Symbolic Processing in Neural Networks

João Pedro Neto[1], Hava T. Siegelmann[2], and J. Félix Costa[3]

jpn@di.fc.ul.pt, iehava@ie.technion.ac.il, and fgc@math.ist.utl.pt

[1] Faculdade de Ciências, Dept. Informática, Bloco C5, Piso 1, 1700 Lisboa – PORTUGAL
[2] Faculty of Industrial Engineering and Management, TECHNION CITY, HAIFA 32 000 – ISRAEL
[3] Instituto Superior Técnico, Dept. Matemática, Av. Rovisco Pais, 1049-001 Lisboa – PORTUGAL

Abstract. In this paper we show that programming languages can be translated on recurrent (analog, rational weighted) neural nets. Implementation of programming languages in neural nets turns to be not only theoretical exciting, but has also some practical implications in the recent efforts to merge symbolic and subsymbolic computation. To be of some use, it should be carried in a context of bounded resources. Herein, we show how to use resource bounds to speed up computations over neural nets, through suitable data type coding like in the usual programming languages. We introduce data types and show how to code and keep them inside the information flow of neural nets. Data types and control structures are part of a suitable programming language called NETDEF. Each NETDEF program has a specific neural net that computes it. These nets have a strong modular structure and a synchronization mechanism allowing sequential or parallel execution of subnets, despite the massive parallel feature of neural nets. Each instruction denotes an independent neural net. There are constructors for assignment, conditional and loop instructions. Besides the language core, many other features are possible using the same method. There is also a NETDEF compiler, available at http://www.di.fc.ul.pt/~jpn/netdef/netdef.htm.

Keywords. Neural Networks, Neural Computation, Symbolic Processing, NETDEF.

1. Introduction

Analog recurrent neural nets can be formulated as dynamic systems. We adapt to our case the definition given in [Sontag 90], that corresponds to the concept of discrete, complete, time-invariant dynamic system. A dynamic system is a triple D = (S, U, ϕ) consisting of: (a) a non-empty set S called the state space of D, (b) a non-empty set U called the control-value or input-value space of D, and (c) a total map ϕ: $S \times U \to S$ called the dynamic map. We will consider S and U as being finite dimensional vector spaces over the reals, or restrictions to them.

An analog neural net is considered a particular case of dynamic system, where ϕ is of the form $\sigma \circ \pi$, being π: $S \times U \to S$ an affine map, and σ: $S \to S$ a possibly discontinuous function. Linearity of π is equivalent to the existence of linear maps A: $S \to S$ and B: $U \to S$ such that $\phi(x,u) = \sigma(Ax+Bu)$. These systems are said to be autonomous whenever B is the null matrix, otherwise they are said to be non-autonomous or net systems with controls. If we assume that there exists one variable with constant value 1, then we recover, using appropriate matrices, the model in the form $\phi(x,u) = \sigma(Ax+Bu+c)$, where c is known as the bias vector.

We will consider two cases for the function σ:

(a) The McCulloch and Pitts neural net model,

$$\sigma(x) = \begin{cases} 0 & , x < 0 \\ 1 & , x \geq 0 \end{cases} \tag{1}$$

(b) The saturated sigmoid model,

$$\sigma(x) = \begin{cases} 0 & , x < 0 \\ x & , x \in [0,1] \\ 1 & , x > 1 \end{cases} \tag{2}$$

For notational purposes we write for the new value of the state, at time (iteration) t+1,

$$x(t+1) = \phi(\ x(t), u(t)\ ) \tag{3}$$

to denote a step of computation, or just

$$x^{+} = \phi(\ x, u\ ) \tag{4}$$

With dynamical systems in general we have computation without programmability, i.e., the extra power these systems exhibit has to do with the decoupling between programming and computation. Up to the power of Turing machines, computations are describable by programs that correspond to the prescription by finite means of some rational parameters of the system. Beyond Turing power we have computations that are not describable by finite means: computation without a program. In this paper we want to shed some light on the programmability of neural nets.

1.1. Computability

The use of analog recurrent neural networks for computability analysis is due to Hava Siegelmann. In [Siegelmann 93, 99] they were used to establish lower bounds on their computational power. These systems satisfy the classical constraints of computation theory, namely, (a) input is discrete (binary) and finite, (b) output is discrete (binary) and finite, and (c) the system is itself finite (control is finite). Neurons may hold values within [0,1] with unbounded precision. To work with such analog systems, binary input is encoded into a rational number between 0 and 1 (using fractal coding), and the rational output is decoded into an output binary sequence.

The input streams u_k, for k=1..M, input bits into the system through time. Input streams are maps $u_k:\mathbb{N}\rightarrow\{0,1\}$, different from 0 only finitely many times (this is the classical constraint of input finiteness). $(u_k)_{k=1..M}$ can also be seen as the set of control symbols, to adopt the flavour of Minsky's description of such systems, that the reader may find in [Minsky 67]. In the absence of control the systems are said to be autonomous and the dynamics is given by

$$x_j(t+1) = \sigma\left(\sum_{i=1}^{N} a_{ji}.x_i(t) + c_j \right) \qquad (5)$$

We may then identify the set of computable functions by analog recurrent neural nets, provided that the type of the weights is given. This research program is systematically presented in [Siegelmann 99]:

- The first level of nets is NET[integers], where the type of the weights is integer. These nets are historically related with the work of Warren McCulloch and Walter Pitts. As the weights are integer numbers, each processor can only compute a linear combination of integer coefficients applied to zeros and ones. The activation values are thus always zero or one. In this case the nets 'degenerate' into classical devices called finite automata. It was Kleene who first proved that McCulloch and Pitts nets are equivalent to finite automata and therefore these models are able to recognize all regular languages (see [Minsky 67] for details).

- The second relevant class we consider in this paper is NET[rationals], where the type of the weights is rational. Rationals are indeed computable numbers in finite time, and NET[rationals] turn to be equivalent to Turing machines. Twofold equivalent: rational nets compute the same functions as Turing machines and, under appropriate coding of input and output, they are able to compute the same functions in exactly the same time.

- The third relevant class is NET[reals], not considered in this paper, where the type of the weights is real. Reals are indeed in general not computable. But theories of physics abound that consider real variables. The advantage of making a theory of computation on top of these systems is that nonuniform classes of computation, namely the classes that arise in complexity theory using Turing machines with advice, are uniformly described in NET[reals]. As shown in [Siegelmann 99] all sets over finite alphabets can be represented as reals that encode the families of boolean circuits that recognize them. Under efficient time computation, these networks compute not only all efficient computations by Turing machines but also some non-recursive functions such as (a unary encoding of) the halting problem of Turing machines. Note that while the networks can answer questions regarding

Turing machines computation, they still can not answer questions regarding their own halting and computation.

1.2. Programmability of Analog Neural Nets: Contributions of this Paper

Within the class of NET[rational] we can develop the implementation of programming languages, providing for each written command a suitable analog neural net. The implementation map will be provided in this paper for a (Turing complete) subset of the Occam® language. A first concern is the size of the resulting nets. In fact the size of the nets will increase with the complexity of programs. However, it is always possible to implement the Occam® interpreter of Occam®, determining a universal neural net for the language interpretation. With this paper (an extended version of the short conference paper [Neto *et al.* 97]) we do not aim at theoretical contributions in neural net computation theory. Proofs of Turing completeness of neural nets appeared in the beginning of the nineties (namely, the most well known proof can be found in [Siegelmann 93]). We aim instead at a strong methodological contribution, showing how to perform symbolic computations over neural nets, using a programming language. As a side-effect, this high-level programming language is useful for high-level construction of particular nets that are relevant in the proof of several results in neurocomputing theory, as in [Siegelmann 99] where a net descriptor is used to encode analog shift maps into neural nets.

1.3. Related Work

There is some related work in the literature on symbolic neural computation. The JaNNeT system (see [Gruau *et al.* 95] for details), introduces a dialect of Pascal with some parallel constructs. This algorithmic description is translated, using several automated steps (first on a tree-like data structure and then on a low-level code, named *cellular code*), to produce a non-homogenous neural network (with four different neuron types) able to perform the required computations, a significant difference with NETDEF which produces homogeneous neural networks. Another difference to NETDEF is the network dynamics. In our model, at each instant, all neurons are updated with their new values. In JaNNeT, every neuron is activated only when all its synapses have transferred their values. Since this may not occur at the same instant, the global dynamics is not synchronous. Special attention is given to design automation of the final neural network architecture.

Another neural language project is NIL (outlined in [Siegelmann 93] and [Siegelmann 96]). The NIL system is also able to perform symbolic computations by using certain sets of constructions that are compiled into an appropriate neural net (NIL and NETDEF use the same homogeneous neural architecture). It has a complex set of data types, from boolean and scalar types, to lists, stacks or sets that are kept inside a single neuron, using fractal coding. Because of this, NIL uses unbounded precision, while NETDEF manages type and operator processing with limited precision. An important difference is that NETDEF has a modular design, while NIL has some non modular features, like the synchronization system and the way processes interact with each other. Also, NIL does not provide essential mechanisms required for a neural language like a mutual exclusion scheme for variable access security, temporal processes for real-time applications, genuine parallel calls of functions and procedures, blocking communication primitives for concurrent process interaction, dynamic array assignment. NETDEF deals with and solves all these subjects without loosing its modular properties. A proposed goal, but just delineated in [Siegelmann 96], was to provide mechanisms for tuning the compiled network, in order to generalize the initial processed information. However, NIL was mainly used as a tool to derive specific theoretical results about neurocomputation, and was not fully developed into a network compiler application.

2. Neural Software

2.1. Neural Net Model

An analog recurrent neural net is a dynamic system $\vec{x}(t+1) = \phi(\vec{x}(t), \vec{u}(t))$, with initial state $\vec{x}(0) = \vec{x}_0$, where $x_i(t)$ denotes the activity (firing frequency) of neuron i at time t within a population of N interconnected neurons, and $u_k(t)$ the input bit of input stream k at time t within a set of M input channels. The application map ϕ is taken as a composition of an affine map with a piecewise linear map of the interval [0,1], known as the saturated sigmoid σ, as in (2). The dynamic system becomes

$$x_j(t+1) = \sigma\left(\sum_{i=1}^{N} a_{ji} . x_i(t) + \sum_{k=1}^{M} b_{jk} . u_k(t) + c_j \right) \qquad (6)$$

where a_{ji}, b_{jk} and c_j are rational weights, assuring that a system can be simulated by a Turing machine. Fig. 1 displays the graphical representation of equation (6) used throughout this paper (when a_{ji}, b_{jk} or a_{jj} take value 1, they are not displayed in the graph).

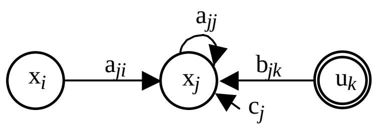

Fig. 1. Graphical notation for neurons, input channels and their interconnections.

Our problem will be to find a suitable neural network for each program written in the chosen programming language.

2.2. The NETDEF Language

We will adopt a syntactic fragment of Occam® for the programming language. Occam® was designed to express parallel algorithms on a network of processing computers (for more information, see [SGS-THOMSOM 95]). With this language a program can be described as a collection of processes executing concurrently, and communicating with each other through channels. Processes and channels are the main concepts of the Occam® programming paradigm.

Occam® programs are built from *processes*. The simplest process is an action. There are three types of action: assignment of an expression to a variable, input and output. Input means to receive a value from a channel and assign it to a variable. Output means to send the value held by a variable through a channel.

There are two primitive processes: skip and stop. The skip starts, performs no action and terminates. The stop starts, performs no action and never terminates. To construct more complex processes, there are several types of construction rules. Herein, we present some of them: *while, if, seq* and *par*. The *if* is a conditional construct that combines a number of processes, guarded by a Boolean expression. The *while* is a loop construct that repeats a process while an associated Boolean expression is true. The *seq* is a block construct combining a number of processes sequentially. The *par* is a block construct combining a number of processes in parallel.

A *communication channel* provides unbuffered, unidirectional point-to-point communication of values between two concurrent processes. The format and type of values are defined by a certain specified protocol.

Here follows the simplified grammar of NETDEF (Network Definition), in EBNF:

> *program ::= "NETDEF" id "IS" def-var process ".".*
> *process ::= assignment | skip | stop | if-t-e | while-do | seq-block | par-block.*

Our goal is to show that all NETDEF programs can be compiled into neural nets. There exists a dynamic system of the kind (6) that runs any NETDEF program on some given input. A first account on the concepts beyond the language NETDEF can be found in [Neto *et al*. 98].

2.3. Information Coding and Operators

With the guidelines provided in [Siegelmann 96], the seminal work on the implementation of information coding in neural networks (see [Gruau *et al*. 95] for a different approach), we introduce data types and show how to encode and keep them inside the information flow of neural nets. NETDEF has the following type definitions.

> *type ::= primitive-type | channel-type | composite-type.*
> *primitive-type ::= "BOOLEAN" | "INTEGER" | "REAL".*
> *channel-type ::= "CHANNEL".*
> *composite-type ::= "ARRAY" "[" number "]" "OF" primitive-type.*

2.3.1. Primitive Types

To be of some use, implementation of programming languages in neural nets should be carried out in a context of bounded resources. Herein we show how to use resource bounds to speed up computations over neural nets, through suitable encoding of suitable data types like in the usual programming languages.

To take into consideration the lower and upper saturation limits of the activation function σ, every value x of a given basic type is encoded into some value of [0,1]. For each type T, there is an injective encoding map α_T:T$\rightarrow$[0,1] mapping a value $x \in T$ onto its specific code. Basic types include: *boolean*, *integer* and *real*.

If resources are bounded, then there exists a limit to the precision of every value (in fact, even reals are bounded rationals). Considering a maximum precision of P digits, the minimum distance between any two values is 10^{-P}. Let us denote 10^{P} by M.

- For booleans, T is B = {TRUE, FALSE}, the encoding map is $\alpha_B(x) = \begin{cases} 0 & , x=\text{FALSE} \\ 1 & , x=\text{TRUE} \end{cases}$
- For integers, T is Z = $\{ -\frac{M}{2}, \ldots, \frac{M}{2} \}$ and $\alpha_Z(x) = \frac{M + 2x}{2M}$
- For reals within [a, b], $\alpha_{[a,b]}(x) = \frac{x-a}{b-a}$

2.3.2. Operators

Together with data types, many different operators are needed to process information. An arbitrary set of operators (with constants, variables and input data) forms an expression that after evaluation returns a result of some type. The net corresponding to each expression starts its execution when it receives signal IN (for details see section 2.4.1). After evaluation, it returns the final result through output RES and at the same time outputs signal OUT.

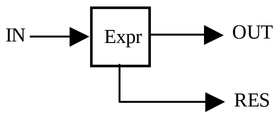

Fig. 2. An expression subnet. Non-labelled arcs default to weight 1.

The next figures show, for each operation, the computations needed to output the expected values through RES. Each subnet have an extra structure to receive the appropriate data and the input signal, and also to synchronize the result with output signal OUT. We present an example of it for binary integer sum, that is easily adapted to all the other operators.

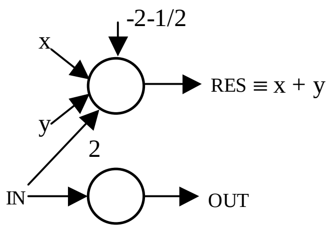

Fig. 3. Net structure for X + Y.

The extra –2 in the bias of the upper neuron stops the flowing of variables x and y activities, until an input signal arrives (overriding the extra –2 in order to exactly compute the sum).

• **Boolean Operators**. These are the typical McCulloch-Pitts Boolean operators (see [McCulloch and Pitts 43]).

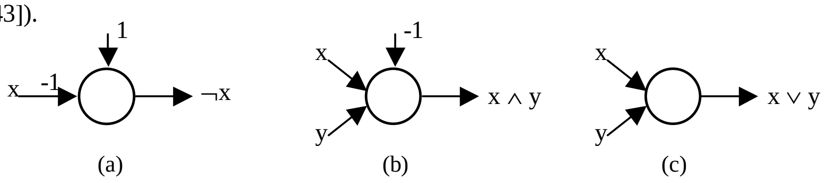

Fig. 4. Boolean operators: (a) NOT X, (b) X AND Y, (c) X OR Y.

• **Integer Operators.** There are arithmetical and relational operators for integers (M, presented in 2.3.1, is the maximum rational number possible to represent within the system bounded resources).

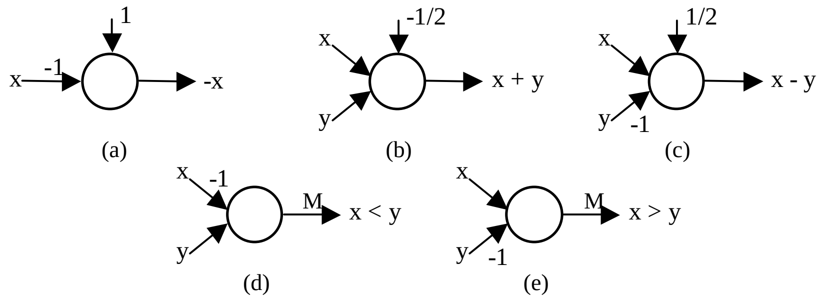

Fig. 5. Integer operators: (a) -X, (b) X + Y, (c) X - Y, (d) X < Y, (e) X > Y.

• **Real Operators.** The encoding $\alpha_{[a,b]}$ is a scaling of the interval [a,b] into [0,1]. Binary sum, subtraction and multiplication by a constant are straightforward.

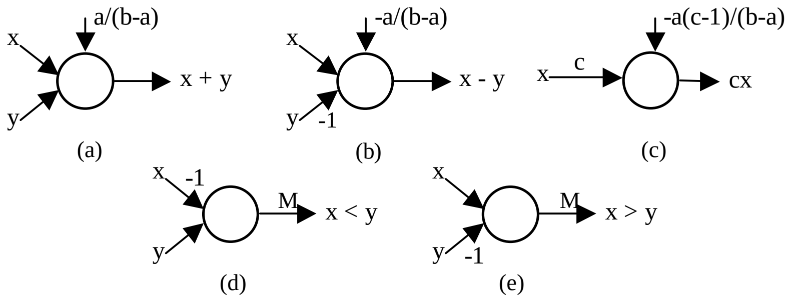

Fig. 6. Real operators: (a) X + Y, (b) X - Y, (c) cX, (d) X < Y, (e) X > Y.

2.3.3. Channel and Composite Types

Each channel is denoted by two neurons, one to keep the processed value and another neuron to keep a Boolean flag (with value one if the channel is empty, or zero otherwise). To know more about channels see section 2.4.2. It is also possible to define array variables. Each one of the data elements is coded by a specific neuron. This means that a composite type is a finite set of neurons. The array has the following structure (Fig. 7):

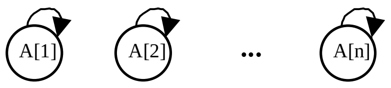

Fig. 7. The basic structure of an array.

The indexing of a position within the array is done by filtering the actual index value, in order to activate just the right element on the structure. These are the most complex neural networks of NETDEF, since they must perform dynamic indexing on fixed neural nets.

2.4. Synchronization Mechanisms

Neural networks are models of massive parallelism. In our model, at each instant, *all* neurons are updated, possibly with new values. This means that a network step with n neurons is a parallel execution of n assignments. Since programs (even parallel programs) have a sequence of well-defined steps, there must be a way to control it. This is done by a synchronization mechanism based on handshaking.

2.4.1. Instruction Blocks

There are two different ways to combine processes, the sequential block and the parallel block. Each process in a sequential block must wait until the previous process ends its computation. In a parallel block all processes start independently at the same time. The parallel block (which is itself a process) terminates only when all processes terminate. This semantics demands synchronization mechanisms in order to control the intrinsic parallelism of neural nets.

To provide with these mechanisms, each NETDEF process has a specific and modular subnet binding its execution and its synchronization part. Each subnet is activated when the value 1 is received through a special input validation line IN. The computation of a subnet terminates when the validation output neuron OUT writes value 1.

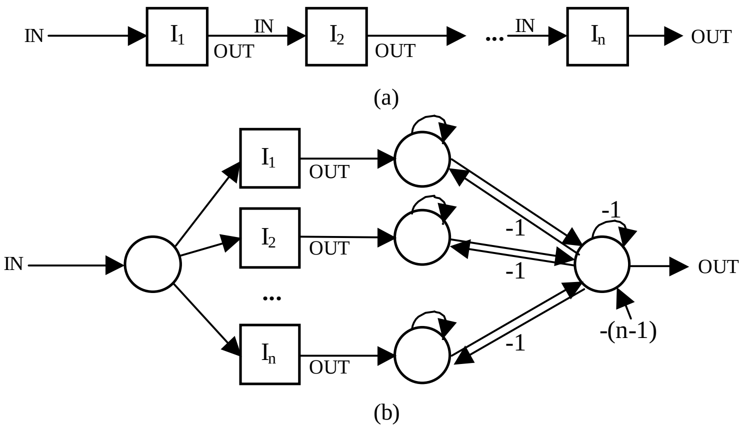

Fig. 8. Block processes: (a) SEQ I_1, …, I_n ENDSEQ,
(b) PAR I_1, …, I_n ENDPAR. All subnets are denoted by squares.

2.4.2. Occam® Channels

The language NETDEF assumes the Occam® channel communication protocol, allowing independent process synchronization.

We introduce two new processes, SEND and RECEIVE.

send ::= "SEND" id "INTO" id.

receive ::= "RECEIVE" id "FROM" id.

The process SEND sends a value through a channel, blocking if the channel is full, and process RECEIVE receives a value through the channel, blocking if the channel is empty, and waiting until some value arrives. To minimise the blocking nature of channels, see sections 2.7 and 2.8.

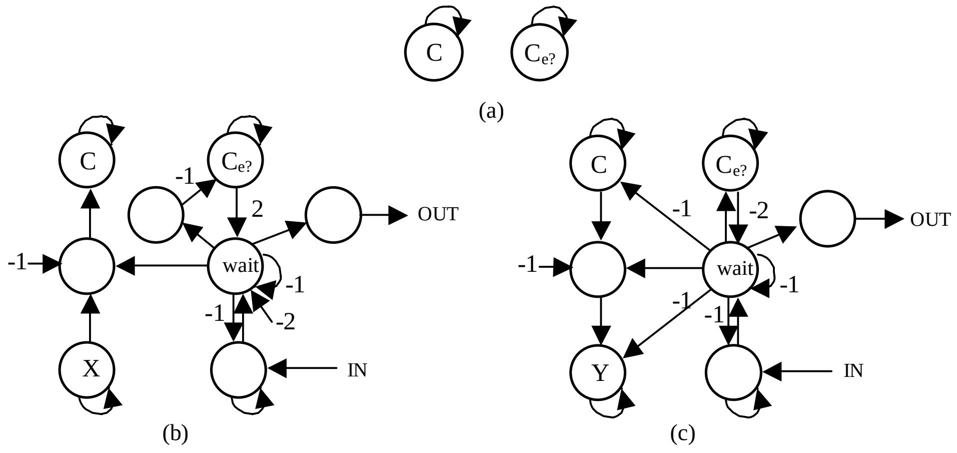

Fig. 9. Channel instructions: (a) VAR C : CHANNEL, (b) SEND X INTO C (c) RECEIVE Y FROM C.

Each channel has a limited memory of one slot. Using several channels in sequence, it is possible to create larger buffers. E.g.,

```
SEQ
 RECEIVE X1 FROM C1;       SEND X1 INTO C-TEMP;
 RECEIVE X2 FROM C-TEMP;   SEND X2 INTO C2;
ENDSEQ;
```

simulates a buffer with two elements.

2.4.3. Shared Variables

Processes can communicate through global variables (defined in the initial block). In principle, each neuron could see every other neuron in the net. The subject of variable scope is an priori restriction made by the compilation process. Several methods in the literature, like semaphores or monitors, are implemented as primitive instructions. These methods are used to promote mutual exclusion proprieties to a certain language, helping the programmer to solve typical concurrency problems. In NETDEF there is also a mutual exclusion mechanism for blocks, providing the same type of service.

2.5. Control Structure

The NETDEF program control structure consists of one block process (SEQ or PAR). This process denotes an independent neural net as seen before. The implementation is then recursive, because each process might correspond to a structure of several processes. The process subnets are built in a modular way, but they may share information (via channels or shared variables).

Besides the IN and OUT synchronization mechanism explained in 2.4.1, there is a special reset input for each instruction module. This reset is connected to every neuron of the subnet instruction with weight -1. So, if the signal one is sent through this channel, all neuron activations terminate in the next instant. For simplification, we do not show these connections. Once more, all subnets are represented by squares and non-labelled arcs default to weight 1.

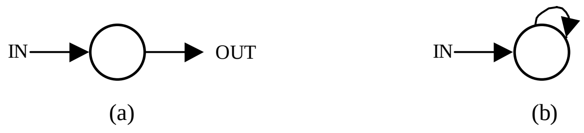

Fig. 10. Skip and Stop processes: (a) SKIP, (b) STOP.

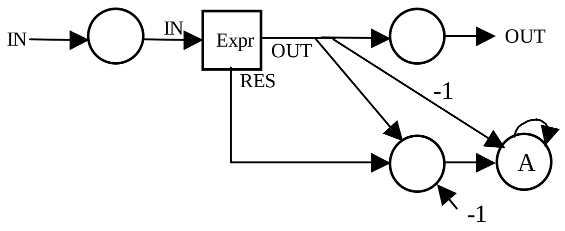

Fig. 11. Assignment process: A := EXPR. All subnets are denoted by squares.

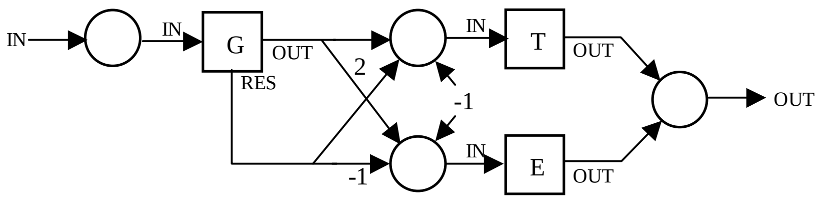

Fig. 12. Conditional process: IF G THEN T ELSE E. All subnets are denoted by squares.

The CASE process can be seen as a parallel block of IF processes. The COND process is a sequential block of IF processes.

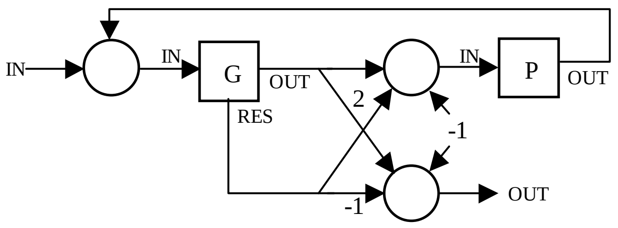

Fig. 13. Loop instruction: WHILE G DO P. All subnets are denoted by squares.

Other loop instructions (like REPEAT-UNTIL) are built in the same way.

2.6. Procedures and Functions

Functions and procedures do not have a specific neural network for each call. They have complex neural nets to ensure that just one call is executed at each time, blocking other calls until the end of execution. This makes effective lock mechanisms on shared data (e.g., accessing data through only one procedure). Functions and procedures have parameters by value (the value of the expression is duplicated into the procedure/function argument) and parameters by result (the value of the variable is duplicated into the argument and when the function/procedure call terminates, the value of the argument is assigned to the initial variable).

However, a drawback exists in NETDEF functions and procedures: there is no recursion. This is a complex problem, since the number of neurons is fixed by compilation. There is no easy way to simulate a stack mechanism of function calls into neural nets.

2.7. Input / Output

To handle input from the environment and output results, NETDEF uses the channel primitives with two special set of channels, IN_k (linked directly with input channel u_k) and OUT_k. The number of in/out channels is defined before compilation. This subject depends on the context of the application, so we do not define the architecture of these interfaces. In principle, input channels must have a FIFO list in order to keep the incoming data, and a structure to maintain the IN_k channels in a coherent state (i.e., update the channel flag of IN_k each time u_k sends a value).

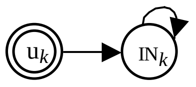

Fig. 14. Input channel u$_k$ connects with NETDEF channel IN$_k$.

In this way, in/out operations are simple channel calls. An in/out example could be,

```
SEQ
 RECEIVE a FROM IN1;
 SEND a INTO OUT2;
ENDSEQ;
```

This process inputs data through the variable 'a' from the first input channel, and sends it through the second output channel.

To obtain asynchronous in/out, there is a boolean function ISEMPTY(channel) returning true if the channel is empty, or false otherwise.

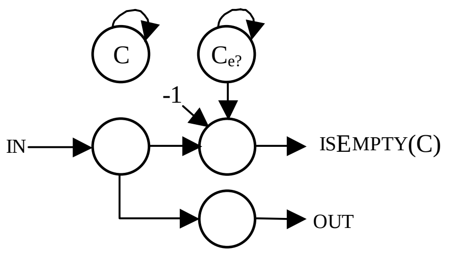

Fig. 15. Asynchronous in/out.

E.g., to implement an asynchronous output,

```
IF ISEMPTY(C)THEN SEND X INTO C
```

2.8. Timers

In real applications, some processes may create deadlock situations. The NETDEF communication primitives (SEND and RECEIVE) are blocking, i.e., they wait until some premises are satisfied (the channel must be empty for SEND and full for RECEIVE). If these premises are never satisfied, then we have a problem: we cannot wait indefinitely for input in real-time applications. To handle this problem, NETDEF has several timer processes.

The first one is TRY. It guarantees termination, if the execution of an instruction does not terminate before the expected time (held by an integer variable).

timed-instruction ::= "TRY" "(" variable ")" instruction.

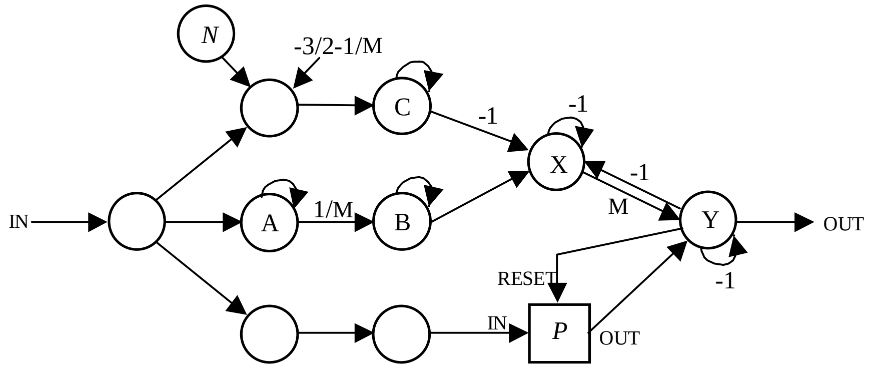

Fig. 16. Timer constructor: TRY(N) P. All subnets are denoted by squares.

Neuron X has arcs with weight –M and neuron Y has arcs with weight –1 to neurons A, B and C.

Two other types of timers exist: delay-timers and cyclic-timers. Delay timers delay the execution of instructions during a given time.

delayed-instruction ::= "DELAY" "(" variable ")" instruction.

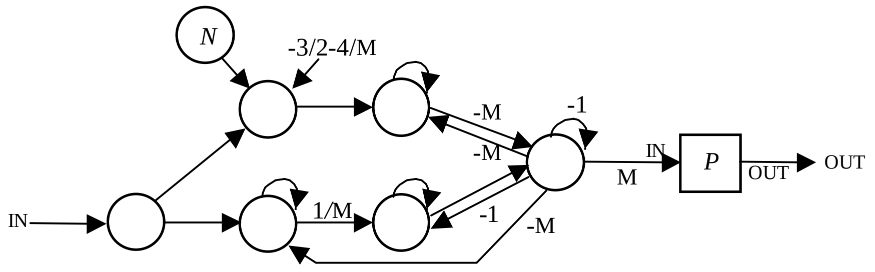

Fig. 17. Timer constructor: DELAY(N) P. All subnets are denoted by squares.

Cyclic-timers restart the execution of an instruction whenever a specific time passed. They can be used to simulate interrupts.

cyclic-instruction ::= "CYCLE" "(" variable ")" instruction.

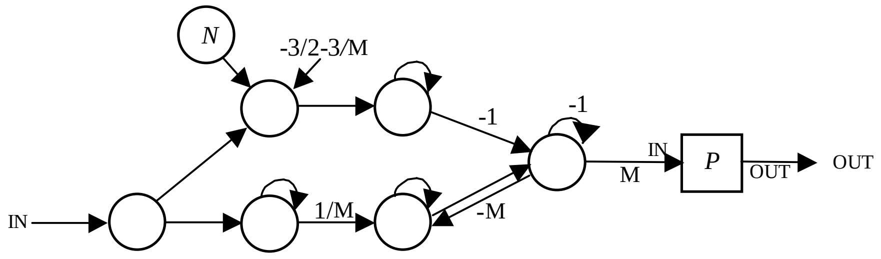

Fig. 18. Timer constructor: CYCLE (N) P. All subnets are denoted by squares.

Several timer constructs can be used sequentially. E.g., the process

```
CYCLE (10000) TRY(50) IF flag = 1 THEN SEND X INTO C;
```

means that on each 10 000 cycles, it will check if an integer variable ‘flag’ has value 1. If it has, the timer sends the value of X through channel C. If the variable cannot be sent in 50 cycles, the timer aborts execution.

2.9. Exceptions

In high-level languages, like Eiffel® [Interactive 89] or Ada® [USDD 83], exceptions are unexpected events occurring during the execution of a system and disrupting the normal flow of execution (e.g., division by zero or an operation overflow). Some exceptions are raised by the system, others by the program itself. The (hypothetical) neural net hardware is homogenous; there is no system disruption other than neuron or sinapse failure.

Despite the possibility of system failures, our concern herein will be only about programmer raised exceptions. These exceptions add some extra block control. They appear as part of a SEQ or PAR block. First, an example,

```
meth1-failed := FALSE;
SEQ
IF meth1-failed  THEN method-1 -- with a ‘RAISE excp-1’
                 ELSE method-2; -- with a ‘RAISE excp-2’
 job-accomplished := TRUE;
EXCEPTION
 WHEN excp-1 DO
 SEQ
  meth1-failed:= TRUE;
  RETRY;
  ENDSEQ;
 WHEN excp-2 DO
 SEQ
  job-accomplished := FALSE;
  TERMINATE;
 ENDSEQ;
ENDSEQ;
```

Suppose we have two methods to do the same work. In this sequential block, if the first method fails, it raises an exception called ‘excp-1’ trapped by the handler feature of the block. It changes the value of the Boolean variable and then executes the block again, trying the second method. If this also fails, then the block terminates with a no job accomplishment status.

Process RAISE E raises exception E. To each exception corresponds an associated process (that can be again a SEQ or PAR block), and some special block handlers. These block handlers define what to do with the actual block:

- RETRY – reset and execute the block again,
- ABORT – reset and terminate the block,
- PROPAGATE – reset the block and raise the same exception in the upper block.

Each instruction block with an exception feature has its net architecture changed.

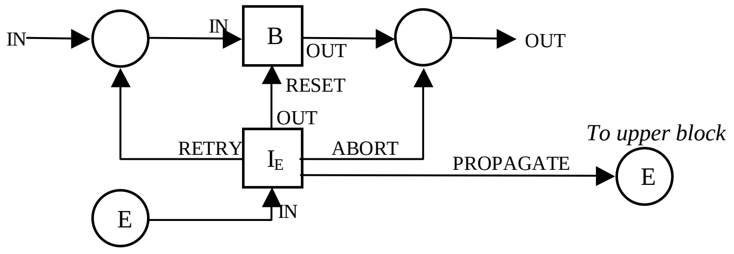

Fig. 19. The exception handler E of instruction block B.
Block I_E is the instruction associated with exception E.
All subnets are denoted by squares.

Each exception has a specific neuron receiving the block process signal. With this type of structure, RAISE is defined as,

```
PAR
 E:=1;   ≡      RAISE E
 STOP;
ENDPAR;
```

There is a cascade effect for no handled exceptions. If a block raises an exception E but has no handler for it, the compiler inserts by default the following handler,

```
WHEN E DO PROPAGATE;
```

Any process sending this signal is not resumed (unless one of the upper blocks retries its execution).

2.10. Space and Time Complexity

The proposed implementation map is able to translate any given NETDEF program to an analog recurrent (rational) neural net that performs the same computations. We wonder what is the space complexity of the implementation, i.e., how many neurons are needed to support a given NETDEF program? We take a close look at each basic process to evaluate its contribution to the size of the final net. The *assignment* inserts 3 neurons plus those that are needed to compute the expression. The SKIP and STOP need only one neuron. The IF-THEN-ELSE needs 4, and the WHILE needs 5 neurons. The SEQ statement needs no neurons and the PAR of n processes needs n+2 neurons. SEND needs 5 neurons and RECEIVE needs 4. Timers also have constant number of neurons. All other processes exhibit the same behaviour with respect to the number of neurons. Data types and operators need a number of neurons linear in the size of the used information. All expressions can be evaluated with a number of neurons linear in the number of neurons needed to hold data. Every process adds a constant or linear complexity to the final net, the same result presented in [Siegelmann 93] for neural nets of the same kind. The spatial complexity of the emulation is linear on the size of the algorithm. Concerning time complexity, each subnet executes its respective command with a linear delay. NETDEF adds a linear time slowdown to the complexity of the corresponding algorithm.

3. Compiling a Process

For a better understanding, let's see how process "WHILE b DO x := x+1" is translated into a neural network using the NETDEF compiler.

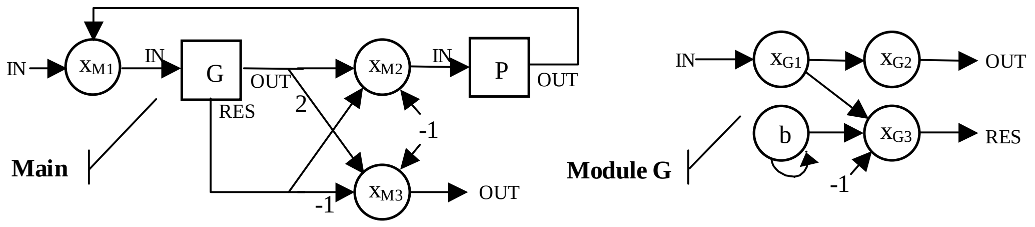

Fig. 20. Main module and module G. All subnets are denoted by squares.

The synapse IN sends value 1 (by some neuron x_{IN}) into x_{M1} neuron, thus starting the computation. Module G (denoted by a square) computes the value of Boolean variable 'b' and sends the 0/1 result through synapse RES. This result is synchronized with an output of 1 through synapse OUT. The next two neurons decide between stopping the process ('b' is false) or executing module P ('b' is true), iterating again. The dynamic system is described by the following equations:

$$x_{M1}(t+1) = \sigma(\ x_{IN}(t) + x_{P2}(t)\ )$$
$$x_{M2}(t+1) = \sigma(\ x_{G2}(t) + x_{G3}(t) - 1.0\ )$$
$$x_{M3}(t+1) = \sigma(\ 2.x_{G2}(t) - x_{G3}(t) - 1.0\ )$$

Module G just accesses the value of ‘b’ and outputs it through neuron x_{G3}. This is achieved because x_{G3} bias –1.0 is compensated by value 1 sent by x_{G1}, allowing the value of ‘b’ to be the activation of x_{G3}. This module is defined by:

$$x_{G1}(t+1) = \sigma(\ x_{M1}(t)\ )$$
$$x_{G2}(t+1) = \sigma(\ x_{G1}(t)\ )$$
$$x_{G3}(t+1) = \sigma(\ x_{G1}(t) + b(t) - 1.0\ )$$

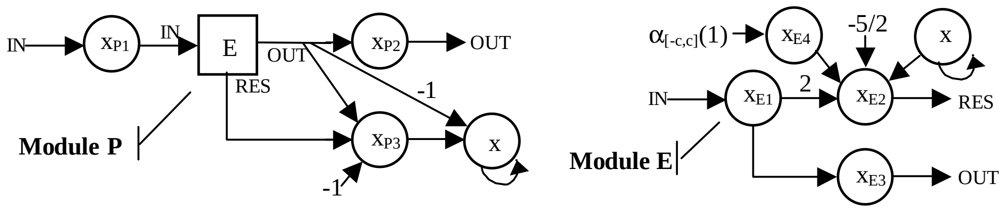

Fig. 21. Modules P and E. All subnets are denoted by squares.

Module P makes an assignment to the real variable ‘x’ with the value computed by module E. Before neuron x receives the activation value of x_{P3}, the module uses the output signal of E to erase its previous value.

$$x_{P1}(t+1) = \sigma(\ x_{M2}(t)\ )$$
$$x_{P2}(t+1) = \sigma(\ x_{E3}(t)\ )$$
$$x_{P3}(t+1) = \sigma(\ x_{E2}(t) + x_{E3}(t) - 1.0\ )$$

In module E the increment of ‘x’ is computed (using $\alpha_{[-c,c]}(1)$ for the code of real 1, where ‘c’ is a predefined value set by the compiler). The extra –1/2 bias of neuron x_{E2} is necessary due to the internal coding:

$$x_{E1}(t+1) = \sigma(\ x_{P1}(t)\ )$$
$$x_{E2}(t+1) = \sigma(\ 2.x_{E1}(t) - x_{E4}(t) + x(t) - 5/2\ )$$
$$x_{E3}(t+1) = \sigma(\ x_{E1}(t)\ )$$
$$x_{E4}(t+1) = \sigma(\ \alpha_{[-c,c]}(1)\ )$$

The dynamics of neuron x is given by:

$$x(t+1) = \sigma(\ x(t) + x_{P3}(t) - x_{E3}(t)\ )$$

However, if neuron x is used in other modules, the compiler will add more synaptic links corresponding to the new dynamic equation for x

4. Conclusions

We introduced the core of a new language, NETDEF. NETDEF develops an easy way to build neural nets performing arbitrarily complex computations. This method is modular, where each process is mapped in an independent neural net. Modularity brings great flexibility. For example, if a certain task is programmed and compiled, the resulting net is a module that can be used elsewhere.

The use of finite neural networks as deterministic machines to implement arbitrarily complex algorithms is now possible by the automation of compilers like NETDEF. If someday, neural net hardware would be as easy to build as von Neumann hardware, then the NETDEF approach will provide a way to insert algorithms into the massive parallel architecture of artificial neural nets. To test our program, able to compile and simulate the dynamics of neural nets described in this paper, go to http://www.di.fc.ul.pt/~jpn/netdef/netdef.htm.

5. References

Interactive Software Engineering Inc., Eiffel®: the Language. TR-EI 17/RM, 1989.

F. Gruau, J. Ratajszczak, and G. Wiber. A Neural Compiler. *Theoretical Computer Science* 141 (1-2), 1-52, 1995.

B. Lester. The Art of Parallel Programming. Prentice Hall, 1993.

W. McCulloch and W. Pitts. A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics* 5, 115-133, 1943.

M. Minsky. Computation: Finite and Infinite Machines. Prentice Hall, 1967.

J. P. Neto, H. Siegelmann, J. F. Costa, and C. S. Araujo. Turing Universality of Neural Nets (revisited). *Lecture Notes in Computer Science 1333,* Springer-Verlag, 361-366, 1997.

J. P. Neto, H. Siegelmann, J. F. Costa, On the Implementation of Programming Languages with Neural Nets, *First International Conference on Computing Anticipatory Systems,* CASYS 97, CHAOS, [1], 201-208, 1998.

H. Siegelmann. Foundations of Recurrent Neural Networks. Technical Report DCS-TR-306, Department of Computer Science, Laboratory for Computer Science Research, The State University of New Jersey Rutgers, October 1993.

H. Siegelmann. On NIL: The Software Constructor of Neural Networks. *Parallel Processing Letters* 6(4), World Scientific Publishing Company, 575-582, 1996.

H. Siegelmann. Neural Networks and Analog Computation, Beyond the Turing Limit. Birkhauser, 1999.

SGS-THOMSON, Occam® 2.1 Reference Manual, 1995.

E. Sontag. Mathematical Control Theory: Deterministic Finite Dimensional Systems. Springer-Verlag, New York, 1990.

United States Department of Defence. Reference Manual for the Ada® Programming Language. American National Standards Institute Inc., 1983.