Proceedings of V Brazilian Conference on Neural Networks - V Congresso Brasileiro de Redes Neurais
pp. 19-24, April 2-5, 2001 - Rio de Janeiro - RJ - Brazil

Competitive Learning and Symbolic Computation Integration

João Pedro Neto[1], Helder Coelho[1], and Ademar Ferreira[2 *]
[1]Faculdade de Ciências, Dept. Informática, Bloco C5, Piso 1, 1700 Lisboa – PORTUGAL
[2]Escola Politécnica, L.A.C., Av. Prof. Luciano Gualberto, 158, Trav. 3, 05508-900, SP – BRASIL
E-mails: jpn@di.fc.ul.pt, hcoelho@di.fc.ul.pt, ademar@lac.usp.br

Abstract

In some recent works, it was shown that any algorithmic description might be mapped on a recurrent neural network. A neural oriented language called NETDEF, such that each program corresponds to a modular neural net that computes it, is the tool to achieve this.

This article focuses on merging symbolic and sub-symbolic computation. Adding high-order neurons to the network model allows learning integration into the NETDEF symbolic computing paradigm, since it is possible to execute learning algorithms in the same neural model that performs symbolic computation. It is shown how the model may process competitive learning methods inside this framework.

1. Introduction

The significance of systems integrating symbolic and sub-symbolic computing techniques is already consolidated (see [15] for a hybrid systems analysis). Motivation for this structural hybridization can be found in biology (humans are able to process high level concepts supported by the brain's neural biochemistry) and in engineering (intelligent control design tends to incorporate symbolic and sub-symbolic processing).

There are several different ways to accomplish this hybridization. Some models separate the computation methodologies, using sub-symbolic output structure as an input to the classical AI control schemata (cf. [6]). Others apply symbolic and sub-symbolic information in the same data structure, as in [8] and in this work. This paper uses a method that merges symbolic and sub-symbolic computation into a single neural network architecture (cf. [11]).

First, we briefly introduce the high-level programming language NETDEF to hard-wire the neural network model in order to perform *symbolic computation*. Programs written in NETDEF can be converted into neural nets through a compiler available at *www.di.fc.ul.pt/~jpn/netdef/netdef.htm*.

Secondly, using special constructs named neuron-synapse connections, it is possible to add learning processes to NETDEF. Since the system is modular, after compilation, we get modules performing programming tasks and modules supporting sub-symbolic tasks.

2. NETDEF

In some recent works, it was shown that any algorithmic description might be mapped on a recurrent neural net (see [9] and [10] for details). Herein, it is presented the mathematical model used to encode and process symbolic computations.

The analog recurrent neural net model is a discrete time dynamic system, $\mathbf{x}(t+1) = \phi(\mathbf{x}(t), \mathbf{u}(t))$, with initial state $\mathbf{x}(0) = \mathbf{x}_0$, where t denotes time, $x_i(t)$ denotes the activity (firing frequency) of neuron i at time t, within a population of N interconnected neurons, and $u_k(t)$ denotes the value of input channel k 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 piecewise linear function σ:

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

The dynamic system becomes,

$$x_j(t+1) = \sigma\left(\sum_{i=1}^{N} a_{ji} x_i(t) + \sum_{k=1}^{M} b_{jk} u_k(t) + c_j \right) \qquad (2)$$

Where a_{ji}, b_{jk} and c_j are rational weights. Figure 1 displays a graphical representation of a single equation, used throughout this paper. When a_{ji} (or b_{jk} or a_{jj}) takes value 1, it is not displayed in the graph.

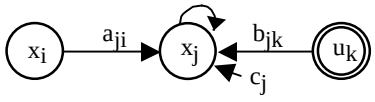

Figure 1: Graphical notation for neurons, input channels and their interconnections.

* The work of A. Ferreira was supported in part by FAPESP via Grant No. 97/04668-1

NETDEF is an imperative language, with syntax and semantic very close to those of Occam. Its main concepts are processes and channels. A program can be described as a collection of processes executing concurrently, and communicating with each other through channels or shared memory.

The language has assignment, conditional and loop control structures (Figure 2 presents a recursive and modular construction of a process), and it supports several data types, variable and function declarations, and many other processes. It uses a modular synchronization mechanism based on handshaking for process ordering (the IN/OUT interface in Figure 2). A detailed description of NETDEF may be found in [10] at *www.di.fc.ul.pt/biblioteca/tech-reports*.

The information flow between neurons, due to the activation function σ, 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 a one to one mapping of [-a, a] into the working set [0, 1]:

$$\alpha(x) = (x + a)/2a \qquad (3)$$

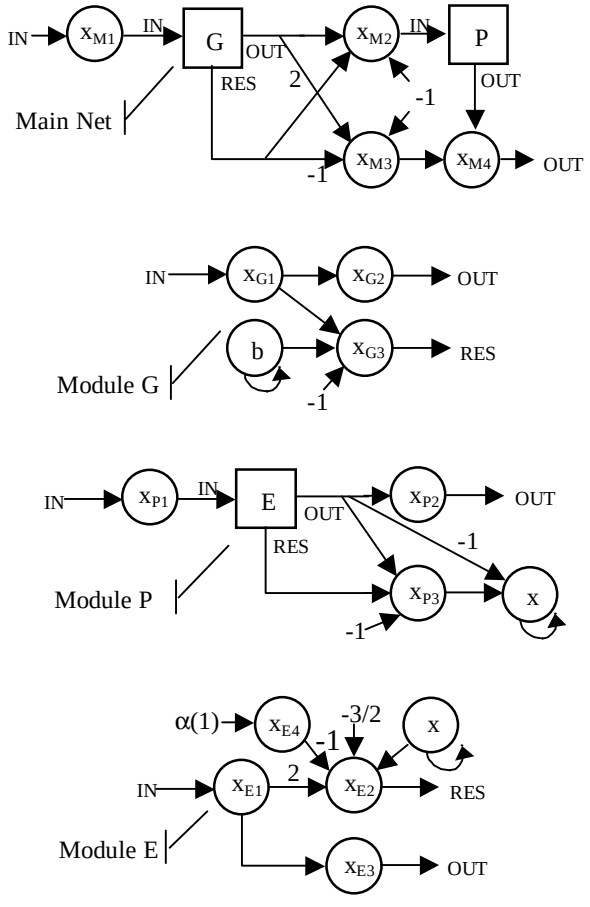

Figure 2. Process construction for IF b DO x := x–1.

Input channels u_i are the interface between the system and the environment. They act as typical NETDEF blocking one-to-one channels. There is also a FIFO data structure for each u_i to keep unprocessed information (this happens whenever the incoming information rate is higher than the system processing capacity).

The compiler takes a NETDEF program and translates it into a text description defining the neural network. Given a neural hardware, an interface would translate the final description into suitable syntax, so that the neural system may execute. The use of neural networks to implement arbitrary complex algorithms can be then handled through compilers like NETDEF.

As illustration of a symbolic module, figure 2 shows the process construction for IF b DO x := x–1. Synapse IN sends value 1 (by some neuron x_{IN}) into x_{M1} neuron, starting the computation. Module G (denoted by a square) computes the value of boolean variable 'b' and sends the 0/1 result through synapse RES. This module accesses the value 'b' and outputs it through neuron x_{G3}. This is achieved because x_{G3} bias -1.0 is compensated by value 1 sent by x_{G1}, allowing value 'b' to be the activation of x_{G3}. This result is synchronized with an output of 1 through synapse OUT. The next two neurons (on the Main Net) decide between entering module P (if 'b' is true) or stopping the process (if 'b' is false). Module P makes an assignment to the real variable 'x' with the value computed by module E. Before neuron x receives the activation value of x_{P3}, the module uses the output signal of E to erase its previous value. In module E the decrement of 'x' is computed (using $\alpha(1)$ for the code of real 1). The 1/2 bias of neuron x_{E2} for subtraction is necessary due to coding α.

The dynamics of neuron x is given by (4). However, if neuron x is used in other modules, the compiler will add more synaptic links to its equation.

$$x(t+1) = \sigma(x(t) + x_{P3}(t) - x_{E3}(t)) \qquad (4)$$

This resulting neural network is homogenous (all neurons have the same activation function) and the system is composed only by first-order neurons. The network is also an independent module, which can be used in some other context. Regarding time and space complexity, the compiled nets are proportional to the respective algorithm complexity.

There are some related works in the literature about symbolic neural computation. Article [3] introduces JANNET, a dialect of Pascal with some parallel constructs. This algorithmic description is translated, using several automated steps to produce a non-homogenous neural network (with four different neuron types) able to perform the required computations. In JANNET, every neuron is activated only when all its synapses have transferred their values. Since this may not occur at the same instant, the global dynamics is not synchronous.

Another neural language project is NIL (cf. [13]). The NIL system is able to perform symbolic computations by using certain sets of constructions that are compiled into an appropriate neural net. An important difference is that NETDEF has a modular design, while NIL has not. Also, NIL does not provide essential mechanisms required for a neural language like a mutual exclusion scheme for variable access security, temporal processes for real-time applications, genuine parallel calls of functions and procedures, blocking communication primitives for concurrent process interaction, dynamic array assignment. NETDEF deals and solves all these subjects without loosing its modular properties.

These works also focus the symbolic potential on neural network computation, and present valid techniques for neural network construction, but they do not try to merge sub-symbolic processing within the same neural framework, while it is in this perspective that rests the originality of this work.

3. Learning Processes

The next step was to integrate learning and hard-wiring mechanisms into the computation tools already developed, merging two standard computation methodologies (symbolic and sub-symbolic) in a single neural architecture. To accomplish these requirements, the model was extended in order to include neuron-synaptic connections. Although we are not concerned with biological plausibility, neuron-synaptic connections in the brain are known to exist in the complex dendritic trees of genuine neural nets, [12]. Herein, their task is to convey values and use them to update and change other synaptic weights.

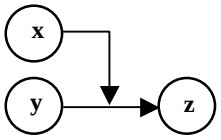

Figure 3. Graphical notation for neuron-synapse connection in equation (5).

Figure 3 displays the diagram of a neuron-synaptic connection, linking neuron x to the connection between neurons y and z. Semantically, synapse of weight w_{zy} receives the previous activation value of x.

The dynamics of neuron z is defined by the high-order dynamic rule[1]:

$$z(t+1) = \sigma(\ 2a.x(t).y(t) - a(x(t) + y(t)) + 0.5a + 0.5)\ ) \qquad (5)$$

[1] The expression is the result of $\alpha(\alpha^{-1}(x(t))*\alpha^{-1}(y(t)))$. This calculation is necessary because the data flow values are encoded through α, given by (3). To avoid ambiguities, the first argument refers to the neuron-synapse connection, and the second, to the input neuron.

High-order nets (i.e., networks having neurons with high-order input-output relation) can be found on the scientific literature (cf. [1], [2]).

This inclusion, introduces some useful tools into NETDEF. Namely, direct multiplication (due to the special transfer function), deletion and insertion of connections in execution time (because it is possible to change synaptic weights at runtime), and the key feature, learning.

Many neural learning algorithms receive inputs and adapt the synaptic weights, adapting the network structure towards the problem solution. Each algorithm uses some appropriate procedures to update the network weights (cf. [5]), inspired by means of pure mathematical reasoning (e.g., the LMS rule) or by biological inspiration (e.g., the Hebb rule).

The *control structure* given by the NETDEF language can be used to regulate learning processes, since it is flexible enough to handle arbitrary algorithms. Usually, learning algorithms consist of several weight calculations and rules that state how the entire module should change in order to respond in a new way to the environment. The *learning structure* consists of a set of neurons, arranged in an appropriate architecture (in layers, in a bidimensional grid), keeping the knowledge acquired during the learning procedure.

The learning module embodies the control structure and the learning structure. This module is affected by outside requests, like processing the information presented by a new learning sample, or resetting the weight values. Control and learning are implemented in the same homogeneous framework, and they are joined together homogeneously. Integration with Hebb-like learning rules was presented at [11]. The focus in this paper is competitive learning networks.

4. Competitive Learning

In an unsupervised learning algorithm, only the attribute values of each training set sample are given. Then, the algorithm must decide how to map them, in order to form a set of reasonable classes or clusters. This method of self-organization is commonly called cluster analysis (cf. [4], [7] and [5]).

The system does not know the desired outputs. The network must rely on what is called competitive learning, where all neurons compete for some learning opportunity and just the 'winner' has the chance to improve. The winner is the neuron that achieves the greater activation value based on those sample input values. If neuron k wins the competition, each weight w_{ki} connected to input **u** is then modified by rule (6),

$$\Delta w_{ki} = \eta\ .\ (\ u_i - w_{ki}\ ) \qquad (6)$$

where η is the learning coefficient.

The weights of the winner are updated in a way to intensify its response to the same input sample. This

way, the winner has strengthened its winning status. After some iteration with the training set, the network will eventually converge to a stability condition and the training process will end. The activation of some specific neurons only occurs on a certain set of input vectors. Then, each neuron embodies a cluster containing those same vectors.

A learning module in NETDEF consists of a set of interconnected neuronal structures. Some of these structures are used to keep the needed information for learning process execution, namely, the input sample, the actual synaptic weight matrix and the desired net response (for supervised learning algorithms).

The internal execution of a learning module in order to learn a sample takes the following steps (after both module definition and initialization have occurred):

1. Loading input sample in appropriate neural structure (called X);
2. Sample initialization procedure (optional step);
3. Matrix multiplication of X by W (the synaptic weight data structure), and assignment of result to Y (another neural structure keeping the net response);
4. Synaptic weight (and bias) update by a given rule;
5. Update of internal variables (like iteration turn number and average synaptic change);
6. Sample finalization procedure (optional).

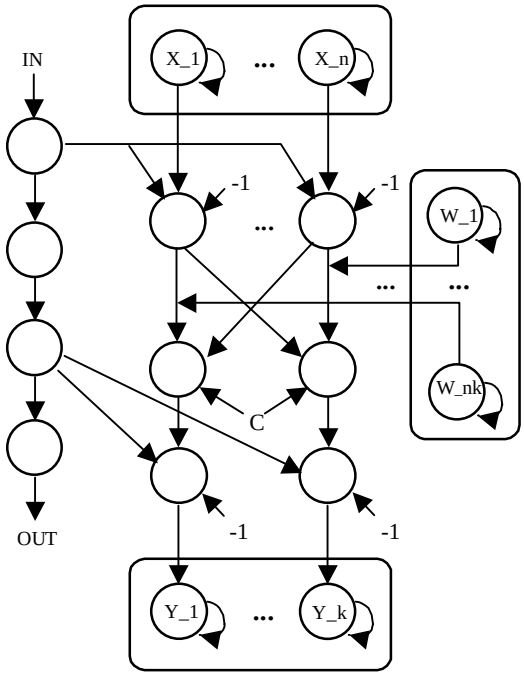

Figure 4. Computing net response.

Classification is simple. The module just executes steps 2 and 3, and saves the result on vector Y. The execution time of learning and classifying takes only a constant time delay. This is achieved because of the neural network massive parallelism and the increase in neuron number (i.e., spatial complexity).Let's examine in detail the various learning process steps. Step 1 is conducted by symbolic structures such as vector assignment, from the input values into the internal input vector X. Step 2 is made optionally in the beginning of the process, in order to execute some procedural task that will prepare the net to learn that specific sample (some symbolic module does this).

The third step computes net response, given the input values already kept in vector X, and the actual weight matrix values computed from previous iterations (kept in matrix W). Figure 4 shows the neural network that performs this task. The boxed neurons represent the data structures keeping input data, weight matrix and proper net response, namely, vectors X, W and Y.

Matrix multiplication is achieved by neuron-synaptic connections (to correctly sum all coded values, the net must insert after each multiplication a value of $C = -0.5*(n-1)$, see figure 4) and the information flow is controlled by NETDEF synchronization mechanism.

Step 4 depends on the specific learning algorithm used. Herein, competitive learning provides equation (6) as the update rule of the winner neuron. In this case, before proper weight update, the module must find which neuron is the winner, keeping all others unaffected. This is the most complex structure of NETDEF competitive learning modules.

In figure 5, it is shown the network that checks if the j^{th} output neuron is the winner (all output neurons will have a similar network).

M is the minimum integer value with the following property: For any x, if $\alpha(x) > 0$ then $M.\alpha(x) \geq 1$. This is used to extract any residual value after an invalid subtraction between two codes (i.e., if there is any residual value from $\alpha(x) - \alpha(y)$, then $x>y$).

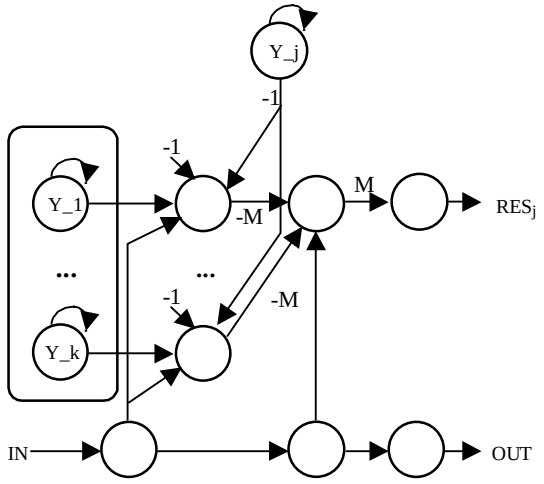

Figure 5. Finding if Y_j is the winner.

The net subtracts all other output activations with the target neuron Y_j. If neuron Y_j achieves maximum

activation, it will cancel all eventual residuals from those subtractions, and the input signal will pass through RES_j outputting 1.

Otherwise, at least one residual will exist, thus canceling the input signal. In this way, only one neuron, the winner, will activate its output result.

If by some rare chance (since weight values are initialized to small random values) two neurons are both declared winners, the net will not learn the sample until next iteration (when the weight matrix will be on a different state).

After finding the winner, the network should update the correct synaptic connections. This computation is outlined in figure 6.This network performs the updating rule described by equation (6). Basically, it subtracts w_{ji} from x_i, to multiply the result by the learning coefficient η. This result is then sent to the proper position at the weight matrix, updating the synaptic connections of the winning neuron.

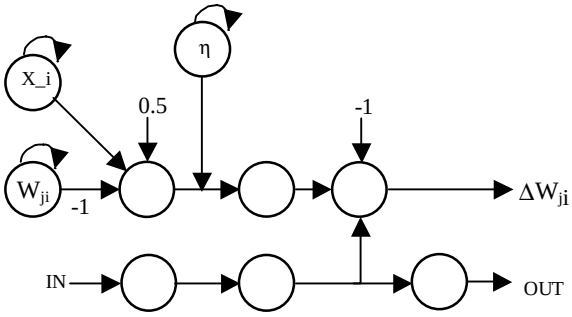

Figure 6. Evaluating ΔW_{ji}.

Finally, one or more symbolic modules conduct any proper internal variable updating and also execute an eventual finalization procedure. One example is now presented in this article. Usually, in competitive learning algorithms, the learning coefficient is a function of time, where its values are given by formula (7).

$$\eta(t) = 0.1 - t / 10^5 \qquad (7)$$

In figure 7, neuron T is an internal variable keeping the number of learning iterations already performed, since the last initialization. The net computes the exact formula given by equation (7). So, in the next learning sample, the updating rule (6) will have a different behavior, since the learning coefficient was changed.

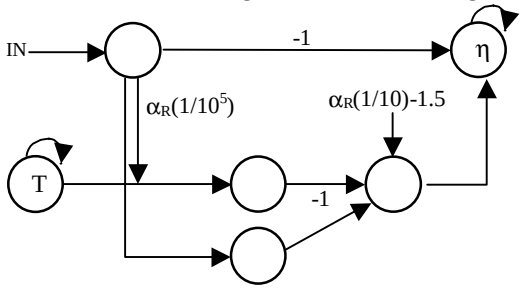

Figure 7. Updating learning coefficient.

The set of neural networks presented in this paper, help the reader perceive the actual interchange between two different ways of computing information and knowledge.

A symbolic module may access, within its scope, any internal structure from the learning module (e.g., the weight matrix or the numeric value of an internal variable) in order to retrieve information gathered from the learning process, and can also consult the proper net response in order to retrieve classification results.

On the other hand, a learning module may access symbolic values from runtime variables of control structures. This may be useful for specific learning algorithms in the initialization and finalization sample procedures.

In this way, a straightforward communication mechanism exists, based on shared variable access, between symbolic and sub-symbolic NETDEF neural modules.

5. Conclusions and Future Work

This paper focuses on competitive learning integration within the NETDEF system, which is a framework for neural networks construction able to describe arbitrarily complex symbolic computations.

An extension of this system, a neuron-synaptic connection model (using second-order neurons), is applied to include learning processes. With this feature, sub-symbolic and symbolic computations are linked together in the same neural framework.

Competitive learning algorithms can be integrated on NETDEF modules, using a modular and recursive approach and with an interface system suitable for information and instruction exchange between symbolic and sub-symbolic elements.

In the future, our main concern focus on how to enlarge the interface mechanism of these NETDEF modules, in order to achieve further integration with other learning algorithms, namely, to extend the net topology of competitive learning nets to include neighbor adaptations like in Kohonen networks.

Other interesting subject is trying to enhance the NETDEF syntax – handling the symbolic and sub-symbolic interaction modules – in order to ease the user programming task.

References

[1] *Giles, C., Miller, C., Chen, D., Chen, H., Sun, G., and Lee, Y.,* Learning and extracting finite state automata with second-order recurrent neural networks, *Neural Computation, [4] 3, 1992, 393-405.*

[2] *Goudreau, M., Giles, C., Chakradhar, S., and Chen, D., First-Order Versus Second-Order Single-Layer Recurrent Neural Networks, IEEE Transactions of Neural Networks, [5] 3, 1994, 511-513.*

[3] Gruau, F., Ratajszczak, J., and Wiber, G., *A neural compiler*, Theoretical Computer Science, [141] (1-2), 1995, 1-52.
[4] Hartigan, J.A., Clustering Algorithms, NY: Wiley, 1975.
[5] Haykin, S., Neural Networks – A Comprehensive Foundation, 2nd ed., Prentice Hall, 1999.
[6] Hendler, J. and Dickens, L., *Integrating Neural and Expert Reasoning: An Example*, In Proceedings of AISB-91, Leeds, 1991. Cited in Wilson, 93.
[7] Kohonen, T., *Self-Organizing Maps*, Berlin: Springer-Verlag, 1997.
[8] Lange, T., Hodges, J., Fuenmayor, M., and Belyaev, L., *Descartes: Development Environment for Simulating Hybrid Connectionism Architectures*, In Proceedings of the 11th Annual Conference of the Cognitive Science Society, Ann Arbor, MI, 1989. Cited in Wilson, 93.
[9] Neto, J.P., Siegelmann, H., and Costa, J.F., *On the Implementation of Programming Languages with Neural Nets*, In First International Conference on Computing Anticipatory Systems, CASYS 97, CHAOS [1], 1998, 201-208.
[10] Neto, J.P. and Costa, J.F., *Building Neural Net Software*, Technical Report DI-99-05, Computer Science Department, University of Lisbon, 1999. Available at *www.di.fc.ul.pt/biblioteca/tech-reports*.
[11] Neto, J., Costa, J., Ferreira, A. (2000). *Merging Sub-symbolic and Symbolic Computation*, In H. Bothe and R. Rojas (eds), Proceedings of the Second International ICSC Symposium on Neural Computation (NC'2000), 329-335. ICSC Academic Press, 2000.
[12] Shepherd, G. M., *Neurobiology*, 3rd ed., Oxford University Press, 1994.
[13] Siegelmann, H., *On NIL: The Software Constructor of Neural Networks*, Parallel Processing Letters [6] 4, World Scientific Publishing Company, 1996, 575-582.
[14] Sun, G., Chen, H., Lee, Y., and Giles, C., *Turing equivalence of neural networks with second-order connections weights*, Proc. International Joint Conference on Neural Networks, IEEE, 1991.
[15] Wilson, A. and Hendler, J., *Linking Symbolic and Subsymbolic Computing*, Technical Report, Dept. of Computer Science, University of Maryland, 1993.