Computing with Structured Neural Networks

Jerome A. Feldman, Mark A. Fanty, and Nigel H. Goddard
University of Rochester

Designing and debugging massively parallel, connectionist systems will require appropriate tools. One such package is now in use at Rochester and other laboratories.

Rapid advances in the neurosciences and in computer science are leading to renewed interest in computational models linking animal brains and behavior. The idea of looking directly at massively parallel realizations of intelligent activity promises to be fruitful for the study of both natural and artificial computation. Much attention has been directed towards the biological implications of this interdisciplinary effort, but there are equally important relations with computational theory, hardware, and software.

Recent work on neural network computation, much of it carried out by physicists,[1] examines the emergent behavior of large, unstructured collections of computing units. We are more concerned with how one can design, realize, and analyze networks that embody the specific computational structures needed to solve hard problems. In this article, we focus on the design and use of massively parallel computational models, particularly in artificial intelligence. We also describe a computing environment for working with structured networks and present some sample applications. Throughout, we treat adaptation and learning as ways to improve structured networks, not as replacements for analysis and design.

Opportunities and limitations

Computationally, animal brains are remarkable machines with properties radically different from those of conventional computers. Our research starts from the assumption that an abstract computer based on the computational properties of animal brains may prove particularly well-suited for problems in vision, locomotion, and language understanding. By taking seriously the computational constraints faced by nature and the structure of tasks known from artificial intelligence and other disciplines, we hope to discover algorithms employed by animals that might be effective for machines.

Even a crude analysis of neural computation reveals several major constraints. When asked to carry out any of a wide range of tasks, such as naming a picture or deciding if some sound is an English noun, people can respond correctly in about a half-second. The human brain, a device composed of neural elements having a basic computing speed of a few milliseconds, can solve such problems of vision and language in a few hundred milliseconds, that is, in about a hundred time steps. The best AI programs for these tasks are not nearly as general and require millions of computational time steps. This hundred-step-rule is a major constraint on any computational model of behavior. The same timing considerations show that the amount of information sent from one neuron to another is very small, a few bits at most. The range of spike frequencies is limited and the system too noisy for delicate phase encodings to be functional. This means that complex structures are not

0018-9162/88/0300-0091$01.00 © 1988 IEEE

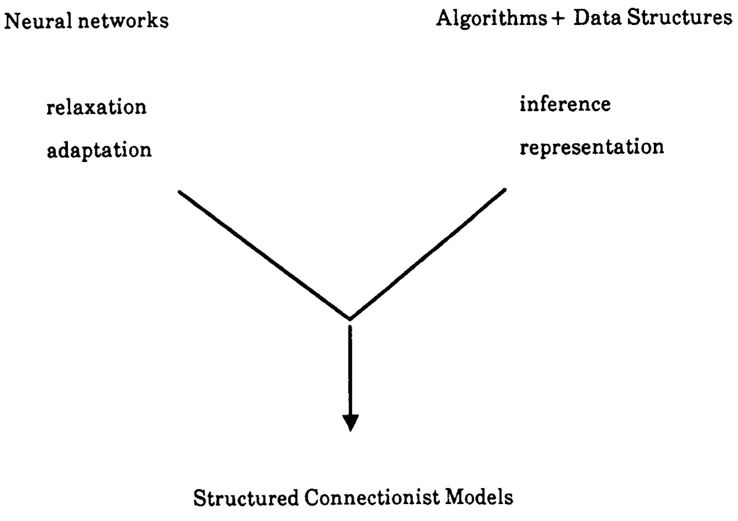


Figure 1. Two approaches to artificial intelligence.

transmitted directly and, if present, must be encoded in some way. Since the critical information must be captured in the connections, this is called a "connectionist" model.

The number of neurons in the human brain, estimated at about 10^{11}, presents serious constraints. For one thing, the number of cells in each retinal ganglion is greater than 10^6, which automatically rules out any vision algorithm requiring N^2 units since there could not be a separate unit for each possible line joining two points on the retina. Many attractive algorithms for higher level tasks run afoul of the size constraint, and considerable work has gone into what amounts to coding tricks to circumvent this constraint.[2,3]

One might think that biological constraints need not be taken seriously in designing neural-net hardware and software. Certainly, electronic switching times are a million times faster then neural ones. But the size constraint and particularly the connectivity constraint are much more serious for artificial systems. Animal brains derive much of their power from their very large fan-in and fan-out: each unit can be connected to thousands of others. For conventional chips, a fan-out of six is quite unusual. In software simulations like the ones described in this article, excessive connectivity leads to serious performance problems. The massively parallel computational solutions evolved by nature will, no doubt, prove useful to computer science and engineering, but not without modification. Connectionist computer research aims to understand the principles of massively parallel computation and to apply them to appropriate tasks.

Connectionist models can be viewed as synthesizing two traditionally opposed approaches to artificial intelligence.[4] (See Figure 1.) Some early AI investigators focused on the parallelism, robustness, and plasticity of animal brains and explored ways of automatically generating high performance. The other group concentrated on the detailed structure of tasks and algorithms and expressed them in conventional computer notation. Structured connectionist models attempt to capture the best of both paradigms. For example, the network fragment of Figure 2 encodes a conventional semantic network and inference structure in a parallel, evidential realization. Efforts like this one by Shastri[5] involve all the analysis and design issues of both AI and network theory. Recent developments in hardware and in connectionist learning theories[6] have led some people to believe that all this might not be necessary and that uniform learning mechanisms will suffice for all the problems of interest. This is a seductive idea, but there are very good reasons—some of which are outlined below—for believing that it cannot work.

Realistic expectations. The current explosion of interest in neural networks (connectionist models, etc.) is based on a number of scientific and economic expectations, some of which are unreasonable. We can be quite sure that neural networks will not replace conventional computers, eliminate programming, or unravel the mysteries of the mind. We can expect better understanding of massively parallel computation to have an important role in practical tasks and in the behavioral and brain sciences, but only through interaction with other approaches to these problems. As always, specific structures of problems, disciplines, and computational systems are the cornerstone of success. The main hope of massively parallel (neural network) research is that it will provide a better basis for such efforts.

One particularly simplistic view of neural networks is that they will support intelligence by implementing a holographic-style memory. The basic problems with any holographic representational scheme are cross-talk, communication, invariance, and the inability to capture structure. Essentially the same problems have prevented the development of holographic computer memories or recognition systems despite considerable effort. Consider the problem of representing the concept *grandmother* as a pattern of activity of *all* the units in some memory network. Notice what happens if two (or more) concepts are presented at the same time—for example, *grandmother* at the *White House.* Obviously, if every single unit must have a specified value for the network to capture *grandmother,* then no other concept can be active at the same time. If each concept is spread over some fraction of the units, the chances are high that the encodings will overlap and cause confusion. One can reduce the probability of cross-talk by having fewer units active for each pattern. Assuming that the cross-talk is randomly distributed, Willshaw[7] shows that the system with many concepts will be reliable (even for single entries) only if the number

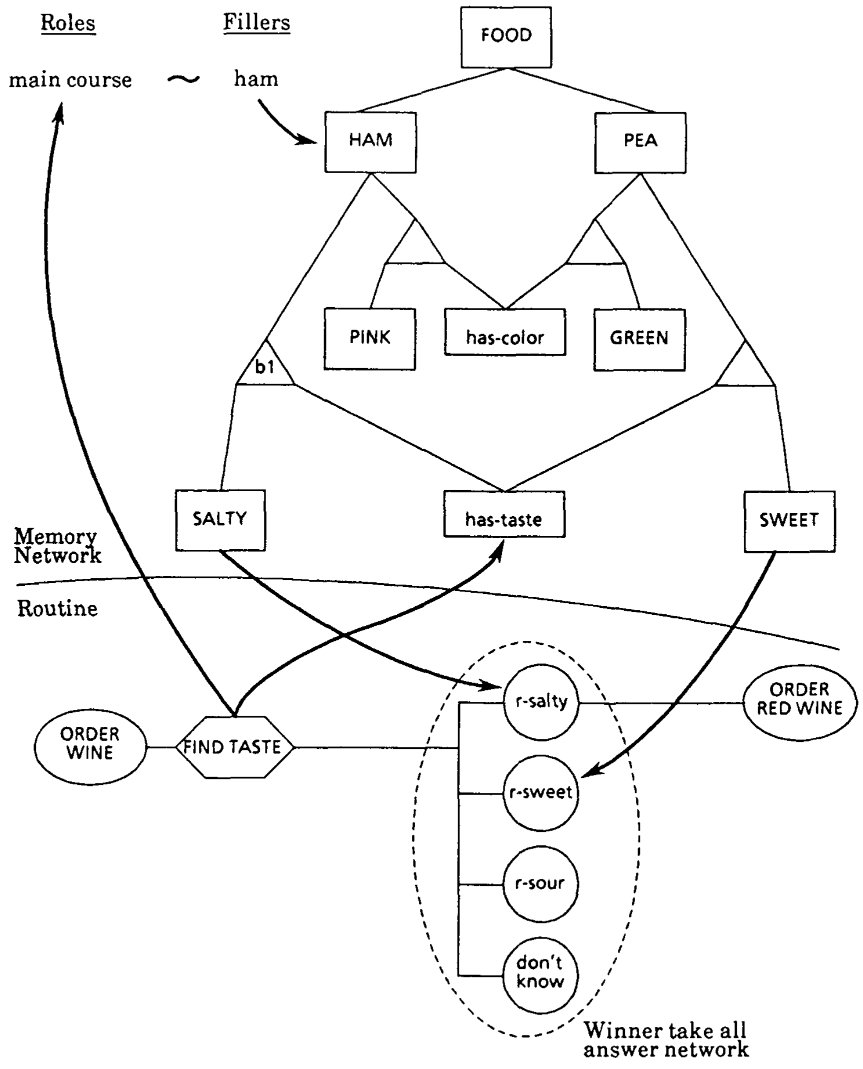


Figure 2. Interaction between a knowledge network and a routine.

of units active for each pattern is proportional to the logarithm of the number of units in the diffuse memory. This means that a network of 1,000,000 units should use an encoding with about 20 active units per concept—essentially a localized representation.

A related problem with diffuse representations is that only one concept at a time can be transmitted between subsystems, if each concept is a pattern on the whole bus. The sequential nature of diffuse representations is particularly troublesome when we consider how information about a complex scene can be transferred from vision to other systems such as language and motor control. There appears no alternative to assuming that, at least for simultaneous communication, representation of concepts must be largely disjoint and thus compact.

The same communication problem arises within the concept memory itself if one tries to build Figure 2's knowledge structure with a diffuse representation. If a concept like "salty" is represented only by a large pattern, the links for this entire pattern must go to all places related to saltiness—and be treated correctly at each of them. If a concept encoded by N units must be linked to M other concepts, a total of $M*N$ links will be needed. The more distributed the representation, the more serious this problem becomes. Again, any serious reduction in this wiring requirement constitutes a compact representation. And, as in the intermodal case, unless these representations are largely disjoint, concept processing must be sequential to avoid cross-talk. This eliminates the massively parallel processing required by the 100-step constraint.

Moreover, no one has suggested how to represent any but the simplest concepts in the holographic style. In fact, it is far from obvious how to make the same invariant distributed pattern active for alternative views of a chair. All proposed holographic structures are totally without internal structure. Another idea would associate the representation's components with microfeatures, but these are still unstructured. Nor do any of the holographic proposals provide a way of answering even simple questions like the color of grandmother's hair. The only general suggestion on how to encode structured knowledge in a holographic system is to encode each proposition (for example, "Dave likes candy") as a separate memory.[8] One can build up arbitrary structures in this way, but at the cost of losing all the advantages that led to connectionist models in the first place. The possible technical advantages of exploring such models have nothing to do with human concept memory—for example, they violate the 100-step constraint by many orders of magnitude.

Another argument for highly distributed representations derives from the large number of input fibers—about 10^4—to cortical neurons. If all these fibers participate actively, then, by its very nature, the representation is diffuse. While there has been no definitive study on the number of presynaptic events required for neural firing, estimates gleaned from papers and conversations run from one event to a few dozen.[9] No one has suggested that several thousand synapses must fire at once for an action potential. Also, many of the connections could represent alternative ways of activating the same concept (for example, from different points in space). Another way of looking at this is that the thousand-fold connectivity is capturing an OR of activation conditions rather than an AND. Finally, learning in a connectionist system requires the potential for many more connections than are ever made functional.

All of this may seem to be flogging a dead-horse model, but purely holographic theories continue to be seriously proposed—for example, the recent flurry of interest in spin-glasses as holographic memory models in the theoretical physics community. When addressing specific problems, Hopfield and Tank[1] use highly structured compact representations, but in general speculations they prefer holographic style models. Any physical system will, in isolation, reach some stable state, and each of these states can be looked on as encoding a different concept—obviously, a massively parallel system.

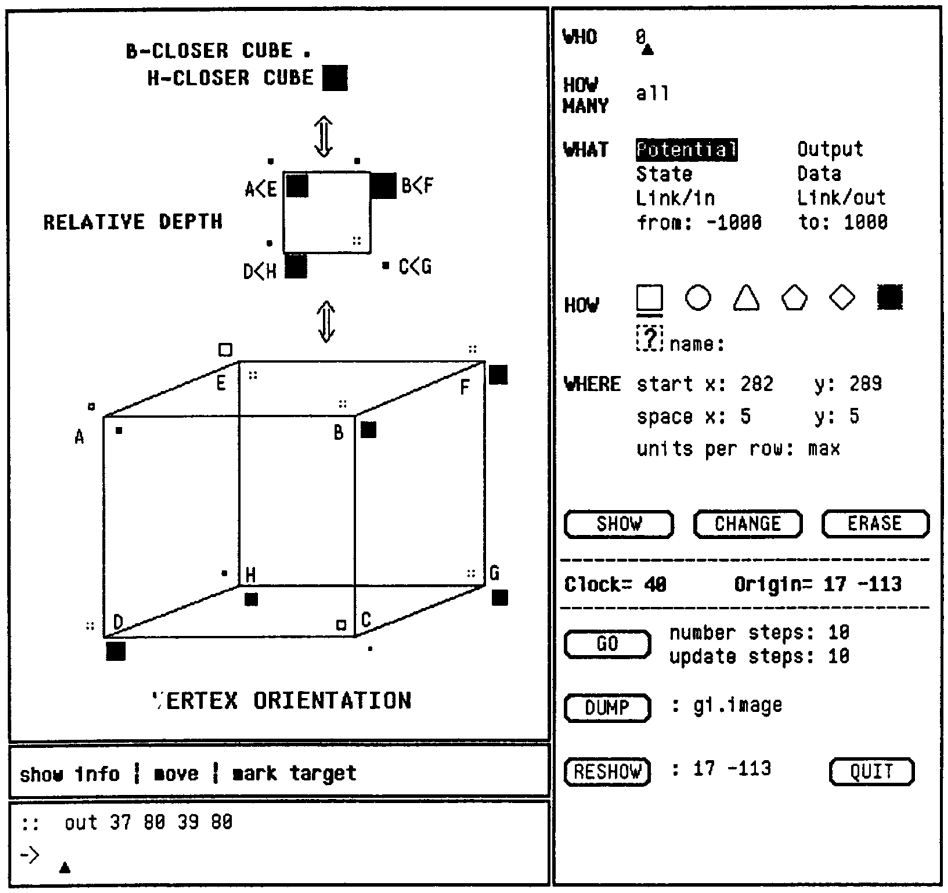
A more sophisticated version of the universal neural-net hope is that new learning techniques will induce the required structure in an initially uniform or random network. There has been significant recent progress in connectionist learning,[6] but any nontrivial neural model also requires a great deal of prior structure. For example, the primate visual system has at least a dozen subsystems, each with elaborate internal and external connection structure.[9] Any notion that a general learning scheme will obviate the need for neuroscience, psychophysics, perceptual psychology, and computer vision research disappears as soon as one takes the vision problem seriously, and there is no reason to believe that language, problem solving, etc., are simpler or less structured. For general computation, Judd[10] has shown that even the problem of learning weights to memorize a lookup table is NP-complete and thus intractable. This implies that our formalisms, simulators, and neurocomputers must support both complex structure specification and dynamic weight change. Programming such systems, understanding their behavior, and controlling how they adapt are major continuing concerns.

The role of specialized hardware. In moving from science to technology, the issues change somewhat. Physical limits on computation speed are forcing a move to parallel systems, but not necessarily to massively parallel neural-style machines. As is well-known, current computer elements are about a million times faster than neurons, but neurons have a thousand-fold greater connectivity. These and other basic differences between electronic and living systems suggest that radically different architectures and algorithms might be appropriate in the two cases. Or maybe not; it is simply too early to tell. We do know that neural models can be effectively simulated on essentially any computer and are among the few computations that can automatically exploit an arbitrary amount of parallelism.[11,12] For the foreseeable future, the bulk of neural network research will (and should) be carried out with simulations on conventional computers. Building general-purpose neurocomputers is, at best, premature.

The interesting question concerns the long-term future of massively parallel systems in computation. A remarkable fact about computer architecture is its essential sameness for the vast majority of existing systems. Special-purpose architectures have had a negligible impact, and this should be a caution against neuro-euphoria. Moreover, basic reasons virtually preclude neurocomputers replacing conventional ones for most tasks. The main reason is the greatly lower cost of passive versus active elements. Many existing tasks, such as word processing, can be done adequately by a very small number of (very fast) active elements. More generally, one can view the development of writing, of mathematics, and later of computers as ways of complementing the natural capabilities of neural-style representation and computation.

If they are not going to replace conventional machines, what future is there for neurocomputers? One possibility is that calculations of physical systems will be expressed best by massively parallel networks, more or less directly simulating the situation of concern. Some low-level signal processing might best be done by parallel analog or hybrid networks.[13] These seem quite plausible, but are a small (though important) part of computation. The best hope for widespread use of neurocomputers is, unsurprisingly, in computationally intensive areas not successfully attacked with conventional systems. The obvious ones are those that require human-like capabilities in perception, language, and inference—the traditional concerns of artificial intelligence. If such efforts succeed, there will be related applications in real-time control of complex systems and others we can't anticipate now.

If the practical future of neurocomputers depends on intelligent activity, we have come full circle—back to the scientific issues. The critical question is *do some features of massively parallel computation make it uniquely suited for calculations associated with intelligence?* The shortest path to an answer may be the indirect one of exploring connectionist models of intelligence.[2,3] If neural-style computation has unique advantages, these should appear in the higher level specifications of perception, language understanding, and the



```
:: out 37 80 39 80
->
```

```
WHO    0
HOW    all
MANY
WHAT   Potential        Output
       State            Data
       Link/in          Link/out
       from: -1000      to: 1000
HOW    □ ○ △ ⬠ ◇ ■
       ?  name:
WHERE  start x: 282     y: 289
       space x: 5       y: 5
       units per row: max

Clock= 40       Origin= 17 -113

number steps: 10
update steps: 10
: gl.image
: 17 -113
```

Figure 3. Networks for the Necker Cube.

like. If the conventional symbolic formalisms suffice at these levels, it is exceedingly unlikely that neurocomputers will be needed. Of course, another consequence of the convergence of scientific and technological goals is a shared interest in the detailed structure and function of natural intelligence.

When the faith in all-encompassing neural networks is abandoned, we are left with a complex set of interacting scientific and engineering issues. Rather than causing the demise of conventional computer science and engineering, neural networks present a wide range of new problems and opportunities. Thus, if we are going to design and debug neural networks, an appropriate set of tools could be very helpful. One such package, described below, has been in use for some time at Rochester and at scores of other laboratories.

Simulation environment

We can capture some of the flavor of connectionist computation with a simple example. The cube shown in Figure 3 is a famous optical illusion originally due to the Swiss naturalist, L.A. Necker (1832). Most people initially see the cube with the vertex B closer to them, but it also can be seen as a cube with vertex H closest to the observer. If you focus on vertex H and imagine it coming out of the paper toward you, the picture will flip to the H-closer cube. Notice also that the flip takes less than a second. The Necker cube is interesting to psychologists because it flips spontaneously between the two views if you keep looking at it. To us, it is interesting because of what it reveals about parallel computation.

You have observed how quickly the Necker cube flips state and know how slow the underlying human computing elements are. It seems unlikely that a sequential program on such a slow device could do the job. But the situation is much more complex. We know that both human and computer vision require several levels of processing. Typical levels include edge segments, lines, vertices, faces, and object descriptions. The edges and lines are the same for both the H-closer and B-closer

```
#define FRONT = 0 BACK = 1 DEPTH = 2-literal constants

CreateUnits ()                                        - make units and sites
  {
    for i = 0 to 25
      j = MAKEUNIT(“node,”UFasymp)                    - asymptotic function
      ADDSITE(j,“excite,”SFweightedsum)               - weighted sum function
    NAMEUNIT(“VIEW1,”ARRAY,0,4,3)                     - view B units
    NAMEUNIT(“B-CLOSER,”SCALAR,12,0,0)                - view B top level
    NAMEUNIT(“VIEW2,”ARRAY,13,4,3)                    - view H units
    NAMEUNIT(“H-CLOSER,”SCALAR,25,2,0)                - view H top level
  }

CreateInhibition ()                                   - make inhibit links
  {
    for i = 0 to 7 BiLink(i,i+13,−575)                - link opposing vertex
    for i = 8 to 11 BiLink(i,i+13,−800)               - and depth units
    BiLink(INDEX(“B-CLOSER”),INDEX(“H-CLOSER”),−1000)
  }                                                   - link top-level units

CreateExcite (front,back,depth,cube)                  - make excitatory links
  {                                                   - for one view
    for i = 0 to 3
      BiLink(front+i,front+((i+1)%4),200)             - around front face
      BiLink(back+i,back+((i+1)%4),200)               - around back face
      BiLink(depth+i,depth+((i+1)%4),300)             - between depth units
      BiLink(front+i,back+i,200)                      - between faces
      BiLink(front+i,depth+i,300)                     - between vertex and
      BiLink(back+i,depth+i,300)                      - depth units
      BiLink(depth+i,cube,500)                        - from depth to cube
  }                                                   - and cube to depth

BiLink (firstunit, secondunit,weight)                 - make symmetric links
  }                                                   - no link function
    MAKELINK(firstunit,secondunit,“excite,”weight,NULL)
    MAKELINK(secondunit,firstunit,“excite,”weight,NULL)
  }                                                   - link top-level units

build()                                               - build function
  {
    CreateUnits()                                     - make units
    CreateInhibition()                                - make inhibit links
    CreateExcite(INDEX(“VIEW1,”FRONT),INDEX(“VIEW1,”BACK),
    INDEX(“VIEW1,”DEPTH),INDEX(“B-CLOSER”))           - VIEW1 links
    CreateExcite(INDEX(“VIEW2,”FRONT),INDEX(“VIEW2,”BACK),
    INDEX(“VIEW2,”DEPTH),INDEX(“H-CLOSER”))           - VIEW2 links
  }
```

Figure 4. Code for setting up the Necker cube of Figure 3. Primitives are in uppercase.

cubes, but many other visual features are seen differently in the two views. For example, vertex C is oriented into the paper in the B-closer reading, but out of the paper in the other reading. Similarly, C is closer than G in the B-closer reading, and all these perceptions are mutually consistent and reinforcing. The remarkable fact is that our visual system simultaneously flips all these perceptual decisions from one mutually consistent reading of the cube to the other. This illustrates the key cooperative property of massively parallel computation and its conceptual difference from Von Neumann computation on standard machines.

Figure 3 also illustrates some details of the connectionist paradigm. In our models, each item of interest is represented as a computational unit with connections to many other units. Each unit has a level of activity (here between −100 and 100) and automatically sends the value of this activity along all its outgoing connections. For example, the top double arrow in Figure 3 depicts the fact that the unit representing the H-closer cube has positive two-way links with the four relative distance detectors: A < E, etc. Each unit at each level in this network has a rival to which it is connected by a mutual inhibition link. The strengths of links into a particular unit can also be displayed. The only other information needed for a complete model is the rule by which a unit computes its new activity from its inputs and its old activity; the simulator described in this section allows a wide variety of update algorithms. We can assume for now that the units compute the sum of their positive and negative inputs. Networks like Figures 2 and 3 are not very sensitive to the exact choice of unit computation rules; this is one of the reasons for their attractiveness. Units that are all mutually connected by negative links are said to comprise a winner-take-all network. Such networks are one of the main decision mechanisms in connectionist models and have known neurophysiological analogs.

Much of the effort in massively parallel AI is dedicated to using computational frameworks like that in Figures 2 and 3 to build models of intelligent activity. Advantages of this approach include its link to natural intelligence, increased noise resistance, and ease of implementation on parallel hardware. But the main advantage of the connectionist approach is that it provides a much better way of specifying some computations. No alternative way appears to describe the Necker cube phenomenon nearly so clearly and concisely as Figure 3.

Specifying and simulating networks. Researchers experimenting with structured connectionist networks must be able to implement and test their ideas. Since our network models presume much structure, and network architecture design is a major component in the research effort, it is necessary to develop a network description language and a simulation system that supports varied architectures. Different researchers working in diverse areas need the ability to build and simulate radically different networks. Three of the most important aspects that may differ are connection topology, activation functions, and the amount of data associated with each network node.

Connectionist networks consist of simple computational elements (units) that communicate by sending their level of activation via links to other elements. The units have a small number of states and compute simple functions of their inputs. Associated with each link is a weight, indicating the "significance" of activation arriving over that link. The pattern of connections, the weights on the links, and the unit functions determine network behavior.

The Rochester Connectionist Simulator has evolved from simple beginnings into a sophisticated research tool that supports construction and simulation of a wide variety of networks. The main design criterion has been flexibility. Each unit can compute a different function and have any amount of associated data; an arbitrary connection pattern may be specified. This flexibility exacts its cost in time and space. Compared to special-purpose simulators, it is time-expensive because each unit and link is simulated by a separate function call, and space-expensive because each unit and link must have an explicit representation.

The network paradigm supported by the simulator is quite general. Each unit has a number of sites at which incoming links arrive. The provision of sites allows differential treatment of inputs, since the links themselves do not indicate their origin at the destination site. Associated with each unit are various pieces of data, including potential, output, and state. The potential corresponds to the unit's level of activation, the output is transmitted along all links emanating from the unit, and the state is used to make simple decisions about how to interpret the unit. Associated with each link, site, and unit is a function representing its action.

Network specification. The construction environment provides a set of primitives for specifying a network. Typical primitives make a unit or link, or name a unit or array of units. The primitives provide a conceptual structure through which networks may be specified. Rather than define an entirely new language for specification, we use an existing programming language, augmented by the primitives.

A network is built in the simulator by a user program written in C. The primitives are implemented as library functions, called from the user program. The specification parameters for units, sites, and links include initial data values and an activation function. These activation functions may be different in every case, either written by the user or supplied from a library. Within each unit, site, and link, a general-purpose data field exists, which can be used to point to an arbitrarily large structure. When displaying, saving, or reloading the network, user-supplied functions are called by the simulator to handle these user-defined structures.

The large library of functions facilitates the construction task by supplying many of the commonly used unit, site, and link functions, as well as functions to create particular network structures. Researchers may add their own unit and structuring functions to the library, thus augmenting the simulator's utility for themselves and others. An example is a library that greatly eases the construction of back-propagation networks, as either stand-alone networks or submodules within a larger network.[14] This library allows arbitrary error propagation functions as well as any unit activation function.

An important consideration in specifying a network is the ability to give descriptions at different levels of abstraction. At the lowest level, the unit, site, and link functions give single-unit descriptions. At the next level, a set of functions, which specify the links and groups of units, describe the pattern of connectivity. This may include modularity within the network, and for any large network necessarily reflects regularity. At a still higher level, network specifications given in user-defined languages may be read in and compiled into units and links by user-supplied functions.[5]

A sample construction program. For an example of a network and its specification program, let's look at the Necker cube network in Figure 3. It consists of 26 units — eight pairs for the vertices, four relation pairs, and the decision units. The units corresponding to the two views are displayed as solid-line squares and dotted-line squares, respectively. The figure shows the situation when the H-closer view has almost been recognized. Most of the units consistent with this view are near maximum activation, and those of the other view at minimum.

The vertex units represent local three-dimensional orientation information. The relation units represent pairwise constraints between corresponding vertices, and the cube units represent the two possible interpretations. The links to or from a unit can be displayed with the graphics interface, which was used to generate Figure 3.

Figure 4 shows a specification program

for this network, written in pseudocode. As can be seen, the structure in the network is reflected in the structure of the specification program. Units 0 to 8 represent the B-closer cube's vertex nodes, ABCDEFGH in order. Units 8 to 11 represent the depth level, A < E, B < F, C < G, and D < H in order. Unit 12 is the B-closer unit. Units 13 to 25 are encoded in a similar manner for the H-closer network. The linking specifications implement this choice of encoding. (See Feldman et al.[12] for details.)

Simulation. Once a network has been constructed, the simulation, run either synchronously or asynchronously, can begin. During synchronous simulation, all units use the output values computed during the previous step as their input. The order of simulation is unimportant since the network behaves as though all units update simultaneously. During asynchronous simulation, a fraction of the units are updated at each step in pseudorandom order, and the new output value is immediately transmitted to the other units. Synchronous simulation is easier to understand and is marginally faster than asynchronous, but in cases like the Necker cube example, synchronous updating with deterministic unit functions always leads to the same outcome. In other cases, synchronous simulation can lead to oscillations or other problems of failing to break symmetry.

A command interface controls the simulation and modifies the network during simulation. The interface allows execution of any user function and of a preloaded set of commands. One current trend in our simulation environment is toward a more interactive system for network specification. Dynamic code-loading capabilities allow runtime redefinition of activation functions and reconstruction of parts of the network. Integration with the Common Lisp and Scheme environments provides the added expressive power of those languages for network specification.

Performance. Two major performance issues in neural network simulation are size and performance. On a SUN3/260, a network of 2000 units (computing the weighted sum of their inputs), each with 100 links, took 25 seconds to construct and performed 100 simulation steps in 83 seconds, giving an interconnect time of approximately four microseconds. The number of links in a network is a crude but effective measure of its size; on the SUN3/260 with eight megabytes of memory, the maximum number of links attainable before thrashing sets in is of the order of a quarter of a million. The parallel simulator, discussed below and in Feldman et al.,[12] can do much better.

While space and time efficiency of the simulation engine is crucial for experiments with large networks, clearly, ease of use, extensibility, and flexibility are equally important attributes. Future needs, such as large networks running on parallel hardware, will require fast simulation systems connected to a high-speed graphics interface.

Graphics interface. The current graphics interface, a first-generation tool developed by Kenton Lynne, has proved indispensable for displaying network information during simulation and for aiding the network debugging process. It allows the user who has created a network with the simulator to graphically view and examine that network before, during, and after execution. Each significant aspect of each unit in the network can be displayed as a separate graphic object (or icon) whose size, shape, or shading varies with the current value of that aspect. As the simulation runs, the icons are constantly updated to reflect changing values, giving an overall view of what the network is doing. In keeping with the philosophy of the simulator, the graphics interface was designed to give maximum flexibility to the user in how the network is displayed.

Lynne wrote the graphics interface specifically to run on our Sun workstations using Sun Microsystems' graphics tool package. He designed it as a separate part of the simulator package; the user specifies it as an option when the network is compiled into object code. If included, the graphics interface code automatically creates its own window, which layers itself between the user and the simulator. The user then interacts with the simulator via the graphics interface window, which either executes user-given graphics commands or passes appropriate commands to the simulator. It provides and maintains a graphics panel, which displays the graphical representation of the user's network as simulation proceeds. Figure 3 shows the graphics interface tool in potential mode examining the activations for the Necker cube example. The simulator is unaware of the graphics interface and, in fact, still generates text output to its own window, just as though the user were interfacing to it directly.

In addition to displaying and running the network, the graphics interface has facilities that allow the user to

- display detailed textual information for specific units,
- show network topology,
- write the graphics display to a raster file,
- put text and line drawings on the display for documentation purposes,
- log the simulator and graphics commands for later replay, and
- map the mouse buttons to execute simulator or graphics commands.

There is also a version that communicates with the parallel connectionist simulator running on the department's BBN Butterfly multiprocessor.

Through commands or mouse actions, the user specifies exactly where the icons are to appear in the display space, which is effectively an unbounded Cartesian plane. The graphics panel, which can be stretched vertically and horizontally, always shows some finite rectangle of the display space and can be moved via commands or mouse actions to show different parts of the display space. Thus, the size of a displayable network is limited only by machine memory. (Each displayed aspect requires 48 additional bytes of memory). The user can also add text and line drawings to the display space to document the network as shown in Figure 3. All objects (icons, text, drawings) appearing on the graphics display can be freely moved around with commands or the mouse as dictated by taste or function.

Using the graphics interface significantly reduces the time needed to get a connectionist network up and running. With the proper unit aspects displayed, one can quickly determine if a network is working properly and if not, where the problem lies. Much of the power of the interface lies in its dynamic properties, which do not show up in static pictures. For example, one can catch oscillations, which can paralyze a network, almost immediately. The graphics interface makes it much easier to specify and debug the large, structured networks we work with at Rochester.

Parallel implementation. We have also implemented the connectionist simulator on a BBN Butterfly multiprocessor. The parallel simulator looks and functions much like the uniprocessor simulator. Code can be ported with little modification, as long as it does not directly access the network data structures. If the user is content with a naive network partition, he

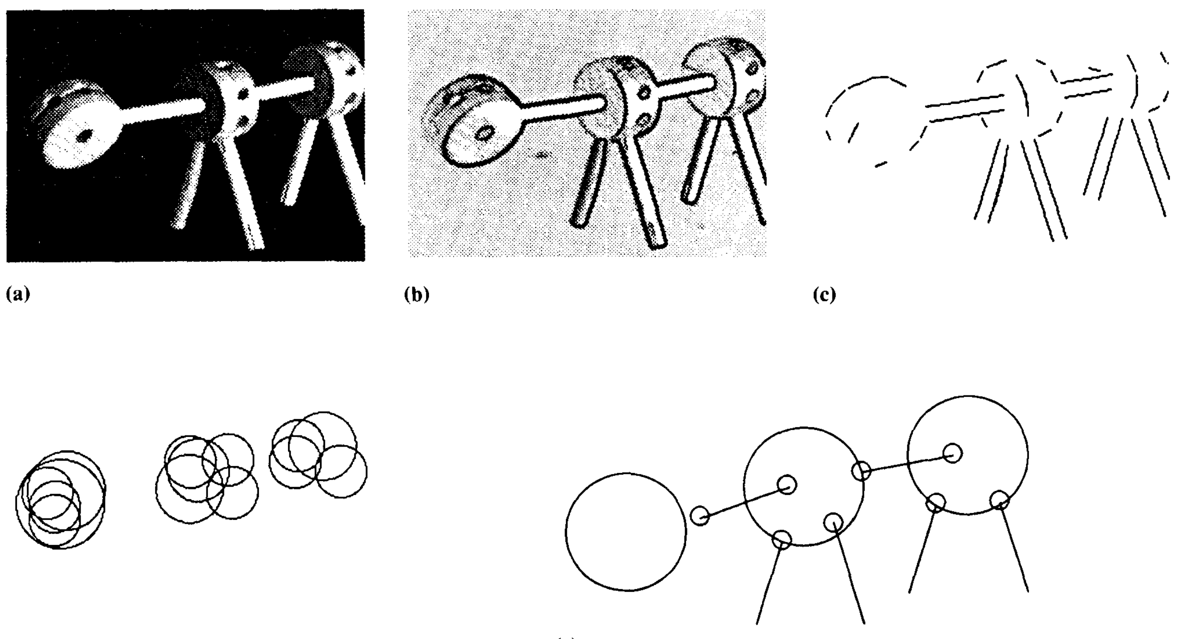


Figure 5. Tinker Toy low-level processing: (a) original figure; (b) response to Kirsch operator; (c) extracted lines; (d) extracted circles; (e) final symbolic representation, including connections detected between parts.

can ignore the fact that the simulator is actually running on several processors.

Using the Butterfly multiprocessor significantly increases the speed and capacity of network simulations. Our largest Butterfly configuration is 120 processors with a total of 120 megabytes of memory. It easily runs networks that do not begin to fit on any of our uniprocessor machines and achieves nearly linear speedup.[12] Even with smaller networks, simulations that would run for hours on a uniprocessor take only minutes.

Sample applications

The driving force behind the system developments described above has been applications of connectionist models, particularly to problems in artificial intelligence. Since all connectionist networks are currently simulated on inappropriate hardware, applications refer to concept demonstrations and scientific models, not to programs for practical use. The literature[2,3] and this issue of *Computer* describe many such applications. We will focus on some recent Rochester work that illustrates the structured connectionist style and the use of the simulator.

The first example is a rather ambitious vision project carried out by Paul Cooper and Susan Hollbach[15] with help from Steven Whitehead and others. The basic problem is to match real images of Tinker Toy objects to stored topological models. This is a simplified model of how a neural style network could perform visual recognition in the required 100 time steps. In previous research, Cooper and Hollbach established that the early stages of vision can be done effectively in parallel; the problem is how to do high-level matching to object models. The low-level vision preprocessing for this is depicted in Figure 5. Figure 5a shows a digitized picture of a Tinker Toy horse, the input to the system. Figure 5b represents the magnitude of the gradient produced by the Kirsch operators (that is, the edge picture.) Figures 5c and 5d have the straight lines and circles found by the Hough technique from the edge information of 5b. Figure 5e demonstrates the connectivities found between rods and disks: the small circles represent joints between a rod and disk.

Figure 6 gives a typical model database. The idea is to match the analyzed input to all the models in parallel using a structured connectionist network. Figure 7 shows a model horse, labeled by letters, and a possible input image, labeled by numbers. The matching network tries to match simultaneously compatible wheels (ones with the same number of rods) and to ensure that adjacent wheels in the image match adjacent model wheels. These mutual constraints are adequate to select the correct model for a wide range of tasks. Figure 8 shows some of the consistency constraints; for example, if the image pair 3-4 matches the model pair B-C, then 3 must match B or C. The details of the model are moderately complex, involving several winner-take-all networks and simultaneous comparison of all models.

Figure 6 also depicts the results of matching the results of Figure 5 with the

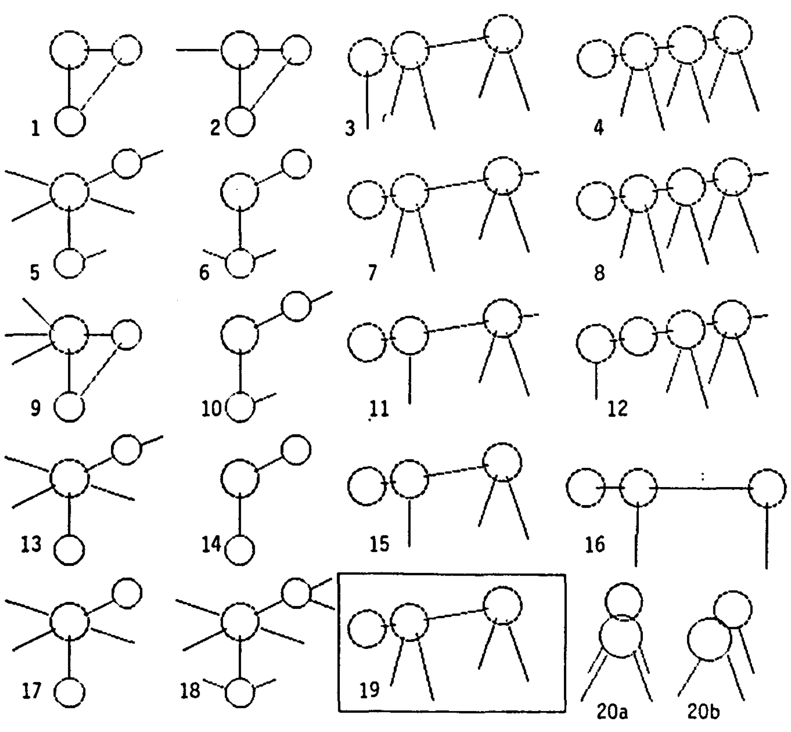


Figure 6. Tinker Toy model base.

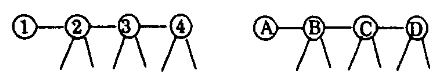


Figure 7. Example of a figure and matching model.

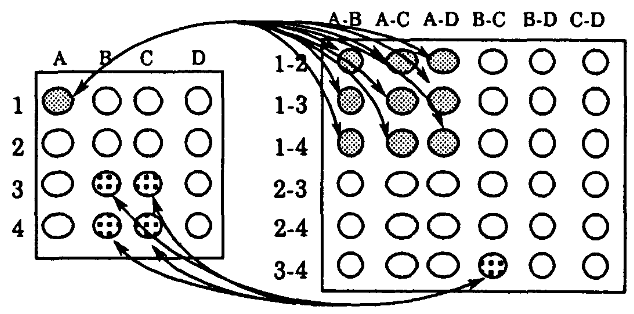


Figure 8. Wheel-matching array, constraint-matching array, and example constraint propagation links.

21 models. The central goal of recognizing the target horse figure is accomplished, and reasonable scores are obtained by partial matching of approximately similar figures. Although the network was designed primarily to accept or reject matches of objects with the same number of disks, it also computes partial matches and matches of objects with differing numbers of disks. For example, compare the extracted figure with models 3 and 7 (judged to be topologically quite similar) and models 4, 6, and 15 (judged fairly similar).

The Necker cube and Tinker Toy examples are instances of AI recognition problems. Several other problems are like this, but many others are not. Can we apply structured connectionist models to other traditional AI issues such as knowledge representation and inference? There is much less completed research along these lines, but some promising starts have been made. The example in Figure 2 should convey the flavor of this work.

The standard way to explore the issue of knowledge representation and inference is in terms of programs that can answer questions. The many AI approaches to developing question-answering systems have the same basic requirements: One needs a way to store knowledge, to pose questions, and to compute and register answers. In a connectionist model, all these aspects must be expressed in terms of activity spreading among simple units like those in the previous examples.

It is easiest to start with the recording of answers. In Figure 2, the possible tastes of foods form a winner-take-all network in which each unit inhibits the others so that only one answer will be active. The answer network is assumed to be part of a routine that also poses the question and acts upon the answer. The units that make up the routine are assumed to be activated in sequence from left to right, just like a standard program. Activating the appropriate units sends a question to the knowledge network; Figure 2 shows this as links from the hexagonal node to the nodes for [has-taste] and [ham]. The key to this network's operation is operation of the triangular-shaped nodes such as [b1] in Figure 2. A unit shown as a triangle is defined to become active when two of its inputs are simultaneously active. In this case, [ham] and [has-taste] are both on, so [b1] becomes active and activates [salty]. Now the [salty] node in the knowledge network spreads activation to the response [r-salty] back in the routine and the question is answered. The same network can answer questions like "Name a salty meat" when activated appropriately. The answers returned by such a network depend on context, as people's answers do; contextual bias is again modeled by activation.

In addition, Shastri[5] showed that structured connectionist knowledge representations can handle problems that have proven difficult for logic-based approaches. Suppose we believe that Quakers tend to be pacifists and that Republicans are generally not pacifist. Given an individual who is both a Quaker and Republican, it is hard to decide how likely he is to be a pacifist. (The recent US President who had these two beliefs was also a Marine officer). Shastri's system allows the relative strengths of conflicting beliefs and correlations to be combined according to maximum entropy rules of evidence and performs quite well. Again, the structured connectionist network provides a natural mechanism for representing the required knowledge and for capturing inferences based on partial, uncertain, and conflicting knowledge.

An obvious question arising in connection with work like Shastri's is how a network like Figure 2 could be learned. The neural substrate of memory and learning is one of the great unsolved scientific questions for which we certainly have no definitive answers. But there are connectionist theories of learning that are compatible with current brain research and are computationally feasible.[6] The key idea is that while new connections are rare, *weight change* in connections appears to be common. We also know that each unit can have thousands of incoming and outgoing connections. Our hypothesis is that most of these connections are only potentially important and that learning involves strengthening the appropriate connections. Suppose, for example, the network of Figure 2 needs to learn that spinach is a salty vegetable. Our model suggests that there are uncommitted triangular nodes that are weakly connected to many combinations of objects, properties, and values. In an ideal case, one of them will be linked to [spinach], [has-taste] and [salty] among other things. This unit will get highly activated by the simultaneous activation of three of its neighbors and, by strengthening its active connections, can become dedicated to the new association. No implementation of this learning scheme has been attempted, but there are several related studies and some theoretical work.[16]

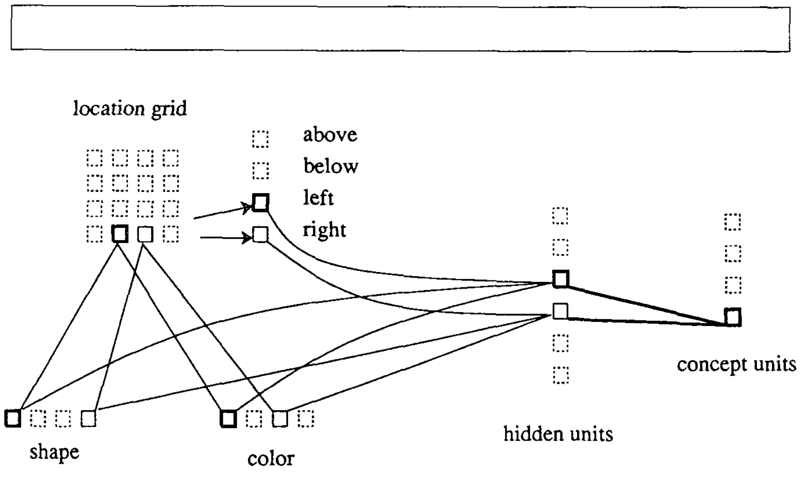


Figure 9. Input consisting of two objects, with attention focused on the leftmost of the pair.

Fanty's work[17] is our most ambitious effort to date on learning in structured neural networks. An example system based on his work learns to classify structured objects in a position-independent manner. Both structured objects and position independence present difficulties to the connectionist researcher for essentially the same reason. If a network is going to perform computations at several locations, or for several subparts of a complicated object, at the same time, then there must be several copies of the computational mechanism. This is problematical. It can lead to a combinatorial increase in the required number of units and links, an especially troublesome problem in learning networks where something learned locally should be shared globally. The proposed solution is to go sequential. At some point, even connectionist networks must begin to compute sequentially. Imagine a story understander for which the input is entered simultaneously: all 100,000 words of a novel, each in its own word slot. The whole network is replete with redundant sentence-understanding subnetworks, subplot-understanding subnetworks, and so on. This, of course, is ridiculous. The idea behind the network described here is that some sequential processing is always necessary in reasoning about multiple objects; there can, however, be resource-intensive exceptions, such as low-level vision.

For the implemented networks, each input consists of some number of subobjects, each with a shape and color. The subobjects are located on a 4×4 grid. So a single input might be a red square over a blue circle, and it could appear in any of 12 different locations. For each input, the network is provided feedback on classification. The goal is to have the network learn to classify the inputs.

Figure 9 illustrates the network's organization. A grid of units corresponding to location is the extent to which the representation is spread in space. There are four other populations of units, each with a distinct role. There is a network of object properties, shown with four values for shape and color. There is a network of relations, shown with the four spatial relations: above, below, left, and right. There is a population of hidden units, which will learn to classify subobjects. Finally, there is some number of classification units. Each complex input belongs to one of the classes. The network learns to classify the inputs based on feedback at this level.

Notice that the position independence of the learning is guaranteed; the network is prestructured, which is the whole point. Learning occurs at the hidden and classification units, which receive input from the property and relation units. The activity of these latter units is independent of the input's location on the grid. The nature of the representation needs more detailed explanation. Each grid unit corresponds to a location in the world. When one of these is active, it means something at that location is being attended to. The grid units inhibit each other so that at most one can be strongly active. The grid units representing other objects in the current input are not totally quiescent. They maintain a low level of activity, not enough to interfere with the objects of attention, but enough to keep them at the ready (and, potentially, enough to prime ongoing computations and shifts of attention). After being active for a while, a grid unit will tire and fall back to a low level of activity, and the strongest of the waiting grid units will reach full activation. External influences could control the shift of attention through priming effects, though this does not happen in the example network.

An object is entered one subpart at a time by externally activating a grid location and some properties. The external activation is very strong and causes a temporary binding between the grid object and the properties. This is effected via a large change in the link weight from the grid unit to the property unit. The weight change will stay in effect until the grid unit becomes totally quiescent. So, as the grid units cycle up and down, the associated properties do as well. In this example, the relations are all spatial and are activated by hard-wired relation detectors (not shown). In Figure 9, grid unit (4,2) is strongly active and grid unit (4,3) is weakly active. This activates the left relation: the focused-upon object is left of something. It also has shape 1 and color 1.

The job of the hidden units is to classify the subobjects. Hidden units are only responsive to currently active units. In Figure 9, an active hidden unit responds to the triple: (shape = 1, color = 1, left). Another hidden unit responds to (shape = 4, color = 3, right), but it is shown at a low level of activation because that object is not being attended to. In the current implementation, the learning algorithm used for this stage is a variation of competitive learning.[14] This was chosen because it does not require feedback, which is harder to provide in such a dynamic environment. However, lack of feedback has numerous disadvantages, and research into other methods is ongoing. At the classification-unit level, evidence for complex objects must be accumulated over time. In this implementation the links to classification units have a memory. Thus, if they were recently active, they continue to provide support even after the source of the activation has died down. This is not the ultimate solution, but it does work in this simple example. The learning rule for these links is simple. Active inputs to an active classification are increased; others are decreased.

Many extensions and improvements are possible. The grid units could be replaced by general-purpose control units with no inherent semantics, which bind to object properties and relations much as the grid units do. Another idea, treating relations as distinct from properties at the hidden-unit level, would allow a distinct subobject learned in one context (for example, "below") to be used in another context. Of course, the subobject distinction is artificial since a complete object in one situation can be a subobject in another. Finding clean ways for connectionist systems to switch levels of elaboration is another topic of current research in several laboratories.

As these examples help point out, massively parallel, neural net-style computation presents a wide range of opportunities and challenges. It is certainly not a magic universal solution, but it does offer potentially important benefits in speed, robustness, and adaptability. To take advantage of these opportunities, we must merge neural-net techniques with those of conventional computing and other sciences. Our methodology and tools are presented as a step in that direction.□

Acknowledgments

Many thanks are due Liudvikas Bukys for his assistance with the parallel implementation of the connectionist simulator and Susan Hollbach, who helped extensively with this article.

This work was supported in part by NSF/CER Grant No. DCR-8320136, in part by ONR/DARPA Research Contract No. N00014-82-K-0193, and in part by ONR Research Contract No. N00014-84-K-0655.

The Xerox Corporation University Grants Program provided equipment used in the preparation of this article.

References

1. J.J. Hopfield and D.W. Tank,, "Computing with Neural Circuits: A Model," *Science*, Aug. 8, 1986.
2. D. Waltz and J.A. Feldman, Eds., *Connectionist Models and Their Implications*, Ablex Publishing Corp., 1987.
3. J.L. McClelland and D.E. Rumelhart, Eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol 2: Applications*, MIT Press/Bradford Books, 1986.
4. A. Newell, "Intellectual Issues in the History of Artificial Intelligence," in *The Study of Information: Interdisciplinary Messages*, F. Machlup and U. Mansfield, eds., John Wiley and Sons, New York, 1983.
5. L. Shastri, *Semantic Nets: Evidential Formalization and Its Connectionist Realization*, Morgan-Kaufman, Los Altos/Pitman Publishing, London, Feb. 1988.
6. G.E. Hinton, "Connectionist Learning Procedures," *AI Journal*, in press.
7. D. Willshaw, "Holography, Associative Memory, and Inductive Generalization," in *Parallel Models of Associative Memory*, G.E. Hinton and J.A. Anderson, eds., Lawrence Erlbaum Assoc., Hillsdale, N.J., 1981.
8. D.S. Touretzky, and G.E. Hinton, "Symbols Among Neurons: Details of a Connectionist Inference Architecture," *Proc. IJCAI*, Los Angeles, 1985, pp. 238-243.
9. G. Shepard, in *The Neurobiology of Learning and Memory*, G. Lynch, ed., MIT Press, Cambridge, Mass., 1986.
10. J.S. Judd, "Complexity of Connectionist Learning with Various Node Functions," *Proc. IEEE Int'l. Conf. on Neural Networks*, San Diego, June 1987.
11. G. Blelloch, and C.R. Rosenberg, "Network Learning on the Connection Machine," in *Connectionist Models and Their Implications*, D. Waltz and J. Feldman, Eds., Ablex Publishing Corp., 1987.
12. J.A. Feldman et al., "Computing with Structured Connectionist Networks," *Comm. ACM*, Feb. 1988.
13. J. Hutchinson et al., "Computing Motion Using Analog and Binary Resistive Networks," *Computer*, Mar. 1988.
14. D.E. Rumelhart and J.L. McClelland, eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol 1: Foundations,* MIT Press/Bradford Books, 1986.
15. P.R. Cooper and S.C. Hollbach, "Parallel Recognition of Objects Comprised of Pure Structure," *Proc. DARPA Image Understanding Workshop*, Los Angeles, Feb. 1987.
16. J.A. Feldman, "Dynamic Connections in Neural Networks," *Biological Cybernetics*, Vol. 46, 1982, pp. 27-39.
17. M.A. Fanty, "Learning in Structured Connectionist Networks," forthcoming doctoral dissertation, Computer Science Dept., University of Rochester, 1988.