IMPLEMENTATION OF PROGRAMMING LANGUAGES WITH NEURAL NETS

J. Pedro Neto[1], Hava T. Siegelmann[2], and J. Félix Costa[1]
jpn@di.fc.ul.pt, iehava@ie.technion.ac.il, and fgc@di.fc.ul.pt

[1]Faculdade de Ciências da Universidade de Lisboa
BLOCO C5 - PISO 1, 1700 LISBOA, PORTUGAL
[2]Faculty of Industrial Engineering and Management
TECHNION CITY, HAIFA 32 000, ISRAEL

Abstract: In this paper we present the implementation of control structures of a high-level concurrent programming language into the machine level of analog recurrent neural nets. Neural nets can thus perform high-level symbolic tasks, together with their reputed subsymbolic learning capabilities. We emphasize the construction of neural software capable of integrating both learning and programmed control modules.

Keywords: Analog Computation, Symbolic and Subsymbolic Computation, Neural Computation, Recurrent Neural Nets.

1.The Model

In this paper we show that programming languages are implementable on neural nets, namely, neural nets can be designed to solve any (computable) high level programming task. In particular we show how to emulate an OCCAM-like programming languages on nets. We implemented a compiler and it is offered for the interested reader from e-mail: jpn@di.fc.ul.pt. Our compiler can be used to built large scale neural nets that integrate learning and control structures. We use a very simple model of analog recurrent neural nets and a number-theoretic approach. Our results can be generalised to other models of neural computation, and also to support structured data types (through coding techniques).

The use of such a model for computability analysis is due to Hava Siegelmann and Eduardo Sontag. In [Siegelmann and Sontag 92, Siegelmann and Sontag 95] Hava Siegelmann and Eduardo Sontag used it to establish lower bounds on the computational power of analog recurrent neural nets.

An analog recurrent neural net is a dynamic system with aplication map of the form

$$\vec{x}(t+1)= \phi(\vec{x}(t),\vec{u}(t)) \qquad (1)$$

where $x_i(t)$ denotes de activity (firing frequency) of neuron i at time t within a population of N interconected neurons, and $u_i(t)$ the input bit of input stream i at time t within a set of M input channels. The aplication map ϕ is taken as a composition of an affine map with a picewise linear map of the interval [0,1], known as the saturated sigmoid:

$$\sigma(x) = \begin{cases} 0 \text{ if } x<0 \\ x \text{ if } 0\leq x\leq 1 \\ 1 \text{ if } x>1 \end{cases} \tag{2}$$

The dynamic system becomes

$$x_i(t+1) = \sigma(\sum_{j=1}^{N} a_{ij}x_j(t) + \sum_{j=1}^{M} b_{ij}u_j(t) + c_i) \tag{3}$$

where a_{ij}, b_{ij} and c_i are rational weights (and therefore Turing computable).

Within this model (a number-theoretic model) integers are coded as rational numbers in]0,1[. Since the sigma function is linear in the unit interval, neurons can hold values with unbounded precision (but always finite). In order to work with bounded resources some fixed precision must be assumed from the very begining (like *maxint* in the PASCAL programming language). We adopt the representation,

$$0 \equiv 0.1,\ 1 \equiv 0.01,\ 2 \equiv 0.011,\ \ldots,\ n \equiv 0.01^n \tag{4}$$

Negative numbers can also be considered as follows:

$$-1 \equiv 0.11,\ -2 \equiv 0.111,\ \ldots,\ -n \equiv 0.1^{n+1} \tag{5}$$

We assume that all inputs are previously coded before the computation starts, using a stack technique similar to that found in, e.g. [Siegelmann and Sontag 95], and decoded after the computation. This notation has already been used in [Neto *et al* 96].

Our problem will be to find a net

$$x_i(t+1) = \sigma(\sum_{j=1}^{N} a_{ij}x_j(t) + c_i) \tag{6}$$

for each program written in a suitable programming language.

2.The Language NETDEF

We will adopt a 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 95]). With this language a program can be described as a collection of processes executing concurrently, and communicating with each other through channels. These two 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 each of which is 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 sequential construct, combining a number of processes which are performed sequentially. The *par* is a parallel construct, combining a number of processes which are performed concurrently.

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 for NETDEF, in EBNF:

program ::= header block.
header ::= "input" id {"," id} "output" id {"," id}.
block ::= "netdef" instruction { ";" instruction } ".".
instruction ::= def-var | attribution | skip | if-then-else | while-do | seq-block | par-block.
def-var ::= "var" id {"," id}.
attribution ::= id ":=" expression.
skip ::= "skip".
if-then-else ::= "if" expression "then" instruction "else" instruction.
while-do ::= "while" expression "do" instruction
seq-block ::= "seq" instruction { ";" instruction } "endseq".
par-block ::= "par" instruction { ";" instruction } "endpar".

Now we show that all NETDEF programs can be compiled into neural nets. There exists a dynamic system of the kind (6) that runs any Occam® program on some given input. This 'universal' neural net is a neural computer able of performing symbolic and subsymbolic computation. A different but seminal approach to the neural net software construction is found in [Siegelmann 96].

3.The Implementation Map

Now we introduce the major constructions of our implementation map, but leaving details to the full paper. Each NETDEF command denotes an independent neural subnet. Subnets may share variables (and channels, as we will see later on). The implementation map is recursive, because each block might correspond to a set of several instructions. Each subnet is activated when the bit 1 is received through the input validation line IN. The computation of a subnet terminates when the validation output neuron OUT writes bit 1, signaling to the following module the availability of the result at that precise moment. Using this method, we can easily control all synchronizations.

Subnets are denoted by squares and non-labelled arcs default to weight 1.

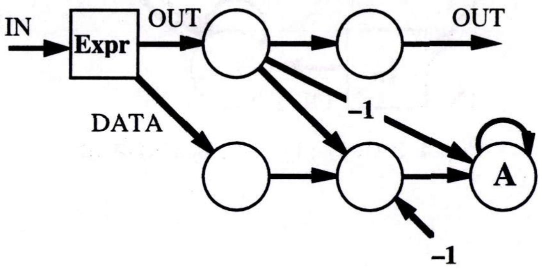

Fig. 1. A := Expr

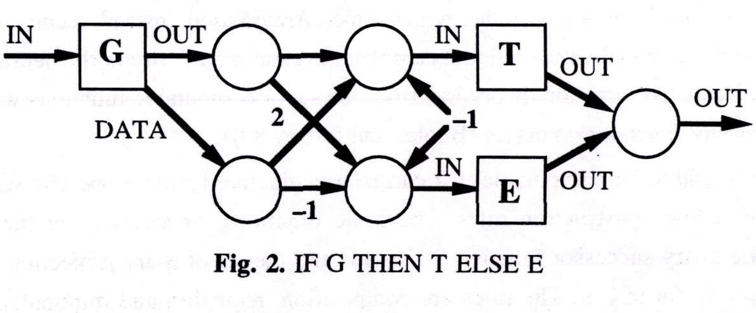

Fig. 2. IF G THEN T ELSE E

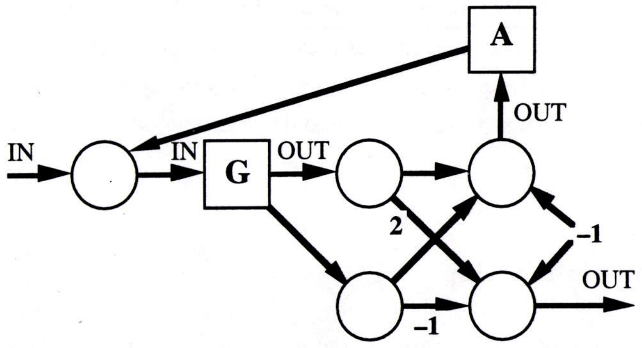

Fig. 3. WHILE G DO A

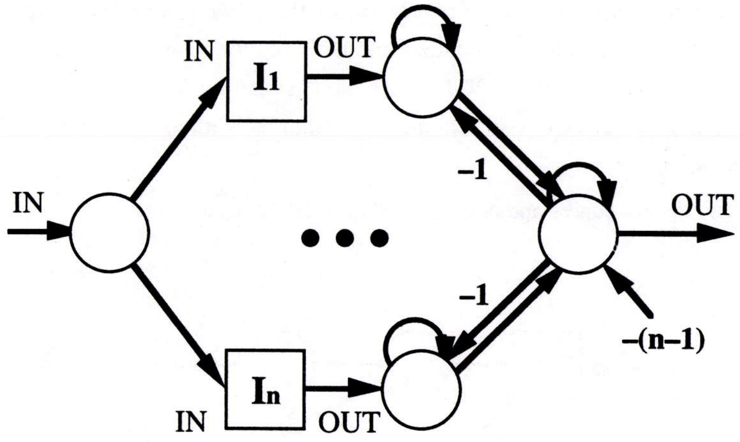

Fig. 4. PAR I_1 ; I_2 ; ... ; I_n ENDPAR

4.Turing Power

The first (constructive) proof of Turing's power of rational neural nets was given by Hava Siegemann and Eduardo Sontag in 1992, closing an open question since the fourties.

In [Neto *et al* 96] the three of us together with Carmen Araujo used, instead, recursive function theory to provide insigts of computational completeness and modularity in the neural network construction. Recursive function theory identifies the set of computable functions with the set of partial *recursive functions* on ω (see [Boolos and Jeffrey 80]).

A function f is said to be computable if it can be manufactured from a specific set of basic functions and a few construction rules. The basic functions, or axioms, are the zero-ary constant **0**, the unary successor function S(x)=x+1, and the set of n-ary projection functions $U_{i,n}(x_1,...,x_n)= x_i$, for $i \in 1..n$. The rules are composition, recursion and minimalisation (for more details see [Neto *et al* 96]).

We will see that these constructors can be clearly and easily implemented with NETDEF commands. In each command we make use of function calls. Each function should be replaced with the specific NETDEF command.

ZERO

```
DATA_OUT:=0;
```

SUCCESSOR

```
DATA_OUT:=x+1;
```

PROJECTION $U_{i,n}$

```
par
   if i=1 then DATA_OUT:=x_1;
   if i=2 then DATA_OUT:=x_2;
   ...
   if i=n then DATA_OUT:=x_n
endpar;
```

COMPOSITION

```
var y_1,...,y_k;
seq
   par
      y_1:=f_1(x_1,...,x_n);
      ...
      y_k:=f_k(x_1,...,x_n)
   endpar;
   DATA_OUT:=g(y_1,...,y_k)
endseq;
```

RECURSION

```
var k,h;
seq
   par
      k:=0;
      h:=f(x_1,...,x_n)
   endpar;
   while y<>k do
      seq
         h:=g(x_1,...,x_n,k,h);
         k:=k+1
      endseq;
      DATA_OUT:=h
endseq;
```

MINIMALISATION

```
var y,k;
seq
   y:=0;
   k:=f(x_1,...,x_n,y);
   while k<>0 do
      seq
         y:=y+1;
         k:=f(x_1,...,x_n,y)
      endseq;
   DATA_OUT:=y
endseq;
```

The six commands given above implement the base functions and construction rules of recursive function theory. Hence, NETDEF expresses all partial recursive functions.

5.Complexity of Computations

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 next calculate the space complexity of the implementation, i.e., how many neurons are needed to support a given NETDEF program?

The *assignment* inserts 5 neurons plus those that are needed to compute the expression. The *skip* needs only one neuron. The *if-then-else* and the *while* statements need 5 neurons plus those that are needed for the evaluation of the guard. The *seq* statement requires 1 neuron and the par of n statements asks for n+2 neurons. All expressions can be evaluated with a linear growth in that number. Every command adds a constant or linear complexity to the final net. The spatial complexity of the emulation is linear in the size of the program.

Concerning time complexity, each subnet executes its respective Occam® command with a constant delay. NETDEF adds a linear time slow down to the complexity to the corresponding program.

6.Channel Synchronisation

NETDEF also assumes the Occam® channel communication protocol, allowing for the synchronisation of two independent instructions. We introduce three new instructions *cha, send* and *receive*.

def-cha ::= "cha" id {"," id}.
send ::= "send" id "into" id.
receive ::= "receive" id "from" id.

cha defines a new channel, *send* sends an integer through a channel, blocking the process if the channel is full, and *receive* receives an integer through the channel, blocking if the channel is empty, and waiting until something arrives. Each channel has memory just for one integer. Using several channels in sequence, it is possible to create larger buffers.

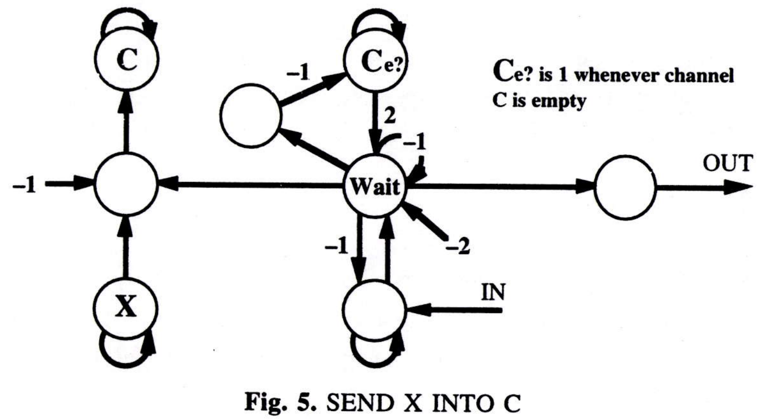

Fig. 5. SEND X INTO C

7.Non Determinism

The *choose* construct introduces non determinism in NETDEF. Command *choose* non deterministicaly selects one of its arguments and executes it. To do so, it includes an oracle that outputs a binary value. This value is random or given in some predefined list of values.

choose ::= "choose" id {";" id} "endchoose".

We next show an example with only two arguments, but the same method can be used for n choices, with a need of $\log_2$ n oracles. In neuron x_{guess}, the oracle *guess* inserts a 0 or 1 into its input.

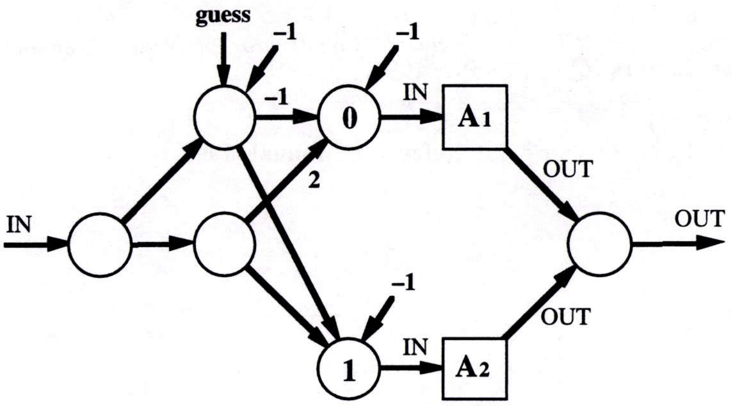

Fig. 6. CHOOSE A_1 ; A_2 ENDCHOOSE

8.Summary

We showed how to handle control structures in neural networks of a picewise-linear activation function to implement programming languages. We provided the construction for the concurrent Occam® programming language. The programmability of neural networks to perform higher level programming tasks, together with their recognised learning capabilities provide an interesting workbench for the integration of symbolic and subsymbolic computation.

References

[Boolos and Jeffrey 80]

Boolos, G. and Jeffrey, R., **Computability and Logic**, *Cambridge University Press*, 1980.

[Neto *et al* 96]

Neto, J. Pedro, Siegelmann, Hava T., Costa, J. Félix, and Carmen Suárez Araujo, *Turing Universality of Neural Nets Revisited*, 1996, Lecture Notes in Computer Science, Springer-Verlag, to appear.

[Siegelmann and Sontag 91]

Siegelmann, H. and Sontag, E., *Neural Nets Are Universal Computing Devices*, SYCON Report 91-08, Rutgers University, 1991.

[Siegelmann and Sontag 95]

Siegelmann, H. and Sontag, E., *On the Computational Power of Neural Nets*, **Journal of Computer and System Sciences** [50] 1, Academic Press, 1995, 132-150.

[Siegelmann 96]

Siegelmann, H. *On NIL: The Software Constructor of Neural Networks*, **Parallel Processing Letters** [6] 4, 575-582, 1996.

[SGS-Thomsom 95]

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