A Framework for Neural Net Specification

Leslie S. Smith

Abstract— **A notation for the specification of neural nets is proposed. The aim is to produce a simple mathematical framework for use in specifying neural nets essentially by defining their transfer functions and connections. Nets are specified as interacting processing elements (nodes), communicating via instant links. Dynamics and adaptation are defined at the processing elements themselves, and all interaction is explicitly specified by directed arcs. Specifications can be built up hierarchically by turning a specification into a generator for a node, or they can be developed top-down. The use of the system is illustrated.**

Index Terms—**Connectionist models, formal models, net specifications, neural net design system.**

I. Introduction

MOST neural nets are specified by describing the processing elements, the topology (including which units are used for input and which for output), the adaptation rule (if any), and the net update dynamics. The notation used for the specification is usually informal, perhaps naming the learning rule, either naming the topology, or defining it by means of a matrix of connectivity, and describing the neurons themselves using equations that implicitly define the net update dynamics. This is then implemented, perhaps using a package, or by writing a program. While this suffices for experimental work, developing, for example, learning rules, it makes the description of a complex net, perhaps made up of a number of subnets, with a number of different types of processing units, and different interactions between these different elements rather difficult. This work aims to provide a notation for neural nets, a notation with enough generality to be useful for many different types of net. The notation is primarily symbolic, but it can be usefully illustrated graphically.

One would like to produce a hierarchical specification technique that describes what the net should do, then allows this to be broken down into manageable pieces, in line with top-down design for computer programs. However, the science of neural network design is not yet advanced enough to allow nets to be formally designed from their functional specifications. The notation does allow nets to be described in a top-down fashion (see Section 6.B), but is not (and is not likely to be) executable. It is not a functional specification, but a description of a network. It is intended primarily as an aid to description and clarification of neural net specification. Further decomposition can be achieved by replacing a network node by a network. The notation describes a set of generators that generate all the elements of the net. The interaction between the elements of the net is via instantaneous connections (rather like lines on a circuit diagram), so that the elements generated must embody all the nontopological specification of the net. This includes both dynamics and adaptation. Generators themselves are skeletal outlines of elements that are turned into elements by a two-stage process that uses parameters to precisely define the operation of the element. The system is made hierarchical by providing a method for forming a new (skeletal) generator from a complete network, thus allowing the specification of networks of networks. Not tying the framework to any implementation means that it is not executable; however, it means that the range of networks that can be described is very broad.

The most influential basis for specification systems for neural networks has been the Actor concept [1]. This has been used by the Pygmalion project [2], [3] and in the commercial system ANSpec [4]. Actors are a process-oriented tool intended for the specification of parallel processes. As such, they have the descriptive power to describe the topology, and use usual programming techniques to describe the input/output relationship. However, they have problems in describing the dynamics of nonsynchronous neural net systems. The AXON language [5] is a C-like language for neural net description. It is a powerful low-level tool: as in the notation described here, the dynamics and the adaptation are described in the (program) specification for the nodes. However, it is not a specification technique as such, but is simply a C-like program. The same comments apply to MENTAL [6]. Perhaps the closest to our notation is that of [7], which is based on CSP [8]. It was felt that CSP limited the temporal specification of the interchange of information: we wanted a specification system that could cope with synchronous, asynchronous, and continuous time systems. This really ruled out any of the existing parallel processing formalisms. This work is very loosely based on CCS [9], at least so far as the graphical form is concerned.

The notation can be used to describe low-level (i.e., biologically realistic) systems. Compartment based simulators, such as SWIM and Saber ([10], [11]), are becoming important at the interface between those interested in neural networks for their computational properties, and those interested in the neurophysiology. SPICE [12], an analog circuit simulator, can be used for membrane patch simulation, approximating the biology by analog circuit elements. The notation can be used to describe networks at a mixture of levels, and could be used to supply a specification for a simulation, possibly of some critical part of a network, using these tools. The notation allows a network to be presented as a whole, and its use of generators makes it a powerful specification tool by permitting

Manuscript received July 25, 1991; revised March 12, 1992. Recommended by E. Gelenbe.

The author is with the Centre for Cognitive and Computational Neuroscience, Department of Computing Science, University of Stirling, Stirling FK9 4LA Scotland.

IEEE Log Number 9200609.

0162-8828/92$03.00 © 1992 IEEE

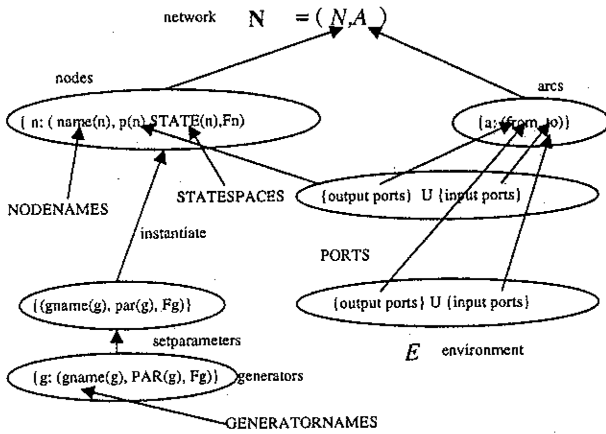

Fig. 1. A graphical representation of the framework. See text for details.

re-use of already specified types of nodes, as well as the hierarchical combination of nodes into one node.

Both the net's dynamics and the way in which the net adapts are described in the notation for the network nodes. This work is restricted to the specification of static nets: dynamic nets (both constructive and decreasing nets) are outside the scope of the project. Section II of the paper describes the framework itself, and Section III gives a simple example. Section IV describes some different types of dynamics that the framework can support, and discusses pulse-based networks. Adaptation is discussed in Section V, and Section VI gives a bottom-up and a top-down example of the framework's usage. Section VII draws some conclusions, and discusses some possible further work.

II. The Framework

Informally, a network consists of *nodes* and directed *arcs* connecting pairs of *ports* each associated with a node. The network exists within an *environment*, which is treated as a special case of a node. With the exception of the environment, nodes are described by instantiation of a *generator*. Fig. 1 illustrates how the environment and the net are made up.

More formally, we start from a number of sets, from which we will build up the notation. These sets follow.

NODENAMES: the set of names for nodes (e.g., strings of bounded length).

GENERATORNAMES: the set of names for generators.

PORTS: the set of ports on nodes.

STATE-SPACES: the set of possible state spaces for nodes.

FUNCTIONS: the set of functions that a node may implement.

The set PORTS consists of two disjoint subsets, INPORTS and OUTPORTS, corresponding to the direction of information flow at that port.

All input to and output from the network is from or to the environment, E. The environment has ports $p(E) \subset$ PORTS, and this set consists of input ports $p_{in}(E) \subset$ INPORTS and output ports $p_{out}(E) \subset$ OUTPORTS. Nothing else about the environment is defined.

We now define generators and nodes. Logically, one should start with the generators, since nodes are generated from these. However, in a network, the fundamental entities are the nodes: though the generator concept allows the construction of a number of nodes from a common startpoint, as well as allowing hierarchical net construction, it is secondary. We therefore start with the node. It is defined as a tuple:

$$n \in \text{NODENAMES} \times set(\text{PORTS}) \times \text{STATE-SPACES} \times \text{FUNCTIONS} \qquad (1)$$

where set(PORTS) is the set of subsets of PORTS. We write

$$n = (\text{name}(n), p(n), \text{STATE}(n), F_n) \qquad (2)$$

where name(n) defines the node's name, and $p(n) = p_{in}(n) \cup p_{out}(n)$, as with the environment, defines the set of ports. We will write $p_{in}^i(n)$ or $p_{out}^i(n)$ for each port. STATE(n) is the set of possible or reachable states for node n. F_n defines the function of the node, that is, how the outputs (i.e., values placed on output ports) are computed, and how the state updates. Where it is useful, we separate out the state update part of F_n, calling it F_n^{int}, from the port output part, calling it F_n^{out}.

Nodes exist over time; indeed, they may evolve over time. The name of the node and the ports do not change, but the state, the values on the ports, and possibly the function do change. We write state$(n,t) \in$ STATE(n) for the node state at time $t \geq 0$, state$(n, 0)$ being the node's initial state, and we write $p_{in}^i(n,t)$ (or $p_{out}^i(n,t)$) for the value on port $p_{in}^i(n)$ (or $p_{out}^i(n)$) at time t. Where the node to which the port belongs is clear, we drop the parameter n. Since the function F_n may adapt, we write $F_{n,t}$ for the function implemented by the node at time t. This defines the values to be placed on the $p_{out}(n)$ ports at time t, and how the state is updated at time t. This will depend on the initial function, $F_{n,0}$, on what has been received on the $p_{in}(n)$ ports up to time t, and on the initial state of the node.

Arcs are defined as pairs:

$$a \in \text{OUTPORTS} \times \text{INPORTS}. \qquad (3)$$

Thus each directed arc a joins one output port to one input port. Thus if $a = (f_a, t_a)$, writing N for the set of nodes,

$$f_a \in \bigcup_{r \in N \cup E} p_{out}(r)$$
$$t_a \in \bigcup_{r \in N \cup E} p_{in}(r).$$

Writing A for the set of arcs, we can characterize the net, $\boldsymbol{N}$, itself as

$$\boldsymbol{N} = (N, A) \qquad (4)$$

Each node, n, is generated from a generator, g. The aim of introducing this secondary entity is twofold: firstly, it provides a common startpoint for a number of nodes, and secondly, by providing a method for producing a generator from a complete network, it allows networks to be built up hierarchically. The generator must be able to define the name, ports, and state-space of the node, the initial state and initial function, and how

these will evolve in time. We have taken a parameter based approach: the generator provides a template for the name and function of the node, and these are precisely defined using parameters. Each g is a triple,

$$g = (\text{gname}(g), \text{PAR}(g), F_g) \quad (5)$$

where $\text{gname}(g) \in$ GENERATORNAMES, $\text{PAR}(g)$ is the parameter space for this generator, and F_g is a template for the function F_n. Again, where this is useful, we may split this into F_g^{int} and F_g^{out} as for F_n. This is not completely general; however, the functions used in nodes are usually relatively simple, and generally fall into a small number of classes. We will write G for the set of generators. The parameter space is used to specify all the other things that need to be specified: that is, how name(n) is derived from gname(g), what the set $p(n)$ should be, what STATE(n) should be, how F_n should be derived from F_g, what state $(n, 0)$ and $F_{n,0}$ should be, and how they should evolve in time. This is accomplished in two steps: firstly the actual parameters of g are set:

$$\text{setparameters}(g) = (\text{gname}(g), \text{par}(g), F_g) \quad (6)$$

(simply choosing $\text{par}(g) \in \text{PAR}(g)$) and then the generator with its parameters set is instantiated:

$$\begin{aligned}&\text{instantiate}(\text{gname}(g), \text{par}(g), F_g) = n\\ &= (\text{name}(n), p(n), \text{STATE}(n), F_n).\end{aligned} \quad (7)$$

This defines the name of the node n, its ports, its state-space, and function. It also implicitly sets up state $(n, 0)$ and $F_{n,0}$ since these are defined by the selection of the parameters. We have not precisely formalized the generation of STATE(n) from the parameters. In general, we will use a subspace of PAR(g), one spanned by some of the parameters. The initial state, state $(n, 0)$, will be defined by the actual value of these parameters.

The framework allows a formalization of the notion of the type of a node: two nodes have the same type if they share a common generator. The inverse of instantiate $\circ$ setparameters, desc, can be defined, mapping the set of nodes to the set of generators:

$$\text{desc} : N \to G; \text{desc}(n) = g \quad (8)$$

where $n = \text{instantiate} \circ \text{setparameters}(g)$. The mapping desc is well defined since every node n has an associated generator g. The set N can be partitioned under the equivalence relation

$$n_1 \sim n_2 \Leftrightarrow \text{desc}(n_1) = \text{desc}(n_2). \quad (9)$$

It is possible to define nets hierarchically by forming a generator from a whole net, and then instantiating this as a node. Considering the net

$$\boldsymbol{N} = (N, A)$$

we need to produce a generator for the new node that will replace $\boldsymbol{N}$:

$$g = g(\boldsymbol{N}) = (\text{gname}(g(\boldsymbol{N})), \text{PAR}(g(\boldsymbol{N})), F_{g(\boldsymbol{N})}). \quad (10)$$

To produce the generator $g(\boldsymbol{N})$ entails selecting a new name gname($g(\boldsymbol{N})$), defining the parameter space PAR($g(\boldsymbol{N})$), and defining a template function $F_{g(\boldsymbol{N})}$. The name can be chosen from GENERATORNAMES, but the parameter space and template function must be constructed. The parameter space can be constructed (for example) by considering its elements to have the form

$$(s_0, s_1, \ldots, s_r) : s_i \in S_i$$

where S_i is the range of parameters for the ith parameter. We can use s_0 and s_1 to define the ports. These come from the arcs of the original net, specifically, from the subsets of A:

$$\begin{aligned}A_O &= \{a \in A : t_a \in p_{in}(E)\} \text{ and}\\ A_I &= \{a \in A : f_a \in p_{out}(E)\}\end{aligned} \quad (11)$$

which are the arcs between the net $\boldsymbol{N}$ and its environment E. Their endpoints in $\boldsymbol{N}$ (i.e., f_a in A_O and t_a in A_I) will generate the ports of the node. Let s_0 parameterize the input ports, and s_1 the output ports. s_0 can be used to define how many input ports will be generated from each t_a in A_I, and s_1 how many output ports will be defined from each f_a in A_O. Note that some of the t_a and f_a may be left unused, while others may give rise to several ports. The parameters $s_2, \ldots, s_r$ parameterize everything we wish to be externally visible (i.e., parameterizable) from the parameters used in the construction of all the nodes of the original net. $F_{g(\boldsymbol{N})}$ is defined implicitly by the F_n where $n \in N$. Working in this way, the initial state of the new node and the initial function may be determined by the initial states and initial functions of all the nodes in the original network, or the parameterization used in setting up these states in the original nodes can be re-used in the setting up of the new node. This construction can be nested to any desired level; however, circular definitions must be avoided so that eventually all the nodes are defined in terms of elementary generators.

A can be considered a directed graph on the nodes N. Thus, there is a path from n_1 to n_2 if there is a path $P \subseteq A$:

$$P = \{(f_r, t_r) : r = 0 \ldots R-1\} \quad (12)$$

where $f_0 \in p_{out}(n_1)$, $t_{R-1} \in p_{in}(n_2)$, and $t_i \in p_{in}(n_i)$, $f_{i+1} \in p_{out}(n_i)$ for $i = 0 \ldots R-2$. Given that the n_i are distinct, the path is of length R. Thus, A has loops (i.e. $\boldsymbol{N}$ is recurrent) if for some $n \in N$, there is a path from n to itself.

This framework can be applied to many different types of interacting entities: clearly it is the form of the elements (as defined by the g, and the mappings instantiate and setparameters) and the pattern of interconnections that make the framework produce something which is recognizably a neural net. Thus, for example, we sometimes identify part of the internal state with the weights of a neuron, or with an activation level.

The restriction that each port may be an endpoint of at most one arc seems odd at first: we are used to axonic outputs going to many other neurons. However, the arcs do not represent either axonic or dendritic links; they are simply instant communication paths. All the active elements including synapses will be contained in the nodes of the net. This restriction allows us to make all the links identical whereas

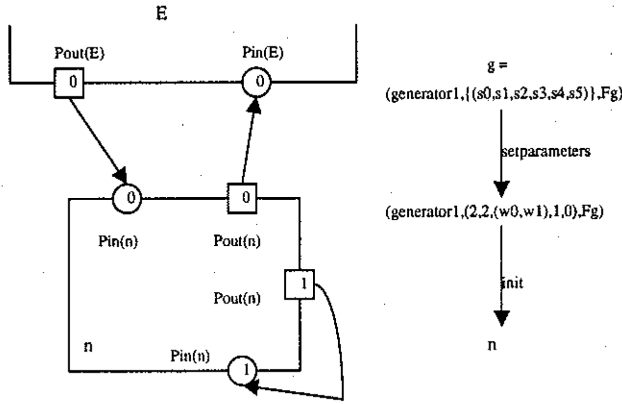

Fig. 2. A simple 1-neuron net. Small squares are output ports, and small circles input ports.

alternative approaches would require us to characterize links as well as nodes. This approach permits both subdivision of each neural element (so that nodes may be parts or compartments of a neuron) and clustering of neural elements (so that nodes may be networks of neurons).

III. A Simple Example

We illustrate the framework by specifying a simple 1-neuron net, as shown in Fig. 2. The environment, E, has two ports:

$$p_{in}(E) = \{p^1_{in}(E)\}$$
$$p_{out}(E) = \{p^1_{out}(E).\}$$

The net itself has one node, n, and this is generated from a generator g. To characterize g, we must supply its name, the parameter space, and the (skeletal) function F_g. We choose these to be:

$$\begin{aligned} \text{gname}(g) &= \text{ generator1} \quad (13)\\ \text{PAR}(g) &= \{(s_0, s_1, s_2, s_3, s_{init}) : \\ &\qquad s_0, s_1 \in \mathcal{I}_0, s_2 \in \mathcal{R}^{s_0}, s_3 \in \mathcal{R}, s_{init} \in \mathcal{R}\} \\ F_g(s_3, s_2, s_0, t) &= 1/(1 + \exp -s_3(\sum_{i=0}^{s_0-1} s_2^i p^i_{in}(t-1))) \end{aligned}$$

where $\mathcal{I}_0$ is the set of nonnegative integers, $\mathcal{R}$ is the set of real numbers, and $p^i_{in}(t-1)$ is the value on the ith input port of the instantiated neuron at time $t-1$. Thus, generator1 will generate nodes that accumulate a weighted set of inputs (where the number of inputs is defined by s_0, and the weights by s_2) and then compute its outputs by applying a logistic function, whose steepness is determined by s_3. The number of output ports (all in this case transmitting the same value) is determined by s_1. The initial state of the node is defined by s_{init} (this is the value at the output ports at $t = 0$), and by the s_2.

Letting setparameters set $s_0 = 2$, $s_1 = 2$, $s_2 = (w_0, w_1)$, $s_3 = 1$, and $s_{init} = 0$ and letting instantiate set

$$\text{name}(n) = \text{generator1.instance1}$$

and set up state$(n, 0)$ from s_{init} and s_2 as above, the node of the net is produced. For this very simple example, the internal state is constant so that F_n is simply F_g with the actual parameters substituted. For the arcs, if we write $p^i_{out}(n)$ for the ith output port of n, and $p^i_{in}(n)$ for the ith input port of N, we can write

$$A = \{(p^0_{out}(E), p^0_{in}(n)), (p^1_{out}(n), p^1_{in}(n)), (p^0_{out}(n), p^0_{in}(E))\}$$

to give the whole net. That the net is recurrent is clear, since the second arc in A represents a path of length 1 from node n to itself.

In general we need to decide the extent to which PAR(g) and F_g can allow different nodes to share generators: should each g be very restricted in the nodes it can generate, or should we have only a small number of different generators. It is useful to be able to use the same generators in many nets. In this particular case, there is only one node, so that we could ignore the question here; however, we use this simple case to illustrate the point: the generator generator1 can generate a large variety of simple nodes that use the logistic function.

Virtually all nodes require the definition of an initial state; this amounts to defining initial boundary conditions for the model. However, for the sake of clarity, we often omit the parameter s_{init} from further discussions.

IV. Dynamics

The term dynamics applied to nets describes how the net functions over time. The framework does not define this directly, leaving it to the definition of the functions F_n. These define the behavior of the nodes over time, and the whole net's behavior over time emerges from this. Although one can characterize the behavior of some nets (e.g., feedforward nets) by simply considering that values on output ports are computed after all the inputs to that node have arrived, proper characterization of nets with feedback entails more precise definitions.

One way to achieve this is to consider time as a sequence of integers, starting at 0. Thus, the environment is considered to produce its input at time 0, 1, 2, etc., and the net's output occurs at time 0, 1, 2, etc. In this case, the nodes all receive their input from their input arcs at time t, and produce output and place it on the output arcs at time $t+1$. This is synchronous dynamics and was used in the example in the previous section. Each value output at time t is accessible at the corresponding input port immediately, and usable in computing outputs at the next timestep. For internal arcs, we need to define the values to be used as input at time 0. Nets using synchronous dynamics can oscillate if they are recurrent. For feedforward nets (where we are only interested in the final stable state) one needs to wait for the net to settle. Given a simple generator for the nodes (such as generator1 of (13)), the net will settle in a number of timesteps equal to the maximum length of a path from an environment output port to an environment input port.

Difficulties arise with this synchronous approach if one wishes to compose a number of nodes and consider this subnetwork as a single node, or decompose a node into a subnetwork. The problem is not too difficult in the first case, as one can arrange for suitable delays at the new node; but in the second case, it may be impossible to define the operation

of the new nodes using the same clock in such a way that the decomposition has no functional effect. This is a real problem, since we may wish to model neurons at differing degrees of detail in different parts of a net. One possible approach is that taken in LUSTRE [13] in which different clocks are defined in different parts of the system, and an additional signal qualifying the validity of each output is introduced; however this has not been investigated here.

These problems of synchronizing networks in which different parts are specified at differing levels of detail can be simplified by using continuous dynamics. Biologically, this is more realistic. The specification problem is simplified since the new nodes operation over continuous time must be specified. Ensuring that the subnetwork has the correct temporal operation becomes a question of the correct specification of the nodes of the subnetwork. Low-level operation of a real neuron is not synchronous, but continuous, being governed by, for example, ionic gradients. This type of operation can be modeled by passing values continuously through the links. A different choice of generator in the earlier example would give this dynamics:

$$\text{gname}(g) = \text{ generator2}$$

$$\text{PAR}(g) = \{(s_0, s_1, s_2, s_3) : s_0, s_1 \in \mathcal{I}_0, s_2 \in \mathcal{R}^{s_0}, s_3 \in \mathcal{R}\}$$

$$F_g(s_3, s_2, s_0, t) = 1/\left(1 + \exp -s_3\left(\sum_{i=0}^{s_0-1} s_2^i \int_0^t W(t-\tau)p_{in}^i(\tau)d\tau\right)\right) \quad (14)$$

where $W(x)$ is a convolving function with integral 1, describing the operation over time of a synapse in response to an input. It will be nonzero only over some small positive range. The output of such a unit is a time-varying signal in the range 0 to 1, compressing the possible range of the sum of the integrated inputs. By making $W(x) = \delta(x-1)$ (where $\delta()$ is the Dirac delta function), this reduces to synchronous dynamics.

Neurons whose output takes the form of impulses can use part of the F_g of (14) as the basis for an internal state. The output will be an impulse of some shape: in this generator, the internal state is reset when the impulse is produced.

$$\text{gname}(g) = \text{ generator3}$$

$$\text{PAR}(g) = \{(s_0, s_1, s_2, s_4, l_{pulse}, \text{pulsetype}) : s_0, s_1, \text{pulsetype} \in \mathcal{I}_0, s_2 \in \mathcal{R}^{s_0}, s_4, l_{pulse} \in \mathcal{R}\}$$

$$F_g^{int}(t) = F_g^{int}(s_2, s_0, t) = \sum_{i=0}^{s_0-1} s_2^i \int_{T_{reset}}^t W(t-\tau)p_{in}^i(\tau)d\tau \quad (15)$$

$$\begin{aligned} F_g^{ext}(l_{pulse}, \text{pulsetype}, s_4, F_g^{int}(t), t) &= 0 \text{ if } F_g^{int}(t) < s_4 \\ &= \text{pulse}(\text{pulsetype}, l_{pulse}, t) \text{ if } F_g^{int}(t) \geq s_4 \end{aligned} \quad (16)$$

where T_{reset} is set to the start of each output pulse, and the function pulse(pulsetype, l_{pulse}, t) produces a pulse whose form is dependent on pulsetype, whose length depends on l_{pulse}, and which starts at time t. Such an element convolves the received impulses with the function $W(x)$, thus modeling actual operation of the synapse, and adding some biological plausibility. Producing an efficient simulation on a digital computer based on this type of dynamics is very difficult; however, the equations might well describe the operation of a system based on an analog computer, or a system where the neural elements are built from analog components.

One can reduce the computational overhead of this pulse-based generator by simplifying (15) and (16) so that the shape and duration of the pulse are ignored, and the pulses simply weighted and counted:

$$\text{gname}(g) = \text{ generator4}$$

$$\text{PAR}(g) = \{(s_0, s_1, s_2, s_4,) : s_0, s_1 \in \mathcal{I}_0, s_2 \in \mathcal{R}^{s_0}, s_4 \in \mathcal{R}\}$$

$$F_g^{int}(t) = F_g^{int}(s_o, s_2, t) = \sum_{i=0}^{s_0-1} s_2^i.\text{count}(T_{reset}, t, p_{in}^i) \quad (17)$$

$$\begin{aligned} F_g^{ext}(s_4, F_g^{int}(t), t) &= 0 \text{ if } F_g^{int}(t) < s_4 \\ &= 1 \text{ if } F_g^{int}(t) \geq s_4 \end{aligned} \quad (18)$$

where count(x, y, p_{in}^i) counts the pulses received on input port p_{in}^i from time x to time y, and the s_2 and s_4 are chosen so that the F_n^{ext} generated produces infinitely short pulses.

As well as synchronous and continuous dynamics, one can have asynchronous dynamics. This is sometimes taken to mean that one node at a time (chosen at random) re-evaluates its inputs, producing a new output, and sometimes that each node re-evaluates its output after a random time delay. The output may be constant, or it may fall away in some predefined fashion. If each node evaluates its output at the start of each interval, and that output is constant, one may define the generator (following generator2) as follows:

$$\text{gname}(g) = \text{ generator5}$$

$$\text{PAR}(g) = \{(s_0, s_1, s_2, s_3) : s_0, s_1 \in \mathcal{I}_0, s_2 \in \mathcal{R}^{s_0}, s_3 \in \mathcal{R}\}$$

Let $r_0 = 0$, and r_j ($j = 1 \ldots$) be a sequence of random positive real numbers, which will be the interval lengths (i.e., the delays between consecutive re-evaluations). Thus the kth time interval (during which the output will be constant) is

$$t \in [\sum_{j=0}^{k} r_j, \sum_{j=0}^{k+1} r_j]. \quad (19)$$

Then, for t in the k'th time interval

$$F_g(s_3, s_2, s_0, t) = 1/(1 + \exp -s_3(\sum_{i=0}^{s_0-1} s_2^i p_{in}^i(\sum_{j=0}^{k} r_j))). \quad (20)$$

The output during the kth interval is determined by the inputs at the beginning of this interval. The random intervals are generated from within the node itself so that they are different for each node.

One can rewrite the simpler pulse-based generator described in (17) and (18) to use asynchronous dynamics by incorporating (19): for t in the kth time interval,

$$F_g^{int}(s_0, s_2, t) = F_g^{int}\left(s_0, s_2, \sum_{j=0}^{k} r_j\right) = \sum_{i=0}^{s_0-1} s_2^i.\text{count}\left(T_{reset}, \sum_{j=0}^{k} r_j, p_{in}^i\right). \quad (21)$$

In this case, incoming pulses are counted at the end of each time interval, and then an instantaneous pulse may be generated. This can be used as a basis for modeling the random neurons of [14] where stochastically generated pulse outputs are sent out on one arc at a time, and can either add 1 or subtract 1 from their target unit's activation (which is itself bounded below at 0). Generation of a pulse occurs only when the activation is positive, and has an average rate of R. Pulse generation results in a decrement of the unit's activation. Each output pulse is sent out of exactly one output port. Neurons of this form can be modeled by a generator based on (21):

$$\text{gname}(g) = \text{generator6}$$

$$\text{PAR}(g) = \{(s_0, s_1, s_2, s_5, \text{prob}(p_{out})) : s_0, s_1 \in \mathcal{I}, s_2 \in \{-1, +1\}^{s_0}, s_5 \in \mathcal{R}, \text{prob}(p_{out}) \in \mathcal{R}^{s_1}\}$$

$$F_g^{int}(0) = F_g^{int}(s_5, 0) = s_5$$

$$\begin{aligned} F_g^{int}(t) &= F_g^{int}(s_0, s_1, s_2, s_5, t) \\ &= F_g^{int}\left(s_0, s_1, s_2, s_5, \sum_{j=0}^{k} r_j\right) \\ &= \min\left(0, F_g^{int}\left(\sum_{j=0}^{k-1} r_j\right) + \sum_{i=0}^{s_0-1} s_2^i \right. \\ &\quad .\text{count}\left(\sum_{j=0}^{k-1} r_j, \sum_{j=0}^{k} r_j, p_{in}^i\right) \\ &\quad \left. - \sum_{i=0}^{s_1-1} \text{pulsexit}(k, i)\right) \end{aligned}$$

and the expected number of output pulses on port i in the kth time interval, $E(p_{out}^i, k)$, is

$$E(p_{out}^i, k) = R.\text{sgn}\left(F_g^{int}\left(\sum_{j=0}^{k-1} r_j\right)\right).\text{prob}(p_{out}^i).r_k$$

where the function pulsexit(k, i) counts pulses output by output port p_{out}^i in time period k, the function sgn(x) returns +1 if $x > 0$, and 0 otherwise, and prob(p_{out}^i) is the predefined probability of an output pulse being sent out on output port i (where $\sum_{i=0}^{s_1-1} \text{prob}(p_{out}^i) = 1$). Thus the probability of the pulse being sent out of a specific output port is local to the output port, but the effect of the pulse (whether excitatory or inhibitory) is defined by the s_2, and this is local to the input ports. For completely correct operation, the r_j should be small enough so that no more than one pulse is received or transmitted in each time interval; however, letting them be larger still allows operation, although the order of counting the input pulses may change the unit's precise operation.

Since the dynamics is set up by F_g (or F_n) it may be different at different nodes. Care must be taken to ensure that the dynamics at the different nodes in N are compatible.

V. Adaptation

Adaptation of a net is the alteration of its behavior as a result of earlier events. This framework does not attempt to specify dynamic nets, so that adaptation here relates only to modulation of the behavior of the processing elements themselves. Thus adaptation is the alteration in the computation of values by $F_{n,t}$ in response to input at the node's input ports and its internal state (itself computed by F_n, so providing a memory for earlier events). Any factors that are to influence the changing of $F_{n,t}$ must be brought to the node. The precise way in which the alteration occurs will be defined by the generator and the mappings setparameters and instantiate.

Adaptation can often be expressed more simply by using the internal state in the computation of the node's output. In this way, F_n itself can be fixed. This permits straightforward specification of Hebb-style learning, where the adaptation takes place at synapses depending only on the pre- and post- synaptic activation. Altering generator1 (which has synchronous dynamics) we can define an adaptive version:

$$\begin{aligned} \text{gname}(g) &= \text{generator1a} \quad (22) \\ \text{PAR}(g) &= \{(s_0, s_1, s_2, s_3, s_4) : s_0, s_1 \in \mathcal{I}_0, s_2 \in \mathcal{R}^{s_0}, s_3 \in \mathcal{R}, s_4 \in \mathcal{R}\} \\ F_g^{int}(t) \text{ defines } w^i(t) &= w^i(t-1) + s_4 p_{in}^i(t-1) F_g^{out}(t-1) \\ F_g^{out}(t) &= 1/\left(1 + \exp\left(- s_3\left(\sum_{i=0}^{s_0-1} w^i(t) p_{in}^i(t-1)\right)\right)\right) \end{aligned}$$

where $w^i(0) = s_2^i$. In this case, s_4 is the learning rate parameter. On instantiation, STATE(n) is the space of possible weights defined by the possible values for s_2, (i.e., $\mathcal{R}^{s_0} \subset$ PAR(g)), and (22) defines state(n, t) and the state update rule. Note that input at time $t-1$ results in output at time t, so that the weight update that happens at time t uses the input at time $t-2$ and the corresponding output at time $t-1$. Other adaptation schemes depending on the same factors can easily be specified. However, if instead of each specifying that each node represent a whole neuron, we subdivide the neurons so that the neural elements are (for example) synapses, activation functions, output functions, and axons, then, since

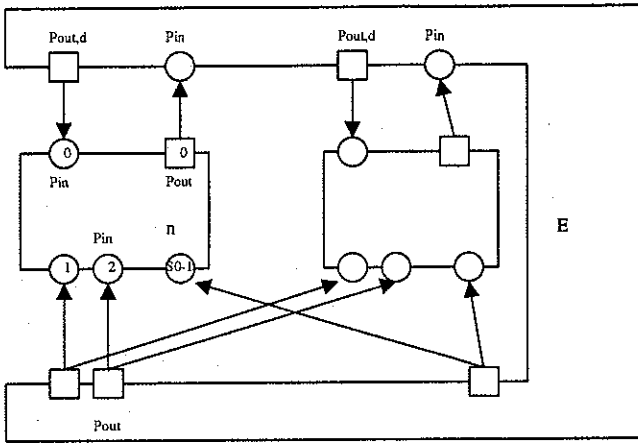

Fig. 3. Illustration of part of Delta-rule based network.

adaptation takes place at the synapses, the relevant data is no longer local. In this case, arcs supplying the post-synaptic (i.e., axonic) output back to the synapse would be needed. Since all interaction is represented by arcs, these "extra" arcs are to be expected, since they represent the effect of the spike at the axon hillock on the dendrites of the neuron.

For rules with teacher input, some of the ports supplying input from the environment (i.e., environment output ports) $p_{out,d}(E)$, supply desired outputs, rather than true inputs to the network. For the Delta rule, there will be a 1:1 correspondence between the elements of $p_{in}(E)$ and $p_{out,d}(E)$. Assuming the usual type of network nodes for a delta-rule net, each node will have exactly one output, and one desired value input corresponding to this output. Every other input port supplies a pre-synaptic value, so that the unit has enough information locally available for calculating weight changes. A generator for this type of node can be defined: writing s_0 for the number of input ports, as before, setting s_1 the number of output ports, to 1, and using the 0th input port to correspond to the desired output:

$$\begin{aligned} \text{gname}(g) &= \text{ generator1d} \\ \text{PAR}(g) &= \{(s_0, s_2, s_3, s_4) : s_0 \in \mathcal{I}_0, s_2 \\ &\quad \in \mathcal{R}^{s_0-1}, s_3 \in \mathcal{R}, s_4 \in \mathcal{R}\} \\ F_g^{int} \text{ defines } w^i(t) &= w^i(t-1) + s_4(p_{in}^0(t-1) \\ &\quad - F_g^{out}(t-1))p_{in}^i(t-2) \\ F_g^{out}(t) &= 1/(1 + \exp - s_3(\sum_{i=1}^{s_0-1} w^i(t)p_{in}^i(t-1))) \end{aligned}$$

where $w^i(0) = s_2^i$. Again, s_4 is the learning rate; different output functions can be set up in F_g^{out}. Note that $F_g^{int}(t)$ needs to be computed before $F_g^{out}(t)$ and that input at time $t-1$ results in output (and therefore desired values) at time t. This is reflected in the weight update rule. Fig. 3 illustrates part of such a network.

This can be extended to the back-propagated delta rule [15]: in a layered feedforward net, only the top (or output) layer has connection to the environment. A generator for these units can be defined in a similar way to generator1d, except that they must feed error contributions back to the previous layer. Thus, as well as the single primary output, $F_g^{out,0}(t)$, there are error outputs, $F_g^{out,i}(t)$ one corresponding to each $p_{in}^i (i = 1 \ldots s_0 - 1)$. All together, there are s_0 output ports and s_0 input ports. Thus, a generator may be defined:

$$\begin{aligned} \text{gname}(g) &= \text{ generator1bpd.top} \\ \text{PAR}(g) &= \{(s_0, s_2, s_3, s_4) : s_0 \in \mathcal{I}_0, s_2 \\ &\quad \in \mathcal{R}^{s_0-1}, s_3 \in \mathcal{R}, s_4 \in \mathcal{R}\}. \end{aligned}$$

The 0th port is the output to the environment. This primary output is defined by:

$$F_g^{out,0}(t) = 1/(1 + \exp - s_3(\sum_{i=1}^{s_0-1} w^i(t)p_{in}^i(t-1)))$$

where the initial state will be generated from $w^i(0) = s_2^i$ and the state update rule will be

$$\begin{aligned} F_g^{int} \text{ defines } w^i(t) &= w^i(t-1) + s_4(p_{in}^0(t-1) \\ &\quad - F_g^{out,0}(t-1))p_{in}^i(t-2). \end{aligned}$$

The error outputs are defined by

$$\begin{aligned} F_g^{out,i}(t) &= w^i(t-1)(p_{in}^0(t-1) \\ &\quad - F_g^{out,0}(t-1)).s_3.F_g^{out,0}(t-1).(1 - F_g^{out,0}(t-1)). \end{aligned}$$

The primary output is transferred to p_{out}^0, and the error output to the other output ports.

Nodes in the intermediate layers no longer have any interaction with the environment: however, they have a number of output ports transferring the node's primary output to other nodes. Additionally, there will be extra input ports receiving the error input from the next layer up. A generator can be defined:

$$\text{gname}(g) = \text{ generator1bpd.mid}$$

$$\begin{aligned} \text{PAR}(g) = &\{(s_0, s_1, s_2, s_3, s_4) : s_0, s_1 \in \mathcal{I}_0, s_2 \\ &\in \mathcal{R}^{s_0}, s_3 \in \mathcal{R}, s_4 \in \mathcal{R}\}. \end{aligned}$$

In this case, s_0 is the number of "true" inputs, and s_1 the number of "true" outputs. In fact, including arcs used to propagate errors, there are $s_1 + s_0$ input arcs, and the same number of output arcs. Number the output arcs so that the first s_1 transfer the primary output, and the next s_0 transfer error values. Number the input arcs so that the first s_0 receive "true" inputs, and the next s_1 receive error inputs. Then we can write

$$F_g^{out,j}(t) = 1/(1 + \exp - s_3(\sum_{i=0}^{s_0-1} w^i(t)p_{in}^i(t-1)))$$

for $j = 0 \ldots s_1 - 1$, and

$$\begin{aligned} F_g^{out,j}(t) &= w^j(t-1).s_3.F_g^{out,0}(t-1).(1 - F_g^{out,0} \\ &\quad \cdot(t-1)) \sum_{i=s_1}^{s_0+s_1-1} p_{in}^i(t-1) \end{aligned}$$

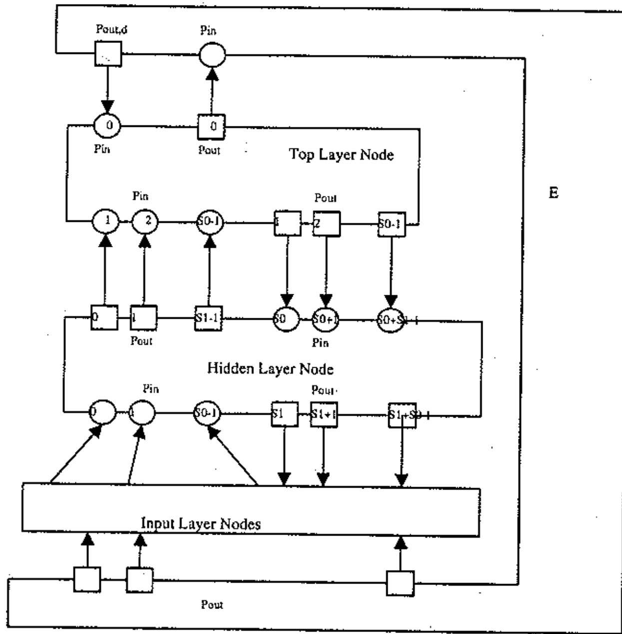

Fig. 4. Illustration of part of a back-propagated Delta-rule network.

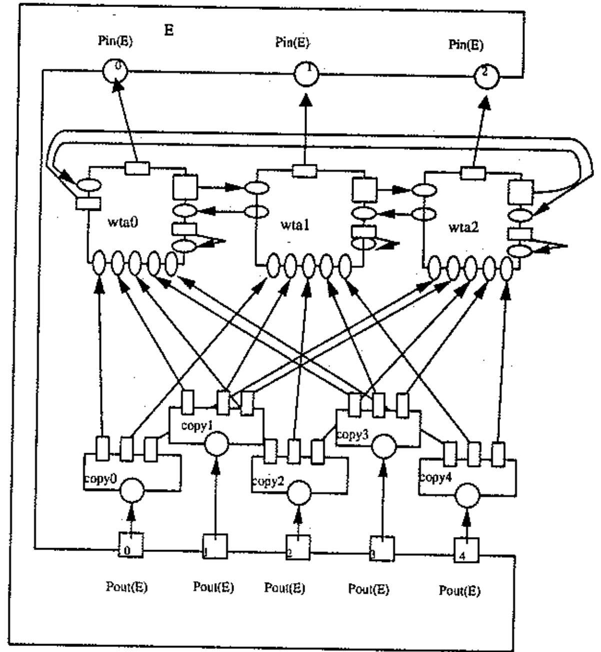

Fig. 5. A 5-input 3-output winner-takes-all net.

for $j = s_1 \ldots s_0 + s_1 - 1$. F_g^{int} defines the state (weight) changes as follows:

$$w^j(t) = w^j(t-1) + s_4 \sum_{i=s_1}^{s_0+s_1-1} p_{in}^i(t-1)p_{in}^j(t-2)$$

for $j = 0 \ldots s_0 - 1$. A generator for the input units can also be defined; it needs no internal state, but simply distributes the input values to nodes in the first layer. It is possible (though rather cumbersome) to combine these generators to produce one that can generate nodes in a recurrent back-propagated net with nodes whose primary output goes both to the environment and to other nodes. This could be used to specify networks like those in [16]. Fig. 4 illustrates a small part of a back-propagated network. The online version of the learning rule has been used here: the batch version would need the errors to be accumulated inside the state of each unit.

Only a small number of possible adaptation schemes have been illustrated. Similar techniques can be used for any learning technique that propagates errors. All processing element interaction is explicitly specified by the $F_g^{out,i}$ and the arcs: this may make descriptions of certain real neural nets more difficult, since these may require the effects of brain hormones to be made explicit in terms of arcs. Such forms of adaptation are beyond the scope of most neural net simulation; however, this model does allow their specification at the cost of additional arcs. We have not needed to make F_n itself adapt; however, this could be useful, e.g., where the adaptation rate was being annealed.

VI. Examples

The framework will be used to specify an adaptive 3-unit 5-input winner-takes-all net in a bottom-up manner, and a more complex net in a top-down way.

A. Winner-takes-all Net

The network is shown in Fig. 5. It has a three-unit cluster of nodes, each connected to each other. Each node makes one output to the environment, and receives input from all five of the outputs from the environment. We use the framework to specify the net, and then to turn the whole net into a generator.

The environment is simply characterized:

$$p_{in}(E) = \{p_{in}^i(E) : i = 0 \ldots 2\}$$
$$p_{out}(E) = \{p_{out}^i(E) : i = 0 \ldots 4\}$$

Since the values from each $p_{out}^i(E)$ need to be transferred to all three nodes in the cluster, and since each port can only be part of one arc, extra nodes will be needed to replicate these values. The network will have continuous dynamics, and the nodes in the cluster will be built up from a partially adaptive version of generator2. This is as follows:

$$\text{gname}(g) = \text{generator2a}$$

$$\text{PAR}(g) = \{(s_0, s_1, s_2, s_3, s_4, s_5) : s_0, s_1, s_2 \in \mathcal{I}_0, s_3 \in \mathcal{R}^{s_0+s_1}, s_4, s_5 \in \mathcal{R}\}$$

$$F_g^{out}(t) = 1/\left(1 + \exp -s_4\left(\sum_{i=0}^{s_0+s_1-1} w^i(t) \int_0^t W(t-\tau)p_{in}^i(\tau)d\tau\right)\right)$$

where the initial internal state will be generated by for $i = 0 \ldots s_0 - 1$

$$w^i(t) = s_3^i$$

and updated by using F_g^{int}, which defines that for $i = s_0 \ldots s_0 + s_1 - 1$

$$w^i(t) = s_3^i + s_5 \int_0^t m(p_{in}^i(\tau), F_g^{out}(\tau))d\tau.$$

In this generator, there are s_0 nonadaptive synapses, s_1 adaptive synapses, s_2 outputs; the s_3 are the initial weights, s_4 is the slope of the logistic output function, and s_5 is the learning rate. The function $m()$ defines the way in which the presynaptic value $p_{in}^i(t)$ and the postsynaptic output $F_g(t)$ are used in weight alteration. For this generator, we shall simply assume $m()$ to return the product of its two parameters.

The generator for the nodes used to replicate the $p_{out}(E)$ is more straightforward:

$$\text{gname}(g) = \text{copy}$$
$$\text{PAR}(g) = \{n : n \in \mathcal{I}_0\}$$
$$F_g(t) = p_{in}(t).$$

Nodes generated by copy have one input port, and n output ports, all with the same output. Their internal state is constant.

All the nodes in the cluster are of the same type. The generator generator2a generates the nodes as follows: first, setparameters is used to fix values for $s_i : i = 1 \ldots 5$: s_0, the number of input ports with nonadaptive synapses will be 3; s_1, the number of input ports with adaptive synapses, will be 5; s_2, the number of output ports will be 4 (one to the environment, and one to each of the nodes); and s_3 will be set to the initial weight matrix for the synapses of that node. s_4 and s_5 can be set as required. All these values can be different at each node. Next, instantiate is used to give the nodes different names, say wta$i$, for $i = 0 \ldots 2$ and to set up the ports, STATE(n), and state($n, 0$) for each node. Similarly, setparameter is used to set $n = 5$ for the nodes to copy the outputs from the environment. Instantiate gives them different names, copy$i$, for $i = 0 \ldots 4$ and sets up the ports and state. This generates the set of nodes, N.

To complete the construction of the net, the set A of arcs must be defined. Writing $p_{in}^i(\text{wta}j)$ for the ith input port of the jth cluster node, and $p_{in}^i(\text{copy}j)$ for the ith input port of the jth copy node, and starting with arcs carrying values to the net from the environment:

$$A_I = \{(p_{out}^i(E), p_{in}^0(\text{copy}i)) : i = 0 \ldots 4\}.$$

Next, nodes carrying values to the environment from the net:

$$A_O = \{(p_{out}^3(\text{wta}j), p_{in}^j(E)) : j = 0 \ldots 2\}.$$

Lastly, the internal arcs A_{Int}:

$$\begin{aligned} A_{Int} = \{&(p_{out}^0(\text{wta}j), p_{in}^0(\text{wta(j)})), \\ &(p_{out}^1(\text{wta}j), p_{in}^1(\text{wta}(j+1 \bmod 3))), \\ &(p_{out}^2(\text{wta}j), p_{in}^2(\text{wta}(j+2 \bmod 3))), \\ &(p_{out}^j(\text{copy}i), p_{in}^{i+3}(\text{wta}j)) : j = 0 \ldots 2, i = 0 \ldots 4\} \end{aligned}$$

$$A = A_O \cup A_I \cup A_{Int}.$$

This, then, specifies the net.

The network (N, A) can be made into a generator, g, for a single node by first providing a new generator name, secondly setting up PAR(g), and thirdly defining F_g. The new generator name is straightforward. Some dimensions of PAR(g) will be defined by replacing the $p_{in}^0(\text{copy}i)$ in the arcs in A_I and the $(p_{out}^3(\text{wta}j))$ in the arcs in A_O by parameterized port generators, and some of the other dimensions of PAR(g) will determine whether each input port receives input or not, and how many arcs leave each of the outputs from the cluster nodes. The other dimensions of PAR(g) are those required from each of the nodes of the original net. The F_g of this new generator cannot be simply stated: it is that set up by the composite effect of all the nodes. F_g^{int} depends on the F_n of both the cluster and copying nodes: F_g^{out} will depend only on the F_n^{out} of the cluster nodes. The dynamics for the composite generator does not present a problem, since continuous dynamics was used from the start. However, had synchronous dynamics been used, a faster internal "tick" would have been needed. It would also be possible to extend this form of generator to one producing an arbitrary n-input m-output winner-takes-all net.

B. Top-Down Specification

One of the main applications of the notation is to provide a method for developing and discussing a proposed network. In this example, we start from a very brief description of the proposed net, and use the notation to discuss some possible networks.

The problem chosen is the network implementation, in a top-down manner, of the classification of sounds into r classes from an input consisting of band-limited power in each of a set of three bands. Each input varies rapidly over time, and the system must attempt to classify this input. Only three bands were specified, purely for simplicity; the problem could use many. Similarly, the problem here is just one case of a commonly occurring problem: given time-varying input from a number of sensors, produce a (more slowly) time-varying classification. This is a nontrivial problem. The most appropriate form of network is not at all clear, and will probably depend on the precise classification task. It is thus a suitable candidate for discussion using the notation proposed in Section II.

A possible top-level is shown in Fig. 6. This level is useful for clarifying the net's interaction with its environment. The system consists simply of the environment and one node, ntop. Characterizing the environment, E, means defining the input ports, $p_{in}(E)$, and the output ports $p_{out}(E)$. Three of the output ports are clear from the initial description: they supply the band-limited signals. Similarly, the input ports must receive the classification from the network. But more precise definition poses more detailed questions, as indeed, one would wish it to do. What form does the classification take? How is it to be represented? Is there another output from the environment, representing the desired value for classification during training, or will the net be self-organizing?

In a real application, the answers to these questions would come from the specifier of the application. Here, we decide that

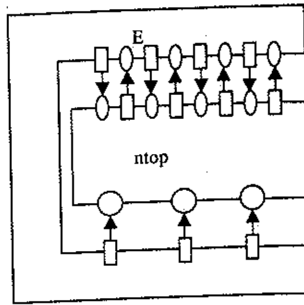

Fig. 6. Top-level diagram for sound classification network.

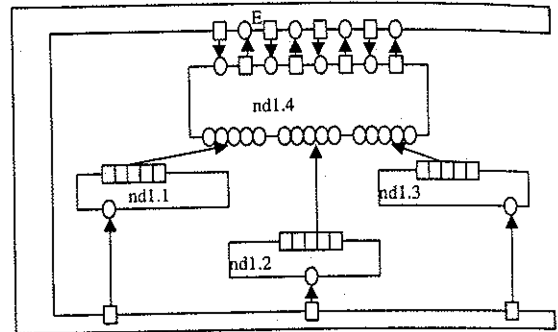

Fig. 7. First-level decomposition of sound classification net.

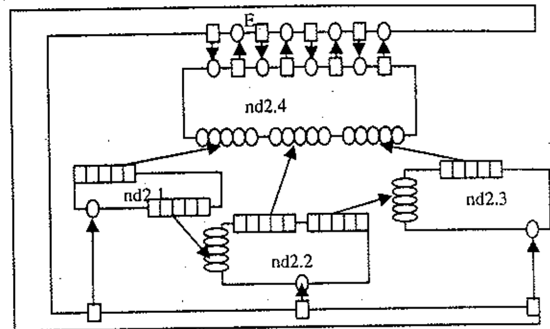

Fig. 8. Another first-level decomposition of sound classification net.

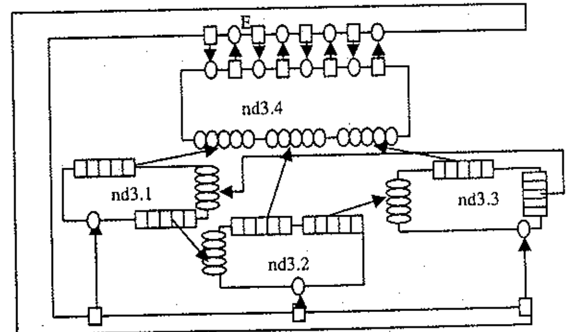

Fig. 9. A third first-level decomposition of sound classification net.

the classification will be represented by a number of inputs to the environment, one per possible classification (locally coded), and that there will be the same number of outputs from the environment, used during training, specifying the correct classification. Then, we can specify the output and input ports of the environment:

$$p_{out}(E) = \{p^i_{out}(E) : i = 0..r+2\}$$

where $p^0_{out}(E)$—$p^{r-1}_{out}(E)$ provide the desired values for training, and $p^r_{out}(E)$—$p^{r+2}_{out}(E)$ provide the band-limited signals,

$$p_{in}(E) = \{p^j_{in}(E) : j = 0..r-1\}$$

providing one input to the environment per classification.

The $p_{out}(E)$ define the p_{in}(ntop), and vice-versa. We also need information on the dynamical nature of these signals. (In a real problem, these would be prespecified.) The band-limited input signals may be continuous or discreticized in time, as may both the training signals from the environment and the net's output to the environment. Considering the nature of the problem, we choose that both be discreticized, but that the band-limited input has a much higher sampling frequency than the net's training signal or output. It is not useful to define the STATE(ntop); discussing the function F_{ntop} at this point serves to restate the problem the net is intended to solve.

There are many ways in which this top level can be decomposed. Three possibilities are shown in Figs. 7–9, corresponding to separate and independent processing of the inputs $p^0_{out}(E)$—$p^2_{out}(E)$; processing each of these in turn and using the output of each stage in the next stage; and separate processing of each using data from the others. In each of these cases, this processing is followed by a classification stage that uses the output from all the earlier stages. However, in either drawing these diagrams, or in attempting to characterize the internal ports, one is rapidly faced with decisions. For example, in Fig. 7, the input port set p_{in}(nd1.1) is a simple subset of p_{in}(ntop) defined earlier; but the output port set p_{out}(nd1.1) may take many forms. They could be similar to those of p_{out}(ntop), if the node nd1.1 is attempting the complete classification task, and node nd1.4 selecting the most appropriate classification. Alternatively, the node nd1.1 may be recoding the input so as to emphasize certain features, in which case p_{out}(nd1.1) may be completely different. Indeed, the nodes nd1.1–nd1.3 may be either supervised (in which case there must be a training signal, probably from node nd1.4, not illustrated in the figure), or they may be self-organizing, adapting purely in response to their inputs.

Similar arguments apply to nodes nd2.1–nd2.3 and nd3.1–nd3.3; additionally, for these nodes, there is the question of the form of the output from their output ports to the other input ports not on nd2.4 or nd3.4.

For the purpose of further discussion, we choose to use the network of Fig. 7 with each p_{out}(nd1.i) (for $i = 1 \ldots 3$) having five elements and each node nd1.i (for $i = 1 \ldots 3$) acting as a self-organizing net. We can then describe the set p_{in}(nd1.4): it has $r+15$ elements; and there are r arcs from the environment supplying the desired values (p^0_{in}(nd1.4)–p^{r-1}_{in}(nd1.4)), and arcs connecting p^k_{out}(nd1.i) to $p^{r+5.(i-1)+k}_{in}$(nd1.4) for $i = 1 \ldots 3$ and $k = 0 \ldots 4$. We also need to choose the nature of the signals output from the nodes nd1.i (for $i = 1 \ldots 3$). These will be discreticized, but how frequently should they change? Theoretically, this could be at any rate, but for simplicity,

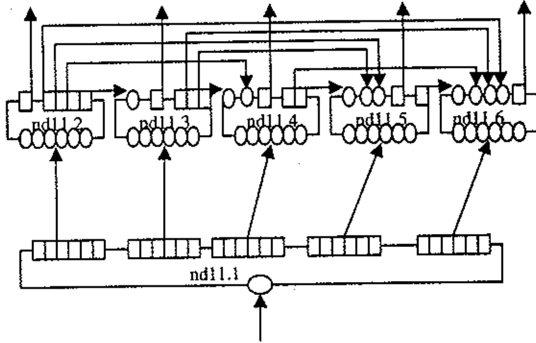

Fig. 10. Second level decomposition of part of sound classification net: node nd1.i of Fig. 7.

it should be either at the rate of change of $p^i_{out}(E,t)$ or $p^j_{in}(E,t)$. We choose the latter; this means that in some way the internal state space of the nodes nd1.i (for $i = 1 \dots 3$) will be remembering some of its previous inputs.

Further decomposition can be attempted on each of the nodes nd1.i separately. For example, node nd1.4 may be decomposed into a 3-layer back-propagated net. Generators for each node of such a net are discussed in Section V; in this case there would be 15 input nodes, r output nodes, and some unspecified number of nodes in the hidden layer. Since such nets are commonly used, one might build up a parameterized net generator, rather in the style of the example in Section 6.A. For the other nodes, we will decompose one of the nd1.1–nd1.3, since they are likely to be similar.

One possible decomposition is illustrated in Fig. 10; this is a principal component analyzer [17]. Here, node nd11.1 distributes input to the other nodes. It could simply send the same input to each other node, or it could (particularly recollecting that the original band-limited input changes rapidly) behave like a shift register, supplying each other node with a continuously moving window onto the input, or it could buffer the input into nonoverlapping groups. We have chosen the last, since it also allows the node to match the rapidly changing input with the less rapidly changing output. In the figure, the blocks are of length 6.

The node has input port set $p_{in}(\text{nd11.1}) = \{p^0_{in}(\text{nd11.1})\}$, and output port set $p_{out}(\text{nd11.1}) = \{p^i_{out}(\text{nd11.1}) : i = 0 \dots 29\}$. (Since all the outputs are sent to all the other nodes, there are 30 output ports.) The node's internal state-space, STATE(nd11.1), is $\mathcal{R}^6$ (assuming $p_{in}(\text{nd11.1}, t) \in \mathcal{R}$), and

$$\begin{aligned}\text{state}(\text{nd11.1}, t) =& (p^0_{in}(\text{nd11.1}, t-5), p^0_{in}(\text{nd11.1}, t-4),\\ & \dots, p^0_{in}(\text{nd11.1}, t)).\end{aligned}$$

Nodes nd11.2–nd11.6 form the adaptive part of the net. These are similar, though not identical: each node receives the same input from its input ports connected to nd11.1, and each node has one output port that sends data out of the subnetwork. However, each node also sends output to all the nodes on its right. Although Fig. 10 shows these as coming from single ports, they are multiple ports, each corresponding to one output port per input port. We start to define the input port set by writing $\{p^j_{in}(\text{nd11.i}) : j = 0 \dots 5\}$ for $i = 2 \dots 6$ for those input ports whose input originates in node nd11.1. We can write the output port set as

$$p_{out}(\text{nd11.i}) = \{p^j_{out}(\text{nd11.i}) : j = 0 \dots 6.(6-i)+1\}$$

with $p^0_{out}(\text{nd11.i})$ being the external output. The rest of the input ports, if any, come from other nodes on that node's left. We can write

$$p_{in}(\text{nd11.i}) = \{p^j_{in}(\text{nd11.i}) : j = 0 \dots 6.(i-1)-1\}$$

where arcs connect

$$(p^j_{out}(\text{nd11.i}), p_{in}^{((j-1)\text{mod}6)+6.r}(\text{nd11.}(i+r)))$$

for $i = 2 \dots 5, r = 1 \dots 4, (i+r) \leq 6, 0 < j < 6.(6-i)+1$ (taking some liberties with the writing of the node name).

The STATE(nd11.i) space is the space of the local weights; each node has 6 weights, so that the state at time t can be written

$$\text{state}(n,t) = (w^0_{nd11.i}(t), w^1_{nd11.i}(t), \dots, w^5_{nd11.i}(t))$$

and the function $F_{nd11.i}$ can be characterized by defining the output on the node's external port, the output on the other ports, and how the state(nd11.i,t) updates. For the first,

$$p^0_{out}(\text{nd11.i}, t) = \sum_{m=0}^{5} w^m_{nd11.i}(t).p^m_{in}(\text{nd11.i}, t).$$

The other outputs are

$$p^{6r+m+1}_{out}(\text{nd11.i}, t) = p^0_{out}(\text{nd11.i}, t).w^m_{nd11.i}(t) \quad \text{for } i = 2 \dots 5, m = 0 \dots 5, r = 0 \dots (5-i).$$

The state update rule is

$$\begin{aligned}\Delta w^m_{nd11.i}(t+1) =& \eta.p^0_{out}(\text{nd11.i}, t)(p^m_{in}(\text{nd11.i}, t) - \\ & \Big(w^m_{nd11.i}(t).p^0_{out}(\text{nd11.i}, t) \\ & + \sum_{j=1}^{i-2} p^{6j+m}_{in}(\text{nd11.}i, t)\Big).\end{aligned}$$

VII. Conclusions and Further Work

A framework for the specification of neural networks has been proposed. This framework does not predefine the dynamics of the net, nor does it presuppose a specific level of implementation. It has, for example, been used to specify the behavior of nonlinear dendrites for a neuron in [18] originally described in [19]. This is a model intermediate in complexity between the purely additive dendrites frequently used in net simulation, and ion-channel based simulations. The framework is general enough to allow many new ideas from neurobiology to be built in.

Using generators allows nets to be built up in a structured way. Similar but nonidentical neurons can be specified in a way that makes their similarity clear. Net elements can be constructed hierarchically, as described, or can be refined by replacing single neural elements with networks which implement the required transfer functions. There remains work to be done on the dynamics of the interaction of nodes specified at differing levels, particularly when synchronous dynamics are in use.

The notation has been shown to be useful both for describing nets, and as a language for discussing how a particular system might be decomposed into subnets. Problems at the level of connections between nodes are found when specifying the ports; using the notation for function specification, one finds out both the requirement for internal state, and whether all the information required is local. One is forced to make the time-dynamics of the nodes explicit. It is thus a useful design tool for neural networks, prior to their implementation. The framework has not been extended into a language, nor has it been automated. It is used as a means of thinking about nets, and as a way of describing and specifying nets prior to their implementation. Frequently, the enforced formality of describing the network using the framework results in errors and inconsistencies being discovered at an early stage. The system is not at a level of refinement where it could be used for direct code or hardware generation. Indeed, the whole field has not yet reached a level where a complete abstract syntax for specifications can be produced.

ACKNOWLEDGMENT

The original purpose of this work was to allow the different groups collaborating on the EEC BRAIN project "Learning Automata: Toward a Machine" to communicate more effectively. It also grew out of an attempt to define (or at least describe) nets prior to producing a new neural net simulator to run on transputer systems. Thanks are due to Prof. P. Jorrand for many useful comments on an early draft, to the other members of the CCCN at Stirling, particularly P. Hancock, M. Roberts, and A. Thomson for discussions and encouragement, and to the anonymous referees and issue editor for useful comments and suggestions.

REFERENCES

[1] C. Hewitt, "Viewing control structures as patterns of passing messages," *J. of Artificial Intelligence*, vol. 8, June 1977.
[2] B. Derot *et al.*, "NACRE: A neuron-oriented programming environment," in *Neuro-Nimes '89*, 1989.
[3] P.C. Treleaven, "PYGMALION: Neural network programming environment," in T. Kohonen *et al.*,Eds., *Artificial Neural Networks*. North-Holland, 1991.
[4] *ANSpec*^{TM} *User's Manual*, Science Applications International Corporation, San Diego, CA, 1989.
[5] R. Hecht-Nielsen, *Neurocomputing*. Reading, MA: Addison-Wesley, 1990.
[6] P. Bessiere *et al., MENTAL: A Virtual Machine Approach to Artificial Neural Networks Programming*, ESPIRIT BRA Project 3049 Final Report, June 30, 1991.
[7] P. Koikkalainen, "MIND: A specification formalism for neural networks," in T. Kohonen *et al.*, Eds., *Artificial Neural Networks*. North-Holland, 1991.
[8] C.A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, 1978.
[9] R. Milner, "Flowgraphs and flow algebras," *J. ACM*, vol. 26, 1979.
[10] O. Ekeberg *et al.*, "SWIM—A simulator for real neural networks," *Royal Institute of Technology Studies in Artificial Neural Systems TRITA-NA-P9014*, 1990.
[11] N.T. Carnevale *et al.*, "Neuron simulations with Saber," *J. Neuroscience Methods*, vol. 33, 1990.
[12] A. Vladimirescu *et al.*, "SPICE Version 2G Users Guide," Univ. California, Aug. 1981.
[13] P. Caspi *et al.*, "LUSTRE: A declarative language for programming synchronous systems," in *Proc.14th Symp. POPL*, 1987.
[14] E. Gelenbe, "Random neural networks with negative and positive signals and product form solution," *Neural Computation*, vol. 1, pp. 502–510, 1989.
[15] D.E. Rumelhart *et al.*, "Learning representations by back-propagating errors," *Nature*, Oct. 1986.
[16] R.J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks, *Neural Computation*, vol. 1, pp. 270–280, 1989.
[17] T.D. Sanger, "Optimal unsupervised learning in a single layer feedforward neural network," *Neural Networks*, vol. 2, pp. 459–473, 1989.
[18] A.W. Thomson, "Modeling neural nets," M.Sc thesis, Univ. Stirling Department of Computing Science, Apr. 1990.
[19] L.S. Smith, "Formalizing neural networks," in *Neural Networks from Models to Applications*, in L. Personnaz *et al.*, Eds. Paris: IDSET, 1989.

Leslie Smith received the B.Sc. in mathematics in 1973 and the Ph.D. in computing science in 1981, both from the University of Glasgow.

He was a founding member of the Centre for Cognitive and Computational Neuroscience, an interdisciplinary group working on neural networks and vision. He has been a lecturer at the University of Stirling, Scotland, since 1984. His research interests are in learning algorithms and net specification, and in the application of neural network technology to real problems.