The Pleasures of SINN:
A System for Programming Connectionist Models

By

Erich J. Smythe
Computer Science Dept.
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 189

The Pleasures of SINN:
A System for Programming Connectionist Models

by

Erich J. Smythe

February, 1986

The Pleasures of SINN: A System for Programming Connectionist Models

Erich J. Smythe[*]

Abstract

This paper describes SINN, a system for the implementation of Connectionist Models (CM's). Nodes in a CM can specify computations to be executed in parallel with the rest of the model. The paper illustrates the features of the programming language SCHEME used to develop abstractions such as objects, and the use of engines to simulate concurrency. These abstractions are used in the development of the nodes, connections, and assemblies that make up SINN, providing freedom from many programming details and reducing the time needed to implement a model. An associative search network [Barto *et al.* 81] and a small letter recognition model [McClelland & Rumelhart 81] are implemented to illustrate the power of SINN.

1. Introduction

Recently there has been a resurgence of interest in Cognitive Science on Connectionist Models (CM's) [Hillis 85; Hinton & Anderson 82; Feldman & Ballard 81] based on an architectural metaphor of the brain. Many of these models can be viewed as nodes or computing elements analogous to neurons, communicating simple information to each other across connections analogous to synapses. Essential to the present discussion is the notion of these nodes as explicit processes computing in parallel with each other. Implementation of these models then becomes a problem in Distributed Processing [Filman & Friedman 84].

This paper describes the SINN programming system (**S**cheme **I**mplementation of **N**eural **N**etworks), and shows how SCHEME [Clinger 85] is used in its implementation. The system contains abstractions for the specification of networks and provides a time–share kernel to allow execution of each node's computation in parallel with others in the model. An Associative Search Network (ASN) presented by Barto [Barto *et al* 81] and a simplified McClelland–Rumelhart letter recognition model [McClelland & Rumelhart 81] are presented to show the flavor of SINN and to demonstrate its flexibility for use in other models.

The version of the programming language SCHEME used in the system [Dybvig & Smith] implements a facility called engines used for the simulation of concurrent processing. SCHEME [Clinger 85] uses first–class functions and closures which along with engines supply powerful abstraction facilities that provide freedom from many details of programming, resulting in convenient and concise implementations of connectionist models. The greatest benefit of SINN lies in the time needed to design a model. The design and coding of the ASN took less than an hour, exclusive of the time spent searching for a suitable noise function. (The letter recognition model was designed incrementally while the system was developed.) Once the code for the basic nodes is defined, the designer of a model need only be familiar with the specialized definition language as seen in the examples.

The next section presents an overview of Scheme and describes the implementation of objects and the use of engines to simulate concurrency. The next section describes the nodes, connections, and assemblies that make up SINN. The system is used to implement the ASN and the Letter Recognition models. Finally, the performance and future of the system are described.

2. Scheme Essentials

This section is an introduction to the features of Scheme. We assume the reader is familiar with Lisp.

2.1 The Basics

Procedures in Scheme are first–class objects, and can, like other objects, be passed to functions as arguments and returned as values. Because they are first–class objects no special syntax is needed for functional

* Department of Computer Science, Indiana University, Bloomington, Indiana

application, removing the need for `funcall` [Steele 84]. Procedures are formed using the `lambda` binding operator as in:

```
(define plus-2 (lambda (n) (+ n 2)))
```

and can be applied as in:

```
(plus-2 4)  ⇒  6
```

or even as the anonymous function

```
((lambda (n) (+ n 2)) 4)  ⇒  6 .
```

Evaluation of a lambda expression returns a procedure containing the expressions to be evaluated and the scope at the time the closure is made. The values of any free variables in the expression are found in the enclosing scope. For example, the procedure formed by the `lambda` expression below contains the free variable a bound to 2.

```
(define test (let ([a 2]) (lambda (n) (+ n a)))) .
```

Then invoking `(test 4)` gives 6. If we globally define a as

```
(define a 7),
```

invoking `(test 4)` still gives 6 since the value of a is determined using the closing environment. Thus Scheme can be considered *block structured*, due to the fact that it is *lexically scoped*.

A *thunk* is a procedure of no arguments used to delay evaluation of an expression. So

```
(let ([hold (lambda () exp)]) ··· (hold) ···)
```

is said to "freeze" evaluation of *exp*, complete with the scope at "freezing time". The application `(hold)` "thaws" *exp*, evaluating *exp* at the appropriate time.

2.2 Important Forms and Expressions

The implementation uses the list constructor `cons`, selectors `car` and `cdr`, control primitives `and`, `if`, `when` (one armed `if`), and `map`, and numerical primitives +, −, *, /, >, <, and =. The form `set!` is a side-effect operator analogous to `setq` in most Lisps. The expression

```
(set! id exp)
```

sets the identifier *id* to the value of *exp*. The expression

```
(rec id exp)
```

is primarily used for locally recursive procedures. For example, we can define `factorial` as:

```
(define factorial
   (rec loop
      (lambda (n)
         (if (= n 0) 1 (* n (loop (1- n))))))) .
```

The form `recur`* is a construct used for iteration and local recursion of the form

```
(recur name ([id val] ...) exp ...) ,
```

* Some Scheme systems call this `iterate` or `let` [Clinger 85]

where *name* is bound to the expression and *id* is a local variable initially bound to *val*. Thus another way to define factorial is:

```
(define factorial
   (lambda (num)
      (recur loop ([n num])
         (if (= n 0) 1 (* n (loop (1- n))))))) .
```

Scheme is *properly tail recursive*, meaning that tail recursive function calls are made without growth to the run-time stack. Neither of the definitions of `factorial` above are tail recursive since the multiplication must wait for the result of the recursive call. Using an accumulator `factorial` becomes tail recursive since the last operation performed is the recursive call.

```
(define factorial
   (lambda (num)
      (recur loop ([n num] [accum 1])
         (if (= n 0)
             accum
             (loop (1- n) (* n accum))))))
```

The proper handling of tail recursion is vital since many computations in CM's are infinite loops.

2.3 Objects

Object-oriented programming [Goldberg 83] is characterized by the use of *objects* responding to a set of *messages*. An object is said to have "internal state", containing data representations and related procedures. (From here on the term *object* will refer to items of this type, rather than the more general *scheme-object*.) Procedures in Scheme encapsulate programs and data of an object, allowing the programmer to manipulate these objects without concern for their underlying representations, increasing the security and reliability of the code. Furthermore, changes in the internal representations of an object are opaque to the rest of the program. For these reasons many of the features of SINN are implemented in an object-oriented style.

For example, suppose we need a special register which has associated with it a "weight factor" (this is somewhat like SINN's input connections). We should be able to read from and write to this register, as well as look at its weight and reset the weight to a new value. (The convention used throughout this paper is that *&obj* returns the value of *obj*, *!obj* side-effects *obj*, and **obj* returns a new *obj*.) We would then use expressions like:

```
(define reg1 (*register))             ; create a register rig1
((reg1 'write) .5)                    ; set reg1's weight to .5
((reg1 '!weight) .3)                  ; set its weight to .3
((reg1 'read))                        ; read the value
(* ((reg1 'read)) ((reg1 '&weight))) ; multiply value by weight
```

This syntax may seem cumbersome, and more so when objects are contained in objects. Special forms are introduced later to alleviate some of this inconvenience. A `register` thus responds to the messages `read`, `write`, `&weight` and `!weight`; the messages `write` and `!weight` require one argument, a new value.

The definition of `*register` is seen in Figure 1. Invocation of `(*register)` returns a procedure of one argument, `msg`, in a scope containing `val`, `read`, *etc.* Thus the expression `(reg1 'write)` binds the argument `write` to `msg` in `reg1`, and evaluates the `case` statement, returning the procedure

```
(lambda (v) (set! val v)) .
```

Thus the expression

```
((reg1 'write) .5)
```

invokes the procedure with `v` bound to `.5`, setting `reg1`'s `val` to `0.5`.

```
(define *register
   (lambda ()
      (let ([val 0]
            [weight 0])
         (let ([read-fn (lambda () val)]
               [write-fn (lambda (v) (set! val v))]
               [&weight-fn (lambda () weight)]
               [!weight-fn (lambda (v) (set! weight v))])
            (lambda (msg)
               (case msg
                  [read read-fn]
                  [write write-fn]
                  [&weight &weight-fn]
                  [!weight !weight-fn]))))))
```

Figure 1
Definition of a Register

```
(let ([complete
         (lambda (trap-val ticks-remaining)
            (let ([new-th (trap-handler trap-val)])
               (engine (new-th))))]
      [expire (lambda (new-engine) new-engine)])
   (recur run-loop ([pcb ((ready-queue 'rotate!))])
      (let ([new-engine
               ((&process pcb)            ; run the engine
                (&time-slice pcb)
                complete
                expire)])
         (!process pcb new-engine))       ; store the engine
      (run-loop ((ready-queue rotate!)))))
```

Figure 2
A sample implementation of a scheduler

2.4 Simulation of Parallel Processing

An important feature of SINN is that the nodes of a connectionist network can conceptually be processes computing in parallel. The lack of availability of parallel architectures requires a simulation on a uniprocessor machine. Multi-user, time sharing systems are one example of several processes, or users, computing concurrently. Essentially the computer works with a user for a specified period of time called a *time-slice*. It then interrupts the user to give the other users their time-slices. In the ideal case each user is unaware of the others, and appears to have a single-user computer. How each process is selected for running is determined by a *scheduler*, operating in the *kernel* of an operating system [Haynes & Friedman 84]. This is roughly how concurrency is simulated in SINN.

Some implementations of Scheme provide a facility known as an *engine* [Haynes & Friedman 85], which allows the timed preëmption of the execution of an expression. An engine is a procedure which, when given a quantity of "fuel", evaluates an expression. This returns the value of the expression if there is enough fuel to complete the computation. If the fuel runs out, the computation is interrupted and a new engine is created which can resume the computation at the point of interruption.

Most nodes in a CM perform computations provided to the kernel as thunks. Process Control Blocks (pcb's) contain engines to thaw the thunks and are placed on a "ready queue" to be executed by a round robin scheduler. Code for a scheduler similar to one used in the present kernel is shown in Figure 2. The functions `&process` and `!process` retrieve and store an engine in a pcb, and `&time-slice` returns a pcb's time-slice. The message `rotate!` is sent to the ready queue which returns a pcb while rotating it to the tail of the queue. The scheduler repeatedly runs the engines of the pcb's on the ready queue. An engine is a procedure of three arguments that, when applied to a positive integer n, a two argument *complete* procedure and a one argument *expire* procedure, runs a thunk for n ticks (units of fuel). If the thunk does not return after n ticks (CM computations never terminate), `expire` is applied to a new engine that, when invoked, continues the thunk's computation. In this case `expire` returns the new engine which is stored in the pcb to await its next turn for execution. A computation can only return by requesting a *trap*. If the trap is requested after m ticks, `complete` is applied to a trap value and the number $t = n - m$ ticks remaining. In this case `complete` calls the trap handler, which returns a thunk to resume the computation after the trap, and an engine thawing the thunk is returned. The kernel is a modified version of the one described by Haynes and Friedman [Haynes & Friedman 84]. The interested reader is urged to consult that reference for the implementation details.

3. The SINN System Description

SINN consists of a set of syntactic–extensions (or macros) and procedures that can be used to construct a connectionist model. SINN creates a set of objects containing the model's nodes and connections. The result is an *assembly* containing a set of network nodes, other assemblies, and special nodes used for interaction with the outside world and user.

3.1 Nodes and Connections

Nodes in a network are objects containing collections of input connections called *input classes*, one or more output values, and thunks containing operations that are executed in parallel with the rest of the model. Suppose node N is part of an assembly with input connections to the nodes A through D. A and B are excitatory inputs x_e with weight w_e in class *excit*, while C and D are inhibitory inputs x_i with weight w_i in class *inhib*. Node N updates its output value based on the weighted sum of the values of the connections in its input classes. Let s_e be the sum, $\sum x_e w_e$, of input class *excit*, and s_i be the sum, $\sum x_i w_i$, of input class *inhib*. The output of node N, a, is given by:

$$a = \begin{cases} 1 & \text{if } s_e - s_i > \theta; \\ 0 & \text{otherwise,} \end{cases}$$

where θ is a threshold value. Node N has one output value *output* that is set to the value of a.

Node N is defined in Figure 3 using the syntactic–extension `*-node`. The second argument to `*-node` (for the moment ignore the first argument, the empty list) specifies that N has two input classes `excit` and `inhib`. Connections in a class have initial weights `w-e` and `w-i` respectively. Each input class is a list of input connection objects, so `excit` is a variable bound to the list containing connections to nodes A and B, and similarly for `inhib`. The third argument is a one item list (`([output 0])`). This is an output variable initially bound to 0, and is the local variable `output` in the node. To the outside, `output` is a message sent by any node using N for an input connection. The current convention is that any node used for an input connection must have an output variable named `output`. Other output values may be specified allowing computational flexibility and giving the user the ability to monitor the internal state of a node.

In SINN, an input connection is an object with an initial weight and a path to the output value of the node associated with the connection. In the present example, the path to node A is simply the node's name, although later we will see more complicated paths. If a connection has a path to node A, giving it the message `read` returns the current value of that connection, *i. e.* the value of A's output value, `&weight` returns the weight associated with that connection and `!weight` sets the weight to a new value. Because of the nature of the process that builds the model, the actual connection cannot be "hooked up" until the model is ready for running, so the message `connect` actually establishes the connection. This is, however, a "broadcast message" sent to the connection through nodes and assemblies and need not be given to connections explicitly.

The final argument to `*-node`, called the *code descriptor*, is a Scheme expression specifying the code for execution. The thunk `init-th` is thawed when the node is initialized. The code of `run-th` is executed concurrently with the rest of the model, and is a non–terminating thunk. The thunk computes the weighted

```
(define N
   (*-node ()
      ([excit w-e (A B)]      ; input classes
       [inhib w-i (C D)])
      ([output 0])            ; output value
      (let ([a 0])            ; code descriptor
         (let ([init-th (lambda () (set! a 0))]
               [run-th
                  (rec loop
                     (lambda ()
                        (let ([s-e (class-sum excit)]
                              [s-i (class-sum inhib)])
                           (set! a
                              (if (> (- s-e s-i) theta)
                                  1
                                  0))
                           (set! output a))
                        (loop)))])
            (*-code-list init-th run-th))))


(define class-sum
   (lambda (connect-class)
      (recur loop ([l connect-class])
         (if (null? l)
             0
             (+ (* (((car l) 'read)) (((car l) '&weight)))
                (loop (cdr l)))))))
```

Figure 3
Sample Node Definition

sum of the excitatory and inhibitory input classes using the function class-sum, sets output to the value of a and repeats. The local variable a is really unnecessary but was used to demonstrate local environments and a use for init-th. The function *-code-list builds a list containing init-th and run-th used by *-node to construct the node object.

As objects, each node responds to messages that are the names of its output variables, and must also respond to the messages init, code, and connect. The message init sets the output variables to their initial values and thaws the init-th. The message code returns a list of one or more thunks normally passed to the kernel to be executed in parallel. Each node may run more than one thunk. The message connect transmits this message to all of its input connections.

There are a few other types of nodes available other than the one built by *-node. A feature node is a simple node with no inputs, used primarily as a feature detector in many models. In addition to the messages above, the message set sets the output of that node to a given value. A trace node is used for following the behavior of parts of a model. A trace node is given a delay value to indicate how often to write its data, an output port (to a terminal or a file), and a set of connection paths whose output values are written to the output port.

3.2 Assemblies

Nodes are grouped in collections known as assemblies. A basic assembly is an object containing nodes and other assemblies. Most non–trivial models are defined by means of a top level `assembly`. All types of assembly objects must respond to "broadcast messages", which among other things, are transmitted to all items in the assembly (hence the term "broadcast"). The messages `init` and `connect` are merely transmitted, while `code` returns a list of thunks from each node in the assembly. The example below creates an object `Eg1`, an assembly named `example` (names are used in tracing), containing two nodes A, B and `sub`, a sub–assembly called, appropriately, `sub-assembly`. (the ···'s are node descriptors)

```
(define Eg1
   (assembly example
      ([A ···]
       [B ···]
       [sub (assembly sub-assembly ([P ···] [Q ···]))])))
```

In addition to broadcast messages assemblies respond to the names of the objects they contain. In this case they are the messages `A`, `B`, and `sub`. In fact, the names are variables bound inside the assembly's nodes and sub–assemblies. This is useful when specifying connection paths. Node `A` specifies `B` in a connection class to connect to node `B`, similarly `P` specifies `B` for an input. Node `A` specifies `(sub 'P)` to connect with `P`. The expression

```
((((Eg1 'sub) 'P) 'output))
```

returns the value of `P`'s output.

Suppose a model needed an array of feature detectors. It would be inconvenient to specify each of the feature nodes explicitly if the array is large. A `vector-assembly`, given a number N and a node descriptor, makes N copies of a node. Thus

```
(vector-assembly feature-array 100 (feature))
```

constructs a vector–assembly called `feature-array` of 100 nodes created by the node descriptor `(feature)`. In addition to the standard broadcast messages a `vector-assembly` responds to an integer message i giving access to the i^{th} node in the assembly. If the assembly `Ex2` were defined as

```
(define Ex2
   (assembly example2
      ([A ···]
       [array (vector-assembly feature-array 100 (feature))]))),
```

node `A` can connect to the fifth feature node in `array` by `(array 5)`, and the node can be queried from the outside by

```
((((Ex2 'array) 5) 'output)).
```

Suppose we need to build a model based on the network in Figure 4. There are two copies of an L–node, each connected to a left–vertical–bar and a low–horizontal–bar. This could be specified as a `vector-assembly` except that L_1 and L_2 have different word inputs. We solve the problem with a `multi-assembly` as demonstrated in Figure 5. The `multi-assembly` builds two copies of the nodes `L`, `left-vert-bar`, *etc.* For each node specified, three arguments are required. The first is a set of "unique classes", the second a set of "common classes", and the third a node descriptor.

To see how the `multi-assembly` works, let us examine the connections of each copy of L. Nodes L_1 and L_2 have at least two input classes. `Word-feedback` is called a "unique class", with initial weight `wf-w`. For a `multi-assembly` of size N there must be N lists of connections, one for each copy of the node. Thus L_1 has an input class `word-feedback` containing a list of the single connection `(LOW)`, while L_2's `word-feedback` class contains two input connections `(ALE ALL)`. Common classes are duplicated for each copy of the node, but with an important distinction. The i_{th} copy of a node is connected to the i_{th} member of its common classes. Thus the `features` class of L_1 has

```
((left-vert-bar 1) (low-horiz-bar 1)),
```

while L_2 has the index `1` replaced with `2`. These classes are passed to `*-node` as its first argument. The feature node `left-vert-bar` has no unique or common classes.

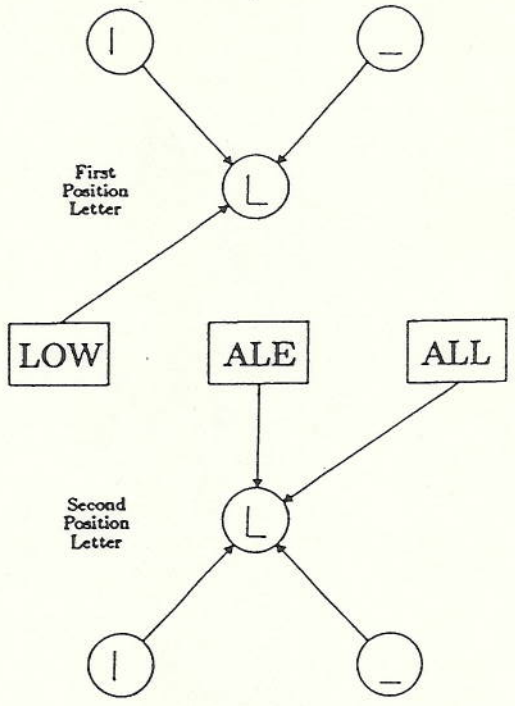

Figure 4
A Word/Letter/Feature Network

```
(define Ex3
   (assembly example3
      ([letters
          (multi-assembly letters 2
             ([L ([word-feedback wf-w (LOW) (ALE ALL)])
                 ([features f-w (left-vert-bar low-horiz-bar)])
                 (*-node (word-feedback features) ···)]
              [left-vert-bar () () (feature)]
              ···))]
       [LOW (*-node ···)]
       ···)))
```

Figure 5
Definition of the Network from Figure 4

4. Examples

Two simplified examples are presented to illustrate the flavor of SINN. Both are adaptations of models presented from the literature. Space permits only the briefest explanations of the models. A few liberties are taken with respect to programming style and efficiency so that the program code of the individual nodes is as close as possible to the form of the actual equations presented in the literature.

4.1 An Associative Search Network

First we consider an associative search network presented by Barto [Barto *et al.* 81], shown in Figure 6. This model consists of a vector Y of adaptive units connected to a vector X of context units. A payoff unit z examines the output from the units in Y to compute a payoff value provided to the adaptive units.

Let $Y = \{y_1, \ldots, y_M\}$ where each y_i is an adaptive unit (node) in the network. Each adaptive unit has an input connection to every context unit in $X = \{x_1, \ldots, x_N\}$, as well as to the payoff unit z. An adaptive unit computes $s(t)$, the weighted sum at time t of the real valued output of its connections to each context

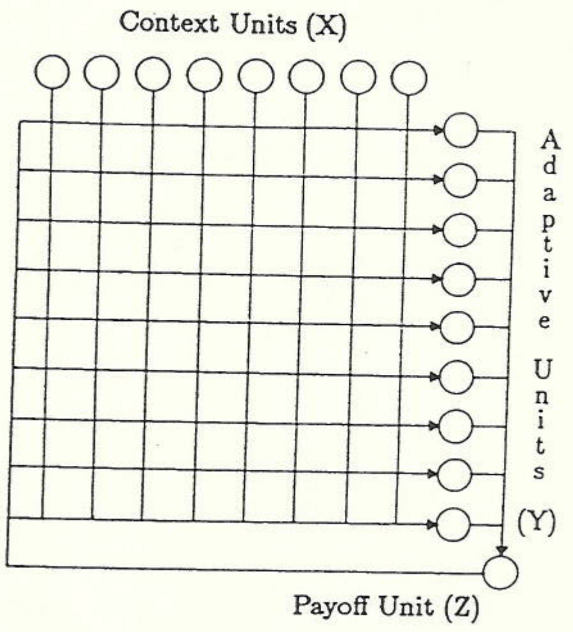

Figure 6
An Associative Search Network

unit by

$$s(t) = \sum_{i=1}^{N} w_i(t)x_i(t),$$

where $w_i(t)$ is the real valued weight at time t associated with the connection to unit x_i. The output of the unit $y \in \{0, 1\}$ is

$$y = \begin{cases} 1 & \text{if } s(t) + \text{NOISE}(t) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

After setting its output, an adaptive unit sets each of its weights w_i according to the equation

$$w_i(t+1) = w_i(t) + c\big[z(t) - z(t-1)\big] \\ \times \big[y(t-1) - y(t-2)\big]x_i(t-1),$$

where c is a real valued learning constant and z is a real valued payoff value. For each context vector $X^\alpha = \{x_1^\alpha, \ldots, x_N^\alpha\}$ providing values for the context units, there is an associated vector $Y^\alpha = \{y_1^\alpha, \ldots, y_M^\alpha\}$, $y_i^\alpha \in \{-1, 1\}$, which represents the optimum output of the adaptive units. The payoff unit computes the value of z according to

$$z = \sum_{i=1}^{M} y_i\big[(y_i^\alpha + 1)/2\big].$$

The necessary code for an adaptive unit in the Associative Search Network from Example 1 [Barto *et al.* 81] is shown in Figure 7. An adaptive unit is formed by giving it three classes of input connections. The first is the x connections to the context units, the second is the payoff line, and the third is the learning constant line. The result is a `*-node` type of node with three input connection classes: the context connections in `x-lines`, the payoff–value in `pay-line`, and the learning constant in `lconst`. In addition to `output`, the output value `connect-list` is bound to `x-list` and is used to examine the weights as they change. The `run-th` contains a loop which grabs the current values of the context–connections (saved later for resetting

```
(adapt-unit x-conns (zc) (lc))  ⇒

(*-node ()
   ([x-lines (init-wt) x-conns]  ; context (x) connections
    [pay-line 1 (zc)]            ; payoff connection
    [lconst 1 (lc)])             ; learning constant
   ([output 0]
    [connect-list x-lines])
   (let ([zt-1 0] [yt-1 0] [yt-2 0] [xt-1-list #!null])
      (let
         ([run-th
             (rec loop
                (lambda ()
                   (let ([xt-list (map (lambda (c) ((c 'read)))
                                       x-lines)]
                         [zt (((car pay-line) 'read))])
                      (set! output
                         (compute-output x-lines))
                      (set-weights! x-lines xt-1-list
                         (((car lconst) 'read)) zt zt-1 yt-1 yt-2)
                      (set! yt-2 yt-1)
                      (set! yt-1 output)
                      (set! zt-1 zt)
                      (set! xt-1-list xt-list))
                   (loop)))]
          [init-th
             (lambda ()
                (set! zt-1 0) (set! yt-1 0) (set! yt-2 0)
                (set! xt-1-list
                   (map (lambda (c) ((c 'read))) x-lines)))])
        (*-code-list init-th code-th))))
```

Figure 7
Definition of an Adaptive Node

the weights), and the payoff value. The procedure compute-output returns a new output value and set-weights! sets the weights of the context connections.

The definitions of the payoff unit, compute-sum and set-weights! are shown in Figure 8. The payoff unit has one input class, y-lines, connected to each the adaptive units. Its computation is a loop that compares the values of the connections on the y-lines to the optimum vector ya-list, provided by the user through the variable **ya-list** and set at initialization time. The delay procedure is used to keep the payoff unit in rough synchronization with the adaptive units, since the loop of the payoff unit executes faster than the loop of the adaptive units. It is synchronized to avoid unnecessary computation and to keep the model in the same "relative time", although it was found that synchronization was not required to reproduce the published behavior of the example.

Below is the definition of the Associative Search Network.

```
(payoff-unit yc-list)  ⇒

(*-node ()
   ([y-lines 1 yc-list])
   ([output 0])
   (let ([ya-list #!null])  ; expected feature vector
      (let ([code-th
               (rec loop
                  (lambda ()
                     (let ([y-list
                              (map (lambda (c) ((c 'read))) y-lines)])
                        (set! output
                           (recur loop ([y y-list] [ya ya-list])
                              (if (and y ya)
                                  (+ (* (car y) (/ (1+ (car ya)) 2))
                                     (loop (cdr y) (cdr ya)))
                                  0))))
                     (delay **env-delay**)
                     (loop)))]
            [init-th
               (lambda () (set! ya-list **ya-list**))])
         (*-code-list init-th code-th))))


(define compute-output
   (lambda (connect-list)
      (let ([sum (class-sum connect-list)])
         (if (> (+ sum (noise)) 0)
             1
             0))))

(define set-weights!
   (lambda (connect-list xt-1-list learn-const zt zt-1 yt-1 yt-2)
      (let ([const
               (* learn-const
                  (* (- zt zt-1) (- yt-1 yt-2)))])
         (recur loop ([cl connect-list] [xl xt-1-list])
            (when cl
               (((car cl) '!weight)
                (+ (((car cl) '&weight)) (* const (car xl))))
               (loop (cdr cl) (cdr xl)))))))
```

Figure 8
Definitions of `payoff--unit`, `compute-output` *and* `set-weights!`

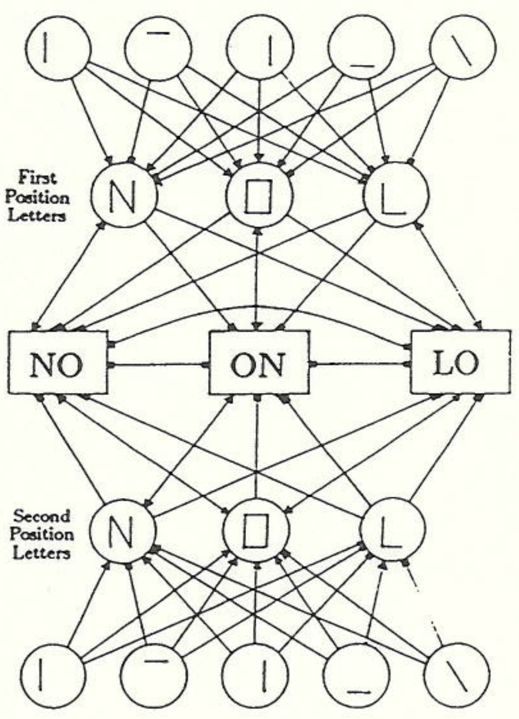

Figure 9
A Simple McClelland–Rumelhart Model

```
(define ASN
   (assembly asn
      ([y (vector-assembly y 9
             (adapt-unit ($r (env 'x) 1 8) ((env 'z)) (c)))]
       [env (assembly environment
                ([x (vector-assembly x 8 (feature))]
                 [z (payoff-unit ($r y 1 9))]))]
       [c (feature)]))) .
```

The new construct `($r (env 'x) 1 8)` is used to specify the range of the connections to `x` units and to the `y` units. This gives a list of connection paths

```
(((env 'x) 1) ··· ((env 'x) 8)) .
```

The learning constant `c` is a feature node, enabling the user or other outside agent to set the value of the learning constant while the model is running.

4.2 A Version of the McClelland–Rumelhart Model

A highly simplified version of a McClelland–Rumelhart model of letter recognition is shown in Figure 9 [McClelland & Rumelhart 81; Rumelhart & McClelland 82; McClelland 85]. The network consists of three word nodes connected to two copies of a letter and feature assembly with three letter and five feature nodes. The types of features are simplified versions of those from the original model.

The model is based on McClelland and Rumelhart's description [McClelland & Rumelhart 81]. Each non-feature node of the network computes the weighted sum of its input connections. The net input to the unit at time t is given by

$$n(t) = \sum_j \alpha_j x_j(t),$$

```
(mr-node (outsides ...) ([locals weight paths] ...))  ⇒

(*-node (outsides ...)
   ([locals weight paths] ...)
   ([output 0])
   (let ([act 0])
      (let ([init-th (lambda () (set! act 0))]
            [run-th
               (rec loop
                  (lambda ()
                     (set! act
                        (new-act act
                           (effect act
                              (+ (connect-sum outsides ...)
                                 (connect-sum locals ...)))))
                     (set! output act)
                     (loop)))])
         (*-code-list init-th run-th))))


(define *-act-fn
   (lambda (theta rest-i)
      (lambda (act effect)
         (+ (- act (* theta (- act rest-i)))
            effect))))

(define *-effect-fn
   (lambda (max min)
      (lambda (act sum)
         (* sum
            (cond [(> sum 0) (- max act)]
                  [(< sum 0) (- act min)]
                  [else 0])))))

(define new-act
   (*-act-fn .07 0))

(define effect
   (*-effect-fn 1 -.20))
```

Figure 10

Functions Used for the Network in Figure 9

where α_j is the weight constant associated with the connection x_j. In this implementation no distinction is made between excitatory and inhibitory connections. An excitatory connection merely has a positive weight while an inhibitory connection has a negative weight. The effect, $\epsilon_i(t)$ on the unit i at time t is given by

$$\epsilon_i(t) = \begin{cases} n_i(t)\,(M - a_i(t)) & \text{if } n_i(t) > 0, \\ n_i(t)\,(a_i(t) - m) & \text{if } n_i(t) < 0, \end{cases}$$

where $a_i(t)$ is the activation of the unit i at time t, M is the maximum activation and m is the minimum activation of the unit. The activation $a_i(t + \Delta t)$ of the node i at time $t + \Delta t$ is

$$a_i(t + \Delta t) = a_i(t) - \theta_i\,(a_i(t) - r_i) + \epsilon_i(t),$$

where θ_i is a decay constant and r_i is the resting activation of node i.

```
(define smallmr
   (assembly smallmr
      ([lf (multi-assembly feat&lets 2
              ([N ([wordex .3 (NO) (ON)])
                  ([featex .005 (sl rvbar lvbar)]
                   [featin -.15 (hibar lobar)])
                  (mr-node (wordex featex featin) ())]
               [O ([wordex .3 (ON) (NO LO)])
                  ([featex .005 (rvbar lvbar hibar lobar)]
                   [featin -.15 (sl)])
                  (mr-node (wordex featex featin) ())]
               [L ([wordex .3 (LO) ()])
                  ([featex .005 (lvbar lobar)]
                   [featin -.15 (sl hibar rvbar)])
                  (mr-node (wordex featex featin) ())]
               [sl () () (feature)] [rvbar () () (feature)]
               [lvbar () () (feature)] [hibar () () (feature)]
               [lobar () () (feature)]
               ))]
       [NO (mr-node ()
              ([letex .07 ((^  lf N 1) (^  lf O 2))]
               [letin -.04 ((^  lf O 1) (^  lf N 2)
                            (^  lf L 1) (^  lf L 2))]
               [wordin -.21 (ON LO)]))]
       [ON (mr-node ()
              ([letex .07 ((^  lf O 1) (^  lf N 2))]
               [letin -.04 ((^  lf L 1) (^  lf L 2)
                            (^  lf N 1) (^  lf O 2))]
               [wordin -.21 (LO NO)]))]
       [LO (mr-node ()
              ([letex .07 ((^  lf L 1) (^  lf O 2))]
               [letin -.04 ((^  lf O 1) (^  lf L 2)
                            (^  lf N 0) (^  lf N 2))]
               [wordin -.21 (NO ON)]))]
       [trword (trace-node wport 1 NO ON LO)]
       [trlet1 (trace-node l1port 1 (^  lf L 1) (^  lf N 1)
                  (^  lf O 1))]
       [trlet2 (trace-node l2port 1 (^  lf L 1) (^  lf N 2)
                  (^  lf O 2))])))
```

Figure 11
Definition of the Network from Figure 9

The definition of the standard node of the model and its related procedures are shown in Figure 10. The node is passed the outside input class names (see the `multi-assembly`), and descriptions for each of its local input classes. The code that is run is a loop that updates its output, its activity, based on the activities of its inputs. `Connect-sum` computes the weighted sum of a series of input classes using `class-sum`. Due to the nature of the syntactic–extension package used, it computes the sums of the local and outside classes separately.

The complete description of the example network is shown in Figure 11. A `multi-assembly` is used to copy the feature and letter network, with different input connections for each copy of the letter nodes. The expression

```
(^ a b c d)  ⇒  (((a 'b) 'c) 'd)
```

is a new syntax for connection paths and serves to clean up the message passing syntax for objects. Trace nodes provide output of the activation of the words and the first and second letters to ports `wport`, `l1port`, and `l2port` respectively.

5. Discussion

Both models performed in accordance with their published behavior. The Associative Search Network (ASN) was run with two orthogonal context vectors from Example 1 [Barto *et al.* 81]. The simple letter recognition model was run with undegraded and degraded stimuli similar to McClelland and Rumelhart's first examples [McClelland & Rumelhart 81], and although the model was highly simplified the performance was as predicted. Although the ASN was described as a "discrete time model" [Barto *et al.* 81] the payoff unit was given different delay values allowing its "relative time" to vary widely with respect to the adaptive units with little effect on performance.

Time, here, must be called "relative" since this is a simulation of concurrency. A unit of relative time roughly corresponds to one complete cycle through the kernel's ready queue. This "time" depends, among other things, on the number of processes in the system, the time–slice of each process, and the exact meaning of a "tick" to an engine. Real time behavior of each model depends not only on this notion of relative time but also on the nature and current state of the host machine and programming language. Scheme [Dybvig 87], is a native code compiler and is quite adequate for the present implementation. For example, the ten learning cycles required for each context vector in the ASN were completed in less than a second, with much of this time being overhead associated with starting the system. The letter recognition model tended to reach equilibrium in about ten seconds. The presence of trace nodes tends to slow the execution since performance must depend on the speed of the output operations.

6. Extensions and Improvements

SINN is an ongoing concern, and is intended for use in connectionist models of learning. As more features are needed they will be incorporated into the system. As models grow large the specification of connections will become a tedious and error–prone operation, and a better syntax is needed to keep these specifications as concise as possible. Different node types may also need to be implemented. One example is the implementational metaphor of a neuron whose axon has synapses along its entire length, with a time delay affecting the arrival of output at each synapse. Different types of assemblies are also foreseen, with assemblies such as mutually interconnected networks (i. e. each node is automatically connected with every other node in the assembly) being implemented as the need arises.

As models grow larger, efficiency will become an issue. The implementation of objects will need to be improved to relieve space problems. The kernel, being the heart of the implementation, will need to be optimized. The compilation process is currently the most expensive and steps can be taken to speed up this stage.

Finally, trace nodes are a prototype of a more general interface to the outside world. The development of a graphics package that can be interfaced to the system is needed to improve the study of the performance of a model.

7. Conclusion

SINN supplies a set of tools for the concise and elegant implementation of connectionist models. These tools provide freedom from many of the details of the programming language, allowing concentration on the specification of the design and interactions of the model. This reduces the time needed to implement a model. The abstraction mechanisms of Scheme lead to the convenient implementation of programming metaphors such as distributed processing and object–oriented programming. This greatly facilitates the continued development of SINN as a powerful tool for the study of connectionist models.

8. References

[Barto *et al.* 81] Barto, A. G., Sutton, R. S., Brouwer, P. S., "Associative Search Networks: A reinforcement learning associative memory", *Biological Cybernetics* **40**, 201–211, 1981.

[Clinger 85] Clinger, W. C., "The revised revised report on SCHEME" Indiana University Computer Science Department Technical Report No. 174, June, 1985.

[Dybvig 87] Dybvig, R. K., *The Scheme Programming Language,* Prentice-Hall 1987, (in press).

[Feldman & Ballard 82] Feldman, J. A., and Ballard, D. H., "Connectionist models and their properties", *Cognitive Science* **6**, 205–254, 1982.

[Filman & Friedman 84] Filman, R. E., and Friedman, D. P., *Coordinated Computing: Tools and Techniques for Distributed Software*, McGraw-Hill, 1984.

[Goldberg 83] Goldberg, A., and Robson, D., *Smalltalk–80: The Language and its Implementation*, Addison-Wesley, 1983.

[Haynes & Friedman 85] Haynes, C. T., and Friedman, D. P., "Abstracting timed preemption with engines" *Journal of Computer Languages*, **12**, 109–121, 1987.

[Hillis 85] Hillis, W. D., *The Connection Machine*, MIT, 1985.

[Hinton & Anderson 81] Hinton, G. E., and Anderson, J. A., *Parallel Models of Associative Memory*, Erlbaum, 1981.

[McClelland & Rumelhart 81] McClelland, J. L., and Rumelhart, D. E., "An interactive activation model of context effects in letter recognition: Part 1. An account of basic findings", *Psychological Review* **88**, 375–407 1981.

[McClelland 85] McClelland, J. L., "Putting knowledge in its place: A scheme for programming parallel processing structures on the fly", *Cognitive Science* **9**, 113–146 1985.

[Rumelhart & McClelland 82] Rumelhart, D. E., and McClelland, J. L., "An interactive activation model of context effects in letter recognition: Part 2: The contextual enhancement effect and some tests and extensions of the model", *Psychological Review* **89**, 60–94 1982.

[Steele 84] Steele, G. L., *Common Lisp: The Language*, Digital Press, 1984.