AD-A244 080

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Sept. 9, 1991	Final Report;12/1/88-11/30/90

4. TITLE AND SUBTITLE
Behavior and Learning in Networks with Differing Amounts of Structure (U)

5. FUNDING NUMBERS
61102F
2305/B3

6. AUTHOR(S)
Professor Leonard Uhr

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, WI 53706

8. PERFORMING ORGANIZATION REPORT NUMBER
UW 144-AS50

AFOSR-TR- 91 0939

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Dept. of the Air Force
Office of Scienctific Research
Building 410
Bolling Air Force Base
Washington, D.C. 20332-6448

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFOSR-89-0178

11. SUPPLEMENTARY NOTES

91-19207

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release;
Distribution Unlimited

DTIC ELECTE DEC 30 1991 S D

12b. DISTRIBUTION CODE
UL

13. ABSTRACT (Maximum 200 words)

This research is investigating how well large networks that are built from neuron-like elements can be made to perform, and learn to perform, by giving them different types and amounts of built-in stucture, and the ability to learn by generating new nodes in addition to changing weights. Substantial improvements in both learning speed and performance have been achieved on both pattern recognition problems and a range of problems typically used to demonstrate the power of connectionist networks. In addition, a number of new micro-circuits and sub-networks have been specified with which more powerful and more flexible networks can be built. These include:

A. Back-cycling nets that handle learning(along with many useful functions), rather than have that handled by the system that executes the net.
B. Nets that handle symbols as well as numbers.
C. Micro-circuits for productions and perceptual transforms.

14. SUBJECT TERMS	15. NUMBER OF PAGES
	27
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

Behavior and Learning in Networks with Differing Amounts of Structure

This ongoing research is investigating how well large networks that are built from neuron-like elements can be made to perform, and learn to perform, by giving them different types and amounts of built-in structure. Systems with relatively little structure (e.g., random nets, simple layered nets) have been compared with several different types of more highly structured architectures (chosen because of good formal characteristics, and to be realizable with plausible networks of neuron-like elements). Several different learning techniques - for changing weights, and especially for generating and discovering new transforming procedures - have been investigated and compared. In particular, generation of new nodes in hierarchically converging networks has been shown to substantially speed up learning and improve performance, when compared with re-weighting (e.g., using back-propation) alone. Generation also results in much faster learning of a variety of touchstone problems (e.g., exculsive-or, parity, addition), as compared with more traditional connectionist networks. In addition, a number of micro-circuits and sub-networks have been specified that, when appropriately incorporated into the larger network, can significantly enhance learning and performance. Especially striking examples are networks that handle feedback-learning (which in today's connectionist networks is handled by the larger program that simulates the net rather than the net itself). For example, locally linked networks with appropriately positioned back-links (essentially, wherever there are forward links to the next layer add back-links) will not only handle feedback-learning but also a wide variety of useful functions (e.g., relaxation, contextual interactions, combined top-down and bottom-up processing).

Learning by Generation in Large Connectionist Networks

We have in the past developed and tested several pyramid computer vision systems (Uhr and Douglass, 1979; Li and Uhr, 1987) that use large numbers of simple local processes to successfully recognize complex real world objects like doors, chimneys, and houses. More recently, we have shown (Honavar and Uhr, 1989, 1990b) how these can be realized as massively parallel networks, using conventional connectionist network nodes.

The general pyramid approach entails using a very large network of relatively simple processors. For example, one or several processors might be assigned to each pixel cell in the retina-like array into which an impage is input. Each processor examines a local window of cells immediately surrounding it, and sends its results to one or more "parent" processors in the next array directly above it. This continues, successively extracting, abstracting, and reducing information, in a logarithmically converging structure. This logarithmically converging pyramid structure imposes a constant bound - m, the windows' upper bound on the number of local links, reducing them from $O(N^2)$ or $O(NV^N)$ to $O(mN)$ or $O(mV^N)$.

In practice, almost all of today's pyramids are simple, e.g., with strict 2-by-2 convergence and 2-by-2, 3-by-3, or 4-by-4 windows. But pyramids can be architectured to many different potentially appropriate configurations of resources with a variety of different tools.

Our implementations of generation in this kind of pyramid-structured net had each node process a 3-by-3 window of information that is input to it, with nodes positioned so that 2-by-2 conver-

gence. These systems were tested on several different pattern sets, and found to perform substantially better than traditional multi-layer nets (often giving 1 or 2 orders of magnitude speed-ups, and handling pattern sets where back-propagation alone failed). For example, they take on the order of 50 epochs to learn to recognize sets of images, compared to 500 or more epochs when backprop alone is used; and they succeed on sets of patterns where traditional nets could not (Honavar and Uhr, 1989).

The Type of Generation Algorithm Being Used and Tested

The following describes the kind of generation presently being used.

Traditional reweighting (using backprop) continues until it no longer improves learning. Generation is triggered when the fedback error indicates the network has output a wrong choice, and the accumulated history indicates that reweighting alone is unlikely to correct that error.

This triggers the generation of new structures of nodes and links, as follows:

A. A small compact window of nodes is chosen from the sub-set of nodes with the fewest output links. Originally, these will include only the nodes in the input layer (plus any interior nodes that were built in a priori). But as generation proceeds it also includes the new nodes in the successive interior layers. (The present system is restricted to using windows all of whose nodes are in the same layer.)

B. A sub-set of nodes in this window are chosen for extraction.

C. A new node is generated, with a link into it from each of the extracted nodes (which will all be in the same layer, call it L); it and its copies (see below) will be placed everywhere in layer L+1.

D. Each of these links is assigned the value that its extracted node has just fired out. This forms the new node's active-mask of values that must be sufficiently well matched in order to fire the newly generated node.

E. Each link is also assigned an initial weight (this can be a fixed value, or some function of the active-mask values and the error signal, or a random value - these will be compared).

F. The new node is linked (with high weights assigned to these links) into those output node(s) that should have been fired by this last input.

G. The following processes are given the new node (either as primitives or micro-circuits):

1. Match incoming values with the active-mask, and fire out: 1. (using a threshold) if it exceeds the threshold, or b. (using a logistic function). Alternately, stored information (e.g., a symbol, symbol-weight, or attribute-value) can be fired out when the threshold is exceeded.

2. Multiply the value fired out over a link by the weight associated with that link.

3. During the learning phase, when the error signal is backpropagated, change the weights associated with its input links in the direction of reducing error.

H. A copy of this whole new structure is placed in each cell-group in its layer (this generalizes the node's function over translation). If layer L contains an M-by-M array of cell-groups, layer L+1 contains an M/2-by-M/2 array.

I. When a weight is changed, the change can either be kept local, or it can be sent to all the other copies of this node (the latter generalizes learning over translation).

Since each layer will contain 1/4th the number of cell-groups as the prior layer, the whole net

will converge logarithmically. Upper bounds are imposed on the number of nodes that can be generated for each cell-group (when they are reached the system can simply discard the least useful to make room for a new generation). As shown above, the use of logarithmic convergence along with small, bounded numbers of links substantially reduces complexity. It can also increase power, at least for the large class problems for which such structures are appropriate. As our preliminary results already indicate, generation within hierarchically converging nets offers real promise of substantial speed-ups in learning, since far fewer links and nodes need be processed, and it is easier to determine which to modify. And useful generations can be very powerful (since they are, essentially, new compound variables that are sensitive to the interactions of lower-level variables).

Cycling Logarithmically Converging Nets That Flow Information to Behave (Perceive) and Learn

This research (Uhr, 1991b) examines how traditional feed-forward connectionist networks (CN) with complete connectivity from layer to layer can be modified and extended slightly, so that they:

A. can be used to realize logarithmically converging "recognition cone" computer vision systems;

B. are capable of executing and integrating a large variety of top-down and bottom-up processes; and

C. handle learning within the network itself (rather than, as is always done today, by the host computer on which the net is simulated).

Standard connectionist networks behave/perform, outputting a response as a function of information received by a set of input nodes. To make them also capable of learning (typically by reweighting values associated with links), a quite precisely structured - and substantially larger - net must be added (Uhr, 1989a). The necessary additions (at least 1 node for each link, plus several nodes for each node) rather than being implemented within the network itself, by augmenting it with the necessary additional sub-nets, are instead handled outside the net, using traditional programs.

This research describes a new kind of continually flowing multi-layered converging network that can handle feedback and learning using the same processes, links, and nodes that are used to behave (e.g., recognize, remember, reason) - at virtually no extra cost, since the added processors are necessary simply to handle learning, but now can be used during the behaving cycle as well. The added back-links between layers - in addition to handling learning - make possible a variety of potentially very useful cycling processes, including inner-driven assessments, successively more global contextual interactions, constraint relaxation, threshold adjustment.

Digital and Analog Micro-Circuit and Sub-Net Structures for Connectionist Networks

There are three major ways to make connectionist networks more powerful (Uhr, 1989a): make their primitive units more powerful; build in more powerful local micro-circuit and successively more global structures over these primitive units; and/or give the network more powerful mechanisms to learn and evolve. This research examines how to build more powerful primitive

units and micro-circuit structures.

First it defines a set of primitive building blocks and a set of primitive functions (e.g., to add, multiply, threshold, match, store), and gives a few examples of how they can be realized using alternate sets of digital and analog components - including traditional connectionist units. It then uses these to define a variety of primitive units, including the multiply-add-threshold units typically used in today's connectionist networks, and also units that convolve, serve in expert production systems and semantic networks, and transform symbolic and numeric information. It examines how these can be structured into larger networks, and sketches out a number of additional micro-circuits, including several suggested by the brain.

There are many potentially extremely useful structures that can be built into, or learned by, connectionist networks of neuron-like elements - as this examination indicates.

The following are more extended descriptions of work to be published. (These have been extracted from papers in progress that have not yet been issued as Technical Reports.)

CN Need $O(N^2)$ Additional Sub-Nets to Handle Learning Within the Actual Net Itself

A connectionist network (CN) behaves, outputting responses, as a function of inputs; but the larger program that executes it - rather than the network itself - handles learning. This research explores the substantial sub-nets that must be added to make the network complete and self-contained, so that the network itself handles feedback-learning, as well as performing-behaving. Several contrasting types of learning are examined: Hebbian re-weighting from nearest-neighbor internal feedback, perceptron re-weighting, and back-propagation. If the original network - that behaves only - has N nodes (of which O are output nodes) and L links (L is typically made as close to N^2 as practicable), then the necessary additions to handle learning need at least 2L (where $L=O(N^2)$) new nodes and 5L new links (Hebbian nets might need fewer, back-propagation needs substantially more). In most cases, each new "node" is actually a functional black box-like module that needs a whole micro-circuit (its size depends upon the complexity of the particular functions computed to handle learning) of the kind of basic multiply-sum-threshold(or squash) units typically used for behaving in today's systems. Thus handling learning within the net itself adds $O(kN^2)$ new nodes and links. So the basic net's links become far more costly than its nodes, and as nets grow larger this becomes an increasingly important reason why complete connectivity is not feasible. In addition to these major overlooked costs, the resulting nets have an interesting characteristic: they are very precise and regular, with highly iterated structures of micro-circuits.

A new type of continually firing and back-cycling information-transforming network that handles learning with virtually no additional mechanisms is briefly described. This becomes possible when a more general type of structure is established, one that treats feedback as simply another field of information that must be input, sensed, recognized and processed.

The Additional Sub-Networks Needed to Handle Learning from Feedback

In striking contrast to the way the specification of the net completely determines - and when realized handles - its behavior, in today's connectionist systems learning is not handled by the net itself, but instead by procedures that are typically written in whatever programming language is used to execute the net-simulation.

Virtually no attention has been given to what the necessary learning sub-nets would actually look like. Yet they are quite complex; indeed they are substantially larger than the networks needed to behave and handle problems; and they appear to benefit from - if not require - far more precision in highly iterated structurings of exactly which nodes should be linked to which.

First, there must be enough links to form paths from feedback to all loci where learning might modify the net.

In addition, there must be nodes whose micro-circuits temporarily remember all fired values until feedback arrives, evaluate whether each value should be changed, determine what these changes should be, and actually change each value as indicated - and also control and coordinate all these processes.

The following examines these issues, and specifies how 3 major types of learning can be handled by augmenting today's traditional connectionist networks, which handle behaving only:

1. Hebbian self-organizing learning (Hebb, 1949).

2. Perceptron re-weighting (Rosenblatt, 1962).

3. Back-propagation (Rumelhart et al., 1986).

Using the Same Links from Input Nodes, and New Kinds of Re-Cycling Nets

The possibility of sending feedback via the network's input nodes is also examined.

In addition, a new type of continually cycling "information transforming" network will be briefly sketched out, one where the sub-nets used for learning also serve a variety of other perceptual and cognitive functions, and where feedback is input into and processed by the same structures of nodes that handle all other types of information (Uhr, 1989b).

That comes much closer to the real situation, since feedback is simply perceived information that is considered relevant to other information: It must be transduced, distributed to the appropriate nodes, and flowed through sub-nets that evaluate it and respond accordingly - as always receiving and evaluating information, and firing accordingly.

Local Nets Needed for Hebbian Re-Weighting, Using Only Immediately-Adjacent Local Feedback

What is often called Hebbian re-weighting is probably the simplest possible learning rule:

Change the value fired by a pre-synaptic node (that is, the weight associated with the link that

represents the synapse) as a function of the strength with which the post-synaptic node fires, or the fact that it fired.

Here learning is a function of the immediate response of the unit being fired into, rather than of feedback from a more distant external environment. So there is no need to propagate information, except extremely locally, from each node that receives inputs to each of the nodes that input to it.

Specification of the Additional Processes and Sub-Nets Needed

To handle this, the value fired by the post-synaptic node must be sent back so that the weight associated with the synapse can be changed accordingly. Although this may sound trivial, an actual digital realization entails the following:

* First, each post-synaptic node must be given a new link from that node into a new processor - a "reweighting sub-network" - for each of the I "synaptic" links from the I nodes that fired into that node. This gives L (that is, up to N^2) new links and L new nodes in toto.

* Second, each of these L reweighting sub-networks must compute how much the weight it is responsible for should be modified.

This entails computing a simple function of the value that was fired into the link, the weight presently associated with the link, and the value that feedback indicates the post-synaptic node fired out. But even the simplest of functions (e.g., "raise the synaptic weight by a constant if the output value exceeds a constant threshold"; or "raise the synaptic weight proportionally to the output value") will need a micro-circuit with several nodes, to compute the function and to modify the weight associated with the synaptic link accordingly.

Arithmetic might be accomplished by using new primitive digital units that are more powerful than McCulloch-Pitts neurons - or analog devices (e.g., capacitors, photocells, or neurotransmitters). But analog devices are usually relatively special-purpose - appropriate only for what they analogize. And the underlying structures with which analogs actually effect their computations are inevitably extremely complex, since they must initiate, combine, measure, and terminate the flows in several channels where large numbers of molecules or photons are assessed. That is, although on the surface analog devices may appear to be simple, they are computing at a qualitatively lower level that is typically extremely complex.

No matter what the lower-level implementation, this would need L new links, back to L new nodes that compute L weight changes, plus L new links from and L new links to the L synapse weights.

The original 1-way links might be replaced by 2-way links - but that introduces its own set of problems: The input and output ports to and from links must be made 2-way, hence more complex. And controls must be implemented - which means still more micro-circuits - to handle the alternations between the forward flow needed to perform and the backward flow that disperses feedback.

* Third, the value originally fired into each link must be remembered until the feedback is received; then it must be forgotten.

That can be handled by storing this value into a new memory node associated with each pre-synaptic node, and giving the new processor access to that memory to compute what to learn, and to erase and overwrite it, to free it for future problems. This is the simple and straightforward black-box way that a traditional program handles temporary "read-write" memories. (Brains may handle things similarly, although analog components - e.g., capacitor-like temporary build-ups of neuro-transmitters - seem more likely.) The actual digital realization of each memory needs a whole micro-circuit, as described below, to send information to and from the store, modify and erase stored information, and control, coordinate, and trigger these processes.

Since memory is needed for only the shortest amount of time - only until the node fired into has immediately fired back - it might be handled much more simply than the memories needed by the other types of learning, which must wait until external feedback has been evaluated and the error signal has been received from some distance. For example, it might be possible to eliminate memory entirely, by giving pre-synaptic nodes new links to fire directly into the nodes that compute re-weighting - if pre-synaptic firing continues the slight extra amount of time until post-synaptic firing begins. This is a good example of a necessary addition, but one where there are many possible implementations. In any case, the additional units needed here may well be simpler than those needed by the kinds of learning examined next.

The Large Amount of Precisely Structured Additional Network Needed

Hebbian re-weighting computes a function of the form:

$$w_{t+1} = h(o_t, i_t, w_t)$$

(o=post-synaptic output node, i=pre-synaptic input node, w=weight associated with the synapse link).

The actual computations needed result in major additions: A network with L synapse-like links will now need 2L new nodes (actually micro-circuits) and 5L new links.

These include, at the minimum, the following:
M: a temporary memory for the pre-synaptic node's activity level (L such nodes in toto);
P: a processor that determines what change should be made to the link's weight (L such nodes in toto);
(C: to make that change and control processing - this will be lumped into the micro-circuit for P);
plus the new links:

i->M (L such links in toto)
M->P (L such links in toto)
o->P (L such links in toto)
w->P (L such links in toto)
P->w (L such links in toto)

(w is this link's weight, i the pre-synaptic node that inputs to it, M the new memory, P the new processor, o the post-synaptic output node, and i the total number of pre-synaptic input nodes).

Thus 2L new processor and memory nodes are needed, plus 5L new links. And each of the nodes is itself a micro-circuit of primitive nodes that may be quite complex (Uhr, 1989d).

Possibly even more striking, the added micro-circuits are structured in quite precise and intricate ways. And they must be replicated everywhere, throughout the network, giving it an extremely regular, and highly iterated, modular structure.

What each of these nodes does might alternately be dispersed over several or many nodes, in a distributed fashion, rather than be realized locally. There might also be random components, and/or fault-tolerant error-correcting capabilities (e.g., partially redundant feature detectors, majority-rule sub-nets). But although the iterated regularity may be obscured by distributed and random processes it must still remain underneath, since it is necessary for success.

Storing and Changing Memories Using Digital and Analog Devices

Today's connectionist systems treat memories as black boxes, without specifying the actual networks needed. They can be augmented to actually handle memory with micro-circuits like the following:

Have the node, a, that actually stores the memory continually fire a unit value into itself, so that it continues to fire out the same weight (whatever value is associated with this link); it also fires into a second node, b, that will fire out this weight into d only if a third node, c, also fires into it (this third node must be triggered, whenever appropriate, by still other nodes, e, in the larger network). This gives the net:

a->a
a->b
b->d
c->b
e->c

Alternately, the node into which the memory node constantly fires can be inhibited by the third node (c-[inhibit]->b), so that only when other nodes in the larger net fire into and inhibit that third node, c, will the second node, b, be dis-inhibited and fire out the first (memory) node's, a's, value.

This kind of unit now "remembers" the weight that it fires out. (Note that the weights associated with links are changeable - much like a memory register in a computer - which is what makes learning possible, and now also makes temporary memory possible. To be realized physically, the node that stores the remembered value must itself be a micro-circuit of more primitive units like - in digital realizations - B 2-state flip-flops or transistors strung together to store B-bit encodings of numbers or symbols.)

Note that the weights associated with links actually must be stored in similar structures of memory nodes when they are implemented digitally.

The node that computes what changes to make can be linked into, so that it fires into, an additional node that sums this and the present memory, to compute the slight change to be made in the direction of reducing error, and then stores this result into the memory. To erase a memory, still another node might be added into which the memory fires that fires, hence adds in, the same value, but inhibiting - that is, a value whose sign has been reversed, hence is subtracted. Thus to actually realize what is typically treated as a single memory, a whole network of properly linked black box nodes is needed.

Alternately, analog devices can be used - e.g., a material whose resistance can be set so that the current passing through it represents a fixed value, but where this resistance can be changed, to represent some new desired value. This is equivalent to replacing an adder circuit in a digital computer by a capacitor whose output (current or charge) represents the sum of its inputs. The problems with analog devices are that they don't always exist, they are often special-purpose, they can't always be fabricated into the total system economically and reliably, and although their resolution is high their precision is low, and variable. (This last may be desirable for handling imprecise, fuzzy, noisy information, but proper exploitation of this capability would need different kinds of processes.) Possibly the biggest problem is simply that relatively little work has been done in developing densely packed analog micro-electronic devices.

The Additional Nets Needed to Handle Re-Weighting in 1-Layer Perceptrons

Hebbian re-weighting learns from values fed back from the immediately adjacent node only. More typically, as in perceptrons, feedback comes from the larger environment external to the graph, in the form of a vector of values - one for each output unit - that indicates what these output units' vector of values should have been.

First an "error signal" is computed, usually by pairwise subtracting the feedback values from the output values; this error signal is then fed back to the net's nodes. Computing the error signal is typically handled outside the network, by human beings. But it can also be done by a (rather precise) sub-net (Uhr, 1989d) that has a node into which each of the O values in the output vector, along with the corresponding feedback values, are input, that computes their signed differences, and outputs via the new links in the augmented network (to be described next) that distribute these values throughout the original behaving net.

Note that a node's output value, or at least the fact that the node fired, must now be remembered until the error signal has been computed and is finally fed back to it - that is, for a substantially longer and more variable period of time than in the case of Hebbian re-weighting. This means that the value fired must be stored in a memory node for a proper period of time, accessed and used to compute the re-weighting, and then erased (or overwritten). Therefore additions are needed to the micro-circuits that store information into memories, access these memories, erase and change memories, and coordinate all these steps. And more complex micro-circuits are almost certainly needed to determine the amount of re-weighting, which is now a function of many pieces of fed-back information.

"Perceptron" is a very general term, but it is usually taken to refer to a 2-layer network consisting of a set of input nodes linked (either completely or randomly) to a set of output nodes. Perceptron learning entails changing weights as a function of the error signal, in the direction of reducing error (e=error signal, i=value fired by the input node, w=weight associated with the link):

$$w_{t+1} = p(e_t, i_t, w_t)$$

To achieve this, the error for each unit in the output vector must be fed back to each of the links that fired into it. This means that for each link there must be a new companion link dedicated to changing the original link's weight. Each such link must carry the error signal to a new sub-net that, as for Hebbian re-weighting, has remembered just-fired values, computes what changes to make, and modifies the old weight accordingly.

Thus 1-layer perceptrons need the same additional internal structures as do Hebbian systems, plus (if the net handles this) the O nodes needed to compute the error signal for each of the O output nodes and 3O links to input the output and feedback values, and output the computed error.

Nodes for Perceptron learning will probably be substantially more complex micro-circuits than those needed for Hebbian re-weighting, since they must compute more complex functions and exercise more complex coordination and control. Although this increases complexity by only a relatively small constant (e.g., 5, 10, 100) it can be of crucial practical importance.

Global Tree-Like Nets Needed for Multi-Layer Back-Propagation

One-layer perceptrons can handle only linearly separable problems, and they are combinatorially explosive. Today's standard way of attempting to cope with these weaknesses is to build multi-layer systems. Typically, learning is handled by passing information about feedback back from layer to layer, as done by "back-propagation."

As before, this needs a new link for each old link, over which information is sent in the opposite direction, plus L new memory nodes (where control and access are handled as well as storage), to temporarily store the values fired into the L links, plus L processors to compute and effect changes to the L links' associated weights, plus links into and out of these memories and processors.

In addition, N new processor nodes (plus necessary links) must be added to execute the necessary new processes of unsquashing and computing the overall error for each parent node, linking in to N new - and far more complex - processors to determine the magnitude of the error signal to be attributed to, and distributed to, each node.

This gives the following recursively-computed multi-layer function: (squash' is the inverse of the function that mapped input to output - it unsquashes the result; P=parent layer; C=child layer; p=parent node; c=child node; (c->p)=link from child to parent; t=target value fed back; v=value fired (activation); e=error signal; w=weight associated with the link between child and

parent; dw=change to be made to the weight; f_1 computes the unsquashed error, f_2 the error for each input link, f_3 the change in weight for each input link, f_4 the new weight of each input link, f_5 the error for each child.)

DO (for all O (nodes in a layer) in parallel):
P = n {initialize the parents to the output region (n=O) }
e <- v_O - t

FOR P=n (n=O), n-1, ... 2: {move down from the output region as parents, to the input region as children}
DO (for all members of the region in parallel):
C = n-1

$ePd = f_1(e_P, squash'(v_{C(i=1..n)}))$ {unsquash the error as a function of children's values}

$e_{(c->p)} = f_2(ePd, {}^aC(i=1..n), {}^w(c->p)(i=1..n))$ {compute and distribute the error to links}

$dw_{(c->p)} = f_3(e_{(c->p)}, t, {}^vC, {}^w(c->p))$ {compute how much to change each weight}

$w_{(c->p)} = f_4(dw, w)$ {change each weight}

$e_C = f_5(e_{(i=1..n)}, {}^v(c->p)(i=1..n), {}^{dw}(c->p)(i=1..n))$ {compute new error for each child}

Note that this is somewhat more general than standard back-propagation. For example, the last step computes the new error for the child as a function of its parents' errors, all values fired, and all weight changes on the links fired into (hence their previous weights).

Each node in the behaving net (except for the input nodes) now needs two new nodes: 1. to compute the un-squashed error signal (with 2 links), and 2. to compute how to distribute this error over the weights associated with the links from this parent node's children (with 2c+1 links for c children).

These include, as for Hebbian and Perceptron learning, the following nodes (micro-circuits) and links:

M: A temporary memory (with controller) to store the child node's activity level during the immediately-preceding behave-perform cycle (L)
R: Re-weighting processors that determine and effect the changes to be made to the links' weights (L)

c->M (L)
M->R (L)
w->R (L)
R->w (L)

In addition, depending upon the particular back-propagation algorithm used, there will be a need

for 2(N-I) additional nodes, each a quite complex micro-circuit, to compute the error signal moving down through the net, and to determine how much to change each weight as a function of the context-for-changing - that is, some larger sub-structures of information that the algorithm considers. Typically, this function considers all the child values that input to a node, so that it can then sub-divide and distribute the constant total change among the weights associated with the input links. But it might consider larger contexts - e.g., all the grandchildren of the parent's parent, or the whole path that leads into, or/and leads out of, that node.

This adds the following nodes and links to the above:

U: Un-squasher nodes (N-I)
D: Processor nodes to partition and distribute the error signal propagated back to the parent among its children (N-I)

p->U (L)
c->U (L)
w->U (L)
U->D (N-I)
c->D (L)
w->D (L)
D->R (L)

A network with L links and N nodes will now need at least L+3N-O-2I new nodes and 9L+2N-O-I new links.

From the point of view of complexity, replacing each node by the necessary micro-circuit is merely a (relatively small) constant increase in size. But a physical realization will easily increase size 10- to 100-fold, or more, hence be of overriding importance from the point of view of actual practical implementations.

Continually Cycling Information-Flow Nets Need Virtually No Additional Hardware for Learning

An alternative approach - and possibly the best - is to establish a new type of continually cycling information-flow network, one that handles feedback-learning with the same mechanisms used for all other types of processing (Uhr, 1991b).

In all of the above, during the behave-perform phase firing occurs in only one direction, initiated by a single field of input nodes, and ending with output nodes. This is variously called "data-driven," "image-driven," "problem-driven," "environment-driven," or/and "bottom-up" processing.

But units firing in other, more or less opposite, directions would serve many useful purposes. The simplest, and possibly the best, way to implement this is to introduce linkings in both directions between layers, to establish cyclings. That is, for each pair of synaptically-linked nodes add one new link in the other direction. [Mixtures of any number of different types of back-linking might be used. For example, each node might link to all nodes 2, 3, or N back, rather

than 1 back, and this might be varied from ply/layer to ply/layer in whatever way desired. But although this could effect some savings in links, it would complicate processing and need more time.]

The cycle might contain one new link only, to carry firing from the output node back into the input node (plus one extra link to the weight associated with the original link, so that it can be modified - the single input node computing and triggering that modification). Or one new node - or several - might be fired back into.

This gives only 2L new links, so the size of the net can be increased 5-, 10-, or 100-fold or more, and still be no larger than the original net augmented to handle back-propagation. Different micro-circuits might be used as before to handle the different functions. But it is now possible to use all nodes and links for all types of processing, whether behaving or learning.

Back-linkings of whatever kinds immediately make possible, when used to behave-perform, the very useful types of processing variously called "model-directed," "inner-driven," "need-driven," "top-down." They also establish cycling paths that can be used for feedback-learning as well as for behaving - with little or no additional mechanism.

The brain appears to use 2-way links almost routinely. The 6-layered sheet of neurons that makes up the cerebral cortex is especially rich in cycling processes (Kuffler et al., 1984). And wherever one of the living visual system's 20 or so areas links into another area, neuro-anatomists have found links from that second area back to the first (Van Essen, 1985).

In back-linking nets, information from all directions is assessed and combined, When the nodes compute appropriate processes of these sorts, there is no need to dedicate substantial additional sub-nets to handle processing, memory, coordination, and switching for learning. For example, processes can constantly reinforce previous computations, compute running averages, merge, relax, and choose among alternatives, combine bottom-driven and top-down processes, assess whether a relatively stable state has been achieved, and choose to output - or to modify some part of the net itself: that is, to learn.

Generation Learns Simple Processes (E.g., Count, Add, Parity, Negate, Encode, X-OR) in Optimal Time

This research shows how X-OR (the logical function "exclusive-or") can be learned in the minimal necessary number of trials, producing a net with the minimal number of nodes and links, by connectionist networks that use algorithms that embed pertinent information extracted from inputs into new nodes and links that are generated when feedback indicates they are needed. Indeed, the same generation-plus-extraction algorithms will learn ANY set of 2-valued or N-valued functions over Boolean variables in minimal time and space.

This research examines why that is the case, and proves that generation will learn any set of Boolean functions of N variables in minimal (linear) time, needing at most one learning trial per function instance (that is, at most one epoch for the entire set of functions), and producing a network that is minimal in terms of number of links and nodes.

This is in striking contrast to traditional re-weighting (e.g., using back-propagation, for which comparisons are given), which is not guaranteed to succeed at all, must be given in advance all the necessary nodes, properly linked, is sensitive to its initial randomly chosen weights and the order in which training is given, and needs many more learning trials (if it learns at all).

A Brief Description of Generation

Generation (Honavar and Uhr, 1988, 1989, Uhr, 1989b) consists in the adding (and deleting) of links and nodes, in addition to (or instead of) using the traditional connectionist network learning method - the changing of weights associated with already-present links to already-present nodes. Essentially, when feedback indicates that the network's output was in error, but the network judges that changing weights is not potentially helpful (e.g., because local peaks appear to have been reached) the network changes its own structure by adding or deleting links and nodes. Depending upon the frequency of generation, this can lead to a small (near minimal necessary) or a large (near an enumeration of each possibility) number of new nodes. The number of nodes can be reduced by additional generalization and de-generation mechanisms.

Re-Weighting can continue to be used to hill-climb up gradients and to fine-tune. It can also be used to help assess the value of new generations.

Generation introduces non-linear components where these are needed (more precisely, where the network's rules for generation infer they are needed). Generation thus ranges through the set of possible non-linear interactions, establishing (this might best be thought of as hypothesising or conjecturing) those it judges likely to be of value. To the extent that the net can be given generation algorithms that make good choices, near-optimal nets, with near-minimal numbers of nodes and links, will be achieved, in near-minimal time.

To succeed, a network that learns by traditional re-weighting only must be given all the necessary nodes, plus all the necessary links, in advance. This is in the worst case impossible realistically, since it needs the combinatorially explosive number of V^N interior nodes and NV^N links for problems that need N input nodes, each ranging over V values (e.g., today's perceptual recognizers typically process images in 512-by-512 arrays, each pixel ranging over 256 grey-scale values). But whenever (as in most cases) it is not known in advance what are the necessary nodes and links, to guarantee success such impossibly large nets are needed.

In striking contrast, generation is a constructive procedure (that also serves as a constructive proof when it converges to a successful net) for building only a sufficient (and given good enough procedures for deciding what to generate necessary or little-more-than-necessary) net. (See Uhr, 1989b, for an examination of generation in terms of its generality and power, and Honavar and Uhr, 1988, 1989, for empirical comparisons where generation gives several orders of magnitude speed-ups in learning on simple recognition tasks.)

Examples of Non-Linear Problems: Learning X-OR, and Other Boolean Functions

Consider X-OR (the logical function "exclusive-or"), along with the whole set of 2-valued and n-valued Boolean functions.

X-OR has been used widely to test connectionist networks, because it is just about the simplest possible non-linear function, one on which it is inappropriate to use the only known guaranteed successful way to find solutions in connectionist networks - to hill climb up a monotnically increasing gradient by changing weights in the direction of reducing fed-back error (Rosenblatt's, 1962, "Perceptron convergence theorem").

The back-propagation ("backprop") learning rule (Rumelhart et al., 1986) was devised to handle learning in nets that have more than one layer of links, hence are capable of computing non-monotonically increasing functions (although there is no algorithm for learning such functions that has been proved to always succeed). Backprop was shown (empirically) to succeed (but with occasional failures) on X-OR, Even-Vs-Odd, addition of two 2-bit numbers, and several other simple non-linear problems - although typically needing hundreds or thousands of epochs (presentations of the entire set of possibilities).

Generation and Re-Weighting Usefully Complement One Another

Generation is designed to work hand-in-glove with re-weighting. Each generation introduces a new dimension (this can be thought of as a new variable, degree of freedom, or feature-detector); then re-weighting hill-climbs through this expanded space; when hill-climbing no longer makes progress new generations are indicated. Re-Weighting also can serve to assess the goodness of a generation, leading to the decision to modify or discard (de-generate) it.

Generation-Plus-Re-Weighting appears to be a potentially extremely powerful combination. Empirical tests (Honavar and Uhr, 1989) suggest that it offers real promise in attacking the enormously difficult search spaces for complex and difficult intelligent functions like language understanding, everyday reasoning, and perceptual recognition. In these problems, the number of, and interrelations among, basic variables is not known. But it is obviously very large, so that (except for simple problems with a great deal of known a priori information) a net that is guaranteed sufficient in advance would be far too large to handle. And although in theory it could be used, Boolean logic appears to be far too cumbersome and rigid, and lacking the potential for flexibility, fault tolerance, and generalizability of the probabilistic threshold logic that re-weighting approximates.

A Basic Algorithm that Generates Optimal Networks for X-OR and Other Boolean Functions

For the deterministic error-free domain of X-OR and other logic functions weights are not necessary - indeed they are stumbling blocks. So we will first examine learning algorithms for X-OR, and also for general sets of 2-valued and n-valued Boolean functions, that use generation alone.

X-OR and the other Boolean functions over N variables can only be handled by taking every bit of information (every variable) into account. Therefore determining what to extract becomes trivial; the network must extract everything - all the details of an input to be learned.

All this information can be stored and used explicitly, to exactly match future inputs, hence always map them to the correct output. That is what the first algorithm does, by permanently setting the weights associated with links to the extracted actual values of the inputs.

Alternately, the extracted information can be used to infer a minimal threshold - which opens up the possibility of generalizing together instances that meet or exceed that threshold. This might best be thought of as a heuristic - something that may frequently work but for which there are clearly counter-examples. The second algorithm below does this, setting the threshold to the sum of the input values, and all weights to 1 (so that future input values will pass through the links unchanged).

These simple algorithms use generation alone, with no re-weighting. Indeed, since each variable can take on only 2 values, there are no hill-climbable gradients in-between for re-weighting to climb.

The first algorithm matches nodes' inputs, rather than multiplying incoming values by weights and thresholding or squashing their sum. Matching is easily handled using weights and thresholds when, as here, all inputs are 2-valued. But for multi-valued inputs an exact match (or a match within a small interval, delta) should be used instead of a threshold match. This is most efficiently handled by nodes whose primitive operations are compare (e.g, subtract, test that the absolute difference is within delta) and logical-AND, rather than the traditional multiply, add, and threshold(or squash). Actually, these operations are already often used in connectionist networks - to compare a network's output with the feedback, in order to compute the error signal.

The second algorithm multiplies activations by weights, sums the results, and thresholds in the traditional manner. It generalizes in certain ways - but this also leads to mistakes (which it immediately corrects) and small inefficiencies. It is more complex and less elegant, although slightly more efficient when threshold functions are present. It is instructive to compare the two algorithms, since their simplicity lays bare important differences between 2-valued logic and multi-valued arithmetic.

A Match Algorithm for Generation-Learning of X-OR, and Other 2-Valued and N-Valued Boolean Logic Functions:

START with a net that has 2 input nodes, a and b, into which the current states - for convenience coded -1(=False), or +1=(True) - of the two Boolean variables, x and y, are sent from the external environment.

REPEAT until the QUIT signal is input (e.g., any symbol other than -1 or +1):

INPUT the next problem.

BEHAVE: Fire nodes through the net, until output nodes are fired. (A node fires if the value fired into each of its input links EXACTLY MATCHES the value stored for that link.)

LEARN: COMPARE: Does the net's output exactly match the correct (target) output?
IF they match, RETURN to input the next problem.
IF they differ, for EACH variable fedback that was not output:
WHEN (this happens the first time the instance is given) there was no output at all:
IF the node whose output would have been correct if it had fired is NOT already present, add

a new output node (threshold = 1) - whose firing will signify (e.g., output the name of) this Boolean function.

IN ALL CASES:

IF there is already an interior node set to fire if a=x, b=y (this can be accomplished if the link from a is weighted a, the link from b is weighted b, and the threshold is a^2+b^2), link this node to fire the value 1 into the output node that should have fired,

ELSE generate a new interior node that links and fires a value of 1 into this (new or already-present) output node, and link a and b into it, with the link from a weighted a, the link from b weighted b, and the threshold, T, set to T= a^2+b^2.

This builds a straightforward 3-layer net with (to handle any sub-set of Boolean functions over 2 variables) at most 4 interior nodes, each fired into by a unique combination of inputs, and firing out into all the output nodes that signify the functions that this combination of inputs belongs to. [Note that this net's nodes differ slightly from the more commonly used node, which subtracts the threshold from its input and fires out the difference. Instead, it always fires out if its threshold is achieved (this is actually closer to actual neurons, since they fire in all-or-none fashion).]

The first time an instance of a new function is input, an output node is established to represent that function, and an interior node is established (if it is not already there) to represent that particular instance, firing into that output node. If the feedback matches an already-present output node's output, then a new interior node is generated - if it is not already there - to represent and match the state of this input instance; this interior node is linked into the output node that represents this function, to fire it. The weights and thresholds are set so that nodes fire to exact matches only.

Consider how X-OR is learned - in 2 trials (the net is given input nodes a and b to start; then learning generates interior nodes c and e, and output node d; T=threshold; weights are shown in parentheses):

Trial 1: Input: a=-1, b=1 {the encoding of False True};
Feedback: X-OR;
Learns: a=>(-1)c, b=>(1)c, T(c)=2; c=>(1)d, T(d)=1, d=>X-OR.

Trial 2: Input: a=1, b=-1
Feedback: X-OR
Learns: a=>(1)e, b=>(-1)e, T(e)=2, e=>(1)d.

Thus this extremely simple algorithm learns to handle X-OR by building an optimal 5-node, 6-link net (starting with only the 2 input nodes), in an optimal 2 steps. (Note that there is no need for an explicit interior node to handle either TT or FF.)

Trials 3 and 4 teach the net to begin to distinguish Inc-OR (inclusive-or):

Trial 3: Input: a=1, b=-1
Feedback: Inc-OR, X-OR
Learns: e=>(1)f, T(f)=1, f=>Inc-OR.

Trial 4: Input: a=1, b=1
Feedback: Inc-OR
Learns: a=>(1)g, b=>(1)g, T(g)=2, g=>(1)f

This same simple algorithm will generate the optimal net (with at most 4 interior nodes and 1 output node for each function) in a minimum number of steps, for any set of 2-valued Boolean functions. Trials 3 and 4 have already started to do this: 3 generates the new output node, f, that outputs Inclusive-OR when TF is input. Then 4 generates the new interior node, g, that links into f and fires into it when TT is input.

There is nothing surprising about any of this - the net is simply moving toward generating 4 interior nodes to handle the 4 possible combinations of x and y: FF, FT, TF, TT, and then linking each of these in to each of the functions it belongs to - as indicated by the Ts in the following example truth-table:

x	y	X-OR	AND	NULL	yORboth	Inclusive-OR
F	F	F	F	T	F	F
F	T	T	F	F	T	T
T	F	T	F	F	F	T
T	T	F	T	F	T	T

If we continue to teach new functions, e.g., AND and Inclusive-OR, we need simply give an input for each each row that contains a T, giving the name of the function being taught as feedback. Thus generation would build a net with 4 interior and 5 output nodes that handles the truth table shown above. The desired function might be any linear or non-linear combination - e.g., "yORboth" (FT OR TT) - as well as those traditionally found useful. Note that the net may correctly fire several output nodes - e.g., both X-OR and Inclusive-OR to the input TF - when that is proper.

The above algorithm will work equally well with 3, 4, or any number of variables, so long as we give the net an input node for each variable. It will learn functions of 3-, 4-, . . . and V-valued as well as 2-valued Boolean variables, by generating an interior node for each of the possible N^V combinations that is actually input to it.

Connectionist Networks (CN) That Handle Symbols as Well as Numbers

This research (Uhr, 1991c) shows how connectionist networks (CN) can conveniently and efficiently be made to handle symbols as well as numbers. Basically, a symbol is a name (or pointer) that links to the (small to gigantic) network of information that it signifies. (It is this network of information that gives the symbol meaning - whether complex or simple, ambiguous or precise.) The CN must be able to treat the symbol in at least the following ways: identify (recognize), transduce, encode, send, receive, decode, and interpret. And it must also be able to interpret and make appropriate use of the body of information (which will usually contain symbols) to which the symbol points.

Traditional artificial intelligence (AI) systems chiefly use structures of lists of symbols, and process them serially. It is sometimes suggested that AI systems are fundamentally different from CN. But they clearly are not, since CN are based on McCulloch-Pitts neurons, and AI on Turing machines - both of which are equivalent models for a universal computer (when allowed to grow to whatever "potentially infinite" size is needed). Therefore anything an AI program can do some CN can do, and vice versa - including handling symbols. This research examines how CN can handle symbols in reasonably efficient and brain-like ways. Several different possible ways to handle symbols are examined, and the necessary micro-circuits specified.

People and AI Systems Use Symbols; Hence CN Must Be Able To

We human beings routinely use mutually understandable public symbols to communicate - as is the case right now, while I write this, or you read it. AI programs - along with virtually all other kinds of programs - use symbols (they are alternately called variables, constants, names, constructs, features, attributes, pointers, atoms, or whatever).

People Communicate and Think Using External and Internal Symbols

In addition, we human beings almost certainly use these public (call them "external") symbols to think. Most people have the impression that we routinely think at least to some extent with symbols, and it is difficult to believe that if we asked people to do so anyone would say they could not. For example, if told: "think about bicycles and their parts," almost certainly symbols like 'wheel', 'spoke', 'weld', 'ride', and so on would cross one's mind. If told: "Convert 3 and 2 to binary, then add these binary numbers and convert the result back to decimal - giving the result at each step," most people will speak or write: 11, 10; 101; 5.

Since we human beings have all that capability, it seems extremely likely that we also use additional "internal" symbols for many - if not most or even all - of our other thought processes. That is almost certainly especially true of not-conscious thinking (which includes all the complex processes that we carry out when we perceive scenes of objects, and try to remember, ruminate about things, and reason).

Traditional Connectionist Networks Do Not Use Symbols - Except Sort of

Connectionist networks are often said not to use symbols, but only numbers (chiefly for weights, activation levels, and thresholds). Everything is handled by the basic procedures that add weights and compare numbers with thresholds, firing accordingly. But connectionist networks must be capable of handling symbols, since they are equivalent to Turing machines and other universal computers, which handle symbols.

Actually, there are two ways that today's CN already at least sort-of handle symbols.

First, they routinely output what is called the "output vector" - simply a set of numbers that are fired by their output nodes. This is then often interpreted by human beings as though a symbol. For example, there might be 8 output units, each standing for one of 8 possible responses (e.g., the particular object-class the CN can judge the present input to be an instance of). The human

observer would then notice which of these 8 units output the largest number, and from this conclude that the CN chose that class, e.g., 'house'. If the correct choice was 'barn', an error signal would be input to the CN showing a negative value for the node that output 'house' and a positive value for the node that should have output 'barn'.

Effectively, each of these output units is a symbol that stands for the particular object-class. But this is really not having the CN handle the symbols fully; rather, it evades, by letting the external environment (usually humans) identify and interpret them instead.

Second, any unit can itself to some extent serve as the symbol. Consider a situation where one node should be able to fire any one of S different symbols down a link to a successor node. That is handled routinely by almost any computer program: one instruction stores a result, the next instruction fetches and uses that result; or one function passes its result directly to the next function. This is one of the central problems that this research addresses: how can we get CNs to pass symbols with similar ease?

One (awkward) way to handle this is to establish a separate node for each of the S symbols, then have the first node fire into the single node that represents the symbol that should be passed, which fires out into the successor. That is, for a set of S symbols the predecessor must link to all S, and all S must link to the successor. The predecessor must also have the necessary mechanisms to choose which node to fire into. And the successor needs mechanisms to note over which link it was fired into (hence which single node initiated the firing). But the additional mechanism needed to identify the particular link is expensive, and it is typically ignored or handled in black-box fashion, rather than by what would be an unwieldy sub-net of add-threshold units. And - as we shall see - it actually contains the mechanism needed to identify a symbol no matter how symbols are handled.

The biggest problem may simply be that handling symbols is inevitably complicated and expensive. Since each symbol is qualitatively different, it must be identified separately. This is in sharp contrast to computing with numbers, for which a few simple rules of arithmetic suffice no matter which numbers are involved. Essentially, the system must be able to represent each symbol, and compare its representation with the alternative possibilities that might be the correct interpretation for that symbol, noting when one is similar enough to conclude that it is. Each symbol must be encoded and searched for separately, since there is only an arbitrary relation between symbols, and between symbol and what is symbolized.

Symbols Vs. Numbers: Qualitative Vs. Quantitative Vs. Mixed

Whereas symbols need large tables of pointers plus processes that match and identify them and access what they point to, numbers merely need a few arithmetic processes (a.g., add, subtract, squareroot).

Thus a million word dictionary needs a million entries, linking each word to its definition, plus the processes that handle things. But to do arithmetic no entries are needed, but only the processes. To add or subtract is the same no matter what the numbers are.

There will sometimes be a need to convert numbers to and from external symbols like five, or 5, or the processor's internal representations. This - as always - needs a symbol table, with an entry for each external symbol that is linked to a process (the processes might be digital counters, or analog capacitors, or sinks into which tallies, fluids, snowflakes or gumdrops drop).

But even here numbers can be handled with far greater efficiency, since they can be composed or decomposed using simple grammar-like rewrite rules. For example, one, two, three, four, five, six, seven eight, nine, ten, tewnty, thirty, forty, fifty, sixty, seventy, eighty, ninety, and hundred (plus a few rewrite rules that combine them appropriately) are enough to encode all the integers from one through nine hundred ninety-nine.

Numbers can be computed by the arithmetic processes alone; their quantitative relations are completely captured by computing them. Symbols are qualitatively different; each must be specified separately.

It is important to note that there are many possibilities in between. For example, the primate visual system has families of 32 or so feature detectors sensitive to bar-edges at different slopes, centered roughly 12^{o} apart. Each appears to be handled by a different (possibly overlapping) sub-net of neurons - the equivalent of a different entry in a symbol table (albeit one that makes complex less-than-exact matches). But because of the relation between them, they could be handled as follows: Represent only one of those sloped bar-edges (e.g., as entries denoting spots in a 2-dimensional array, or using an equation). Input this to some appropriate process that rotates it (e.g., a table, or an equation).

Essentially, whenever the different members of a set are related, that relation can be handled by a process that then generates the members. When - as is the case with pure symbols - there is no relation, each member must be specified individually.

The "Problem" of Icons Vs. Symbols Clarified

Perceptual recognition - and how it handles and uses icons and symbols - throws instructive light on these issues from a slightly different perspective.

A recognizer must input an image array of intensity values (e.g., the amount of light at each tiny but finite point in a 2-dimensional image, giving a grey-scale representation of the sort found in a black-white photo). It then processes this input image. Finally, it must output a set of symbols (e.g., the name or names of the objects in the input scene, along with pertinent descriptive information). The conversion from what is often called the (numerical) "icon" to verbal symbol becomes a major problem (but see Uhr and Schmitt, 1984, which shows how this can usefully take place in small steps).

The input is often called the "icon" ("picture" or "image" would be more appropriate, since icons are actually schematized abstractions). In any case, an icon of this sort is a 2-dimensional array of spots containing numbers - each number reflecting the amount of light at that spot.

The output is usually symbolic - something like 'chair' or 'wooden chair - but bench-like (be-

cause very wide)'.

Researchers working with the raw image (whether to enhance it, pick out features, or recognize objects) often rely heavily, if not exclusively, on numerical operations. In sharp contrast, most researchers attempting to use traditional AI techniques chiefly use symbolic representations, and concentrate on the "higher levels" of processing (e.g., combining the legs and other parts of a chair into to the chair). This has led to an often expressed feeling (e.g., Pylyshin, 1973) that the iconic image must be processed numerically, then at some point transformed into a symbolic representation. Where that point is, and how to make the transformation, appear to pose major problems.

But it simply is not true that the input image is numeric, or that the output is symbolic. Each is both.

A simple way to demonstrate this is to examine what happens when a color image, rather than a grey-scale image, is input. Now the input image is a 3-dimensional array, where the 3d dimension has 3 values - the 3 primary colors Red, Green, and Blue (RGB).

The point is, the recognizer never really inputs a numeric value, it inputs a (possibly numeric) value for some attribute, some variable - that is, some symbol. (Remember - attribute, variable, symbol, name, thing, entity, feature, pointer - these are all for the present purposes synonyms.) The attribute is often built in and assumed - hence not noticed. It is known only implicitly, and it is easy to ignore it entirely.

Since grey-scale images are arrays of values ranging over only one single variable (total intensity of visible light integrated over all frequencies), there is no need to specify, or even think about, the attribute. Hence the fact that the attribute "grey-scale" exists implicitly is usually overlooked.

The dual attribute-value nature of the information becomes apparent as soon as the recognizer begins to process the image - even a grey-scale image. It almost immediately begins to look for such "low-level" attributes as gradients, and edges of different slopes; and then for more complex features like angles, curves, and textures. All these attributes are symbols that must be noted.

Often they are noted only when the recognizer decides that they are present and - since they can either be present or absent - it is sufficient simply to list the feature-name; so it appears that there is only a symbol, but no value. That is, rather than always list either verticaledge=1 or verticaledge=0, the system need only list verticaledge (or the null string).

Often a value is associated with a variable (sometimes several - numbers or/and symbols). Thus a feature-detector might output something like: verticaledge=9; verticalcurve=2 (indicating, where values range from 0 to 9, a very strongly implied vertical edge, and also a weakly implied vertical curve). Or the detector might output, e.g., verticaledge=faint; or verticalcurve=(9, shallow).

Now these attribute-value pairs are very convenient little structures of information that can be

handled in several important ways. The system can choose between them by examining their values and outputting the attribute associated with the preferred (e.g., largest). Or it can sort and collect this information, by examining the attributes and combining the values of all the instances of the same attribute.

At the highest level, a recognition system must decide which among alternate possible symbols to output. This is typically handled by examing the combined value for each of the alternate possible variables (the possible output names), choosing the max, and outputting the associated name.

Thus the basic primitive, at every level from lowest to highest, is neither a number nor a symbol, but a symbol-value pair. Where there is only one possible symbol it can be ignored; when a symbol can have only two possible values, signifying existence or absence, the symbol alone will suffice. And a "complex" symbol like 'window' or 'house' differs from a "simple" symbol like 'angle', 'gradient', or 'grey-scale' only because the nets it points to are larger and contain much more semantic information.

Augmenting Connectionist Networks To Handle Symbols

Connectionist networks can be made to efficiently handle symbol-value pairs as follows: Rather than have nodes capable of computing only enough primitive functions to give general-purpose networks (e.g., ADD, NOT, THRESHOLD), for efficiency some nodes should be able to compare two values, thus computing a MATCH. MATCH entails SUBTRACT (exactly like ADD, except one of the quantities is treated as though negative); NEGATE if the result was negative; TEST against a THRESHOLD to assess whether this discrepancy is small enough (actually, this entails subtracting the result from the threshold and testing for negative) - and finally firing out if this sequence of processes succeeds.

These can all be built from any general-purpose set of primitives - including AND, NOT, and THRESHOLD, or the traditional CN primitives. These micro-nets of nodes (consider them black-box sub-nets) must further be strung together appropriately to handle whatever whole string of primitive firings is needed to encode each b-bit symbol, and to pair together the attribute and the value.

At a still higher level, S (S = number of logS-bit different symbols) of these black-box sub-nets that look for MATCHes can be combined into a code book that finds the particular match. This can be handled serially, by stringing each matcher after the next, and passing the symbol-to-be-matched through this pipe-line serially, in O(S) time. (S output links will also be needed, along with nodes that decide to initiate firing, so that the successful node can send notification that it succeeded.) Or it can be handled entirely in parallel, in 1 time cycle, by adding S links, one to each matcher, and sending the symbol to be matched to all matchers in parallel. Or it can be handled in any desired mixture of parallel and serial, using tree-structured nets that combine both of the above techniques.

The above techniques need O(kSlogS) nodes, and O(kS) time for the serial implementation and O(k) time for the parallel implementation. Often more efficient is a discrimination tree (also

called a sorting net). This makes a 20-questions-like sort, moving down a tree from the first bit to the last, which points to the encoding. It needs only O(kS) nodes and O(klogS) time.

Plausible Neuronal Implementations of a Sub-Net that Matches Code Book Entries

The following sketch out several ways that these black-box functions can be achieved.

A Single Common Code Book at Each Node: Possibly the most straightforward way to implement a network that handles symbols is with a code book that contains each of the symbols that might be encountered, plus the ability to scan that code book for matches with patterns coming into the sub-net where the code book resides. That can be handled by a whole set of new sub-nets, one for each code book entry. Each sub-net has built into it the pattern for one entry, and subtracts this pattern from the incoming pattern, eliminates a negative sign, and fires out if the absolute difference is below some (small) threshold.

Each entry's sub-net has the capability of matching. All sub-nets fire into whatever nodes the original node fired into (possibly an intervening sub-net chooses the single max when more than one code book entry fires out).

This will enormously increase the size of the net. To build a net where each and every black-box node is able to encode and decode S symbols, each sub-net representing a black-box node needs: O(kS) nodes to decode, plus O(kS) nodes to choose the decoding; plus O(kS) nodes to encode, plus O(kS) nodes to choose the encoding. (Each of these steps needs O(klogS) time.) This turns an N-node network into an O(kSN) Node network.

Thus for a network of the complexity of a human being, who uses at least hundreds of thousands of symbols (the words in a dictionary), this multiplies the size of the original network that does not handle symbols by 10^5-10^6, or more.

A Single Common Code Book At Only One Location: An alternate implementation would handle things much as do conventional computers, storing the code book only once. That would cut down from O(kSN) to N+kS nodes. But O(2kSN) new links would be needed, to access this single code book from any other node. In an actual physical implementation, some wires would be much longer than others, raising additional problems in driving and synchronizing pulses. And there would be enormous problems (possibly at least partially solvable by adding still more nodes, but probably necessitating substantial slow-downs) in handling the bedlam of contentions for this single precious resource.

Since conventional computers can access any part of "main" memory in one instruction, and are serial, hence limited to executing only one instruction at a time, they evade these problems. (But these are major causes of the limitations of serial stored-program computers that have been called "the Von Neumann bottleneck." Computers large and fast enough to handle realistically hard AI problems like visual recognition need many processors all working in parallel, and cannot store everything in a single commonly accessible memory.)

Fortunately, more efficient implementations exist.

Small Partial Code Books That are Shared By Adjacent Nodes: There is no need to have the entire code book at every node. All that is needed is for each node to have entries for only those symbols for which those nodes directly connected to it have entries. That is sufficient so that whatever encoding the sending node produces the receiving node will be able to decode.

This reduces complexity to O(kpN) nodes, where p is the average number of entries in the partial code books.

Parallel-Hierarchical Encoders: Encodings and decodings can be effected using serial hierarchies of small parallel nets. A W-symbol pattern can be processed in $O(\log_C W)$ serial moves up a tree-hierarchy, when chopped into C-length chunks. The depth of hierarchical processing can be controlled by choosing the size of C. Thus much longer "words" (e.g., sentences, and 2-dimensional images) can be handled with efficiency and speed by decomposing them in this way. Indeed, this basically handles symbols by using pyramid vision techniques (e.g., Tanimoto and Klinger, 1980; Uhr, 1987) to recognize them.

Combinations of Shared, Partial, and Parallel-Hierarchical Code Books: Rather than a single common code book, the system can have several, or many. Rather than small local code books, it can have small, or somewhat larger, code books that are linked to by local clusters of processors. Parallel-hierarchical structures can be used to handle relatively large and complex encodings, and simpler parallel or serial structures for small sets of symbols. Different ones of these structures can be used, as appropriate, in different parts of the network.

Discussion, Conclusions, Summary

Connectionist networks are general-purpose; therefore they obviously can handle symbols and lists, and structures of any sort. Yet it is widely thought that they cannot. The problem appears to be that handling symbols is inherently complex and expensive, and it can be extremely complex to process the information symbols point to. But these are necessary aspects of the complexity that networks must confront.

Each symbol is qualitatively different, hence must be identified separately. This introduces major additional costs when using connectionist networks. Since the necessary mechanisms are built into conventional stored-program computers - e.g., to build and access stored symbol tables - it is easy to overlook their complexity.

Restricting connectionist networks - to only add, multiply, and/or threshold numerical values - is equivalent to insisting that computer programs use only some very simple and restricted (yet general-purpose) set of instructions (e.g., a minimal Turing machine set like FETCH, STORE, SHIFT, IF x THEN y; or an even lower-level set of primitives, like NOT-OR). They are sufficient (indeed, NOT-OR is widely used as the primitive building-block for general-purpose computers), but quite frustrating to work with or to understand.

Adding higher-level and more powerful black-box functions is equivalent to adding more powerful instructions to a computer's primitive set (e.g., MULTIPLY, FLOATING-POINT MULTIPLY, in addition to ADD and LOGICAL-AND).

The black-box function can always be translated into and replaced by an equivalent network of simpler functions. Ideally, this should be a natural translation, one that results in an efficient (and - if this is one of the goals - brain-like) network. Thus connectionist networks that handle symbols can be realized quite straightforwardly if one uses types of nodes that can match (hence identify) symbols.

Matching can be a very simple analog operation - e.g., for an optical device that shines light through two templates (a positive and a negative), and focuses it on a collector that integrates, thus summing the amount of mis-match. It is also the routine operation for conventional stored-program computers, where built-in match instructions are executed on universally accessible tables of symbols (the code books).

But matching is expensive to realize using networks of simple add-threshold-multiply units. A whole micro-circuit of such nodes is needed to encode or to decode each symbol. And matching and choosing among alternatives entail carefully synchronized sets of operations in several sub-nets of nodes.

The basic symbol is used simply to point to other symbols - to build and access the strings, lists, and more complex list-structures that contain the symbol's meaning (whether simple or complex). The richness of its meaning is a function of, and lies entirely in, these accessible structures of information. (See Uhr, 1989d, for an examination of network structures and processes for traditional serial, and also network-oriented parallel, list-processing procedures.) When the symbol serves as a variable there must be additional capabilities to assign and change its value. For this, a memory that can be accessed and changed is the simplest solution; but memories are expensive. The variable's value will be determined by those structures of nodes in the larger net that fire into it. Without a memory the system must work much like an applicative language (e.g., the original Pure Lisp), by having the larger net continually re-fire the same value into it. (See Uhr, 1989c, for an examination of how memories and variables can be handled in networks.)

Code books must be available where they are needed. This necessitates combinations of the following:

Keep several, or many, copies spread throughout.

Add enormous numbers of often long links (on the order of SN for an N-node net that uses S symbols).

Or - almost certainly the best - as much as possible store only very small code books with nodes (paying the price that each node can handle only a very small number of symbols).

Numbers are simply values that certain symbols (e.g., 1, 2, ...; one, two, ...) point to that are related by the simple arithmetic procedures. A single procedure, like ADD or DIVIDE, can be applied to any set of numbers; the procedure's output is the result. So simple sequences of numerical processes typically have no need for code books.

When there is absolutely no computable relation between the members of a set, a separate encoding is needed to identify each. Letters and word-stems are good examples. Each must be looked for and matched using a qualitatively different entry in a code book. This entails computing the absolute difference between the symbol to be identified and each of the entries in the

code book, and evaluating whether it is small enough (using either a threshold, or a more complex procedure that chooses the closest among all the alternatives).

It is often possible to combine procedures (as for numbers) and tabled code books (as for symbols). For example, a procedure can rotate, magnify, translate, and in other ways transform a table that contains one edge, one 45^{o} angle, and so on. Thus the simple procedure can turn a single symbol into a whole family.

Connectionist networks can indeed handle symbols; but at the cost of additional complexity. Essentially, each node needs a small micro-circuit of traditional nodes that can encode and decode each symbol it must handle (or links that access such a micro-circuit that is used in common by several, many, or all nodes).

Almost always, a symbol has an associated value (e.g., a weight or probability), and a numeric value has an associated symbol (e.g., greyscale or some other feature). So it may often be desirable to build nets that link nodes in pairs - one to handle the symbol (attribute), the other to handle the number (value).

References

Honavar, V. and Uhr, L., Experimental results indicate that generation, local receptive fields and global convergence improve perceptual learning in parallel networks. *Proc. Int. Joint Conf. on Artificial Intell.*, 1989.

Honavar, V. and Uhr, L., Coordination and control structures and processes: possibilities for connectionist networks (CN), *J. Expt. Theor. Artif. Intell.*, 1990, *2*, 277-302. (a)

Honavar, V. and Uhr, L., Brain-structured networks that perceive and learn, *Connection Science*, 1990, *1*, 139-159. (b)

Uhr, L., Increasing the power of massively parallel networks, by improving structure, processes, learning, *Connection Science*, 1990, *2*, 179-193. (a)

Uhr, L., Appropriate structures for reducing complexity of perceptual recognizers, *Behavioral and Brain Sciences*, 1990. (b)

Uhr, L., Specifying useful sub-net and micro-circuit structures for parallel networks, *Computer Sciences Dept. Tech. Rept.*, 1991. (a)

Uhr, L., Cycling logarithmically converging nets that flow information to behave (perceive) and learn. In: *Neural Networks for Human and Machine Perception*, Harry Wechsler (Ed.), 1991, in press. (b)

Uhr, L. Connectionist Networks (CN) That Handle Symbols as Well as Numbers. *Computer Sciences Dept. Tech. Rept.*, 1991. (c)