February, 1997

Turing Universality of Neural Nets (revisited)

J. Pedro Neto
Hava Siegelmann, *University of Massachusetts - Amherst*
J. Félix Costa
C. P. Suárez Araujo

Turing Universality of Neural Nets (revisited).
Lecture Notes in Computer Science – 1333, 361-366, Springer-Verlag, 1997.

Turing Universality of Neural Nets (Revisited)[*]

J. Pedro Neto[1], Hava T. Siegelmann[2], J. Félix Costa[1], and C.P.Suárez Araujo[3]
jpn@di.fc.ul.pt, iehava@ie.technion.ac.il, fgc@di.fc.ul.pt, and paz@neurona.dis.ulpgc.es

[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

[3]Dpt. of Computer Sciences and Systems. Univ. of Las Palmas de G.C.
CAMPUS UNIVERSITARIO DE TAFIRA, 35017 LAS PALMAS DE G.C., SPAIN

Abstract. We show how to use recursive function theory to prove Turing universality of finite analog recurrent neural nets, with a piecewise linear sigmoid function as activation function. We emphasize the modular construction of nets within nets, a relevant issue from the software engineering point of view.
Keywords. Neural computation, recursive function theory, modularity.

1 Introduction

In this paper we work with analog recurrent neural nets (ARNN's) as in [Siegelmann and Sontag 95]. In each instant t each neuron i updates its activity x_i in the following non-linear way:

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

where a_{ij}, b_{ij} and c_i are rational weights (and therefore Turing computable); N is the number of neurons, M the number of input streams u_j; and σ is the piecewise linear sigmoid function,

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

which is a continuous function as opposed to the Heaviside function. The latter, when used in the context of analog neural nets allows the instantaneous computation of equality between reals, which is rather unphysical. Using σ-processors we can directly import Siegelmann and Sontag constructs for stacks, together with coding and uncoding devices.

As showed in [Siegelmann and Sontag 91], using classical constraints (finite binary input/output and a finite number of processors), these nets have Turing power when we allow for rational weights. Herein, we remake this proof by means of recursive

[*] This work was supported by JNICT PBIC/TIT/2527/95 and a fellowship from the Gobierno Autonomo de Canarias.

function theory. The idea is to emphasize modular constructions of nets within nets. This approach will provide insights of modularity that we are expecting to use latter in a *local learning* theory for hybrid systems.

2 Recursive Function Theory

Recursive function theory identifies the set of *computable functions* with the set of *partial recursive functions* on $\mathbb{N}$ (see [Boolos 80]). A function f is said to be computable if it can be manufactured from a specific set of basic functions and some construction rules.

The primitive functions, also called *axioms*, are:

- **W**, the zero-ary constant 0;
- **S**, the unary successor function $S(x) = x+1$;
- The set of n-ary projection functions, $\mathbf{U_{i,n}}(x_1,\dots,x_n) = x_i$ $(1\leq i\leq n)$.

The construction rules are:

- **Composition (C)**: If $g(y_1,\dots,y_k)$ and $f_1(x_1,\dots,x_n)$, …, $f_k(x_1,\dots,x_n)$ are computable functions, then $h(x_1,\dots,x_n) = g(f_1(x_1,\dots,x_n),\dots,f_k(x_1,\dots,x_n))$ is a computable function.

- **Recursion (R)**: If $f(x_1,\dots,x_n)$ and $g(x_1,\dots,x_n,y,z)$ are computable functions, then the unique function $h(x_1,\dots,x_n,y)$, defined by $h(x_1,\dots,x_n,0) = f(x_1,\dots,x_n)$ and $h(x_1,\dots,x_n,y+1) = g(x_1,\dots,x_n,y,h(x_1,\dots,x_n,y))$ is a computable function.

For the last rule, we introduce the μ functional: for any function $f(x_1,\dots,x_n,y)$,

$$\mu_y(f(x_1,\dots,x_n,y)=0) = \begin{cases} \text{the least y such that} \\ \quad f(x_1,\dots x_n,z) \text{ is defined for all } z\leq y \\ \quad f(x_1,\dots x_n,y)=0 \\ \text{undefined, if there is no such y} \end{cases}$$

- **Minimalisation (M)**: If $f(x_1,\dots,x_n,y)$ is a computable function, then $h(x_1,\dots,x_n) = \mu_y(f(x_1,\dots,x_n,y)=0)$ is a computable function.

For instance, the function $h(x,y)=x+2$ is computable and given by $C(C(U_{1,2},S),S)$. Also, $h(x,y)=x+y$ is a computable function given by $R(U_{1,1},C(S,U_{3,3}))$. It can be shown that all Turing computable functions are partial recursive (see [Boolos 80]).

3 Number Representation

Each natural number, $i\in\mathbb{N}$, is coded as a rational number, $x_i\in\,]0,1[$. This is mandatory because we want neurons to hold values. We adopt the unary coding:

$0 \equiv 0.1,\ 1 \equiv 0.11,\ 2 \equiv 0.111,\ \dots,\ n \equiv 0.1^{n+1}$

Every initial natural input must be coded before the computation starts and decoded after the computation to provide the final output. The coding and uncoding techniques used in [Siegelmann and Sontag 91] will do the job.

4 Net Examples

In the following neural net diagrams, non-labelled arcs default to weight one. The first net finds if a given number is positive or zero, outputting a 1 through the appropriate channel.

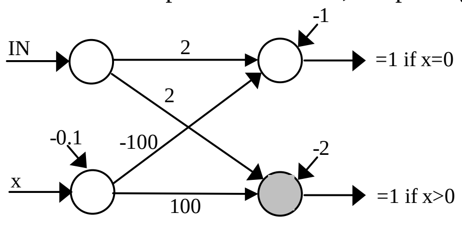

Fig. 1. The Signal (Sg) Net.

For instance, if we take a positive number coded as a rational number greater than 0.1, subtract 0.1 and multiply by 100, the result is greater than 1 and the shadowed neuron will output 1, else it will output 0. We use a box labelled Sg as a macro for this net.

The following net receives a positive integer and returns its successor (we will use a box labelled Succ as a macro for this net)

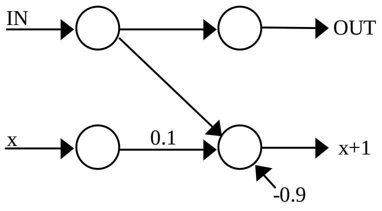

Fig. 2. The Successor (Succ) Net.

These nets begin the specified computation if and only if they receive a 1 through the input IN. The output OUT signals to the following module the availability of the result at that precise moment. Using this method, we can easily control all synchronizations. The next net synchronizes two different incoming signals,

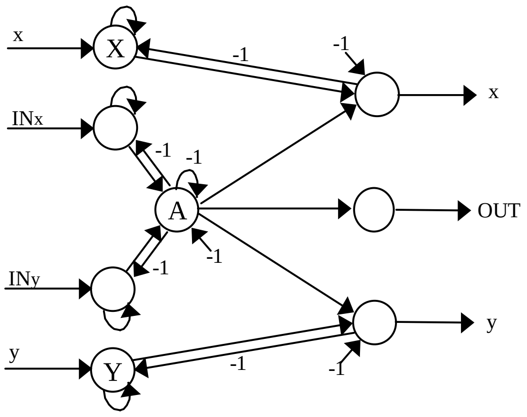

Fig. 3. The Sync-2 Net.

The Sync-2 net can be easily transformed to synchronize n inputs. The neuron *A* must have its bias changed to -(n-1). *A* is activated only when both signals arrive. Either *X* or *Y* will keep the first value until the second arrives.

5 Net Schemas

5.1 The axioms

The following three net schemas compute the three axioms of recursive function theory: the zero-ary constant 0, the (unary) successor and the set of projection functions.

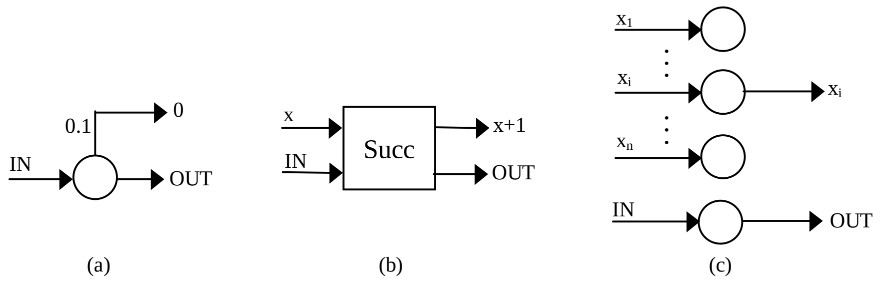

Fig. 4. The three axioms, (a) W, (b) S, (c) $U_{i,n}$.

The three rules, *composition*, *recursion* and *minimalisation* have their particular net schemas.

5.2 Composition

For the composition schema, $h(x_1,\ldots,x_n) = g(f_1(x_1,\ldots,x_n),\ldots,f_k(x_1,\ldots,x_n))$, each $f_i(x_1,\ldots,x_n)$, i=1,…,k, is computed first and partial results are trapped by a Sync-k net of k inputs until all of them are available. Then, they are all inputed into the g net.

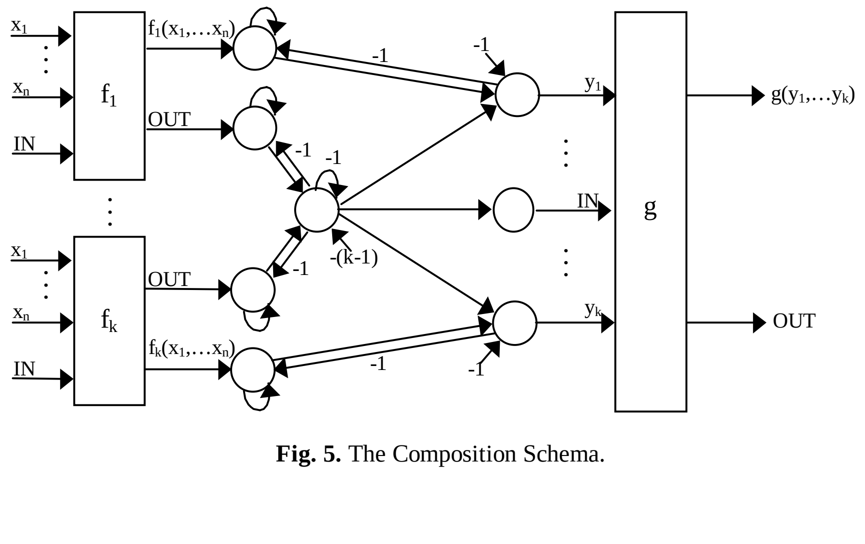

Fig. 5. The Composition Schema.

5.3 Recursion

To compute $h(x_1,\dots,x_n,0) = f(x_1,\dots,x_n)$, $h(x_1,\dots,x_n,y+1) = g(x_1,\dots,x_n,y,h(x_1,\dots,x_n,y))$, we introduce an algorithm that inspires our proposal for the recursion schema. It iterates from 0 until y, computing all partial results.

```
K←0;
H←f(x₁,…,xₙ);
while y>0 do
 begin
  H←g(x₁,…,xₙ,K,H);
  K←K+1;
  y←y-1;
 end;
h(x₁,…,xₙ,y)←H;
```

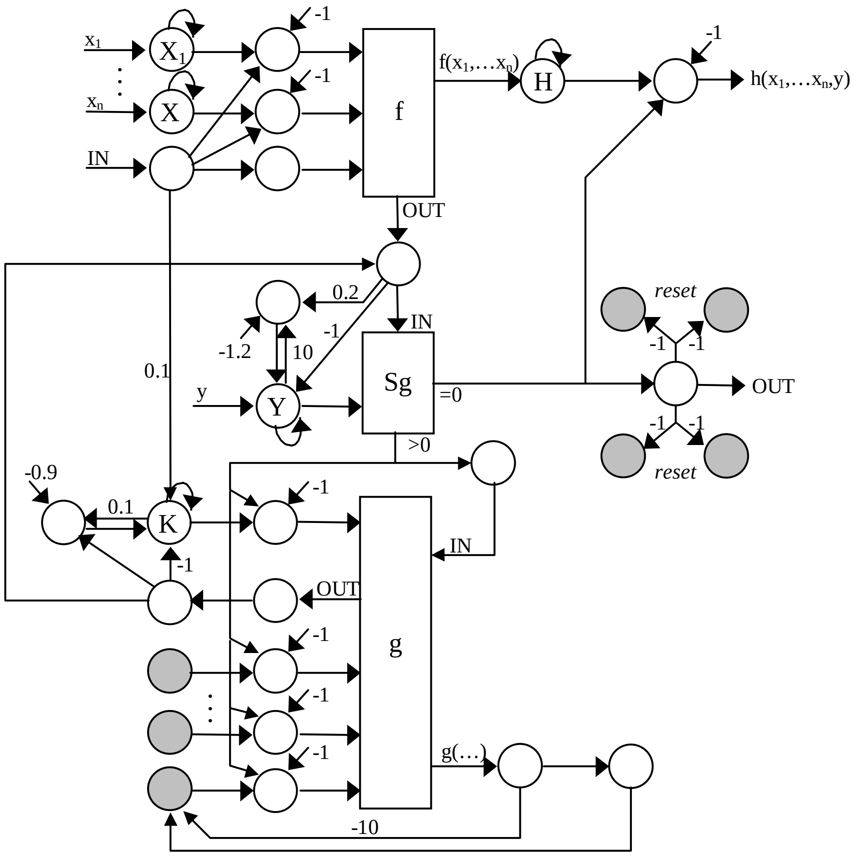

Fig. 6. The Recursion Schema.

5.4 Minimalisation

For the minimalisation, we must find the least y such that $f(x_1,\ldots,x_n,y)=0$. Both algorithm and the computation of the resulting net schema will diverge if no such y exists. The f box denotes the net that computes function f.

```
$Y \leftarrow 0$;
while $f(x_1,\ldots,x_n,Y) \neq 0$ do
  $Y \leftarrow Y+1$;
$h(x_1,\ldots,x_n) \leftarrow Y$;
```

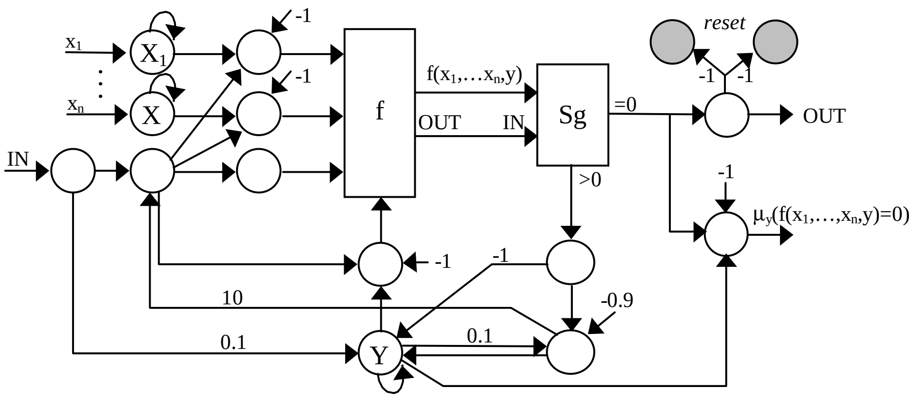

Fig. 7. The Minimalisation Schema.

6 An Axiomatic Proof...

We will provide one example of an axiomatic proof. A proof is a list of interconnected steps, each one prescribing some function, and explaining how we can construct it from earlier steps. For instance, consider binary addition, $h(x,y) = x+y$.

$$h(x,0) = x \ (= f(x)) \text{ and,}$$

$$h(x,y+1) = x+(y+1) = (x+y)+1 = h(x,y)+1 \ (= g(x,y,h(x,y)))$$

If we can prove that both f and g are computable (that is, if both f and g can be built using the axioms and the construction rules), then we can use recursion to find h. Fig. 8 displays a possible proof:

	Step	Function	Reason
$f \rightarrow$	1	$\exists f_1$: $f_1(x) = x$	$U_{1,1}$
	2	$\exists f_2$: $f_2(x) = x+1$	S
	3	$\exists f_3$: $f_3(x,y,z) = z$	$U_{3,3}$
$g \rightarrow$	4	$\exists f_4$: $f_4(x,y,z) = z+1$	Composition of 3 in 2
	5	$\exists f_5$: $f_5(x,y) = x+y$	Recursion with 1 and 4

Fig. 8. The proof of $h(x,y) = x+y$, $R(U_{1,1},C(S,U_{3,3}))$

Each step denotes a computable function. Then we are able to compile it into the corresponding net (using the algorithm introduced so far). The resulting net becomes an independent module. Every module can be inserted where it is needed. In this example, modules 1, 2 and 3 are straightforward. Module 4 uses module 1 for function f(…) and module 3 as g(…) (see fig. 5). Module 5 uses module 1 as f(…) and module 4 as g(…) (see fig. 6). When this procedure ends, we have a module for binary addition. It can be used to build more complex functions.

In this way, a library of functions, or *theorems*, can be set. All working independently from each other and having no synchronization problems between them. The net programming task becomes modular.

7 Conclusion

With the compilation of an universal recursive function we find an "universal" analog recurrent neural net. Our proof not only confirms that neural nets can compute all Turing computable functions, but also gives an explicit method to build those same nets. If we have the axiomatic proof of a function, there is an algorithmic way to built the respective net. This net is build in a modular way, solving at the same time, all synchronization problems, and minimising the associate complexity of assembling elaborate functions.

8 References

[Boolos 80]
BOOLOS, G. and JEFFREY, R., *Computability and Logic*, (2º Ed), Cambridge University Press, 1980.

[Siegelmann and Sontag 91]
H.SIEGELMANN and E.SONTAG, "Neural Nets are Universal Computing Devices". SYCON Report 91-08, Rutgers University, 1991.

[Siegelmann and Sontag 95]
H.SIEGELMANN and E.SONTAG, "On the Computational Power of Neural Nets", in *Journal of Computer and System Science* [50]1, Academic Press, 1995.