# Merging Subsymbolic and Symbolic Computation

João Pedro Neto
J. Félix Costa
Ademar Ferreira

## Motivation

- The majority of the work done in last decades on neural networks comes from the connectionist school. The complementary approach of symbolic computation using neural nets is not well known.
- Integrating symbolic processes in the typical neural architectures and algorithms may result on greater expressiveness and usability of neural nets.
- Application demands for neural nets use are getting larger and wider. Modular mechanisms are needed in order to minimize and control the subsequent increase of system complexity.
- If an automated translation between a symbolic processing algorithm and a neural network is found, the conceptual bridge between symbolic and connectionist paradigms may be narrowed.

## Historical Remarks

- 1943 – W. McCulloch and W. Pitts introduced neural networks of binary neurons as a computational paradigm, and they have shown that these nets have the power of finite automata.
- 1987 – J. Pollack shows that recurrent neural nets can be universal. However, neurons possess high-order transfer functions.
- 1989 – S. Franklin and M. Garzon present another universal model. The authors assume an unlimited number of neurons, in order to simulate the Turing unbounded memory model.
- 1991 – D. Wolpert shows that neural nets may have Turing and super-Turing computational lower bounds, in a model with an uncountable number of neurons.
- 1993 – H. Siegelmann and E. Sontag introduced their recurrent finite neural network model with Turing power, using rational weights.

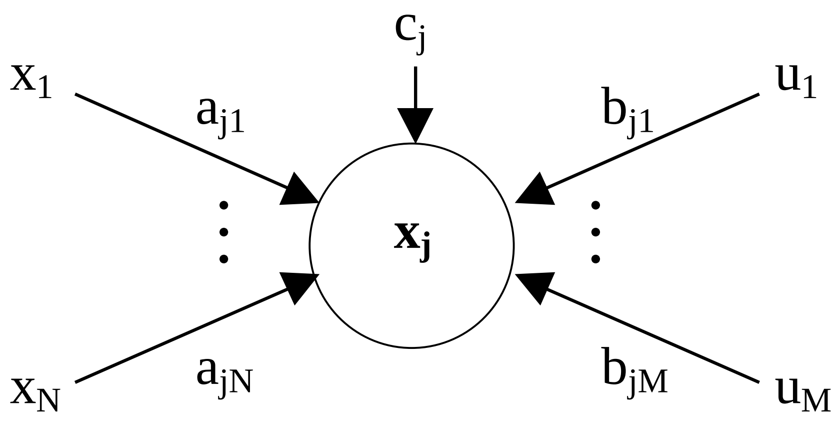
## Objectives

- To find a neural net model able to support symbolic computation.
- To define a base set of processes with universal properties, so that they can be composed to express whatever desirable algorithm.
- To build computational systems in a modular way, in such a way that the ‘symbolic properties’ of a net can be reused, controlled and applied to larger architectures.
- To automate the translation mechanism between the algorithm and the final neural net.
- To establish interfaces between symbolic processes and learning processes to achieve hybrid computational systems.

# The Neural Model

- The recurrent neural net is a discrete time dynamic system, where each neuron is connected through rational weighted wires ($a_{ji}$ and $b_{jk}$) to neurons $x_i$ and to input channels $u_k$. The update neuron value is given by the following equation:

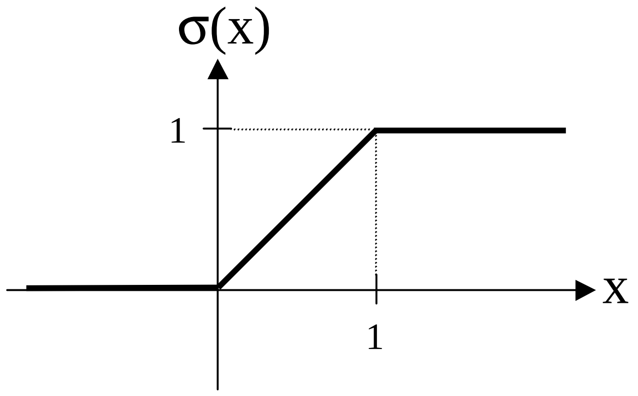
$$x_j(t+1) = \sigma\left( \sum_{i=1}^{N} a_{ji} x_i(t) + \sum_{k=1}^{M} b_{jk} u_k(t) + c_j \right)$$



## The Neural Model

- The network is homogeneous. All neurons have the sigma activation function, also known as saturated linear function σ:

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



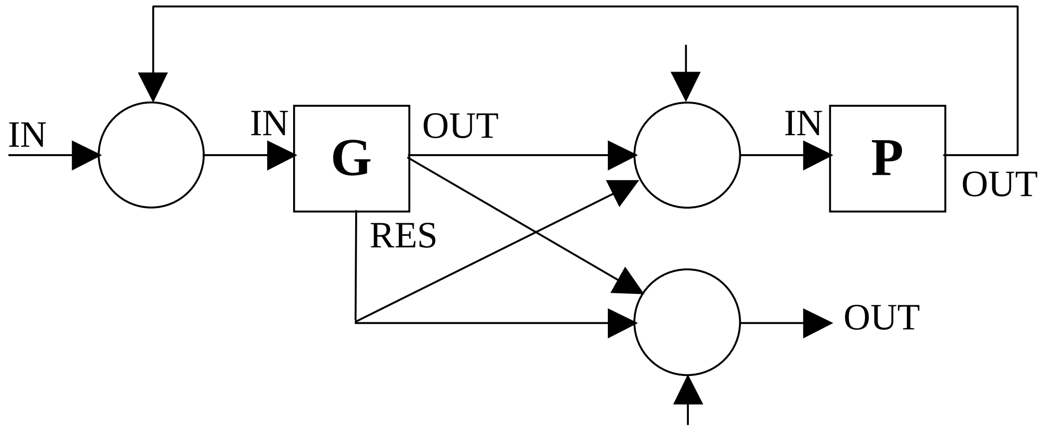
## Fundamental Concepts

- We based our language design on the Occam language. Main concepts: Variables, Processes and Channels.
- A **Variable** is a memory cell able to hold a value for an unspecified amount of time.
- A **Process** (or **Module**) is an independent and modular execution block, including an optional set of internal variables.
- A **Channel** is an unbuffered, unidirectional point-to-point communication of values between two concurrent processes.
- An Algorithm can be described as a collection of processes executing concurrently and communicating with each other through channels or shared memory (i.e., the scoped variables).

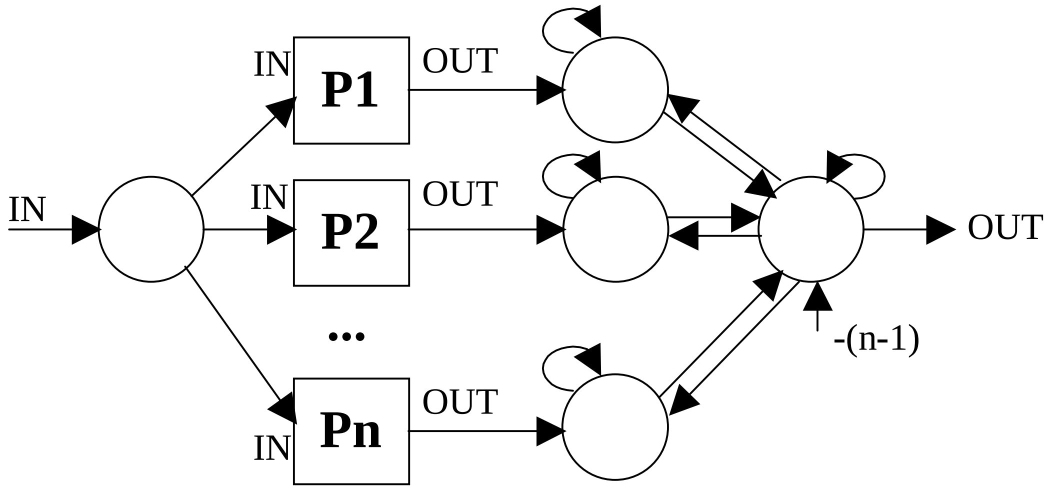
## NETDEF

- NETDEF is a high level parallel language able to describe algorithms.
- It has assignment, conditional (IF ... THEN ... ELSE) and iteration processes (WHILE ... DO).
- Has sequential (SEQ) and parallel (PAR) composing block processes.
- Includes channel handling processes (SEND and RECEIVE).
- Uses a modular synchronization mechanism based on handshaking for process ordering.
- Is able to represent different value types, namely boolean, integers and reals, and apply operations on them.

# NETDEF Processes



**WHILE G DO P;**



```
PAR
   P1;
   P2;
   ...
   Pn;
ENDPAR;
```

# NETDEF Programs

- It is possible to describe algorithms in NETDEF. E.g.,

```
SEQ
  VAR i, j : integer;

  i := 0;
  WHILE i < 100 DO
   SEQ
     PAR
       i := i + 1;
       RECEIVE j FROM chIN;
     ENDPAR;
     IF j < 0 THEN
       SEND j TO chOUT
     ELSE SKIP;
   ENDSEQ;
ENDSEQ;
```

# NETDEF Programs

```
NETDEF n(1,1)IS

VAR a: INTEGER;
    g: GUARD;

SEQ
  a := succ(a);
  START g;
ENDSEQ;
```

$\Rightarrow$

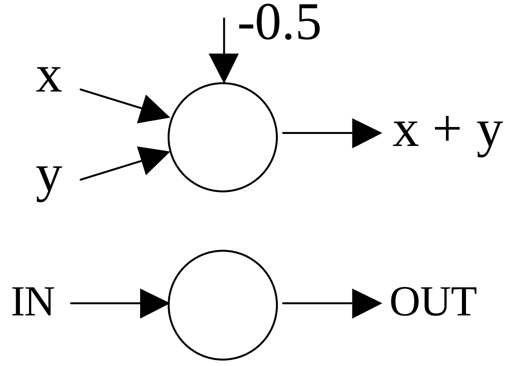
```
Mn0In(i+1) = sigma( )
Mn0Out(i+1) = sigma( + 1.0
 * SEQ1Out(i) )
VARCMn0:IN1(i+1) = sigma(
 + 1.0 * VARCMn0:IN1(i) +
 1.0 * SEND1Val(i) – 1.0 *
 RECV1Wait(i) )
VARCMn0:IN1Flg(i+1) =
 sigma( + 1.0 * Mn0In(i) +
 1.0 * VARCMn0:IN1Flg(i) -
 1,0 * SEND1Flg(i) + 1,0 *
 RECV1Wait(i) )
```

# NETDEF Coding Values

- The information flow between neurons is preserved only within [0, 1], implying that data types must be coded in this interval.
- The real coding for values within [-a, a], where a is a positive integer, is:

$$\alpha(x) = (x + a)/2a$$

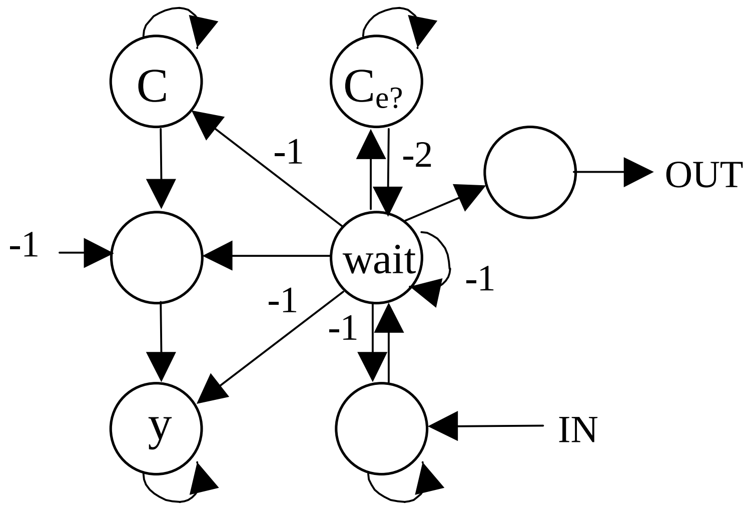
- This coding is a one to one mapping of [-a, a] into the working set [0, 1]. Each operation on real values must deal with this coding. E.g., for the sum x + y, the following network is used:



$$\begin{aligned}
&\alpha(\alpha^{-1}(x) + \alpha^{-1}(y)) = \\
&\quad \alpha(2ax\text{-}a + 2ay\text{-}a) = \\
&\quad \alpha(2a(x+y) -2a) = \\
&\quad ((2a(x+y) -2a) + a) / 2a = \\
&\quad x + y - 0.5
\end{aligned}$$

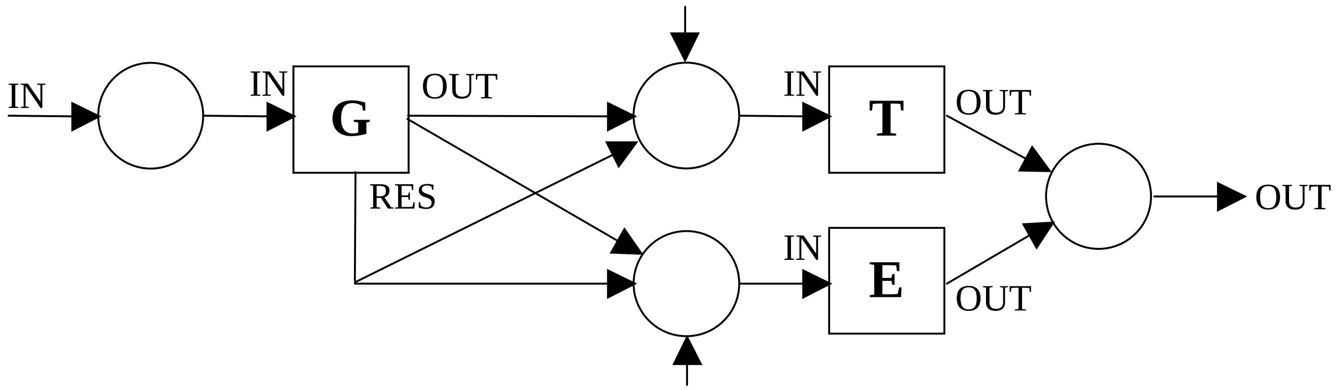
# NETDEF Channels

- Channels are used to communicate between processes and to receive information from outside through **input channels**.
- E.g., RECEIVE y FROM C



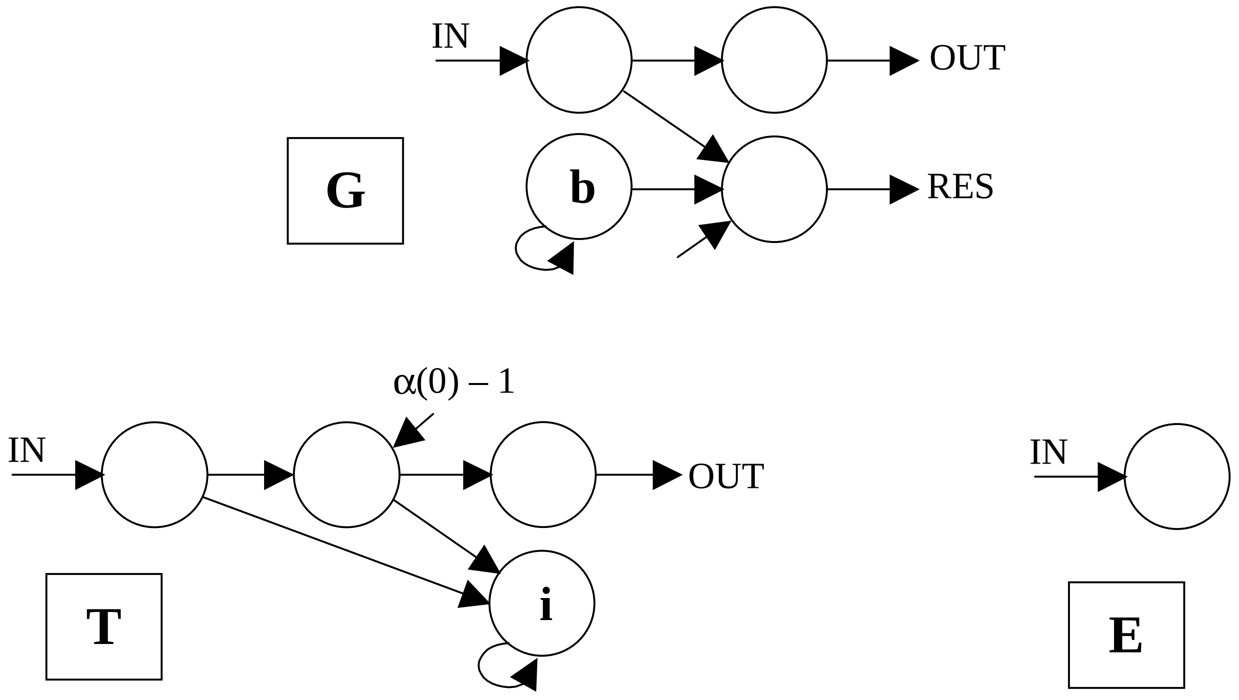
## An example

- Each complex process is decomposed recursively until the basic processes are reached. Then the neural network is composed accordingly.
- An example: IF b THEN i := 0 ELSE STOP;



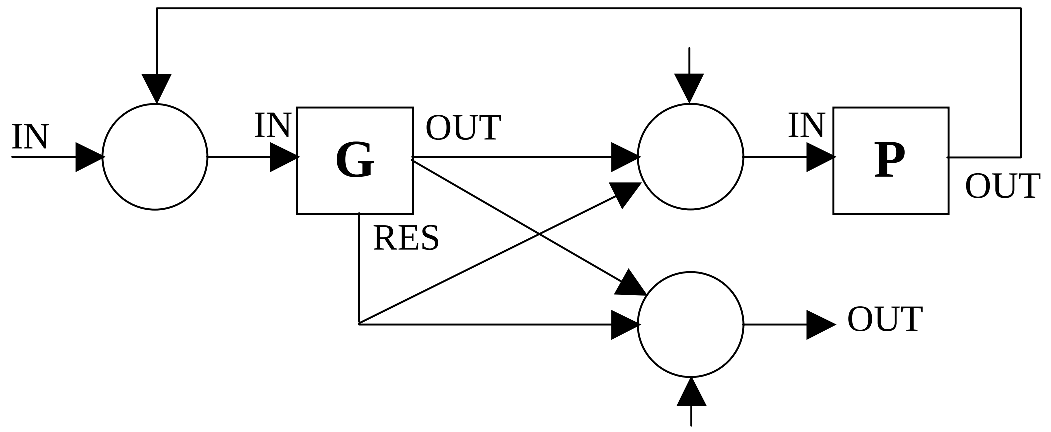
# An example

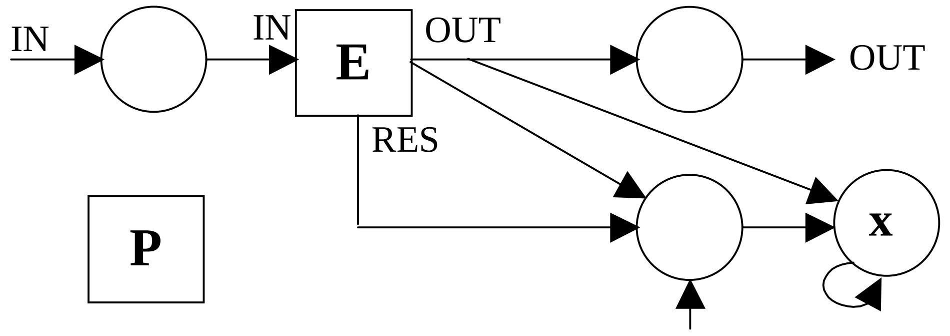
- IF b THEN i := 0 ELSE STOP;

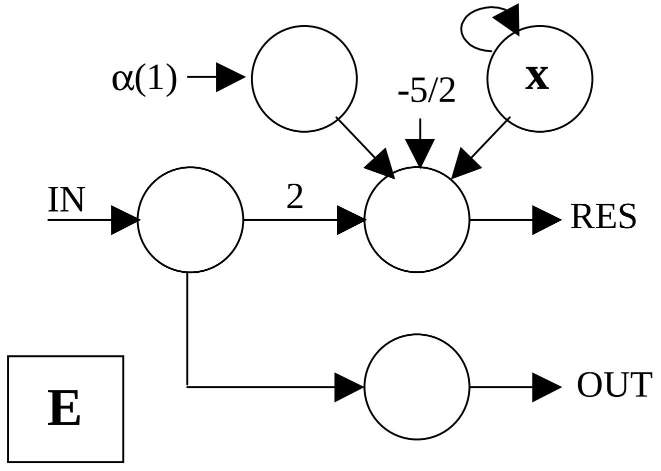


# Another example

- WHILE b DO x := x + 1







## NETDEF Features

- Three primitive data types (boolean, integer and real). Unidimensional arrays and channel data type.
- **Guard** data type. A specific modular net is only activated when an associated guard is activated. This enables independent and parallel neural networks to perform tasks only when a certain condition occurs.
- **Lock** data type. These complex structures can be used to perform mutual exclusion mechanisms in modules. Useful in circumstances where it is possible for two processes to write in a variable at the same time.
- **Functions** and **Procedures** declarations. However, due to the fixed network size, there is no recursion mechanism in NETDEF.

## NETDEF Features

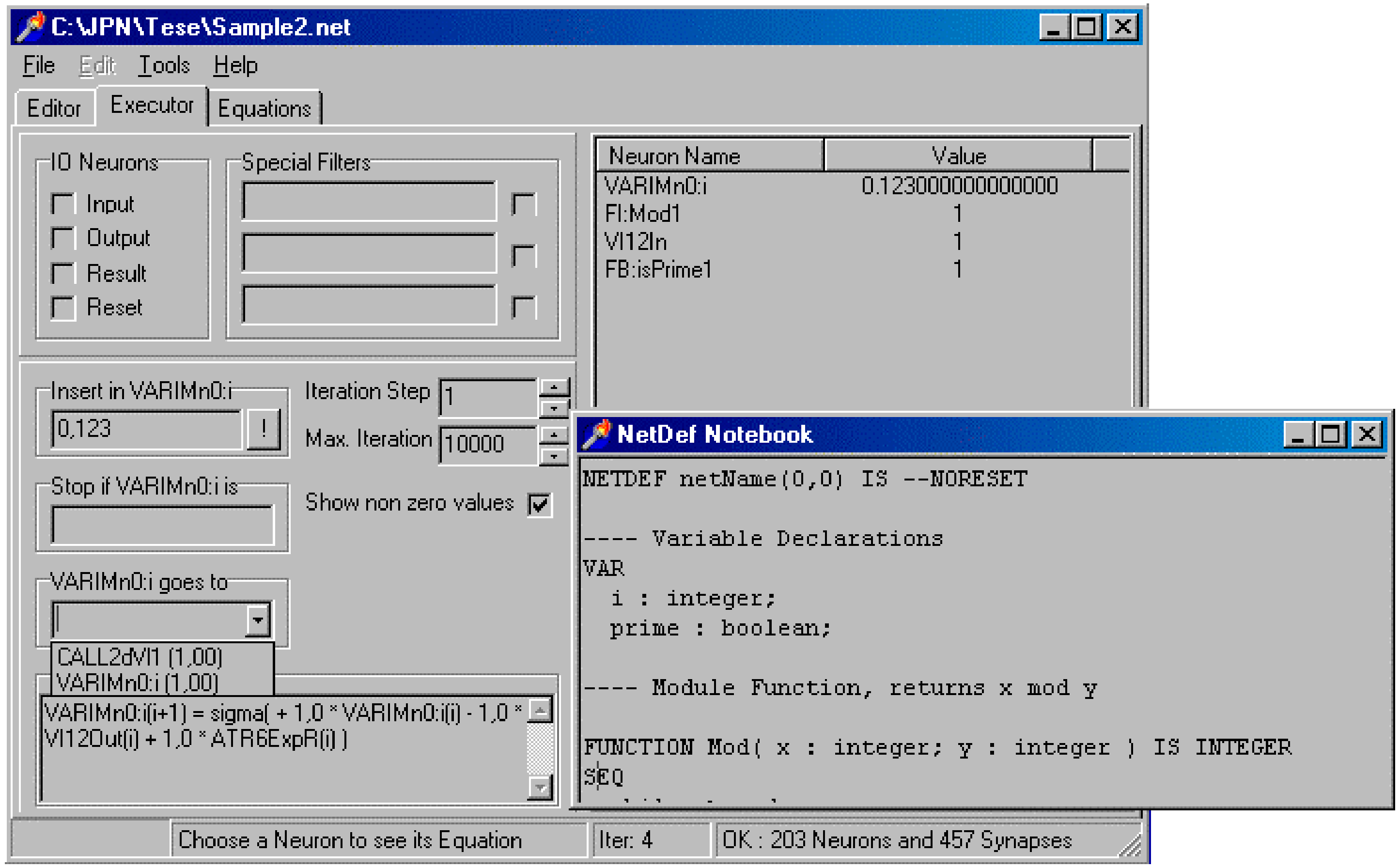
- **Timers**. The main idea of timers is to give a certain amount of time for a module to execute. If the module is blocked or enters a deadlock state, the timer can end or repeat the module execution, after a certain time limit is reached. E.g.,

```
X := 1000;
Y := 50;
CYCLE (x) TRY(y) SEND x TO c;
```

- **Exceptions**. There is a simple exception handling mechanism for SEQ and PAR blocks giving more control of the module internal structure. There are also some special directives like ABORT, RETRY or RESET.

# NETDEF Compiler

- There is neural compiler based on NETDEF language (available at *www.di.fc.ul.pt/~jpn/netdef/netdef.htm*)



0,123

Iteration Step: 1
Max. Iteration: 10000

CALL2dVI1 (1,00)
VARIMn0:i (1,00)

VARIMn0:i(i+1) = sigma( + 1,0 * VARIMn0:i(i) - 1,0 * VI12Out(i) + 1,0 * ATR6ExpR(i) )

| Neuron Name | Value |
|---|---|
| VARIMn0:i | 0.123000000000000 |
| FI:Mod1 | 1 |
| VI12In | 1 |
| FB:isPrime1 | 1 |

```
NETDEF netName(0,0) IS --NORESET

---- Variable Declarations
VAR
  i : integer;
  prime : boolean;

---- Module Function, returns x mod y

FUNCTION Mod( x : integer; y : integer ) IS INTEGER
SEQ
```

Choose a Neuron to see its Equation | Iter: 4 | OK : 203 Neurons and 457 Synapses

# NETDEF+

- Expands the model. It should be:
  - *Simple* – the neural net model should remain as close as possible to its initial formulation.
  - *Expressive* – the neural net model should be expressive enough to model new tools and new mechanisms.
  - *Modular* – there is always specific concern about modularity; which must be preserved in the extension.

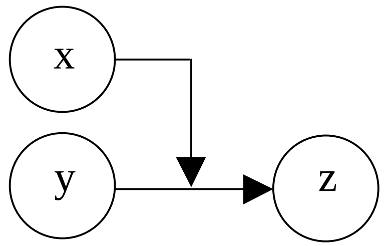
- The update neuron equation becomes:

$$x_j(t+1) = \sigma(\ P_j(x_1(t), \ldots, x_N(t), u_1(t), \ldots, u_M(t))\ )$$

where $P_j(.)$ is a polynomial with rational weights.

# NETDEF+

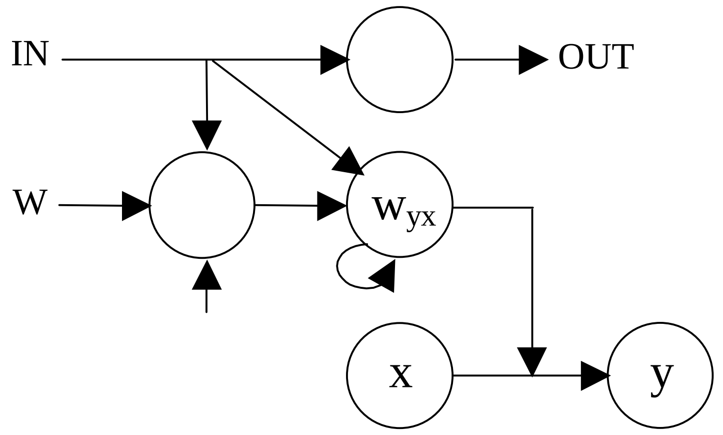
- This new model includes what we call neuron-synaptic synapses. E.g.,



$$z(t+1) = \sigma(2a.x(t).y(t) - a(x(t) + y(t)) + 0.5a + 0.5) \longleftarrow \alpha(\alpha^{-1}(x) * \alpha^{-1}(y))$$

## Synaptic Weight Changes

- It is possible for a module to change some of its synaptic weights. E.g.,



- This feature can be used to implement learning algorithms.

## Learning Modules

- Integrating Control and Learning
  - **Control structure:** the NETDEF language is used to regulate learning processes (it is flexible to handle arbitrary algorithms). Usually, learning algorithms consist of several weight calculations and they define how the entire module should change in order to respond in a new way to the environment.
  - **Learning structure:** a set of neurons arranged in an appropriate architecture (in layers, in a bidimensional grid), keeping the knowledge acquired during the learning procedure.

# Learning Modules

- **Interface structure:** to manage the communications with the remaining network. To achieve maximum transparency between symbolic and sub-symbolic computations, each learning module is seen from the outside as a special kind of function.

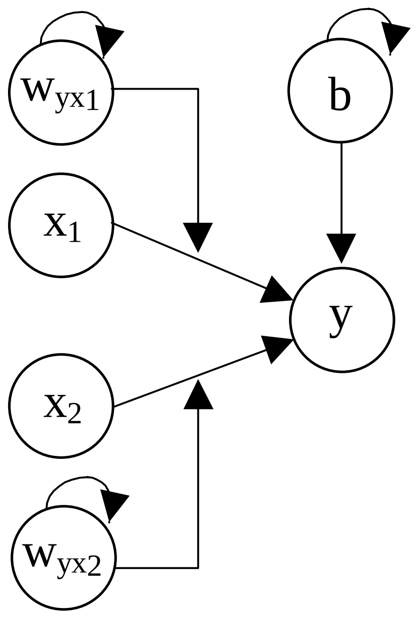
- Two principal operators:
  - LEARN – insert a training sample into the learning module, and adapt the network.
  - CLASSIFY – classify a certain test sample.

```
F[X] := [-1, 1, 0];   -- sample
F[D] := [0,1];        -- desired result
LEARN F;
```

```
F[X] := [-1, -1, -1]; -- sample
CLASSIFY F;
F[Y]                  -- network result
```

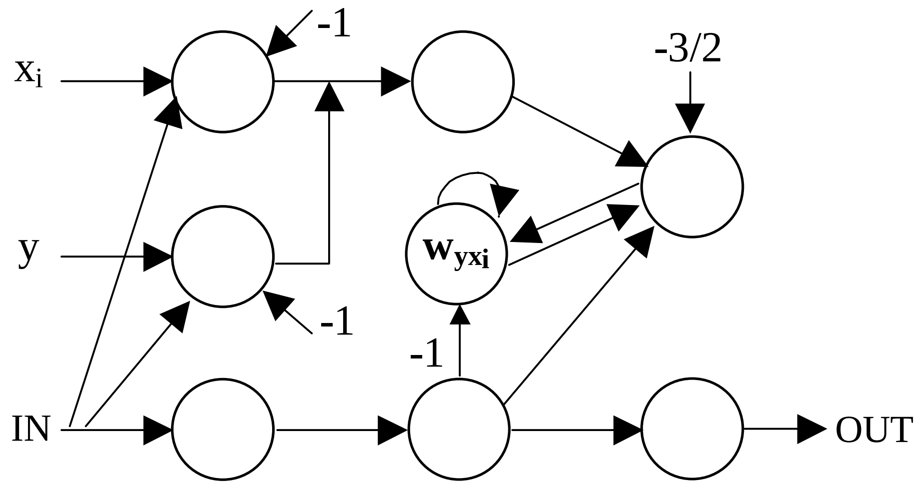
# Hebb Rule

- Associative rule $\Delta w_{yx} = \eta . x . y$
- Hebb Learning structure:



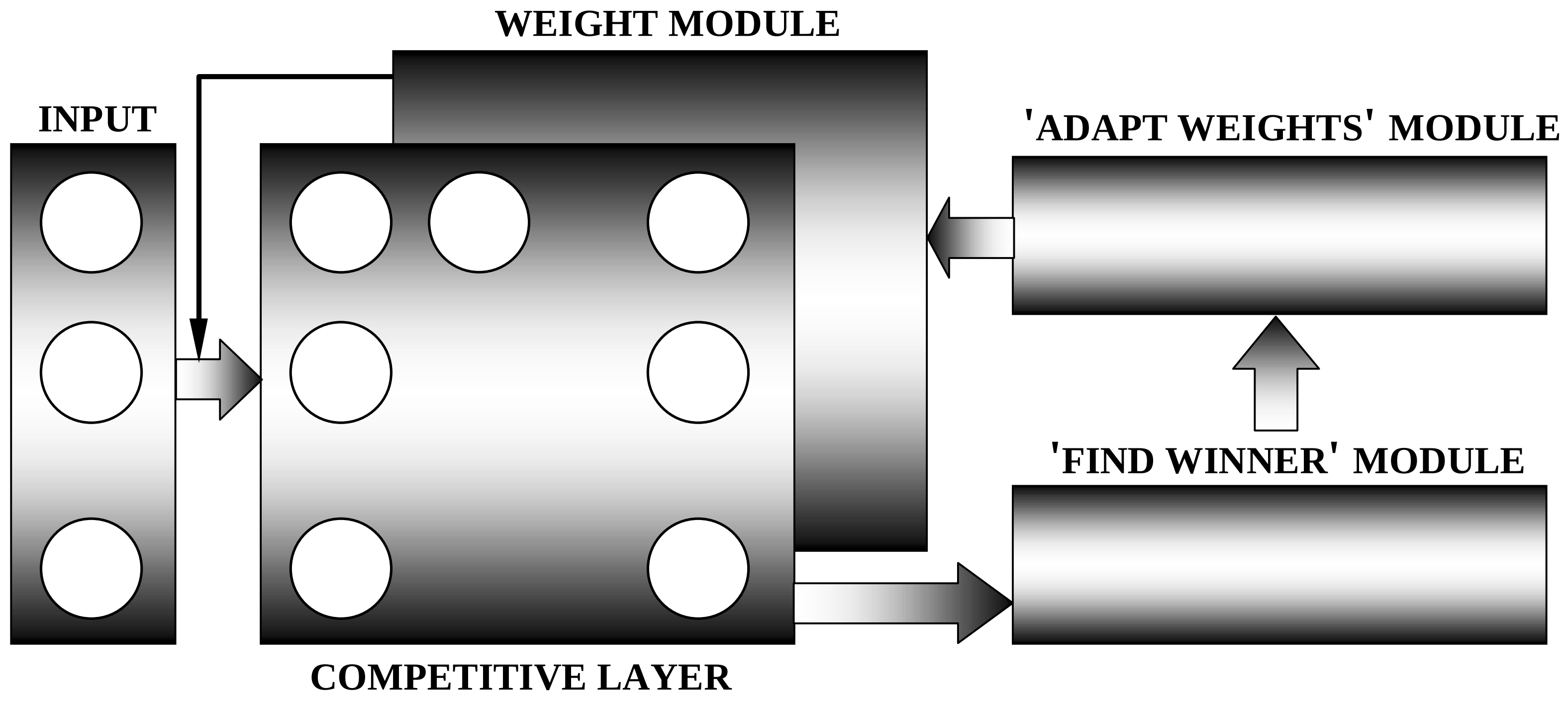
# Hebb Rule

- Hebb Control structure (updating synapse $w_{yx}$):



# Competitive Learning

- Module interaction: The input is connected to the competitive layer via a sub-symbolic module that keeps all synaptic weight values (defined by competitive learning rules, using the typical "winner takes all" strategy).



## Competitive Learning

- All computations are made by modular neural networks.
- The learning module executes symbolic and sub-symbolic computations within the same neural net framework.
- The symbolic sub-modules (finding the winner, adapting the winner weights and changing any needed parameter, as in the learning step) are rather complex. For a N dimensional input with a $M\times M$ competitive layer, the symbolic part needs approximately $2.N.M^3$ neurons.
- The execution time of a sample adaptation does not depend on the problem dimension.

## Conclusions and Future Work

- NETDEF data types, operations and control structures show that it is possible to perform symbolic computation over neural nets.
- Since NETDEF has memory, assignment, conditional, and iteration processes, it is able to compute any algorithm within the available memory.
- It is also a modular approach to computation, based on process and channel primitive concepts, and the handshaking mechanism of IN/OUT interface.
- NETDEF programs are compiled by a specific and available compiler.
- The modular approach of NETDEF networks eases the integration with other tools, namely with learning processes.

# Conclusions and Future Work

- To effectively integrate learning. This can be done by using higher-order transfer functions. The new model simulates a type of neuron-synaptic synapses capable to self-modify their own weights.
- Investigate what learning algorithms can be executed.
- Optimize the compiler and the language. A NETDEF network has just a linear delay factor and a linear space complexity compared to the original algorithm.
- Study what other activation functions are capable of performing symbolic computational tasks (specially the logistic function). Related problems are noise and computation robustness.