Merging Sub-symbolic and Symbolic Computation

João Pedro Neto[*], J. Félix Costa[**], and Ademar Ferreira[***]
jpn@di.fc.ul.pt, fgc@math.ist.utl.pt, and ademar@lac.usp.br

[*] Faculdade de Ciências, Dept. Informática, Bloco C5, Piso 1, 1700 Lisboa – PORTUGAL
[**] Instituto Superior Técnico, Dept. Matemática, Av. Rovisco Pais, 1049-001 Lisboa – PORTUGAL
[***] Escola Politécnica, L.A.C., Av. Prof. Luciano Gualberto, 158, Trav. 3, 05508-900, São Paulo – BRASIL

Abstract. In a recent short paper and a report (see [Neto *et al.* 98] and [Neto and Costa 99]) it was shown that programming languages can be translated efficiently on recurrent (analog, rational weighted) neural nets, using bounded resources. This fact was achieved by creating a neural programming language called NETDEF, such that each program corresponds to a modular neural net that computes it.

This framework has some practical implications in recent efforts to merge symbolic and sub-symbolic computation. Adding neuron-synapse connections (high-order neurons) to the neural network model allows us to integrate learning into the NETDEF computing paradigm.

Some possible enhancements are presented using this framework, namely structure self-modification, and integration of sub-symbolic learning into the NETDEF neuron architecture. The Hebb learning rule is used to provide an illustrative example.

Keywords. Neural Nets, Symbolic Computation, Sub-symbolic Computation, Neural Software.

1. Introduction

In the last decade, the significance of systems which integrate symbolic and sub-symbolic computing techniques has been consolidated (see [Wilson and Hendler 93] for an analysis of some particular hybrid systems). Motivation for this structural hybridization can be found both in biology (persons are able to process high level concepts supported by the neural biochemistry of the brain) and in engineering (intelligent control design tends to incorporate symbolic and sub-symbolic processing, in order to achieve better performances).

There are several different ways to accomplish this hybridization. Some models separate completely the two computation methodologies, using the output of the sub-symbolic structure as an input to the classical AI control schemata (cf. [Hendler and Dickens 91]). Others use symbolic and sub-symbolic information in the same data structure (as we do in this paper), blurring the distinctions among them, like in [Lange *et al.* 89]. Others still encapsulate subsystems of both kinds, and interface them through supervisors that control and hide different computing demands and resources of the subsystems (cf. [Wilson and Hendler 93]).

This paper presents a method to merge symbolic and sub-symbolic computation into a single neural network architecture.

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

Secondly, using second-order neurons, corresponding to special constructs called neuron-synapse connections, we show how to add learning processes to NETDEF. The end result is a neural net partially *hard*-wired and partially *soft*-wired by a suitable learning algorithm. This new language is NETDEF+. Since NETDEF+ is modular we have, after compilation, modules performing programming tasks and modules supporting sub-symbolic tasks.

2. NETDEF

Computability analysis of the analog neural net model is due to Hava Siegelmann and Eduardo Sontag. They used a quite simple model to establish lower bounds on the computational power of analog recurrent neural nets (see [Siegelmann 99] for details). These systems satisfy the classical constraints of computation theory, namely, (a) input is discrete and finite, (b) output is discrete, and (c) the system is itself finite (control is finite).

The functions computable by such a model depend on the type of the weights. With integer type, the neural

network has the power of finite automata, like the McCulloch and Pitts neural net model (see [Minsky 67] for details). Rational weights give Turing power to neural nets and with real weights we can compute non recursive functions (see [Siegelmann 99] for a systematic approach).

The analog recurrent neural net 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:

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

The dynamic system becomes,

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

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.

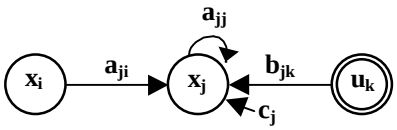

Figure 1. Graphical notation for neurons, input channels and their interconnections.

NETDEF is a high level parallel programming language able to describe arbitrarily complex algorithms. It 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 (see Figure 2 for a recursive and modular construction of a process), it supports several data types, variable and function declarations, and several other processes. It uses a modular synchronization mechanism based on handshaking for process ordering (the IN/OUT interface in Figure 2). A detailed description may be found in [Neto and Costa 99] 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:

$$\alpha(x) = (x + a)/2a \qquad (1)$$

This coding is a one to one mapping of [-a, a] into the working set [0, 1].

Input channels u_i are the interface between the system and the environment. They act has 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).

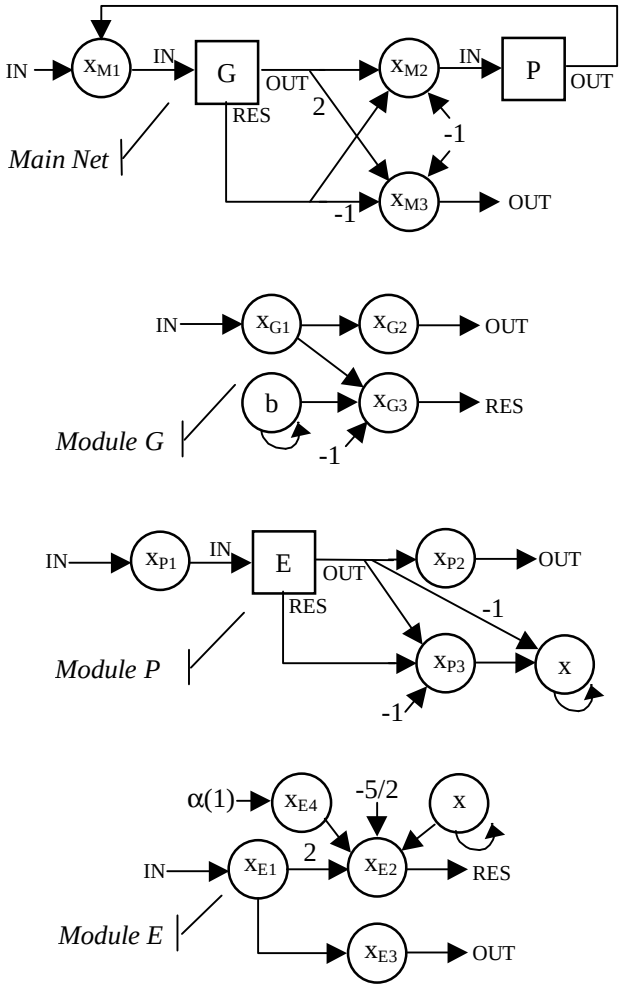

Figure 2. Process construction for WHILE b DO x := x+1.

In Figure 2, synapse IN sends value 1 (by some neuron x_{IN}) into x_{M1} neuron, thus 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 result is synchronized with an output of 1 through synapse OUT. The next two neurons decide between stopping the process ('b' is false) or executing module P ('b' is true), iterating again. The dynamic system is described by the following equations:

$$x_{M1}(t+1) = \sigma(x_{IN}(t) + x_{P2}(t))$$
$$x_{M2}(t+1) = \sigma(x_{G2}(t) + x_{G3}(t) - 1.0)$$
$$x_{M3}(t+1) = \sigma(2.x_{G2}(t) - x_{G3}(t) - 1.0)$$

Module G just 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 module is defined by:

$$x_{G1}(t+1) = \sigma(x_{M1}(t))$$
$$x_{G2}(t+1) = \sigma(x_{G1}(t))$$
$$x_{G3}(t+1) = \sigma(x_{G1}(t) + b(t) - 1.0)$$

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.

$$x_{P1}(t+1) = \sigma(x_{M2}(t))$$
$$x_{P2}(t+1) = \sigma(x_{E3}(t))$$
$$x_{P3}(t+1) = \sigma(x_{E2}(t) + x_{E3}(t) - 1.0)$$

In module E the increment of 'x' is computed (using $\alpha(1)$ for the code of real 1). The extra –1/2 bias of neuron x_{E2} is necessary due to the internal coding:

$$x_{E1}(t+1) = \sigma(x_{P1}(t))$$
$$x_{E2}(t+1) = \sigma(2.x_{E1}(t) - x_{E4}(t) + x(t) - 5/2)$$
$$x_{E3}(t+1) = \sigma(x_{E1}(t))$$
$$x_{E4}(t+1) = \sigma(\alpha(1))$$

The dynamics of neuron x is given by:

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

However, if neuron x is used in other modules, the compiler will add more synaptic links to its equation.

The compiler takes a NETDEF program and translates it into a text description defining the neural network (see Figure 3). 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.

This neural network is homogenous (all neurons have the same activation function) and the system is composed only by first-order neurons. The final 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.

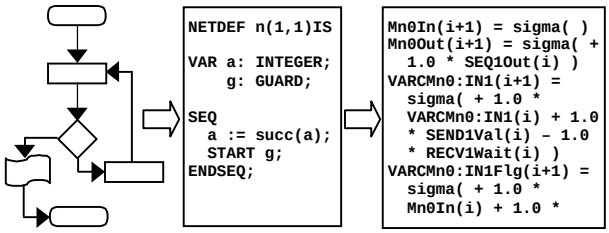

Figure 3. Obtaining a NETDEF neural network.
(a) The schematics of an algorithm, (b) A NETDEF program,
(c) The neural network description.
The NETDEF compiler automates step (b) to (c).

There are some related works in the literature about symbolic neural computation. The JaNNeT system (see [Gruau 95] for details), introduces a dialect of Pascal with some parallel constructs. This algorithmic description is translated, using several automated steps (first on a tree-like data structure and then on a low-level code, named *cellular code*), to produce a non-homogenous neural network (there are four different neuron types) able to perform the required computations.

Other difference to NETDEF is the network dynamics. In our model, at each instant, all neurons are updated with their new values. In JaNNeT, every neuron is activated only when all its synapses had transferred their values. Since this may not occur at the same instant, the global dynamics is not synchronous. A special attention is given to design automation of the final neural network architecture.

Another neural language project is NIL (outlined in [Siegelmann 93] and [Siegelmann 96]). The NIL system is also able to perform symbolic computations by using certain sets of constructions that are compiled into an appropriate neural net (using the same homogeneous neural architecture of NETDEF). It has a complex set of data types, from boolean and scalar types, to lists, stacks or sets that are kept inside a single neuron, using fractal coding.

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.

A proposed goal, but just delineated in [Siegelmann 96], was to provide mechanisms for tuning the compiled network, in order to generalize the initial processed information. However, to our knowledge, NIL was mainly used as a tool to derive specific theoretical results about neurocomputation, and was not fully developed into a network compiler application.

3. Extending the model

Our goal now is to integrate learning and hard-wiring mechanisms into the computation tools already developed, merging two standard computation methodologies (symbolic and sub-symbolic) in one single neural architecture. A further extension of the current conceptual schema, called NETDEF+, was devised to 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* – in NETDEF we had a specific concern about modularity; modularity must be preserved in the extension.

To accomplish these requirements, we decided to include neuron-synaptic connections into the neural network model. Although we are not concerned with biological plausibility, the existence of neuron-synaptic connections in the brain is known to exist in the complex dendritic trees of genuine neural nets (see [Shepherd 94] for details). In the model their main task is to convey values and use them to update and change other synaptic weights.

In this new high-order model, each neuron can compute a rational polynomial of its inputs, i.e., $\mathbf{x}(t+1) = \phi(\mathbf{x}(t), \mathbf{u}(t))$, with initial state $\mathbf{x}(0) = \mathbf{x}_0$, where the application map ϕ is the composition of a polynomial with rational coefficients with the piecewise sigmoid σ. The new dynamic system becomes,

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

Figure 4 displays a 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]:

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

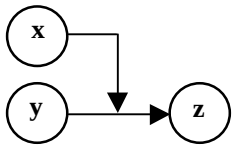

Figure 4. Graphical notation for neuron-synapse connection in equation (2).

High-order nets (i.e., networks having neurons with high-order transfer functions) have been used before. Most of the previous work on high-order neural nets study the superior computational power achieved by these types of transfer function, that can possibly multiply activations, and not only take a linear function of them.

Pollack, in [Pollack 87], built a finite high-order neural network model with universal properties. [Sun *et al.* 91] shows another Turing computational equivalence of second-order nets. [Goudreau *et al.* 94] deals with networks with the Heaviside (step) activation function. They show that single layer second-order nets are strictly more powerful than single layer first-order networks. An application to learn finite state automata using second-order neural nets is described in [Giles *et al.* 92].

Is it worthwhile to include high-order equations? There is no increase of computational power, since NETDEF is already Turing equivalent (with bounded resources). However, there are some pertinent advantages that justify the extension. The next sections will illustrate these benefits.

4. Dynamic Structures

Features associated with neuron-synapse connections (of second-order neurons) are deletion and insertion of

[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 (1). To avoid ambiguities, the first argument refers to the neuron-synapse connection, and the second, to the input neuron.

connections in execution time. If a synapse receives a zero weight, it can be seen as being removed from the architecture. Likewise, a zero weight synapse receiving a non-zero value will be added to the net. This feature implies a self-modifiable neural network. In order to allow NETDEF+ to perform deletions and insertions, we introduce a new process named LINK. A LINK process changes the synaptic weight of a pair of neurons. Its syntax is,

LINK (<*input-neuron*>,
<*new-weight-value*>,
<*output-neuron*>)

In Figure 5, neuron w_{yx} keeps the synaptic weight until an input signal comes through channel IN. At that moment, the neuron w_{yx} resets its activation to zero. At the same time the incoming signal through IN cancels the bias of the left data neuron, and the new value 'w' is inserted in the next step of computation as the new activation of neuron w_{yx}. The synaptic weight is changed at runtime. The diagram of the compiled net is given in Figure 5,

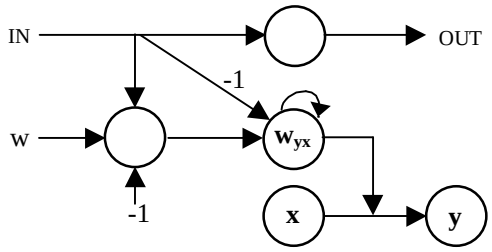

Figure 5. Neural net schema for LINK (x, w, y) process.

By using a LINK(x,0,y) process, the synapse between neurons x and y is deleted. Inputting a non-zero value reinserts the synapse. Deleting all inputs/outputs of a neuron separates it from the remaining network. This feature adds to NETDEF+ an architectural self-modifying mechanism.

5. Learning

A system that undergoes changes stimulated by the environment is capable of learning. A learning algorithm is a well-defined procedure that specifies how the system changes according to outside information. In this way, learning can be seen as the execution in run time of some hard-wiring algorithm, running under the inputs from the environment.

Many neural learning algorithms, like backpropagation, receive inputs and adapt the synaptic weights, pulling the network wiring towards the solution of the problem. Each algorithm uses some appropriate procedures to update the network weights, inspired by means of pure mathematical reasoning (e.g., the Least Mean Squares rule) or by biological inspiration (e.g., the Hebb rule).

The NETDEF+ model makes possible the change of synaptic weights at runtime, so that it should also be conceivable to implement some kind of learning algorithm with it. Starting with NETDEF, an already available neural programming language, how can this new feature be used to implement the most general learning algorithms?

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 they define how the entire module should change in order to respond in a new way to the environment (see [Haykin 99] for a description of several learning algorithms). 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. The integration of the learning process in NETDEF+ implies combining symbolic and sub-symbolic computations using modular high-order (second-order) neural nets.

6. The Hebb Rule revisited

The Hebb rule, one of the first learning rules presented in the literature, had a biological inspiration. Succinctly, it says that if two connected neurons are simultaneously active, their synaptic interconnection should be selectively strengthened.

In this section, we discuss Hebb's learning rule built on top of NETDEF+. We choose the classical problem of learning binary Boolean operators. The input vector **x** has two dimensions and the output vector **y** has one dimension.

Hebb rule states that for each sample $<x_1, x_2; y>$, synapse w_{yx_i} should be strengthened if and only if $x_i = y$. The sample values are bipolar (using the α codes for –1 and +1), introducing a compensation mechanism to avoid synapse saturation presented in the original Hebb rule. The common interpretation in the literature, is that the synaptic connection is updated by $\Delta w_{yx_i} =$

$\eta.x_i.y$ (the learning coefficient η is taken to be 1 for simplicity).

As said before, the Hebb module is divided into two parts, the *learning net* and the *control net*.

Learning net. The learning structure with two layers (input and output) is outlined in Figure 6. Each synapse has a neuron that keeps its current weight, in order to make synaptic changes simple and straightforward. There is also a neuron keeping the current bias. In general, the learning net reflects the topology of the network data structure needed to implement the learning algorithm.

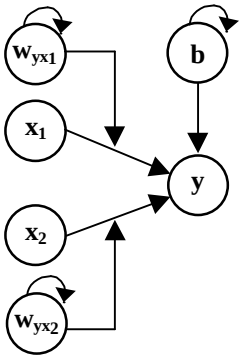

Figure 6. The learning net.

Control net. In this net we find the control mechanism performing learning and classification tasks over the knowledge kept in the learning net. This net is divided into the following components:

- The data structure. In this example, it consists of input (the **x** vector), and desired output (the **y** vector) that needs to be kept. This is done using a specific neural configuration, named L-array. An L-array is a set of neurons, one for each vector component, as in Figure 7 (even if it not strictly necessary to compute the Hebb rule, it will be needed to execute iterative learning algorithms).

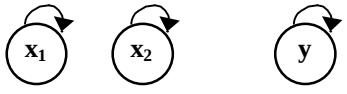

Figure 7. Two L-arrays.

- The synaptic updating structure, where the synaptic change formula is evaluated accordingly. For the Hebb learning algorithm, the actual formula is $w_{yx_i}(t+1) = w_{yx_i}(t) + x_i(t).y(t)$ (the bias update formula is simpler, $b(t+1) = b(t) + y(t)$, the corresponding net is not displayed). The subnet which performs this computation is automatically built by the compiler, and it is showed in Figure 8 (the actual compiled subnet is more complex, but we have simplified it for the clarity of exposition).

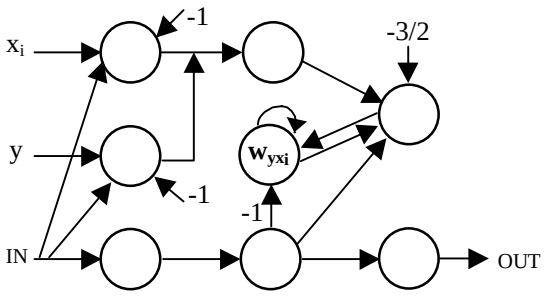

Figure 8. Updating synapse w_{yx_i}.

- The interface structure, where all communications with the remaining network are made. To achieve maximum transparency between symbolic and sub-symbolic computations, each learning module is seen from the outside as a function. These *learning functions* can be used as any other function when programming with NETDEF, except that they need training before being called. This interface structure is autonomous relative to learning, so learning processes and classification processes are independent and can, whenever needed, run in parallel. We will not present it due to space limitations, since its configuration is complex.

7. Conclusion and Future Work

This paper explores a new neural network model and uses it to extend NETDEF, a high-level programming language to hard-wire piecewise linear neural nets. The extension, a neuron-synaptic connection model (second-order neurons), is applied to define new process types.

We present three new tools. Direct real multiplication between two inputs is easily hard-wired, applying a neuron-synaptic link within two neurons. It is also possible to handle self-modifying mechanisms within neuron-synaptic neural nets, allowing dynamic architectures. Most relevant is that the model supports learning. With this feature, sub-symbolic and symbolic computations are linked together in the same neural framework.

The next step is to present an appropriate process set, in order to implement a general setting for learning algorithms. These processes should interact consistently with NETDEF control structures.

8. References

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.

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.

Gruau, F., Ratajszczak, J., and Wiber, G., *A neural compiler*, **Theoretical Computer Science**, [141] (1-2), 1995, 1-52.

Haykin, S., **Neural Networks – A Comprehensive Foundation**, 2nd ed., Prentice Hall, 1999.

Hendler, J. and Dickens, L., *Integrating Neural and Expert Reasoning: An Example*, In **Proceedings of AISB-91**, Leeds, 1991. Cited in Wilson, 93.

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.

Minsky, M., **Computation: Finite and Infinite Machines**, Prentice Hall, 1967.

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.

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*.

Pollack, J., **On Connectionism Models of Natural Language Processing**, Ph.D. thesis, Computer Science Dept., Univ. of Illinois, Urbana, 1987.

Shepherd, G. M., **Neurobiology**, 3rd ed., Oxford University Press, 1994.

Siegelmann, H., **Foundations of Recurrent Neural Networks**, *Technical Report DCS-TR-306*, Department of Computer Science, Laboratory for Computer Science Research, The State University of New Jersey Rutgers, October 1993.

Siegelmann, H., *On NIL: The Software Constructor of Neural Networks*, **Parallel Processing Letters** [6] 4, World Scientific Publishing Company, 1996, 575-582.

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

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.

Wilson, A. and Hendler, J., *Linking Symbolic and Subsymbolic Computing*, Technical Report, Dept. of Computer Science, University of Maryland, 1993.