A Concurrent Object-Oriented Framework for the Simulation of Neural Networks

A. Weitzenfeld and M.A. Arbib
Center for Neural Engineering
University of Southern California
Los Angeles, CA 90089-2520
alfredo@rana.usc.edu and arbib@pollux.usc.edu

Extended Abstract

This paper discusses the issues in simulating neural networks using an object-oriented concurrent programming framework, based on our experience in developing two generations of the NSL (Neural Simulation Language) simulation system. The second generation simulation system, NSL 2.0, was designed and implemented utilizing object-oriented programming concepts. We close with future design and implementation directions.

Neural Networks as Concurrent Object-Oriented Structures

Our group has approached brain modeling and neural engineering at two levels, a top-down functional analysis in terms of computing agents called *schemas*, and a bottom-up analysis of how interacting schemas may be implemented in neural networks [Arbib 1981, 1987]. We have developed programming languages for both schemas [Lyons and Arbib 1989] and for neural networks [Weitzenfeld 1989, 1990]. In developing a unified environment for schemas and neural networks, we have noted a convergence of schema theory with recent work in object-oriented concurrent programming [e.g., Agha 1986, Agha and Hewitt 1987, Yonezawa et al. 1987], and have also found a great similarity between the requirements of neural network simulation and object-oriented programming. It is the latter theme that we emphasize in the present article. We start by describing neural network concepts in terms of such concepts of object-oriented concurrent languages as *objects*, *instantiation*, *inheritance*, *message-passing*, and *concurrency*.

1) Neurons are self-contained *objects*.
2) In creating a neural model, neurons are *instantiated* from a uniform and basic class structure.
3) Neural networks are described as a hierarchy of modular objects where higher level structures *inherit* the characteristics of lower level ones.
4) Communication among neurons is a form of *message passing*.
5) Neural networks are by nature *concurrent* processes.

Other requirements that we have tried to achieve with the incorporation of object-oriented programming are the following:

6) Models should be easy to describe.
7) The language should be able to describe neural network models in any application area.
8) The language should be as general as possible, and should not enforce a particular level of internal neural detail.
9) The model should be as independent as possible from a particular numerical analysis technique or learning algorithm.

Neural Network Simulation

A neural network is, in our abstraction, a set of simple concurrently processing units (neurons), connected in a particular topology, sending small amounts of information to other units along each (directed) connection. Therefore, a model description includes (1) declarations of the units in the model, (2) connections between the units, and (3) descriptions of inputs external to the network.

A neuron may receive input from many different neurons, while it has only a single output. We will focus on neurons whose internal state is described by a single scalar quantity, its membrane potential m, whose time course τ_m is described by a differential equation

$$\tau_m \frac{dm(t)}{dt} = f(S_m, m, t)$$

depending on its input vector S_m. The choice of f defines the particular neural model utilized, including the dependence of m on the neuron's previous history.

The *firing rate* or output of the neuron, M, is obtained by applying some "threshold function" to the neuron's membrane potential,

$$M(t) = \sigma(m(t))$$

where σ is usually a non-linear sigmoid saturation function.

When building neural networks, the output of a neuron serves as input to other neurons. Links among neurons carry a connection weight which describes how neurons affect each other. Links are called excitatory or inhibitory depending on whether the weight is positive or negative. The most common formula for the input to a neuron is

$$\sum_{i=1}^{n} w_i \, M_i(t)$$

where $M_i(t)$ is the firing rate of the neuron whose output is connected to the i^{th} input line to the neuron, and w_i is the weight on that link.

It is important to realize that neurons may be modelled with different levels of detail, from the sophisticated biophysical models to simple binary models where the neuron is either *on* or *off* at each time step. Thus neurons are best suited to be treated as *objects* whose internal details are completely hidden away from the rest of the network.

A neural network can also be treated as a whole, as one single object, which may communicate externally through input and output ports. The neural network internal connectivity is completely hidden away. By putting together neurons we have shown how neural networks are built. Neural networks can themselves be inter-connected in a hierarchical way, creating higher level structures known as neural network *assemblages*. These assemblages have the power to perform very sophisticated tasks while keeping a modular design.

Different languages have been written for the simulation of neural networks, with different modelling capabilities, and for different application areas [Goddard et al. 1987, Paik and Skrzypek 1987, Teeters 1989, Wilson et al. 1989, Wang and Hsu 1990]. In our particular simulation language, NSL [Weitzenfeld 1989, 1990], we describe neural connectivity in the form of mathematical equations, where the assignment operator is utilized to describe connectivity in the network.

Layers and Masks

As part of our modelling primitives, we have extended the basic neuron abstraction into neuron layers and connection masks, describing spatial arrangements among neurons and their connections, respectively. The reason for defining such abstractions is that, in the brain as well as in many neural engineering applications, we often find neural networks structured into two-dimensional layers, with regular connection patterns between various layers.

The computational advantage of introducing such concepts when describing a neural network is that neural layers and interconnection masks can then be concisely described as higher level data structures. Instead of describing neurons on a one by one basis, a layer can be described as an array and, similarly, the connections between layers can be described by a mask storing synaptic weights. An interconnection among neurons would then be processed by computing a spatial convolution of a mask and a layer. For example, if A represents an array of outputs from one layer of neurons, and B represents the array of inputs to another layer, and if the mask $W(k,l)$ (for $-d \le k,l \le d$) represents the synaptic weight from the $A(i+k,j+l)$ (for $-d \le k,l \le d$) elements to $B(i,j)$ element for each i and j, we then have

$$B(i,j) = \sum_{k=-d}^{d} \sum_{l=-d}^{d} W(k,l)A(i+k,j+l)$$

which can be expressed by the single array operation of convolution

$$B = W*A$$

giving greater computing power to a simple descriptive expression.

Numerical Methods

Up until now we have left out the internal neural model detail. One widely used neuron model is the 'leaky integrator', whose membrane potential is described by the following differential equation

$$\tau_m \frac{dm(t)}{dt} = - m(t) + S_m(t)$$

where $S_m(t)$ incorporates the input from all the other cells, and τ_m is the time constant. The overall dynamics will depend upon the actual choice of excitatory and inhibitory weights in each $S_m(t)$ and of the time constants.

While different numerical methods may be used to solve a particular neuron model, the neural network architecture and connection weights should *not* have to change depending on this. For this reason we treat numerical methods as *orthogonal* object classes totally independent from network specification. A numerical method is instantiated only after the neural network has been completely described. Different numerical methods keep on evolving and they may be more appropriate according to the sophistication of the model and the processing power of the computing machine.

A common characteristic of the various neuron models, and the corresponding numerical methods describing them, is that they are processed as *continuous* time systems, but with the estimate of the state of each neuron updated at constant time steps Δt. This gives rise to *synchronous* message passing as means of communication among neurons.

As an example consider two different methods, the 'Euler' method and 'interpolation' method, for solving the previous differential equation. The 'Euler' method replaces the differential equation by

$$m(t+\Delta t) = (1 - \frac{\Delta t}{\tau_m})\, m(t) + \frac{\Delta t}{\tau_m}\, S_m(t)$$

where Δt is the integration time step, while the 'interpolation' method replaces the differential equation by

$$m(t+\Delta t) = (p)\, m(t) + (1 - p)\, S_m(t)$$

where $p = e^{- \Delta t/\tau}$

Learning Methods

An important part of neural network modelling is to be able to introduce learning in a model. There are many different learning algorithms commonly utilized in neural network simulation, the most popular being *back propagation* [Rumelhart, Hinton, and Williams, 1986]. This learning algorithm is *not* biological and thus differentiates biological modelling, which is primarily concerned with modelling the brain in a faithful way, from the study of artificial neural networks, otherwise known as connectionism or neural engineering, where the main concern is in applying neural network computing techniques to varied technological applications. Artificial neural networks take advantage of newly developed neural learning algorithms to approach problem domains where traditional programming approaches may not have been very successful.

Similarly to numerical methods, learning rules are treated as *orthogonal* object classes where learning is abstracted away from the network description. The reason being that learning methods also keep on evolving. Moreover, different learning rules may be more appropriate for different kinds of applications, and so it is valuable to be able to readily change the learning algorithm while exploring the applicability of a given network architecture.

NSL: Neural Simulation Language

The first simulator, NSL 1.0 [Weitzenfeld 1989], has been extensively used in our research laboratory, and by students taking the brain theory course given at USC. Experience with this work has prompted us to develop NSL 2.0 [Weitzenfeld 1990] following an object-oriented design. While NSL 1.0 is implemented using C [Kernighan and Ritchie 1978], NSL 2.0 is implemented using C++ [Stroustrup 1987].

The functionality of the simulator is basically the same in the two versions. The most important modification has been the change in the conceptual description of neural networks language elements, and the incorporation of classes of numerical methods and learning methods, while the simple *objectization* of the original design has given the system a much cleaner implementation.

A common characteristic of the two implementations, is that *concurrency* is simulated by updating layers in the network only when the entire network has been completely processed after each time cycle. This way during an entire processing cycle only previous layer values will be utilized; the newly computed ones will then be used during the next processing cycle.

Other aspects of the NSL system, such as interfaces and graphics, are also implemented following the object-oriented design.

Data Structures

LAYER - The layer is an abstract super-class.

DATA_LAYER - Data layers are layer derived classes which may have several dimensions.

INPUT_LAYER - External network input is managed by a special input layer class derived from the layer super class.

NEURON_LAYER - A neuron layer is derived using the basic layer classes. The purpose of this class is to hide away the details for specific neural circuitry.

MODULE - Modules group together the different layer equations. The user can control which modules are to be active at every time, while distributing the tasks performed in the network.

NETWORK - This object class is at the top of the conceptual neural network hierarchy. The model class identifies a particular neural network, made up of a list of layers and modules describing a particular inter-connection topology.

Numerical methods

All numerical methods are treated as orthogonal object classes to be instantiated at run time. Thus, a model may be run using varied numerical methods without need for re-compilation. Moreover, the syntax accommodates any differential equation

$$\tau_m \frac{dm(t)}{dt} = f(S_m, m, t)$$

for updating the membrane potential, and not just the leaky integrator form, using the syntax

$$diff.eq(m,tm) = f\ (Sm,m,t)$$

with the details on the numerical method completely hidden. Here, m is the layer name, tm (for τ_m) is the time constant, *diff.eq()* applies the appropriate integration method, and S_m specifies the input to the layer.

Learning methods

Similarly to numerical methods, NSL incorporates learning rules as object classes to be instantiated at run time. A model may be run with different learning rules without re-compilation. The user specifies the network architecture, and includes a general 'learn' call for the respective weight matrices to be learned,

$$learn(W_{ab}, A, B)$$

where W_{ab} is the weight matrix to learned, A and B are the interconnected layers, for some chosen *rate* of learning.

Discussion

As part of our ongoing research, our next goal is to fully distribute the system both in terms of processing and graphics. The system will be run in a distributed heterogeneous environment, where graphics processes will reside on a graphics engine, and neural network processing will take advantage of massively parallel hardware specially designed for that kind of processing. Independent neural network modules may be running on different machines while at the same time they may communicate with each other. An important research aspect in linking together different neural models is to define what kind of communication is best suited and how to maximize both processing and communication performance in such a distributed environment.

References

Agha, G., 1986, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press.

Agha, G., and Hewitt, C., 1987, *Concurrent Programming using Actors*, Yonezawa, A., Tokoro M. (Eds), Object-Oriented Concurrent Programming, MIT Press.

Arbib, M.A., 1981, Perceptual structures and distributed motor control. In *Handbook of Physiology — The Nervous System II. Motor Control* (V.B. Brooks, Ed.), Bethesda, MD: Amer. Physiological Society., pp. 1449-1480.

Arbib, M.A., 1987, Levels of modeling of mechanisms of visually guided behavior (with commentaries and author's response), *The Behavioral and Brain Sciences*, 10:407-465.

Arbib, M.A., 1989,*The Metaphorical Brain 2: Neural Networks and Beyond*, Wiley.

Goddard, N., Lynne, K.J., Mintz, T., 1987, *Rochester Connectionist Simulator*, (User's Manual), University of Rochester, Department of Computer Science.

Kernighan, B.W., and Ritchie, D.M., 1978, *The C Programming Language*, Prentice-Hall.

Lyons, M.A., and Arbib, M.A., 1989, A Formal Model of Computation for Sensory-Based Robotics, *IEEE Trans. on Robotics and Automation*, 5:280-293.

Paik, E., and Skrzypek, J., 1987, *SFINX: Neural Network Simulation Environment*, (Technical Report), Department of Computer Science, University of California at Los Angeles.

Rumelhart, D.E., Hinton, G.E., and Williams, R.J., 1986, Learning Internal Representations by Error Propagation, in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* (Rumelhart, D., and McClelland, J., Eds.), The MIT Press/ Bradford Books, Vol.1:318-362.

Stroustrup, B., 1987, *The C++ Programming Language*, Addison-Wesley.

Teeters, J., 1989, *A Simulation System for Neural Networks and Model for the Anuran Retina*, TR 89-01 (PhD Thesis), Center for Neural Engineering, University of Southern Califronia.

Wang, D., and Hsu, C., 1989, SLONN: A Simulation Language for modeling Of Neural Networks, *Simulation*, Vol. 55:69-83.

Weitzenfeld, A., 1989, *NSL, Neural Simulation Language, Version 1.0*, TR 89-02, Center for Neural Engineering, University of Southern California.

Weitzenfeld, A., 1990, *NSL, Neural Simulation Language, Version 2.0*, TR 90-01, Center for Neural Engineering, University of Southern California.

Wilson, M.A., Bhalla, U.S., Uhley, J.D., Bower, J.M., 1989, *GENESIS: A System for Simulating Neural Networks*, in *Advances in Neural Network Information Processing System* (Touretzky, Eds.), Morgan Kauffman.

Yonezawa, A. and Tokoro, M., Eds., 1987, *Object-oriented concurrent programming*, The MIT Press.