning rule is basically just the combination of SoftCl and Zsh.

It turns out that both of these rules can be derived from an objective function which seeks to maximize the input information a unit receives, which is defined as the signal-to-noise ratio of the unit's response to a given input signal (O'Reilly, 1994). The formal derivation is based on a different kind of activation function than those implemented here, and it has a special term which weights the Zsh-like term according to how well the signal is already being separated from the noise. Thus, this implementation is simpler, and it just combines Zsh and SoftCl in an additive way:

```
  cn->dwt += ru->act * (su->act - cg->avg_in_act) +
             k_scl * ru->act * (su->act - cn->wt);
```

Note that the parameter @code{k_scl} can be adjusted to control the influence of the SoftCl term. Also, the soft_wt_bound option applies here as well

## Activation Functions (UnitSpecs and LayerSpecs)

Activity of units in the So implementation is determined jointly by the unit specifications and the layer specifications. The unit specifications determine how each unit individually computes its net input and activation, while the layer specifications determine the actual activation of the unit based on a competition that occurs between all the units in a layer.

The So implementation uses

Reference info for type **SoLayerSpec**: Wiki | Emergent Help Browser

objects extensively. These layer

specifications implement competition among units in the same layer, which is central to the self-organizing algorithms. There are different layer specs all derived from a common SoLayerSpec.

All So Algorithms use the same

Reference info for type **SoUnit**: Wiki | Emergent Help Browser

. The only thing

this type adds to the basic Unit is the act_i value, which reflects the "independent" activation of the unit prior to any modifications that the layer-level competition has on the final activations. This is primarily useful for the soft Cl units, which actually transform the net input term with a Gaussian activation function, the parameters of which can be tuned by viewing the resulting act_i values that they produce.

There are three basic types of unit specifications, two of which derive from a common

Reference info for type **SoUnitSpec**: Wiki | Emergent Help Browser

. The SoUnitSpec does a very simple

linear activation function of the net input to the unit. It can be used for standard competitive learning, or for Hebbian learning on linear units.

Reference info for type **ClLayerSpec**: Wiki | Emergent Help Browser

,

Reference info for type **SoUnitSpec**: Wiki | Emergent Help Browser

-- hard competitive learning:

selects the winning unit (based on netin_type), and assigns it an activation value of 1, and it assigns all other units a value of 0. Thus, only the winning unit gets to learn about the current input pattern. This is a "hard" winner-take-all competition.

Reference info for type **SoftClLayerSpec**: Wiki | Emergent Help Browser

,

Reference info for type **SoftClUnitSpec**: Wiki | Emergent Help Browser

-- soft competitive learning: computes a Gaussian function of the distance between the weight and activation vectors. The variance of the Gaussian

is given by the var parameter, which is not adapting and shared by all weights in the standard implementation, resulting in a fixed spherical Gaussian function. Note that the net variable on units using this spec is the distance measure, not the standard dot product of the weights and activations. The SoftClLayerSpec does not explicitly select a winning unit. Instead, it assigns each unit an activation value based on a SoftMax function: $a_j = \frac{e^{g_j}}{\sum_k e^{g_k}}$ Where g_j is the Gaussian function of the distance between the unit's weights and activations (stored in act_i on the SoUnit object). Thus, the total activation in a layer is normalized to add up to 1 by dividing through by the sum over the layer. The exponential function serves to magnify the differences between units. There is an additional softmax_gain parameter which multiplies the Gaussian terms before they are put through the exponential function, which can be used to sharpen the differences between units even further. Note that @b{SoftClLayerSpec} can be used with units using the @b{SoUnitSpec} to obtain a "plain" SoftMax function of the dot product net input to a unit.

Reference info for type **SomLayerSpec**: Wiki | Emergent Help Browser

,

Reference info for type **SomUnitSpec**: Wiki | Emergent Help Browser

-- the self-organizing map:

simply computes a sum-of-squares distance function of the activations and weights, like the SoftClUnitSpec, but it does not apply a Gaussian to this distance. The winning unit in the SOM formalism is the one with the weight vector closest to the current input activation state, so this unit provides the appropriate information for the layer specification to choose the winner. The SomLayerSpec provides a means of generating a "neighborhood kernel" of activation surrounding the winning unit in a layer. First, the unit whose weights are closest to the current input pattern is selected (assuming the SomUnitSpec is being used, and the netin_type is set to MIN_NETIN_WINS). Then the neighbors of this unit are activated according to the neighborhood kernel defined on the spec. The fact that neighboring units get partially activated is what leads to the development of topological "map" structure in the network. The shape and weighting of the neighborhood kernel is defined by a list of NeighborEl objects contained in the neighborhood member. Each of these defines one element of the kernel in terms of the offset in 2-D coordinates from the winning unit (off), and the activation value for a unit in this position (act_val). While these can be created by hand, it is easier to use one of the built-in functions on the SomLayerSpec.

# Modeling Principles

Pragmatic tips for getting (big, complex) models to work.

- There are many parameters in the model, and many ways for it to fail.
- You do not want or need to figure out all the ways that the model can fail. There are far too many, and often there are complex interactions, etc. *Don't play 20 questions with a broken model.*
- Therefore, do everything you can to get your model working: throw the kitchen sink at it, and simplify the task to the point where it works. *Start from success.*
- Once you get it to work, *then* you can start playing 20 questions and find out what was critical, and start incrementally ramping up the difficulty of the problem, etc.
- The key difference is that the search gradients for a broken model are often flat: even if you change critical parameters in the right direction, other bad parameter settings prevent you from seeing this, and therefore you have no way of knowing what to change to fix your model. In contrast, in a working model, the effect of changing a given parameter should be evident in either worse or better performance, and this can then allow you to quickly tune parameters.
- Even though it is not always true, it is best to start with the assumption that parameters are independent: search parameter space along each parameter independently, and then pick the best combination of these separate parameter manipulations. You can then find out if the combination works best, and try out various conjunctions if it does not. This is essentially how neural networks themselves learn: they have a linear bias, and only introduce conjunctive representations when forced by the task.
- The human motivational system is a critical factor as well: success is much more motivating than failure, and it gives you reason to continue.
- In short: start small, simple, and work incrementally toward more complex models, and if it isn't working, then try everything you can think of to make it work.

## The Bread Baking Story

These same principles apply to baking bread: Yuko and Randy had a bread machine, and were making lousy hard "stone bread". They were trying to change one parameter at a time (heat, yeast, water, etc), and nothing was working. Yuko's dad suggested: "do everything you can think of all at once to make it work, then you can figure out what was critical later" -- this worked, and, in fact, they never bothered with the "science" tests afterward -- just enjoyed the tasty bread!

## The Importance of Learning

Although we often tell others repeatedly about how important learning is to our models, sometimes we forget this important lesson. The bottom line is: **always think about ways in which you can train your network to get it to do what you want it to do** instead of wasting time with other forms of coercion (e.g., manual parameter tweaking).

# emergent keyboard shortcuts

## Standard navigation keys

A variety of standard emacs/readline navigation keys have analogs throughout emergent. The first line of each top-level section lists which of these work there. On OSX you can swap the Cmd and Ctrl keys and Cmd+V is paste while Ctrl+V is Page Down.

| | |
|---|---|
| `Tab/Shift+Tab` | Forward/Backwards through elements/interface |
| `Page Up/Down` | Move cursor to the top/bottom or first/last element |
| `Ctrl+Space` | Enable select as you navigate mode. |
| `Ctrl+F` | Move cursor forwards or expand node. |
| `Ctrl+B` | Move cursor backwards or expand node. |
| `Ctrl+N` | Move cursor down one line. |
| `Ctrl+P` | Move cursor up one line. |
| `Ctrl+A` | Move cursor to first character |
| `Ctrl+E` | Move cursor to last character |
| `Ctrl+D` | Delete item in focus or all selected items. |
| `Ctrl+G` | Deselect text or tree selection. |
| `Ctrl+X/W` | Cut. |
| `Ctrl+C/Alt+W` | Copy. |
| `Ctrl+V/Y` | Paste. |

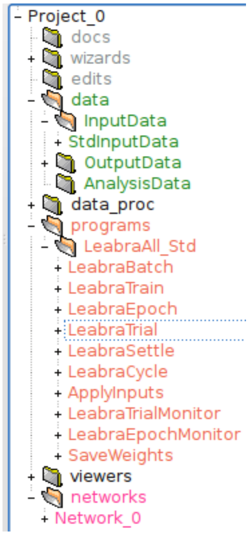
## Global project

| | |
|---|---|
| `Standard` | Tab |
| `Ctrl+S` | Save project. |
| `Ctrl+left` | Backwards in navigation history |
| `Ctrl+right` | Forwards in navigation history |
| `F5` | Refresh GUI. |
| `Ctrl/Alt+J` | Move global focus left |
| `Ctrl/Alt+L` | Move global focus right |

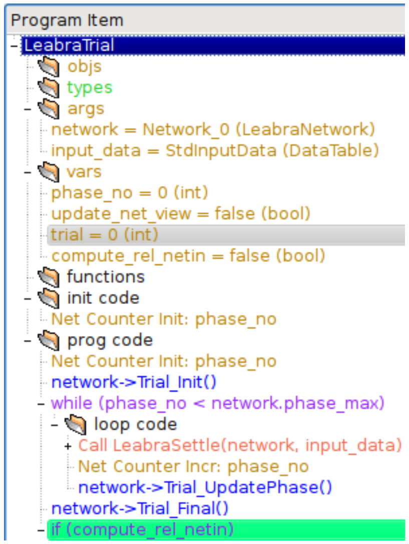
## Control panels

| | |
|---|---|
| `Init,Run,Step,Stop,Abort` | F8,F9,F10,F11,F12 |

## Project tree and program tree



```
- Project_0
    docs
  + wizards
    edits
  - data
    - InputData
      + StdInputData
    + OutputData
      AnalysisData
  + data_proc
  - programs
    - LeabraAll_Std
      + LeabraBatch
      + LeabraTrain
      + LeabraEpoch
      + LeabraTrial
      + LeabraSettle
      + LeabraCycle
      + ApplyInputs
      + LeabraTrialMonitor
      + LeabraEpochMonitor
      + SaveWeights
  + viewers
  - networks
    + Network_0
```



```
Program Item
- LeabraTrial
    objs
    types
  - args
      network = Network_0 (LeabraNetwork)
      input_data = StdInputData (DataTable)
  - vars
      phase_no = 0 (int)
      update_net_view = false (bool)
      trial = 0 (int)
      compute_rel_netin = false (bool)
    functions
  - init code
      Net Counter Init: phase_no
  - prog code
      Net Counter Init: phase_no
      network->Trial_Init()
    - while (phase_no < network.phase_max)
      - loop code
        + Call LeabraSettle(network, input_data)
          Net Counter Incr: phase_no
          network->Trial_UpdatePhase()
      network->Trial_Final()
    - if (compute_rel_netin)
```

| | |
|---|---|
| `Standard` | Page,Ctrl+Space/F/B/N/P/D/G/X/W/C/V/Y. |
| `Any 1-3 chars` | Find as you type. Moves cursor to next matching lin |
| `Ctrl+I` | New item below cursor. |
| `Alt+F` | Find from selected node. |
| `Ctrl+M` | Duplicate element(s). |

## New elements in project tree

| | |
|---|---|
| `do Ctrl+I` | New Doc |
| `da Ctrl+I` | New DataTable |
| `la Ctrl+I` | New Layer |
| `P Ctrl+I` | New Project |
| `pr Ctrl+I` | New Program |
| `n Ctrl+I` | New Network |
| `sp Ctrl+I` | New Spec |

## New elements in program tree

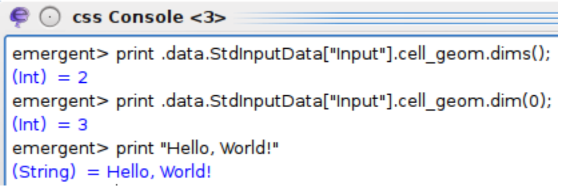
These sequences insert new items. Press Ctrl+left,left afterwards to navigate back to where you were.

| | |
|---|---|
| `obj Ctrl+I Type` | New obj of Type |
| `var Ctrl+I` | New var |
| `arg Ctrl+I` | New arg |
| `fun Ctrl+I` | New fun |
| `init Ctrl+I Name` | New init code Name |
| `prog Ctrl+I Name` | New prog code Name |

## css console and text fields

| | |
|---|---|
| `Standard` | Ctrl+F/B/N/P/A/E/D/X/W/C/V/Y. |
| `Ctrl+K` | Kill text until end of line. |
| `Ctrl+right` | Move cursor one word forward. |
| `Ctrl+left` | Move cursor one word backwards. |
| `Ctrl+shift+right` | Highlight one word forward. |
| `Ctrl+shift+left` | Highlight one word backwards. |

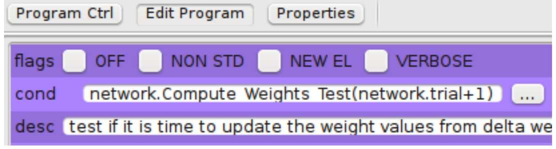
### css console



```
emergent> print .data.StdInputData["Input"].cell_geom.dims();
(Int)  = 2
emergent> print .data.StdInputData["Input"].cell_geom.dim(0);
(Int)  = 3
emergent> print "Hello, World!"
(String)  = Hello, World!
```

| | |
|---|---|
| `Ctrl+L` | Clear console buffer history. |
| `Ctrl+L` | Delete highlighted text. |

## Text fields



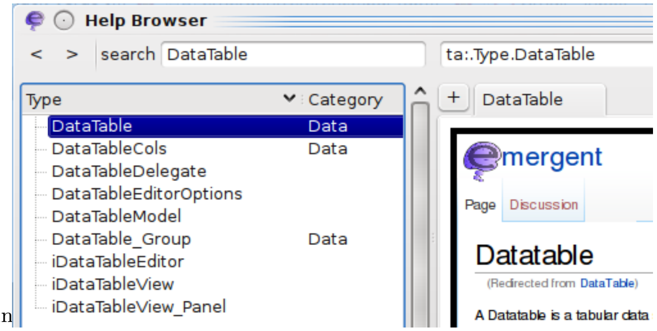
flags ○ OFF ○ NON STD ○ NEW EL ○ VERBOSE

cond network.Compute_Weights_Test(network.trial+1)

desc test if it is time to update the weight values from delta we

| | |
|---|---|
| `Ctrl+L` | Text completion lookup. |
| `Ctrl+U` | Highlight all. |

## Help Browser



search DataTable

ta:.Type.DataTable

| Type | Category |
|---|---|
| DataTable | Data |
| DataTableCols | Data |
| DataTableDelegate | |
| DataTableEditorOptions | |
| DataTableModel | |
| DataTable_Group | Data |
| iDataTableEditor | |
| iDataTableView | |
| iDataTableView_Panel | |

DataTable

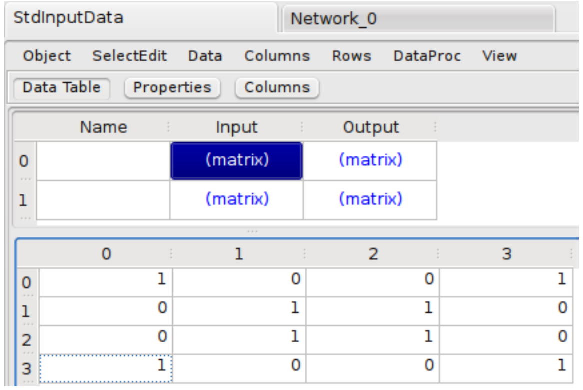
Datatable

(Redirected from DataTable)

A Datatable is a tabular data

| | |
|---|---|
| `Standard` | Tab,Page,Ctrl+F/B/N/P/A/E/X/W/C/V/Y. |
| `F1` | Help Browser. |
| `Ctrl+S` | Toggle Search/Find focus. |

## DataTables and matrices



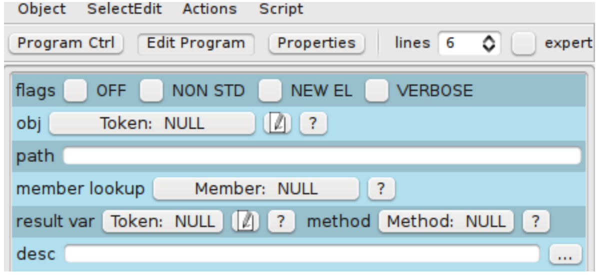
| | Name | Input | Output |
|---|---|---|---|
| 0 | | (matrix) | (matrix) |
| 1 | | (matrix) | (matrix) |

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 2 | 0 | 1 | 1 | 0 |
| 3 | 1 | 0 | 0 | 1 |

| | |
|---|---|
| `Standard` | Page,Ctrl+F/B/N/P/A/E/C/V/D/G/Y. |
| `Ctrl+T` | Switch between table and matrix focus. |
| `Ctrl+I` | Insert new row. |
| `Ctrl+M` | Duplicate row. |
| `Ctrl+Space` | Start editing cell. |

## Edit panels

See the "Text fields" section for shortcuts that work on the text fields of edit panels.



lines 6

flags ○ OFF ○ NON STD ○ NEW EL ○ VERBOSE

obj Token: NULL

path

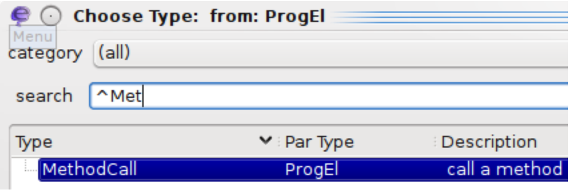
member lookup Member: NULL

result var Token: NULL method Method: NULL

desc

| | |
|---|---|
| `Standard` | Tab |
| `Up/Down arrow` | (numeric field) Increase/decrease value. |
| `Up/Down arrow` | (dropdown) Move up/down. |
| `ESC` | Revert changes. |
| `Ctrl+Enter` | Apply changes. |
| `Space` | Open token chooser and Check/uncheck flag. |
| `Ctrl+L` | (expression fields) Lookup information. |

## Choosers



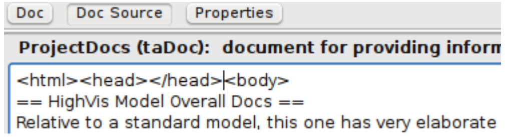
category (all)

search ^Met

| Type | Par Type | Description |
|---|---|---|
| MethodCall | ProgEl | call a method |

| | |
|---|---|
| `Standard` | Tab,Ctrl+F/B/N/P/A/E/D/X/W/C/V/Y. |

## Docs



ProjectDocs (taDoc): document for providing inform

```
<html><head></head><body>
== HighVis Model Overall Docs ==
Relative to a standard model, this one has very elaborate
```

| | |
|---|---|
| `Standard` | Tab,Space,Ctrl+F/B/N/P/A/E/D/X/W/C/V/Y. |

# Color coding

Color hints are used widely throughout *emergent*'s interface to allow you to assess the attributes of elements at a glance.

## Main project tree view

### Foreground Colors

| Type | Color | Example | Description |
|---|---|---|---|
| docs, wizards and edits | grey | docs | **Only docs, wizards and edits have this color** |
| data | green | data | **This is the standard color for datatables, datatable processing operations, data analysis, data functions, and image processing operations** |
| programs | red | programs | **This is the standard color for programs** |
| networks | pink | networks | **This is the standard color for networks** |
| Viewers and miscellaneous objects | black | viewers | **Miscellaneous program objects that do not inherit the standard object colors above simply have a black font** |

### Network Hierarchy Colors

| Type | Color | Example | Description |
|---|---|---|---|
| networks | pink | networks | **This is the standard color for networks** |
| spec groups | violet | specs | **Any lower level spec grouping under the specs catagory will adopt the color of whichever specification is listed first in the sub-group** |
| Unit Specifications | violet | LeabraUnitSpec | **This is the standard color for unit specs** |
| Layers and Layer Specifications | slate blue | LeabraLayerSpec | **This is the standard color for layers and layer specs** |
| Lesioned Layers | *grey highlight* | Hidden_Layer | This layer is lesioned |
| Projection Specifications | orange | FullPrjnSpec | **This is the standard color for projection specs** |
| Connection Specifications | bright green | LeabraConSpec | **This is the standard color for networks** |

# Program Colors

## Program Text Colors

The primary text colors for various program elements are as follows:

| Program Element | Text Color | Example | Description |
|---|---|---|---|
| Comment | brick red | NOTE: The following custom code is used to modify... | Program comment |
| Ctrl | purple | return() | Program keywords: return, for, while, if, stop/step, data loop |
| Function | blue | memb.mth() | Program functions: meth(), memb.mth(), fun def, fun(), various other functions |
| Type | lime green | enum RndInitType (3 items) | Program types: enumerated data types |
| objs, vars, or args | goldenrod | network = Network_0 (LeabraNetwork) | Program variables: declarations and assignment statements |
| datatable functions | green | ResetDataRows of: input_data | Program datatable functions/operations |

## Program Check Configuration (or compile) Indicator Colors

After check configuration is run program elements which contain problems are highlighted as follows:

| Program Element | Hightlight Color | Example | Description |
|---|---|---|---|
| Item contains invalid (child) item | *orange highlight* | SaveWeights | Orange highlight used to indicate that the program or program element contains invalid item |
| Invalid Item | *red highlight* | network = NULL (network) | Red highlight used to indicate invalid program items |

## Background Highlight Colors for program FLAGS

The background colors of programs are highlighted according to that program's flags as follows:

| Flag | Color | Example | Description |
|---|---|---|---|
| NO_STOP_STEP | *ivory highlight* | ApplyInputs | Flag set on **program**. Don't stop at this program when the step button is pushed. |
| TRACE | *grey highlight* | LeabraStartUp | Flag set on **program**. Record each line to the css console as this program runs. |
| STARTUP_RUN | *green highlight* | LeabraStartUp | Flag set on **program**. Run this program when *emergent* starts. |
| LOCKED | *red highlight* | FooBarProgram | Flag set on **program**. This program should not be edited. |
| | | | |
| OFF | *grey highlight* | Call ApplyInputs (network, input_data) | Flag set on **program element**. Program element is off, or program variable or argument is unused |
| NON_STD | *yellow highlight* | Print: tag | Flag set on **program element**. Program element is not part of the standard code for this program |
| NEW_EL | *green highlight* | network->Compute_ExtRew() | Flag set on **program element**. Program element was flagged as new (used to highlight a recently added elements) |
| VERBOSE | *tan highlight* | network->SaveWeights (fname, Network::NET_FMT) | Flag set on **program element**. Program element was flagged to print informative information about the operation of this program element when the program is running. |
| BREAKPOINT | *violet highlight* | if (compute_rel_netin) | Flag set on **program element**. Breakpoint was set for the program element. |