A compiler and simulator for partial recursive functions over neural networks

João Pedro Neto
Dept. de Informática, Faculdade de Ciências, C5 – Piso 1, 1700 Lisboa, PORTUGAL
email: `jpn@di.fc.ul.pt`

José Félix Costa
Dept. de Matemática, Instituto Superior Técnico, Av. Rovisco Pais, 1049-001 Lisboa, PORTUGAL,
email: `fgc@math.ist.utl.pt`

Paulo Carreira
Oblog Consulting S.A. Rua da Barruncheira, 4, 2795-477 Carnaxide, PORTUGAL,
email: `paulo.carreira@oblog.pt`

Miguel Rosa
email: `mar@fccn.pt`

Abstract. *It was shown that Artificial Recurrent Neural Networks have the same computing power as Turing machines (cf. [6,8]). A Turing machine can be programmed in a proper high-level language - the language of partial recursive functions. In this paper we present the implementation of a compiler that directly translates high-level Turing machine programs to Artificial Recursive Neural Networks. The application contains a simulator that can be used to test the resulting networks. We also argue that these experiments provide clues to develop procedures for automatic synthesis of Neural Networks from high-level descriptions.*

Keywords*: Neural Networks, Partial Recursive Functions, Modular Networks.*

1. Introduction

The field of Artificial Recurrent Neural Networks (ARNNs), mainly in the last two decades, was able to solve engineering problems while keeping the simplicity of the underlying principles that allow them to mimic their biological counterparts. All this attracts people from many different fields such as Neurophysiology and Computer Science. We introduce our subject from a Computer Science perspective: the ARNN is seen as a computing mechanism able to perform computation based on a program coded as a specific arrangement of neurons and synapses. This work implements a compiler and a simulator based on [4]. In [3,7,5] similar ideas are presented but they are based on higher-level languages. We start by presenting the underlying theoretical context on which this work is based. In section 2 we give a brief review of the concept of partial recursive function. In section 3 we present our approach for building neural networks from partial recursive functions. The explanation of how we adapted the design of [4] into a compiler is given in section 4. Section 5 refers to the simulator and usage examples and section 6 concludes this paper. The simulator is freely available at *www.di.fc.ul.pt/~jpn/ netdef/nwb.html*.

2. Partial Recursive Functions

The language that we will use to express computation is the one of partial recursive functions (PRF). Although primitive when compared to modern computer languages, it is simple and powerful enough to program any mechanism with the same computing power as a Turing machine. Surely, building complex programs with this language would be very difficult, and more appropriate languages exist. For our purposes however, this language is well suited. The PRF theory identifies the set of computable functions with the set of partial recursive functions. We shall use $a(x_1,\ldots,x_n) \equiv b(x_1, \ldots, x_n)$ to denote that for all $x_1,\ldots,x_n$, $a(x_1,\ldots,x_n)$ and $b(x_1,\ldots,x_n)$ are both defined and coincide or are both undefined.

The axioms – also called primitive functions – are:

- **W** that denotes the *zero-ary constant* 0;
- **S** that denotes the *unary successor function* $S(x)=x+1$;
- **U(i,n)** that for i and n fixed, $1 \le i \le n$, denotes the *projection function* $U_{i,n}(x_1, ..., x_n) = x_i$.

The construction rules are:

- **C**, denoting *composition*. If $f_1, \ldots, f_k$ are n-ary PRFs, and g is a k-ary PRF, then the function h defined by composition, $h(x_1, \ldots, x_n) \equiv g(f_1(x_1, \ldots, x_n), \ldots, f_k(x_1, \ldots, x_n))$, is a PRF;
- **R,** denoting *recursion*. If f is a n-ary PRF and g is a (n+2)-ary PRF, then the unique (n+1)-ary function h, defined by

  1) $h(x_1,\ldots, x_n, 0) \equiv f(x_1,\ldots,x_n)$ and
  2) $h(x_1,\ldots, x_n, y+1) \equiv g(x_1,\ldots,x_n, y, h(x_1,\ldots, x_n, y))$;

  is a PRF;
- **M,** denoting *minimization*. If f is a (n+1)-ary PRF, then $h(x_1,\ldots, x_n) \equiv \mu_y(f(x_1,\ldots, x_n, y)=0)$ is also a PRF, where $\mu_y(f(x_1,\ldots, x_n, y)=0) =$

$$\begin{cases} \text{least } y \text{ such that } f(x_1,..,x_n,y)=0 \text{ and} \\ \qquad\qquad \forall z \le y\text{: } f(x_1,\ldots,x_n,z) \text{ is defined} \\ \text{undefined, if no such } y \text{ exists} \end{cases}$$

For instance, $f(x,y)=x+1$ is a PRF and is described by the expression `C(U(1,2),S)`. The function $f(x,y)=x+y$ is also a PRF described by the expression `R(U(1,1), C(U(3,3),S)`. In fact, it can be shown that every Turing computable function is a PRF. Details on PRF theory are found in [1,2].

3. Coding PRF into ARNNs

Finding a systematic way of generating ARNNs from given descriptions of PRF's greatly simplifies the task of producing neural nets to perform certain specific tasks. It also provides a proof that neural nets can effectively compute all Turing computable functions (cf. [4]). In this section we briefly describe the computing rule of each processing element, i.e., each neuron. Furthermore, we present the coding strategy of natural numbers that is used to load the network. Finally, we will see how to code a PRF into an ARNN.

3.1 How do the processing elements work?

We make use of σ-processors, just as in [4]. In each instant *t* each neuron *j* updates its activity x_j in the following non-linear way:

$$x_j(t+1) = \sigma\left(\sum_{i=1}^{N} a_{ji}x_i(t) + \sum_{k=1}^{M} b_{jk}u_k(t) + c_j \right) \qquad \sigma(x) = \begin{cases} 1 & , x\geq 1 \\ x & , 0<x<1 \\ 0 & , x\leq 0 \end{cases}$$

where a_{ji}, b_{jk}, and c_j are rational weights; N is the number of neurons, M the number of input streams u_k; and σ is the continuous function defined above.

3.2 How do we represent the numbers?

We use an unary representation where each natural number is represented as a rational number by means of a mapping α where, for each n, α(n) is given by

$$\sum_{i=0}^{n} 10^{-i-1}$$

3.3 How to build the ARNN?

The following three net schemata were implemented to compute the corresponding three axioms of recursive function theory. Changes were made with respect to [4]. First, the W axiom is provided with two additional neurons. Second, each U(i,n) axiom is constructed with only five neurons making it more efficient.

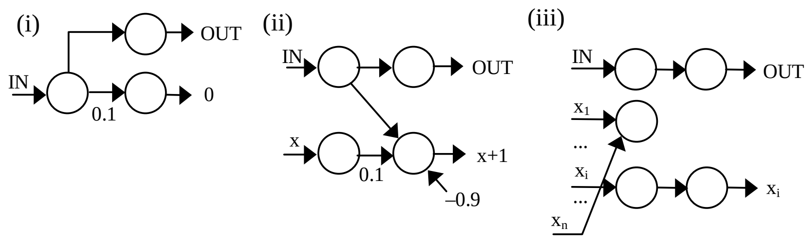

Figure 1. Axioms (i) W; (ii) S; (iii) U(i,n).

The net schemata of figures 3, 4 and 5 (see appendix) illustrate these rules, where gray colored circles represent repeated neurons. The `Sg` box is the subnet that finds whether a number is positive or zero.

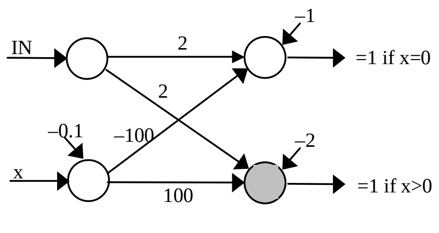

Figure 2. The signal (Sg) network.

For each PRF expression an ARNN is generated using the structure of the expression. The construction of the ARNNs is made in a top-down fashion, beginning with the outermost ARNN and then continuously instantiating ARNNs until reaching the axioms. For an expression `R(U(1,1),C(S,U(3,3))` we would first build the network for the recursion, then instantiate it with the projection axiom network and with the composition rule network, that in turn would accommodate the successor and projection axiom networks. This instantiation mechanism for the network schemata consists of replacing the boxes by compatible networks. A box is said to be compatible with a network schemata if both have the same number of inputs and outputs. The substitution operation of a box by network schemata consists of connecting the box inputs (outputs) to the network inputs (outputs).

4. Putting the pieces together

The tool we want to present to the user should be capable of handling more complex functions than the simple examples used in the previous sections. Specifying a complex function implies writing a complex expression. This motivates a modular and yet simple language to increase readability.

A PRF description is a sequence of statements where each statement is a pair that contains an identifier and an expression. Each identifier labels an expression. Each expression is written in the language described in section 2 and may refer to a previous statement. For example, when specifying the product PRF we can initially write the statement for the sum PRF and follow it by a shortened product PRF expression in which the sum expression is substituted by the initial `sum` identifier as shown below (so, expressions make use of those defined previously).

```
sum R(U(1,1), C(U(3,3), S))
product R(Z, C(U(3,3), U(1,3), sum))
```

5. Using the simulator

An application was built to provide the user not only with a compiler of PRF into ARNNs but also with a simulator that, among other things, allows step-by-step simulation and inspection of groups of neurons. In our application there are two ways of building ARNNs: by compiling the PRFs or by reading the dynamic equations system that define the neural network. The simulator takes the already built ARNN and configures it to read the input. Then it requests each neuron to process its input data. This process is iterated until a signal is seen in the OUT line of the outmost network. During the ARNN computation, the program allows the user to inspect the computations that are taking place inside the net. The user can also see the corresponding equation of each neuron and define a limit for the iteration number. Each neuron has a name given by the compiler with some special meaning. The name of each neuron expresses information about itself:

- *Func* is one of W, S, U(i,j), R, C, M.
- *Type* is one or more of the following: *In*, stands for the IN line supporting neurons; *Out*, stands for the OUT line supporting neurons; *Data*, stands for neurons that receive parameters from external networks or that send result of

computation to other networks; and *Res* stands for neurons that handle results of functions; *Hid*, stands for neurons not in a border layer.

- *Num* is the number of the neuron inside its network schemata.
- *Depth* is the same as the depth in the corresponding PRF parsing tree.
- *Id* is used to ensure an unique identifier for each neuron.

Some examples of function descriptions are given in the table below:

Table 1. Some PRF descriptions

Function	Expression
f=0	`W`
f(x)=x+1	`S`
f(x,y,z)=y	`U(2,3)`
f(x)=0	`R(W, U(2,2))` → *also called Z*
f(x,y,z)=z+1	`C(U(3,3),S)`
f(x,y)=x+y	`R(U(1,1),z+1)`
f(x,y)=x*y	`R(Z,C(U(3,3),U(1,3),x+y))`
f(x)=sg(x)	`R(C(W,S),C(W,U(1,2)))`
f(x,y)=x-1	`R(W,U(1,2))`
f(x,y)=x-y	`R(U(1,1),C(U(3,3),x-1))`
f(x,y)=|x-y|	`C(x-y,C(U(2,2),U(1,2),x-y),x+y)`

As an example, let us consider the following description of the *sum* function:

```
proj/1    U(1,1)
proj/2    U(3,3)
comp      C(proj/2,S)
sum       R(proj/1,comp)
```

Notice that this function is not limited by any integer value (there is no MAXINT), since the rational precision is not limited (it is, however, finite). After the compilation of these four statements we have an ARNN with 39 neurons and 70 synapses. Below we present the first lines of the dynamic equation system:

```
XRin_1_0(i+1)    = σ(Ein(i))
XRhid2_1_1(i+1)  = σ(XRin_1_0(i))
XRhid3_1_2(i+1)  = σ(0.1*XRin_1_0(i) + XRhid3_1_2(i) +
   XRhid4_1_3(i) - XRhid5_1_4(i) - XREout_1_15(i))
XRhid4_1_3(i+1)  = σ(0.1*XRhid3_1_2(i) + XRhid5_1_4(i) - 0.9)
XRhid5_1_4(i+1)  = σ(XRhid6_1_5(i))
XRhid6_1_5(i+1)  = σ(XSout_4_7(i))
XRhid7_1_6(i+1)  = σ(XRhid3_1_2(i) + XRhid15_1_14(i) - 1)
  ...
```

The simulator is freely available at *www.di.fc.ul.pt/ ~jpn/netdef/nwb.html*.

6. Conclusion

We hope to have taken one step further in understanding the relationship between programming languages and ARNNs. We started presenting a theoretical framework of a very simple programming language. Next, we described a systematic way of obtaining ARNNs from programs written in the PRF language and concluded with the presentation of a compiling tool. Although the ideas here presented may seem oversimplified when facing real-world applications, the problem itself (can some type of ARNNs perform exact symbolic computation?) has an affirmative answer. Moreover, there is an automatic translation from the mathematical description of any Turing computable function into a Neural Network able to compute that function. This establishes a computational equivalence of this simple model of ARNNs and Partial Recursive Functions (itself an Universal model of computation), which is modular-oriented (making complexity management of large programs easier to deal with). Adapting them to other more elaborate frameworks like those of high-level languages is straightforward, as in [5]. Finally, we also defend that the search for a more suited language to the architecture of ARNNs can give us fresh views on efficient approaches for systematic construction of ARNNs.

Acknowledgements

We would like to thank the technical and financial support given by LabMAg and the Portuguese R&D Agency Fundação de Ciência e Tecnologia. Also, special thanks to Hava Siegelmann for all the scientific collaboration within this project.

References

[1] Boolos, G. and Jeffrey, R., *Computability and Logic* (second edition), Cambridge University Press, 1980.

[2] Cutland, N., *Computability – An Introduction to Recursive Function Theory*, Cambridge University Press, 1980.

[3] Gruau, F., Ratajszcza J., and Wiber, G., *Fundamental study – A neural compiler*, Theoretical Computer Science, Elsevier, 141, 1995, 1-52.

[4] Neto, J., Siegelmann, H., Costa, J., and Araújo, C., *Turing universality of neural nets (revisited*), Proceedings of Computer Aided Systems Theory – EUROCAST'97, 1997.

[5] Neto, J., Siegelmann, H., and Costa, J., *Symbolic processing in neural networks*, to be published at the Journal of Brasilian Computer Society, 2001.

[6] Siegelmann, H. and Sontag, E., *On the computational power of neural nets*, J. of Computer and System Sciences, Academic Press, [50] 1, 1995, 132-150.

[7] Siegelmann, H., *On NIL: the software constructor of neural networks*, Parallel Processing Letters, [6] 4, World Scientific Publ. Company, 1996, 575-582.

[8] Siegelmann, H., *Neural Networks and Analog Computation*, *Beyond the Turing Limit*, Birkhäuser, 1999.

Appendix

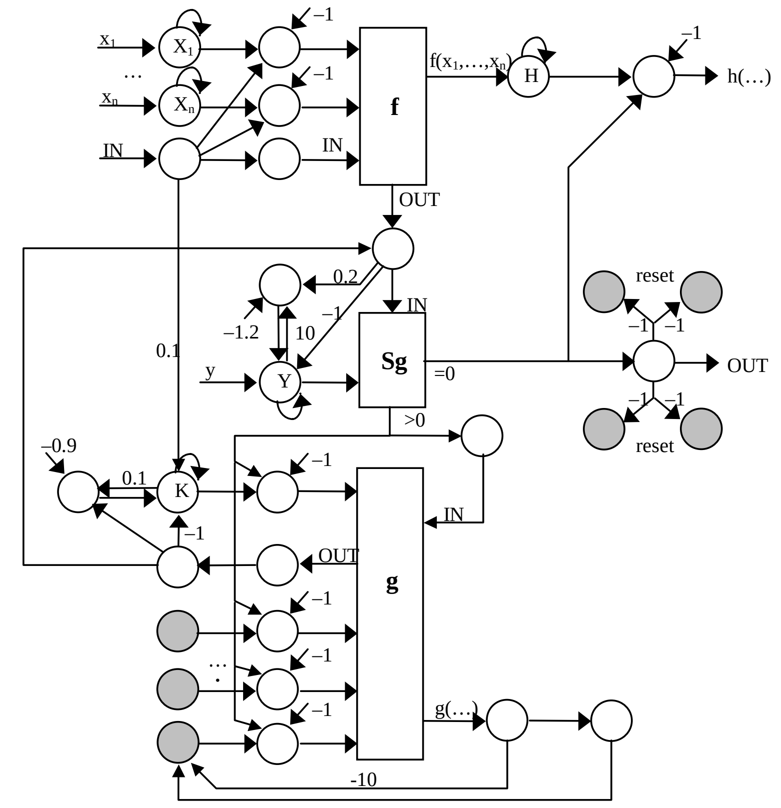

Figure 3. Recursion.

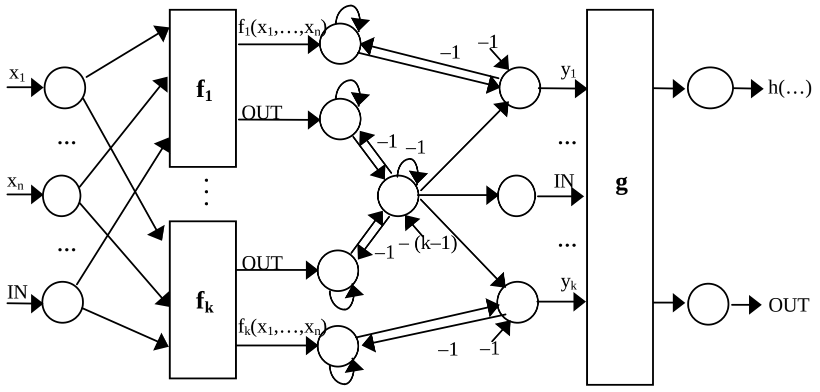

Figure 4. Composition.

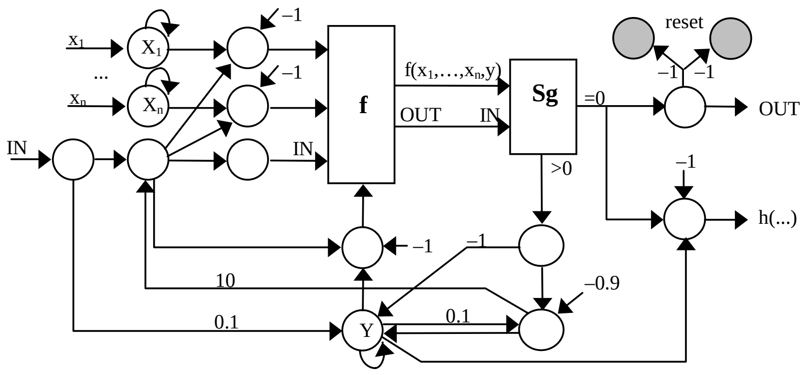

Figure 5. Minimization.