INFORMATION CODING AND NEURAL COMPUTING

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 [2,5] it is showed that programming languages can be translated into recurrent 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 the context of bounded resources. With the guidelines provided in [4,5], we introduce data types and show how to encode and keep them inside the information flow of neural nets.

1 Introduction

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 $u_i(t)$ the input bit of input stream i 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:

$$\sigma(x) = \begin{cases} 0 & \text{if } x<0 \\ x & \text{if } 0\le x\le 1 \\ 1 & \text{if } x>1 \end{cases} \qquad (1)$$

The dynamic system becomes

$$x_i(t+1) = \sigma\Big(\sum_{j=1}^{N} a_{ij}x_j(t) + \sum_{j=1}^{M} b_{ij}u_j(t) + c_i \Big) \qquad (2)$$

where a_{ij}, b_{ij} and c_i are rational weights (and therefore Turing computable). The use of such a model for computability analysis is due to Hava Siegelmann and Eduardo Sontag. In [3] they used it to establish lower bounds on the computational power of analog recurrent neural nets. Our problem will be to find a net with no input streams and with inputs encoded in the initial state $\vec{x_0}$

$$x_i(t+1) = \sigma\Big(\sum_{j=1}^{N} a_{ij}x_j(t) + c_i \Big) \qquad (3)$$

for each program written in a suitable programming language.

In [2] (see [5] for a seminal presentation), we showed that programming languages can be translated on recurrent (rational) neural nets. The goal of this implementation was not efficiency but simplicity. Indeed we used a number-theoretic approach to machine programming, where (integer) numbers were encoded in an unary fashion,

introducing an exponential slowdown in the computations with respect to a two-symbol (plus a blank character) tape Turing machine.

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, implementation of programming languages in neural nets should be carried in a context of bounded resources. Herein we show how to use resource boundedness to speed up computations over neural nets, through suitable encoding of suitable data types like in the usual programming languages. With the guidelines provided in [4,5], we introduce data types and show how to encode and keep them inside the information flow of neural nets.

2 Data Types

Every value *x* of a given data type, is encoded into some value of [0,1], to take in consideration the lower and upper saturation limits of the activation function σ. For each data type T, we will have some injective encoding map $\alpha_T: T \rightarrow [0,1]$ that maps a value $x \in T$ onto its specific code. The encoding map will determine the neural architecture of the operators.

Data types include: *boolean*, *scalars*, *integer*, *real*, and *list* (keeping elements of a given type). If resources are bounded, then there exists a limit to the precision of every value (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.

2.1 Pre-defined types

- For booleans, with B = {0,1}, the encoding map is $\alpha_B(x) = \begin{cases} 0 & , x=\text{FALSE} \\ 1 & , x=\text{TRUE} \end{cases}$
- For scalars (see [4]), with S = { 0, 1, ... , n }, the map is $\alpha_S(x) = \frac{2x + 1}{2n}$
- For integers, with $Z = \{ -\frac{M}{2}, \ldots, \frac{M}{2} \}$, the map is $\alpha_Z(x) = \frac{M + 2x}{2M}$
- For reals, with R = [a,b], the map is $\alpha_R(x) = \frac{x - a}{b - a}$
- For a lists of scalars of type S with cardinality n (see [4]), the map is
  $\alpha_{\mathcal{L}}(L) = \sum_{i=1}^{k} \frac{\alpha_S(x_i)}{(2n)^{i-1}}$ with $L = < x_1, \ldots, x_k >$, $x_i \in S$

2.2 Arrays and Records

There is also the possibility to build complex data types with array and record constructors. Each structured data type consists of several data elements of the same type (if it is an array), or of different types (if it is a record). Each one of the data elements is coded by a specific neuron. This means that a structured data type is a

finite set of neurons. For example, the assignment of one structure to another of the same type, is just a parallel assignment of every element of the first to the specific element of the second.

3 Operators

With the introduction of data types, many different operators will be needed to process information. There are arithmetic, logical and relational functions available on integers. All expressions when evaluated returns a result of a specific type. The net corresponding to each expression starts its execution when it receives an input signal. After evaluation, it returns the final result through a special output channel (named RES) and at the same time outputs an end signal (through special channel named OUT). Each expression net have an extra structure to receive the appropriate data and the input signal, and also to synchronize the result with output signal OUT. We show some nets that implement some operators. Weight 1 is default for non-labelled arcs.

3.1 Boolean Operators

These are the McCulloch-Pitts boolean operators (see [1]).

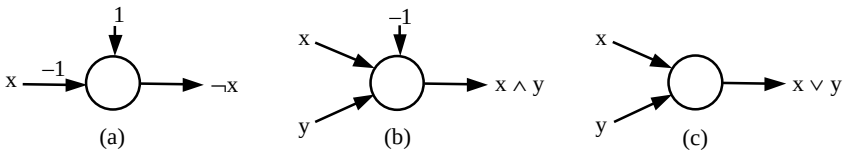

Fig. 1. Boolean operators: (a) NOT x (b) x AND y (c) x OR y.

3.2 Scalar Operators

There are a set of standard order operators for scalar types (see [4] for more information). Notice that SUCC is not well defined for the last element, and PRED is not well defined for the first (i.e., they return invalid values in those cases). ORD returns the ordinal value of the argument.

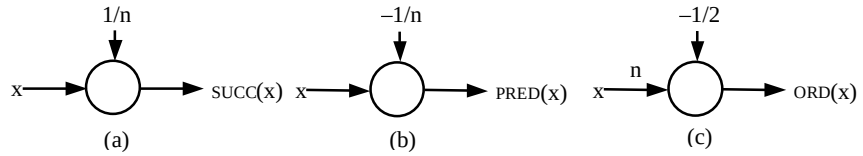

Fig. 2. Scalar operators: (a) SUCC(x) (b) PRED(x) (c) ORD(x).

3.3 Integer Operators

These operators are the standard ones used on high-level languages, like PASCAL. There are arithmetical and relational operators for integers (remember that M is the maximum rational number possible to represent with the limited resources available).

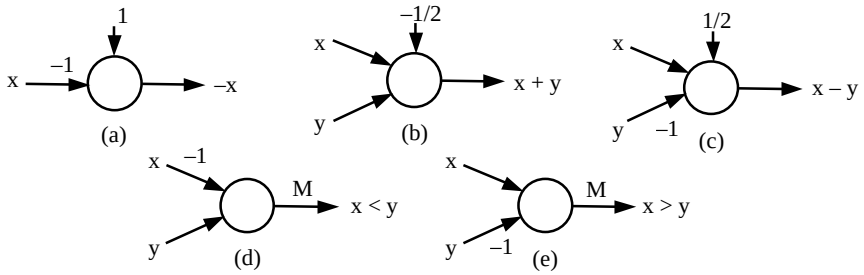

Fig. 3. Integer operators: (a) –X , (b) X + Y, (c) X – Y, (d) X < Y, (e) X > Y.

To build the other relational operators, the following expressions can be used:

$x=y \Leftrightarrow \neg(x>y \vee x<y)$ $\quad x\neq y \Leftrightarrow x>y \vee x<y$

$x\geq y \Leftrightarrow x>y \vee x=y$ $\quad x\leq y \Leftrightarrow x<y \vee x=y$

3.4 Real Operators

The encoding α_R is a scaling of the interval [a,b] into [0,1]. Binary sum, subtraction and multiplication by a constant are straightforward.

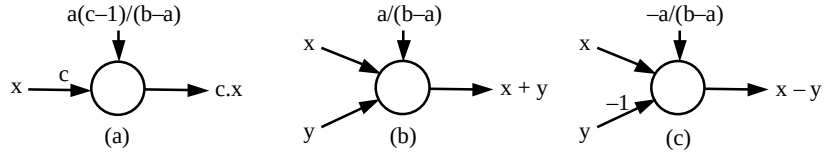

Fig. 4. Real operators: (a) c.x (b) x + y (c) x – y.

The main problem with the scaling technique lies in the relational operators. Two reals may be arbitrarily close to each other. How to compare them? In this paper, we discuss information encoding with bounded resources. So, there is a limit to the precision of every real (in fact, all values are bounded rationals) and number M can also be used to produce the required results.

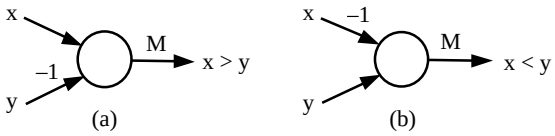

Fig. 5. Real relational operators: (a) x > y (b) x < y.

3.5 List Operators

A list keeps a set of finite elements of a certain scalar type with cardinality *n* (with possible repetitions). The encoding map $\alpha_{\mathcal{L}}$ has the 2*n*-Cantor set as codomain. Not every point on [0,1] is used, just some points are valid list values. When a value is inserted, the previous list is divided by 2*n*, and only then it is added the new value (encoded via α_S). So, the input and output of a list obeys to a First In - Last Out discipline. The empty list has value zero.

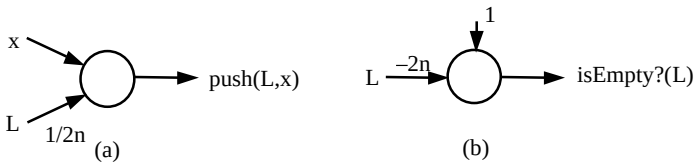

Fig. 6. List operators: (a) push(L,x) (b) isEmpty?(L).

4 Conclusions

Data types and control structures are part of a suitable programming language called NETDEF. Each NETDEF program has a specific neural net that simulates it. These nets have a strong modular structure and a synchronisation system that allows the sequential and parallel execution of subnets despite the massive parallel feature of neural nets.

Each instruction denotes an independent neural net. They may access external variables (and use channels to communicate). The implementation map is modular, because each block corresponds to a set of several independent instructions (see [2] for details). There are constructors for assignments, conditional instructions and loop instructions. There are also sequential and parallel instruction blocks. Modularity brings great flexibility. For example, with this type of neurons, binary multiplication (integer or real) is not trivial. To have multiplication we need to build a program for it. But once this program has been made, it can be used elsewhere in the same way as a basic operator.

This is the language core, many other features can be implemented using the same method, like channels for module to module communication, synchronous and asynchronous input/output, function definition and clocks. If someday, neural net hardware be as easy to build as the usual von Neumann hardware, then the NETDEF approach will provide a way to insert algorithms into the massive parallel architecture of artificial neural nets.

5 References

1. McCulloch, W. and Pitts, W., *A logical calculus of the ideas immanent in nervous activity*, **Bulletin of Mathematical Biophysics**, 5, 1943, 115-133.
2. Neto, J. Pedro, Siegelmann, H. T., and Costa, J. Félix, *On the Implementation of Programming Languages with Neural Nets*, **First International Conference on Computing Anticipatory Systems**, CHAOS, [1], 1997, 201-208.
3. 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.
4. Siegelmann, H., *Foundations of Recurrent Neural Networks*, Technical Report DCS-TR-306, Rutgers University, 1993.
5. Siegelmann, H., *On NIL: The Software Constructor of Neural Networks*, **Parallel Processing Letters**, [6] 4, World Scientific Publishing Company, 1996, 575-582.