Lower Bounds of Computational Power of a Synaptic Calculus[*]

João Pedro Neto, J. Félix Costa and Helder Coelho
{jpn, fgc, hcoelho}@di.fc.ul.pt

Dept. Informática, Faculdade de Ciências da Universidade de Lisboa, Bloco C5 - Piso 1,
Campo Grande 1700 Lisboa - Portugal. Tel. 7573141. Fax. 7570084

Abstract

The majority of neural net models presented in the literature focus mainly in the neural structure of nets, leaving aside many details about synapses and dendrites. This can be very reductionist if we want to approach our model to real neural nets. These structures tend to be very elaborate, and are able to process information in very complex ways (see [Mel 94] for details).

We will introduce a new model, the S-Net (Synaptic-Net), in order to represent neural nets with special emphasis on synaptic and dendritic transmission. First, we present the supporting mathematical structure of S-Nets, initially inspired on Petri-Net formalism, adding a transition to transition connection type. There are two main components of S-Nets, neurones and synaptic/dendritic units (s/d units). All activation values are integers. Neurones are similar to McCulloch-Pitts neurones, and s/d units will process information within certain class of functions.

S-Nets are able to represent spatial nets representations in a very natural way. We can easily modulate the length of an axon, the connection or branching of two dendrites or synaptic connections. Some examples are shown.

Next, the focus will be on what kind of functions are suited to s/d units. We will present three function types: sum, maximization and simple negation (changing an excitatory impulse to an inhibitory one, or vice-versa). With these functions for s/d units and with simple neurones, we will prove that all recursive functions can be computed by at least one specific S-Net. In order to achieve this, we will use the Register Machine, and show a way to build for each symbolic program, a S-Net capable of computing the function defined for that specific program. This computational power will be achieved without any use of synaptic weights (i.e., all weights are one as in McCulloch's model) or neurones activation values (i.e., all values are set to zero).

Finally, some aspects for future investigation are presented, namely, the possibility of synaptic-synaptic connections, how can noise be handled, and some other features intended to approach this mathematical model to our reality.

Keywords: Neural Networks, Synapses, Dentritic Trees, Neural Computation, Theory of Computation, Spatial Representation.

Introduction

Many neural network models proposed in the literature have their major focus in neurones as distributed processors with machine learning purposes. Although they were inspired initially in the way brain processes information (see [McCulloch 43]), almost all systems leave aside the complex structure involving neurones. Namely, dentritic trees that process spatial integration of signals sent from presynaptic cells; the terminal arborization at the end of each axon; the subtle structure of synaptic transmissions, that can make direct contact with dendrites, with axons, with the soma, or even with other synapses (a good overview can be found in [Shepherd 94]).

Usually, the ultimate reference are synaptic weights between neurones. If it is true that some of these models had succeeded on classification and learning tasks, it is also a fact that they put themselves far away in modelling real neural networks. Our main concern is not to search for a new learning algorithm, but to find how much computability power can retain a specific denditric architecture, that can be used for explanation purposes.

Several neurobiological studies had consistently shown how complex and intricate are dentritic trees and synaptic connections. These structures may possess a great

[*] This work was supported by JNICT PBIC/TIT/2527/95

computational symbolic potential. This capability is not only an effect of how the single units process information, but it is also a consequence of its spatial configuration, of its different external influences or even of its capacity for multiple pseudoindependent processing subunits inside the same dentritic tree (see [Mel 94]).

Synaptic-Nets

Our aim with this paper is to introduce a mathematical system able to model some of those complex connection structures, in a direct and natural way. The present result, still in development, is coined *Synaptic-Nets* (S-Nets).

The earliest idea came from the common way we see an Artificial Recorrent Neural Net (ARNN): as a direct graph. Usually, an ARNN uses graph nodes to represent neurones, and directed arcs to represent connections between neurones, also called synapses.

Here arises the first difference. In this new structure, we are able to perform calculations in neurones as well as in dentritic and synaptic units. So, one type of node is no longer sufficient. The first question is, how many different nodes should we allow? We think that two types are enough, one for neurones, and one for all others. It is not relevant to introduce more node types. What seems important is the connection point. They can be dendrite to dendrite connections, axon to dendrite, axon to axon, and so on… (see [Sheperd 94] for a good overview of different types of junctions between nerve cells). What is common to all, is the linking synapses. We also decided to include the concept of dendritic convergence and divergence.

With two types of nodes, the next question is whether S-Nets are formally a kind of Petri-Nets? Despite some similarities, there is a fundamental difference between both. In Petri-Nets, we must have an alternate sequence event to transition to event, but in S-Nets, the existence of arbitrary complex synapse to synapse connections is necessary. The binary relation of Petri-Nets must be generalized.

Definition: A Synaptic-Net (**S-Net**) is a tuple $< G, S, R, F >$, where G is a finite non-empty set whose elements are from now on called *neurones*; S is a finite set whose elements are from now on called *synapses*; R is a binary relation, $R \subseteq (G\times S)\cup(S\times G)\cup(S\times S)$; and F is a function, $F:(G\cup S) \times \omega \rightarrow Z$.

Remark: ω is the set of natural numbers. R determines the corresponding S-Net structure; F associates to each neurone or synapse an *activation value* in Z, for each time $t\in\omega$. To compute this value, each element $x\in G\cup S$ has a specific *information processing function*, $\phi_x:Z^n\rightarrow Z$, being n the number of inputs of x, i.e., $n = \#\{y: (y,x)\in R\}$.

In graphical terms, we will use circles to identify neurones, and squares to identify synaptic connections and dendritic trees (s/d units). These graphs are admissable S-Nets:

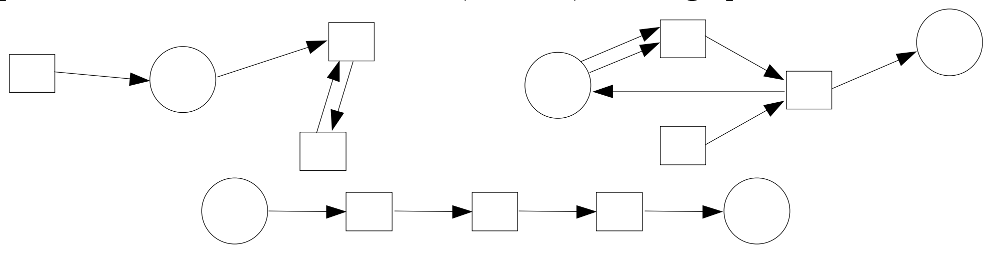

Figure 1 - S-Nets examples.

Here in, we work with dynamic maps F of the following form:

For each $x \in G \cup S$,

$$F(x,0) = 0,$$

$$F(x,t+1) = \phi_x(F(x_1,t), \ldots, F(x_n,t)) \quad \forall_{i=1..n}: (x_i,x) \in R$$

where ϕ_x is a specific function for each element of $x \in G \cup S$

All activation values are set to zero at time t=0, and the activation value at time t+1 is computed by the values of its predecessors at time t.

We will focus on a few set of simple functions, in order to get as close as possible of what might happen in real neural nets. One of the main concerns is to obtain functions able to support n inputs, and to consider all inputs in the same way (what could or could not be correct…). If there is more than one output, the signal is the same to all.

For neurones, there is only one function. Each neuron adds its inputs, both excitatory and inhibitory (it does not exist absolute inhibition), and checks if the sum is greater or equal to zero. If it is, the neuron becomes active and sends an excitatory signal through its outputs (value 1 for each output). If it is not, the neuron is inactive (value 0).

For all $g \in G$,

$$\phi_g(\vec{y}) = \phi_H(\vec{y}) = \begin{cases} 1, & \sum_{i=1}^{n} y_i \geq 0 \\ 0, & \sum_{i=1}^{n} y_i < 0 \end{cases}$$

All internal activities are set to zero, so all neurones have an identical internal structure. There are 3 functions for s/d units: sum, maximum and negation. If $s \in S$ then $\phi_s \in \{\phi_\Sigma, \phi_M, \phi_\neg\}$. For each $s \in S$, we have one of the following functions,

$$\phi_s(\vec{y}) = \phi_\Sigma(\vec{y}) = \sum_{i=1}^{n} y_i$$

$$\phi_s(\vec{y}) = \phi_M(\vec{y}) = \begin{cases} \max(y_1,\ldots,y_n), & \forall_{y_i \in G \cup S} : y_i \geq 0 \\ 0, & \exists_{y_i \in G \cup S} : y_i < 0 \end{cases}$$

$$\phi_s(\vec{y}) = \phi_\neg(\vec{y}) = -\phi_\Sigma(\vec{y})$$

Therefore, $\phi_x \in \{\phi_H, \phi_\Sigma, \phi_M, \phi_\neg\}$, for all $x \in G \cup S$.

Spatial Representation of Synapses and Dendritic Trees

One of the main advantages of S-Nets is its capacity to apprehend spatial structures of real neural networks. Is this relevant? Yes, because some neural proprieties depend on the neuron's physical structure. For instance, action potentials travel between neurones with some finite velocity, after all, they are just flows of electrical current along nerves. So, the length of those nerves is essential. We can model axons with different sizes very directly, adding more s/d units in proportion to their sizes (see fig. 2a). This also has an effect on transmission time, as required (long axons take more time to transmit the action potential to their postsynaptic cells).

Hence, these different lengths determine *when* the action potential arrives at the next neurones, which is fundamental to determine if a neuron is activated or not. Moreover, each neuron has an absolute refractory period (the period, after the neuron activation, during which it is impossible to stimulate the nerve a second time) and a relative refractory period (the period during which it can be stimulated, but only by using a larger current than usual), both functions of time.

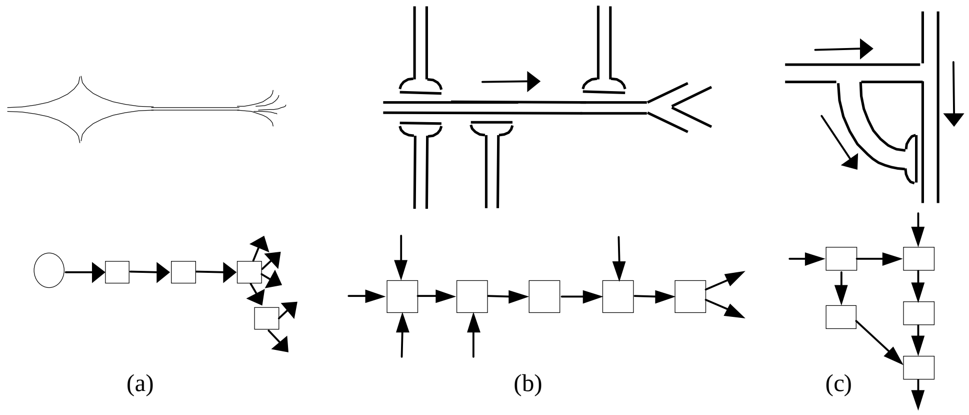

Figure 2- Modelling examples.

There are important spatiotemporal interactions inside neural nets as we will see in the following examples. The geometry of dendritic trees or the complex organization of synaptic contacts are more than just a useful way to connect neurones. They are special mechanisms that delay, attenuate and synchronize action potentials. S-Nets can represent this spatial complexity in a very direct way (see fig. 2b and 2c).

A well-known example, first introduced by Wilfred Rall in 1964, and very common in the literature (see, e.g., [Mel 94, Arbib 94, Anderson 95]), confirms that a passive dendritic branch, because of its spatial extension, can act as a filter that selects specific sequences of synaptic inputs. In Fig. 3, if input I_1 is activated at time t, I_2 at time t+1, and I_3 at time t+2, the neuron is activated. If the activation sequence is $I_3 \rightarrow I_2 \rightarrow I_1$, the neuron is not activated.

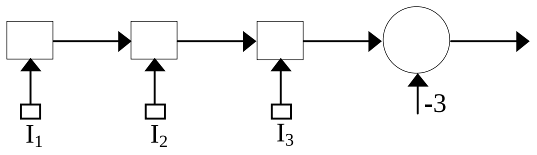

Figure 3- Input sequence selection.

Another example is taken from [Arbib 89], about *lateral inhibition*. This happens when a network structure is made so that neurones inhibit all but their close neighbours. Inhibition can be recurrent or nonrecurrent. In *nonrecurrent inhibition*, the inhibitory signal is a combination of the current excitations, $O_1 = I_1 - k.I_2$ and $O_2 = I_2 - k.I_1$, in which the local excitation is reduced by *k* times the neighbours excitation.

In *recurrent inhibition*, the signal sent to the neighbours is itself subject to their own inhibitory effect. In fig. 4b, $O_1(t+2) = I_1 - k.O_2(t)$ and $O_2(t+2) = I_2 - k.O_1(t)$. This type of inhibition is observed in the lateral eye of *Limulus*, the horseshoe crab. These two equations can be seen as a dynamic system, and the final outputs O_1 and O_2 will held an eventual equilibrium pair for this system.

The events of nonrecurrent inhibition are strictly localized, since the outputs depend exclusively on the inputs. In recurrent inhibition, the effects can spread to distant neighbours, since the actual changes effect their immediate neighbours, and these changes produce further changes in the next neighbours, and so on. Recurrent inhibition has an important feature, *disinhibition*, the inhibitions can themselves be inhibited.

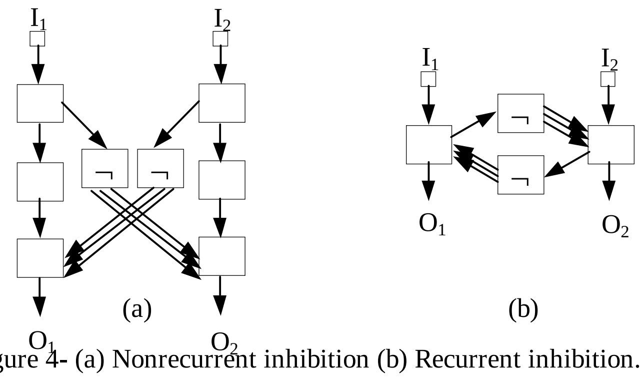

Figure 4- (a) Nonrecurrent inhibition (b) Recurrent inhibition.

Lower Bounds of Computational Power of S-Nets

Hava Siegelmann and Eduardo Sontag, [Siegelmann 95], proved that it is possible to simulate all Turing Machines by finite size neural networks. As in their paper, we will show that S-Nets are universal, using another mathematical formalism (but equivalent to Turing Machines), the Universal Register Machine (URM).

Having $\{\phi_H, \phi_\Sigma, \phi_M, \phi_\neg\}$, we state the following.

Proposition: All partial recursive functions $f:\omega^n \to \omega^m$ can be computed by at least one specific S-Net, using only $\phi \in \{\phi_H, \phi_\Sigma, \phi_M, \phi_\neg\}$.

Proof:

We will show that each URM instruction (see appendix) can be simulated by a specific S-Net. First, we see how to make a register with only 4 s/d units. So, for each program P, we need 4*ρ(P) s/d units to simulate all necessary memory (ρ(P) is the greatest register index used in P).

Graphically, each s/d function is represented by one of the following diagrams:

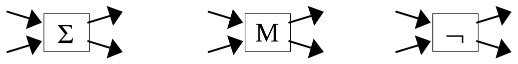

Figure 5- s/d units diagrams.

We can increment, reset and access a register value, that is done by the following sub-net,

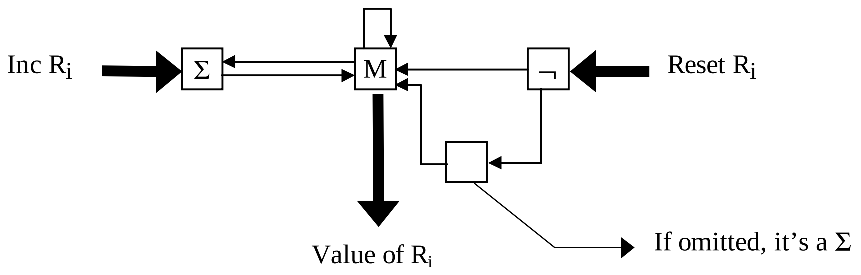

Figure 6- Memory Register.

In the following diagrams (where we build each URM instruction), the signals mean:

I_{n-1}	- activate this instruction;	I_0	- activate first instruction;
I_{n+1}	- activate next instruction;	I_f	- end of the program.

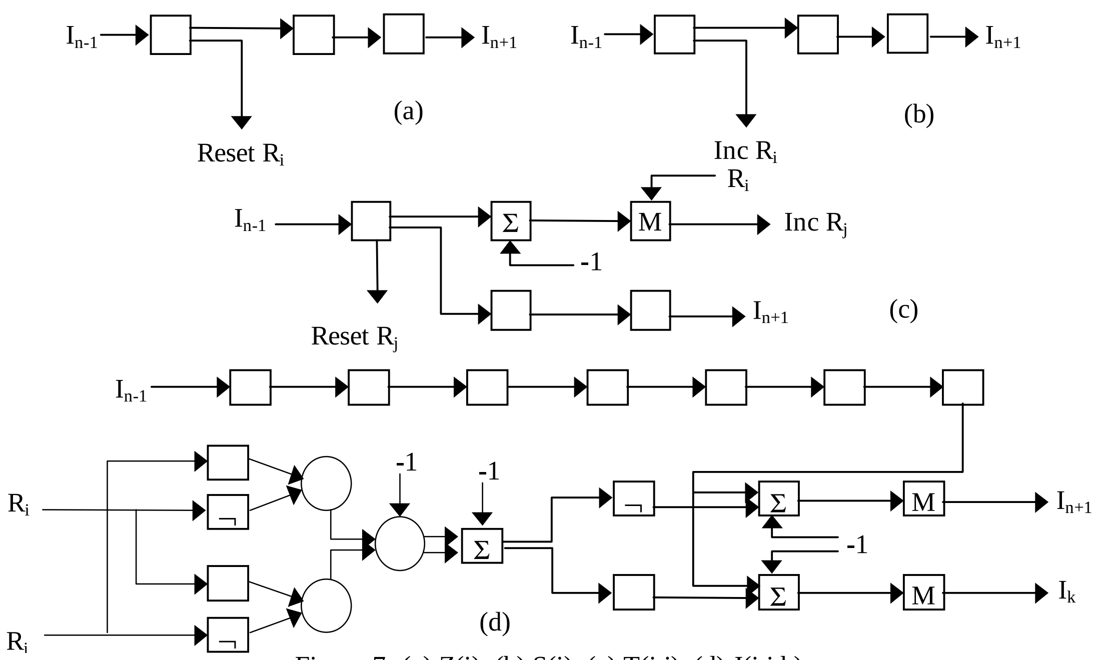

Figure 7- (a) Z(i), (b) S(i), (c) T(i,j), (d) J(i,j,k).

At this point, we already know how to build all instruction types and also to make memory registers. The next two diagrams show how to receive inputs, and how to return outputs. There are two special activation channels indicating when the input and the output are available. They are activated when they have value one.

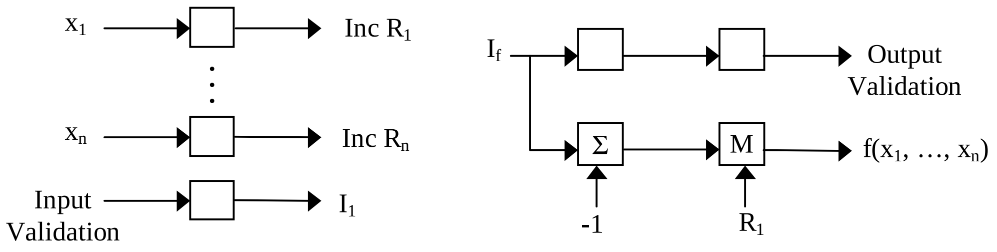

Figure 8 - Input and Output.

Finally, to create constant -1, we use the following sub-net,

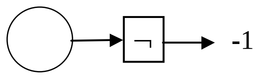

Figure 9 - Constant definition.

Using this method we see that each computable function $f:\omega^n \rightarrow \omega^m$, is also computed by a S-Net with 3.J+1 neurones and 1+(n+1)+4.R+3.(Z+S)+18.J+5.T+(2.m+2) s/d units, where R is the number of registers used in P, and, Z, S, J and T, the number of Z, S, J and T instructions of P. We have a linear complexity in time and in space, with respect to the Register Machine. ■

How this is done? Find P, and then for each sequential instruction of P, use the respective sub-net, and link them in the same sequential order.

To present an example, we will enclose each instruction in a box, labelling each one with its specific instruction (e.g., fig. 10). We only explicit the inputs and outputs. Also, we do not represent any register sub-net, only some of their reset, increment and value connections.

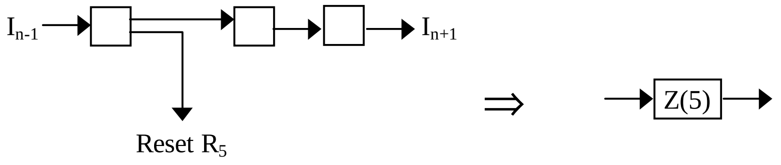

Figure 10 - Input and Output.

Let's compute binary multiplication, x*y. A possible URM program is, P = <J(1,4,9), J(2,5,6), S(5), S(3), J(1,1,2), Z(5), S(4), J(1,1,1), T(3,1)>. We will need ρ(P)=5 memory registers. The S-Net will be,

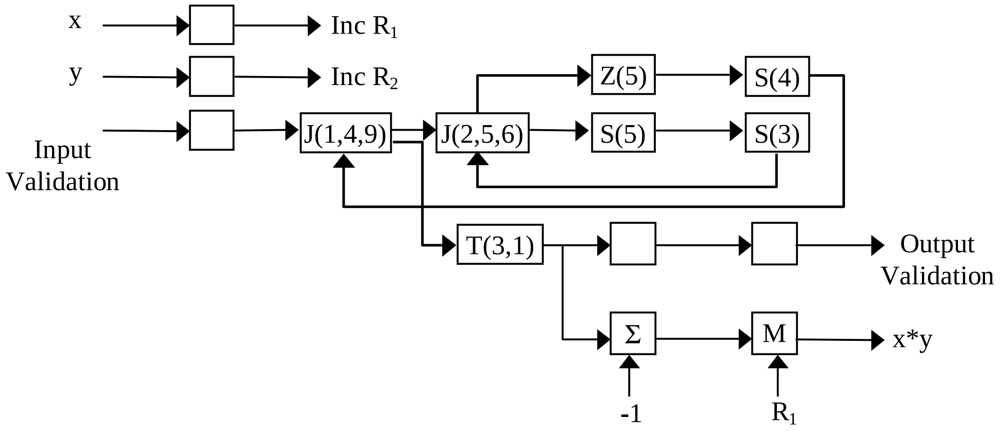

Figure 11 - Computing the product.

The URM program has two inconditional jumps, J(1,1,2) and J(1,1,1). For these, we do not use a Jump sub-net, but only a straight arrow to the next instruction.

Obviously, if there is a countable infinity of URM programs that compute each partial recursive function (and in fact, there is), it is also possible to construct a countable infinity of S-Nets capable of handle those same computations.

A S-Net is intrinsically a massive parallel machine. However, the method used to compute a function is based on the URM, which is a sequential machine. It is important to understand that our main goal in this section is to check the lower computational bounds of S-Nets, not to seek the fastest or simplest way to compute those same functions.

Since S-Nets are made of local units of information processing, it is not difficult to generalize this method to perform parallel computations. There are only the usual problems, like synchronising access to shared resources. We believe that S-Nets do not introduce new fundamental problems in this area.

Future Developments

There are several paths open to exploration in S-Nets. We shall point here some of the most promising.

S/d units refer to synaptic connections with axons, dendrites or with the soma. But they cannot model synaptic-synaptic connections. In graphical terms, we need something like this:

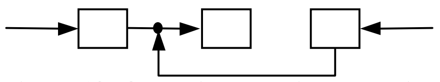
Figure 12- Synaptic-synaptic connection.

What kind of functions can be used in this new connection type? Some proposals fall into partial or total inhibition of that specific connection (or perhaps a probabilistic inhibition); simple addition of both currents, or even multiplication (introducing synaptic weights).

In the Definition of S-Nets, R is defined as a sub-set of $(G\times S)\cup(S\times G)\cup(S\times S)$. It remains one possibility, $(G\times G)$ connections. This means, neurones linked directly, soma with soma. Some interpretations are possible:

a) The introduction of Slow Potential Theory (see, e.g., [Anderson 95]). This theory suggests that the important feature of cell activity is the value of its slow potential, not the presence/absence of action potentials. Action potentials are used only as a way to transmit information through, an otherwise long and poor conductor, the axon. If neurons are close together, perhaps action potentials are not needed.

b) The event of neuronal death or neuronal merging (interesting as a simplification method in a future S-Net construction algorithm).

Another interest point is noise. Several components of the neuron are intrinsically noise sources, like ion flows through ion channels, or the rate of neurotransmitter release stored on synaptic vesicles. Complex dendritic trees and intricate synaptic connections can increase noise and create distortion, which can affect information transmission. These can be approached by S-Nets, if we change the information processing functions in order to handle noise. Noise probably inserts the need of rational numbers in activation values.

Conclusion

We have presented a model that tries to grasp the internal complexity of real neural networks. With only four simple types of information processing units, we have shown that S-Nets can compute all partial recursive functions.

We think that S-Nets have potential to represent many subtle structures existing in central nervous systems, and perhaps eventually, they can help us to understand a little more of what is going on. There is a lot to do, but fortunately, there are many new directions to improve this model, as shown in the previous section.

Bibliography

ANDERSON, J. (1995). *An Introduction to Neural Networks*. Massachusetts Institute of Technology.

ARBIB, M (1989). *The Metaphorical Brain 2. Neural Networks and Beyond*. John Wiley & Sons.

CUTLAND, N. (1988). *Computability. An Introduction to Recursive Function Theory*. Cambridge University Press.

McCULLOCH, W.; PITTS, W.(1943). *A logical calculus of the ideas immanent in nervous activity*. *Bulletin of Mathematical Biophysics*, 5, pp.115-33.

MEL, B. (1994). “Information Processing in Dendritic Trees”. *Neural Computation*, 6, pp. 1031-85. Massachusetts Institute of Technology.

SHEPHERD, G. (1994). *Neurobiology*. Oxford University Press.

SIEGELMANN, H.; SONTAG, E. (1995). “On the Computational Power of Neural Nets”, in *Journal of Computer and System Sciences*, Vol. 50, No.1. Academic Press.

Appendix: The Unlimited Register Machine (URM)

The URM has an infinite number of *registers* labelled R_1, R_2, … each one containing a natural number. The value contained in R_i, is denoted by r_i. These values can be altered by the URM in reply of some very simple *instructions* that the machine do recognise. A finite list of instructions establishes a *program*. There are four types of instructions,

- Zero instructions. Syntax: Z(n); $n \in \omega^+$. Change value of R_n to 0 ($r_n := 0$).
- Successor instructions. Syntax: S(n); $n \in \omega^+$. Increase value of R_n by 1 ($r_n := r_n+1$).
- Transfer instructions. Syntax: T(n,m); $n,m \in \omega^+$. Replace content of R_n, by r_m ($r_n := r_m$).
- Jump instructions. Syntax: J(n,m,k); $n,m,k \in \omega^+$. If the values of R_n and R_m are equal, jump to the k*th* instruction. If not, proceed to the next instruction (if $r_n=r_m$ then goto k else nil). Jump instructions do not change any registers, only the program *execution*.

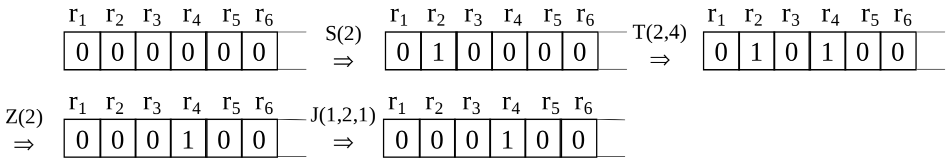

Figure 13- Some possible instructions.

We can define the execution of program P = $<I_1, I_2, \ldots, I_{|P|}>$, as follow. The URM starts executing I_1. Suppose the URM as just performed I_i. Then it proceeds to the next instruction, defined as: if I_i is a Zero, Successor or Transfer instruction, then the next instruction is I_{i+1}; if it is a Jump instruction, and $r_n=r_m$, then the next one is I_k, if $r_n \neq r_m$ the next is I_{i+1}. The URM continues as long as possible, i.e., the machine stops if and only if there is no next instruction.

A URM-program P computes a function $f:\omega^n \rightarrow \omega$, iff, $\forall(x_1,\ldots, x_n) \in Dom(f)$, P with input $(x_1,\ldots, x_n)$, converges to $f(x_1,\ldots, x_n)$, i.e., P ends and $f(x_1,\ldots, x_n)$ is stored in some register (usually in R_1). A function is URM-computable if there is a program that computes f. All partial recursive functions are URM-computable (for a good introduction see [Cutland 88]).