brains, minds media
ISSN: 1861-1680

Teaching Cognitive Modeling Using PDP++

David C. Noelle

University of California, Merced

5200 North Lake Road Merced, California 95344USA, dnoelle@ucmerced.edu

urn:nbn:de:0009-3-14069

Abstract. PDP++ is a freely available, open source software package designed to support the development, simulation, and analysis of research-grade connectionist models of cognitive processes. It supports most popular parallel distributed processing paradigms and artificial neural network architectures, and it also provides an implementation of the LEABRA computational cognitive neuroscience framework. Models are typically constructed and examined using the PDP++ graphical user interface, but the system may also be extended through the incorporation of user-written C++ code. This article briefly reviews the features of PDP++, focusing on its utility for teaching cognitive modeling concepts and skills to university undergraduate and graduate students. An informal evaluation of the software as a pedagogical tool is provided, based on the author's classroom experiences at three research universities and several conference-hosted tutorials.

Keywords: PDP++, connectionism, neural networks, cognitive modeling, computational neuroscience, education

Citation: Noelle DC (2008). Teaching Cognitive Modeling Using PDP++. Brains, Minds & Media, Vol.3, bmm1406, *in: Lorenz S, Egelhaaf M (eds): Interactive Educational Media for the Neural and Cognitive Sciences,* Brains, Minds & Media, 2008.

Published: May 26th, 2008; revised references on August 4th, 2008.

1 Introduction

Computational modeling has been a central component of cognitive science since the field's inception. Computational cognitive models have also played an increasingly central role in many areas of cognitive psychology and cognitive neuroscience. While the utility of computational models for formalizing theories is now broadly recognized, the skills necessary to construct, analyze, and evaluate computational cognitive models have often been difficult to instill in students and new researchers. A commonly cited cause for this difficulty is the need for disparate intellectual skills when modeling, ranging from mathematical analysis and computer programming to an understanding of theories of cognition and an appreciation for the complexities of experimentally testing such theories. Courses on computational cognitive modeling often attract both computationally sophisticated students and computational novices. In such courses, students with strong backgrounds in psychology or neuroscience will sometimes be seated next to students who are completely ignorant of the long history of studies and speculations concerning the mind and the brain. It is rare to find students who are uniformly strong in all of the foundational skills that are important for the modeling enterprise. This difficulty is magnified when teaching connectionist, parallel distributed processing, artificial neural network, and computational cognitive neuroscience modeling. These approaches to the modeling of

cognition involve mathematical formalisms unfamiliar to many computer scientists and they depart radically from familiar characterizations of cognitive process in terms of stages or functional modules. Many students face substantial challenges when attempting to learn these modeling frameworks, and even those who obtain a reasonable conceptual understanding often struggle to master the skills necessary to construct models of their own.

Like other skills, the acquisition of cognitive modeling skills is facilitated by hands-on exploration and practice. The centrality of such computational experience can introduce obstacles for students who lack strong computer skills, however. While some courses on connectionist modeling insist that students face these obstacles head-on, requiring them to write computer programs in some standard high-level language like C++, Java[1] or MATLAB[2] many other courses attempt to reduce the computational burden through the use of cognitive modeling software packages. These software packages typically allow for the construction of cognitive models through the instantiation of provided template model types, offering the user the ability to specify a variety of model parameters while offering core connectionist algorithms as fully implemented modules. Through the use of such simulation software, students without computer programming skills are often able to gain experience in building models, executing simulations, and collecting and analyzing data concerning model performance.

PDP++ is one such cognitive modeling simulation software package. Like many other simulators, PDP++ was primarily developed to support broad and ongoing research activities involving the construction, analysis, and evaluation of computational models of cognition. It's use in education settings has been a secondary, though still important, concern influencing its design. Thus, students learning cognitive modeling using PDP++ gain experience using a research-grade software tool, supporting a more direct transfer of knowledge acquired in the classroom to actual research practice. PDP++ provides direct support for the most common connectionist architectures and learning algorithms, including:

- spreading activation, constraint satisfaction networks, including Hopfield networks (Hopfield 1982)
- Hebbian learning (Hebb 1949, Grossberg 1998), competitive learning (Rumelhart and Zipser 1986, Grossberg 1987), and self-organizing feature maps (Von der Malsburg 1973, Grossberg 1976a, Grossberg 1976b, Kohonen 2001)
- feed-forward backpropagation of error networks (Rumelhart et al., 1986a)
- recurrent backpropagation of error networks, including simple recurrent networks (Elman 1990) and the "long short-term memory" architecture (Hochreiter and Schmidhuber 1997)

The package also provides support for computational cognitive neuroscience models intended to make more substantial contact with biological measures, offering an implementation of the LEABRA framework (O'Reilly and Munakata, 2000) as well as the Real-time Neural Simulator program (see Section 3.3). Models may be fabricated and manipulated in PDP++ using its elaborate graphical user interface, avoiding any need for students to write program code. The simulator is an open source software project, however, and utilities are provided to augment PDP++ with user-written C++ code if substantial departures from the provided algorithms are required. A detailed reference manual is

[1] Java is a trademark of Sun Microsystems.
[2] MATLAB is a registered trademark of The MathWorks.

available both as a printable document and as a collection of linked web pages. The entire PDP++ package, including example models, may be downloaded free of charge.

This paper provides a brief description of the most salient features of PDP++, focusing on the role that it can play in teaching connectionist modeling concepts and skills. The use of PDP++ in classroom settings, involving either undergraduate or graduate students, is of primary interest here, though many of the issues raised are equally relevant to the education of established researchers or to self-learning situations. After providing an overview of the software, an informal evaluation of the pedagogical strengths and weaknesses of PDP++ is offered. This evaluation is based on the author's experience using PDP++ when teaching courses on parallel distributed processing, artificial neural networks, and computational cognitive neuroscience at Carnegie Mellon University (1999), Vanderbilt University (2002–2006), and the University of California, Merced (2007). This evaluation also draws on experience using PDP++ in tutorial settings, including its use at an International Brain Research Organization (IBRO) summer school (2004), and during single day presentations at the International Conference on Cognitive Modeling (ICCM 2007) and the Cognitive Science Society Conference (CSS 2007). Importantly, this report does not provide instruction in the use of PDP++ for research purposes. It offers only an overview and informal assessment of the role that this free software package can play in the education of future cognitive scientists.

2 Overview of PDP++

2.1 Development History

PDP++ gets its name from a previous cognitive modeling software package. The letters abbreviate "Parallel Distributed Processing," which was the title of the seminal volumes that reinvigorated interest in connectionist models several decades ago (Rumelhart et al. 1986b; McClelland et al. 1986). The PDP initials also labeled the software that accompanied the simulation exercises workbook associated with those tomes (McClelland and Rumelhart 1988). The developers of PDP++ saw the package as the progeny of the original PDP programs. Since PDP++ was written in C++, it was natural to augment the name with the C++ "increment" operator, suggesting that PDP++ moved one step beyond its predecessor.

The initial development of PDP++ took place at the Center for the Neural Basis of Cognition, hosted by Carnegie Mellon University and the University of Pittsburgh. The principal architect of the software was Randall C. O'Reilly, with additional substantial development provided by James L. McClelland and Chadley K. Dawson. A fully functional version of PDP++ was available by 1995. This package is now primarily supported by the O'Reilly laboratory at the University of Colorado at Boulder. This review discusses Version 3.1 of the software, which was released in 2003. This is the most current release at the time of this writing. Since 2003, efforts have been underway to produce a major rewrite of the PDP++ system, and a substantially updated release appeared late in 2007, as this article was being completed. The software was given a new name with this release — Emergent — and this major update is briefly discussed in Section 2.5.

2.2 Supported Platforms, Access, & Installation

Originally, PDP++ was developed for use on UNIX[3] machines. Since its initial release, PDP++ has been ported to a number of other operating systems, though it still retains traces of its roots. In particular, support on a variety of Microsoft Windows[4] platforms (including 95, 98, 2000, NT, and XP) is provided through the use of the Cygwin[5] package. CYGWIN provides some aspects of UNIX-like functionality within a Windows environment, allowing PDP++ to run on Windows machines without requiring an excessive amount of platform-specific code. Similarly, the Darwin project for Mac OS X[6] provides sufficient UNIX-like functionality to allow PDP++ to run under Max OS X Versions 10.3 or 10.4 (Tiger). Ongoing development of PDP++ is primarily performed within Linux[7] environments, and software updates have been most thoroughly tested under UNIX-like operating systems.

It is worth noting that PDP++ provides support for symmetric multiprocessing (SMP), allowing the software to leverage parallel processors when such hardware is available. Parallel processing support is built into a number of the network algorithms, and it is fairly fine-grained, distributing the computations associated with individual connectionist units or batches of connections across machine processors. The degree of threading over processors is parameterized, with controls provided in the PDP++ graphical user interface. Performance improvements due to the use of parallel processors vary largely with the nature of the connectionist simulation being conducted, but some substantial time savings have been reported using this feature of PDP++.

The PDP++ software package may be freely downloaded from the internet. There are three primary ways in which this software has been packaged for downloading. The first two ways are accessible from The PDP++ Software Home Page, located at:

http://psych.colorado.edu/~oreilly/PDP++/PDP++.html

The software is available from the “FTP” links. At these FTP sites, the PDP++ software is packaged in two general ways: binaries and source code. Binary executable files are provided for a variety of operating system platforms. In addition, an auto-installation executable for Windows is provided, and a Mac OS X package is available. Downloading and unpacking the PDP++ binary executables is the easiest way to get PDP++ up and running quickly. Alternatively, the PDP++ source code can be downloaded, compiled on your own computer, and installed. While “makefile” support and installation scripts are provided with the PDP++ source code, compiling PDP++ from the source code is often prone to minor dependency glitches, making the process inappropriate for those without extensive programming experience. A source code based installation is required, however, if you intend to augment PDP++ with your own C++ components. Fortunately, few PDP++ users need to incorporate new C++ modules, and this is particularly true in classroom situations, so downloading binary executables generally remains the option of choice. Details are provided in the “INSTALL” file provided on the FTP sites.

The third way to download PDP++ is only appropriate when all models to be developed and run make use of the LEABRA framework for computational cognitive neuroscience modeling. Binary executables for the PDP++ implementation of LEABRA, along with a collection of simulation exercises from O’Reilly & Munakata (2000), may be found on the web site supporting this textbook:

http://psych.colorado.edu/~oreilly/cecn_download.html

[3] UNIX is a registered trademark of The Open Group.
[4] Windows and Windows NT are registered trademarks of Microsoft.
[5] Cygwin is a registered trademark of Red Hat.
[6] Mac OS is a registered trademark of Apple.
[7] Linux is a registered trademark of Linus Torvalds.

This is the downloading option of choice if the full range of connectionist architectures and learning algorithms supported by PDP++ are not needed. Using the LEABRA framework to teach cognitive modeling is discussed further in Section 3.3.

It is worth noting that PDP++ may require the installation of other software in order to function properly. In particular, the Windows version requires Cygwin (www.cygwin.com), and the Mac OS X version requires support for the X Window System (www.apple.com). Also, compilation of the PDP++ source code requires the availability of a variety of standard C++ libraries, with details provided in the online installation instructions.

2.3 Features

2.3.1 System Overview

The functional organization of PDP++ reflects the object-oriented nature of the C++ programming language in which it is written. This structural organization is evident both to those augmenting PDP++ with user-written C++ code and to those using the graphical user interface. All aspects of connectionist cognitive models implemented in PDP++ are composed of objects. This structure is natural for describing the components of a connectionist network. Processing units are objects. Layers of such units are objects. Weighted connections between units are objects. Collections of such connections, called projections, are also objects. There are network objects that group all of these components together. The experiences from which networks learn their connection weights are also naturally described as objects. The structure of inputs and outputs to a network are encoded as pattern objects. Individual experiences, including both input patterns and target output patterns, are grouped into event objects. Collections of events form environment objects. Less intuitive for many students is the implementation of connectionist activation propagation algorithms and learning algorithms as hierarchies of process objects, which control the timing of event presentation and information processing within network structures. Even measures of network performance are reified as statistic, or stat, objects, and the data collected can be recorded by log objects. Components of the graphical user interface are also C++ objects which the user can customize.

Unlike standard C++ programs, PDP++ maintains information about the class structure of its objects during the execution of model simulations. Data structures recording the properties of each class are automatically generated, based on a parsing of the PDP++ source code at compile time, including any user-written augmentations to the system. This is accomplished through use of a subsystem of PDP++ called TypeAccess. Having class information available while PDP++ is running makes it easy for users to dynamically examine and modify the objects that make up a model in the midst of analysis and debugging. The components (members) of objects can be referenced by name, and type checking can be done dynamically when objects are modified. Maintaining this kind of dynamic type information offers extreme flexibility during the development and refinement of models, without requiring any sort of recompilation process.

The object-based structure of PDP++ models is strongly reflected in the system's graphical user interface. While this interface contains a number of tools specifically designed to ease the most common model construction and analysis tasks, virtually every object in a model may also be directly examined and modified through the use of default object editing windows. These editing windows are automatically constructed using class structure information. This makes them somewhat generic and often cluttered with rarely modified information, but it allows virtually all of the objects that make up a simulation to be modified through a forms-based graphical interface. The TypeAccess system allows

these editing windows to do dynamic type checking, keeping users from modifying objects in grossly invalid ways. TypeAccess also extracts comments from the underlying C++ source code, and this allows editing windows to contain short online help messages describing the various components of each object, based on the source code comments. Thus, even a generic editing window of this kind includes some guidance concerning the structure of the object being edited and the meaning of its various fields. An example generic editing window, for a layer object, is shown in Figure 1.

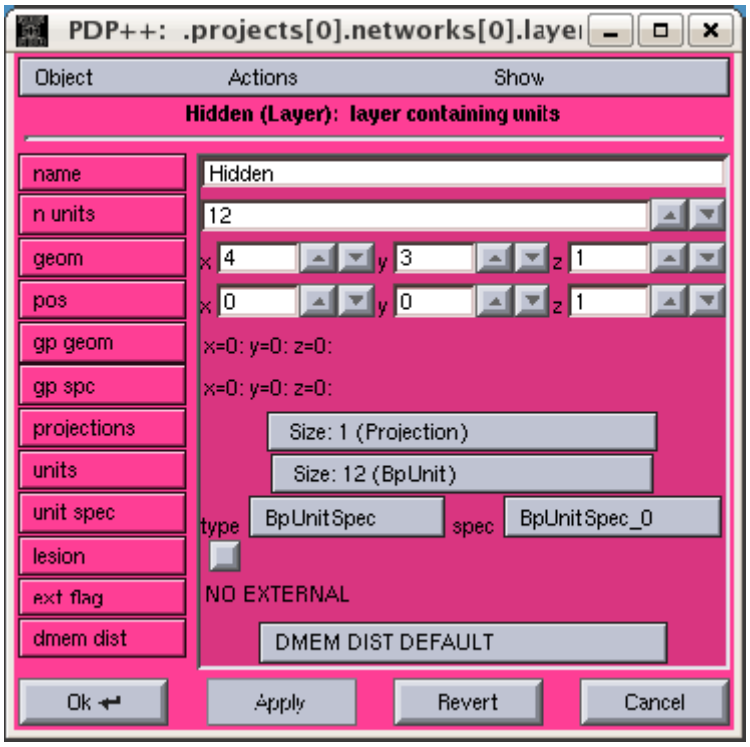

Figure 1: Generic Editing Window for a Layer Object

In many connectionist modeling frameworks, some parameters and features of a model component are relatively static and others are highly dynamic. For example, a processing unit's maximum possible activation level does not typically change during the course of model simulation, and this upper bound is typically shared by all of the units in a layer or in an entire network. Conversely, the activation level of each unit is typically highly dynamic, changing radically over the course of each event and changing in different ways for different units. PDP++ leverages this distinction between rarely changing shared parameters and more dynamic variables through its inclusion of specification objects, or specs. Each spec contains a collection of fairly static parameter values for a particular kind of network component. Thus, a unit spec contains information about the maximum activation level, while a unit object, proper, contains the unit's current activation level. This organization allows spec objects to be shared, when appropriate. For example, all of the units in a network can make use of the same unit spec object, allowing the user to change the maximum activation level of all units in the network simply by modifying that single unit spec object. Example editing windows for a unit and for a unit spec are shown in Figure 2.

Any object in PDP++ can be saved to a file for later use. Thus, a particular unit specification developed for one model can be saved into a file and then loaded into another model. Similarly, an environment object containing a collection of training events can be saved for later presentation to other model networks. By default, saved object files are written in plain text, allowing them to be viewed in any standard text editor, though the format of the files does not make for easy reading.

Because these plain text files can become quite large, they are compressed by default. This is done using the GNU Zip compression algorithm (for information on GNU Zip, see http://www.gnu.org/software/gzip/). PDP++ automatically compresses and uncompresses such files, as appropriate. Object files are given name extensions that reflect the type of object. For example, saved environments typically appear with an “env” filename extension. Files compressed with GNU Zip are traditionally given a “gz” filename extension, so PDP++ object files typically include this extension, as well. Thus, a saved environment might have a file name like “xor.env.gz”. When an object is saved, many of the objects that it references are also saved into the same file. Thus, saving an environment object also saves all of the events in that environment. PDP++ provides a top-level object type for individual models, called project objects, and each project references all of the key components of a model simulation. Thus, saving a project object into a file is sufficient to save all of one’s work on a given model. Such files have names like “xor.proj.gz” and are commonly called “project files”. Example project files are available from the PDP++ web site, and it is common for students and researchers using PDP++ to share their models with others by sharing project files. Instructors can provide demonstration models to students as project files, and student models can be submitted for evaluation in this same format.

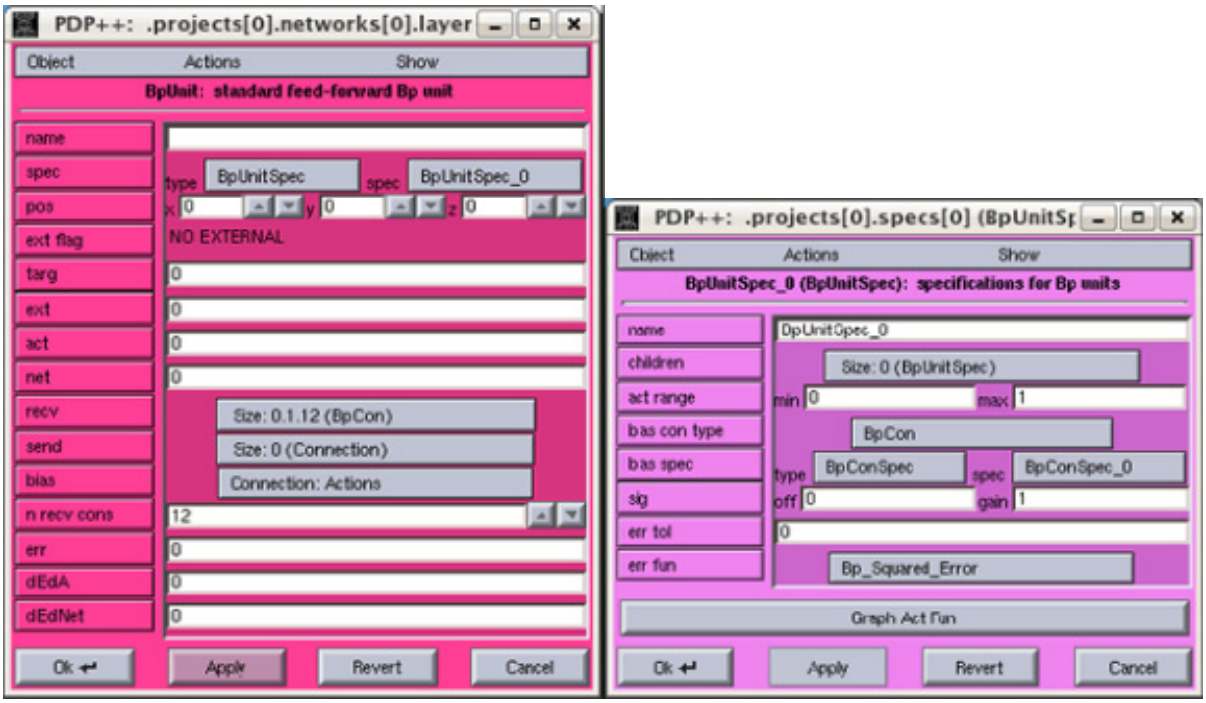

Figure 2: Generic Editing Windows for a Unit and for a Unit Spec

2.3.2 Supported Architectures

The PDP++ system is not committed to a particular connectionist modeling framework. Instead, support is provided for a variety of the most common parallel distributed processing paradigms. Different connectionist frameworks vary in their standard practices for model construction and model simulation, however. Since PDP++ provides support for many of these frameworks, it risks confusion and difficulties if components of the various frameworks are inappropriately mixed. In order to minimize such problems, PDP++ offers a collection of separate executable programs, each configured to support one general framework, by default. When starting PDP++, the user simply selects the framework of interest by executing the corresponding program. These programs include:

- cs++ — constraint satisfaction — This executable configures PDP++ to model constraint satisfaction networks. These networks focus on the dynamics of activation flow through recurrent connections to solve pattern completion problems. Attractor neural networks (Amit 1989, Grossberg 1988), such as Hopfield networks (Hopfield 1982), are easily modeled in this framework, as are networks similar to the Interactive Activation and Competition (IAC) model, which was used to explain the word superiority effect in letter perception (McClelland and Rumelhart 1981).

- so++ — self-organization — This executable configures PDP++ to model a variety of unsupervised learning networks, focusing on variants of Hebbian learning (Hebb 1949, Grossberg 1998). Extracting statistical regularities from experienced input patterns is the primary focus of these networks. Competition between units plays an important role in these algorithms, and PDP++ provides direct support for such. Competitive learning (Rumelhart and Zipser 1986, Grossberg 1987) and self-organizing maps (Von der Malsburg 1973, Grossberg 1976a, Grossberg 1976b, Kohonen 2001) are easily implemented with this program.

- bp++— backpropagation — This executable configures PDP++ for networks using the generalized delta rule (Rumelhart et al. 1986a) to learn connection weights based on error feedback. This includes both single-layer networks (involving directly connected input and output units) and multi-layer networks (including hidden units). This program also supports forms of the backpropagation of error algorithm appropriate for recurrently connected networks, including simple recurrent networks (Elman 1990), backpropagation through time (Rumelhart et al. 1986a), and various techniques for learning fixed-point attractors in recurrent networks (Almeida 1987; Pineda 1989).

- bpso++ — backpropagation with self-organization — This executable integrates the features of backpropagation networks with those of unsupervised self-organization networks, supporting models akin to counterpropagation networks (Hecht-Nielsen 1989).

- lstm++ — long short-term memory — This executable configures PDP++ to employ the kind of gated recurrent backpropagation networks used in the “long short-term memory” algorithm (Hochreiter and Schmidhuber 1997). This algorithm uses multiplicative connections in order improve the learning of long sequences, as compared to backpropagation through time.

- rns++ — real-time neural simulator — This executable configures PDP++ to use unit models and activation propagation algorithms appropriate for comparing processing unit activation directly to the time course of biological neural firing rates. Additional support for fitting these models to data is also provided. (Details may be found at the RNS++ web site at http://ccsrv1.psych.indiana.edu/rns++/.)

- leabra++ — LEABRA — This executable configures PDP++ for computational cognitive neuroscience modeling using the LEABRA framework (O’Reilly and Munakata 2000). These networks share many properties with other connectionist networks, but they also incorporate biological constraints at multiple scales. (See Section 3.3.)

It is important to note that all of these programs share a common structure and interface. Typically, skills involving the use of one executable largely transfer to the others. Thus, while broken into separate programs, PDP++ does offer a unified approach to building and exploring connectionist models. The only difference between the various executables is the library of specific object types used during network construction and the algorithms for activation propagation and weight learning that are readily available.

2.3.3 Environments

Specifying the inputs and expected outputs for connectionist network simulations is an important part of the modeling process. Models of human or animal learning, in particular, will only produce relevant results if the collections of training experiences presented to the models adequately capture the key properties of learning environments actually experienced by experimental participants. Rich training environments for computational models can be both difficult and tedious to generate, however. Typically, either every input to the model must be specified by hand, which can be tiresome if the learning environment is large and varied, or some algorithm must be implemented for automatically generating training experiences. PDP++ attempts to ease the process of developing training and testing environments by supporting three primary ways for entering experiential events into a model.

The first method for loading events into PDP++ involves the use of "pattern files" generated outside of the system. This is a common approach, taken by a variety of other connectionist simulation programs. Events are listed in a plain text file, with numerical input activation levels and target output activation levels provided explicitly. Such a file can be generated by hand, using a standard text editor, or it can be produced as the output of some other computer program. Thus, any available software that can produce plain text output can be used to generate a file of network input and output patterns. If a user is comfortable using a scripting language, like Perl, then script code may be used to automatically generate a pattern file. Similarly, spreadsheet programs like OpenOffice Calc or Microsoft Excel[8] may be used to generate such files. This approach to generating PDP++ environments provides the maximum of flexibility to the model developer, but it requires that the set of events be relatively static and unresponsive to model outputs, as the events are written to a file and loaded into PDP++ before simulations are run.

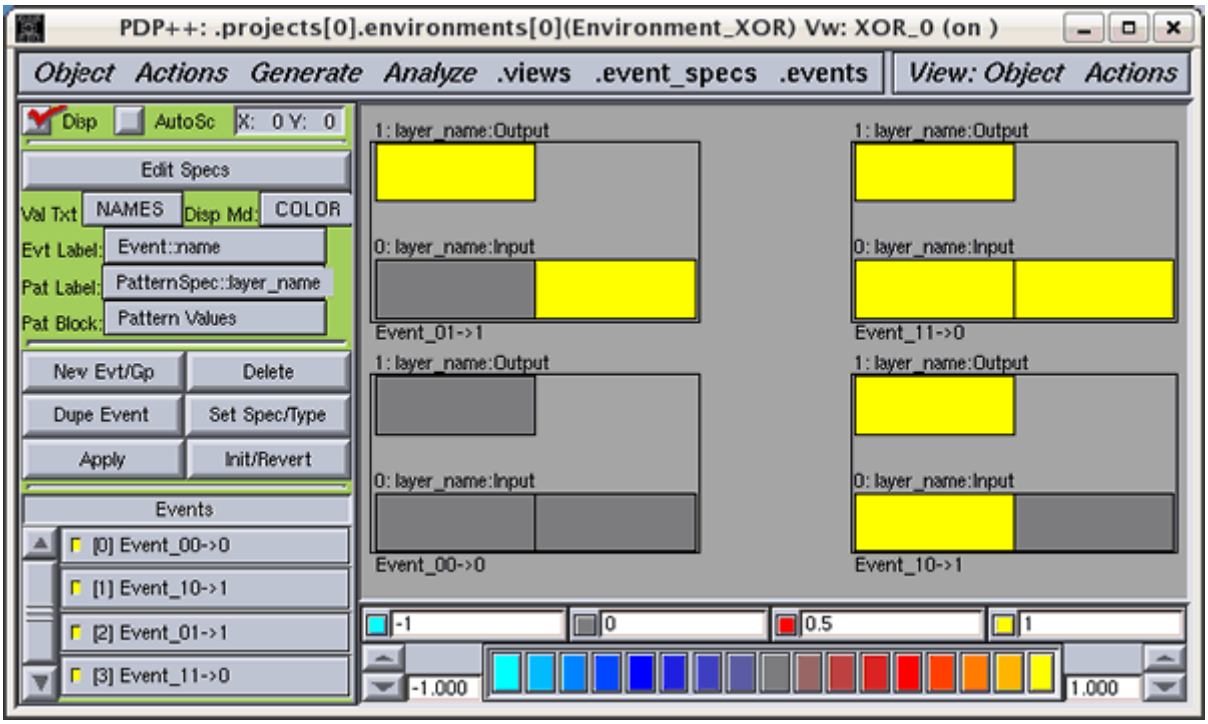

Figure 3: Environment View Window for the XOR Problem

[8] Excel is a registered trademark of Microsoft.

The second method for producing a PDP++ environment involves use of the system's graphical user interface to manually construct events. The PDP++ Environment View window, shown in Figure 3, allows users to populate an environment object with individual events and to specify the input and output activation values for each of those events. Using a kind of "painter's palette" at the bottom of the window, users can select an activation level and then "paint" the input and output values for different events with that level. This process of manually constructing events is easy and appropriate for environments containing a small number of events, but it can become tedious and error-prone when large environments are needed.

The third technique for populating a PDP++ environment with events is the most dynamic, but also the most intellectually demanding. PDP++ includes types of environments that automatically construct events by running user-written scripts written in a C++-like scripting language called CSS (see Section 2.3.5). Modelers willing to learn this scripting language, as well as the inner structure of the relevant environment and event classes in PDP++, can generate environments that produce events in a manner that is sensitive to such things as model performance over time. Arbitrarily complex and responsive environments can be produced in this way, but only at the cost of writing code. While such an effort is within the reach of many modeling researchers, many students new to PDP++ find this approach difficult.

Once constructed, environment objects may be saved into their own files. Frequently, such environments can be loaded into other PDP++ projects for use with other models, allowing the labor cost of environment construction to be amortized over multiple modeling efforts.

2.3.4 Graphical User Interface

The primary mode of interaction with PDP++ is through its graphical user interface. Virtually all of the common tasks associated with model construction and analysis can be accomplished through such point-and-click interactions. In addition to the generic editing windows that were previously described, PDP++ provides a number of graphical tools that are specifically designed to support the most regular operations on connectionist models. If, at any point during model development or use, the graphical interface is found lacking, a command line interface which provides direct access to the underlying objects is also available. This textual command line interface appears in the operating system shell window from which PDP++ was launched on some platforms (e.g., Linux), and it appears in a separate window generated by PDP++ on others (e.g., Windows). Textual commands are entered using an interpreted scripting language that has syntax similar to that of C++ (see Section 2.3.5). This scripting language allows objects to be directly inspected, and it allows objects to be modified in fairly arbitrary ways. While almost all interaction with PDP++ can be accomplished graphically, this command line interface allows advanced users to quickly and easily get "under the hood" of the system.

The graphical interface requires computational power to function, with CPU time needed to refresh displays and the like, and this means that use of the interface can slow simulations. Many of the graphical tools make it easy to temporarily disable updating of their contents, which can speed processing of a simulation, but there are times when it is desirable to disable the graphical user interface in its entirety. This is particularly useful when running a fully developed PDP++ simulation on a remote compute server or server cluster, where the time costs of keeping a graphical interface operating over a network connection can be prohibitive. In this case, PDP++ can be run "in batch mode", with the graphical interface disabled and, optionally, top-level control of the simulation orchestrated by a short script. Example batch scripts are available with the PDP++ software.

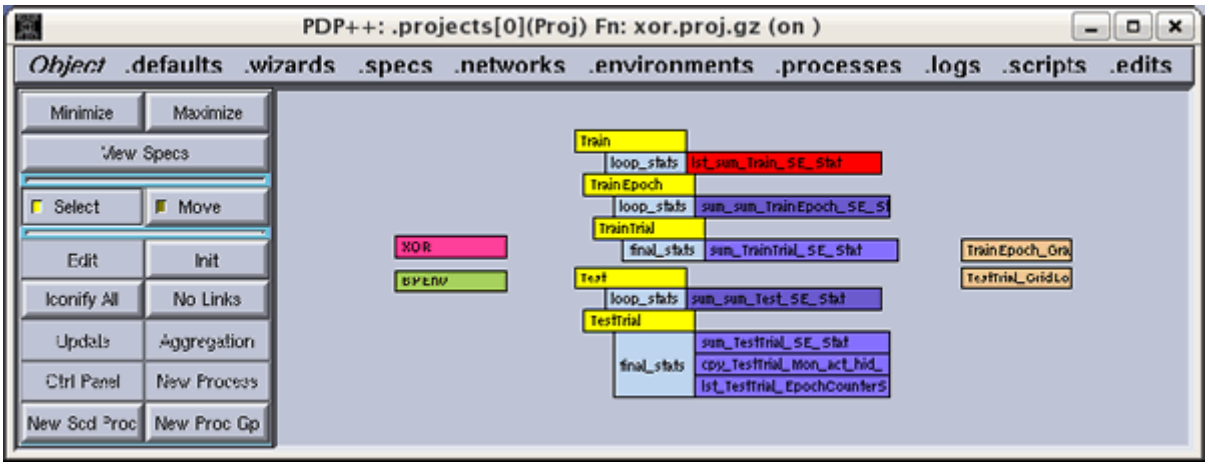

Figure 4: Project View Window for the XOR Problem

There are a number of graphical interface tools that are worthy of special mention, as they play central roles in the design and operation of PDP++ simulations. One such tool is the Environment View Window, which was previously discussed. Of the other important tools, the highest level one is the Project View Window. This window provides an overview of an entire model simulation, collecting and organizing all of the key objects that make up that simulation. This tool also provides an array of menus for creating and modifying the primary components of a simulation, including: specs, networks, environments, processes, and logs. An example Project View Window is shown in Figure 4.

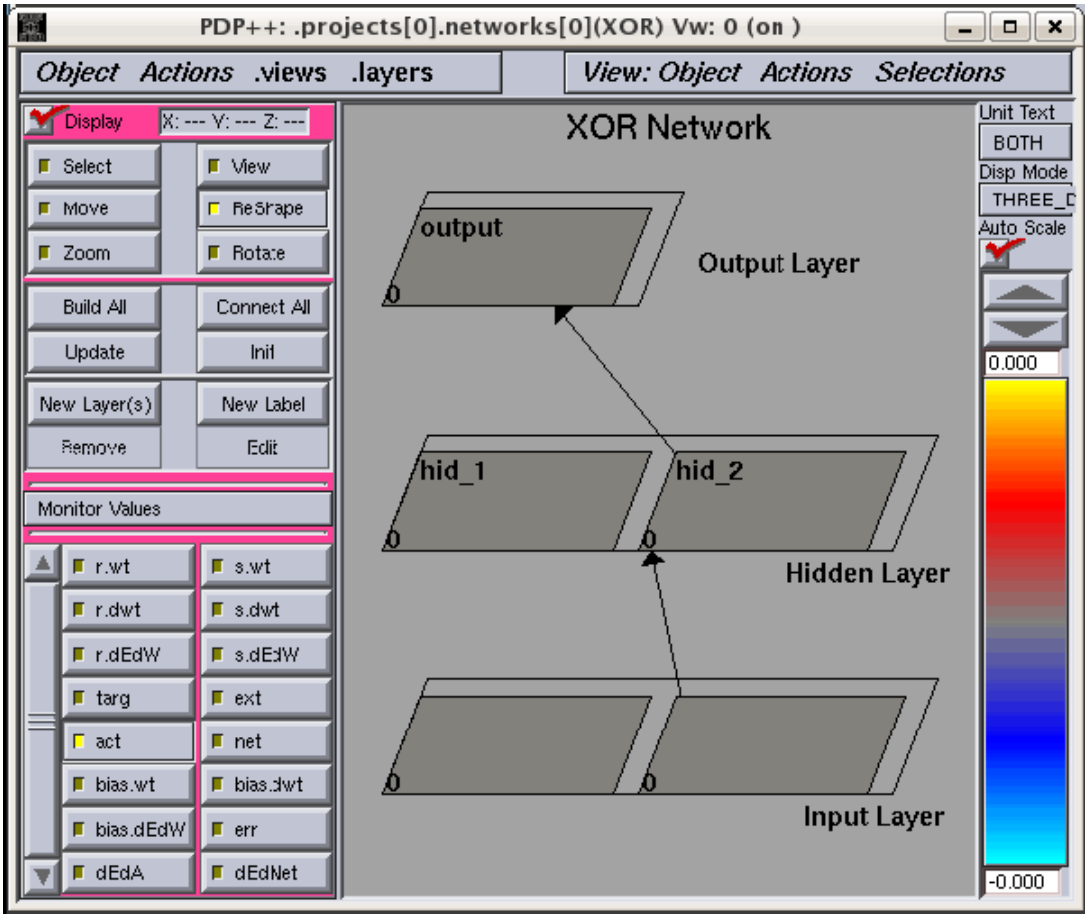

Figure 5: Network View Window Displaying the Architecture of an XOR Network

The main tool used to construct and examine connectionist network models in PDP++ is the Network View Window. The central pane of this tool displays the network architecture, and various properties of the network can be displayed in an overlayed fashion on this architectural diagram. An example Network View Window is shown in Figure 5. By default, layers of processing units are displayed as large rectangles, with each unit displayed as an embedded square. Collections of connections between layers, called projections, are displayed as arrows between the connected layers. These projections are often “complete” in that every unit in the sending layer is connected to every unit in the receiving layer, even though only one arrow is displayed between the layers, but projections can also involve random or structured patterns of connectivity between layers. By default, the network architecture is displayed in a form of 3D perspective, with layers stacked one above the other.

Networks can be constructed and modified through direct interaction with the Network View Window. A button on the left side of the window allows for the generation of new layers. Layers selected with the mouse can be edited using the corresponding generic editing window, or they can be reshaped directly with the mouse in the Network View Window, with processing units being added or deleted from the layer as it is reshaped. Projections are quickly and easily created by a pair of mouse selections on the layers to be connected. Mnemonic textual labels can be easily added to any part of the network display.

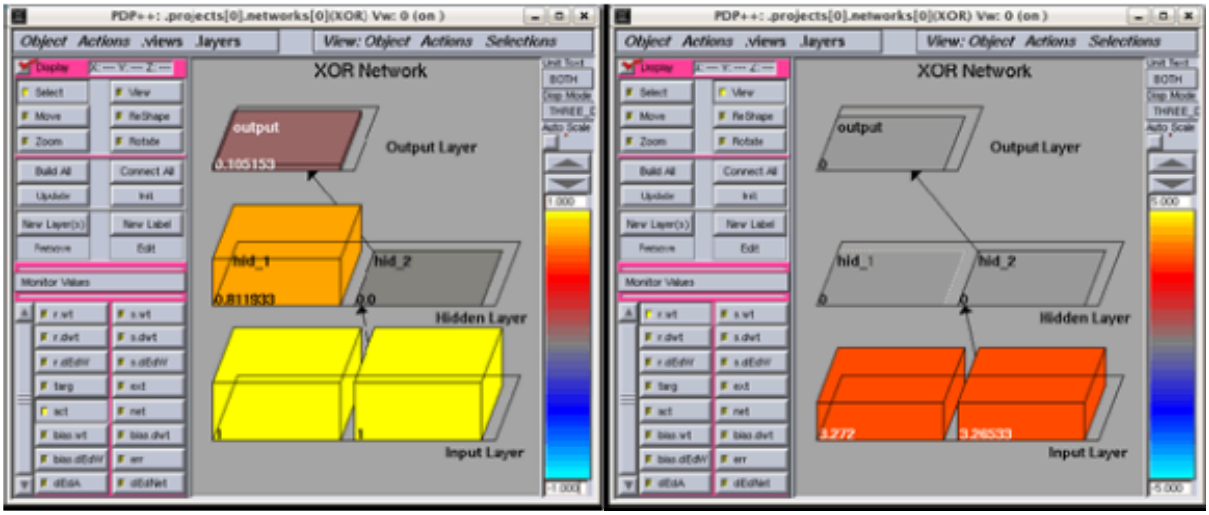

Figure 6: Browsing Values in a Network View Window: (a) Activation Values, (b) Connection Weight Values Entering Hidden Unit One

Numeric properties of a network can be graphically displayed on the network architecture diagram. For example, values associated with individual processing units, such as their current activation level, their current net input, or their current backpropagated error (delta) value, can be graphically superimposed over each displayed unit. The way in which these values are displayed — textually, through the height of a bar, through the degree to which a square is filled, through a color-coded scale, or through some combination of these — is under user control. For example, Figure 6(a) shows the display of unit activation values encoded textually, through the height of individual unit columns, and color-coded according to the scale on the right side of the window. Variable values associated with connections are more difficult to display on the network architecture diagram, as there are typically many more connections than units in a connectionist network. PDP++ resolves this problem by allowing for the display of connection-specific values, such as connection weights, only for connections associated with a selected unit, at any one time. Figure 6(b) shows an example of this, with the connection weight values received by Hidden Unit 1 displayed as text, column height, and color-coded according to the scale on the right edge of the window.

PDP++ offers much fiexibility in the way in which quantitative information is displayed in the Network View Window. The perspective view on the network can be flattened. Graphical bars can take on various shapes. The numerical value associated with a unit can be encoded by the degree to which that unit's square is filled, as is done in Hinton diagrams. The user may sample from a range of color scale options, and custom color scales may also be created. Some examples of customized network diagrams are shown in Figure 7.

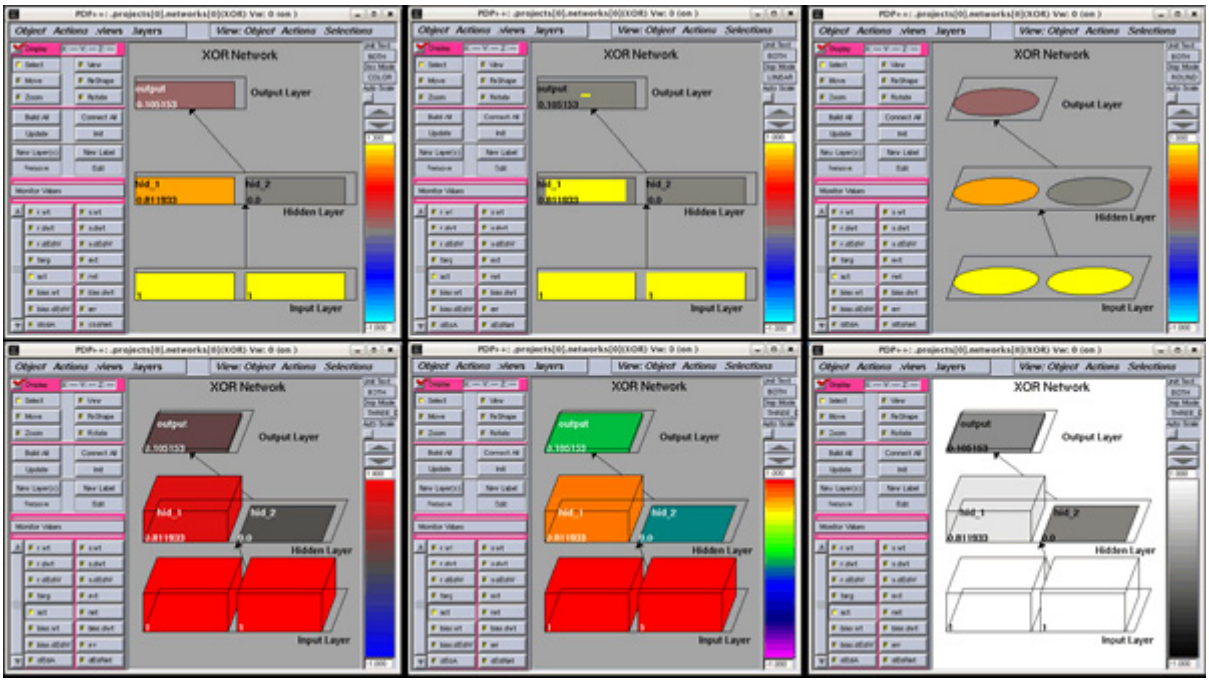

Figure 7: Some Network View Window Display Variations

Information about network performance can also be recorded from the Network View Window. After selecting a model component in the central pane, such as a layer or unit, pressing the "Monitor Values" button on the left side of the window will initiate a dialog during which the user can specify the properties of the selected component to be recorded (e.g., a units activation level) and the temporal granularity at which this data should be recorded (e.g., at the end of every event presentation). The result of this dialog will be the creation of a statistic, or stat, object which can be graphically displayed and logged to a plain text file.

While the Network View Window contains most of the instruments necessary to construct and examine a connectionist network model, the step-by-step execution of the algorithms for activation propagation and connection weight learning is orchestrated by a hierarchy of objects called process objects. Different process objects control the timing of the simulation at different levels of granularity, with some coordinating the settling of the recurrent flow of activation through the network, while others operate at the level of event presentations, and still others function at the level of complete passes (i.e., epochs) through entire sets of training events. The organization of the hierarchy of processes is one of the more complicated aspects of PDP++, and many students have difficulty initially grasping the role of these objects. In order to ease the use of process objects, PDP++ provides special windows for performing the most common operations on processes. These windows are called control panels. An example control panel for a training process is shown in Figure 8. This example control panel allows the user to change the training environment object being used for training. The user can also specify the maximum number of training epochs to be executed. Buttons near the bottom of the window allow the simulation to be initialized, to run the simulation, and to step through the simulation

one epoch at a time. Thus, the actual execution of the simulation can be completely controlled through the use of process control panels.

As network simulations are run, it is often useful to display measures of network behavior and performance. In PDP++, this is accomplished through the use of log objects and their corresponding view windows. Log objects typically record sequences of statistic object values, and present those sequences in some easy-to-read form. The most basic kind of log is a simple textual log, which consists of columns of textual, often numeric, data. Such tables can be displayed in a window in the PDP++ interface, and they can be written to plain text files. Additionally, the PDP++ web site contains a number of tools for parsing log files and converting them into other formats. Some quantitative data are best viewed as a line curve over time, and PDP++ supports such displays through its Graph Log objects. The kinds of data displays provided by the Network View Window, organized by network layer, are also sometimes worth recording, so PDP++ provides Grid Log objects to display data in a format similar to that shown over grids of processing units. More sophisticated log objects are also provided to perform calculations on collected data, such a principal component analysis and hierarchical clustering of vectors. Some simple example log views are shown in Figure 9.

Display of network performance in Network View Windows and Log View Windows provides useful guidance as models are being developed and analyzed, but it is often useful to share the results displayed in such windows with others. For the researcher, it is useful to extract these results for use in scientific papers and presentation slides. For the student, captured exercise results can be included in assignment reports. PDP++ provides support for recording the data displayed in these windows. Still images of the central pane of a Network View Window can be saved, as can images of displayed logged results. PDP++ can also produce animated movies of these displays, allowing users to share their observations concerning network dynamics in a fairly direct manner.

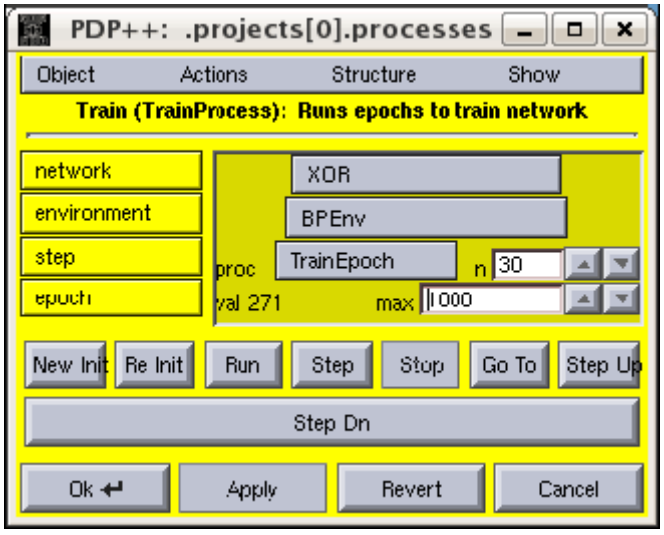

Figure 8: An Example Training Process Control Panel

The graphical user interface of PDP++ is rich, elaborate, and fairly comprehensive. It is designed to maximize flexibility and the user's access to the underlying C++ objects that make up a PDP++ model simulation. This flexibility can sometimes make it difficult for novice users to find a particular bit of information or tool of interest, as the desired item may be hidden among a variety of other tools and options. With practice, however, the graphical user interface can provide extensive easy-to-use support for a wide range of modeling tasks.

2.3.5 Scripting Language

While the graphical user interface provides extensive means to examine and modify the objects that make up a PDP++ simulation, some modeling activities require the specification of algorithmic model components that are not well captured by the range of defined object types. For example, one might wish to normalize the activation levels in a layer in some unusual way, or one might want to automatically change the size or number of layers in a network while a simulation is running, according to some criterion. In many simulation systems, such radical and unforeseen algorithmic modifications would require a modification to the system's source code. In PDP++, an interpreted scripting language based on C++ syntax, called CSS, is provided to the user as a means to allow for the integration of small program fragments into a model simulation. The CSS interpreter built into PDP++ has full access to the dynamic type information provided by TypeAccess, allowing runtime type checking to be done in CSS scripts, as well as other operations that are sensitive to class properties. The CSS scripting language is used in PDP++ in three main ways.

First, interaction with the PDP++ system through its command line interface is done in the CSS language. While the graphical user interface usually provides adequate support for interacting with PDP++ objects, there are times when a more algorithmic interface is welcome. For example, if one wanted to provide textual names to each processing unit in a large network, where the name of a unit was to be a systematic function of the names of the units providing it with input, it might easier to write a small snippet of CSS code that iterates over all of the units and computes their textual names, rather than manually clicking on each unit and setting its name by hand.

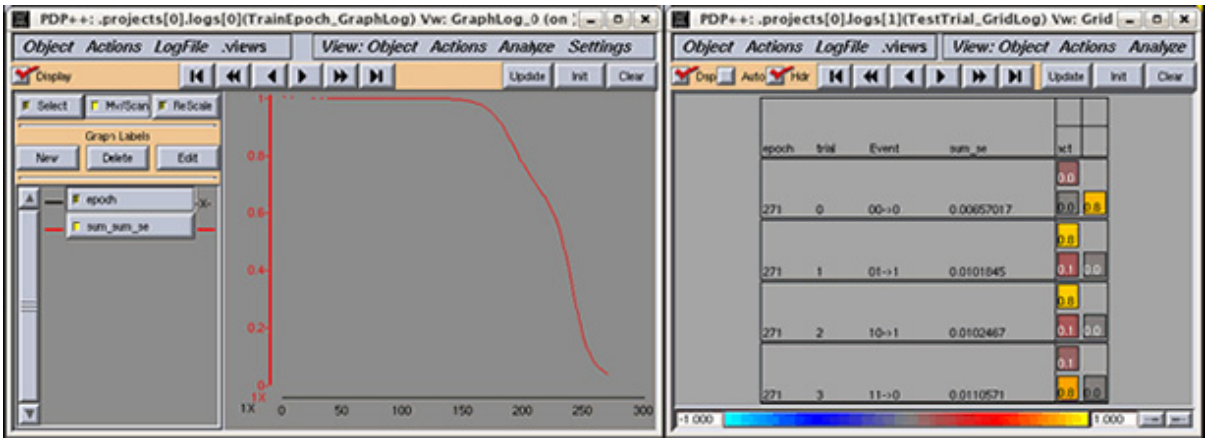

Figure 9: Simple Example Log View Windows: A Graph Log and a Grid Log

Second, many PDP++ objects provide hooks for user-defined CSS scripts, where the hooks specify when the scripts are to be called during the normal functioning of those objects. For example, environment objects are designed to call a user-written script, if provided, at the beginning of any pass through the set of events contained in that environment. Thus, a modeler could provide a script for this hook that generates a whole new collection of events, using some event generation algorithm, with each pass through the environment. In this way, the standard behavior of many PDP++ objects can be customized with user-written script code.

Lastly, CSS scripts provide a good way to coordinate the overall execution of a large simulation. For example, a top-level CSS script might initialize network connection weights, invoke a training process until a particular criterion is reached, save the resulting connection weights into a file, and then test the performance of the trained network on a different collection of events by running a testing process,

saving the results into a log file. Each of these steps could be taken manually through the graphical user interface, but the use of a top-level CSS script allows the whole simulation to be automated.

A variant on this last use of CSS involves defining a new C++ class from within a CSS script, with this class acting as a kind of “front end” or “control panel” for the whole simulation. Such a class can contain member variables for the most interesting model parameters and member functions that orchestrate the training and testing of models in the same manner that a top-level CSS script would. When an object of this new class is instantiated, the PDP++ graphical user interface will automatically generate a generic editing window for the newly defined “control panel” object, providing edit fields for the member variables and buttons corresponding to specified member functions. In this way, a CSS script can be used to produce a window that contains only the most relevant controls for the simulation at hand.

In classroom environments, students almost never need to write CSS code. The graphical interface is sufficient for learning exercises, removing any need to use the command line interface. Students will typically be learning about established connectionist frameworks, so there will be little need to customize PDP++ objects with user-written scripts. Pedagogical exercises are rarely so large that they require the automation of the simulation process, and students gain valuable experience by stepping through the stimulation steps manually. Despite this lack of relevance of CSS for the student, CSS scripts provide a very valuable tool to the instructor. Providing CSS defined control panel objects along with the models to be examined by the students offers important scaffolding and guidance to students as they explore the demonstration models on their own. Such control panels help students focus on the most salient aspects of the specific models being presented, keeping them from becoming lost among the huge array of options offered by the general PDP++ interface. In short, use of the CSS scripting language is in no way mandatory, and much productive work can be done in PDP++ without use of CSS. However, student learning can be facilitated when instructors take the time to learn CSS and, following examples from the PDP++ web site, construct control panel objects for the simulations to be provided to their students.

2.3.6 Customizing Executables

PDP++ provides tools for compiling new executable programs that include the functionality of PDP++ along with user-written C++ augmentations. In many cases, such recompilation is unnecessary, as the flexibility of CSS scripts allows many augmentations to be accomplished through the use of various hooks in existing PDP++ objects. Still, more radical modifications are not always easy in CSS, and the execution of CSS code can be slow, since it is interpreted at runtime. Thus, researchers working with PDP++ sometimes opt to make their experimental modifications in C++ and compile new versions of PDP++ that make use of those changes.

In classroom environments that include students with strong computer programming skills, it is sometimes tempting to offer this option to more advanced students, perhaps as part of ambitious term projects. This is a risky option, however. In order to augment the PDP++ system directly, one must be intimately familiar with the existing source code. The object classes offered by PDP++ are highly interactive, with apparently local modifications often having far-reaching effects. While students with strong programming skills may be able to easily cast the algorithmic modifications of interest into C++, there is rarely enough time within the scope of a single course to allow students to become sufficiently familiar with the PDP++ source code to avoid the pitfalls of the potential side effects of their modifications. Most of the time, it is better to direct strong students to the power of the CSS scripting language, rather than encouraging them to modify the PDP++ source.

2.4 Software Support

The PDP++ system is very powerful and highly flexible. It has been used to develop research-grade connectionist cognitive models in domains as diverse as classical conditioning and cognitive dissonance. It's power and flexibility come at a cost, however, and that is its typical learning curve. While demonstration simulations for classroom use are usually readily accessible to students, particularly when they are accompanied by simulation-specific CSS control panels, initial student efforts to construct new models and to make modifications to standard connectionist frameworks are often met with substantial learning obstacles. PDP++ provides two main resources to assist in overcoming these learning obstacles.

A largely comprehensive users manual for PDP++ has been written by the system's original designers (Dawson et al. 1997). A formatted version of this reference manual is available in PDF and PostScript[9] from the PDP++ web site. A hyperlinked web-based version is also provided at:

http://www.cnbc.cmu.edu/Resources/PDP++//manual/pdp-user_toc.html

It is important to note that this document is a reference manual and not a tutorial. While Chapter 4 of the manual provides some tutorial guidance, the bulk of this document was not written with pedagogy in mind. A more tutorial introduction to PDP++ is available in O'Reilly & Munakata (2000), but this textbook focuses exclusively on the LEABRA framework, rather than the full range of connectionist frameworks available in the PDP++ system. (For a discussion of the advantages of using LEABRA in a classroom setting, see Section 3.3.)

A second resource for learning PDP++ is an electronic mailing list that has been established for the PDP++ user community. This is a low traffic mailing list used to address questions from both students and researchers concerning the use and augmentation of the PDP++ system. Questions are broadcast to the whole list and addressed by volunteers. The principal architect of PDP++, Randall O'Reilly, regularly contributes to this mailing list. Instructions for subscribing to this mailing list may be found on the PDP++ web site.

While further learning and teaching support for PDP++ would certainly be welcome, these two resources, along with the example project files provided with the PDP++ system, are frequently adequate to bring new users up to speed in the use of this powerful modeling tool.

2.5 Ongoing Development Efforts: Emergent

At the time of this writing, development efforts for a substantial upgrade to PDP++ have been proceeding for several years. These efforts culminated in an initial release of the successor of PDP++, called Emergent, late in 2007. Extensive work in the O'Reilly laboratory at the University of Colorado at Boulder has gone into improving the graphical user interface of this system, and the process of installation, even when recompiling from source, has been made more seamless. There have also been substantial efforts to improve the documentation for the system.

Computational cognitive neuroscience courses at five or more major research universities are scheduled to make use of Emergent in the classroom during the Spring of 2008, but it is too early to evaluate the pedagogical strengths and weaknesses of this upgrade of the PDP++ software. A few

[9] PostScript is a registered trademark of Adobe Systems.

initial observations are made here, in hopes of assisting educators in anticipating the effects of placing Emergent in the hands of their students.

The Emergent software is supported by a new web site, maintained by the O'Reilly laboratory at the University of Colorado at Boulder, located at:

http://grey.colorado.edu/emergent/

This web site makes use of the MediaWiki system (www.mediawiki.com), which provides tools to allow the site to be easily augmented and edited by a collection of geographically distributed authors. This feature of the web site is being employed by the Emergent development team in order to quickly enrich system documentation. Users of the software are contributing online documentation on topics ranging from proper installation of the software on a variety of computer platforms to tips for the construction of particular kinds of cognitive models. The use of this technology is one way in which the Emergent developers hope to produce a more comprehensive and useful collection of software support documents than what is available for the previous PDP++ releases. Upon the initial release of Emergent, however, this documentation is still fairly sparse, despite the inclusion of many helpful guides penned by the software developers. These initial documents include a small collection of tutorials, including tutorials on building a connectionist network from scratch, on backpropagation networks in Emergent, on the use of temporal difference learning in Emergent, on various new components of the LEABRA framework, and on using some of Emergent's new graphical and multimedia capabilities. Importantly, Emergent contains methods for constructing links in its graphical user interface to web-based resources, allowing the Emergent Wiki to act as a kind of online help system that can be directly engaged from the software.

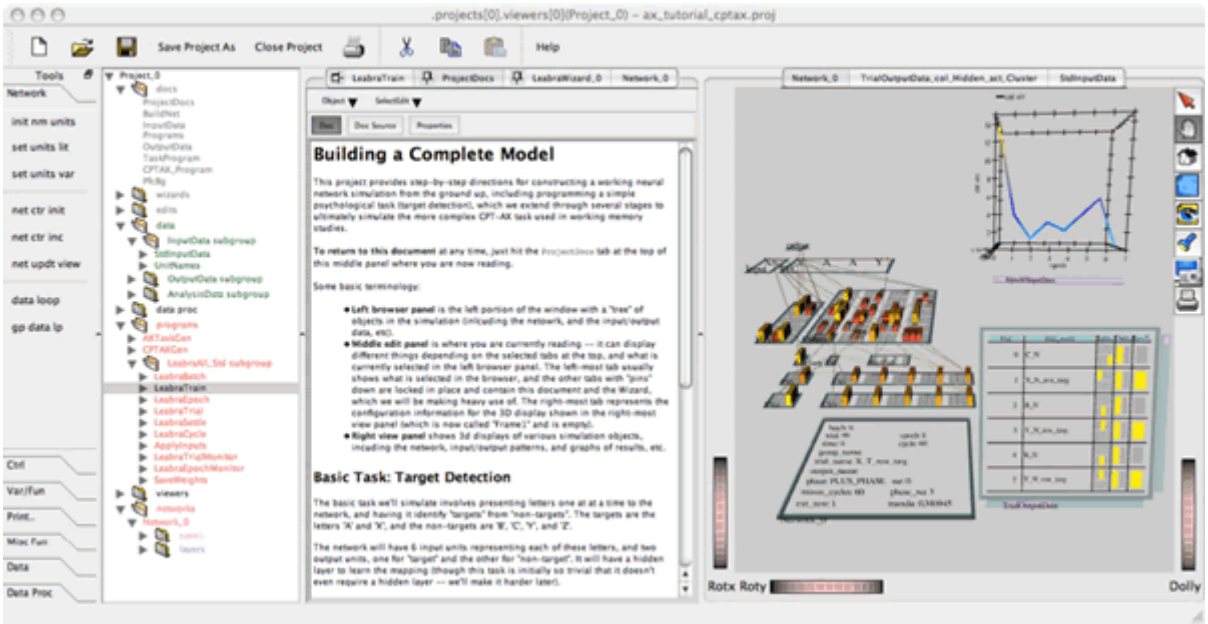

Figure 10: Emergent Project Viewer Window

The foundational structure of PDP++ has not changed much in Emergent, with simulations constructed from collections of objects, such as network objects, environment objects, and the like. Object editing dialogs continue to exist in the new version, offering a graphical means to inspect and modify the various components of any object in the system, as before. The basic organization of the graphical user interface has changed in Emergent, however. In PDP++, separate windows are used for each interaction with the system. For example, editing three different layer objects would typically

produce three object editing windows, one for each layer object. This user interface design can result in a large number of windows without any apparent overarching organization. Emergent attempts to impose some structure on simulation projects by introducing an extensive Project Viewer Window. This large window contains multiple panes, laying out common components of a simulation in a manner that avoids occlusion. Rather than generating many different windows for interacting with various objects, the active dialogs are “stacked” within the panes of the Project Viewer Window and are accessible through “tabs” at the top of each pane. Thus, selecting a “tab” brings the desired dialog to the top of the stack. A menu option allows any of these dialogs to be separated from the Project Viewer Window and placed in its own window, but the default behavior of the interface is to avoid the creation of such stand-alone windows. An example Project Viewer Window is shown in Figure 10. As displayed in that figure, the Project Viewer Window contains three main panes, arrayed from left to right. On the left is a pane that offers functionality similar to that provided by the PDP++ Project View Window (Figure 4). A pallet of tools is offered in this pane alongside a hierarchically organized listing of all of the objects in the project. This listing assists the user in quickly finding any specific object of interest, allowing the object to be inspected or modified. The middle pane contains a stack of control panels, editing dialogs, and other tools for interacting with the current simulation. Importantly, as shown in Figure 10, this pane can include embedded documents, as described below. The rightmost pane is used for graphical displays, including those that previously appeared in the PDP++ Network View Window (Figure 6) and those that previously appeared in various PDP++ log view windows (Figure 9). The graphical capabilities of Emergent are much more extensive than those of PDP++, as discussed below.

One of the most substantive innovations of Emergent, over PDP++, is the inclusion of embedded documents. These documents appear in the Emergent interface as typeset prose, like the content of a web page, but they are written in an easy wiki-like text formatting language. These documents can include active hyperlinks to the web, allowing direct points of contact to online documentation or any other web resource. These documents can also contain active hyperlinks to Emergent objects, and this feature allows the Emergent interface to be directly manipulated by simply selecting links in an embedded document. This is a highly useful feature for the fabrication of pedagogical materials. Prose guiding students through an exercise can now be incorporated into the simulation itself, avoiding the need for shifts of attention between multiple resources. Also, educational scaffolding during early experiences with Emergent can be had by directing students to manipulate initial simulations by simply selecting appropriate document links, rather than forcing them to navigate the system’s control panels. Preparing an embedded document is somewhat easier than producing a simulation specific control panel using CSS, reducing some of the burden on instructors generating demonstrations and exercises. Indeed, penning these embedded documents is sufficiently simple that some instructors expecting to use Emergent in 2008 have suggested requiring that student term project reports take the form of such embedded documents, allowing students to submit both their simulation work and their “write-up” as a single Emergent project file. Support for embedded documents may very well become the most educationally salient new feature of Emergent.

While previous PDP++ releases have provided a variety of means for graphically displaying connectionist models and the results of simulations, Emergent contains a much expanded pallet of visualization tools. The rightmost pane of the project viewer window typically contains a workspace for the display of 3D graphical objects. These objects can include network displays and various graphs and charts, as in PDP++, but they can also include a broader range of objects, including 3D plots, embedded images containing relevant information (e.g., brain imaging data), and even simulations of simple physical environments with which models can interact. A simple mouse-driven interface allows the user to navigate through this graphical workspace, changing viewpoints in order to better examine various components of the display. Graphical objects can also be directly manipulated with the mouse

pointer. The result is a powerful collection of tools for producing engaging and informative dynamic graphical displays of simulation information. While these capabilities are sure to find use in the design of classroom demonstrations and exercises, they do come with some costs. First, the reliance on 3D graphics, using OpenGL[10], increases the platform demands of Emergent. Computer systems without graphics cards that provide hardware support for 3D rendering run Emergent simulations extremely slowly. This is true even for simulations that avoid elaborate graphics, as even simple 2D graphs are rendered as 3D graphical objects in Emergent. Second, it is likely that the attractive graphics capabilities of Emergent will introduce a temptation for some students working on their own simulations, causing them to direct an inappropriate amount of energy toward form over substance.

It is too early to evaluate the pedagogical strengths and weaknesses of Emergent, in comparison to previous PDP++ releases. There is a clear effort to provide further system documentation with this release, mostly through the collaborative construction of the Emergent web site, but the fruits of this effort are still largely forthcoming. Efforts to improve the graphical user interface have produced striking 3D visualization tools which may increase engagement by students and clarify the dynamics of connectionist models, but further classroom experience will be needed to assess the utility of these tools. The organization of the graphical user interface has been greatly simplified in Emergent by gathering disparate PDP++ windows into a single Project Viewer Window. Support for embedded documents will also allow for the generation of easy-to-use interfaces for specific model simulations, and it is likely that such documents will find broad use in communicating model concepts and results within an executable simulation. Active development of this software is ongoing, with electronic mailing lists providing forums for reporting difficulties and discussing prospective improvements.

3 Using PDP++ in the Classroom

3.1 Flexibility, Usability, & Efficiency

PDP++ is primarily designed to be used by cognitive modeling researchers. It offers a powerful array of highly flexible tools to those who are designing and testing new models and new connectionist approaches to the modeling of cognitive phenomena. Those familiar with PDP++ find it fairly easy to use it to generate new models, customize connectionist algorithms, and collect useful data on model behavior.

In classroom settings, using PDP++ trains students in the use of research-grade software. If students are asked to prepare term projects of their own design, it is unlikely that their imaginations will be constrained by software limitations. If students are hoping to construct models after their classroom training is complete, perhaps as part of graduate research activities, they will already be familiar with a software system that can support their research needs. There will be no need for their learning process to start over, moving from a limited pedagogically-friendly system to one with sufficient power for research innovation. Thus, using PDP++ in the classroom can provide valuable practical skills to students who intend to include cognitive modeling work as part of their professional activities.

Unfortunately, flexibility in software often introduces problems with usability. Offering myriads of options for customization can make it difficult for the novice to find the specific controls of interest. PDP++ clearly suffers from this problem of usability, with substantial amounts of training often being necessary before students are comfortable with the important components of the interface. PDP++

[10] OpenGL is a registered trademark of SGI.

tries to overcome usability obstacles in the classroom in two main ways. First, PDP++ provides support for the development of model-specific control panels, allowing instructors to easily construct a graphical interface window that collects only those model parameters and model operations that are important for student learning (see Section 2.3.5). Orienting students to these compact control panels is often sufficient to keep them from becoming lost in the many options available in the full PDP++ interface. Second, in order to assist in the construction of new models of fairly standardized types (e.g., a feedforward multi-layer backpropagation network), recent versions of PDP++ have supported wizard objects. Wizard Windows provide tools which automate the process of constructing standard model components, such as network objects, environment objects, statistics objects, log objects, and process object hierarchies. Buttons on the face of a Wizard Window launch short dialogs with the user, querying for design parameters and then constructing objects according to the specified design. As long as the desired model is moderately prototypical for a given connectionist framework, Wizard Windows can greatly ease the process of creating new models while shielding novice users from the complexities of the full PDP++ interface. In summary, the richness of the PDP++ interface can sometimes make it difficult for students to master, but a number of tools exist for guiding students as they perform common tasks. As long as students remain oriented to these tools, the danger of hindering the learning process by introducing usability frustrations is greatly reduced.

The extreme flexibility of the internal structure of PDP++ also introduces a trade-off with regard to efficiency of simulation execution. Since PDP++ is designed to support a wide variety of connectionist frameworks, it is not optimized for any one approach. Thus, large models may run more slowly than expected, particularly if full support of the graphical user interface is used during execution. This is rarely a problem in classroom environments, where models designed to make instructional points are often small, but issues may arise if students are allowed to design course projects on their own. Students may not understand the computational demands imposed by large models, and it is not uncommon for student project proposals to be overly ambitious. For example, many students show interest in models that operate on photographic images, but they overlook the computational cost of processing such large inputs. While PDP++ provides methods for disabling the graphical user interface in order to speed processing, as well as some support for parallel processing when multiple hardware processors are available, the large time cost of simulating large connectionist networks should be clearly communicated to students considering such large scale models.

3.2 Teaching Connectionism

As a tool for teaching connectionist cognitive modeling skills, PDP++ has many strengths. Training students in the use of PDP++ in a classroom environment provides them with a set of practical research skills, as PDP++ is primarily a research tool. Success with PDP++ in the classroom should provide a good initiation to the use of PDP++ in the laboratory. PDP++ is also well designed to cover the full range of topics contained within typical course curricula on connectionist modeling. Most courses on connectionism include some form of survey of connectionist frameworks, often comparing the relative strengths and weaknesses of various common frameworks. Since PDP++ provides rich support for a wide variety of network architectures and learning algorithms, it can be used to provide students with hands-on experience with all of these connectionist frameworks without demanding that they learn different software interfaces for each framework explored. While the PDP++ graphical user interface is extensive and complex, it contains a number of features that are particularly useful in educational settings. First, the ability to construct simulation-specific control panels allows instructors to guide student exploration of demonstration models, highlighting the aspects of the simulations most important for the current learning goals. Second, the rich and varied graphical displays, including dynamic displays in the Network View Window and the dynamic plotting of data in various graphical

logs, provide students with multiple perspectives on model behavior and performance. Providing multiple views of the same underlying phenomena, in this way, can often provide the key that unlocks the door to a deeper understanding of the mechanisms being simulated.

While PDP++ possesses many pedagogical strengths, it has weaknesses, as well. Perhaps the greatest of these weaknesses is the lack of thorough tutorial materials supporting either the learning of PDP++ or the learning of connectionist modeling concepts from a PDP++ perspective. While the PDP++ users manual (Dawson et al. 1997) contains a wealth of information, it does not supply the kind of step-by-step instruction that is needed to guide students toward mastery of this software system. Ideally, materials for learning PDP++ should be integrated with materials for learning cognitive modeling skills, allowing instructors to directly draw on these materials when designing their own courses. Perhaps all that is needed is a more focused effort to compile teaching materials that have been prepared by instructors who have used PDP++ in the classroom, so that the fruits of these efforts can be shared.

It is worth noting that there does exist a strong tutorial introduction to cognitive modeling concepts using PDP++ in the form of a textbook (O'Reilly and Munakata 2000). This text, which is discussed further in Section 3.3, focuses exclusively on the LEABRA modeling framework, however, so it may not be appropriate for all courses on connectionist cognitive modeling.

Many classes on cognitive modeling initially expose students to prepared demonstration models and only expect students to build models of their own design later in course. This approach allows students to become familiar with the properties of working model simulations before presenting them with the challenges of model construction. In PDP++, this educational sequence introduces some challenges, however. In order to quickly engage students with model simulations, it is common to initially provide extensive guidance and "hand holding" concerning the specific model manipulations to be performed during exercises. This guidance can even be built into simulations by providing model-specific control panel windows that contain only the most relevant operations to be explored. Students often come to depend on these simulation-specific control panels and other forms of guidance, leaving them vulnerable to difficulties when they are later expected to use the standard PDP++ interface to build models of their own design. These difficulties can sometimes be ameliorated through the use of Wizard Windows, which provide generic guidance in the construction of new model components, such as networks and environments. In general, however, extensive use of simulation-specific control panels and wizard objects serves to mask the full PDP++ interface from students, providing them with fewer opportunities to learn the full range of tools provided by the system. Thus, students often stumble when they are expected, or expect themselves, to use the broader graphical user interface. Perhaps the only way to address this problem is through the careful design of a sequence of exercises, slowly removing the scaffolding provided by simulation-specific control panels and wizards as the course develops. In this way, students become accustomed to standard PDP++ interface tools in an incremental fashion.

When students are asked to design their own modeling projects, they often see the design of a connectionist network as the primary task to be accomplished. They rarely recognize the labor required to construct appropriate training and testing environments for their models. In many cases, the construction of environment objects is the most time consuming component of such a term project. This fact should be communicated clearly to students, and students should be encouraged to seek out the easiest methods for environment fabrication. For most students, the easiest way to specify the collection of events that make up an environment is within a plain text file, which can later be loaded into a PDP++ environment object. Students can use their favorite tools (e.g., spreadsheets) to produce such files, without needing to learn additional features of PDP++ (e.g., the CSS scripting language, which could be used to dynamically create events).

PDP++ runs on a variety of platforms, and most pedagogical model demonstrations do not seriously tax modern commodity hardware, including notebook computers. Thus, it is generally reasonable to expect students to install this software on their personal machines, allowing them to explore connectionist modeling within their standard work environments. Perhaps surprisingly, the main limitation that sometimes arises when students use PDP++ on their own computers is not a computational power limitation but a limitation on display size. The graphical user interface of the PDP++ system is composed of many separate windows, and fitting all of the windows of interest on a single desktop screen can become impossible on low resolution displays. Thus, if multiple computers are available to students, they should be encouraged to seek out platforms for their PDP++ work that offer large amounts of display real estate.

In summary, exercises using PDP++ can contribute positively to the teaching of connectionist cognitive modeling skills. The primary challenge that PDP++ presents to instructors of cognitive modeling courses is that of providing adequate guidance in simulator use early in the course, through the use of such tools as simulation-specific control panels, while incrementally removing such scaffolding throughout the course to encourage students to eventually master the full range of powerful tools provided by the PDP++ system.

3.3 Teaching Computational Cognitive Neuroscience

Many courses on connectionist cognitive modeling contain a survey of various connectionist frameworks. Thus, students examine different models, using different algorithms, for each framework. These surveys sometime roughly follows the chronology of the history of connectionist research, but they more often are organized to highlight the similarities and differences between frameworks. This approach to teaching connectionism has a number of strengths, including the way in which it refiects the diversity of connectionist frameworks still being explored in the scientific literature. There is an alternative approach, however.

Connectionist cognitive modeling can also be taught from a more unifying perspective, stressing the ways in which the conceptual contributions of various connectionist frameworks can be brought together in order to share the strengths of these frameworks. This approach has been taken with the LEABRA framework for computational cognitive neuroscience modeling. LEABRA is a collection of computational formalisms for developing cognitive models that make contact with both observable behavior and detailed biological mechanisms. LEABRA models are constrained by our knowledge of processes at the level of membrane channels and individual neural functioning and also by our knowledge of gross brain anatomy and the role of various neurotransmitter systems. From an educational perspective, LEABRA is of particular interest because it incorporates many of the mechanisms that have appeared in the history of connectionist research. Its recurrent activation dynamics allow it to exhibit pattern completion and soft constraint satisfaction performance akin to that seen in Hopfield networks, other attractor networks, and spreading activation models. Synaptic weight learning in LEABRA includes a Hebbian learning algorithm, allowing for self-organization learning, and an error-correction learning algorithm formally related to the backpropagation of error technique. LEABRA networks can also make use of a reinforcement learning algorithm based on the role of the dopamine neurotransmitter system in learning. By bringing all of these mechanisms together, LEABRA provides a single focal framework through which a wide variety of connectionist concepts might be taught. Rather than being required to master a library different connectionist frameworks, students can be encouraged to focus on a single framework while learning about the relative strengths and weaknesses of the various connectionist concepts and mechanisms that contribute to that framework. In addition to focusing students' attention, this unifying approach to teaching connectionism

discourages students from seeing the field as merely a "bag of tricks" by explicitly juxtaposing and relating various connectionist mechanisms within a single framework.

One substantial advantage of adopting this unifying pedagogy is the availability of a rich educational resource that takes this approach. The textbook entitled Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain (O'Reilly and Munakata 2000) provides a rich introduction to computational cognitive neuroscience, including foundational connectionist concepts, using the LEABRA framework. LEABRA is fully supported in PDP++, and the textbook contains an extensive array of PDP++ simulation exercises embedded within expository prose. When a new modeling concept is introduced and discussed, the flow of the text is paused to guide the reader, in a step-by-step fashion, through a PDP++ simulation illustrating the concept at hand. All of the PDP++ project files, and related software, for these exercises are available from the textbook's web site at:

http://psych.colorado.edu/~oreilly/comp_ex_cog_neuro.html

Thus, while focusing on LEABRA, this textbook provides an incremental tutorial introduction to PDP++, computational cognitive neuroscience, and connectionism in the form of accessible prose tightly integrated with hands-on exercises. Example syllabi, lecture slides, and other teaching support materials are also provided at the textbook's web site.

At the time of this writing, at least twenty research universities worldwide have offered courses on computational cognitive neuroscience using this textbook as a primary resource. There is good evidence that this unifying approach to teaching cognitive modeling can equip students with a strong understanding of foundational connectionist concepts while providing them with the practical skills necessary to develop research-grade model simulations using the PDP++ system. The author of this report has taught both survey-structured courses and courses using LEABRA, and these experiences suggest that students are somewhat more engaged by the unified approach and they also welcome the opportunity to relate their modeling explorations to findings from the field of cognitive neuroscience.

4 Summary & Conclusion

This article has provided a brief overview of the PDP++ connectionist cognitive modeling simulation system, with a focus on its role in educating future cognitive modelers. In broad strokes, the primary strength of PDP++ is its flexibility, making it an appropriate tool for cutting edge research in the computational modeling of human behavior and brain function. Thus, teaching students to use PDP++ provides them with skills that transfer directly into research practice. This strength of PDP++ is also one of its main weaknesses, from a pedagogical standpoint, as its flexibility makes the standard PDP++ interface difficult for the novice to master quickly. PDP++ provides some tools to scaffold early learning, however, including the means for instructors to design simple "front ends" or "control panels" for demonstration simulations. Additional tutorial documentation and teaching support materials for PDP++ would greatly improve its utility in the classroom, though an excellent textbook that makes use of PDP++ throughout already exists for those who are willing to take a more unifying approach to the teaching of connectionist concepts.

Several of the recommendations that have appeared in this article warrant highlighting, based on the author's experience using PDP++ in educational settings. These recommendations include:

- Encourage students to install PDP++ on their own computers. Encourage them to select platforms that provide good graphics support, including those offering as much screen real estate as possible. Encourage them to install the software using the available binary packages, rather than recompiling the software from the source code.

- Direct the attention of students to all available system documentation as early as possible. This includes online help tools, such as the descriptions of object members in object editing windows, and the PDP++ reference manual.

- Construct compact “control panels” for early simulation demonstrations and exercises, hiding the complexities of the PDP++ interface during initial exposure to the software.

- Incrementally remove the scaffolding of provided simulation specific “control panels” as the course advances, introducing components of the general interface, such as process objects and their respective generic control panels, one by one. This is important if students are to build simulations of their own design.

- Encourage students to regularly disable the graphical display of results while large simulations are running, allowing the simulations to run faster. Encourage them to examine simulation results in the form of logs, such as graphs and tables, once simulation runs are complete.

- Warn students of the extensive labor needed to construct environments for connectionist simulations. Encourage them to generate large collections of events using tools with which they are familiar, such as spreadsheets, loading them into PDP++ as plain text files.

- Warn students of the time cost of running large simulations. In particular, be wary of projects involving the use of real visual image data.

- Discourage students from writing C++ augmentations to PDP++ as part of course term projects, directing them to the CSS scripting language, instead.

Every educational forum is different, and conscientious instructors should certainly adapt their teaching activities to the particular backgrounds and needs of their students. Still, these recommendations have been supported by experiences in a variety of settings, ranging from the university classroom to academic conference tutorials.

At the time of this writing, PDP++ is an active ongoing open source software project, though it has taken a new name, Emergent, in its most recent instantiation. Emergent offers several improvements to the ease-of-use of the system’s graphical user interface, and technical support for further system documentation, integrated with the software, is included. As the community of instructors using PDP++ in their classrooms grows, it would be useful to compile teaching resources in a central repository associated with the PDP++ web site. These resources should include both documents, such as syllabi and lecture slides, and also PDP++ project files containing carefully packaged and pedagogically relevant simulation exercises. In the end, such auxiliary materials are as critical to teaching success as the properties of the educational software, itself.

References

Almeida LB (1987). A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. In Caudil, M. and Butler, C., editors, Proceedings of the IEEE First International Conference on Neural Networks, volume 2, pages 609–618, New York. IEEE.

Amit DJ (1989). Modeling Brain Function: The World of Attractor Neural Networks. Cambridge University Press.

Dawson CK, O'Reilly, R. C., and McClelland, J. L. (1997). The PDP++ Software Users Manual. Version 1.2.

Elman JL (1990). Finding structure in time. Cognitive Science, 14(2):179–211.

Grossberg S (1976a). Adaptive pattern classification and universal recoding I: Parallel development and coding of neural feature detectors. Biological Cybernetics, 23:121–134.

Grossberg S (1976b). Adaptive pattern classification and universal recoding II: Feedback, expectation, olfaction, and illusions. Biological Cybernetics, 23:187–202.

Grossberg S (1987). Competitive learning: From interactive activation to adaptive resonance. Cognitive Science, 11:23–63.

Grossberg S (1988). Nonlinear neural networks: Principles, mechanisms, and architectures. Neural Networks, 1:17–61.

Grossberg S (1998). Birth of a learning law. INNS/ENNS/JNNS Newsletter, 21:1–4. appearing with Neural Networks, 11(1).Hebb DO (1949). The Organization of Behavior. Wiley, New York.

Hecht-Nielsen R (1989). Neurocomputing. Addison-Wesley, New York.

Hochreiter S and Schmidhuber J (1997). Long short-term memory. Neural Computation, 9:1735–1780.

Hopfield JJ (1982). Neural networks and physical systems with emergent collective computational abilities. In Proceedings of the National Academy of Sciences, volume 79, pages 2554–2558, Washington, D.C.

Kohonen T (2001). Self-Organzing Maps, volume 30 of Springer Series in Information Sciences. Springer, New York, third edition.

McClelland JL and Rumelhart DE (1981). An interactive activation model of context effects in letter perception: Part 1. an account of basic findings. Psychological Review, 88:375–407.

McClelland JL and Rumelhart DE (1988). Explorations in Parallel Distributed Processing. MIT Press, Cambridge, Massachusetts.

McClelland JL, Rumelhart DE, and the PDP Research Group (1986). Parallel Distributed Processing: Explorations in the Microstructure of Cognition, volume 2. MIT Press, Cambridge, Massachusetts.

O'Reilly RC and Munakata Y (2000). Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain. MIT Press, Cambridge, Massachusetts.

Pineda FJ (1989). Recurrent backpropagation and the dynamical approach to adaptive neural computation. Neural Computation, 1(2):161–172.

Rumelhart DE, Hinton GE, and Williams RJ (1986a). Learning internal representations by error propagation. In (Rumelhart et al., 1986b), chapter 8, pages 318–362.

Rumelhart DE, McClelland JL, and the PDP Research Group (1986b). Parallel Distributed Processing: Explorations in the Microstructure of Cognition, volume 1. MIT Press, Cambridge, Massachusetts.

Rumelhart DE and Zipser D (1986). Feature discovery by competitive learning. In (Rumelhart et al., 1986b), chapter 5, pages 151–193.

Von der Malsburg C (1973). Self-organization of orientation sensitive cells in the striate cortex. Kybernetik, 14:85–100.