



Implementing Recurrent Back-Propagation on the Connection Machine

E. M. DEPRIT

*Concept Development Branch
Spacecraft Engineering Department*

December 2, 1988

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.			
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NRL Report 9167			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION Naval Research Laboratory		6b. OFFICE SYMBOL (if applicable) Code 8342	7a. NAME OF MONITORING ORGANIZATION		
6c. ADDRESS (City, State, and ZIP Code) Washington, DC 20375-5000			7b. ADDRESS (City, State, and ZIP Code)		
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Office of Naval Research		8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8c. ADDRESS (City, State, and ZIP Code) Arlington, VA 22217			10. SOURCE OF FUNDING NUMBERS		
	PROGRAM ELEMENT NO. 61153N14	PROJECT NO. RR014-02-41	TASK NO.	WORK UNIT ACCESSION NO. DN155-017	
11. TITLE (Include Security Classification) Implementing Recurrent Back-Propagation on the Connection Machine					
12. PERSONAL AUTHOR(S) Deprit, E. M.					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 1988 December 2		15. PAGE COUNT 112
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	Neural networks		
			Recurrent back-propagation		
			Connection Machine		
19. ABSTRACT (Continue on reverse if necessary and identify by block number)					
<p>Pineda's Recurrent Back-Propagation algorithm for neural networks has been implemented on the Connection Machine, a massively parallel processor. Two fundamentally different graph architectures underlying the nets were tested—one based on arcs, the other on nodes. Confirming the predominance of communication over computation, performance measurements underscore the necessity to make connections the basic unit of representation. Comparisons between these graphs algorithms lead to important conclusions concerning the parallel implementation of neural nets in both software and hardware.</p>					
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL E. M. Deprit			22b. TELEPHONE (Include Area Code) (202) 767-2818	22c. OFFICE SYMBOL Code 8242	

CONTENTS

1. INTRODUCTION	1
2. RECURRENT BACK-PROPAGATION	2
3. NETTALK ARCHITECTURE	5
4. TOMBOULIAN ARCHITECTURE	9
5. PERFORMANCE	15
6. CONCLUSIONS	19
7. ACKNOWLEDGMENTS	20
8. REFERENCES	20
APPENDIX A – Sample Nets	23
A1 Nettetalk Architecture	23
A2 Tomboulian Architecture	26
APPENDIX B – Timings	29
APPENDIX C – *Lisp Code for Nettetalk Implementation	37
APPENDIX D – *Lisp Code for Tomboulian Implementation	65

IMPLEMENTING RECURRENT BACK-PROPAGATION ON THE CONNECTION MACHINE

1. INTRODUCTION

The recent resurgence in connectionist models of cognition has been spurred by exciting advances in parallel computing. As computational models of cognition, neural networks appear particularly well suited to fine-grained parallel processing. Researchers exploit powerful new parallel processors to push the bounds of size and speed in their models. Indeed, this process must logically lead to implementation of massively parallel networks in Very Large-Scale Integrated (VLSI) technology.

The Connection Machine (CM) provides a unique test-bed for the exploration of neural network models and their underlying graph architectures. The CM is a "Single Instruction Multiple Data" (SIMD) parallel processor. It consists of up to 64K one-bit processors arranged in a 16-dimensional hypercube [1]. The processors communicate through a flexible connection scheme, allowing the machine to be configured readily to match the structure of the problem [2]. By spreading the problem data over the entire set of processors in a proper manner, CM programs may exceed the performance of conventional supercomputers like the Cray-2 or the ETA-10.

Among the many computational paradigms for neural networks, recurrent back-propagation (RBP) presents unique advantages for parallel implementations in both software and hardware. The algorithm is due to Pineda [3,4]. It treats a neural network as a dynamical system where the behavior of the network obeys a system of coupled differential equations without making any distinction between input and output nodes. Thus, RBP can obviously be rendered in parallel since the network's global behavior results from homogeneous nodes performing only local computations.

Furthermore, by requiring the network to perform integrations where only the steady state solutions are of interest, the RBP may be realizable in analog, rather than more complicated digital VLSI technology. In analog circuitry, the network would reach steadystate asynchronously after the presentation of inputs; whereas in digital circuitry, complex synchronization would control the integration of the feed-forward equations. Studying the behavior and implementation details of this algorithm on the CM will give insights to the problems facing the designers of a neural network chip.

This report presents two implementations of the RBP algorithm on the CM. The first one uses the graph architecture proposed by Rosenberg and Belloch [5] for Nettek. In their scheme, connections constitute the basic unit of representation, with most of the processors in the CM acting as connections rather than units. The results from Nettek show that extremely large nets may be simulated by taking advantage of the unique routing and virtual processor features of the CM [6,7].

Manuscript approved September 12, 1988.

The scheme of Rosenberg and Blleloch, however, relies heavily on the sophisticated routing hardware of the CM. Thus, another implementation was tried where nodes form the basis of representation. Graph algorithms developed by Tomboulian [8,9] provide a method for embedding and using arbitrary directed graphs on SIMD machines much less capable than the CM. Tomboulian considers a network architecture of exceedingly simple processors, smart memories, which can communicate only through connections to a small set of nearest neighbors. In this regard, the Tomboulian algorithms may provide a link from a general parallel processor implementation to a possible hardware implementation of RBP. Having done away with the need for the complicated routing hardware of the CM, one could now think of building a network of smart memories coded to act as a neural net, all residing on a single chip.

By comparing these two schemes, one is able to draw conclusions concerning the parallel implementation of RBP in both software and hardware. The questions to be examined are the relative importance of computation and communication and the effectiveness of net representations based on either connections or nodes. These questions of graph representation must be answered if neural networks are to be realized successfully in VLSI technology.

Section 2 reviews the RBP equations for both continuous mapping and associative memory nets. In Section 3 the original Nettalk scheme is extended to the general connectivity nets of RBP. In Section 4, Tomboulian's graph algorithm is extended to apply it as a basis for communications in RBP nets. Section 5 presents timing experiments performed on both implementations, and Section 6 draws conclusions concerning the effectiveness of these two schemes and their implications in regard to hardware implementations of neural networks.

2. RECURRENT BACK-PROPAGATION

In contrast with Rumelhart, Hinton, and Williams [10] who specify the δ algorithm for discrete, feed-forward networks, Pineda [3,4] treats neural networks as dynamical systems, more precisely as continuous dynamical systems with arbitrary connectivity. The behavior of these recurrent networks is governed by systems of coupled differential equations.

A continuous mapping net consists of input units, hidden units, and output units. Input signals are delivered to the input units; the differential equations propagate the signals through the net. The purpose is to obtain the activation levels at the output units.

For the feed-forward equations, Pineda takes the differential system

$$\frac{dx_i}{dt} = -x_i + f(u_i) + I_i,$$

where $f(\xi)$ is the logistic function $(1 + e^{-\xi})^{-1}$. Without the nonlinearity introduced by the logistic function, the network could learn only linear maps. The variable x_i represents the activity of the i th neuron; I_i is the input ($= 0$ if the neuron is not an input unit). The coupling among neurons is introduced at the i th neuron by the term

$$u_i = \sum_j w_{i,j} x_j.$$

Pineda chooses the right-hand members so that the solution tends to a constant value x_i^∞ for any choice of the initial conditions.

The network is trained to learn a given mapping from a set of input vectors (I_i) to a set of target vectors (T_i). The weights ($w_{i,j}$) are adapted by least-squares fit to minimize the error function

$$E(x_i) = \frac{1}{2} \sum_i J_i^2,$$

where J_i is equal to $t_i - x_i$ if i is an output unit, to 0 otherwise. This is done by gradient descent on the error function; thus the weights are adjusted in a direction opposite the gradient of the error function according to

$$\frac{dw_{i,j}}{dt} = -\eta \frac{\partial E}{\partial w_{i,j}}.$$

The learning rate η must be between 0 and 1. Normally small values for η are chosen to ensure that the gradient descent converges; nonetheless it must be said that larger values of η may speed the learning. The difficulty resides in obtaining the partial derivatives $\partial E / \partial w_{i,j}$. Pineda proposes to do it locally by defining a second system of differential equations to propagate the error corrections throughout the net.

In the back-propagation equations,

$$\frac{dy_i}{dt} = y_i + f(u_i^\infty)(v_i + J_i),$$

where

$$u_i^\infty = \sum_j w_{i,j} x_j^\infty \quad \text{and} \quad v_i = \sum_r w_{r,i} y_r$$

while $J_i = t_i - x_i$ if i is an output unit, 0 otherwise. Pineda proves that the gradient update is such that

$$\frac{dw_{i,j}}{dt} = \eta x_i^\infty y_j^\infty,$$

y_j^∞ being the steady state for the error signal at the j th unit. Let $w_{i,j}^n$ be the weight on the connection between units i and j after n training iterations. To begin with, $w_{i,j}^0$ is chosen at random in a small interval around zero. Then, $\Delta w_{i,j}^n$ is the weight change specified by the n th iteration, and the next iteration adopts for weights the quantities

$$w_{i,j}^{n+1} = w_{i,j}^n + \eta \Delta w_{i,j}^n + \alpha \Delta w_{i,j}^{n-1}.$$

Adding the momentum term $\alpha \Delta w_{i,j}^{n-1}$, where α is chosen empirically between 0 and 1, was first proposed by Rumelhart, Hinton, and Williams with the suggestion that it damps out oscillations and keeps the weight corrections going in one direction, thereby speeding up convergence in the network.

In addition to continuous mapping nets, associative memory nets will be processed in the CM. An associative memory net consists of visible and hidden units. The input and output of the net is read from the activation levels of the visible units. In this scheme, a master network and a slave network have the same topology and share the same weight space. The master network, obeying a constrained dynamical system, is trained on a set of target vectors representing the memories to be stored. The slave network is not trained; it is used to recall the stored memories. Perturbed versions of the memories are presented to the visible units, the slave network is allowed to reach steady state, and the original stored inputs are recovered. Thus the slave network has the same dynamics as a continuous mapping net, and the dynamics of the master network must

be constrained to allow introduction of several basins of attraction (fixed points) into the weight space.

The feed-forward equations for the master net have the form

$$\frac{dx_i}{dt} = -x_i + f(u_i),$$

where $u_i = \sum_j w_{i,j} z_j$ with z_j constrained to be t_j if j is a visible unit, otherwise equal to the activation level x_j if j is a hidden unit. In accordance with this definition, the modified back-propagation equation for the correction signal at the i th unit becomes

$$\frac{dy_i}{dt} = -y_i + f'(u_i^\infty)(\vartheta_{iH} v_i + J_i),$$

with $v_i = \sum_r w_{r,i} y_r$. The difference between the target value and the activation level J_i is either $t_i - x_i$ if i is a visible unit or 0 if i is a hidden unit. To ensure that correction signals are received only from hidden units and not from visible units, one introduces the factor ϑ_{iH} , which is either 0 if i is a visible unit or 1 if i is a hidden unit.

Similar modifications to the continuous mapping case yield the gradient update in the master net:

$$\frac{dw_{i,j}}{dt} = \eta y_i^\infty z_j^\infty,$$

where y_i^∞ and z_j^∞ are respectively the constrained activation level of the i th unit and the correction signal of the j th unit, both in the steady state. Obviously, the weight update equation takes the same form as for the continuous mapping net.

The slave net in turn is characterized by the unconstrained dynamical system

$$\frac{dx_i}{dt} = -x_i + f(u_i),$$

similar to that of the continuous mapping net.

The same RBP routines serve for both types of nets on the CM since only small modifications separate the equations for the continuous mapping from those for the associative memory. Admittedly, the RBP algorithm requires numerical integration of coupled systems of differential equations. Nevertheless, these equations converge rapidly; moreover, only their steady state solutions are of interest. Hence, however crude, the Euler method suffices to quickly solve feed-forward, back-propagation, and weight update equations.

The numerical integration depends critically on the solutions of the differential equations converging rapidly to steady-state solutions. The well-behaved nature of these equations also points out the usefulness of RBP for VLSI implementation. The feed-forward and back-propagation equations could be realized by analog circuitry. Inputs would be presented to the feed-forward circuits, and the output would be read after the circuit reaches equilibrium — eliminating the need for digital circuitry to enforce timing constraints.

3. NETTALK ARCHITECTURE

As previously mentioned, the simple homogeneous computations of the RBP algorithm lend themselves quite naturally to parallel processing. Nevertheless, considering the enormous combinatorial complexity inherent to a neural net, special attention must be paid to the representation of the net inside the computer. This report offers two representation schemes. The first is based on Rosenberg and Blalock's implementation of Nettek. Through their close association with researchers at Thinking Machines Corporation, these authors gained an intimate understanding of the insides of the CM. As a matter of fact, one cannot appreciate the architecture for Nettek without understanding pertinent details about communications and virtual processors as handled by a CM.

In the CM, processors communicate in several ways. The basic ones are the router and the scan operations. Through the router, processors read from the memory of any other processor or write into it. The principal power of the CM lies in the router; it makes of the machine a gigantic telephone system in which processors can communicate by knowing each other's cube address, that is, their phone number. This centrex becomes a computer thanks to the the routing cycle that combines multiple values sent to a single processor according to various logical and arithmetic operations. The router, however, trades speed for flexibility. In problems where the communications pattern is localized, the scan operations provide a generally faster communications scheme. In scanning, values in contiguous segments of processors are easily copied or summed. In general, the scan operations operate more quickly than do the router operations.

Another important feature of the CM is the ability to use virtual processors. Although a full CM contains only 64K physical processors, each physical processor may be multiplexed into some power of two virtual processors. Virtualization of the machine is accomplished in microcode and is invisible to the applications programmer. Consequently problems requiring more processors than physically available may still run on the machine. An interesting side effect of the virtualization is that communication operations become more efficient for higher virtual processor (VP) ratios. A machine running a VP ratio of 4 will probably execute code less than 4 times as slowly; the reason is that more communication operations are performed on-chip. This increased efficiency is especially true with scan operations. In particular, scan operations become very advantageous for simulating large networks when the CM is configured with high VP ratios.

The Nettek scheme represents nets in the CM with one processor per unit and two processors per connection. Each connection corresponds to a fan-out weight from its source unit and a fan-in weight to its destination unit. There is a processor for the fan-in weight and another one for the fan-out weight, and each unit is preceded by its fan-in weights and followed by its fan-out weights. Processors in a fan-in/fan-out pair are linked through their processor address so that values may be passed by a global send operation. This interleaving of weights and units allows one to take advantage of the very fast segmented scan operations provided by the CM. Figure 1 shows the layout of a very simple or-net on the machine. The Nettek scheme as originally designed by Rosenberg and Blalock considered only feed-forward, layered networks, but this Nettek architecture extends naturally to the general connectivity nets treated by RBP. This extension is made obvious in Fig. 1 where, at the bias node e , one of the fan-out weights feeds back to a fan-in weight.

Construction of these nets may seem daunting at first glance, but actually it is simple because of the powerful sorting facilities on the CM. The net is built by first loading in all fan-in weights, then loading the processors representing units, and finally loading the fan-out weights. Next, the

processors in the net are ranked according to the following key — either the unit address for units, the to-unit address for fan-in weights, or the from-unit address for fan-out weights. The sorting operation in this case is very fast because the CM performs all sorts in logarithmic time. In the final step of the construction the units, fan-in weights and fan-out weights are rearranged according to the ranking produced by the sort. Figure 2 shows the construction steps on the standard net introduced in Fig. 1. The reshuffling involved in the last step is the most time-consuming part of the construction operation, since many bits of information must be communicated through the machine by a general router cycle. Nonetheless the scheme allows extremely large nets to be constructed in a matter of seconds.

An explanation of the feed-forward cycle in Nettalk will serve to motivate the interleaving scheme of fan-in and fan-out units. Although the RBP feed-forward equation is solved, units must propagate their activation levels to all connected units. First, all units “copy-scan” their activation levels to their fan-out weights for the latter processors to form the product $W \circ X$. With a general “send” operation, the fan-out weight sends the result to their corresponding fan-in weights. A “plus-scan” operation then sums these activation values into the units. Thereafter, the next step in the numerical integration is performed locally in each unit. The integration loop is repeated until a steady state solution is reached. Figure 3 details the cycle of copy-scan, send, and plus-scan that is central to the solution of the feed-forward equations. Moreover, Fig. 4 exhibits the *Lisp [11] code that implements the algorithm. As expected, the solution of the back-propagation follows a similar pattern, with correction signals being propagated backward from units to their connected units. The weight update equation is solved in each of the fan-out weights, which are designated to hold the value W for each connection.

The original Nettalk dealt exclusively with layered, feed-forward networks. In that elementary case, it is possible to simultaneously pipeline the activation levels forward and the error signals backwards from layer to layer, hence the phenomenal throughput achieved by Rosenberg and Belloch. Unfortunately, this savings appears impossible to realize in the general case of recurrent nets. It is easy to understand why. The network must reach equilibrium for a single input vector before the next one is presented; similarly, the error signals must reach steady state for a single target vector before a new one is considered. One might also object that it is singularly wasteful of processors — two processors are required for each connection. The fan-in weights and fan-out weights could be collapsed into a single processor but at the cost of an extra routing step in the feed-forward and back-propagation cycles. In fact, performance requires spreading out the net across as many processors as possible and relying on the virtualization mechanism to provide the necessary resources.

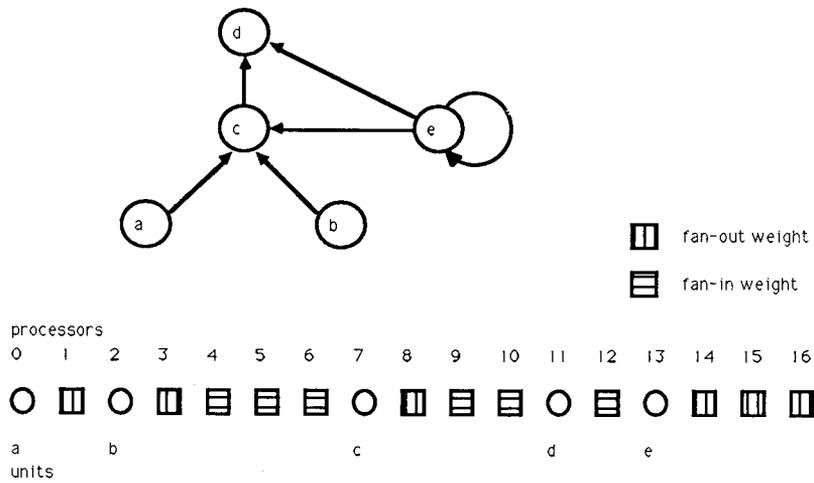


Fig. 1 — Nettek architecture adapted to accommodate the recurrent node "e". The figure published by Rosenberg and Bletloch has been modified for the recurrent net.

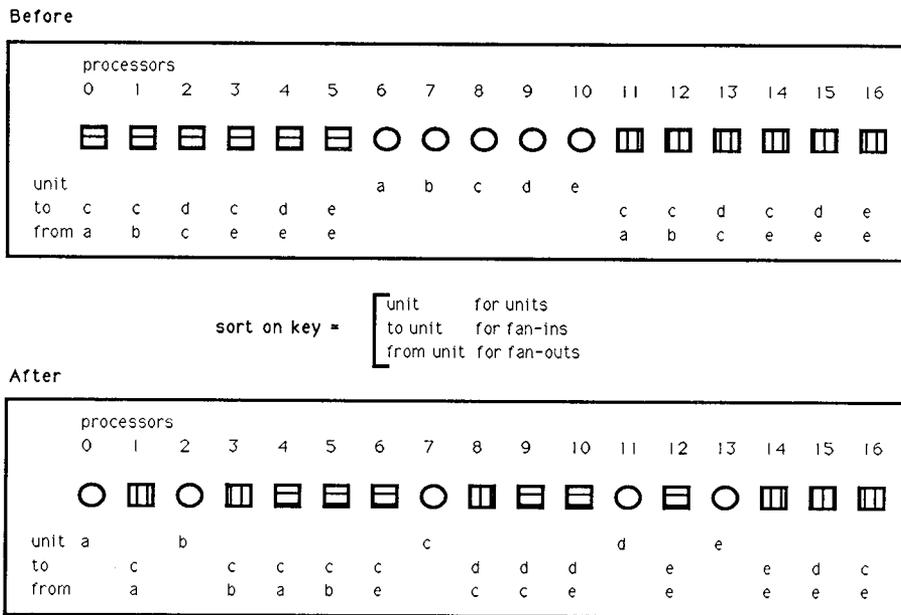


Fig. 2 — Shuffling of processors after sorting on key

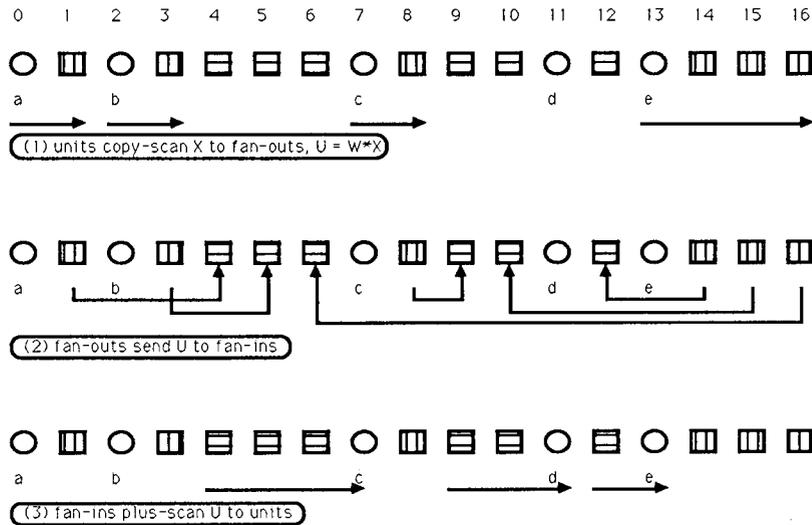


Fig. 3 — The feed-forward cycle in a Nettek architecture. The figure is drawn according to the conventions adopted by Rosenberg and Bletloch.

```
(defun feed-forward (net input!! &key latched-p)
  (*all
    (*when netp!!
      (*when unitp!!
        (*set I!! (!! 0.0) X!! (!! 0.5) dX!! (!! 0.5))
        (*when inputp!!
          (if (memory-netp net)
              (*set X!! (the float-pvar input!!))
              (*set I!! (the float-pvar input!!))))
          (*set Z!! X!!))
        (*set Z!! (scan!! Z!! 'copy!! :segment-pvar forward-fan-out-seg!!))
        (do ()
          ((*when unitp!! (*and (<!! (abs!! dX!!) epsilon-x!!))))
          (dotimes (i (net-x-iterations net))
            (*when fan-outp!!
              (*set U!! (*!! W!! Z!!))
              (*pset :no-collisions U!! U!! to-addr!!))
            (*when unitp!! (*set U!! (!! 0.0)))
            (*set U!! (scan!! U!! '+!! :segment-pvar forward-fan-in-seg!!))
            (*when unitp!!
              (*set LogU!! (logistic!! U!!)
                dX!! (+!! (*!! a!! (-!! X!!)) (*!! b!! LogU!!) I!!)
                X!! (+!! X!! dX!!)
                Z!! X!!)
              (if latched-p
                  (*when inputp!! (*set Z!! (the float-pvar input!!))))
              (*set Z!! (scan!! Z!! 'copy!! :segment-pvar forward-fan-out-seg!!)
                ))))))))
```

Fig. 4 — *Lisp function for the Nettek feed-forward cycle. The numbered pointers highlight the forms realizing the three basic steps of the cycle.

4. TOMBOULIAN ARCHITECTURE

Beside Nettek, another type of graph architecture should be considered. The plan is to compare the respective advantages of these methods in regard to their implementation on a massively parallel processor.

As does Nettek, Tomboulia [8,9] deals with directed graphs. By contrast with Nettek, she intends to embed them in arbitrary SIMD network architectures, restricting herself to nearest-neighbor connections for communications. Her design makes assumptions on the graphs and on the SIMD hardware. In regard to the graphs, Tomboulia requires

- that the graphs have a sparse set of arcs such that the number of arcs per vertex is much smaller than the total number of vertices;
- that the graphs be semidynamic, with the majority of edges remaining fixed through their lifetime; and
- that the operations to be performed at each vertex be homogeneous.

It is evident that the neural networks considered in this report fulfill the latter two requirements. The first requirement, however, is not satisfied by neural nets with high density of connections.

In regard to the SIMD architecture, Tomboulia specifies a very simple machine model.

- The processors execute a single instruction stream;
- processors have a small amount of local memory;
- processors cannot access global memory nor can they perform indirect addressing;
- a processor instruction consists of an op-code plus a local memory address;
- processors communicate only through physical links to a small subset of nearest neighbors;
- each processor has a unique identification number; and
- there exists a data channel to the front-end computer.

This machine model is definitely much simpler than the actual hardware and software provided by the CM. In particular, the router provides the kind of global communications disallowed by Tomboulia. On the other hand, grid addressing on the CM provides the kind of local communications to nearest neighbors that Tomboulia envisages. In grid addressing, the CM is configured as an N -dimensional grid, where processors can read from or write to their nearest neighbors along each dimension. This is the communications pattern referred to as the NEWS network. Currently, under Release 4.3, the CM software supports only two-dimensional grids. Release 5 CM software, however, will implement N -D NEWS, allowing experimentation with grid arrays of higher dimensions.

To complete her model, Tomboulia makes several assumptions on the communications network underlying the processors.

- The neighbor connections are full duplex;
- the labeling scheme for neighbor links is unique and consistent across all processors;
- there exists a small number of neighbor links per processor;
- a path of neighbor links exists between any two processors; and
- the network diameter, that is, the maximum number of hops between processors, is not large.

The Tombouliau model machine lacks the global routing mechanism necessary for graph processing; it must be implemented in software. To this effect, when embedding arbitrary directed graphs on a SIMD machine, each vertex of the graph is assigned a processor. Then sending information along an arc amounts to routing messages from neighbor to neighbor. Considering that a message may require many hops to reach its destination, Tombouliau defines the parallel traversal of all arcs to be an uninterruptible operation requiring T time steps; T is what she calls the time quantum. An arc in the graph materializes as a sequence of contiguous links between processors, beginning at a given step in the time quantum. In addition, Tombouliau forces paths to be invariant in time and space, that is, once the sequence of links and the start time for a path are chosen, they are never changed. No message buffering is allowed; once under way a message proceeds from link to link without waiting at any processor. Finally, no message collisions are allowed; a processor can receive only a single message at a given time step.

From these specifications, two rules for path construction follow. First, only one link can enter and leave a processor at any given time step. Second, any path between processors representing an arc must consist of links at contiguous time steps. Given these rules, the paths can be implemented by a table of T routing slots, as described in Fig. 5. The ‘startp’ flag indicates the start of an arc whereas the ‘endp’ flag signals the end of an arc. The ‘arc-label’ field contains information attached to the start of each arc. The ‘forward’ and ‘backward’ slots form the heart of the routing algorithm. Note that Tombouliau did not introduce a ‘backward’ slot since she was interested only in forward routing. The ‘backward’ slot must be added to accommodate backward routing since this feature is necessary in solving the back-propagation equations in a neural net.

```

struct Slot {
    startp          ; boolean for start of arc
    forward         ; forward read direction,
    backward        ; backward read direction,
    endp            ; boolean for end of arc,
    arc-label       ; information attached to arc.
}
Slots[T]

```

Fig. 5 — Routing table local to a processor in the net

At each time step, the processors examine the ‘forward’ or ‘backward’ slot to see over which neighbor link they must read for forward or backward routing, respectively. Hence, traversing all

arcs of the graph in parallel takes time T as the processors loop through their local routing tables. Figure 6 presents an example of a small graph embedded in a grid of four processors. Figure 7 summarizes the basics of the forward routing algorithm; Fig. 8 contains the corresponding *Lisp code.

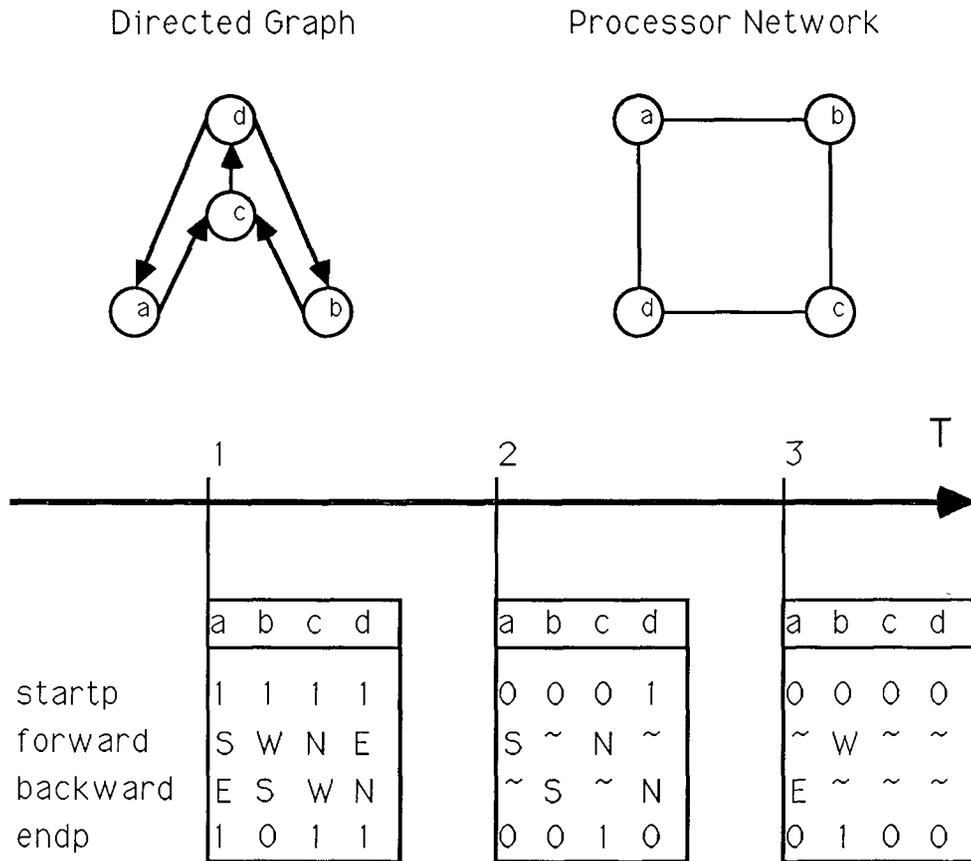


Fig. 6 — The directed graph embedded in a grid of four processors

Examining the basics of routing in Tomboulian's scheme leads to a discussion of its use in solving the feed-forward equations. Units propagate their activation levels by forward routing along the connections of the net. On each outgoing connection, units send their activation level multiplied by the weight stored in that connection's arc-label. Processors also accumulate the activation messages received from incoming connections. Tomboulian's forward routing cycle replaces the general send and scan operations used in the Nettek implementation. Note that the performance of the feed-forward equations depends critically on the time quantum T that governs the speed of routing operations.

In the Tomboulian scheme, graphs are built serially, one arc at a time. The first step requires building all possible trial paths from the source processor to the destination. The current *Lisp implementation only propagates the shortest trial path forward when several paths meet. Next, if the destination can be reached, pick the shortest trial path and trace it backward, updating the slots array along the way (see Fig. 9).

This contrasts vividly with the Nettek construction, which performs a parallel construction.

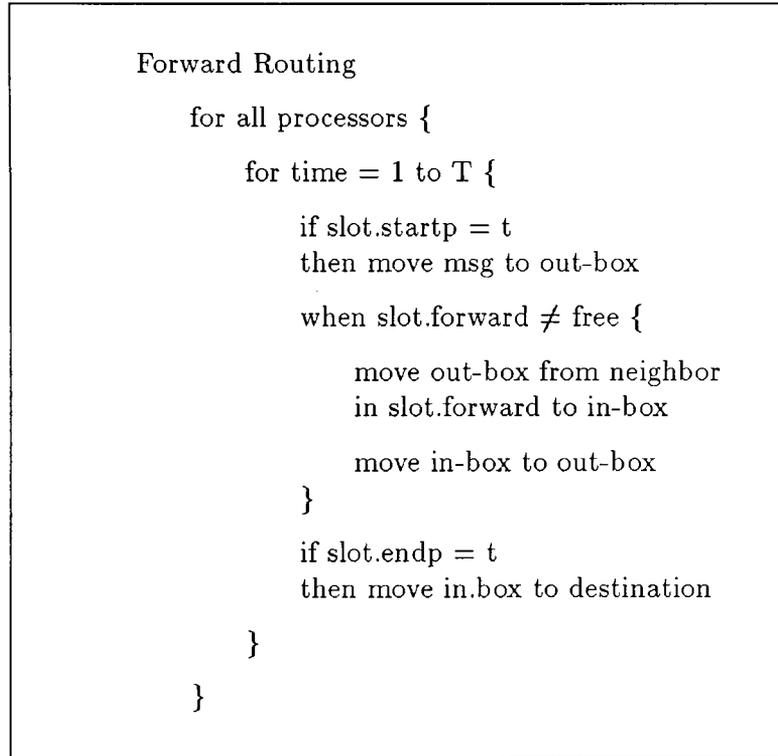


Fig. 7 — Tombouliau's forward routing algorithm traverses all arcs of the graph in time T .

Nevertheless, the Tombouliau scheme does have interesting implications for dynamic configuration and fault tolerance. Graph edges may be deleted and added to an existing graph easily, opening the possibility of dynamically reconfiguring a neural net. Next, this dynamic configuration offers the possibility of fault tolerance. Upon discovering a faulty processor, all arcs going through this processor could be recovered and reconstructed with another processor assigned to the unit. In short, although much more expensive, the Tombouliau construction scheme exhibits greater flexibility. Needless to say, dynamic reconfiguration of nets is especially important for hardware implementations of neural nets. Indeed think of a general neural net chip that could be configured for arbitrary net topologies and that could recover from component failures.

Experimenting with various SIMD architectures by simulator and on a CM, Tombouliau established empirically that the time quantum T is approximately equal to the network diameter times the average degree of each vertex. In the case of sparse graphs, the time quantum remains small, parallel traversal of the arcs proceeds quickly, and the Tombouliau scheme proves quite effective. Note that Tombouliau experimented exclusively with graphs of low connectivity such as n -ary trees and random graphs with few arcs per vertex. Neural nets, however, exhibit high connectivity. This report assesses the dire results that follow for the size of the time quantum. As the time quantum grows, so does the time required for routing, causing unacceptably degraded performance in the solution of the feed-forward and back-propagation equations.

```

(defmacro route-forward (label-name
                        in-box!!
                        in-box-type
                        arc-start-function
                        arc-end-function)

  (let ((slot (gensym))
        (out-box!! (gensym)))
    '(let (,label-name)
      (*all
        (*let (,in-box!!
              ,out-box!!)
          (declare (type ,in-box-type
                    ,in-box!! ,out-box!!))

          (map nil
               #'(lambda (,slot)
                   (setf ,label-name
                        (slot-arc-label!! ,slot))
                   (*if (slot-startp!! ,slot)
                       (*set ,out-box!!
                            ,arc-start-function))
                   (*when (/=!! (slot-forward!! ,slot)
                               (neighbor-limit!!))
                       (pref-neighbor!! ,in-box!!
                                         ,out-box!!
                                         (slot-forward!! ,slot))
                       (*set ,out-box!! ,in-box!!))
                   (*if (slot-endp!! ,slot)
                       ,arc-end-function))
               slots[]!!)
        )))
  ))

```

Fig. 8 — *Lisp code for forward routing

Shortest Path Construction:

Flood all trial paths from source processor
 If destination processor reached, then trace
 shortest trial path backward & update slots

Shortest Trial Path Flooding:

```

for all processors {
  reset trial-slots
  set source processor active
  for time = 1 to T {
    when active and free to send {
      for n = 1 to neighbor-limit {
        if has neighbor n and shorter trial path to n
          then update neighbor's trial-slot
      }
    }
    if trial-slot.direction ≠ free
      then mark as active
    if destination processor active {
      mark as inactive
      if free to receive
        then mark as reached
      else clear trial-slot
    }
    set source processor active
  }
}

```

Fig. 9 — Path construction for Tombouliau's graphs. The scheme at bottom details the trial-path flooding algorithm.

5. PERFORMANCE

Before discussing the RBP algorithm's performance on the CM, it may prove reassuring to discuss how the *Lisp routines were verified, given the obvious difficulties of checking results in massive networks. Both the Nettek and Tomboulion implementations were exercised on several small, well-studied problems with well-known behavior. Appendix A contains two such toy problems. In the first example, a continuous mapping net implemented with the Nettek architecture successfully learns the inclusive-or function. The second example shows an associative memory net implemented with the Tomboulion scheme properly storing a target set corresponding to the exclusive-or function.

To assess the performance of the RBP algorithm on the CM, consider the two network models pictured in Fig. 10. The continuous mapping net consists of N output units, $2N$ hidden units, and $4N$ input units, with all layers fully connected to their superiors. Also, a bias node connects to all the units in the hidden and output layers. For the feed-forward and back-propagation equations, both the input and target vectors are taken to be unit vectors. The associative memory net model consists of a hidden layer of N units and a visible layer of $8N$ units. The two layers are linked by bundles of connections where each possible connection is made with a probability of 25%. For the feed-forward and back-propagation equations, a unit vector serves as the target.

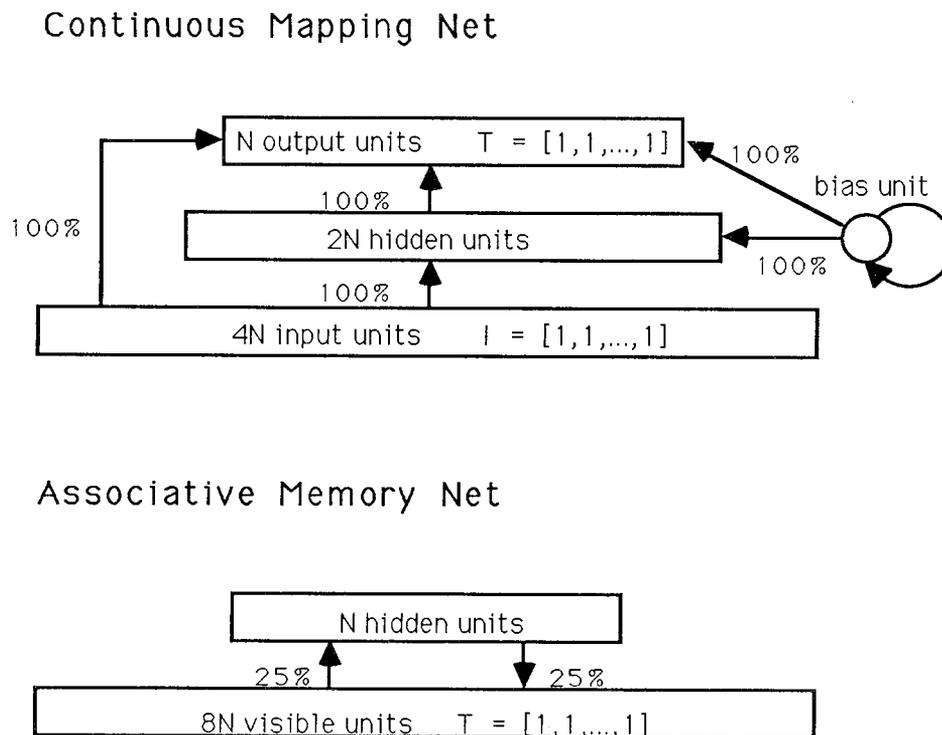


Fig. 10 — Neural net models used in measuring the performance of the RBP algorithm on the CM

The RBP timings were performed for varying network model sizes N , for both the Nettek and Tomboulion implementations. In the Nettek implementation, the *Lisp routines were compiled with both software and hardware floating-point. For each value of N , we measured the time to construct the net and to solve both the feed-forward and back-propagation equations. Each such

timing specifies both the total execution time as measured on the front end and the actual execution time on the CM.

Suffice to say, the timing experiments clearly demonstrate the superiority of the Nettek architecture over the Tomboulia scheme. The Nettek implementation allowed the simulation of much larger networks. These networks could be constructed much more quickly, and they were far more efficient in solving the feed-forward and back-propagation equations. The Tomboulia implementation permitted only the simulation of small nets, since the processor memory size sharply limits the size of the routing tables. Indeed, for the continuous mapping net of size N equals 8, the routing tables required exceed the capacity of local processor memory. Appendix B provides the complete set of timing data collected.

Instead of reciting numbers, examining five charts extracted from the timing data provides clearer insight into some characteristics of the CM and into the performance of the RBP algorithm. The first three charts illustrate important lessons drawn from the Nettek implementation of RBP. The last two charts present some trends exhibited by the Tomboulia architecture.

The secret of success in the Nettek implementation lies in its choice of connections as the basic unit of representation. Figure 11 plots the numbers of units, connections, and processors allocated for increasingly larger models of the associative memory net. As may be apparent, the number of connections grows exponentially with the number of units in a neural network. The Nettek scheme tackles this difficulty head-on – the allocation curve for processor resources follows the growth in connections, not in units.

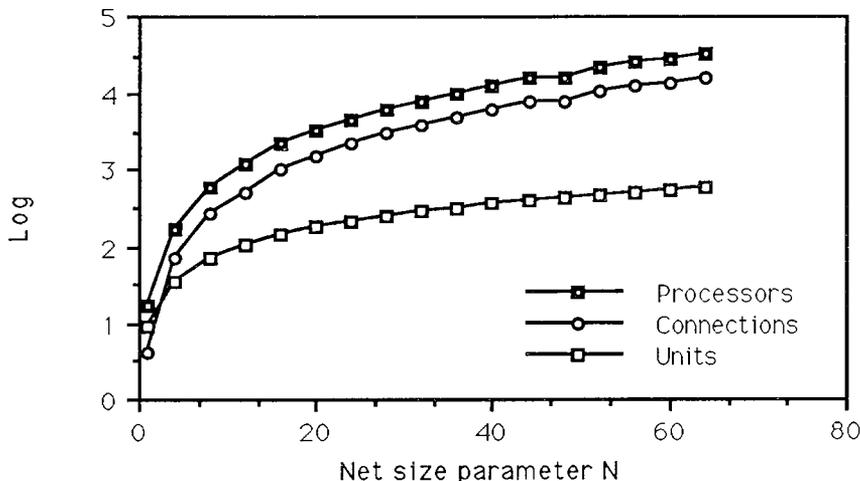


Fig. 11 — Allocation of processor resources for an associative memory net under the Nettek architecture

Figure 12 reveals some interesting implications of virtualization on the CM. The front-end time and CM time required to construct the continuous mapping net mirror the staircase shape of the VP ratio. With each increase in the VP ratio, a physical processor must emulate larger numbers of virtual processors. Note, however, that the construction time for larger mapping nets grows less than linearly with the VP ratio, since sorting and communications operations become more efficient for larger numbers of virtual processors. Using high VP ratios on the CM carries a smaller

penalty than one might intuitively expect.

The next conclusion concerns the relative importance of communications versus computation in the RBP algorithm. Figure 13 depicts the CM time for solution of the feed-forward equations in the continuous mapping net using software and hardware floating-point. For software floating-point, the CM processors perform all operations bit-serially. For hardware floating-point, the feed-forward routines take advantage of the CM's WEITEK floating-point accelerator chips. As readily observed, the floating-point hardware provides almost no increase in performance, providing dramatic proof that the net spends most of its time in communications.

The RBP timings taken from the Tombouliau architecture provide interesting contrast to those taken from the Nettalk scheme. Figure 14 shows the numbers of units and connections along with the time quantum and percentage of routing slots used for the associative memory net. As the net model size increases, the time quantum and slot use grow linearly with the number of units in the net. These results agree perfectly with Tombouliau's empirical determination of T . As the number of units increases, the average number of connections per unit multiplies, resulting in rapid growth of the time quantum. Accordingly, processors representing units consume burgeoning amounts of local memory space and processing time to perform Tombouliau's routing algorithm.

The growth in the time quantum holds dire consequences for network performance. Figure 15 shows that the front end and CM times for a feed-forward cycle in the associative memory net increase linearly with the time quantum, resulting in rapid deterioration of RBP efficiency.

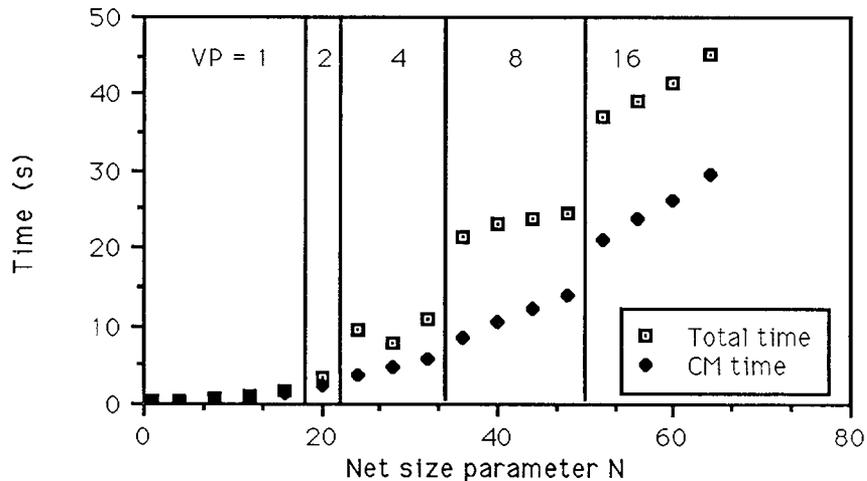


Fig. 12 — Effects of CM virtualization on net construction efficiency for a continuous mapping net under the Nettalk architecture

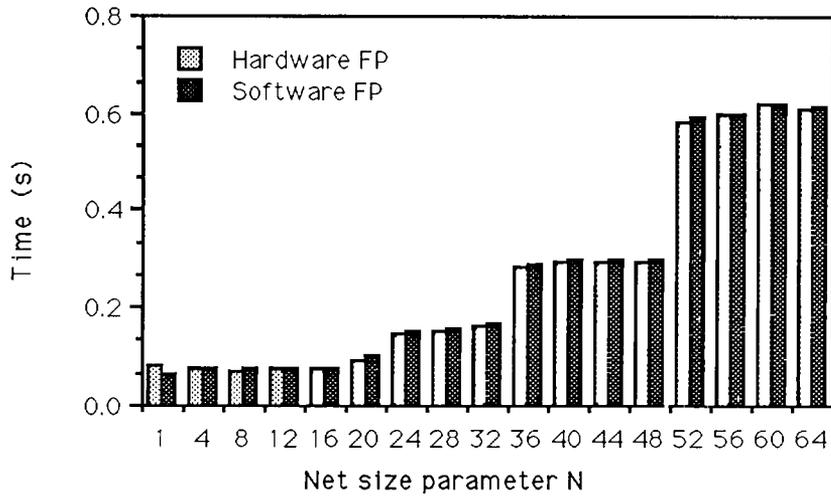


Fig. 13 — Communications versus computation in the Nettalk architecture. Feed-forward cycle speed in a continuous mapping net is measured for both hardware and software floating-point.

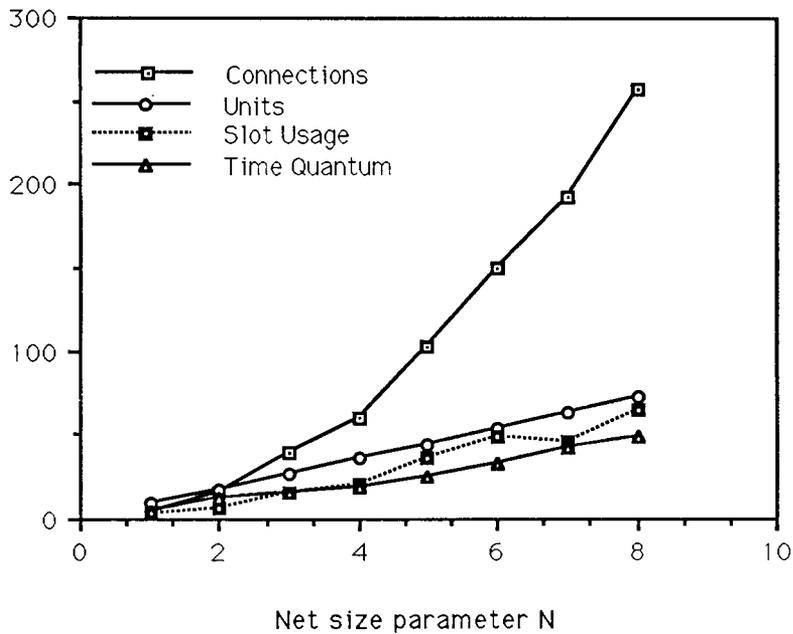


Fig. 14 — Growth of the time quantum and slot usage with net size for an associative memory net under the Tombouliau architecture

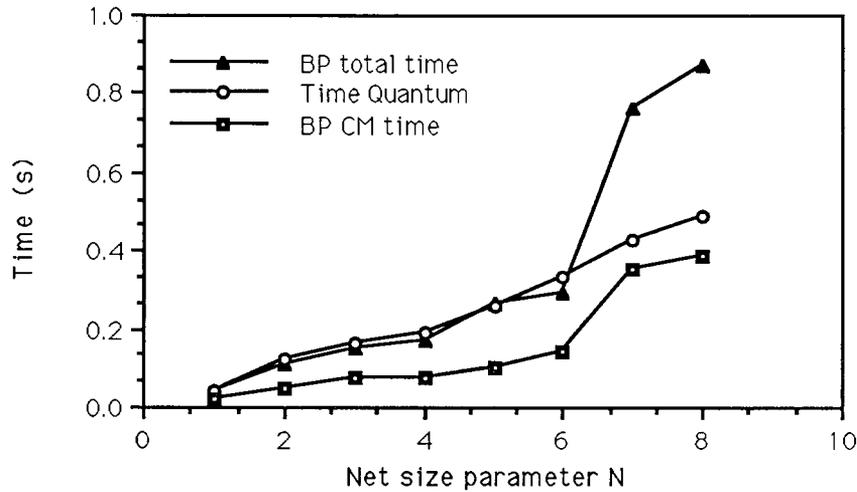


Fig. 15 — Decreasing speed of back-propagation with growth of the time quantum for an associative memory net under the Tomboulian architecture

6. CONCLUSIONS

As a design tool assisting in the hardware realization of neural nets, the CM proves eminently suited for simulating candidate neural net algorithms and architectures. Because of its flexibility, the CM lends itself to exploring all kinds of network architectures and communications patterns. The CM may be used as a universal parallel processor on which particular architectures may be tested. In regard to neural networks, the CM plays the same role traditionally performed by serial computers in the hardware design process.

The CM simulation clearly establishes that networks spend most of their activity in communicating rather than in calculating. In a serial implementation, communication is simulated in the form of matrix operations. In the CM, on the other hand, the communications are actual, not simulated: units send their activation levels to connected units and, in turn, receive correction signals from them. In either case, these operations are time consuming. Therefore, most of the resources of the parallel machine should be spent on representing connections rather than units. This preponderance of communication over computation is especially true for the RPB scheme since it iterates over the net. In regard to hardware implementation, much attention has been paid to the problem of realizing arithmetic operations in analog circuitry. In other words, researchers have spent most of their time implementing the units of the net. Insofar as it reflects reality, the RBP simulation indicates that a far bigger problem lies in creating the means for efficient hardware communications — representing the connections of the net.

Nettalk works well for highly connected nets; Tomboulian's scheme fails for dense graphs. The Nettalk implementation, however, makes full use of the communications power of the CM. Such sophisticated routing capability seems impossible to realize on a single chip in present technology. A neural net chip must be based on simpler routing constructs. For sparse nets with small time quanta, the Tomboulian architecture offers an alternative, more so because it presents interesting implications for dynamic reconfiguration. Nevertheless, a neural net chip still cannot be based on

Tombouliau's routing scheme.

Ideally, one might conceive of combining the two paradigms to create a neural net chip set. On the one hand, there would be an analog chip containing collections of computing units laid on top of a reconfigurable network of wires. The feed-forward and back-propagation equations would be solved by analog computations according to Pineda's RBP algorithm. This chip resembles the Nettek architecture in that most of the hardware resources would be devoted to a flexible network of connections among the units. On the other hand, there would be a digital controller chip to configure the communications network on the analog chip by running a path-finding algorithm, such as Tombouliau's. Such a pairing would allow the general analog chip to actuate arbitrary neural net topologies, and provide a measure of fault tolerance by dynamic reconfiguration.

7. ACKNOWLEDGMENTS

Dr. Fernando Pineda of the Applied Physics Laboratory (Johns Hopkins University) suggested the topic of this work and reviewed the results. Without the support of Dr. Shannon Coffey at the Navy Center for Space Technology (Naval Research Laboratory), this research would not have been possible. Robert Whaley of Thinking Machines Corporation has been of considerable assistance in the coding and the use of the Connection Machine. Comments by Dr. Liam Healy, NRC Research Associate at the Naval Research Laboratory, and by Dr. André Deprit of the Center for Applied Mathematics (National Bureau of Standards) have been very helpful.

8. REFERENCES

1. W. D. Hillis, *The Connection Machine* (The MIT Press, Cambridge, MA, 1985).
2. W. D. Hillis and G. L. Steele, Jr., "Data Parallel Algorithms," *Commun. ACM* **29**, 1170-1183 (1986).
3. F. J. Pineda, "Generalization of Back-Propagation to Recurrent Neural Networks," *Phys. Rev. Lett.* **59**, 2229-2232 (1987).
4. F. J. Pineda. "Generalization of Back-Propagation to Recurrent and Higher Order Neural Networks," to appear in the *Proceedings of IEEE Conference on Neural Information Processing Systems*, Denver, CO (1987).
5. C. R. Rosenberg and G. E. Blelloch, "An Implementation of Network Learning on the Connection Machine," Technical report, Thinking Machines Corporation, Cambridge, MA, 1986.
6. *Connection Machine Parallel Instruction Set* (Thinking Machines Corporation, Cambridge, MA, 1986).
7. *Introduction to Data Level Parallelism* (Thinking Machines Corporation, Cambridge, MA, 1986).
8. S. J. Tombouliau, *A System for Routing Arbitrary Communication Graphs on SIMD Architectures*, Ph. D. Dissertation, Duke University, 1986.

9. S. J. Tombouliau. "A Brief Overview of a System for Routing Directed Graphs on SIMD Architectures," to appear in the *Proceedings of 2nd Symposium on the Frontiers of Massively Parallel Computation*, Fairfax, VA (1988).
10. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, in *Parallel Distributed Processing*, D. E. Rumelhart and J.L. McClelland, eds. (The MIT Press, Cambridge, MA, 1986).
11. **Lisp Reference Manual* (Thinking Machines Corporation, Cambridge, MA, 1987).

Appendix A SAMPLE NETS

As explained in Section 5, several elementary problems serve to verify the implementation of RBP on the CM. These sample runs also prove useful in demonstrating how to use the *Lisp routines.

A1 Nettek Architecture

The first sample run teaches the net depicted in Fig. A1 the inclusive-or function.

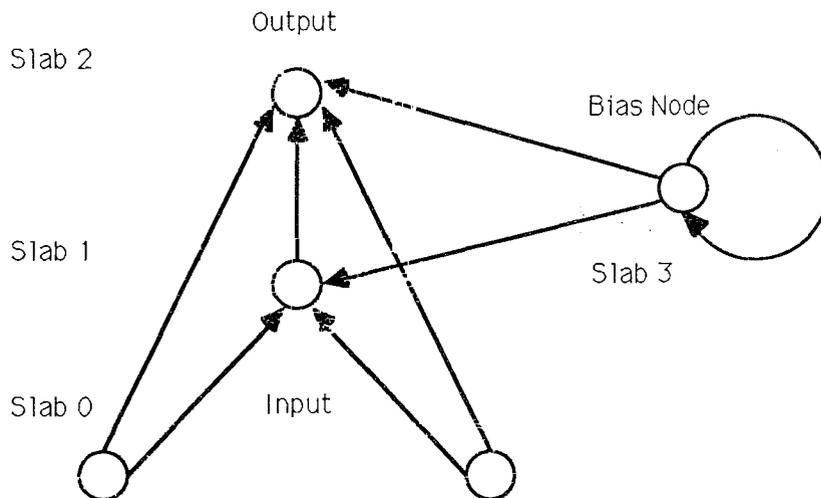


Fig. A1 — Inclusive-or continuous mapping net constructed by the DEF-MAPPING-NET macro

The following forms typed to the Lisp listener construct the IOR net, define and load the input/target pairs, and start the training.

```
(def-mapping-net or-mapping-net
  :slabs '(2 1 1 1)
  :input-slab-no 0
  :output-slab-no 2
  :bundles '((1 0 100)
            (2 0 100)
            (2 1 100)
            (1 3 100)
            (2 3 100)
            (3 3 100))
```

```

)
)

(defvar *ior-mapping-pairs*)
(setf *ior-mapping-pairs*
      (list-to-array-pairs '(((0.0 0.0) (0.0))
                            ((0.0 1.0) (1.0))
                            ((1.0 0.0) (1.0))
                            ((1.0 1.0) (1.0)))))

(defvar *ior-mapping-set*)
(setf *ior-mapping-set*
      (cm-load-mapping-set 'ior-mapping-set *ior-mapping-pairs*))

(train-net or-mapping-net
          *ior-mapping-set*
          :print-training-set #'print-training-set
          :print-interval 20
          :print-net-io #'print-io-vecs)

```

The report produced by the TRAIN-NET function is excerpted below and summarized in Table A1.

Net Training

CONTINUOUS-MAPPING net: OR-MAPPING-NET

4 slabs, 6 bundles

5 units, 8 connections ->21 processors

a = 1.0, b = 1.0

Feed-forward convergence = 0.001, min 4 iterations

Back-propagate convergence = 0.001, min 4 iterations

eta = 0.25, alpha = 0.9

Weight update convergence = 0.1, max 10000 iterations

MAPPING-SET: IOR-MAPPING-SET

i: 0.0 0.0 t: 0.0

i: 0.0 1.0 t: 1.0

i: 1.0 0.0 t: 1.0

i: 1.0 1.0 t: 1.0

Iteration 0

i: 0.0 0.0 o: 0.47596744

i: 0.0 1.0 o: 0.49091664

i: 1.0 0.0 o: 0.4085974

i: 1.0 1.0 o: 0.42314819

Error = 2.1533053

Iteration 20

i: 0.0 0.0 o: 0.60783666

i: 0.0 1.0 o: 0.7946026

i: 1.0 0.0 o: 0.77972716

i: 1.0 1.0 o: 0.89824456

Error = 1.1352623

⋮

Iteration 300

i: 0.0 0.0 o: 0.0999451
 i: 0.0 1.0 o: 0.9384478
 i: 1.0 0.0 o: 0.93790066
 i: 1.0 1.0 o: 0.9991285
 Error = 0.22446814

Training set learned after 301 iterations.

i: 0.0 0.0 o: 0.09965193
 i: 0.0 1.0 o: 0.9386314
 i: 1.0 0.0 o: 0.93808526
 i: 1.0 1.0 o: 0.99913496
 Error = 0.22446814

Table A1 — Sample Run for the Nettek Implementation of an IOR Mapping Net

Iteration	Input → Output				Error
	[0,0] → [0]	[0,1] → [1]	[1,0] → [1]	[1,1] → [1]	
0	0.475967440	0.49091664	0.40859740	0.42314819	2.15330530
20	0.607836660	0.79460260	0.77972716	0.89824456	1.13526230
40	0.523044400	0.78751314	0.78820790	0.92609406	1.02122930
60	0.445310150	0.79706100	0.80016330	0.95087760	0.89720820
80	0.374926000	0.81468093	0.81675434	0.96944827	0.77404240
100	0.312892900	0.83520794	0.83605444	0.98171127	0.65991926
120	0.261516780	0.85550890	0.85557880	0.98894240	0.56148670
140	0.221299750	0.87344560	0.87311420	0.99305516	0.48168480
160	0.190724300	0.88835150	0.88783570	0.99538165	0.41915548
180	0.167220180	0.90026370	0.89967410	0.99674326	0.37053907
200	0.149121730	0.91006210	0.90944266	0.99759334	0.33202365
220	0.134924490	0.91799843	0.91737980	0.99814450	0.30140173
240	0.123531714	0.92450523	0.92390060	0.99851924	0.27660662
260	0.114199980	0.92991730	0.92933120	0.99878496	0.25616658
280	0.106417686	0.93448216	0.93391600	0.99898010	0.23903945
300	0.099945100	0.93844780	0.93790066	0.99912850	0.22446814
301	0.099651930	0.93863140	0.93808526	0.99913496	0.22446814

A2 Tomboulian Architecture

This section parallels the results of the previous one for an exclusive-or associative memory net underlied by the Tomboulian architecture. The Lisp forms below construct and train the exclusive-or net specified in Fig. A2. Table A2 summarizes the sample run presented.

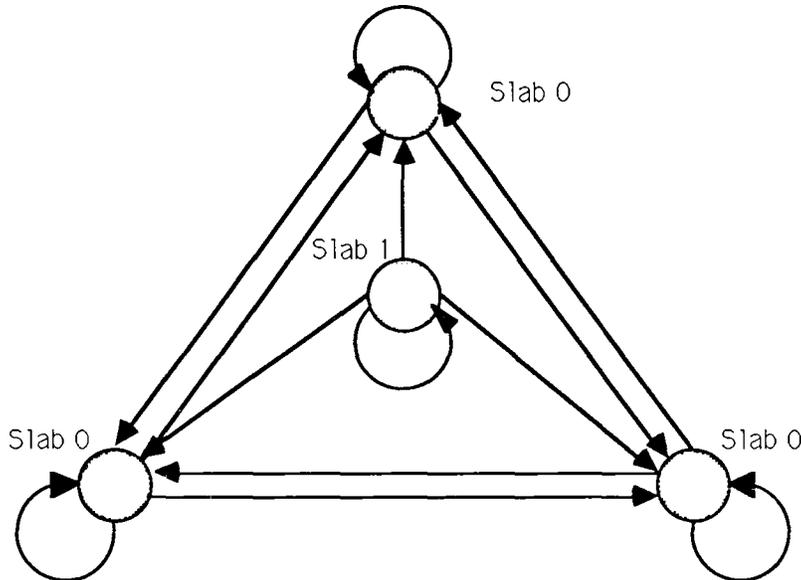


Fig. A2 — Exclusive-or associative memory net constructed by the DEF-MEMORY-NET macro

The following forms typed to the Lisp listener construct the XOR net, define and load the target vectors, and start the training.

```
(def-memory-net or-memory-net
  :slabs '(3 1)
  :input-slab-no 0
  :bundles '((0 0 100)
            (0 1 100)
            (1 1 100)
            )
  :epsilon-w 0.05
)

(defvar *xor-memory-list*)
(setf *xor-memory-list*
      (list-to-array '((0.0 0.0 0.0)
                    (0.0 1.0 1.0)
                    (1.0 0.0 1.0)
                    (1.0 1.0 0.0))))

(defvar *xor-memory-set*)
(setf *xor-memory-set*
      (cm-load-memory-set 'xor-memory-set *xor-memory-list*))

(train-net or-memory-net
  *xor-memory-set*
  :print-training-set #'print-training-set)
```

```
:print-interval 20
:print-net-io #'print-io-vecs)
```

The TRAIN-NET function produces the reported excerpted below; Table A2 documents the complete training results.

Net Training

ASSOCIATIVE-MEMORY net: OR-MEMORY-NET

2 slabs, 3 bundles
4 units, 13 connections ->4 processors

a = 1.0, b = 1.0

Feed-forward convergence = 0.001, min 4 iterations
Back-propagate convergence = 0.001, min 4 iterations

eta = 0.25, alpha = 0.9
Weight update convergence = 0.05, max 10000 iterations

MEMORY-SET: XOR-MEMORY-SET

```
i: 0.0 0.0 0.0
i: 0.0 1.0 1.0
i: 1.0 0.0 1.0
i: 1.0 1.0 0.0
```

Iteration 0

```
i: 0.0 0.0 0.0   o: 0.53553134 0.55844945 0.5587541
i: 0.0 1.0 1.0   o: 0.66410667 0.5466608 0.5988787
i: 1.0 0.0 1.0   o: 0.66019136 0.57686245 0.5815371
i: 1.0 1.0 0.0   o: 0.6112226 0.58896685 0.47078228
Error = 3.3785267
```

Iteration 20

```
i: 0.0 0.0 0.0   o: 0.39624417 0.42216888 0.42507586
i: 0.0 1.0 1.0   o: 0.20121947 0.82004374 0.83080995
i: 1.0 0.0 1.0   o: 0.82579315 0.19658758 0.82820785
i: 1.0 1.0 0.0   o: 0.81852823 0.8260062 0.18509121
Error = 1.6629202
```

⋮

Iteration 420

```
i: 0.0 0.0 0.0   o: 0.050692994 0.051062807 0.05110865
i: 0.0 1.0 1.0   o: 0.03370365 0.9651646 0.96524113
i: 1.0 0.0 1.0   o: 0.96526337 0.03356958 0.9652226
i: 1.0 1.0 0.0   o: 0.9652148 0.9652037 0.03343686
Error = 0.2669139
```

Training set learned after 438 iterations.

```
i: 0.0 0.0 0.0   o: 0.07672652 0.07758254 0.07768838
i: 0.0 1.0 1.0   o: 0.044398107 0.95486647 0.9549862
```

i: 1.0 0.0 1.0 o: 0.95502245 0.04418476 0.9549585
i: 1.0 1.0 0.0 o: 0.95495033 0.95493317 0.043976907
Error = 0.2611845

Table A2 — Sample Run for the Tomboulian Implementation of an XOR Memory Net

Iteration	Fixed Points				Error
	[0,0,0]	[0,1,1]	[1,0,1]	[1,1,0]	
0	0.535531340	0.664106670	0.660191360	0.611222600	3.378526700
	0.558449450	0.546660800	0.576862450	0.588966850	
	0.558754100	0.598878700	0.598878700	0.470782280	
20	0.396244170	0.201219470	0.825793150	0.818528230	1.662920200
	0.422168880	0.820043740	0.196587580	0.826006200	
	0.425075860	0.830809950	0.828207850	0.185091210	
40	0.275940400	0.120262250	0.868772030	0.866055200	1.160470500
	0.292070450	0.865704660	0.118025504	0.867952100	
	0.293652100	0.869275100	0.868246800	0.113853940	
80	0.149614990	0.077436500	0.908097900	0.907281640	0.720232400
	0.155146170	0.906636600	0.076124990	0.907307000	
	0.155658470	0.907629300	0.907315900	0.074514830	
120	0.110390110	0.063607864	0.928866000	0.928474600	0.551653400
	0.113153750	0.928096300	0.062789350	0.928415240	
	0.113438090	0.928614100	0.928464100	0.061894290	
160	0.090866710	0.055146575	0.940168600	0.939931000	0.462379520
	0.092580350	0.939691840	0.054593630	0.939884660	
	0.092768740	0.940025030	0.939934250	0.054013073	
200	0.078779950	0.049278720	0.947459400	0.947296400	0.405272000
	0.079972155	0.947130500	0.048876950	0.947262400	
	0.080108650	0.947367670	0.947305300	0.048463512	
240	0.070399430	0.044914193	0.952646800	0.952526300	0.364817650
	0.071289600	0.952403200	0.044606360	0.952500460	
	0.071394210	0.952582900	0.952536940	0.044293456	
280	0.064166170	0.041510116	0.956574500	0.956481100	0.334269300
	0.064863300	0.956385100	0.041264930	0.956460540	
	0.064946750	0.956527200	0.956491530	0.041017827	
320	0.059303710	0.038762380	0.959679250	0.959604000	0.310167850
	0.059868710	0.959526800	0.038561273	0.959587300	
	0.059937287	0.959642600	0.959613860	0.038359870	
360	0.055377590	0.036485903	0.962211900	0.962149700	0.290535570
	0.055847496	0.962085840	0.036317125	0.962135800	
	0.055905145	0.962182500	0.962158800	0.036149006	
400	0.052123690	0.034560820	0.964328050	0.964275540	0.274150460
	0.052522577	0.964221400	0.034416642	0.964263600	
	0.052571874	0.964303900	0.964283900	0.034273600	
438	0.076726520	0.044398107	0.955022450	0.954950330	0.261184500
	0.077582540	0.954866470	0.044184760	0.954933170	
	0.077688380	0.954986200	0.954958500	0.043976907	

Appendix B TIMINGS

To evaluate the performance of the RBP algorithm on the CM, the following tables present the timing data collected for the neural net models of Fig. 10.

Tables B1 to B4 present statistics for the continuous mapping and associative memory net models under the Nettek architecture. The following data are given for each value of the net size parameter N :

- the number of units in the timing net,
- the number of connections,
- the number of virtual processors used,
- the ratio of virtual to physical processors.

The next columns give the time taken to perform the three basic operations:

- making the net,
- feed-forward cycle,
- back-propagation cycle.

Each timing is given both as the total elapsed time on the front end and on the CM.

Table B2 provides the same timings as Table B1. In contrast to Table B1, however, the timings in Table B2 were collected by using software rather than hardware floating-point in an attempt to assess the dominance of communication over computation in the nets.

The same relationship exists between Tables B3 and B4 for an associative memory net in the Nettek implementation. Tables B3 and B4 provide timing data for associative memory nets rather than continuous mapping nets in order to verify the modifications required to the RBP algorithm and assess its performance on lower density nets.

Tables B5 and B6 give timing data for continuous mapping and associative memory nets by using the Tomboulia architecture. Two columns have been added: one for the time quantum and one for the slot use in the routing tables. The results in these tables provide a vivid contrast to those in Tables B1 to B4, thereby revealing the superiority of the Nettek scheme.

Table B1 — Timings for the Nettek Implementation of a Continuous Mapping Net
Compiled with Hardware Floating Point

N	Units	Connections	Processors	VP Ratio	Make (s)	FF (s)	BP (s)
					CM Total	CM Total	CM Total
1	8	18	44	1	0.116	0.083	0.017
					0.220	0.190	0.020
4	29	237	503	1	0.210	0.078	0.027
					0.320	0.160	0.030
8	57	921	1899	1	0.468	0.069	0.027
					0.580	0.150	0.030
12	85	2053	4191	1	0.892	0.076	0.027
					1.010	0.160	0.030
16	113	3633	7379	1	1.467	0.078	0.028
					1.580	0.160	0.030
20	141	5661	11463	2	2.304	0.092	0.038
					3.390	0.160	0.040
24	169	8137	16443	4	3.585	0.146	0.078
					9.600	0.200	0.080
28	197	11061	22319	4	4.642	0.155	0.078
					7.680	0.210	0.080
32	225	14433	29091	4	5.879	0.165	0.154
					10.900	0.220	0.160
36	253	18253	36759	8	8.663	0.281	0.163
					21.410	3.300	4.090
40	281	22521	45323	8	10.469	0.295	0.176
					23.170	4.300	4.090
44	309	27237	54783	8	12.076	0.293	0.173
					23.770	4.300	4.100
48	337	32401	65139	8	13.796	0.292	0.183
					24.610	4.300	4.090
52	365	38013	76391	16	21.192	0.583	0.359
					36.930	8.400	4.090
56	393	44073	88539	16	23.726	0.601	0.377
					38.990	8.430	4.090
60	421	50581	101583	16	26.319	0.618	0.378
					41.510	8.390	4.090
64	449	57537	115523	16	29.587	0.609	0.403
					45.210	8.410	4.090

Table B2 — Timings for the Nettek Implementation of a Continuous Mapping Net
Compiled with Software Floating Point

N	Units	Connections	Processors	VP Ratio	Make (s)	FF (s)	BP (s)
					CM Total	CM Total	CM Total
1	8	18	44	1	0.117 0.220	0.065 0.150	0.017 0.020
4	29	237	503	1	0.218 0.320	0.077 0.160	0.027 0.030
8	57	921	1899	1	0.478 8.630	0.078 0.160	0.027 0.030
12	85	2053	4191	1	0.885 0.990	0.075 0.170	0.027 0.030
16	113	3633	7379	1	1.454 1.560	0.077 0.160	0.028 0.030
20	141	5661	11463	2	2.334 3.430	0.101 0.170	0.038 0.040
24	169	8137	16443	4	3.591 10.600	0.151 0.210	0.079 0.080
28	197	11061	22319	4	4.684 7.740	0.157 0.210	0.075 0.080
32	225	14433	29091	4	5.910 10.950	0.167 0.220	0.075 0.080
36	253	18253	36759	8	8.694 21.600	0.288 3.310	0.161 4.090
40	281	22521	45323	8	10.282 23.010	0.297 4.310	0.180 4.090
44	309	27237	54783	8	11.998 23.710	0.297 4.300	0.178 4.090
48	337	32401	65139	8	13.933 23.860	0.300 4.300	0.182 4.090
52	365	38013	76391	16	21.165 36.950	0.593 8.400	0.369 4.090
56	393	44073	88539	16	23.880 39.200	0.600 8.390	0.389 4.100
60	421	50581	101583	16	26.345 41.510	0.621 8.390	0.389 4.090
64	449	57537	115523	16	29.512 45.100	0.616 8.420	0.409 4.090

Table B3 — Timings for the Nettek Implementation of an Associative Memory Net
Compiled with Hardware Floating Point

N	Units	Connections	Processors	VP Ratio	Make (s)	FF (s)	BP (s)
					CM Total	CM Total	CM Total
1	9	4	17	1	0.075	0.063	0.008
					0.120	0.140	0.010
4	36	68	172	1	0.110	0.131	0.018
					0.150	0.310	0.020
8	72	264	600	1	0.179	0.135	0.018
					0.220	0.300	0.020
12	108	527	1162	1	0.280	0.137	0.018
					0.320	0.300	0.020
16	144	1033	2210	1	0.468	0.126	0.018
					0.510	0.290	0.020
20	180	1564	3308	1	0.659	0.136	0.017
					0.700	0.300	0.020
24	216	2265	4746	1	0.917	0.136	0.017
					0.960	0.300	0.020
28	252	3152	6556	1	1.235	0.128	0.018
					1.280	0.290	0.020
32	288	3988	8264	2	1.627	0.185	0.028
					2.640	0.320	0.030
36	324	5071	10466	2	2.012	0.173	0.028
					3.030	0.310	0.030
40	360	6413	13186	2	2.547	0.175	0.028
					3.570	0.310	0.030
44	396	7821	16038	2	3.005	0.267	0.028
					4.020	0.470	0.030
48	432	8241	16914	4	3.457	0.431	0.059
					10.370	0.590	0.060
52	468	10733	21934	4	4.317	0.415	0.059
					7.250	0.570	0.060
56	504	12607	25718	4	5.030	0.424	0.059
					6.950	0.580	0.060
60	540	14300	29140	4	5.629	0.430	0.058
					6.550	0.590	0.060
64	576	16475	33526	8	7.223	0.752	0.122
					21.480	0.870	4.090

Table B4 — Timings for the Nettek Implementation of an Associative Memory Net
Compiled with Software Floating Point

N	Units	Connections	Processors	VP Ratio	Make (s)	FF (s)	BP (s)
					CM Total	CM Total	CM Total
1	9	4	17	1	0.084	0.077	-0.005
					0.130	0.160	0.030
4	36	64	164	1	0.113	0.078	0.027
					0.160	0.160	0.030
8	72	258	588	1	0.189	0.147	0.027
					0.230	0.310	0.030
12	108	638	1384	1	0.319	0.103	0.027
					0.360	0.300	0.030
16	144	1041	2226	1	0.479	0.136	0.027
					0.520	0.300	0.030
20	180	1544	3268	1	0.668	0.146	0.027
					0.710	0.320	0.030
24	216	2294	4804	1	0.940	0.138	0.027
					0.990	0.300	0.030
28	252	3070	6392	1	1.205	0.145	0.027
					1.250	0.310	0.030
32	288	4076	8440	2	1.668	0.195	0.038
					2.690	0.330	0.040
36	324	5165	10654	2	2.059	0.194	0.038
					3.080	0.330	0.040
40	360	6369	13098	2	2.507	0.284	0.038
					3.530	0.490	0.040
44	396	7637	15670	2	2.943	0.290	0.038
					3.960	0.490	0.040
48	432	8228	16888	4	3.454	0.434	0.069
					8.380	0.590	0.070
52	468	10811	22090	4	4.354	0.437	0.059
					6.290	0.600	0.060
56	504	12592	25688	4	5.126	0.442	0.068
					6.060	0.600	0.070
60	540	14342	29224	4	5.689	0.436	0.069
					7.610	0.600	0.070
64	576	16362	33300	8	7.276	0.779	0.131
					21.490	0.900	4.090

Table B5 — Timings for the Tomboulian Implementation of a Continuous Mapping Net
Compiled with Hardware Floating Point

N	Units Processors	Connec- tions	VP Ratio	TQ	Slot Use (%)	Make (s)	FF (s)	BP (s)
						CM Total	CM Total	CM Total
1	8	18	1	9	0.076	0.685	0.098	0.049
						2.060	0.200	0.070
2	15	63	1	21	0.169	3.853	0.150	0.113
						12.080	0.300	0.160
3	22	136	1	33	0.273	12.673	0.432	0.173
						38.800	0.740	0.250
4	29	237	1	45	0.426	29.023	0.269	0.232
						94.400	0.440	0.330
5	36	366	1	64	0.507	63.336	0.722	0.329
						195.100	1.160	0.460
6	43	523	1	82	0.606	112.862	0.906	0.416
						348.150	1.400	0.580
7	50	708	1	109	0.694	192.492	1.159	0.556
						594.000	1.770	0.780
8	57	921	1	***	***	***	***	***
						***	***	***

Table B6 — Timings for the Tomboulian Implementation of an Associative Memory Net
Compiled with Hardware Floating Point

N	Units Processors	Connec- tions	VP Ratio	TQ	Slot Use (%)	Make (s)	FF (s)	BP (s)
						CM Total	CM Total	CM Total
1	9	4	1	4	0.034	0.153	0.028	0.018
						0.752	0.063	0.039
2	18	15	1	12	0.065	0.444	0.121	0.050
						3.061	0.249	0.112
3	27	39	1	16	0.160	1.378	0.158	0.072
						8.564	0.363	0.151
4	36	60	1	19	0.206	2.264	0.167	0.077
						13.821	0.379	0.169
5	45	102	1	26	0.371	5.205	0.221	0.105
						32.456	0.526	0.265
6	54	150	1	33	0.496	9.033	0.285	0.146
						52.893	0.626	0.290
7	63	193	1	43	0.458	14.250	0.355	0.357
						83.755	0.768	0.762
8	72	257	1	49	0.655	22.298	0.421	0.390
						126.665	0.946	0.868

Appendix C

*LISP CODE FOR NETTALK IMPLEMENTATION

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

(in-package 'lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Nettalk Implementation
;;;
;;; Pvar Types
;;;
;;;   MAX-INT-PVAR           unsigned byte   [0 , 216-1]
;;;   CUBE-ADDRESS-PVAR     unsigned byte   [0 , *log-number-of-processors-limit*]
;;;   SLAB-NO-PVAR          unsigned byte   [0 , 216-1]
;;;   UNIT-NO-PVAR          unsigned byte   [0 , 216-1]
;;;   BUNDLE-NO-PVAR        unsigned byte   [0 , 216-1]
;;;   CONNECTION-NO-PVAR    unsigned byte   [0 , 232-1]
;;;

; Pvar type field sizes
;
; (defvar *cm-max-int*)
; (defvar *slab-no-size*)
; (defvar *unit-no-size*)
; (defvar *bundle-no-size*)
; (defvar *connection-no-size*)

; Set field sizes in compiler's environment
;
; (eval-when (compile load eval)
;   (setf *cm-max-int* (expt 2 16))
;   (setf *slab-no-size* 16)
;   (setf *unit-no-size* 16)
;   (setf *bundle-no-size* 16)
;   (setf *connection-no-size* 32)
; )

; Define pvar types
;
; (deftype max-int-pvar () '(pvar (unsigned-byte
;                                #.(1+ (ceiling (log *cm-max-int* 2))))))

; (deftype cube-address-pvar () '(pvar (unsigned-byte
;                                       *log-number-of-processors-limit*)))

; (deftype slab-no-pvar () '(pvar (unsigned-byte #.*slab-no-size*)))

; (deftype unit-no-pvar () '(pvar (unsigned-byte #.*unit-no-size*)))

; (deftype bundle-no-pvar () '(pvar (unsigned-byte #.*bundle-no-size*)))
```

```

(deftype connection-no-pvar () '(pvar (unsigned-byte #.*connection-no-size*)))

; When using simulator, add new pvar type symbols
;
#+:lisp-simulator
(progn
  (pushnew 'max-int-pvar **lisp-exported-type-symbols*)
  (pushnew 'cube-address-pvar **lisp-exported-type-symbols*)
  (pushnew 'slab-no-pvar **lisp-exported-type-symbols*)
  (pushnew 'unit-no-pvar **lisp-exported-type-symbols*)
  (pushnew 'bundle-no-pvar **lisp-exported-type-symbols*)
  (pushnew 'connection-no-pvar **lisp-exported-type-symbols*)
)

;;; EOF

#+:ccl
(format t "~%\nPvar Types\ " loaded")



---




---



;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

(in-package 'lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Mettalk Implementation
;;;
;;; Utilities
;;;

; COUNT-CSS returns the number of processors in the currently selected set.
;
(defun count-css ()
  (*sum (! 1)))

; MAX-INT!! returns a field pvar containing *CM-MAX-INT*.
;
(defmacro max-int!! ()
  '(the max-int-pvar (! *cm-max-int*)))

; RANDOM-FLOAT!! returns a random float pvar evenly distributed in the interval
; [mean-interval , mean+interval] in each processor.
;
#+:lisp-simulator
(*proclaim '(ftype (function (t) (pvar single-float)) random-float!!))

(*defun random-float!! (mean!! interval!!)
  (declare (type float-pvar mean!! interval!!))
  (+!! mean!!
    (*!! interval!!
      (if!! (=!! (random!! (! 2)) (! 1))
        (!! 1.0)
        (!! -1.0))
      (/!! (random!! (max-int!!))
           (max-int!!))))))

; FORMAT-PVARS pretty prints the given list of PVARs. Additional keyword arguments
; will be passed in to PRETTY-PRINT-PVAR.

```

```

;
(defmacro format-pvars (pvars &rest keys &key &allow-other-keys)
  '(progn
    ,@(mapcan #'(lambda (pvar)
      '((format t "%~a" (pvar-name ,pvar)) (ppp ,pvar ,@keys)))
      pvars)))

; MULTIPLE-VALUE-SETF sets the locations referenced by ACCESSOR-FORMS
; to the multiple values returned by VALUES-FORM.
;
(defmacro multiple-value-setf (accessor-forms values-form)
  (let ((values-list (gensym))
        (i -1))
    '(let ((values-list (multiple-value-list ,values-form)))
      (setf ,@(mapcan #'(lambda (accessor-form)
        '(',accessor-form (nth ,(incf i) ,values-list)))
        accessor-forms))))))

; PRINT-VEC prints the array VEC on STREAM with the given ELEMENTS-PER-LINE.
; Each element is printed using ELEMENT-FORMAT, and each line of output is
; preceded by NEW-LINE-FORMAT.
;
(defun print-vec (vec &optional (stream t)
                 &key elements-per-line (element-format "~s ") (new-line-format "%~n"))
  (dotimes (i (length vec))
    (if (and elements-per-line
              (zerop (mod i elements-per-line)))
        (format stream "~@?" new-line-format)
        (format stream "~@?" element-format (aref vec i)))
    (values)))

; PRINT-IO-VECS prints the INPUT and OUTPUT vectors.
;
(defun print-io-vecs (input output)
  (format t "%i: ") (print-vec input)
  (format t " o: ") (print-vec output))

; LIST-TO-ARRAY-PAIRS coerces the list of LIST-PAIRS into a list of array pairs.
;
(defun list-to-array-pairs (list-pairs)
  (let (input target)
    (map 'list
      #'(lambda (pair)
        (setf input (first pair)
              target (second pair))
        (list
          (make-array (length input) :initial-contents input)
          (make-array (length target) :initial-contents target)))
      list-pairs)))

; LIST-TO-ARRAY coerces the list of lists into a list of arrays.
;
(defun list-to-array (list)
  (map 'list
    #'(lambda (sub-list)
      (make-array (length sub-list) :initial-contents sub-list))
    list))

;;; EOF

#+:ccl
(format t "%~n\"Utilities\" loaded")

```

```

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

(in-package '*lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Nettalk Implementation
;;;
;;; Processor Allocation
;;;

;
; CM Dimensions
;

; *CM-DIMENSIONS* is a list of CM configurations. Each configuration is a list
; of the total number of processors and the corresponding CM dimensions.
;
(defvar *cm-dimensions*
  (mapcar #'(lambda (dims)
            (list (reduce #'* dims)
                  #+*lisp-hardware ; 8K machine
                  '(( 64 128)      ; VP ratio    = 1
                    ( 128 128)    ;           = 2
                    ( 128 256)    ;           = 4
                    ( 256 256)    ;           = 8
                    ( 256 512)    ;           = 16
                    ( 512 512)    ;           = 32
                    ( 512 1024)   ;           = 64
                    (1024 1024))  ;           = 128
                  #+*lisp-simulator
                  '((4 4)
                    (6 4)
                    (6 6)
                    (8 6)
                    (8 8))
                ))

; CM-BEST-FIT-DIMS returns the minimum CM dimensions necessary
; to satisfy the request for NO-PROCESSORS.
;
(defun cm-best-fit-dims (no-processors)
  (second (assoc no-processors
                *cm-dimensions*
                :test #'<=)))

;
; Processor Allocator - allocate contiguous blocks of free processors, no deallocation
;

; *NEXT-FREE-PROCESSOR* contains the cube address of the next free processor.
;
(defvar *next-free-processor* 0)

; RESET-PALLOC reset the processor allocator.
;
(defun reset-palloc ()
  (setf *next-free-processor* 0)
  t)

; N-PROC-LEFT-P returns T if there are N free processors available or
; signals an error if there are too few free processors left.

```

```

;
(defun n-proc-left-p (n)
  (or (<= (+ *next-free-processor* n) *number-of-processors-limit*)
      (error "PALLOC can't allocate ~a processor~:~[s~;~;~s~]" n)))

; PALLOC allocates the next N free processors.
;
(defun palloc (n)
  (when (n-proc-left-p n)
    (progn
      *next-free-processor*
      (incf *next-free-processor* n))))

; PALLOC-1 allocates a single free processor.
;
(defmacro palloc-1 ()
  '(palloc 1))

; FOR-PROCESSOR-BLOCK executes BODY with the currently selected set composed
; of the block of SIZE processors beginning at START-ADDR.
;
(defmacro for-processor-block ((start-addr size) &body body)
  '(when (<=!! (the cube-address-pvar (!! ,start-addr))
             (self-address!!)
             (the cube-address-pvar (!! (1- (+ ,start-addr ,size)))))
    ,@body))

; WITH-N-PROC-ALLOCATED allocates a block of N processors, sets ADDR
; to the starting address of the block and executes BODY with the
; currently selected set composed of the newly allocated processors.
;
(defmacro with-n-proc-allocated ((n addr)
                                &body body)
  '(let ((,addr (palloc ,n)))
    (for-processor-block (,addr ,n)
      ,@body)))

; In Allegro CL, set FRED indentation for macros
;
#+:ccl
(progn
  (pushnew '(with-n-proc-allocated . 1)
    ccl::*fred-special-indent-alist*
    :test #'equal)
  (pushnew '(for-processor-block . 1)
    ccl::*fred-special-indent-alist*
    :test #'equal))

; Reset Processor Allocator after *COLD-BOOT.
;
#+:*lisp-hardware
(add-initialization "Reset Palloc"
  '(reset-palloc)
  '*after-*cold-boot-initializations*)

#+:*lisp-simulator
(add-initialization :name-of-form "Reset Palloc"
  :form '(reset-palloc)
  :variable '*after-*cold-boot-initializations*)

;;; EOF

#+:ccl
(format t "~%\"Processor Allocation\" loaded")

```

```

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

(in-package '*lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Nettalk Implementation
;;;
;;; Neural Net Front-End Structures
;;;
;;; SLAB set of units
;;; BUNDLE set of connections between 2 slabs with density 0-100%
;;; NET sets of slabs and bundles,
;;; with input & output slab (possibly the same)
;;; represents CONTINUOUS-MAPPING or ASSOCIATIVE-MEMORY net
;;;

;
; Slab of units
;
(defstruct (net-slab
  #:symbolics
  (:named)
  (:conc-name slab-)
  (:constructor fe-make-slab-internal)
  (:print-function print-slab)
)
  no ; slab id
  inputp ; input slab?
  outputp ; output slab?
  size ; no of units
)

; PRINT-SLAB prints SLAB on STREAM.
;
(defun print-slab (slab stream &optional depth)
  (declare (ignore depth))
  (format stream "%<Slab ~a, ~a unit~:*~[s~;~;~;~]~::~~[~; I~]~::~~[~; O~]>"
    (slab-no slab) (slab-size slab)
    (slab-inputp slab) (slab-outputp slab)))

; SLAB-NO!! returns a slab-no-pvar pvar containing the SLAB id in each processor.
;
(defmacro slab-no!! (slab)
  '(the slab-no-pvar (!! (slab-no ,slab))))

; SLAB-INPUTP!! returns a boolean pvar containing T if SLAB is the input slab.
;
(defmacro slab-inputp!! (slab)
  '(the boolean-pvar (!! (slab-inputp ,slab))))

; SLAB-OUTPUTP!! returns a boolean pvar containing T if SLAB is the output slab.
;
(defmacro slab-outputp!! (slab)
  '(the boolean-pvar (!! (slab-outputp ,slab))))

;
; Bundle of connections
;
(defstruct (net-bundle
  #:symbolics
  (:named)

```

```

      (:conc-name bundle-)
      (:constructor fe-make-bundle-internal)
      (:print-function print-bundle)
    )
  no ; bundle id
  to-slab ; connections to slab
  from-slab ; connections from slab
  density ; density of connections, 0-100%
  size ; no of connections
      ; connection = (to-no,from-no)
  to-no ; array of to-slab unit ids
  from-no ; array of from-slab unit ids
)

; PRINT-BUNDLE prints BUNDLE on STREAM.
;
(defun print-bundle (bundle stream &optional depth)
  (declare (ignore depth))
  (format stream "#<Bundle ~a <- ~a, ~a%>"
    (slab-no (bundle-to-slab bundle))
    (slab-no (bundle-from-slab bundle))
    (bundle-density bundle)))

; BUNDLE-NO!! returns a bundle-no-pvar pvar containing the BUNDLE id in each processor.
;
(defmacro bundle-no!! (bundle)
  '(the bundle-no-pvar (!! (bundle-no ,bundle))))

;
; Neural net
;
(defstruct (neural-net
  #+:symbolics
  (:named)
  (:conc-name net-)
  (:constructor fe-make-net-internal)
  (:print-function print-net)
)
  name
  type ; CONTINUOUS-MAPPING or ASSOCIATIVE-MEMORY

  slabs ; array of slabs
  input-slab-no ; input slab id
  output-slab-no ; output slab id
  bundles ; array of bundles
  no-units ; total number of units
  no-connections ; connections
  no-processors ; processors

  (a 1.0) ; dynamical system equation constants
  (b 1.0)

  (eta 0.25) ; learning rate
  (alpha 0.9) ; momentum term

  (epsilon-x 0.001) ; feed-forward convergence criterion
  (x-iterations 4) ; min iterations before convergence test
  (epsilon-y 0.001) ; back-propagate convergence criterion
  (y-iterations 4) ; min iterations before convergence test
  (epsilon-w 0.1) ; weight update convergence criterion
  (max-updates 10000) ; max weight updates in training
)

; PRINT-NET prints NET on STREAM.
;
(defun print-net (net stream &optional depth &key verbose-p)
  (declare (ignore depth))
  (if (not verbose-p)
      (format stream

```

```

    "Net ~a ~a slab~:*~[s~;~;s~], ~a bundle~:*~[s~;~;s~]"
    (net-name net)
    (length (net-slabs net))
    (length (net-bundles net))
    (format stream "~%a net: ~a" (net-type net) (net-name net))
    (format stream "~2%a slab~:*~[s~;~;s~], ~a bundle~:*~[s~;~;s~]"
      (length (net-slabs net)) (length (net-bundles net)))
    (format stream "~%a unit~:*~[s~;~;s~], ~a connection~:*~[s~;~;s~] ->
      ~a processor~:*~[s~;~;s~]"
      (net-no-units net) (net-no-connections net) (net-no-processors net))
    (format stream "~2%a = ~a, b = ~a" (net-a net) (net-b net))
    (format stream "~2%Feed-forward convergence = ~a, min ~a iteration~:*~[s~;~;s~]"
      (net-epsilon-x net) (net-x-iterations net))
    (format stream "~2%Back-propagate convergence = ~a, min ~a iteration~:*~[s~;~;s~]"
      (net-epsilon-y net) (net-y-iterations net))

    (format stream "~2%eta = ~a, alpha = ~a" (net-eta net) (net-alpha net))
    (format stream "~2%Weight update convergence = ~a, max ~a iteration~:*~[s~;~;s~]"
      (net-epsilon-w net) (net-max-updates net))
    (terpri stream)
  )
)

; GET-SLAB returns the slab with id SLAB-NO in NET.
;
(defmacro get-slab (net slab-no)
  '(aref (net-slabs ,net) ,slab-no))

; GET-INPUT-SLAB returns the input slab in NET.
;
(defmacro get-input-slab (net)
  '(aref (net-slabs ,net) (net-input-slab-no ,net)))

; GET-OUTPUT-SLAB returns the output slab in NET.
;
(defmacro get-output-slab (net)
  '(aref (net-slabs ,net) (net-output-slab-no ,net)))

; GET-BUNDLE returns the bundle with id BUNDLE-NO in NET.
;
(defmacro get-bundle (net bundle-no)
  '(aref (net-bundles ,net) ,bundle-no))

; MEMORY-NETP returns T if NET is an ASSOCIATIVE-MEMORY net.

(defmacro memory-netp (net)
  '(eq (net-type ,net) 'associative-memory))

;;; EOF

#:ccl
(format t "~%\\"Net FE Structures\\" loaded")

```

```

;;; -- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 --

```

```

(in-package 'lisp)

```

```

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

```

```

;;;

```

```

;;; Nettek Implementation
;;;
;;; Net CM Structures
;;;

;
; Net
;

(*proclaim '(type boolean-pvar netp!!))

(*defvar netp!! nil!!)                ; processor in net?

;
; Units
;

(*proclaim '(type boolean-pvar unitp!!))
(*proclaim '(type boolean-pvar inputp!!))
(*proclaim '(type boolean-pvar outputp!!))
(*proclaim '(type slab-no-pvar slab-no!!))
(*proclaim '(type unit-no-pvar unit-no!!))

(*defvar unitp!! nil!!)                ; processor is unit?
(*defvar inputp!! nil!!)               ; input unit?
(*defvar outputp!! nil!!)              ; output unit?
(*defvar slab-no!!)                    ; unit slab id
(*defvar unit-no!!)                    ; unit id

; Variables appearing in feed-forward and back-propagation equations
;
(*proclaim '(type single-float-pvar a!!))
(*proclaim '(type single-float-pvar b!!))
(*proclaim '(type single-float-pvar X!!))
(*proclaim '(type single-float-pvar Z!!))
(*proclaim '(type single-float-pvar dX!!))
(*proclaim '(type single-float-pvar U!!))
(*proclaim '(type single-float-pvar LogU!!))
(*proclaim '(type single-float-pvar I!!))
(*proclaim '(type single-float-pvar Y!!))
(*proclaim '(type single-float-pvar dY!!))
(*proclaim '(type single-float-pvar V!!))
(*proclaim '(type single-float-pvar J!!))

(*defvar a!!)
(*defvar b!!)
(*defvar X!!)
(*defvar Z!!)
(*defvar dX!!)
(*defvar U!!)
(*defvar LogU!!)
(*defvar I!!)
(*defvar Y!!)
(*defvar dY!!)
(*defvar V!!)
(*defvar J!!)

(*proclaim '(type single-float-pvar epsilon-x!!))
(*proclaim '(type single-float-pvar epsilon-y!!))

(*defvar epsilon-x!!)                  ; feed-forward convergence criterion
(*defvar epsilon-y!!)                  ; back-propagate convergence criterion

;
; Connections
;

(*proclaim '(type boolean-pvar fan-inp!!))
(*proclaim '(type boolean-pvar fan-outp!!))

```

```

(*proclaim '(type bundle-no-pvar bundle-no!))
(*proclaim '(type connection-no-pvar connection-no!))
(*proclaim '(type slab-no-pvar to-slab-no!))
(*proclaim '(type unit-no-pvar to-unit-no!))
(*proclaim '(type slab-no-pvar from-slab-no!))
(*proclaim '(type unit-no-pvar from-unit-no!))
(*proclaim '(type cube-address-pvar from-addr!))
(*proclaim '(type cube-address-pvar to-addr!))

(*defvar fan-inp!! nil!!)           ; fan-in weight?
(*defvar fan-outp!! nil!!)         ; fan-out weight?
(*defvar bundle-no!!)              ; connection bundle id
(*defvar connection-no!!)          ; connection id
(*defvar to-slab-no!!)              ; to slab id
(*defvar to-unit-no!!)             ; to unit id
(*defvar from-slab-no!!)           ; from slab id
(*defvar from-unit-no!!)          ; from unit id
(*defvar from-addr!!)             ; cube address of from unit
(*defvar to-addr!!)               ; cube address of to unit

(*proclaim '(type single-float-pvar W!))
(*proclaim '(type single-float-pvar dW!))
(*proclaim '(type single-float-pvar dWold!))

(*defvar W!!)                      ; connection weight
(*defvar dW!!)                     ; current weight change
(*defvar dWold!!)                  ; last weight change (momentum term)

(*proclaim '(type single-float-pvar epsilon-w!))
(*proclaim '(type single-float-pvar eta!))
(*proclaim '(type single-float-pvar alpha!))

(*defvar epsilon-w!!)              ; weight update convergence criterion
(*defvar eta!!)                    ; learning rate
(*defvar alpha!!)                  ; momentum term

;
; Fan-in & Fan-out Segments
;

(*proclaim '(type boolean-pvar forward-fan-in-seg!))
(*proclaim '(type boolean-pvar forward-fan-out-seg!))
(*proclaim '(type boolean-pvar backward-fan-in-seg!))
(*proclaim '(type boolean-pvar backward-fan-out-seg!))

(*defvar forward-fan-in-seg!!)     ; feed-forward fan-in weights
(*defvar forward-fan-out-seg!!)    ; feed-forward fan-out weights
(*defvar backward-fan-in-seg!!)    ; back-propagate fan-in weights
(*defvar backward-fan-out-seg!!)   ; back-propagate fan-out weights

;;; EOF

#+:ccl
(format t "~%\\"CM Net Structures\\" loaded")

|
|
|

|
|
|

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

(in-package '*lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

```

```

;;;
;;; Nettek Implementation
;;;
;;; Make Net Structures on Front-End
;;;

; Front-End slab structure

; FE-MAKE-SLAB returns a net slab of SIZE units with id NO. INPUTP and OUTPUTP
; indicate if this slab is the input or output slab, respectively.
;
(defun fe-make-slab (no size inputp outputp)
  (fe-make-slab-internal :no no
                        :size size
                        :inputp inputp
                        :outputp outputp))

; Front-End bundle structure

; RANDOM-CONNECTIONS returns ROW number and COL number arrays representing a bundle
; of connections to a slab of TO-SIZE units from a slab of FROM-SIZE units.
; Each possible connection is formed with probability given by DENSITY.
;
(defun random-connections (to-size from-size density)
  (let (n row col)
    (setf density (/ density 100.0))
    (*all
     (*let ((rendezvous!! (!! 0)))
       (declare (type cube-address-pvar rendezvous!!))
       (*when (and!! (<!! (self-address!!)
                          (the max-int-pvar (!! (* to-size from-size))))
              (<!! (random-float!! (!! 0.5) (!! 0.5))
                   (the float-pvar (!! density))))
         (setf n (count-css))
         (*pset :no-collisions
                (self-address!!)
                rendezvous!!
                (enumerate!!)))
       (setf row (make-array n)
              col (make-array n))
       (pvar-to-array (truncate!! rendezvous!! (!! from-size))
                      row
                      :cube-address-end n)
       (pvar-to-array (mod!! rendezvous!! (!! from-size))
                      col
                      :cube-address-end n)))
    (values row col)))

; FE-MAKE-BUNDLE returns a bundle with id NO connecting TO-SLAB and FROM-SLAB
; with the probability of each connection given by DENSITY.
;
(defun fe-make-bundle (no to-slab from-slab density)
  (let ((bundle (fe-make-bundle-internal
                :no no
                :to-slab to-slab
                :from-slab from-slab
                :density density)))
    (multiple-value-setf
     ((bundle-to-no bundle)
      (bundle-from-no bundle))
     (random-connections (slab-size to-slab)
                        (slab-size from-slab)
                        density))
     (setf (bundle-size bundle) (length (bundle-to-no bundle)))
     bundle))

; Front-End net structure

```

```

; A bundle spec is a list of the form (<to slab id> <from slab id> <density>).
;
(defunstruct (bundle-spec
             (:type list))
  to-slab
  from-slab
  density)

; FE-MAKE-NET returns the net NAME of the given TYPE (CONTINUOUS-MAPPING or
; ASSOCIATIVE-MEMORY). SLABS must be a list of total units in each slab, and
; INPUT-SLAB-NO and OUTPUT-SLAB-NO identify the input and output slabs,
; respectively. BUNDLES must be a list of bundle specifications. Additional
; keyword arguments are passed in to FE-MAKE-NET-INTERNAL allowing other
; net parameters to be set.
;
(defun fe-make-net (name type slabs input-slab-no output-slab-no bundles
                  &rest other-net-keys &key &allow-other-keys)
  (let ((net (apply #'fe-make-net-internal
                   :name (string name)
                   :type type
                   :input-slab-no input-slab-no
                   :output-slab-no output-slab-no
                   other-net-keys)))
    (let ((slab-no -1))
      (setf (net-slabs net)
            (map 'array
                 #'(lambda (slab-size)
                     (fe-make-slab (incf slab-no)
                                   slab-size
                                   (eq input-slab-no slab-no)
                                   (eq output-slab-no slab-no)))
                 slabs)))
      (let ((bundle-no -1))
        (setf (net-bundles net)
              (map 'array
                   #'(lambda (bundle)
                       (fe-make-bundle (incf bundle-no)
                                       (get-slab net (bundle-spec-to-slab bundle))
                                       (get-slab net (bundle-spec-from-slab bundle))
                                       (bundle-spec-density bundle)))
                   bundles)))
          (fe-size-net net)
          net)))

; FE-SIZE-NET sets the total number of units, connections and processors required by NET.
;
(defun fe-size-net (net)
  (setf (net-no-units net)
        (reduce #'+
                (map 'list #'slab-size (net-slabs net))))
        (net-no-connections net)
        (reduce #'+
                (map 'list #'bundle-size (net-bundles net))))
        (net-no-processors net)
        (+ (net-no-units net)
            (* 2 (net-no-connections net))))))

; CONTINUOUS-MAPPING net

; DEF-MAPPING-NET returns a CONTINUOUS-MAPPING net called NAME specified by
; the keyword arguments SLABS, INPUT-SLAB-NO, OUTPUT-SLAB-NO and BUNDLES.
; Additional keyword arguments can be used to specify other net parameters.
;
(defmacro def-mapping-net (name &rest other-net-keys
                          &key slabs input-slab-no output-slab-no bundles
                          &allow-other-keys)
  `(progn
    (defvar ,name)
    (setf ,name (fe-make-net ',name
                              slabs input-slab-no output-slab-no bundles
                              &allow-other-keys))))

```

```

        'continuous-mapping
        ,slabs
        ,input-slab-no
        ,output-slab-no
        ,bundles
        ,@other-net-keys))
    (cm-net-cold-boot ,name)
    (cm-make-net ,name)))

; ASSOCIATIVE-MEMORY net

; DEF-MEMORY-NET returns an ASSOCIATIVE-MEMORY net called NAME specified by
; the keyword arguments SLABS, INPUT-SLAB-NO and BUNDLES. Additional keyword
; arguments can be used to specify other net parameters.
;
(defmacro def-memory-net (name &rest other-net-keys
                        &key slabs input-slab-no bundles
                        &allow-other-keys)
  '(progn
    (defvar ,name)
    (setf ,name (fe-make-net ',name
                            'associative-memory
                            ,slabs
                            ,input-slab-no
                            ,input-slab-no
                            ,bundles
                            ,@other-net-keys))
    (cm-net-cold-boot ,name)
    (cm-make-net ,name)))

;;; EOF

#+:ccl
(format t "~%" "FE Make Net" loaded")

```

```

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

```

```

(in-package 'lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Nettalk Implementation
;;;
;;; Make Net Structures on CM
;;;

;
; CM slab structure
;

; CM-MAKE-SLAB creates the structure for SLAB on the CM.
;
(defun cm-make-slab (slab)
  (let ((slab-size (slab-size slab)))
    (with-n-proc-allocated (slab-size slab-start)
      (*set netp!! t!!
        unitp!! t!!
        inputp!! (slab-inputp!! slab)
        outputp!! (slab-outputp!! slab)

```

```

        slab-no!! (slab-no!! slab)
        unit-no!! (enumerate!!)))
slab)

;
; CM bundle structure
;
; CM-MAKE-BUNDLE creates the structure for BUNDLE on the CM.
; IN-OUT may specify a bundle of FAN-IN or FAN-OUT weights.
;
(defun cm-make-bundle (bundle in-out)
  (assert (member in-out '(:fan-in :fan-out))
          (in-out)
          "IN-OUT must be :FAN-IN or :FAN-OUT")
  (let ((to-slab (bundle-to-slab bundle))
        (from-slab (bundle-from-slab bundle))
        (bundle-size (bundle-size bundle)))
    (with-n-proc-allocated (bundle-size bundle-start)
      (*set netp!! t!!
        bundle-no!! (bundle-no!! bundle)
        connection-no!! (enumerate!!)
        to-slab-no!! (slab-no!! to-slab)
        from-slab-no!! (slab-no!! from-slab))
      (if (eq in-out :fan-in)
          (*set fan-inp!! t!!)
          (*set fan-outp!! t!!))
      (array-to-pvar (bundle-to-no bundle)
                    to-unit-no!!
                    :cube-address-start bundle-start
                    :cube-address-end (+ bundle-start bundle-size))
      (array-to-pvar (bundle-from-no bundle)
                    from-unit-no!!
                    :cube-address-start bundle-start
                    :cube-address-end (+ bundle-start bundle-size))))
  bundle)

;
; CM net structure
;
; CM-NET-COLD-BOOT cold boots the CM with the dimensions necessary for NET.
;
(defun cm-net-cold-boot (net)
  (let ((cm-dims (cm-best-fit-dims (net-no-processors net))))
    (or cm-dims
        (error "Net ~a too large for CM" (net-name net)))
    (*cold-boot :initial-dimensions cm-dims)))

; CM-MAKE-NET creates the structure for NET on the CM.
;
(defun cm-make-net (net)
  (map nil #'(lambda (bundle)
              (cm-make-bundle bundle :fan-in))
        (net-bundles net))
  (map nil #'(lambda (slab)
              (cm-make-slab slab))
        (net-slabs net))
  (map nil #'(lambda (bundle)
              (cm-make-bundle bundle :fan-out))
        (net-bundles net))
  (cm-sort-net net)
  (*all
   (*when unitp!!
    (*set a!! (the float-pvar (!! (net-a net)))
          b!! (the float-pvar (!! (net-b net)))
          epsilon-x!! (the float-pvar (!! (net-epsilon-x net)))
          epsilon-y!! (the float-pvar (!! (net-epsilon-y net))))
    (*when outputp!!
```

```

    (*set epsilon-w!! (the float-pvar (!! (net-epsilon-w net))))))
(*when fan-outp!!
  (*set eta!! (the float-pvar (!! (net-eta net)))
    alpha!! (the float-pvar (!! (net-alpha net))))))
(cm-make-segments)
(cm-reset-weights)
net)

;
; Sort CM net structures
;

; CM-SORT-NET sorts the structures of NET on the CM to establish the
; proper interleaving of fan-in weights, units and fan-out weights.
;
(defun cm-sort-net (net)
  (*all
    (*let (key!!
          rank!!)
      (declare (type (pvar (unsigned-byte (+ *slab-no-size* *unit-no-size*))) key!!))
      (declare (type cube-address-pvar rank!!))
      (*when (or!! unitp!! fan-inp!! fan-outp!!)
        (*set key!!
          (cond!! (fan-inp!! to-slab-no!!)
                  (unitp!! slab-no!!)
                  (fan-outp!! from-slab-no!!)
                  (t!! (!! 0))))
          (*set key!! (ash!! key!! (the unit-no-pvar (!! *unit-no-size*))))
          (*set key!!
            (+!! key!!
              (cond!! (fan-inp!! to-unit-no!!)
                      (unitp!! unit-no!!)
                      (fan-outp!! from-unit-no!!)
                      (t!! (!! 0))))))
          (*set rank!! (rank!! key!! '<=!!))

          (cm-sort-units rank!!)
          (cm-sort-connections rank!!)

          (map nil #'cm-link-bundle (net-bundles net))
          )))
    net)

; CM-SORT-UNITS copies the units to the cube addresses in TO-ADDR!!.
;
(defun cm-sort-units (to-addr!!)
  (declare (type cube-address-pvar to-addr!!))
  (*when unitp!!
    (*set unitp!! nil!!)
    (*pset :no-collisions t!! unitp!! to-addr!!)
    (*when inputp!!
      (*set inputp!! nil!!)
      (*pset :no-collisions t!! inputp!! to-addr!!))
    (*when outputp!!
      (*set outputp!! nil!!)
      (*pset :no-collisions t!! outputp!! to-addr!!))
    (*pset :no-collisions slab-no!! slab-no!! to-addr!!)
    (*pset :no-collisions unit-no!! unit-no!! to-addr!!)
    ))

; CM-SORT-CONNECTIONS copies the fan-in and fan-out weights to
; the cube addresses in TO-ADDR!!.
;
(defun cm-sort-connections (to-addr!!)
  (declare (type cube-address-pvar to-addr!!))
  (*when (or!! fan-inp!! fan-outp!!)
    (*when fan-inp!!
      (*set fan-inp!! nil!!)
      (*pset :no-collisions t!! fan-inp!! to-addr!!))

```

```

(*when fan-outp!!
  (*set fan-outp!! nil!!)
  (*pset :no-collisions t!! fan-outp!! to-addr!!))
(*pset :no-collisions bundle-no!! bundle-no!! to-addr!!)
(*pset :no-collisions connection-no!! connection-no!! to-addr!!)
(*pset :no-collisions to-slab-no!! to-slab-no!! to-addr!!)
(*pset :no-collisions to-unit-no!! to-unit-no!! to-addr!!)
(*pset :no-collisions from-slab-no!! from-slab-no!! to-addr!!)
(*pset :no-collisions from-unit-no!! from-unit-no!! to-addr!!)
))

; CM-LINK-BUNDLE links the fan-in and fan-out weights of BUNDLE.
;
(defun cm-link-bundle (bundle)
  (let ((to-slab (bundle-to-slab bundle))
        (from-slab (bundle-from-slab bundle)))
    (*all
      (*let (in-rendezvous!!
              out-rendezvous!!)
          (declare (type cube-address-pvar in-rendezvous!! out-rendezvous!!))
          (*when (and!! (or!! fan-inp!! fan-outp!!)
                      (==! to-slab-no!! (the slab-no-pvar (!! (slab-no to-slab))))
                      (==! from-slab-no!! (the slab-no-pvar (!! (slab-no from-slab))))))
            (*when fan-inp!!
              (*pset :no-collisions (self-address!!) in-rendezvous!! connection-no!!))
            (*when fan-outp!!
              (*pset :no-collisions (self-address!!) out-rendezvous!! connection-no!!))
            (*when fan-inp!!
              (*set from-addr!!
                (the cube-address-pvar
                  (pref!! out-rendezvous!! connection-no!!
                    :collision-mode :no-collisions))))
            (*when fan-outp!!
              (*set to-addr!!
                (the cube-address-pvar
                  (pref!! in-rendezvous!! connection-no!!
                    :collision-mode :no-collisions))))
            ))))
    ))))

;
; Make CM segments for feed-forward and back-propagate cycles.
;
; CM-MAKE-SEGMENTS makes the segments pvars used during scanning operations
; in the feed-forward and back-propagate cycles.
;
(defun cm-make-segments ()
  (*when netp!!
    (*set forward-fan-out-seg!! (or!! unitp!! fan-inp!!)
          forward-fan-in-seg!!
          (or!! (scan!! (or!! unitp!! fan-outp!!) 'and!!
                      :segment-pvar (or!! unitp!! fan-outp!!)
                      :include-self nil)
              fan-outp!!)
          backward-fan-in-seg!! (or!! unitp!! fan-outp!!)
          backward-fan-out-seg!!
          (or!! (scan!! (or!! unitp!! fan-inp!!) 'and!!
                      :segment-pvar (or!! unitp!! fan-inp!!)
                      :direction :backward
                      :include-self nil)
              fan-inp!!))
    ))

; CM-RESET-WEIGHTS resets the fan-out weights for each connection in the net
; to a random float in the interval [mean-interval , mean+interval].
;
(defun cm-reset-weights (&optional (mean 0.0) (interval 0.5))
  (declare (type float mean interval))

```



```

(*when (and!! fan-outp!! (=? bundle-no!! (bundle-no!! bundle)))
  (*pset :no-collisions pvar mail-box!! (enumerate!!))
  (pvar-to-array mail-box!! (make-array (bundle-size bundle)
    :cube-address-end (count-css))))))

; GET-BUNDLE-W returns an array containing the values of W!!
; for the bundle with id BUNDLE-NO in NET.
;
(defun get-bundle-W!! (net bundle-no)
  (get-bundle-pvar ,net ,bundle-no W!!))

;;; EOF

#+:ccl
(format t "~%" "CM Net Access\" loaded")

```

```

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

```

```

(in-package 'lisp)

```

```

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

```

```

;;;
;;; Nettalk Implementation
;;;
;;; Training Sets
;;;

```

```

; Training Exemplar
;

```

```

(defun exemplar
  (:type list))
  input-pvar
  input-vec
  target-pvar
  target-vec)

```

```

; Training Set
;

```

```

(defun training-set
  (:type list))
  type ; MAPPING-SET or MEMORY-SET
  name
  exemplars) ; list of exemplars

```

```

; GET-EXAMPLAR returns the exemplar with id EXAMPLAR -NO in TRAINING-SET.
;

```

```

(defun get-exemplar (training-set exemplar-no)
  (nth ,exemplar-no (training-set-exemplars ,training-set)))

```

```

; CM-LOAD-TRAINING-PAIR loads the INPUT/TARGET training vectors into the
; CONTINUOUS-MAPPING net structure on the CM and returns an exemplar
; containing the INPUT and TARGET vectors. The pvars corresponding
; to the training pair are marked with the given SET-NAME and PAIR-NO.
;

```

```

(defun cm-load-training-pair (set-name pair-no input target)
  (*all
    (let ((input!! (allocate!! nil
      (format nil "~a~a-I" set-name pair-no)
      'float-pvar)))

```

```

(target!! (allocate!! nil
          (format nil "~a~a-T" set-name pair-no)
          'float-pvar)))
(*let (mail-box!!)
  (declare (type float-pvar mail-box!!))
  (array-to-pvar input mail-box!! :cube-address-end (length input))
  (*when inputp!!
    (*set (the float-pvar input!!)
          (pref!! mail-box!! unit-no!! :collision-mode :no-collisions)))
  (array-to-pvar target mail-box!! :cube-address-end (length target))
  (*when outputp!!
    (*set (the float-pvar target!!)
          (pref!! mail-box!! unit-no!! :collision-mode :no-collisions))))
(list input!! input target!! target)))

; CM-LOAD-MAPPING-SET loads the TRAINING-PAIRS labeled SET-NAME into the
; CONTINUOUS-MAPPING net structure on the CM and returns the resulting
; training set. TRAINING-PAIRS must be a list of input/target vector lists.
;
(defun cm-load-mapping-set (set-name training-pairs)
  (let ((pair-no -1))
    (list 'mapping-set set-name
          (mapcar #'(lambda (pair)
                     (cm-load-training-pair set-name
                                             (incf pair-no)
                                             (first pair)
                                             (second pair)))
              training-pairs))))

; CM-LOAD-MEMORY-INPUT loads the INPUT vector into the CONTINUOUS-MAPPING
; net structure on the CM and returns an exemplar. The pvar corresponding
; to the INPUT vector is marked with the given SET-NAME and INPUT-NO.
;
(defun cm-load-memory-input (set-name input-no input)
  (*all
   (let ((input!! (allocate!! nil
                   (format nil "~a~a-I" set-name input-no)
                   'float-pvar)))
     (*let (mail-box!!)
       (declare (type float-pvar mail-box!!))
       (array-to-pvar input mail-box!! :cube-address-end (length input))
       (*when inputp!!
         (*set (the float-pvar input!!)
               (pref!! mail-box!! unit-no!! :collision-mode :no-collisions))))
     (list input!! input input!! input))))

; CM-LOAD-MEMORY-SET loads the TRAINING-LIST labeled SET-NAME into the
; CONTINUOUS-MAPPING net structure on the CM and returns the resulting
; training set. The TRAINING-LIST must be a list of input vectors.
;
(defun cm-load-memory-set (set-name training-set)
  (let ((input-no -1))
    (list 'memory-set set-name
          (mapcar #'(lambda (input)
                     (cm-load-memory-input set-name
                                             (incf input-no)
                                             input))
              training-set))))

; CM-UNLOAD-TRAINING-SET unloads TRAINING-SET from the CONTINUOUS-MAPPING or
; ASSOCIATIVE-MEMORY net structure on the CM. The pvars in the TRAINING-SET
; array are deallocated and should no longer be accessed.
;
(defun unload-training-set (training-set)
  (let ((type (training-set-type training-set)))
    (map nil
         #'(lambda (exemplar)
             (*deallocate (exemplar-input-pvar exemplar))
             (if (eq type 'mapping-set)
                 (cm-load-memory-input set-name
                                         (incf input-no)
                                         input))))
         training-set)))

```

```

      (*deallocate (exemplar-target-pvar exemplar)))
      (training-set-exemplars training-set)))

; PRINT-TRAINING-SET prints the TRAINING-SET's input/target or
; input vectors for a MAPPING-SET or MEMORY-SET, respectively.
;
(defun print-training-set (training-set)
  (let ((type (training-set-type training-set))
        (format t "~2~a: ~a" type (training-set-name training-set))
        (map nil
              #'(lambda (exemplar)
                  (format t "%i: ") (print-vec (exemplar-input-vec exemplar))
                  (when (eq type 'mapping-set)
                    (format t " t: ") (print-vec (exemplar-target-vec exemplar))))
              (training-set-exemplars training-set))))

;;; EOF

#+:ccl
(format t "%\\"Training Sets\\" loaded")

```

```

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

```

```

(in-package '*lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Nettalk Implementation
;;;
;;; Net Learning
;;;

; DEBUG-LEARNING sets toggles the :NET-DEBUG flag in the features list.
;
(defun debug-learning (&optional (debug-on t))
  (if debug-on
      (pushnew :net-debug *features*)
      (setf *features* (delete :net-debug *features*))))

; Scalar LOGISTIC function
;
(defun logistic (x)
  (/ (1+ (exp (- x)))))

; Parallel LOGISTIC function
;
(defmacro logistic!! (x!!)
  '(!! (the single-float-pvar (1!! (exp!! (-!! ,x!!))))))

; Scalar LOGISTIC derivative
;
(defun dLogistic (x)
  (let ((logistic (logistic x)))
    (* logistic (- 1 logistic))))

; Parallel LOGISTIC derivative
;
(defmacro dLogistic!! (x!!)
  (let ((logistic!! (gensym)))

```

```

(*let ((,logistic!! (logistic!! ,x!)))
  (declare (type float-pvar ,logistic!))
  (*!! ,logistic!! (-!! (! 1) ,logistic!)))
)

; *NORM of pvar
;
(defmacro *norm (x!!)
  '(sqrt (*sum (*!! ,x!! ,x!)))
)

;
; Feed-forward
;

; FEED-FORWARD computes a single feed-forward cycle of NET with the given INPUT!!.
; If NET is an ASSOCIATIVE-MEMORY net and LATCHED-P is T, then FEED-FORWARD
; operates on the master network rather than the slave network.
;
(defun feed-forward (net input!! &key latched-p)
  (*all
    (*when netp!!
      (*when unitp!!
        (*set I!! (! 0.0)
          X!! (! 0.5)
          dX!! (! 0.5))
        (*when inputp!!
          (if (memory-netp net)
              (*set X!! (the float-pvar input!))
              (*set I!! (the float-pvar input!))))
          (*set Z!! X!))
        (*set Z!! (scan!! Z!! 'copy!! :segment-pvar forward-fan-out-seg!)))
      (do ()
        ((*when unitp!! (*and (<!! (abs!! dX!!) epsilon-x!))))
        (dotimes (i (net-x-iterations net))
          #+:net-debug
          (progn
            (format-pvars (U!! LogU!! dX!! X!! Z!))
            (format t "%Hit any key to continue: ") (read-char))
            (*when fan-outp!!
              (*set U!! (*!! W!! Z!))
              (*pset :no-collisions U!! U!! to-addr!))
            (*when unitp!!
              (*set U!! (! 0.0)))
            (*set U!! (scan!! U!! '+!! :segment-pvar forward-fan-in-seg!))
            (*when unitp!!
              (*set LogU!! (logistic!! U!))
              dX!! (+!! (*!! a!! (-!! X!)) (*!! b!! LogU!!) I!))
              X!! (+!! X!! dX!))
              Z!! X!))
            (if latched-p
                (*when inputp!!
                  (*set Z!! (the float-pvar input!))))
            (*set Z!! (scan!! Z!! 'copy!! :segment-pvar forward-fan-out-seg!))
          ))
      )))
)

;
; Back-Propagate
;

; BACK-PROPAGATE computes a single back-propagation cycle of NET with the given TARGET!!.
;
(defun back-propagate (net target!!)
  (*all
    (*when netp!!
      (*when unitp!!

```

```

(*if outputp!!
  (*set J!! (-!! (the float-pvar target!!) X!!!)
    (*set J!! (!! 0.0)))
(*when (or!! unitp!! fan-outp!!)
  (*set Y!! (!! 0.0))
  (*set dY!! (!! 0.5)))

(do ()
  ((*when unitp!! (*and (<!! (abs!! dY!!) epsilon-y!!!))))

(dotimes (i (net-y-iterations net))
  #+:net-debug
  (progn
    (format-pvars (LogU!! V!! dY!! Y!!!)
      (format t "~%Hit any key to continue: ") (read-char))

    (*when fan-outp!!
      (*set V!! (*!! W!! Y!!!))
    (*when unitp!!
      (*set V!! (!! 0.0)))
    (*set V!! (scan!! V!! '+!!
      :segment-pvar backward-fan-out-seg!!
      :direction :backward))

    (*when unitp!!
      (if (memory-netp net)
        (*when inputp!!
          (*set V!! (!! 0.0)))
        (*set dY!! (+!! (*!! a!! (-!! Y!!!)
          (*!! b!! LogU!! (-!! (!! 1.0) LogU!!)
          (+!! V!! J!!!))
          Y!! (+!! Y!! dY!!!)))
        (*set Y!! (scan!! Y!! 'copy!!
          :direction :backward
          :segment-pvar backward-fan-in-seg!!!))

      #+:*lisp-simulator
      (progn (*when fan-inp!!
        (*pset :no-collisions Y!! V!! from-addr!!!)
        (*when fan-outp!!
          (*set Y!! V!!!))
        #+:*lisp-hardware
        (*when fan-inp!!
          (*pset :no-collisions Y!! Y!! from-addr!!!)
        ))
      )))

;
; Gradient Update
;
; GRADIENT-UPDATE increments the current weight-space gradient.
;
(defun gradient-update ()
  (*when fan-outp!!
    (*set dW!!
      (+!! dW!! (*!! Y!! Z!!!))))

;
; Weight Update
;
; WEIGHT-UPDATE updates the connection weights using the current and last gradients.
;
(defun weight-update ()
  (*when fan-outp!!
    (*set W!! (+!! W!!
      (*!! eta!! dW!!!)
      (*!! alpha!! dWold!!!)
      dWold!! dW!!!)))

```

```

;
; Net Training
;
; STEEPEST-DESCENT performs a true steepest-descent adjustment of the connection weights
; for the input/target pairs in TRAINING-SET. STEEPEST-DESCENT returns T or NIL
; indicating if TRAINING-SET has been learned within the weight update criterion and
; the current target error. If provided, the PRINT-NET-IO function is called to report
; the net's input and output.
;
(defun steepest-descent (net training-set &key print-net-io)
  (let ((learned-p t)
        (target-error 0.0))
    (*all
     (*when fan-outp!!
      (*set dW!! (!! 0.0)))

     (dolist (exemplar (training-set-exemplars training-set))

      (feed-forward net (exemplar-input-pvar exemplar) :latched-p (memory-netp net))
      (back-propagate net (exemplar-target-pvar exemplar))
      (*when outputp!!
       (setf learned-p
              (and learned-p
                   (*and (<!! (abs!! J!!) epsilon-w!))))
       (incf target-error (*norm J!)))

      (if print-net-io
          (funcall print-net-io (exemplar-input-vec exemplar) (get-net-output net)))

      (gradient-update))

     (weight-update))

    (values learned-p target-error))
  )

; TRAIN-NET trains NET using the given TRAINING-SET. If specified, PRINT-TRAINING-SET
; is called to print the current TRAINING-SET. In addition, PRINT-NET-IO may be used
; to report the net's input and output each PRINT-INTERVAL iterations.
;
(defun train-net (net training-set &key print-training-set print-interval print-net-io)
  (format t "~2%Net Training~%" )
  (print-net net t nil :verbose-p t)
  (if print-training-set
      (funcall print-training-set training-set))

  (*all
   (*when fan-outp!!
    (*set dWold!! (!! 0.0)))

   (do ((iteration 0 (1+ iteration))
        (learned-p nil)
        target-error
        (print-net-io-p print-interval
         (and print-interval
              (zerop (mod (1+ iteration) print-interval))))))

    ((or learned-p
         (and (net-max-updates net)
              (= iteration (net-max-updates net))))
     (format t "~2%Training set ~:[not ~;~]learned after ~a iteration~:*~[s~;~;~].~%"
             learned-p iteration)
     (when (and print-interval print-net-io)
       (map nil #'(lambda (exemplar)
                    (feed-forward net (exemplar-input-pvar exemplar))
                    (funcall print-net-io
                             (exemplar-input-vec exemplar)
                             (get-net-output net))))

```

```

      (training-set-exemplars training-set))
      (format t "~%Error = ~a" target-error)))

  (if print-net-io-p
      (format t "~2%Iteration ~a" iteration))

  (multiple-value-setf
   (learned-p target-error)
   (steepest-descent net
    training-set
    :print-net-io (if print-net-io-p print-net-io)))

  (if print-net-io-p
      (format t "~%Error = ~a" target-error))
  )

(values))

;;; EOF

#+:ccl
(format t "~%\\"Net Learning\\" loaded")

```

```

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

```

```

(in-package 'lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Nettalk Implementation
;;;
;;; OR Test Nets
;;;

;
; IOR Continuous Mapping Net
;

(def-mapping-net or-mapping-net
  :slabs '(2 1 1)
  :input-slab-no 0
  :output-slab-no 2
  :bundles '((1 0 100)
             (2 0 100)
             (2 1 100)
             (1 3 100)
             (2 3 100)
             (3 3 100)
             )
  )

(defvar *ior-mapping-pairs*)
(setf *ior-mapping-pairs*
      (list-to-array-pairs '((0.0 0.0) (0.0))
                           ((0.0 1.0) (1.0))
                           ((1.0 0.0) (1.0))
                           ((1.0 1.0) (1.0))))

(defvar *ior-mapping-set*)

```

```

(setf *ior-mapping-set*
  (cm-load-mapping-set 'ior-mapping-set *ior-mapping-pairs*))

(train-net or-mapping-net
  *ior-mapping-set*
  :print-training-set #'print-training-set
  :print-interval 10
  :print-net-io #'print-io-vecs)

;
; XOR Associative Memory Net
;

(def-mem-net or-memory-net
  :slabs '(3 1)
  :input-slab-no 0
  :bundles '((0 0 100)
             (0 1 100)
             (1 1 100)
             )
  :epsilon-w 0.05
  )

(defvar *xor-memory-list*)
(setf *xor-memory-list*
  (list-to-array '((0.0 0.0 0.0)
                  (0.0 1.0 1.0)
                  (1.0 0.0 1.0)
                  (1.0 1.0 0.0))))

(defvar *xor-memory-set*)
(setf *xor-memory-set*
  (cm-load-memory-set 'xor-memory-set *xor-memory-list*))

(train-net or-memory-net
  *xor-memory-set*
  :print-training-set #'print-training-set
  :print-interval 20
  :print-net-io #'print-io-vecs)

;;; EOF

#+:ccl
(format t "~%" "Test Nets\" loaded")


```

```

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

(in-package 'lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Nettek Implementation
;;;
;;;
;;; Time Nets
;;;

; CM-TIME-AND-PRINT times the execution of FORM and reports the timing statistics.
;

```

```

(defmacro cm-time-and-print (form)
  (let ((elapsed-time (gensym))
        (cm-time (gensym))
        (percent (gensym)))
    '(multiple-value-bind (,elapsed-time ,cm-time ,percent)
      (cm:time ,form :return-statistics-only-p t)
      (print-cm-timing ',(if (listp form) (first form) form)
                       ,elapsed-time
                       ,cm-time
                       ,percent))))

; PRINT-CM-TIMING prints the FE ELAPSED-TIME, CM-TIME and CM usage PERCENT
; statistics for the given OPERATION.
;
(defun print-cm-timing (operation elapsed-time cm-time percent)
  (format t "~%~a: ~7,3f secs elapsed time, ~7,3f secs CM time (~4,1f%)"
          operation elapsed-time cm-time percent))

;
; Mapping Net Timings
;
; TIME-MAPPING-NET compiles timing statistics for the CONTINUOUS-MAPPING test net
; up to MAX-N. INCREMENT controls granularity of the increment in net size.
;
(defun time-mapping-net (max-n &key (increment 1))
  (cmi::calibrate-cm-timer)
  (let (mapping-net
        mapping-set)
    (do ((n 1 (+ increment n)))
        ((> n max-n))
      (setf mapping-net
            (fe-make-net 'mapping-net
                        :continuous-mapping
                        (list (* 4 n) (* 2 n) n 1)
                        0
                        2
                        '((1 0 100)
                          (2 0 100)
                          (2 1 100)
                          (1 3 100)
                          (2 3 100)
                          (3 3 100)
                          )
                        ))
      (format t "~3%*** N = ~a ***" n)
      (format t "%Mapping net: ~a units, ~a connections -> ~a processors"
              (net-no-units mapping-net)
              (net-no-connections mapping-net)
              (net-no-processors mapping-net))
      (cm-net-cold-boot mapping-net)
      (format t "%VP ratio = ~a" cm:*virtual-to-physical-processor-ratio*)
      (cm-time-and-print
       (cm-make-net mapping-net))
      (setf mapping-set
            (cm-load-mapping-set 'mapping-set
                                (list
                                 (list (make-array (* 4 n) :initial-element 1.0)
                                       (make-array n :initial-element 1.0))))
            ))
      (cm-time-and-print
       (feed-forward mapping-net (exemplar-input-pvar (get-exemplar mapping-set 0))))
      (cm-time-and-print
       (back-propagate mapping-net (exemplar-target-pvar (get-exemplar mapping-set 0))))

      (if (and (= n 1) (/= increment 1)) (decf n))
      )))

; TIME-MEMORY-NET compiles timing statistics for the ASSOCIATIVE-MEMORY test net
; up to MAX-N. INCREMENT controls granularity of the increment in net size.

```

```

;
(defun time-memory-net (max-n &key (increment 1))
  (cmi::calibrate-cm-timer)
  (let (memory-net
        memory-set)
    (do ((n 1 (+ increment n)))
        ((> n max-n))
      (setf memory-net
            (fe-make-net 'memory-net
                        :associative-memory
                        (list (* 8 n) n)
                        0
                        0
                        '( (1 0 25)
                          (0 1 25)
                        )
                        ))
      (format t "~3%*** N = ~a ***" n)
      (format t "~%Memory net: ~a units, ~a connections -> ~a processors"
              (net-no-units memory-net)
              (net-no-connections memory-net)
              (net-no-processors memory-net))
      (cm-net-cold-boot memory-net)
      (format t "%VP ratio = ~a" cm:*virtual-to-physical-processor-ratio*)
      (cm-time-and-print
        (cm-make-net memory-net))
      (setf memory-set
            (cm-load-memory-set 'memory-set
                                (list (make-array (* 8 n) :initial-element 1.0))))
      (cm-time-and-print
        (feed-forward memory-net (exemplar-input-pvar (get-exemplar memory-set 0))))
      (cm-time-and-print
        (back-propagate memory-net (exemplar-input-pvar (get-exemplar memory-set 0))))

      (if (and (= n 1) (/= increment 1)) (decf n))
    )))

;;; EOF

#+:ccl
(format t "~%\\"Time Nets\\" loaded")

```


Appendix D

*LISP CODE FOR TOMBOULIAN IMPLEMENTATION

```

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

(in-package '*lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Tomboulian Implementation
;;;
;;; Pvar Types
;;;
;;;   MAX-INT-PVAR           unsigned byte  [0 , 216-1]
;;;   CUBE-ADDRESS-PVAR     unsigned byte  [0 , *log-number-of-processors-limit*]
;;;   SLAB-NO-PVAR          unsigned byte  [0 , 216-1]
;;;   UNIT-NO-PVAR         unsigned byte  [0 , 216-1]
;;;   BUNDLE-NO-PVAR       unsigned byte  [0 , 216-1]
;;;   CONNECTION-NO-PVAR   unsigned byte  [0 , 232-1]
;;;

; Pvar type field sizes
;
(defvar *cm-max-int*
(defvar *slab-no-size*
(defvar *unit-no-size*
(defvar *bundle-no-size*
(defvar *connection-no-size*

(defvar *cm-single-float-size* 32)           ; IEEE single-float size, WEITIEK chips

; Set field sizes in compiler's environment
;
(eval-when (compile load eval)
  (setf *cm-max-int* (expt 2 16))
  (setf *slab-no-size* 16)
  (setf *unit-no-size* 16)
  (setf *bundle-no-size* 16)
  (setf *connection-no-size* 32)
  )

; Define pvar types
;
(deftype max-int-pvar () '(pvar (unsigned-byte
                                #(1+ (ceiling (log *cm-max-int* 2))))))

(deftype cube-address-pvar () '(pvar (unsigned-byte
                                       *log-number-of-processors-limit*)))

(deftype slab-no-pvar () '(pvar (unsigned-byte #.*slab-no-size*)))

(deftype unit-no-pvar () '(pvar (unsigned-byte #.*unit-no-size*)))

```

```

(deftype bundle-no-pvar () '(pvar (unsigned-byte #.*bundle-no-size*)))

(deftype connection-no-pvar () '(pvar (unsigned-byte #.*connection-no-size*)))

; When using simulator, add new pvar type symbols
;
#+:lisp-simulator
(progn
  (pushnew 'max-int-pvar **lisp-exported-type-symbols*)
  (pushnew 'cube-address-pvar **lisp-exported-type-symbols*)
  (pushnew 'slab-no-pvar **lisp-exported-type-symbols*)
  (pushnew 'unit-no-pvar **lisp-exported-type-symbols*)
  (pushnew 'bundle-no-pvar **lisp-exported-type-symbols*)
  (pushnew 'connection-no-pvar **lisp-exported-type-symbols*)
)

;;; EOF

#+:ccl
(format t "~%\\"Pvar Types\\" loaded")

```

```

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

(in-package 'lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Tombouliau Implementation
;;;
;;; Utilities
;;;

; COUNT-CSS returns the number of processors in the currently selected set.
;
(defun count-css ()
  (*sum (! 1)))

; MAX-INT!! returns a field pvar containing *CM-MAX-INT*.
;
(defmacro max-int!! ()
  '(the max-int-pvar (! *cm-max-int*)))

; RANDOM-FLOAT!! returns a random float pvar evenly distributed in the interval
; [mean-interval , mean+interval] in each processor.
;
#+:lisp-simulator
(*proclaim '(ftype (function (t) (pvar single-float)) random-float!!))

(defun random-float!! (mean!! interval!!)
  (declare (type float-pvar mean!! interval!!))
  (*let (temp!!)
    (declare (type float-pvar temp!!))
    (*set temp!!
      (+!! mean!!
        (*!! interval!!
          (if!! (=!! (random!! (! 2)) (! 1))
            (! 1.0)
            (! -1.0))
        )
      )
    )
  )

```

```

      (/!! (random!! (max-int!!))
         (max-int!!))))
temp!))

; FORMAT-PVARS pretty prints the given list of PVARs. Additional keyword arguments
; will be passed in to PRETTY-PRINT-PVAR.
;
(defun format-pvars (pvars &rest keys &key &allow-other-keys)
  '(progn
    ,@(mapcan #'(lambda (pvar)
      ((format t "%~a" (pvar-name ,pvar)) (ppp ,pvar ,@keys)))
      pvars)))

; ENUMERATE returns the list (0,1,..N).
;
(defun enumerate (n)
  (let (l)
    (dotimes (i n)
      (push i l))
    (nreverse l)))

; MULTIPLE-VALUE-SETF sets the locations referenced by ACCESSOR-FORMS
; to the multiple values returned by VALUES-FORM.
;
(defun multiple-value-setf (accessor-forms values-form)
  (let ((values-list (gensym))
        (i -1))
    '(let ((,values-list (multiple-value-list ,values-form)))
      (setf ,@(mapcan #'(lambda (accessor-form)
        (,(accessor-form (nth ,(incf i) ,values-list)))
        accessor-forms))))))

; PRINT-VEC prints the array VEC on STREAM with the given ELEMENTS-PER-LINE.
; Each element is printed using ELEMENT-FORMAT, and each line of output is
; preceded by NEW-LINE-FORMAT.
;
(defun print-vec (vec &optional (stream t)
                 &key elements-per-line (element-format "~s ")
                 (new-line-format "~%"))
  (dotimes (i (length vec))
    (if (and elements-per-line
              (zerop (mod i elements-per-line)))
        (format stream "~@?" new-line-format))
        (format stream "~@?" element-format (aref vec i))))
  (values))

; PRINT-IO-VECS prints the INPUT and OUTPUT vectors.
;
(defun print-io-vecs (input output)
  (format t "%i: " (print-vec input))
  (format t " o: " (print-vec output)))

; LIST-TO-ARRAY-PAIRS coerces the list of LIST-PAIRS into a list of array pairs.
;
(defun list-to-array-pairs (list-pairs)
  (let (input target)
    (map 'list
      #'(lambda (pair)
        (setf input (first pair)
              target (second pair))
        (list
          (make-array (length input) :initial-contents input)
          (make-array (length target) :initial-contents target))))
      list-pairs)))

; LIST-TO-ARRAY coerces the list of lists into a list of arrays.
;
(defun list-to-array (list)
  (map 'list
    #'(lambda (sub-list)

```

```

      (make-array (length sub-list) :initial-contents sub-list))
    list))

;;; EOF

#+:ccl
(format t "~%" "Utilities\" loaded")

|-----|
|-----|

;;; -- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 --

(in-package '*lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Tombouliau Implementation
;;;
;;; N-Dimensional Grid
;;;

; Grid parameters
;
(defvar *max-hops*)           ; max hops in grid

(defvar *max-hops-size*)     ; max hops field size

(defvar *grid-dim-size*)     ; grid dimension field size

(defvar *grid-center*)      ; addr of grid center

(defvar *max-hops-to-center*) ; max hops to grid center

; Define grid pvar types
;
;   GRID-DIMENSION-PVAR      [0,max dimension)
;   GRID-OFFSET-PVAR        -1,0,+1
;   GRID-DISTANCE-PVAR      [0,max hops]
;
(deftype grid-dimension-pvar () '(pvar (unsigned-byte *grid-dim-size*)))

(deftype grid-offset-pvar () '(pvar (signed-byte 2)))

(deftype grid-distance-pvar () '(pvar (unsigned-byte *max-hops-size*)))

; When using simulator, add new pvar type symbols
;
#+:lisp-simulator
(progn
  (pushnew 'grid-dimension-pvar **lisp-exported-type-symbols*)
  (pushnew 'grid-offset-pvar **lisp-exported-type-symbols*)
  (pushnew 'grid-distance-pvar **lisp-exported-type-symbols*)
  )

; MAX-HOPS returns the maximum number of hops for the given GRID-DIMENSIONS.
;
(defun max-hops (grid-dimensions)
  (- (reduce #'* grid-dimensions)
     (length grid-dimensions)))

; GRID-CENTER returns the center of GRID-DIMENSIONS.

```

```

;
(defun grid-center (grid-dimensions)
  (mapcar #'(lambda (n)
              (truncate n 2))
          grid-dimensions))

; DIMENSION-NO!! returns a grid-dimension-pvar containing DIM number.
;
(defmacro dimension-no!! (dim)
  '(the grid-dimension-pvar (!! ,dim)))

; GRID-CENTER!! returns a grid-distance-pvar containing the grid center along DIM.
;
(defmacro grid-center!! (dim)
  '(the grid-distance-pvar (!! (nth ,dim *grid-center*)))

; INIT-N-DIMENSIONAL-GRID initializes the n-dimensional-grid parameters.
;
(defun init-n-dimensional-grid ()
  (setf *max-hops* (max-hops *current-cm-configuration*)
        *max-hops-size* (ceiling (log *max-hops* 2))
        *grid-dim-size* (ceiling (log (apply #'max *current-cm-configuration*) 2))
        *grid-center* (grid-center *current-cm-configuration*)
        *max-hops-to-center* (reduce #'+ *grid-center*)
        *ppp-default-mode* :grid))

; Reset N-dimensional-grid parameters after *COLD-BOOT.
;
#*:lisp-hardware
(add-initialization "Init N Dimensional Grid"
  '(init-n-dimensional-grid)
  '*after-*cold-boot-initializations*)

#*:lisp-simulator
(add-initialization :name-of-form "Init N Dimensional Grid"
  :form '(init-n-dimensional-grid)
  :variable '*after-*cold-boot-initializations*)

; *COLD-BOOT CM to current grid dimensions.
;
(*cold-boot)

;;; EOF

#*:ccl
(format t "~%\\"N Dimensional Grid\\" loaded")

```

```

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

```

```

(in-package '*lisp)

```

```

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

```

```

;;;
;;; Tomboulian Implementation
;;;
;;; Processor Allocation
;;;
;

```

```

; CM Dimensions
;
; *CM-DIMENSIONS* is a list of CM configurations. Each configuration is a list
; of the total number of processors and the corresponding CM dimensions.
;
(defvar *cm-dimensions*
  (mapcar #'(lambda (dims)
            (list (reduce #'* dims) dims))
          #+*lisp-hardware
          '(( 64 128)
            ( 128 128)
            ( 128 256)
            ( 256 256)
            ( 256 512)
            ( 512 512)
            ( 512 1024)
            (1024 1024))
          #+*lisp-simulator
          '((4 4)
            (6 4)
            (6 6)
            (8 6)
            (8 8))
          ))
; CM-BEST-FIT-DIMS returns the minimum CM dimensions necessary
; to satisfy the request for NO-PROCESSORS.
;
(defun cm-best-fit-dims (no-processors)
  (second (assoc no-processors
                *cm-dimensions*
                :test #'<=)))
;
; Processor Allocator
;
(*proclaim '(type boolean-pvar freep!!))

(defvar freep!! t!!) ; is processor free?

(defvar distance-from-grid-center!!) ; processor distance from grid center,
; allocated during *COLD-BOOT

; DISTANCE-FROM-GRID-CENTER returns a grid-distance-pvar containing
; each processor's distance from the grid center.
;
#-*lisp-simulator
(*proclaim '(ftype (function (t) grid-distance-pvar) distance-from-grid-center!!))

(defun distance-from-grid-center!! ()
  (*let ((distance!! (!! 0))
        (declare (type grid-distance-pvar distance!!))
        (dotimes (n *number-of-dimensions*)
          (*set distance!!
                (+!! distance!!
                    (abs!! (-!! (self-address-grid!! (the grid-dimension-pvar (!! n)))
                                (grid-center!! n)))))))
    distance!!))

; INIT-PROCESSOR-ALLOCATION initializes the processor allocator.
;
(defun init-processor-allocation ()
  (setf distance-from-grid-center!!
        (allocate!! (distance-from-grid-center!!)
                    'distance-from-grid-center!!
                    'grid-distance-pvar))
  )

```

```

; Reset the processor allocator after *COLD-BOOT.
;
#+:*lisp-hardware
(add-initialization "Init Processor Allocation"
  '(init-processor-allocation)
  '*after-*cold-boot-initializations*)

#+:*lisp-simulator
(add-initialization :name-of-form "Init Processor Allocation"
  :form '(init-processor-allocation)
  :variable '*after-*cold-boot-initializations*)

;
; Processor Allocation Modes
;
;   cube address
;   random
;   grid address
;   distance from grid center
;   weighted distance from grid center
;

; PALLOC-BY-CUBE-ADDR!! returns a cube address ordering of the free processors.
;
#-:*lisp-simulator
(*proclaim '(ftype (function (t) cube-address-pvar) palloc-by-cube-addr!!))

(*defun palloc-by-cube-addr!! ()
  (enumerate!!))

; PALLOC-RANDOM!! returns a random ordering of the free processors.
;
#-:*lisp-simulator
(*proclaim '(ftype (function (t) cube-address-pvar) palloc-random!!))

(*defun palloc-random!! ()
  (rank!! (random!! (max-int!!))
    '<=!!))

; PALLOC-BY-GRID-ADDR!! returns a grid address ordering of the free processors.
;
#-:*lisp-simulator
(*proclaim '(ftype (function (t) cube-address-pvar) palloc-by-grid-addr!!))

(*defun palloc-by-grid-addr!! ()
  (*let ((addr!! (self-address-grid!! (dimension-no!! (1- *number-of-dimensions*))))
    (declare (type cube-address-pvar addr!!))
    (do ((dim-no (- *number-of-dimensions* 2) (1- dim-no))
        (dim-sizes (rest (reverse *current-cm-configuration*))
          (rest dim-sizes)))
      ((minusp dim-no))
      (*set addr!!
        (+!! (*!! addr!!
          (the cube-address-pvar
            (!! (expt 2 (ceiling (log (first dim-sizes) 2))))))
        (self-address-grid!! (dimension-no!! dim-no))))))
    (rank!! addr!!
      '<=!!)))

; PALLOC-FROM-GRID-CENTER!! returns an ordering of the free processors
; by increasing distance from the grid center.
;
#-:*lisp-simulator
(*proclaim '(ftype (function (t) cube-address-pvar) palloc-from-grid-center!!))

(*defun palloc-from-grid-center!! ()
  (rank!! distance-from-grid-center!! '<=!!))

```

```

; PALLOC-FROM-GRID-CENTER-WEIGHTED!! returns a random ordering of the free processors
; weighted by increasing distance from the grid center.
;
#-:*lisp-simulator
(*proclaim '(ftype (function (t) cube-address-pvar) palloc-from-grid-center-weighted!!))

(defun palloc-from-grid-center-weighted!! ()
  (rank!! (*!! distance-from-grid-center!!
           (random!!
            (?! (truncate *cm-max-int*
                          *max-hops-to-center*))))
          '<=!!))

; Legal allocation modes and associated ordering functions
;
(defvar *palloc-legal-allocation-modes*
  '(:cube-addr . palloc-by-cube-addr!!)
    (:random . palloc-random!!)
    (:grid-addr . palloc-by-grid-addr!!)
    (:grid-center . palloc-from-grid-center!!)
    (:grid-center-weighted . palloc-from-grid-center-weighted!!))

;
; Processor Allocator
;

; OUT-OF-PROC-P signals an error if there are fewer than N free processors.
;
(defun out-of-proc-p (n)
  (if (< (*when freep!! (count-css)) n)
      (error "PALLOC!! can't allocate ~a processor~:*~[s~;~::~s~]>" n)))

; GET-ALLOCATION-FUNCTION returns the ordering function for ALLOCATION-MODE.
; If ALLOCATION-MODE is illegal, GET-ALLOCATION-FUNCTION asserts an error.
;
(defun get-allocation-function (allocation-mode)
  (assert (assoc allocation-mode *palloc-legal-allocation-modes*)
          (allocation-mode)
          "Unknown allocation mode ~s" allocation-mode)
  (rest (assoc allocation-mode *palloc-legal-allocation-modes*)))

; FOR-FIRST-N-PROC executes BODY with the currently selected set
; composed of the first N processors by cube address.
;
(defmacro for-first-n-proc ((n) &body body)
  '(*when (<!! (self-address!!) (the cube-address-pvar (?! ,n)))
    ,@body))

; In Allegro CL, set FRED indentation for macro
;
#+:ccl
(pushnew '(for-first-n-proc . 1)
        ccl::*fred-special-indent-alist*
        :test #'equal)

; PALLOC!! returns a cube-address-pvar containing the cube addresses of N
; free processors allocated according to ALLOCATION-MODE.
;
#-:*lisp-simulator
(*proclaim '(ftype (function (t) cube-address-pvar) palloc!!))

(defun palloc!! (n &optional (allocation-mode :cube-addr))
  (let ((allocation-function
        (get-allocation-function allocation-mode)))
    (*all
     (unless (out-of-proc-p n)
       (*let (rendezvous!!)
          (declare (type cube-address-pvar rendezvous!!))
          (*if freep!!
```

```

      (*pset :no-collisions
        (self-address!!)
        rendezvous!!)
      (*funcall allocation-function))
(for-first-n-proc (n)
  (*pset :no-collisions
    nil!!
    freep!!
    rendezvous!!)
  (sort!! rendezvous!! '<=!!))))))

; PALLOC returns an array containing the cube addresses of N free processors
; allocated according to ALLOCATION-MODE.
;
(defun palloc (n &optional (allocation-mode :cube-addr))
  (pvar-to-array (palloc!! n allocation-mode)
    (make-array n)
    :cube-address-end n))

; PALLOC-ONE returns the cube address of the next free processor
; allocated according to ALLOCATION-MODE.
;
(defmacro palloc-one (&optional (allocation-mode :cube-addr))
  '(aref (palloc 1 ,allocation-mode) 0))

; WITH-N-PROC-ALLOCATED allocates N free processors according to ALLOCATION-MODE and
; sets ADDR to an array containing the cube addresses of the allocated processors.
; WITH-N-PROC-ALLOCATED then executes BODY with the currently selected set
; composed of the newly allocated processors.
;
(defmacro with-n-proc-allocated ((n addr
                                &optional (allocation-mode :cube-addr))
  &body body)
  (let ((p-addr!! (gensym))
        (new-proc-p!! (gensym)))
    '(let (,addr)
      (*all
        (*let ((,p-addr!! (palloc!! ,n ,allocation-mode))
              (,new-proc-p!! nil!!))
          (declare (type cube-address-pvar ,p-addr!!)
                    (type boolean-pvar ,new-proc-p!!))
          (setf ,addr (pvar-to-array ,p-addr!!
                                     (make-array ,n)
                                     :cube-address-end ,n))
          (*if (<!! (self-address!!) (the cube-address-pvar (!! ,n)))
              (*pset :no-collisions
                    t!!
                    ,new-proc-p!!
                    ,p-addr!!))
            (*when ,new-proc-p!!
              ,@body))))))

; In Allegro CL, set FRED indentation for macro
;
#+:ccl
(pushnew '(with-n-proc-allocated . 1)
  ccl::*fred-special-indent-alist*
  :test #'equal)

;;; EOF

#+:ccl
(format t "~%" "Processor Allocation" loaded")

```

```

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

(in-package '*lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Tombouliau Implementation
;;;
;;; Neighbor Addressing
;;;

; Neighbor Addressing parameters
;
(defvar *self-loops-p* t) ; allow self-neighbors?

(defvar *neighbor-limit*) ; max number of neighbors

(defvar *neighbor-no-size*) ; max neighbors field size

; Define neighbor addressing pvar type
;
; NEIGHBOR-NO-PVAR [0,max neighbors)
;
(eval-when (compile load eval)
  (deftype neighbor-no-pvar () '(pvar (unsigned-byte *neighbor-no-size*)))
  (deftype neighbor-no () '(unsigned-byte *neighbor-no-size*'))
  )

; When using simulator, add new pvar type symbols
;
#+:lisp-simulator
(pushnew 'neighbor-no-pvar **lisp-exported-type-symbols*)

; NEIGHBOR-LIMIT!! returns a neighbor-no-pvar containing neighbor limit.
;
(defmacro neighbor-limit!! ()
  '(the neighbor-no-pvar (!! *neighbor-limit*)))

; NO-NEIGHBORS returns the neighbor limit for a grid with NO-DIMENSIONS.
; If SELF-LOOPS-P is T, then self-neighbors are allowed.
;
(defun no-neighbors (no-dimensions self-loops-p)
  (+ (* 2 no-dimensions)
     (if self-loops-p 1 0)))

; SELF-NEIGHBOR-P returns T if NEIGHBOR-NO represents the self-neighbor.
;
(defun self-neighbor-p (neighbor-no)
  (and *self-loops-p*
       (= neighbor-no (1- *neighbor-limit*))))

; INIT-NEIGHBOR-ADDRESSING initializes the neighbor addressing parameters.
;
(defun init-neighbor-addressing ()
  (setf *neighbor-limit*
        (no-neighbors *number-of-dimensions* *self-loops-p*)
        *neighbor-no-size* (ceiling (log *neighbor-limit* 2))))

```

```

; Reset neighbor addressing parameters after *COLD-BOOT.
;
#+:*lisp-hardware
(add-initialization "Init Neighbor Addressing"
  '(init-neighbor-addressing)
  '*after-*cold-boot-initializations*)

#+:*lisp-simulator
(add-initialization :name-of-form "Init Neighbor Addressing"
  :form '(init-neighbor-addressing)
  :variable '*after-*cold-boot-initializations*)

; *COLD-BOOT CM to current grid dimensions.
;
(*cold-boot)

; Neighbor Addressing Utilities

; NEIGHBOR-NO-INVERSE returns the inverse link for NEIGHBOR-NO.
;
(defun neighbor-no-inverse (neighbor-no)
  (if (and *self-loops-p* (= neighbor-no (1- *neighbor-limit*)))
      neighbor-no
      (if (evenp neighbor-no)
          (1+ neighbor-no)
          (1- neighbor-no))))

; Parallel NEIGHBOR-NO-INVERSE
;
#+:*lisp-hardware
(*proclaim '(ftype (function (t) neighbor-no-pvar) neighbor-no-inverse!))

(*defun neighbor-no-inverse!! (neighbor-no!!)
  (declare (type neighbor-no-pvar neighbor-no!!))
  (*let (inverse!!)
    (declare (type neighbor-no-pvar inverse!!))
    (*set inverse!!
      (if!! (=?!! neighbor-no!! (1-!! (neighbor-limit!!)))
          neighbor-no!!
          (if!! (evenp!! neighbor-no!!)
              (1+!! neighbor-no!!)
              (1-!! neighbor-no!!))))
    inverse!!))

; GRID-OFFSET-FROM-NEIGHBOR-NO returns the grid-offset for NEIGHBOR-NO along DIMENSION.
;
(defun grid-offset-from-neighbor-no (neighbor-no dimension)
  (if (/= (truncate neighbor-no 2) dimension)
      0
      (if (evenp neighbor-no) -1 +1)))

; GRID-OFFSETS-FROM-NEIGHBOR-NO returns the grid-offsets for NEIGHBOR-NO.
;
(defun grid-offsets-from-neighbor-no (neighbor-no)
  (mapcar #'(lambda (dim)
    (grid-offset-from-neighbor-no neighbor-no dim))
    (enumerate *number-of-dimensions*)))

; Parallel GRID-OFFSET-FROM-NEIGHBOR-NO
;
#+:*lisp-hardware
(*proclaim '(ftype (function (t) grid-offset-pvar) grid-offset-from-neighbor-no!))

(*defun grid-offset-from-neighbor-no!! (neighbor-no!! dimension!!)
  (declare (type neighbor-no-pvar neighbor-no!!))

```

```

      (type grid-dimension-pvar dimension!!))
    (cond!! ((/=!! (truncate!! neighbor-no!! (!! 2)) dimension!!) dimension!!)
      (evenp!! neighbor-no!!) (!! -1))
    (t!! (!! +1)))

; CUBE-FROM-NEIGHBOR-NO returns the cube address of the processor connected
; by the NEIGHBOR-NO link to the processor at CUBE-ADDR.
;
(defmacro cube-from-neighbor-no (cube-addr neighbor-no)
  '(cube-from-grid-address
    ,@(mapcar #'(lambda (d)
      '(+ (grid-from-cube-address ,cube-addr ,d)
        (grid-offset-from-neighbor-no ,neighbor-no ,d)))
      (enumerate *number-of-dimensions*))))

; OFF-GRID-NEIGHBOR-P!! returns a boolean-pvar indicating if each processor
; has a NEIGHBOR-NO link.
;
(defmacro off-grid-neighbor-p!! (neighbor-no)
  (let ((off-p!! (gensym)))
    '(the boolean-pvar
      (*let ((,off-p!! nil!!))
        (declare (type boolean-pvar ,off-p!!))
        (cond
          ,@(mapcar #'(lambda (n)
            '( (= ,neighbor-no ,n)
              (*set ,off-p!!
                (off-grid-border-relative-p!!
                  ,@(mapcar #'(lambda (offset)
                    '(!! ,offset))
                    (grid-offsets-from-neighbor-no n))))))
            (enumerate *neighbor-limit*)))
          ,off-p!!))
    ))

; Neighbor Addressing Read Operations

; PREF-1-NEIGHBOR!! reads SOURCE-PVAR into DEST-PVAR from the direction NEIGHBOR-NO.
;
(defmacro pref-1-neighbor!! (dest-pvar source-pvar neighbor-no)
  '(case ,neighbor-no
    ,@(mapcar
      #'(lambda (n)
        '(,n
          (*set ,dest-pvar
            ,(if (self-neighbor-p n)
              source-pvar
              '(news!!
                ,source-pvar
                ,@(mapcar #'(lambda (d)
                  (grid-offset-from-neighbor-no n d))
                  (enumerate *number-of-dimensions*)))
                )))
          ))
      (enumerate *neighbor-limit*)))
  )

; PREF-NEIGHBOR!! reads SOURCE-PVAR into DEST-PVAR from the directions in NEIGHBOR-NO-PVAR.
;
(defmacro pref-neighbor!! (dest-pvar source-pvar neighbor-no-pvar)
  (let* ((temp-neighbor (if (listp neighbor-no-pvar)
    (gensym)
    neighbor-no-pvar))
    *cond-exp)
    (setf *cond-exp

```

```

(*cond
,@(mapcar
  #'(lambda (n)
    '(=!! ,temp-neighbor (!! ,n))
    (*set
      ,dest-pvar
      ,(if
        (self-neighbor-p n)
        source-pvar
        '(news!! ,source-pvar
          ,@(mapcar #'(lambda (d)
            (grid-offset-from-neighbor-no n d))
            (enumerate *number-of-dimensions*))))
      )))
    (enumerate *neighbor-limit*)))
(if (listp neighbor-no-pvar)
  (append
    '(*let ((,temp-neighbor ,neighbor-no-pvar))
      (declare (type neighbor-no-pvar ,temp-neighbor)))
    (list *cond-exp))
  *cond-exp)))

; Neighbor Addressing Write Operations

; *PSET-1-NEIGHBOR writes SOURCE-PVAR into DEST-PVAR according to the direction NEIGHBOR-NO.
;
(defmacro *pset-1-neighbor (source-pvar dest-pvar neighbor-no)
  '(case ,neighbor-no
    ,@(mapcar
      #'(lambda (n)
        '(,n
          ,(if (self-neighbor-p n)
              '(*set ,dest-pvar ,source-pvar)
              '(*pset :no-collisions
                ,source-pvar
                ,dest-pvar
                (cube-from-grid-address!!
                  ,@(mapcar
                    #'(lambda (d)
                      '(+!! (self-address-grid!! (!! ,d))
                        (!! ,(grid-offset-from-neighbor-no n d))))
                    (enumerate *number-of-dimensions*))))
              )))
        (enumerate *neighbor-limit*)))
    ))

; *PSET-NEIGHBOR writes SOURCE-PVAR into DEST-PVAR using COMBINER according
; to the directions in NEIGHBOR-NO-PVAR.
;
(defmacro *pset-neighbor (combiner source-pvar dest-pvar neighbor-no-pvar)
  (let* ((temp-neighbor (if (listp neighbor-no-pvar)
    (gensym)
    neighbor-no-pvar))
    *pset-exp)
    (setf *pset-exp
      '(*pset
        ,combiner
        ,source-pvar
        ,dest-pvar
        (cube-from-grid-address!!
          ,@(mapcar
            #'(lambda (d)
              '(+!! (self-address-grid!! (!! ,d))
                (grid-offset-from-neighbor-no!! ,temp-neighbor (!! ,d))))
            (enumerate *number-of-dimensions*))))
        ))
    (if (listp neighbor-no-pvar)

```

```

(append
  (*let ((,temp-neighbor ,neighbor-no-pvar)
        (declare (the neighbor-no-pvar ,temp-neighbor)))
    (list *pset-exp)
    *pset-exp)
  ))

;;; EOF

#+:ccl
(format t "~%" "Neighbor Addressing" loaded")

|-----|
|-----|

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

(in-package 'lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Tomboulian Implementation
;;;
;;; Graph Structures
;;;

; Graph Structure parameters
;
(defvar *time-quantum*           ; time quantum for routing

(defvar *free*                  ; free slot flag

(defvar *max-path-length*      ; max path length in graph

(defvar *path-len-size*        ; max path length field size

; Define graph structures pvar type
;
;   PATH-LENGTH-PVAR      [0,max path length)
;
(eval-when (compile load eval)
  (deftype path-length-pvar () '(pvar (unsigned-byte *path-len-size*)))
  )

; When using simulator, add new pvar type symbols
;
#+:lisp-simulator
(pushnew 'path-length-pvar **lisp-exported-type-symbols*)

(defvar has-neighbor-p[!!])      ; array of pvars indicating neighbor links

(defvar slots[!!])              ; array of slot structures

(defvar trial-slots[!!])        ; array of trial-slot structures

(*proclaim '(type boolean-pvar activep!!))
(*proclaim '(type boolean-pvar next-activep!!))
(*proclaim '(type boolean-pvar destinationp!!))

```

```

(*proclaim '(type boolean-pvar reachedp!!))

(*defvar activep!!) ; boolean-pvar indicating active processors

; MAKE-HAS-NEIGHBOR-P returns an array of pvars indicating the existence
; of neighbor links in each neighbor direction.
;
(defun make-has-neighbor-p ()
  (let ((has-neighbor-p (make-array *neighbor-limit*)))
    (*all
      (dotimes (n *neighbor-limit*)
        (setf (aref has-neighbor-p n)
              (allocate!! nil!!
                          (format nil "has-neighbor-~a-p" n)
                          'boolean-pvar)))
        (set (the boolean-pvar (aref has-neighbor-p n))
              (not!! (off-grid-neighbor-p!! n))))
      ))
    has-neighbor-p))

; HAS-NEIGHBOR-N-P!! returns the boolean-pvar indicating the existence of link NEIGHBOR-NO.
;
(defmacro has-neighbor-n-p!! (neighbor-no)
  '(the boolean-pvar (aref has-neighbor-p[]!! ,neighbor-no)))

;
; Routing Slots
;
; Arc label stubs
;
(defstruct arc-label
  )

(defun *copy-arc-label (label!! addr label)
  (declare (ignore label!! addr))
  label)

; Routing slot
;
(defstruct (slot
  (:conc-name slot-i-)
  (:constructor make-slot-internal))
  startp!! ; beginning of arc?
  forward!! ; forward link, neighbor-no for read
  backward!! ; backward link, neighbor-no for read
  endp!! ; end of arc?
  arc-label!! ; label if beginning of arc
  )

; Accessors for slot structure
;
(defmacro slot-startp!! (slot)
  '(the boolean-pvar (slot-i-startp!! ,slot)))

(defmacro slot-forward!! (slot)
  '(the neighbor-no-pvar (slot-i-forward!! ,slot)))

(defmacro slot-backward!! (slot)
  '(the neighbor-no-pvar (slot-i-backward!! ,slot)))

(defmacro slot-endp!! (slot)
  '(the boolean-pvar (slot-i-endp!! ,slot)))

(defmacro slot-arc-label!! (slot)

```

```

'(slot-i-arc-label!! ,slot))

; RESET-SLOT resets the SLOT structure.
;
(defun reset-slot (slot)
  '(progn (*set (slot-startp!! ,slot) nil!!
              (slot-forward!! ,slot) (neighbor-limit!!)
              (slot-backward!! ,slot) (neighbor-limit!!)
              (slot-endp!! ,slot) nil!!)
          ,slot))

; MAKE-SLOT returns a new slot structure for TIME step.
;
(defun make-slot (time)
  (let ((slot
        (make-slot-internal
         :startp!!
         (allocate!! nil (format nil "startp-~a" time) 'boolean-pvar)
         :forward!!
         (allocate!! nil (format nil "forward-~a" time) 'neighbor-no-pvar)
         :backward!!
         (allocate!! nil (format nil "backward-~a" time) 'neighbor-no-pvar)
         :endp!!
         (allocate!! nil (format nil "endp-~a" time) 'boolean-pvar)
         :arc-label!!
         (funcall 'make-arc-label)
         )))
    (reset-slot slot)))

; RESET-SLOTS resets the array of SLOTS.
;
(defun reset-slots (slots)
  (*all
   (dotimes (time (length slots))
     (reset-slot (aref slots time))))
  slots)

; MAKE-SLOTS returns an array of slots of size TIME-QUANTUM.
;
(defun make-slots (time-quantum)
  (let ((slots (make-array time-quantum :adjustable t)))
    (dotimes (time time-quantum)
      (setf (aref slots time) (make-slot time)))
    slots))

;
; Trial-Slots for path construction
;

; Trial-slot structure
;
(defun trial-slot
  (:conc-name trial-slot-i-)
  (:constructor make-trial-slot-internal))
  direction!! ; neighbor-no direction of trial-path
  length!! ; length of trial-path
)

; Accessors for trial-slot structure
;
(defun trial-slot-direction!! (trial-slot)
  '(the neighbor-no-pvar (trial-slot-i-direction!! ,trial-slot)))

(defun trial-slot-length!! (trial-slot)
  '(the path-length-pvar (trial-slot-i-length!! ,trial-slot)))

; RESET-TRIAL-SLOT resets the TRIAL-SLOT structure.

```

```

;
(defmacro reset-trial-slot (trial-slot)
  '(progn (*set (trial-slot-direction!! ,trial-slot) (neighbor-limit!!)
              (trial-slot-length!! ,trial-slot) (the path-length-pvar (!! 0)))
          ,trial-slot))
; MAKE-TRIAL-SLOT returns a new trial-slot structure for TIME step.
;
(defun make-trial-slot (time)
  (let ((trial-slot
        (make-trial-slot-internal
         :direction!!
         (allocate!! nil (format nil "direction-~a" time) 'neighbor-no-pvar)
         :length!!
         (allocate!! nil (format nil "length-~a" time) 'path-length-pvar))))
    (reset-trial-slot trial-slot)))
; *DEALLOCATE-TRIAL-SLOT deallocates the pvars in the TRIAL-SLOT structure.
;
(defmacro *deallocate-trial-slot (trial-slot)
  '(progn (*deallocate (trial-slot-direction!! ,trial-slot))
          (*deallocate (trial-slot-length!! ,trial-slot))))
; RESET-TRIAL-SLOTS resets the array of TRIAL-SLOT structures.
;
(defun reset-trial-slots (trial-slots)
  (*all
   (dotimes (time (length trial-slots))
     (reset-trial-slot (aref trial-slots time))))
  trial-slots)
; MAKE-TRIAL-SLOTS returns an array of trial-slots of size TIME-QUANTUM.
;
(defun make-trial-slots (time-quantum)
  (let ((trial-slots (make-array time-quantum :adjustable t)))
    (dotimes (time time-quantum)
      (setf (aref trial-slots time) (make-trial-slot time)))
    trial-slots))
; INC-TIME-QUANTUM grows the SLOTS and TRIAL-SLOTS arrays by DELTA time steps.
;
(defun inc-time-quantum (&optional (delta 1))
  (adjust-array slots[]!! (+ *time-quantum* delta))
  (adjust-array trial-slots[]!! (+ *time-quantum* delta))
  (dotimes (i delta)
    (setf (aref slots[]!! (+ *time-quantum* i))
          (reset-slot (make-slot (+ *time-quantum* i)))
          (aref trial-slots[]!! (+ *time-quantum* i))
          (reset-trial-slot (make-trial-slot (+ *time-quantum* i)))))
  (incf *time-quantum* delta))
; INIT-GRAPH-STRUCTURES initializes the graph structures.
;
(defun init-graph-structures ()
  (setf *time-quantum* 1
        *free* *neighbor-limit*
        *max-path-length* (* 2 *max-hops*)
        *path-len-size* (ceiling (log *max-path-length* 2))
        has-neighbor-p[]!! (make-has-neighbor-p)
        slots[]!! (make-slots *time-quantum*)
        trial-slots[]!! (make-trial-slots *time-quantum*))
; Reset graph structures after *COLD-BOOT.
;
#+:lisp-hardware
(add-initialization "Init Graph Routing Strucs"
  '(init-graph-structures))

```

```

      '*after-cold-boot-initializations*)

#+:lisp-simulator
(add-initialization :name-of-form "Init Graph Routing Strucs"
                   :form '(init-graph-structures)
                   :variable '*after-cold-boot-initializations*)

;;; EOF

#+:ccl
(format t "~%\\"Graph Structures\\" loaded")



---




---



;;; -- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 --

(in-package 'lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Tombouliau Implementation
;;;
;;; Graph Hooks
;;;

; Arc label slot specification
;
(defun slot-spec (:type list)
  label           ; slots fed to ALLOCATE!!
  initial-value  ; slot label
  name           ; initial value
  type           ; slot name
                ; pvar type

; Arc label slot accessors
;
(defun slot-label (slot)
  (if (listp slot)
      (slot-spec-label slot)
      slot))

(defun slot-initial-value (slot)
  (if (listp slot)
      (slot-spec-initial-value slot)))

(defun slot-name (slot)
  (if (listp slot)
      (slot-spec-name slot)))

(defun slot-type (slot)
  (if (listp slot)
      (slot-spec-type slot)))

; STRINGS-TO-SYMBOL returns the symbol formed by the concatenation of STRINGS.
;
(defmacro strings-to-symbol (&rest strings)
  '(read-from-string (funcall #'concatenate 'string ,@strings)))

; MAKE-ARC-SLOTS returns slots for the CM arc label structure.
;
(defun make-arc-slots (slots)
  (mapcar

```

```

#' (lambda (slot)
  '(,(strings-to-symbol (string (slot-label slot)) "!!")
    (allocate!! ,(if (slot-initial-value slot)
                     '(!! ,(slot-initial-value slot)))
                ,(slot-name slot)
                ,@(if (slot-type slot)
                      '(',(slot-type slot))))))
slots))

; MAKE-CM-ARC writes the defstruct form to create the CM arc label structure
; called LABEL-NAME and containing the given SLOTS.
;
(defun make-cm-arc (label-name slots)
  '(defstruct (arc-label
              (:conc-name
               ,(strings-to-symbol (string label-name) "-i-"))
              ,@(make-arc-slots slots)
              )
    )

; MAKE-ARC-ACCESSORS writes the accessor macros for the CM arc label structure.
;
(defun make-arc-accessors (label-name slots)
  (mapcar
   #'(lambda (slot)
       (let* ((slot-label (string (slot-label slot)))
              (slot-accessor
                '(list ',(strings-to-symbol (string label-name) "-i-" slot-label "!!")
                      arc)))
         '(defmacro
            ,(strings-to-symbol (string label-name) "-" slot-label "!!")
            (arc)
            ,(if (slot-type slot)
                  '(list 'the ',(slot-type slot) ,slot-accessor)
                  slot-accessor))))
        slots)
    )

; MAKE-FE-ARC writes the defstruct form to create the FE arc label structure
; called LABEL-NAME and containing the given SLOTS.
;
(defun make-fe-arc (label-name slots)
  '(defstruct ,label-name
    ,@(mapcar #'slot-label slots))

; MAKE-ARC-COPIER writes the arc label copy function to copy an FE arc label
; into the CM arc label at the processor with the given cube ADDR.
;
(defun make-arc-copier (label-name slots)
  '(defun *copy-arc-label (label!! addr label)
    (setf
     ,@(mapcar
        #'(lambda (slot)
            '((pref ,(strings-to-symbol
                    (string label-name) "-" (string (slot-label slot)) "!!")
                  label!!) addr)
              ,(strings-to-symbol
                (string label-name) "-" (string (slot-label slot)))
                label))
          slots))
        label)
    )

; DEF-ARC-LABEL defines corresponding FE and CM arc label structures
; called LABEL-NAME and containing the given SLOTS.
;
(defmacro def-arc-label (label-name &body slots)
  '(progn

```

```

,(make-cm-arc label-name slots)
,@(make-arc-accessors label-name slots)
,(make-fe-arc label-name slots)
,(make-arc-copier label-name slots)
))

;
; Arc Label Access
;
;
; FOR-ALL-ARC-STARTS loops through all slot structures and executes BODY
; with the currently selected set composed of all processors containing
; the beginning of a graph arc. LABEL-NAME is bound to the current CM
; arc label structure.
;
(defmacro for-all-arc-starts ((label-name) &body body)
  (let ((time (gensym)))
    '(let (slot
          ,label-name)
        (dotimes (,time *time-quantum*)
          (setf slot (aref slots[]!! ,time)
                ,label-name (slot-arc-label!! slot))
          (*when (slot-startp!! slot)
                ,@body))))))

; FOR-ALL-ARC-ENDS loops through all slot structures and executes BODY with
; the currently selected set composed of all processors containing the end
; of a graph arc. LABEL-NAME is bound to the current CM arc label structure.
;
(defmacro for-all-arc-ends ((label-name) &body body)
  (let ((time (gensym)))
    '(let (slot
          ,label-name)
        (dotimes (,time *time-quantum*)
          (setf slot (aref slots[]!! ,time)
                ,label-name (slot-arc-label!! slot))
          (*when (slot-endp!! slot)
                ,@body))))))

; In Allegro CL, set FRED indentation for macros
;
#+:ccl
(progn
  (pushnew '(for-all-arc-starts . 1)
    ccl::*fred-special-indent-alist*
    :test #'equal)
  (pushnew '(for-all-arc-ends . 1)
    ccl::*fred-special-indent-alist*
    :test #'equal))

;;; EOF

#+:ccl
(format t "~%\"Graph Hooks\" loaded")



---




---



;;; -- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 --

(in-package '*lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

```

```

;;;
;;; Tomboulian Implementation
;;;
;;; Graph Construction
;;;

; FREE-TO-SENDP!! returns a boolean-pvar indicating if each processor
; is free to send in the current time step.
;
(defmacro free-to-sendp!! ()
  '(=!! (slot-backward!! current-slot)
        (neighbor-limit!!)))

; WHEN-HAS-NEIGHBOR-P executes BODY with the currently selected set
; composed of processors with NEIGHBOR-NO links.
;
(defmacro when-has-neighbor-p ((neighbor-no) &body body)
  '(*when (has-neighbor-n-p!! ,neighbor-no)
    ,@body))

; In Allegro CL, set FRED indentation for macro
;
#+:ccl
(pushnew '(when-has-neighbor-p . 1)
  ccl::*fred-special-indent-alist*
  :test #'equal)

; WHEN-NEIGHBOR-FREE executes BODY with the currently selected set composed
; of processors with neighbors free to receive along NEIGHBOR-NO link.
;
(defmacro when-neighbor-free ((neighbor-no) &body body)
  '(progn
    (pref-1-neighbor!! neighbor-slot-forward!!
      (slot-forward!! current-slot)
      ,neighbor-no)
    (*when (=!! neighbor-slot-forward!! (neighbor-limit!!))
      ,@body)))

; In Allegro CL, set FRED indentation for macro
;
#+:ccl
(pushnew '(when-neighbor-free . 1)
  ccl::*fred-special-indent-alist*
  :test #'equal)

; WHEN-SHORTER-PATH-TO-NEIGHBOR executes BODY with the currently selected set
; composed of processors with shorter trial-paths along NEIGHBOR-NO link.
;
(defmacro when-shorter-path-to-neighbor ((neighbor-no) &body body)
  '(progn
    (pref-1-neighbor!! neighbor-slot-length!! trial-slot-length!! ,neighbor-no)
    (*when (or!!
      (=!! neighbor-slot-length!! (!! 0))
      (<!! (trial-slot-length!! last-trial-slot)
        neighbor-slot-length!!))
      ,@body)))

; In Allegro CL, set FRED indentation for macro
;
#+:ccl
(pushnew '(when-shorter-path-to-neighbor . 1)
  ccl::*fred-special-indent-alist*
  :test #'equal)

; UPDATE-TRIAL-PATH-TO-NEIGHBOR updates a trial-path for each processor
; in the currently selected set along NEIGHBOR-NO link.
(defmacro update-trial-path-to-neighbor (neighbor-no)
  '(progn

```

```

(*pset-1-neighbor (!! (the neighbor-no (neighbor-no-inverse ,neighbor-no)))
  (trial-slot-direction!! current-trial-slot)
  ,neighbor-no)
(*pset-1-neighbor (the path-length-pvar (1+!! (trial-slot-length!! last-trial-slot)))
  (trial-slot-length!! current-trial-slot)
  ,neighbor-no)))

; FLOOD-TRIAL-PATHS floods all possible shortest trial-paths from the processor
; with cube address FROM-ADDR to the processor at TO-ADDR. FLOOD-TRIAL-PATHS
; returns T if some trial-path reached TO-ADDR, NIL otherwise. INC-TIME-QUANTUM-P
; allows the time quantum to be incremented if the destination is not reached.
;
(defun flood-trial-paths (from-addr to-addr &key inc-time-quantum-p)
  (*all
    (reset-trial-slots trial-slots[]!!)
    (*set activep!! nil!!)
    (setf (pref activep!! from-addr) t)

    (let ((reached-p nil)
          (base-time 0)
          current-slot
          current-trial-slot
          (last-trial-slot (make-trial-slot -1)))

      (*let (neighbor-slot-forward!!
            trial-slot-length!!
            neighbor-slot-length!!)
        (declare (type neighbor-no-pvar neighbor-slot-forward!!)
                  (type path-length-pvar trial-slot-length!! neighbor-slot-length!!))

        (do ()
          ((or reached-p
               (and (plusp base-time) (not inc-time-quantum-p))))

          (do ((time base-time (1+ time)))
              ((= time *time-quantum*))

              (setf base-time *time-quantum*
                    current-slot (aref slots[]!! time)
                    current-trial-slot (aref trial-slots[]!! time))

              (*set trial-slot-length!!
                    (trial-slot-length!! last-trial-slot))

              (*when (and!! activep!! (free-to-sendp!!))
                (dotimes (n *neighbor-limit*)
                  (when-has-neighbor-p (n)
                    (when-neighbor-free (n)
                      (when-shorter-path-to-neighbor (n)
                        (update-trial-path-to-neighbor n))))))

              (*set activep!! nil!!)
              (*when (/!! (trial-slot-direction!! current-trial-slot) (neighbor-limit!!))
                (*set activep!! t!!))

              (when (pref activep!! to-addr)
                (setf (pref activep!! to-addr) nil)
                (if (= (pref (slot-forward!! current-slot) to-addr) *neighbor-limit*)
                    (setf reached-p t)
                    (setf (pref (trial-slot-length!! current-trial-slot) to-addr) 0)))

              (setf (pref activep!! from-addr) t)

              (*set (trial-slot-length!! last-trial-slot)
                    (trial-slot-length!! current-trial-slot))
              (setf (pref (trial-slot-length!! last-trial-slot) from-addr) 0)
              )
            (if (and (not reached-p) inc-time-quantum-p )
                (inc-time-quantum))
          ))
  ))

```

```

(*deallocate-trial-slot last-trial-slot)
reached-p
)))

; SHORTEST-TRIAL-PATH-TIME returns the time step on arrival and path length
; of the shortest trial-path reaching the destination processor at cube ADDR.
;
(defun shortest-trial-path-time (addr)
  (let ((path-length (1+ *max-path-length*))
        path-time
        trial-slot-length)
    (dotimes (time *time-quantum*)
      (setf trial-slot-length
            (pref (trial-slot-length!! (aref trial-slots[]!! time)) addr))
      (if (and (plusp trial-slot-length)
              (< trial-slot-length path-length))
          (setf path-length trial-slot-length
                path-time time))
      (if path-time
          (values path-time path-length))))))

; TRACE-TRIAL-PATH-BACKWARDS traces the shortest trial-path backwards from
; TO-ADDR to FROM-ADDR and updates the slots array to establish the arc.
;
;
(defun trace-trial-path-backwards (from-addr to-addr)
  (let ((path-time (shortest-trial-path-time to-addr)))
    (when path-time
      (setf (pref (slot-endp!! (aref slots[]!! path-time)) to-addr) t)

      (let (slot
            trial-slot
            neighbor-no
            neighbor-addr)
        (do ((time path-time (1- time))
              (done-p nil)
              (done-p (1+ time)))

            (setf slot (aref slots[]!! time)
                  trial-slot (aref trial-slots[]!! time))

            (setf neighbor-no (pref (trial-slot-direction!! trial-slot) to-addr)
                  neighbor-addr (cube-from-neighbor-no to-addr neighbor-no)
                  (pref (slot-forward!! slot) to-addr) neighbor-no
                  (pref (slot-backward!! slot) neighbor-addr) (neighbor-no-inverse neighbor-no)
                  to-addr neighbor-addr)
            (if (= from-addr to-addr)
                (setf (pref (slot-startp!! slot) to-addr) t
                      done-p t))))))

; CONNECT-NODES creates an arc starting at FROM-ADDR and ending at TO-ADDR
; and returns the start time step of the arc in the source processor.
; If provided, the FE ARC-LABEL is copied into the CM arc label structure
; at the start time in the source processor.
;
(defun connect-nodes (from-addr to-addr &optional arc-label)
  (flood-trial-paths from-addr to-addr :inc-time-quantum-p t)
  (let ((path-start-time (trace-trial-path-backwards from-addr
                                                    to-addr)))
    (if arc-label
        (*copy-arc-label
         (slot-arc-label!! (aref slots[]!! path-start-time))
         from-addr
         arc-label))
    path-start-time))

; GRAPH-SLOTS-USAGE returns the percentage of graphs slots currently used.

```

```

;
(defun graph-slots-usage ()
  (let (current-slot
        (no-slots-used 0)
        total-no-slots)
    (setf total-no-slots (* *time-quantum* (count-css)))
    (dotimes (time *time-quantum*)
      (setf current-slot (aref slots[]! time))
      (*when (not!! (free-to-sendp!!))
        (incf no-slots-used (count-css))))
    (/ (* 100.0 no-slots-used) total-no-slots)))

;;; EOF

#':ccl
(format t "~%\\"Graph Construction\\" loaded")


```

```

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

(in-package 'lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Tombouliau Implementation
;;;
;;; Graph Utilities
;;;

; DELETE-ARC deletes the graph arc beginning in the processor at START-ADDR at START-TIME.
;
(defun delete-arc (start-addr start-time)
  (setf (pref (slot-startp!! (aref slots[]! start-time)) start-addr) nil)

  (let (slot)
    (do ((time start-time (1+ time))
        (from-addr start-addr)
        to-addr
        (done-p nil))
      ((or done-p (= time *time-quantum*)))

      (setf slot (aref slots[]! time)
            to-addr (cube-from-neighbor-no from-addr
                                           (pref (slot-backward!! slot) from-addr))
                    (pref (slot-backward!! slot) from-addr) *neighbor-limit*
                    (pref (slot-forward!! slot) to-addr) *neighbor-limit*
                    from-addr to-addr)

      (if (pref (slot-endp!! slot) to-addr)
          (setf (pref (slot-endp!! slot) to-addr) nil
                done-p t)))

    ))

;;; EOF

#+:ccl
(format t "~%\\"Graph Utilities\\" loaded")

```

```

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

(in-package 'lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Tombouliau Implementation
;;;
;;; Graph Routing
;;;

; ROUTE-FORWARD implements the forward routing cycle by looping through all slot
; structures in the time quantum. LABEL-NAME is bound to the current CM arc label
; structure. IN-BOX!! is bound to the in-box pvar of type IN-BOX-TYPE. At each time
; step, ROUTE-FORWARD calls ARC-START-FUNCTION with the currently selected set composed
; of processors at the beginning of an arc to inject new messages into the routing cycle.
; ROUTE-FORWARD also calls ARC-END-FUNCTION with the currently selected set composed
; of processors at the end of an arc to receive messages.
;
(defmacro route-forward (label-name
                        in-box!!
                        in-box-type
                        arc-start-function
                        arc-end-function)
  (let ((slot (gensym))
        (out-box!! (gensym)))
    `(let (,label-name)
      (*all
        (*let (,in-box!!
              ,out-box!!)
          (declare (type ,in-box-type ,in-box!! ,out-box!!))
          (map nil
               #'(lambda (,slot)
                   (setf ,label-name (slot-arc-label!! ,slot))
                   (*if (slot-startp!! ,slot)
                       (*set ,out-box!!
                             ,arc-start-function))
                   (*when (/!! (slot-forward!! ,slot) (neighbor-limit!!))
                       (pref-neighbor!! ,in-box!! ,out-box!! (slot-forward!! ,slot))
                       (*set ,out-box!! ,in-box!!))
                   (*if (slot-endp!! ,slot)
                       ,arc-end-function))
                  slots[]!!))
      )))

; ROUTE-BACKWARD implements the backward routing cycle by looping through all slot
; structures in the time quantum. LABEL-NAME is bound to the current CM arc label
; structure. IN-BOX!! is bound to the in-box pvar of type IN-BOX-TYPE. At each time
; step, ROUTE-FORWARD calls ARC-START-FUNCTION with the currently selected set composed
; of processors at the end of an arc to inject new messages into the routing cycle.
; ROUTE-FORWARD also calls ARC-END-FUNCTION with the currently selected set composed
; of processors at the beginning of an arc to receive messages.
;
(defmacro route-backward (label-name
                         in-box!!
                         in-box-type
                         arc-start-function
                         arc-end-function)
  (let ((slot (gensym))
        (out-box!! (gensym)))
    `(let (,label-name)
      )))

```

```

(*all
  (*let (,in-box!!
        ,out-box!!)
    (declare (type ,in-box-type ,in-box!! ,out-box!!))
    (map nil
      #'(lambda (,slot)
          (setf ,label-name (slot-arc-label!! ,slot))
          (*if (slot-endp!! ,slot)
              (*set ,out-box!!
                    ,arc-start-function))
          (*when (/=!! (slot-backward!! ,slot) (neighbor-limit!!))
              (pref-neighbor!! ,in-box!! ,out-box!! (slot-backward!! ,slot))
              (*set ,out-box!! ,in-box!!))
          (*if (slot-startp!! ,slot)
              ,arc-end-function))
      (reverse slots[]!!))
    )))
)

;;; EOF

#+:ccl
(format t "~%\"Graph Routing\" loaded")

```

```

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

(in-package 'lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Tombouliau Implementation
;;;
;;; Neural Net Front-End Structures
;;;
;;; SLAB    set of units
;;; BUNDLE set of connections between 2 slabs with density 0-100%
;;; NET    sets of slabs and bundles,
;;;        with input & output slab (possibly the same)
;;;        represents CONTINUOUS-MAPPING or ASSOCIATIVE-MEMORY net
;;;

; Slab of units
;
(defstruct (net-slab
  #+:symbolics
  (:named)
  (:conc-name slab-)
  (:constructor fe-make-slab-internal)
  (:print-function print-slab)
)
  no ; slab id
  inputp ; input slab?
  outputp ; output slab?
  size ; no of units
  addr ; array of cube addrs of units
)

; PRINT-SLAB prints SLAB on STREAM.
;

```

```

(defun print-slab (slab stream &optional depth)
  (declare (ignore depth))
  (format stream "#<Slab ~a, ~a unit~:*~[s~;~::~s~]~::~[~; I~]~::~[~; O~]>"
    (slab-no slab) (slab-size slab)
    (slab-inputp slab) (slab-outputp slab)))

; SLAB-NO!! returns a slab-no-pvar pvar containing the SLAB id in each processor.
;
(defmacro slab-no!! (slab)
  '(the slab-no-pvar (!! (slab-no ,slab))))

; SLAB-INPUTP!! returns a boolean pvar containing T if SLAB is the input slab.

(defmacro slab-inputp!! (slab)
  '(the boolean-pvar (!! (slab-inputp ,slab))))

; SLAB-OUTPUTP!! returns a boolean pvar containing T if SLAB is the output slab.

(defmacro slab-outputp!! (slab)
  '(the boolean-pvar (!! (slab-outputp ,slab))))

; Bundle of connections
;
(defstruct (net-bundle
  #+:symbolics
  (:named)
  (:conc-name bundle-)
  (:constructor fe-make-bundle-internal)
  (:print-function print-bundle)
)
  no ; bundle id
  to-slab ; connections to slab
  from-slab ; connections from slab
  density ; density of connections, 0-100%
  size ; no of connections
  ; connection = (to-no,from-no)
  to-no ; array of to-slab unit ids
  from-no ; array of from-slab unit ids
  start-time ; start time step of connections
)

; PRINT-BUNDLE prints BUNDLE on STREAM.
;
(defun print-bundle (bundle stream &optional depth)
  (declare (ignore depth))
  (format stream "#<Bundle ~a <- ~a, ~a%>"
    (slab-no (bundle-to-slab bundle))
    (slab-no (bundle-from-slab bundle))
    (bundle-density bundle)))

; BUNDLE-NO!! returns a bundle-no-pvar pvar containing the BUNDLE id in each processor.
;
(defmacro bundle-no!! (bundle)
  '(the bundle-no-pvar (!! (bundle-no ,bundle))))

; Neural net
;
(defstruct (neural-net
  #+:symbolics
  (:named)
  (:conc-name net-)
  (:constructor fe-make-net-internal)
  (:print-function print-net)
)
  name
  type ; CONTINUOUS-MAPPING or ASSOCIATIVE-MEMORY
  (allocation-mode :grid-center) ; allocation mode for slabs

  slabs ; array of slabs
)

```

```

input-slab-no          ; input slab id
output-slab-no        ; output slab id
bundles               ; array of bundles
no-units              ; total number of      units
no-connections        ;                      connections
no-processors         ;                      processors

(a 1.0)                ; dynamical system equation constants
(b 1.0)

(eta 0.25)             ; learning rate
(alpha 0.9)           ; momentum term

(epsilon-x 0.001)     ; feed-forward convergence criterion
(x-iterations 4)     ; min iterations before convergence test
(epsilon-y 0.001)    ; back-propagate convergence criterion
(y-iterations 4)     ; min iterations before convergence test
(epsilon-w 0.1)      ; weight update convergence criterion
(max-updates 10000)  ; max weight updates in training
)

; PRINT-NET prints NET on STREAM.
;
(defun print-net (net stream &optional depth &key verbose-p)
  (declare (ignore depth))
  (if (not verbose-p)
      (format stream
        "#<Net ~a ~a slab~:*~[s~;~::~s~], ~a bundle~:*~[s~;~::~s~]>"
        (net-name net)
        (length (net-slabs net))
        (length (net-bundles net)))
      (format stream "~%~a net: ~a" (net-type net) (net-name net))
      (format stream "~2~a slab~:*~[s~;~::~s~], ~a bundle~:*~[s~;~::~s~]"
        (length (net-slabs net)) (length (net-bundles net)))
      (format stream "~%~a unit~:*~[s~;~::~s~], ~a connection~:*~[s~;~::~s~] ->~"
        ~a processor~:*~[s~;~::~s~]"
        (net-no-units net) (net-no-connections net) (net-no-processors net))
      (format stream "~2~a = ~a, b = ~a" (net-a net) (net-b net))
      (format stream "~2~a Feed-forward convergence = ~a, min ~a iteration~:*~[s~;~::~s~]"
        (net-epsilon-x net) (net-x-iterations net))
      (format stream "~2~a Back-propagate convergence = ~a, min ~a iteration~:*~[s~;~::~s~]"
        (net-epsilon-y net) (net-y-iterations net))

      (format stream "~2~a eta = ~a, alpha = ~a" (net-eta net) (net-alpha net))
      (format stream "~2~a Weight update convergence = ~a, max ~a iteration~:*~[s~;~::~s~]"
        (net-epsilon-w net) (net-max-updates net))
      (terpri stream)))

; GET-SLAB returns the slab with id SLAB-NO in NET.
;
(defun get-slab (net slab-no)
  (aref (net-slabs ,net) ,slab-no))

; GET-INPUT-SLAB returns the input slab in NET.
;
(defun get-input-slab (net)
  (aref (net-slabs ,net) (net-input-slab-no ,net)))

; GET-OUTPUT-SLAB returns the output slab in NET.
;
(defun get-output-slab (net)
  (aref (net-slabs ,net) (net-output-slab-no ,net)))

; GET-BUNDLE returns the bundle with id BUNDLE-NO in NET.
;
(defun get-bundle (net bundle-no)
  (aref (net-bundles ,net) ,bundle-no))

; MEMORY-NETP returns T if NET is an ASSOCIATIVE-MEMORY net.

```

```

;
(defmacro memory-netp (net)
  '(eq (net-type ,net) 'associative-memory))

;;; EDF

#+:ccl
(format t "~%\\"Net FE Structures\\" loaded")



---




---



;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

(in-package 'lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Tombouliau Implementation
;;;
;;; Net CM Structures
;;;

; Units

(*proclaim '(type boolean-pvar unitp!))
(*proclaim '(type boolean-pvar inputp!))
(*proclaim '(type boolean-pvar outputp!))
(*proclaim '(type slab-no-pvar slab-no!))
(*proclaim '(type unit-no-pvar unit-no!))

(*defvar unitp!! nil!!) ; processor is net unit?
(*defvar inputp!! nil!!) ; input unit?
(*defvar outputp!! nil!!) ; output unit?
(*defvar slab-no!!) ; unit slab id
(*defvar unit-no!!) ; unit id

; Variables appearing in feed-forward and back-propagation equations
;
(*proclaim '(type single-float-pvar a!))
(*proclaim '(type single-float-pvar b!))
(*proclaim '(type single-float-pvar X!))
(*proclaim '(type single-float-pvar Z!))
(*proclaim '(type single-float-pvar dX!))
(*proclaim '(type single-float-pvar U!))
(*proclaim '(type single-float-pvar LogU!))
(*proclaim '(type single-float-pvar I!))
(*proclaim '(type single-float-pvar Y!))
(*proclaim '(type single-float-pvar dY!))
(*proclaim '(type single-float-pvar V!))
(*proclaim '(type single-float-pvar J!))

(*defvar a!)
(*defvar b!)
(*defvar X!)
(*defvar X!)
(*defvar Z!)
(*defvar dX!)
(*defvar U!)
(*defvar LogU!)
(*defvar I!)
(*defvar Y!)

```

```

(*defvar dY!!)
(*defvar V!!)
(*defvar J!!)

(*proclaim '(type single-float-pvar epsilon-x!!))
(*proclaim '(type single-float-pvar epsilon-y!!))
(*proclaim '(type single-float-pvar epsilon-w!!))

(*defvar epsilon-x!!)           ; feed-forward convergence criterion
(*defvar epsilon-y!!)           ; back-propagate convergence criterion
(*defvar epsilon-w!!)           ; weight update convergence criterion

(*proclaim '(type single-float-pvar eta!!))
(*proclaim '(type single-float-pvar alpha!!))

(*defvar eta!!)                 ; learning rate
(*defvar alpha!!)               ; momentum term

; Connections

; Define arc labels for net connections
;
;
(def-arc-label connection
  (W nil 'weight single-float-pvar)           ; connection weight
  (dW nil 'dW single-float-pvar)             ; current gradient
  (dWold nil 'dWold single-float-pvar)       ; previous gradient
)

;;; EOF

#+:ccl
(format t "~%" "CM Net Structures\" loaded")


```

```

;;; -- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 --

(in-package '*lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Tomboulian Implementation
;;;
;;; Make Net Structures on Front-End
;;;

; Front-End slab structure

; FE-MAKE-SLAB returns a net slab of SIZE units with id NO. INPUTP and OUTPUTP
; indicate if this slab is the input or output slab, respectively.
;
(defun fe-make-slab (no size inputp outputp)
  (fe-make-slab-internal :no no
    :size size
    :inputp inputp
    :outputp outputp))

; Front-End bundle structure

; RANDOM-CONNECTIONS returns ROW number and COL number arrays representing a bundle

```

```

; of connections to a slab of TO-SIZE units from a slab of FROM-SIZE units.
; Each possible connection is formed with probability given by DENSITY.
;
(defun random-connections (to-size from-size density)
  (let (n row col)
    (setf density (/ density 100.0))
    (*all
      (*let ((rendezvous!! (!! 0)))
        (declare (type cube-address-pvar rendezvous!!))
        (*when (and!! (<!! (self-address!!)
                          (the max-int-pvar (!! (* to-size from-size))))
                (<!! (random-float!! (!! 0.5) (!! 0.5))
                    (the float-pvar (!! density))))
          (setf n (count-css))
          (*pset :no-collisions
                (self-address!!)
                rendezvous!!
                (enumerate!!)))
        (setf row (make-array n)
              col (make-array n))
        (pvar-to-array (truncate!! rendezvous!! (!! from-size))
                       row
                       :cube-address-end n)
        (pvar-to-array (mod!! rendezvous!! (!! from-size))
                       col
                       :cube-address-end n)))
      (values row col)))

; FE-MAKE-BUNDLE returns a bundle with id NO connecting TO-SLAB and FROM-SLAB
; with the probability of each connection given by DENSITY.
;
(defun fe-make-bundle (no to-slab from-slab density)
  (let ((bundle (fe-make-bundle-internal
                :no no
                :to-slab to-slab
                :from-slab from-slab
                :density density)))
    (multiple-value-setf
      ((bundle-to-no bundle)
       (bundle-from-no bundle)
       (random-connections (slab-size to-slab)
                           (slab-size from-slab)
                           density))
      (setf (bundle-size bundle) (length (bundle-to-no bundle)))
      bundle))

; Front-End net structure

; A bundle spec is a list of the form (<to slab id> <from slab id> <density>).
;
(defstruct (bundle-spec
           (:type list))
  to-slab
  from-slab
  density)

; FE-MAKE-NET returns the net NAME of the given TYPE (CONTINUOUS-MAPPING or
; ASSOCIATIVE-MEMORY). SLABS must be a list of total units in each slab, and
; INPUT-SLAB-NO and OUTPUT-SLAB-NO identify the input and output slabs,
; respectively. BUNDLES must be a list of bundle specifications. Additional
; keyword arguments are passed in to FE-MAKE-NET-INTERNAL allowing other
; net parameters to be set.
;
(defun fe-make-net (name type slabs input-slab-no output-slab-no bundles
                  &rest other-net-keys &key &allow-other-keys)
  (let ((net (apply #'fe-make-net-internal
                    :name (string name)
                    :type type
                    :input-slab-no input-slab-no
                    :output-slab-no output-slab-no
                    bundles
                    other-net-keys
                    &allow-other-keys)))
    net))

```

```

        :output-slab-no output-slab-no
        other-net-keys)))
(let ((slab-no -1))
  (setf (net-slabs net)
        (map 'array
              #'(lambda (slab-size)
                  (fe-make-slab (incf slab-no)
                                slab-size
                                (eq input-slab-no slab-no)
                                (eq output-slab-no slab-no)))
              slabs)))
(let ((bundle-no -1))
  (setf (net-bundles net)
        (map 'array
              #'(lambda (bundle)
                  (fe-make-bundle (incf bundle-no)
                                  (get-slab net (bundle-spec-to-slab bundle))
                                  (get-slab net (bundle-spec-from-slab bundle))
                                  (bundle-spec-density bundle)))
              bundles)))
(fe-size-net net)
net))

; FE-SIZE-NET sets the total number of units, connections and processors required by NET.
;
(defun fe-size-net (net)
  (setf (net-no-units net)
        (reduce #'+
                (map 'list #'slab-size (net-slabs net)))
        (net-no-connections net)
        (reduce #'+
                (map 'list #'bundle-size (net-bundles net)))
        (net-no-processors net)
        (net-no-units net)))

; CONTINUOUS-MAPPING net

; DEF-MAPPING-NET returns a CONTINUOUS-MAPPING net called NAME specified by
; the keyword arguments SLABS, INPUT-SLAB-NO, OUTPUT-SLAB-NO and BUNDLES.
; Additional keyword arguments can be used to specify other net parameters.
;
(defmacro def-mapping-net (name &rest other-net-keys
                          &key slabs input-slab-no output-slab-no bundles
                          &allow-other-keys)
  '(progn
    (defvar ,name)
    (setf ,name (fe-make-net ',name
                             'continuous-mapping
                             ,slabs
                             ,input-slab-no
                             ,output-slab-no
                             ,bundles
                             ,@other-net-keys))
    (cm-net-cold-boot ,name)
    (cm-make-net ,name)))

; ASSOCIATIVE-MEMORY net

; DEF-MEMORY-NET returns an ASSOCIATIVE-MEMORY net called NAME specified by
; the keyword arguments SLABS, INPUT-SLAB-NO and BUNDLES. Additional keyword
; arguments can be used to specify other net parameters.
;
(defmacro def-memory-net (name &rest other-net-keys
                         &key slabs input-slab-no bundles
                         &allow-other-keys)
  '(progn
    (defvar ,name)
    (setf ,name (fe-make-net ',name
                             'associative-memory

```

```

        ,slabs
        ,input-slab-no
        ,input-slab-no
        ,bundles
        ,@other-net-keys))
    (cm-net-cold-boot ,name)
    (cm-make-net ,name)))

;;; EOF

#+:ccl
(format t "~%\\"FE Make Net\\" loaded")

|-----|
|-----|

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

(in-package 'lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Tombouliau Implementation
;;;
;;; Make Net Structures on CM
;;;

; CM slab structure
; CM-MAKE-SLAB creates the structure for SLAB on the CM according to ALLOCATION-MODE.
;
(defun cm-make-slab (slab allocation-mode)
  (let ((slab-size (slab-size slab)))
    (with-n-proc-allocated (slab-size slab-addr allocation-mode)
      (setf (slab-addr slab) slab-addr)
      (*set unitp!! t!!
         inputp!! (slab-inputp!! slab)
         outputp!! (slab-outputp!! slab)
         slab-no!! (slab-no!! slab)
         unit-no!! (enumerate!!))))
    slab)

; CM bundle structure
; CM-MAKE-BUNDLE creates the structure for BUNDLE on the CM.
;
(defun cm-make-bundle (bundle)
  (let ((to-addr (slab-addr (bundle-to-slab bundle)))
        (from-addr (slab-addr (bundle-from-slab bundle))))
    (setf (bundle-start-time bundle)
          (map 'array #'(lambda (from-no to-no)
                        (connect-nodes (aref from-addr from-no)
                                       (aref to-addr to-no)))
              (bundle-from-no bundle)
              (bundle-to-no bundle))))
    bundle)

; CM net structure
; CM-NET-COLD-BOOT cold boots the CM with the dimensions necessary for NET.
;
(defun cm-net-cold-boot (net)
  (let ((cm-dims (cm-best-fit-dims (net-no-processors net))))
    (or cm-dims
        (error "Net "a too large for CM" (net-name net))))

```

```

(*cold-boot :initial-dimensions cm-dims))

; CM-MAKE-NET creates the structure for NET on the CM.
;
(defun cm-make-net (net)
  (map nil #'(lambda (slab)
    (cm-make-slab slab (net-allocation-mode net)))
    (net-slabs net))
  (map nil #'(lambda (bundle)
    (cm-make-bundle bundle))
    (net-bundles net))
  (*all
    (*when unitp!!
      (*set a!! (the float-pvar (!! (net-a net)))
        b!! (the float-pvar (!! (net-b net)))
        epsilon-x!! (the float-pvar (!! (net-epsilon-x net)))
        epsilon-y!! (the float-pvar (!! (net-epsilon-y net)))
        epsilon-w!! (the float-pvar (!! (net-epsilon-w net)))
        eta!! (the float-pvar (!! (net-eta net)))
        alpha!! (the float-pvar (!! (net-alpha net))))))
    (cm-reset-weights)
    net)

; CM-RESET-WEIGHTS resets the weights for each connection in the net
; to a random float in the interval [mean-interval , mean+interval].
;
(defun cm-reset-weights (&optional (mean 0.0) (interval 0.5))
  (for-all-arc-starts (connection!!)
    (*set (connection-W!! connection!!)
      (random-float!!
        (the float-pvar (!! mean))
        (the float-pvar (!! interval))))))

;;; EOF

#+:ccl
(format t "~%" "CM Make Net" loaded")

```

```

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

```

```

(in-package 'lisp)

```

```

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

```

```

;;;
;;; Tombouliau Implementation
;;;
;;; CM Net Access
;;;

```

```

;
; Slab Access
;

```

```

; GET-SLAB-PVAR returns an array containing the values of PVAR
; for the slab with id SLAB-NO in NET.
;

```

```

(defun get-slab-pvar (net slab-no pvar)
  (let ((slab (get-slab net slab-no)))
    (*all

```

```

(*let (mail-box!!)
  (*when (and!! unitp!! (=?!! slab-no!! (slab-no!! slab)))
    (*pset :no-collisions pvar mail-box!! (enumerate!!))
    (pvar-to-array mail-box!! (make-array (slab-size slab)
      :cube-address-end (count-css))))))

; GET-SLAB-X returns an array containing the values of X!!
; for the slab with id SLAB-NO in NET.
;
(defmacro get-slab-X!! (net slab-no)
  `(get-slab-pvar ,net ,slab-no X!!))

; GET-NET-OUTPUT returns an array containing the output values of NET.
;
(defmacro get-net-output (net)
  `(get-slab-X!! ,net (net-output-slab-no ,net)))

;
; Bundle Access
;

; GET-BUNDLE-W returns an array containing the values of W!!
; for the bundle with id BUNDLE-NO in NET.
;
(defun get-bundle-W!! (net bundle-no)
  (let* ((bundle (get-bundle net bundle-no))
    (from-slab (bundle-from-slab bundle))
    (from-addr (slab-addr from-slab))
    (from-no (bundle-from-no bundle))
    (start-time (bundle-start-time bundle))
    (W (make-array (bundle-size bundle))))
    (dotimes (c (bundle-size bundle))
      (setf (aref W c)
        (pref (aref slots[]!! (aref start-time c))
          (aref from-addr (aref from-no c))))))
  ))

;;; EOF

#+:ccl
(format t "~%" "CM Net Access" loaded")

```

```

;;; -- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 --

```

```

(in-package '*lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Tomboulian Implementation
;;;
;;; Training Sets
;;;

; Training Exemplar
;
(defstruct (exemplar
  (:type list))
  input-pvar
  input-vec

```

```

target-pvar
target-vec)

; Training Set
;
(defstruct (training-set
           (:type list))
  type                ; MAPPING-SET or MEMORY-SET
  name
  exemplars)         ; list of exemplars

; GET-EXAMPLAR returns the exemplar with id EXAMPLAR -NO in TRAINING-SET.
;
(defmacro get-exemplar (training-set exemplar-no)
  `(nth ,exemplar-no (training-set-exemplars ,training-set)))

; CM-LOAD-TRAINING-PAIR loads the INPUT/TARGET training vectors into the
; CONTINUOUS-MAPPING net structure on the CM and returns an exemplar
; containing the INPUT and TARGET vectors. The pvars corresponding
; to the training pair are marked with the given SET-NAME and PAIR-NO.
;
(defun cm-load-training-pair (set-name pair-no input target)
  (*all
    (let ((input!! (allocate!! nil
                               (format nil "~a~a-I" set-name pair-no)
                               'float-pvar))
          (target!! (allocate!! nil
                               (format nil "~a~a-T" set-name pair-no)
                               'float-pvar)))
      (*let (mail-box!!)
        (declare (type float-pvar mail-box!!))
        (array-to-pvar input mail-box!! :cube-address-end (length input))
        (*when inputp!!
          (*set (the float-pvar input!!)
                (pref!! mail-box!! unit-no!! :collision-mode :no-collisions)))
        (array-to-pvar target mail-box!! :cube-address-end (length target))
        (*when outputp!!
          (*set (the float-pvar target!!)
                (pref!! mail-box!! unit-no!! :collision-mode :no-collisions)))
        (list input!! input target!! target))))))

; CM-LOAD-MAPPING-SET loads the TRAINING-PAIRS labeled SET-NAME into the
; CONTINUOUS-MAPPING net structure on the CM and returns the resulting
; training set. TRAINING-PAIRS must be a list of input/target vector lists.
;
(defun cm-load-mapping-set (set-name training-pairs)
  (let ((pair-no -1))
    (list 'mapping-set set-name
          (mapcar #'(lambda (pair)
                      (cm-load-training-pair set-name
                                             (incf pair-no)
                                             (first pair)
                                             (second pair)))
              training-pairs))))

; CM-LOAD-MEMORY-INPUT loads the INPUT vector into the CONTINUOUS-MAPPING
; net structure on the CM and returns an exemplar. The pvar corresponding
; to the INPUT vector is marked with the given SET-NAME and INPUT-NO.
;
(defun cm-load-memory-input (set-name input-no input)
  (*all
    (let ((input!! (allocate!! nil
                               (format nil "~a~a-I" set-name input-no)
                               'float-pvar)))
      (*let (mail-box!!)
        (declare (type float-pvar mail-box!!))
        (array-to-pvar input mail-box!! :cube-address-end (length input))
        (*when inputp!!
          (*set (the float-pvar input!!)
                (pref!! mail-box!! unit-no!! :collision-mode :no-collisions))))))

```

```

(pref!! mail-box!! unit-no!! :collision-mode :no-collisions)))
(list input!! input input!! input)))

; CM-LOAD-MEMORY-SET loads the TRAINING-LIST labeled SET-NAME into the
; CONTINUOUS-MAPPING net structure on the CM and returns the resulting
; training set. The TRAINING-LIST must be a list of input vectors.
;
(defun cm-load-memory-set (set-name training-set)
  (let ((input-no -1))
    (list 'memory-set set-name
          (mapcar #'(lambda (input)
                      (cm-load-memory-input set-name
                                             (incf input-no)
                                             input))
                training-set))))

; CM-UNLOAD-TRAINING-SET unloads TRAINING-SET from the CONTINUOUS-MAPPING or
; ASSOCIATIVE-MEMORY net structure on the CM. The pvars in the TRAINING-SET
; array are deallocated and should no longer be accessed.
;
(defun unload-training-set (training-set)
  (let ((type (training-set-type training-set)))
    (map nil
         #'(lambda (exemplar)
              (*deallocate (exemplar-input-pvar exemplar)
                            (if (eq type 'mapping-set)
                                (*deallocate (exemplar-target-pvar exemplar))))
            (training-set-exemplars training-set))))

; PRINT-TRAINING-SET prints the TRAINING-SET's input/target or
; input vectors for a MAPPING-SET or MEMORY-SET, respectively.
;
(defun print-training-set (training-set)
  (let ((type (training-set-type training-set)))
    (format t "~2%a: ~a" type (training-set-name training-set))
    (map nil
         #'(lambda (exemplar)
              (format t "%i: " (print-vec (exemplar-input-vec exemplar))
                    (when (eq type 'mapping-set)
                        (format t " t: ") (print-vec (exemplar-target-vec exemplar))))
            (training-set-exemplars training-set))))

;;; EOF

#+:ccl
(format t "%\n" "Training Sets\n" loaded")

```

```

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

```

```

(in-package '*lisp)

```

```

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

```

```

;;;
;;; Tombouliau Implementation
;;;
;;; Net Learning
;;;

```

```

; DEBUG-LEARNING sets toggles the :NET-DEBUG flag in the features list.

```

```

;
(defun debug-learning (&optional (debug-on t))
  (if debug-on
      (pushnew :net-debug *features*)
      (setf *features* (delete :net-debug *features*))))

; Scalar LOGISTIC function
;
(defun logistic (x)
  (/ (1+ (exp (- x)))))

; Parallel LOGISTIC function
;
/defmacro logistic!! (x!!)
  '(!! (the single-float-pvar (1+!! (exp!! (-!! ,x!!)))))

; Scalar LOGISTIC derivative
;
(defun dLogistic (x)
  (let ((logistic (logistic x)))
    (* logistic (- 1 logistic))))

; Parallel LOGISTIC derivative
;
(defmacro dLogistic!! (x!!)
  (let ((logistic!! (gensym)))
    `(*let ((,logistic!! (logistic!! ,x!!))
            (declare (type single-float-pvar ,logistic!!))
            (*!! ,logistic!! (-!! (!! 1) ,logistic!!)))
      )

; *NORM of pvar
;
(defmacro *norm (x!!)
  '(sqrt (*sum (*!! ,x!! ,x!!)))

;
; Feed-forward
;
; FEED-FORWARD computes a single feed-forward cycle of NET with the given INPUT!!.
; If NET is an ASSOCIATIVE-MEMORY net and LATCHED-P is T, then FEED-FORWARD
; operates on the master network rather than the slave network.
;
(defun feed-forward (net input!! &key latched-p)
  (*all
   (*when unitp!!
    (*set I!! (!! 0.0)
      X!! (!! 0.5)
      dX!! (!! 0.5))
    (*when inputp!!
     (if (memory-netp net)
         (*set X!! (the single-float-pvar input!!)
           *set I!! (the single-float-pvar input!!))))
    (*set Z!! X!!)

   (do ()
    ((*and (<!! (abs!! dX!!) epsilon-x!!)))

    (dotimes (i (net-x-iterations net))
     #+:net-debug
     (progn
      (format-pvars (U!! LogU!! dX!! X!! Z!!))
      (format t "~%Hit any key to continue: ") (read-char))

     (*set U!! (!! 0.0))

     (route-forward connection!! WZi!! single-float-pvar
      (*!! Z!! (connection-W!! connection!!)))

```

```

                (*set U!! (+!! U!! WZi!!))
(*set LogU!! (logistic!! U!!)
  dX!! (+!! (*!! a!! (-!! X!!)) (*!! b!! LogU!!) I!!)
  X!! (+!! X!! dX!!)
  Z!! X!!)
(if latched-p
  (*when inputp!!
    (*set Z!! (the single-float-pvar input!!)))
))
)))

;
; Back-Propagate
;

; BACK-PROPAGATE computes a single back-propagation cycle of NET with the given TARGET!!.
;
(defun back-propagate (net target!!)
  (*all
    (*when unitp!!
      (*if outputp!!
        (*set J!! (-!! (the single-float-pvar target!!) X!!))
        (*set J!! (!! 0.0)))
      (*set Y!! (!! 0.0)
        dY!! (!! 0.5))

      (do ()
        ((*and (<!! (abs!! dY!!) epsilon-y!!)))

        (dotimes (i (net-y-iterations net))
          #+:net-debug
          (progn
            (format-pvars (LogU!! V!! dY!! Y!!))
            (format t "~%Hit any key to continue: ") (read-char))
            (*set V!! (!! 0.0))
            (route-backward connection!! Yj!! single-float-pvar
              Y!!
              (*set V!! (+!! V!! (*!! Yj!! (connection-W!! connection!!))))))
            (if (memory-netp net)
              (*when inputp!!
                (*set V!! (!! 0.0)))
              (*set dY!! (+!! (*!! a!! (-!! Y!!))
                (*!! b!! LogU!! (-!! (!! 1.0) LogU!!)
                (+!! V!! J!!)))
                Y!! (+!! Y!! dY!!))
              ))
          )))

;
; Gradient Update
;

; GRADIENT-UPDATE increments the current weight-space gradient.
;
(defun gradient-update ()
  (route-backward connection!! Yj!! single-float-pvar
    Y!!
    (*set (connection-dW!! connection!!)
      (+!! (connection-dW!! connection!!)
        (*!! Yj!! Z!!))))

;
; Weight Update
;

; WEIGHT-UPDATE updates the connection weights using the current and last gradients.
;
(defun weight-update ()
  (for-all-arc-starts (connection!!)

```

```

(*set (connection-W!! connection!!)
      (+!! (connection-W!! connection!!)
           (*!! eta!! (connection-dW!! connection!!))
           (*!! alpha!! (connection-dWold!! connection!!))))
(connection-dWold!! connection!!)
(connection-dW!! connection!!)
))

;
; Net Training
;

; STEEPEST-DESCENT performs a true steepest-descent adjustment of the connection weights
; for the input/target pairs in TRAINING-SET. STEEPEST-DESCENT returns T or NIL
; indicating if TRAINING-SET has been learned within the weight update criterion and
; the current target error. If provided, the PRINT-NET-IO function is called to report
; the net's input and output.
;
(defun steepest-descent (net training-set &key print-net-io)
  (let ((learned-p t)
        (target-error 0.0))
    (*all
      (for-all-arc-starts (connection!!)
        (*set (connection-dW!! connection!!) (!! 0.0)))

      (dolist (exemplar (training-set-exemplars training-set))

        (feed-forward net (exemplar-input-pvar exemplar) :latched-p (memory-netp net))
        (back-propagate net (exemplar-target-pvar exemplar))
        (*when outputp!!
          (setf learned-p
                (and learned-p
                     (*and (<!! (abs!! J!!) epsilon-w!))))
          (incf target-error (*norm J!)))

        (if print-net-io
            (funcall print-net-io (exemplar-input-vec exemplar) (get-net-output net)))

        (gradient-update))

      (weight-update))

    (values learned-p target-error))
)

; TRAIN-NET trains NET using the given TRAINING-SET. If specified, PRINT-TRAINING-SET
; is called to print the current TRAINING-SET. In addition, PRINT-NET-IO may be used
; to report the net's input and output each PRINT-INTERVAL iterations.
;
(defun train-net (net training-set &key print-training-set print-interval print-net-io)
  (format t "~2%Net Training~%"
    (print-net net t nil :verbose-p t)
    (if print-training-set
        (funcall print-training-set training-set)))

  (*all
    (for-all-arc-starts (connection!!)
      (*set (connection-dWold!! connection!!) (!! 0.0)))

    (do ((iteration 0 (1+ iteration))
        (learned-p nil)
        (target-error)
        (print-net-io-p print-interval
          (and print-interval
              (zerop (mod (1+ iteration) print-interval)))))

      ((or learned-p
           (and (net-max-updates net)
                (= iteration (net-max-updates net))))))
  )

```

```

(format t "~2%Training set ~:[not ~;~]learned after ~a iteration~:*~[s~;~;~s~].~%"
  learned-p iteration)
(when (and print-interval print-net-io)
  (map nil #'(lambda (exemplar)
    (feed-forward net (exemplar-input-pvar exemplar))
    (funcall print-net-io
      (exemplar-input-vec exemplar)
      (get-net-output net))))
  (training-set-exemplars training-set))
(format t "~%Error = ~a" target-error))

(if print-net-io-p
  (format t "~2%Iteration ~a" iteration))

(multiple-value-setf
  (learned-p target-error)
  (steepest-descent net
    training-set
    :print-net-io (if print-net-io-p print-net-io)))

(if print-net-io-p
  (format t "~%Error = ~a" target-error))
)

(values))

;;; EOF

#+:ccl
(format t "~%\\"Net Learning\\" loaded")

```

```

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

```

```

(in-package '*lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Tombouliau Implementation
;;;
;;; OR Test Nets
;;;

; IOR Continuous Mapping Net

(def-mapping-net or-mapping-net
  :slabs '(2 1 1 1)
  :input-slab-no 0
  :output-slab-no 2
  :bundles '((1 0 100)
            (2 0 100)
            (2 1 100)
            (1 3 100)
            (2 3 100)
            (3 3 100))
  )

(defvar *ior-mapping-pairs*)
(setf *ior-mapping-pairs*

```

```

(list-to-array-pairs '(((0.0 0.0) (0.0))
                      ((0.0 1.0) (1.0))
                      ((1.0 0.0) (1.0))
                      ((1.0 1.0) (1.0))))

(defvar *ior-mapping-set*)
(setf *ior-mapping-set*
      (cm-load-mapping-set 'ior-mapping-set *ior-mapping-pairs*))

(train-net or-mapping-net
  *ior-mapping-set*
  :print-training-set #'print-training-set
  :print-interval 10
  :print-net-io #'print-io-vecs)

; XOR Associative Memory Net

(def-memory-net or-memory-net
  :slabs '(3 1)
  :input-slab-no 0
  :bundles '((0 0 100)
             (0 1 100)
             (1 1 100)
             )
  :epsilon-w 0.05
  )

(defvar *xor-memory-list*)
(setf *xor-memory-list*
      (list-to-array '(((0.0 0.0 0.0)
                       (0.0 1.0 1.0)
                       (1.0 0.0 1.0)
                       (1.0 1.0 0.0)))))

(defvar *xor-memory-set*)
(setf *xor-memory-set*
      (cm-load-memory-set 'xor-memory-set *xor-memory-list*))

(train-net or-memory-net
  *xor-memory-set*
  :print-training-set #'print-training-set
  :print-interval 20
  :print-net-io #'print-io-vecs)

;;; EOF

#+:ccl
(format t "~%" "Test Nets\" loaded")



---




---



;;; -*- Mode: LISP; Syntax: Common-lisp; Package: *LISP; Base: 10 -*-

(in-package '*lisp)

;;;
;;; Etienne Deprit
;;; Naval Research Lab, Code 8242
;;;

;;;
;;; Tombouliau Implementation
;;;
;;;
;;; Time Nets

```



```

    (feed-forward mapping-net
      (exemplar-input-pvar (get-exemplar mapping-set 0)))
  (cm-time-and-print
    (back-propagate mapping-net
      (exemplar-target-pvar (get-exemplar mapping-set 0))))

  (if (and (= n 1) (/= increment 1)) (decf n)
    )))

; TIME-MEMORY-NET compiles timing statistics for the ASSOCIATIVE-MEMORY test net
; up to MAX-N. INCREMENT controls granularity of the increment in net size.
;
(defun time-memory-net (max-n &key (increment 1))
  (cmi::calibrate-cm-timer)
  (let (memory-net
        memory-set)
    (do ((n 1 (+ increment n)))
        ((> n max-n))
      (setf memory-net
            (fe-make-net 'memory-net
                          :associative-memory
                          (list (* 8 n) n)
                          0
                          0
                          '((1 0 25)
                            (0 1 25)
                            )
                          ))
        (format t "~3%*** N = ~a ***" n)
        (format t "~%Memory net: ~a units, ~a connections -> ~a processors"
                (net-no-units memory-net)
                (net-no-connections memory-net)
                (net-no-processors memory-net))
        (cm-net-cold-boot memory-net)
        (format t "~%VP ratio = ~a" cm:*virtual-to-physical-processor-ratio*)
        (cm-time-and-print
          (cm-make-net memory-net))

        (format t "~%Allocation mode ~a" (net-allocation-mode memory-net))
        (format t "~%Time quantum = ~a" *time-quantum*)
        (format t "~%Routing table usage = ~6,3f%" (*all (graph-slots-usage))))

      (setf memory-set
            (cm-load-memory-set 'memory-set
                                (list (make-array (* 8 n) :initial-element 1.0))))

      (cm-time-and-print
        (feed-forward memory-net (exemplar-input-pvar (get-exemplar memory-set 0))))
      (cm-time-and-print
        (back-propagate memory-net (exemplar-input-pvar (get-exemplar memory-set 0))))

      (if (and (= n 1) (/= increment 1)) (decf n)
        )))

;;; EOF

#+:ccl
(format t "~%\\"Time Nets\\" loaded")

```