LENS: The light, efficient network simulator

Douglas L. T. Rohde

August, 1999
CMU-CS-99-164

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

LENS is a neural network simulator designed for speed and ease of customization. On large networks, LENS is several times faster than most commonly used simulators. Although intended primarily for backpropagation networks, it also currently supports deterministic Boltzmann machines and Kohonen networks and could easily be extended to other Hebbian or Bayesian models. LENS is written entirely in the C and Tcl languages and operates on both Unix and Windows platforms. This report gives a brief overview of LENS and describes some of the interesting aspects of its design.

Special thanks to David C. Plaut, Michael Harm, and other members of the Center for the Neural Basis of Cognition for help with the design and debugging of LENS. This research was partially supported by NIMH Program Project Grant MH47566 Part 1 (M. Seidenberg, M. MacDonald, D. Plaut co-PIs; J. McClelland, PD) and an NSF Graduate Fellowship. Correspondence may be sent to Douglas Rohde (dr@cs.cmu.edu), School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213–3890, USA.

Keywords: neural network, simulator, backpropagation, Carnegie Mellon University, Lens

1 Introduction

Neural network simulators tend to be of two basic varieties: very simple, fast programs designed for a specific type of network on the one hand, and large, graphically intensive systems targeted at diverse users on the other. Although generally quite fast, the former suffer primarily from inflexibility. They are not easily extended to new network architectures. This increases development time for new simulations and prevents multiple users from sharing a single platform, which hinders collaboration and verification of results. Without a sufficient command language, simple simulators are typically limited to a few command-line instructions and must be changed at the source code level and recompiled to perform more complex experiments. Without adequate visualization tools, understanding and debugging a network can be very difficult. A final drawback of simple simulators is that they are often less accessible to non-programmers and are thus not good platforms for introductory courses or for broad publication of methods.

On the other end of the spectrum, large-scale systems attempt to provide enough flexibility to satisfy most user's needs. However, it is quite impossible to anticipate every feature that might be desired and, ultimately, sophisticated users will need to modify the source code. This is not easy in most complex programs. Even if the code can be understood and modified, without a convenient method of dissociating the original code from a user's changes, those changes will have to be reapplied by hand to any new releases.

LENS was designed to fill a middle ground between large and small simulators, with three primary goals in mind:

1. **Speed**: The development of ever faster machines does not reduce the need for an efficient simulator. Speed results from fast inner loops and conservative memory use, with particular attention to cache performance.

2. **Flexibility**: A scripting language and large command set allows most experiments to be performed without the need to modify source code. Unit behaviors can be composed from several input, transfer, integration, and noise functions, providing a wide variety of standard unit types.

3. **Customizability**: When it is necessary to make modifications to the source, it can be done with minimal interaction between generic and user code by registering new types and functions and creating new shell commands.

LENS is primarily a backpropagation simulator, designed for feed-forward, simple-recurrent, backprop-through-time, and fully recurrent networks. However, the basic network framework can be easily adapted to other models and deterministic Boltzmann machines and Kohonen networks have been implemented as well. LENS operates on most Unix platforms and has recently been ported to Microsoft Windows. It was written in C and uses the Tcl/Tk libraries to support graphics and a shell interface.

Section 2 of this document compares the performance of LENS to several other popular backpropagation simulators. Section 3 explains some of the optimizations that lead to its good performance. Section 4 briefly describes facilities for training on multiple machines in parallel. Section 5 explains some of the principles that ease customization of LENS and Section 6 talks a bit about its user interface.

2 Performance Benchmarks

LENS[1] has been benchmarked along with five other commonly used, non-commercial simulators: SNNS[2], UTS[3], PDP++[4], RCS[5], and TLEARN[6]. The simulators performed backpropagation training using momentum descent on a feed-forward network having two hidden layers. The input and output layers each contained four units. The two hidden layers were approximately equal in size and were adjusted to control the total number of links in the network, which ranged from 100 to 1 million. The training set consisted of 40 random patterns and, where possible, batch learning was used. Thus, 40 forward and backward passes were performed before each weight update. All simulations were run on an unloaded 450 MHz Pentium II.

[1] LENS, v. 2.02, was run in batch mode. It is available at http://www.cs.cmu.edu/~dr/Lens.

[2] SNNS, v. 4.2, is the Stuttgarter Neural Network Simulator from the University of Tuebingen, Germany. Run with the batchman program using the BackpropMomentum learning function. It is available at http://www-ra.informatik.uni-tuebingen.de/SNNS.

[3] UTS, v. 4.1p1, is the sequel to XERION, and was developed at the University of Toronto. It is available at ftp://ftp.cs.toronto.edu/pub/xerion.

[4] PDP++, v. 1.2, was developed at Carnegie Mellon University. bp++ was run in -nogui mode. It is available at http://www.cnbc.cmu.edu/PDP++/PDP++.html.

[5] RCS, v. 4.2, is the Rochester Connectionist Simulator, developed at the University of Rochester. It was run using 6 settling steps and online learning. It is available at ftp://ftp.cs.rochester.edu/pub/packages/simulator.

[6] TLEARN, v. 1.0, was developed at the University of California, San Diego. It is available at http://crl.ucsd.edu/innate/tlearn.html.

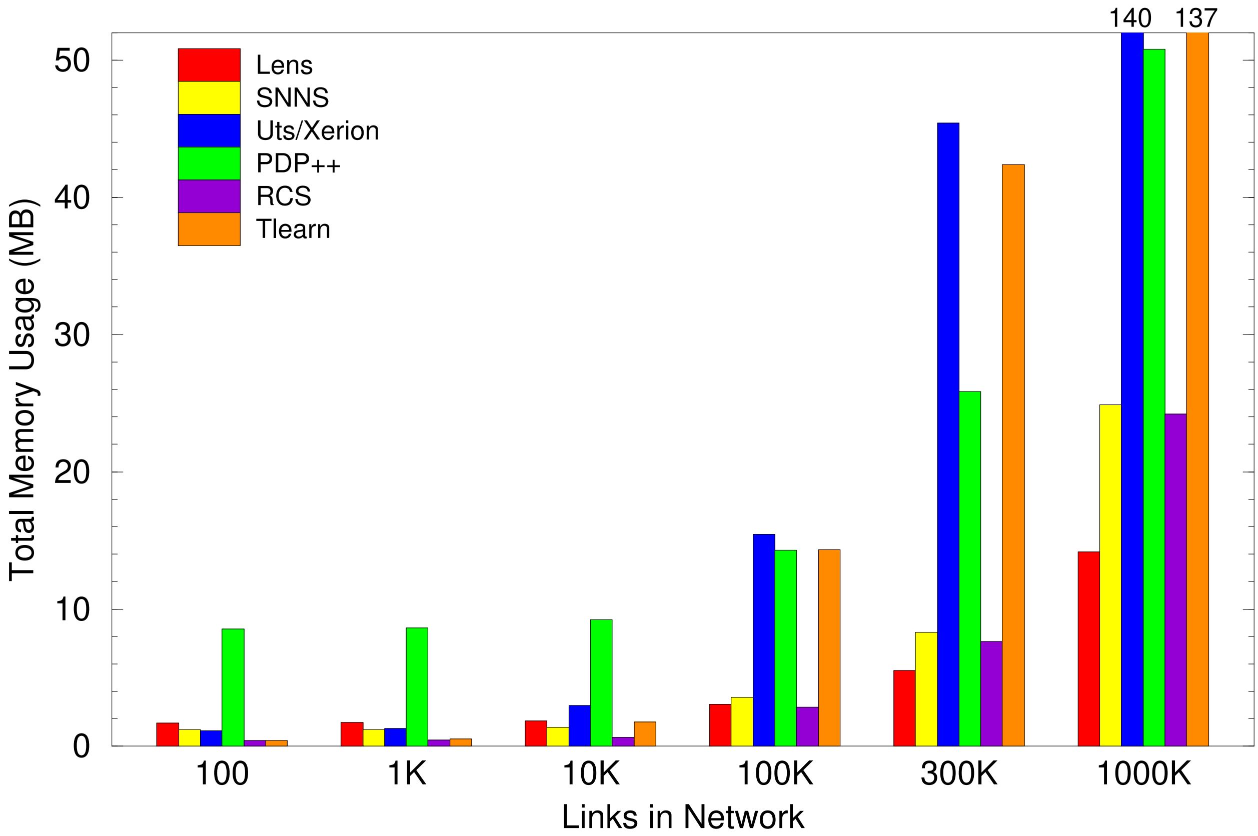

Figure 1: Total memory usage with networks of varying size.

2.1 Memory Usage

With medium to large networks, simulators are memory-intensive applications and memory use is a critical factor in their performance. Time spent on memory operations can dominate that spent on floating point operations. Although small structures allow larger networks to fit into main memory, a reasonable simulator will not push the limits of current machines even with a network having several million links. A more important result of conservative memory use is its contribution to cache performance. Simulator speed is primarily bounded by the rate at which the program can cycle through the links. Using small link representations leads to fewer cache misses and greater speed. Due to interactions between the network's working set size, the sizes of primary and secondary caches, and the cache replacement protocols, simulator speed is not a linear function of network size and is not easily predicted based on the machine's floating point performance.

Figure 1 shows the memory requirements of the six simulators. With small networks, the memory is almost entirely due to simulator overhead. Next to PDP++, which has a very large profile, LENS has the second highest at just under 1.7 MB. Nevertheless, the overhead of most programs does not affect their working set size and memory usage is not a critical factor on small networks.

As the networks grow, memory devoted to link structures dominates. Here the important differences between the simulators becomes evident. LENS uses 3 (4-byte) words per link: one for its weight, one for its error derivative, and one for the previous weight change, which contributes the momentum term.[7] SNNS and RCS each appear to use 6 words, which is reasonable. PDP++, TLEARN, and UTS, on the other hand, use roughly 10, 34, and 35 words, respectively.

Although not a factor in these experiments, the amount of memory devoted to data sets can also be an important issue. Experiments on language can involve corpora consisting of hundreds of thousands or millions of examples. LENS is able to reduce the memory and time required for example sets by using a mixture of dense and sparse representations, loading examples on-the-fly from a file or pipeline, and drawing examples from biased distributions.

[7] LENS can also be compiled with flags that create a fourth field for each link, which allows the delta-bar-delta and quick-prop algorithms to be used.

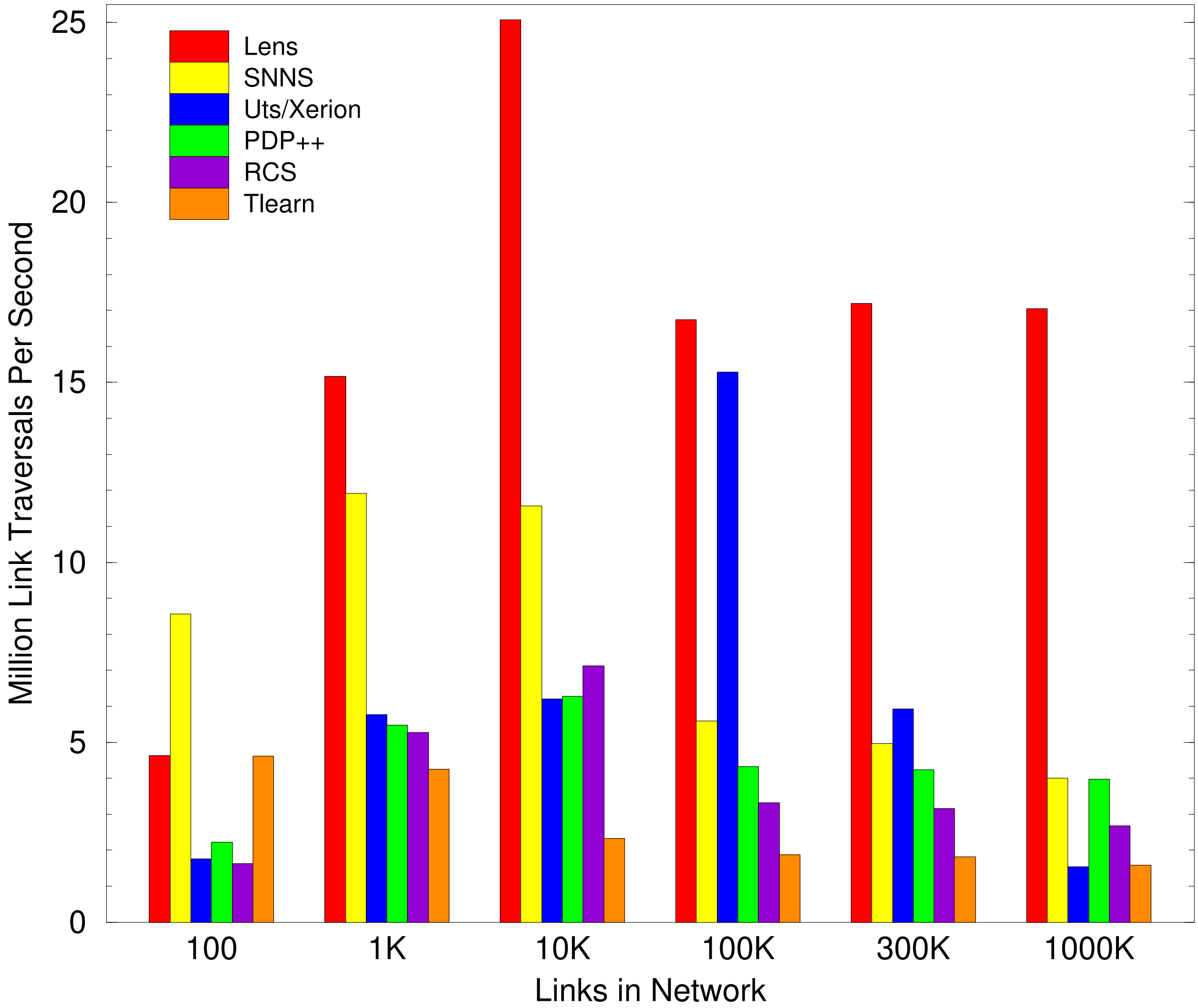

Figure 2: Speed as measured in millions of link traversals per second during training of a feed-forward network.

2.2 Speed

Simulator speed was evaluated by training the networks for a number of weight updates equal to 10 million over the number of links in the network. A *link traversal* occurred for each link once in the forward pass, once in the backward pass, and once during a weight update. Thus, all networks were trained for 810 million link traversals. Training time included starting and stopping the simulators. Three trials were run for each simulator on each size of network and the best of the three times was used. The results are shown in Figure 2.

On the smallest network, LENS performs only moderately well. Tlearn does as well and SNNS is nearly twice as fast due to smaller overhead outside of the inner link-traversal loop. However, LENS was not designed for networks of this size. On larger networks, which are increasingly used in connectionist modeling, it is the clear winner. At 300,000 links LENS is over three times faster than the competitors and at a million links it is over four times faster.

Each simulator tends to have a sweet spot at which the relatively lower overhead of a large network and the better cache performance of a small network result in optimal link traversal rate. On this machine, LENS appears to have a peak at around 10,000 weights, while UTS has a surprisingly strong sweet spot at 100,000 links. These are replicable on similar machines but may shift significantly on other architectures.

3 Optimizations

This section discusses some of the design principles and optimizations that result in the good performance of LENS on medium to large networks. To begin with, units in a network are organized into groups. The groups tend to correspond directly to layers of the network. All units in a group are of the same type, using the same input and output functions. In most models, units calculate their input as the dot-product of the incoming weight vector and the vector of corresponding unit activations and their output as the sigmoid of their input. In the backward pass the reverse processes occur. The unit calculates its input derivative (the derivative of the error w.r.t. the unit's input) as the product of its output derivative and the derivative of the input w.r.t. the output. It then sends error derivatives back to its incoming links and to the output derivatives of the sending units. Computing the unit inputs and sending back the input derivatives comprise the inner loops of backpropagation training and they will typically use around 99% of the processing time on a large network.

In optimizing the inner loops, it is important to minimize the number of memory accesses they require by properly structuring the link representations. There are two basic ways to organize the links: They could be controlled by their sending unit (the one from which the link projects) and activation "pushed" over the link, or links could be controlled by the receiving unit and activation "pulled". LENS uses the latter method. Each unit maintains the arrays of its incoming links but has no direct access to its outgoing links. During the forward pass, as a unit traverses its incoming links it need only access the link weight and the sending unit activation from memory. The accumulating input value can remain in a register. Thus, there are just two memory reads per link. If links were owned by the sender and the values were pushed to the receivers, the output of the sending unit could remain in register but the input to the receiving unit would have to be retrieved from memory, incremented, and then stored back, requiring an extra memory access.

Because the values propagated on the backward pass depend on the receiving unit's input combining function, which need not be a dot-product, it is easier for the receiving unit to push the derivatives backward over its incoming links. This involves six memory accesses: the link weight and the sending unit output are loaded and the link derivative and sending unit output derivative are incremented. If one knew that all receiving units used the dot-product, this could be reduced to 5 accesses by having the sending unit pull values backward across its outgoing links, but this would likely result in worse cache performance.

Although a receiving unit could get projections from several different groups or receive sparse inputs, projections tend to arise from a few blocks of consecutive units. LENS uses this block structure to its advantage. Because we know, by definition, that all sending units in a block are consecutive and belong to the same group, if those units are allocated in a single array of memory we could simply walk down the array as we access the output or output derivative of each unit.

The four critical values that must be retrieved in the backward pass are the links' weights and derivatives and the sending units' outputs and output derivatives. To get good performance, we would like to be able to fill the cache with as many of these values as possible. Therefore, the weights and derivatives of incoming links to each unit are stored in a single array. The last weight change and any other less critical values for the link are stored in a separate array. Rather than accessing the output and output derivative directly from the sending unit structures, which would involve loading the entire sending unit array into the cache, we keep separate arrays for each group that just contain a copy of the units' outputs or output derivatives. This minimizes the amount of non-critical information that enters the cache.

To further improve the speed, the inner loops are unrolled ten times by hand, which is noticeably better than any unrolling that may be performed by most optimizing compilers.

Finally, although unit-level costs are relatively insignificant for large networks, some improvement can be gained by using a fast sigmoid function. In place of the straight-forward sigmoid, which requires an exponentiation and a division, LENS uses a lookup table of 32K values and performs linear interpolation between values. This is accurate to within 2×10^{-7}. Although the fast sigmoid can speed up small networks by 15-20%, its effect on large networks is minimal.

4 Parallel Training

LENS provides utilities for training networks in parallel on multiple machines. Parallel training is at the batch-level. That is, the network itself is not partitioned among machines. Each machine has a complete copy of the network and its own example set. In order for parallel training to be effective, the batch size must be large enough that the work can be partitioned among the clients without the overhead of communication dominating any benefits of parallelism.

Nevertheless, some networks do see a performance advantage. There are two different forms of parallel training: synchronous and asynchronous.

Synchronous training is functionally equivalent to single-processor batch learning. At the start of each batch, the server sends a copy of the network's link weights to each of the clients and tells each client how many examples to process. The clients run the network on the assigned examples, accumulating link derivatives, and then ship the link derivatives back to the server. When the server has summed the derivatives from each of the clients, the weights are updated and the process repeats. A potential drawback of synchronous training is that it is only as fast as the slowest client. However, LENS maintains a running estimate of the speed of each client and adjusts the assignments so that all machines complete in approximately the same amount of time. A more serious drawback is that the clients are idle for the time it takes the server to update the weights and either send or receive from the other clients.

In the second form, asynchronous training, each client is given the same size batch of examples to run. When a client is done, it returns the derivatives and the server immediately updates the weights and sends back the new weight information. The primary advantage is that clients need not wait for one another unless the server is really overloaded. However, the drawback is that the clients are working with slightly different versions of the network and training can be unstable at high learning rates or with large batch sizes.

Although parallel training is useful when a single network must be trained as quickly as possible and machines are available, it is less useful than one might expect. It is typically the case when working with neural networks that a range of network or training parameters must be searched to find the best performance. Therefore, users rarely want to run just one network. In this case, it is more efficient to simply devote each machine to its own network, thus obtaining perfect parallelism.

5 Customization

The novel way in which LENS organizes unit-level functions provides considerable flexibility without the need to modify the program. Operations on units are divided into three main classes: procedures for computing the unit input, for computing the unit output, and for attributing cost or error directly to the unit. For each of these operations, the type of the unit's group defines a pipeline of simple procedures. The procedures can be combined to produce various behaviors.

For example, most units will have a basic output procedure, such as linear, sigmoidal, or exponential, which determines the output as a function of the input. Once that is computed, a secondary procedure in the output pipeline might inject noise. Another procedure could then integrate the unit's output over time or normalize the outputs across a group. Each procedure has a corresponding inverse procedure which operates in the backward pass to compute the unit's input derivative from its output derivative. Without the ability to combine simple operations in this way, one would have to create many more group types to handle all reasonable combinations of simple procedures.

However, no simulator can satisfy all users and many modelers will eventually need to get inside and make their own changes. In most simulators, changes would need to be made in various places throughout the code, which leads to problems whenever a new version is released. LENS was largely born out of frustration with the difficulty of tracing and modifying the code in other simulators. To ease customization, LENS provides an *extension* module, in which user modifications can be contained. Three main features of its design facilitate encapsulation of changes: extension structures are provided to allow the user to augment the major structures, network behavior is controlled by a modifiable hierarchy of function pointers, and new function types for controlling various aspects of the simulator can be registered to make them easily accessible from the command interface.

The network maintains pointers to functions for such actions as training for a number of weight updates, training on a single batch of examples, training on an example, training on an event within an example, and so forth. The functions tend to become simpler as we descend the hierarchy and each function typically uses the pointer to the one below it. Many changes to network behavior can be made by replacing a single network function, minimizing the amount of new code that must be written and enabling the new code to remain in the extension module. Having written a new function, the user might then create a shell command that causes the network to use the new function in place of the old one.

However, to make new network or group types more naturally accessible from the shell interface, types can be *registered*. This creates a name by which the user will be able to refer to that type, making it equivalent to the built-in types. Rather than creating a special shell command, a new network type could be created and an initialization procedure defined to configure any new networks of that type. Customizations that currently may be registered include

```
#define SINE_OUT ((mask) 1 << 20)

static void sineOutput(Group G, GroupProc P) {
  FOR_EACH_UNIT(G, U->output = sin(U->input););
}
static void sineOutputBack(Group G, GroupProc P) {
  FOR_EACH_UNIT(G, U->inputDeriv = U->outputDeriv * cos(U->input));
}
static void sineOutputInit(Group G, GroupProc P) {
  P->forwardProc  = sineOutput;
  P->backwardProc = sineOutputBack;
}
flag userInit(void) {
  registerGroupType("SINE_OUT", SINE_OUT, GROUP_OUTPUT, sineOutputInit);
  return TCL_OK;
}
```

Figure 3: The code to add a custom unit output function.

additions of basic network types, unit input, output, and cost functions, algorithms for updating the weights, for selecting the next example, and for creating link projection patterns. Figure 3 contains all of the code necessary to create and register a new unit output function that computes the sine of the input.

6 Interface

The primary interface to LENS is a Tcl/Tk-based command language. The user can either enter commands to a shell or run programs out of script files. Over 120 commands are currently available allowing the user to, among other things, build and lesion networks, save and load example and weight files, describe the network layout, and, of course, train and test. New procedures can be written in Tcl, which are especially helpful in running experiments. Commands can also be written in C and compiled if speed is an issue. Finally, the fields in the C structures of the network and example sets can be accessed from the shell.

The design of LENS was guided by the philosophy that common things should be easy and difficult things should be possible. One useful feature is the relative ease with which networks can be constructed. Most feed-forward and simple-recurrent networks can be described with a single command. For example, the command:

```
addNet myNet 10 20 ELMAN 5 SOFT_MAX
```

would create a simple-recurrent network with 10 input units, a 20-unit hidden layer with corresponding context layer, and a 5-unit output layer which uses a soft-max constraint and the appropriate divergence error measure. The input and context groups project to the hidden layer which projects to the output layer. To build the same network, other simulators might require that six or more commands be issued, a C program be written and compiled, or that the network be constructed by pointing and clicking on a graphical interface. If more complicated networks are desired in LENS, such as ones that use non-simple recurrence or sparse connectivity, the network can be partially built with `addNet` and then extended piece-by-piece.

The script language makes it possible to parameterize aspects of network building. For example, a procedure might be defined to create a network with a specified number of hidden units, thus making it easy to experiment with different architectures. Some simulators require that a separate file be created for each architecture, which can be a serious hindrance.

6.1 Displays

Although LENS can be operated using only the shell, graphical interfaces are convenient for providing better visualization and quick access to common operations. Some of the LENS displays are shown in Figure 4. By default, a main window, which gives access to the most useful commands and training parameters, is opened. The eight panels in the main window can be individually hidden to conserve screen space. A shell console window is also optional and provides a nicer command-line environment than the basic Tcl shell, including the ability to edit commands, traverse

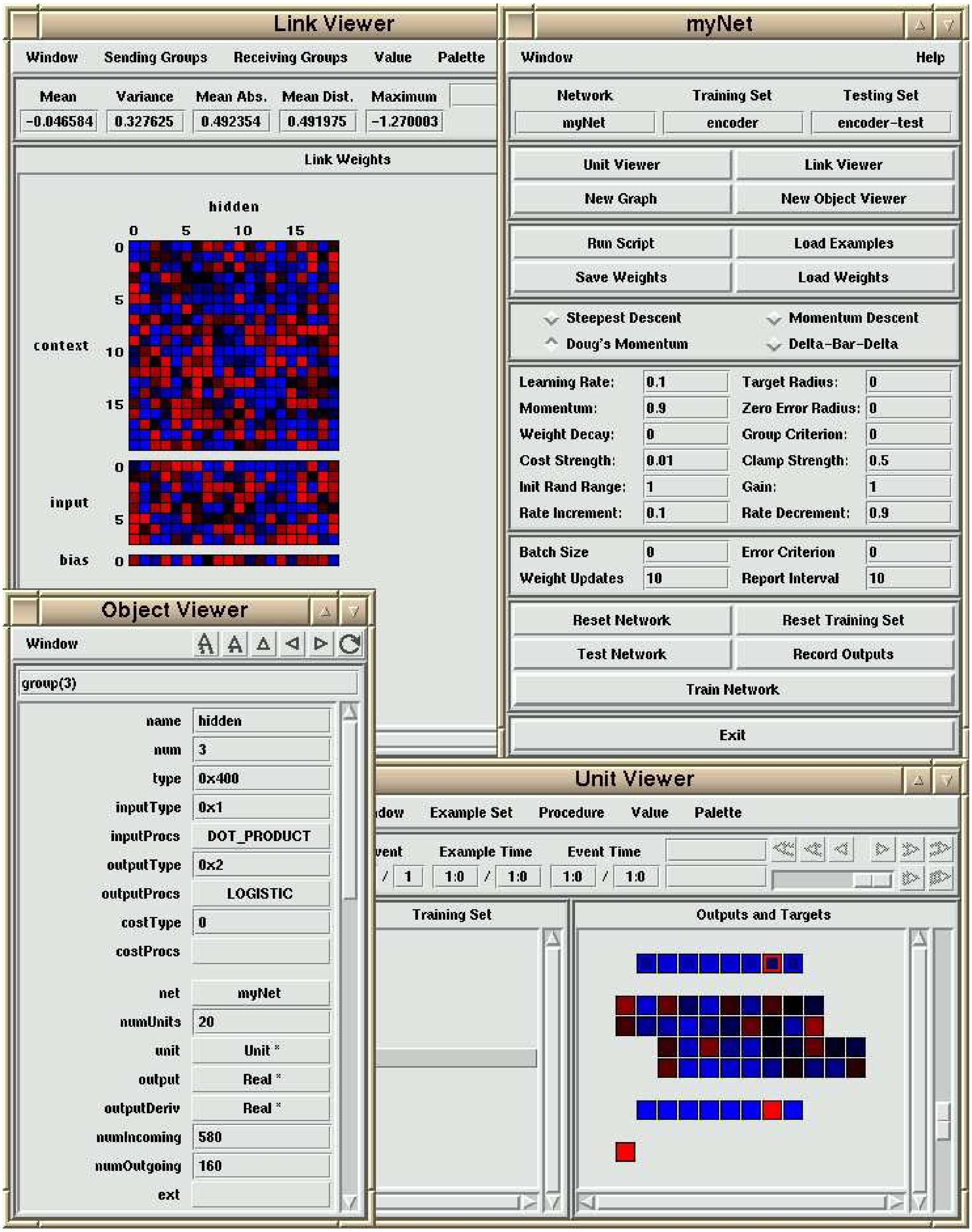

Figure 4: A collection of LENS displays.

the command history, perform command- and file-name completion, and execute commands while LENS is busy. The *object viewer* allows the user to view and edit the C data structures which represent the networks and example sets and their components. Where appropriate, fields in the C structures can be hidden or write-protected.

The *unit viewer* is perhaps the most useful display. It shows the training or testing examples and the activations, inputs, derivatives, or other values associated with the units or the links projecting to or from a single unit. This is helpful in observing the behavior of the network and quickly diagnosing problems. By default, the layout of the network in this window will be created automatically, but commands are also provided for customizing the network representation. The *link viewer* depicts the values associated with some or all of the links and calculates summary statistics. Finally, any real-valued field, typically the network's error, may be graphed over time.

7 Conclusion

LENS is a fast, flexible neural network simulator with the potential to satisfy the needs of a wide variety of users. Although currently used mainly by experienced modelers, the relatively straightforward interface, the ease of creating new simulations, and the ability to run on a variety of platforms make it well-suited for use in introductory courses. LENS is available free-of-charge to those teaching or conducting research at academic institutions. The complete manual and installation instructions can be found on the web at

http://www.cs.cmu.edu/~dr/Lens