Neural Programming Language

Hava T. Siegelmann
Department of Computer Science
Bar-Ilan University, Ramat-Gan 52900, Israel
E-mail: hava@bimacs.cs.biu.ac.il

Abstract

Analog recurrent neural networks have attracted much attention lately as powerful tools of automatic learning. We formally define a high level language, called NEural Langage, which is rich enough to express any computer algorithm or rule-based system. We show how to compile a NEL program to a network which computes exactly as the original program and requires the same computation time. We suggest this language along with its compiler as the ultimate bridge from symbolic to analog computation, and propose its outcome as an initial network for learning. *

1 Introduction

Classical approaches of Computer Science and Artificial Intelligence are based on understanding and explaining key phenomena in a discrete, symbolic manner. A list of rules or an algorithm is then developed and given to the computer to execute. These approaches have the limitations of human understanding and analysis power.

An alternative approach to elicit knowledge and express it by symbols is the neural network modeling. Neural networks are trainable dynamical systems which learn by observing a training set of input-output pairs. They estimate functions without a mathematical model of which the output is assumed to depend on the input.

The choice between the above two approaches depends on the particular application. For example, the first approach is a better fit for controlling simple engines; in the complex-task of generating a functional electric stimulation (FES) in locomotion of subjects with incomplete spinal cord injury, the classical methodology yields limited functionality, while the learning approach generates far better results (Armstrong *et al.* 1993). The pure neural network approach, although rich and adaptable, may lose simple hints that are easily tractable by a human expert (and thus are provided in "hand-crafted rules") but are hard to deduce from a sample set. In the functional electric stimulator, for example, a list of rules is thus preferred over adaptable approaches for particular simple functions, while the adaptable approach is still generally preferred. Another drawback of the nets is that they suffer from sensitivity of the convergence rate to initial state.

The two approaches, symbolic algorithms and adaptive analog nets, are suggested in this work to be interleaved in a manner that takes the best of both models. This is based on very recent theoretical findings in the area of artificial neural networks: that the computational power of such nets is universal (Siegelmann & Sontag 1991). We provide a novel method for translating algorithms (or rules) expressed in a Pascal-like programming language into a corresponding neural networks. This research can be thought of a basis for acquiring a function estimator in any area of expertise, using the following four step paradigm.

1. Knowledge will be elicited from experts in the field, and a program (or a rule-based system) will be written in a high level language. In our example of functional electric stimulator, the physician deduces rules from patterns recorded in able subjects as a first approximation for locomotion of paralyzes.
2. The program will be compiled into an equivalent neural net of analog neurons.
3. The applicability of the network for adapting and generalizing will be raised: nodes and edges of low computational significance will be added to yield a homogeneous architecture; the net may be pruned, and the activation function smoothed up.
4. The network will be provided with a sample data of input-output pairs, and will adapt itself to comply with them, thus, tuning and fixing the original expert's knowledge.

The fields of knowledge engineering and programming will cover the first step. Learning algorithms of recurrent neural nets will cover the

*This research was partially supported by US Air Force Grant AFOSR-91-0343

fourth step. For various learning methods see (Hertz, Krogh, & Palmer 1991). Our contribution is the methodology for the second step; the one that translates a computer program into a recurrent network. Our requirement of the translation are strict. The translation should be fast, and the network should simulate the program without slowing down the computation. Note that the network itself should consist of analog neurons only and do not allow for any threshold (or other discontinuous) neuron, so that to better fit methods of adaptation and learning. The third step is recommended for a practical reason: when a network adapts to perform very accurately according to a training set, it tends to be "overfitting", that is, to loose the capability of generalizing well on new data. Because the network which is built at the second stage imitates a particular algorithm, its architecture and parameter values overfit the algorithm and may have the overfitting problem. Nodes and edges are added to prevent this problem. At this stage, other considerations may be taken, e.g., achieving the effective number of parameters (Moody 1992) by pruning the network.

Our approach of building an initial network from expert knowledge can be desirable when a particular behavior is mandatory, e.g. safety conditions or security policy. In this case, the mandatory behavior is coded into the network prior to learning, and we force parts of the network to remain fixed during the adaptation process, see for example the distal learning paradigm (Jordan 1992).

1.1 The Network Model

We focus on recurrent neural networks which consist of a finite number of neurons. In these networks, each processor's state is updated by an equation of the type

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

where x_i are the processors ($i = 1, \ldots, N$), u are the external input, a, b, c are constants, N is the number of processors, and M is the number of external input signals. The function σ is the simplest possible "sigmoid," namely the saturated-linear function:

$$\sigma(x) := \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } 0 \le x \le 1 \\ 1 & \text{if } x > 1\,. \end{cases} \quad (2)$$

When learning is desirable, the σ will be substituted (during the third step) by a fully differentiable sigmoidal function. (Note that the precision of the neurons is not limited; thus our model describes an analog rather than a digital machine.)

As part of the description, we assume that we have singled out a subset of the N processors, say $x_{i_1}, \ldots, x_{i_l}$; these are the l *output processors*, and they are used to communicate the outputs of the network to the environment. Thus a net is specified by the data (a_{ij}, b_{ij}, c_i) together with a subset of its nodes.

1.2 The Computational Power

Some efforts have been directed towards practical implementations of the applications, including those in the areas of pattern and speech recognitions, robot control, time series prediction, and more—see (Hertz, Krogh, & Palmer 1991). Only recently, rigorous foundations to the recurrent neural network model were developed, see (Siegelmann & Sontag 1991; 1994b; Balcázar *et al.* 1993; Siegelmann & Sontag 1994a). (See (Kilian & Siegelmann 1993) for a related model with different activation functions.) The computational power of the recurrent network (with finite number N of neurons and analog activations values) depends on the type of numbers utilized as weights (i.e. the constants a, b, c.)

1. If the weights are integers, the neurons may assume binary activation values only. Thus, the network computes a regular language.
2. If the weights are rational numbers, the network is *equivalent in power to a Turing Machine* (Siegelmann & Sontag 1991). In particular, given any function ϕ computed by a Turing Machine M in time T, one can construct a network $\mathcal{N}$ that computes the function ϕ in exactly time T. That is, there is no slow down in the computation (Siegelmann & Sontag 1994b). Furthermore, the size of the network is independent of the computation time T. A corollary is the existence of a universal network consisting of 886 neurons and simple rational weights that computes all recursive functions.
3. When weights are general real numbers (specifiable with unbounded precision), the network turns out to *have super-Turing capabilities.* However, it is sensitive to resource constrains and thus is not a tautology. The exact characterization of the computational class associated with such networks is disclosed in (Siegelmann & Sontag 1994a)..

1.3 Previous Related Work

Previous work in inserting apriori knowledge to nets was shown to make the process of training faster for both feedforward, e.g. (Abu-Mostafa 1990; Al-Mashouq & Reed 1991; Berenji 1991; Giles & Maxwell 1987; Perantonis & Lisboa 1992; Pratt 1992; Suddarth & Holden 1991; Towell, Craven, & Shavlik 1990), and recurrent networks, e.g. (Frasconi *et al.* 1993; Omlin & Giles 1992). In all cases

studied, the rules were very simple, that is, only regular rules of simple finite automata. We, on the other hand, insert rules that stem at any computer algorithm and not finite state automata only.

Some work dealt with inserting rules with the emphasize of correcting them, e.g.. (Fu 1989; Ginsberg 1988; Omlin & Giles 1993; Oursten & Mooney 1990; Pazzani 1989). The paper (Towell, Shavlik, & Noordewier 1990) faced an expert system based on propositional calculus, and suggested to transform the original propositional domain theory into a network. The connection weights were elegantly adjusted in accordance with the observed examples using standard backpropagation techniques.

We provide a general technique to translate first order logic (not only propositional) or any general algorithm (not only finite automata) to recurrent nets, rather than feedforward-acyclic architectures, which computationally are very limited (i.e. the computation ends in constant number of steps). We, however, do not provide yet an algorithm for tuning and correcting the encoded rules. This task is one of the future directions of our work.

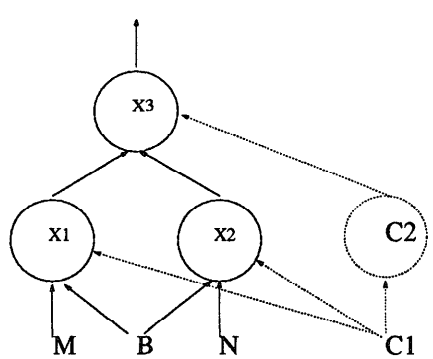
1.4 Programming Networks

Given an algorithm, how does one construct a network that executes it? We demonstrate such a construction by an example.

Example 1.1 Let M and N be values in $[0, 1]$ and let B be a Boolean expression. The conditional statement

$$\begin{array}{l}\textbf{If } (B) \textbf{ then } x = M \\ \quad\textbf{else } x = N\end{array}$$

can be executed by the following network:



$$\begin{aligned} x_1(t) &= \sigma(M + B - 1) \\ x_2(t) &= \sigma(N - B) \\ x_3(t+1) &= \sigma(x_1(t) + x_2(t)) \,. \end{aligned}$$

The neuron x_1 attains the value $\sigma(M)$ when $B = 1$. As σ is the linear-saturated function of Equation 2, and M is assumed to lie in the range $[0, 1]$, $x_1(t) = \sigma(M) = M$. When $B = 0$, $x_1(t) = \sigma(M - 1) = 0$. The neuron x_2 computes $\sigma(N - 1) = 0$ for $B = 1$, and $\sigma(N) = N$ for $B = 0$. Summing the above two values into x_3 results in

$$\begin{aligned} \sigma(M + 0) &= M \quad \text{for } B = 1\,, \\ \sigma(0 + N) &= N \quad \text{for } B = 0 \end{aligned}$$

as desired.

To synchronize the update, an "If" statement requires two sub-statement counters: one for the first update level, c_1, and one for the second update, c_2. The full update for the "if statement" is thus:

$$\begin{aligned} x_1^+ &= \sigma(M + B + c_1 - 2) \\ x_2^+ &= \sigma(N - B + c_1 - 1) \\ x_3^+ &= \sigma(x_1 + x_2 + c_c - 1)\,, \end{aligned}$$

The update equations of the counters are excluded. □

In general, tasks may be composed of a large number of interrelated subtasks. The entire task may thus be highly complex, and designing an appropriate network from scratch becomes infeasible. We introduce a high level language (**NE**ural **L**angage) for automatic construction of recurrent nets. One could compare the relationship between coding networks directly and writing in NEL with the relationship between coding in a machine language and programming in a high level language.

1.5 The Organization of The Paper

The rest of this paper is organized into three sections: In Section 2, we provide a brief syntactic description of the language; in Section 3 we show how to compile a subset of NEL into a network; and in Section 4 we conclude the NEL compiler.

2 Syntax Of NEL

NEL is a procedural, parallel language. It allows for the subprograms procedure and function. A sequence of commands may either be executed sequentially (*Begin, End*) or in parallel (*Parbegin, Parend*). There is a wide range of possible **data types** for constants and variables in NEL, including the simple types: Boolean, character, scalar type, integer, real, and counter (i.e., an unbounded natural number or 0); and the compound types: lists (with the operations defined in LISP), stacks, sets. records and arrays. For each data type, there are a few associated predefined functions, e.g. *Isempty*(stack), *In*(element, set), and *Iszero*(counter).

The language is *strongly typed* in the sense that applying a function that is defined on a particular data type to a different data type may yield an error.

Expressions are defined on the different data types. Examples of expressions are:

1. $\sum_{i=1}^{7} c_i x_i$ for constants c and either real or integer values of the variables x_i.
2. (B_1 And B_2) Or ($x > \frac{1}{2}$) for Boolean values B_1, B_2 and an integer value x.
3. **Pred** and **Succ** of an element e of a finite ordered type T returns another element of the same type.
4. **Chr** operates on an integer argument and returns a character.

Statements of NEL include atomic statements (e.g., assignments, procedure calls, I/O statements), sequential compound statements (*Begin, End*), parallel compound statements (*Parbegin, Parend*), flow control statements which include both conditional (e.g., *If-then, If-then-else, case,* and *cond*) and repetition statements (such as *while* and *repeat*). Full syntax of NEL is provided in (Siegelmann 1993).

3 Compiling NEL

We next overview the compiler which translates programs written in the language NEL into neural networks. A network operates generally in the following manner: there are N neurons; at each tick of the clock, all neurons are updated with new values. Thus, a network step consists of a parallel execution of N assignments.

When simulating the program on a network, some of its neurons represent variables, some represent the program commands, and, practically, about half of the neurons in the network constitute the program counters. More specifically, each statement is associated with a special neuron, called the "statement counter" neuron. These neurons take Boolean (i.e., binary) values only. When a statement counter neuron is True, the statement is executed. Note that several statement counters may assume the value True simultaneously. Full discussion on controlling the counters is provided in (Siegelmann 1993).

Here, we describe the compilation of a small subset of NEL statements into a network. The four most basic commands are the parallel block, the serial block, the conditional if statement, and the goto statement. Other flow control statements — such as Case, Cond, While, Repeat, and Dolist— can be viewed as a combination of the above four. We, thus, overview how to compile the four building blocks:

1. A parallel block consists of the commands enclosed by **ParBegin** and **Parend**. Each of these commands is associated with a statement counter. All these counters are set simultaneously upon reaching the Parbegin. A concluding mechanism is required for synchronization. This mechanism keeps track of the termination of the various commands, and announces finishing upon termination of them all. Only then, the parallel block is concluded with the Parend. Details are provided in (Siegelmann 1993).
2. A serial block consists of the commands between the **Begin** and **End**. This involves an extensive use of counters.
3. The compilation of a simple if statement was provided in Example 1.1. We compile a general if statement

   If (B) **then** stat1
   else stat2

   by

   Parbegin
   If (B) **then** pc-stat$_1$ = 1 ;
   If ($\neg(B)$) **then** pc-stat$_2$ = 1
   Parend
4. A Goto statement is implemented simply by a change in the neurons simulating the statement counters of the program.

Next, we consider a subset of the data types. Each variable, except for records and arrays, is represented via one neuron in the network.

- Boolean values are represented via the numbers $\{0, 1\}$. The logical operations are:

Operation	Network's emulation
Not(x)	$\sigma(1 - x)$
Or(x_1, x_2)	$\sigma(x_1 + x_2)$
And(x_1, x_2)	$\sigma(x_1 + x_2 - 1)$

(3)

  Relational operations are defined in a straightforward manner: $x > y$ is $\sigma(x - y)$, $x \geq y$ is $\sigma(x - y + 1)$, and $x \neq y$ is $x_1(t) = \sigma(x - y)$, $x_2(t) = \sigma(y - x)$ and $x_3(t + 1) = \sigma(x_1 + x_2)$.
- List of T. Assume, for simplicity, that T has only two elements T=$\{0, 1\}$; later we generalize T to arbitrary cardinality. Given a list of elements $\omega_1\omega_2\cdots$, we regard it as a string $\omega = \omega_1\omega_2\cdots\omega_n$. We wish to represent this string as a number in the range $[0, 1]$, so that to be held in a neuron. If we were to represent the string as a number $\sum_{i=1}^{n} \frac{\omega_i}{2^i}$, one would not be able to differentiate between the string "β" and "$\beta \cdot 0$", where '$\cdot$' denotes the concatenation operator. Worse than that, the continuity of the activation function σ makes it impossible to retrieve the most significant bit (in radix 2) of a list in a constant amount of time. (For example, the values .100000000000 and .011111111111111 are almost indistinguishable by a net.) We encode the list by

$$\sum_{i=1}^{n} \frac{2\omega_i + 1}{4^i} .$$

(For examples, the list $\omega = 1011$ is encoded by the number $q = .3133_4$.) This number ranges in $[0, 1)$, but not every value in $[0, 1)$ appears. If the list started with the value 1, then the associated number has a value of at least $\frac{3}{4}$, and if it started with 0, the value is in the range $[\frac{1}{4}, \frac{1}{2})$. The empty list is encoded into the value 0. The next element in the list restricts the possible value further.

The set of possible values is not continuous and has "holes". Such a set of values "with holes" is a Cantor set. Its self-similar structure means that bit shifts preserve the "holes." The advantage of this approach is that there is never a need to distinguish among two very close numbers in order to read the most significant digit in the base-4 representation. We next demonstrate the usefulness of our encoding of the binary lists.

1. **CAR(ω), Reading the First Element:** The value of q is at least $\frac{3}{4}$ when the Car of the list is 1, and at most $\frac{1}{2}$ otherwise. The linear operation $4q - 2$ transfers q to at least 1 when the Car element is 1, and to a non-positive value otherwise. Thus, the function Car $(q) = \sigma(4q - 2)$ provides the value of the Car element.
2. **CDR(ω), Removing the Left Element:** Cdr a list, transfers the list $\omega = 1011$ to 011, or the encoding from $q = .3133_4$ to $.133_4$. When the Car element is known, the operation Cdr $(q) = 4q - (2 \text{ Car } (q) + 1)$ (or equivalently $\sigma(4q - (2 \text{ Car } (q) + 1))$) has the effect of CDRing the list.
3. **CONS(e, ω) Pushing a New Element to the Left of ω:** Pushing 0 to the left of the list $\omega = 1011$ changes the value into $\omega = 01011$. In terms of the encoding, $q = .3133_4$ is transferred into $q = .13133_4$. That is, the suffix remains the same and the new element $e \in \{0, 1\}$ is entered into the most significant location. This is easily done by the operation $\frac{q}{4} + \frac{2e+1}{4}$ (which is equivalent to $\sigma(\frac{q}{4} + \frac{2e+1}{4})$ given that $q \in [0, 1)$.)
4. **IsNull(ω):** The predicate IsNull indicates whether the list w is empty or not, which means in terms of the encoding, whether $q = 0$ or $q \geq .1_4$. This can be decided by the operation IsNull $(q) = \sigma(4q)$.

Assume T has a general cardinality, n. The operations **Car**(ω), **Cdr**(ω), **Cons**(e, ω) , and the predicate **IsNull**(ω) are implemented by: $\sigma(\frac{1}{2n} + \frac{1}{n}(\sigma(2nq - 2) + \sigma(2nq - 4) + \cdots + \sigma(2nq - (2n - 2))))$, $\sigma((2nq - 1 - 2(\sigma(2nq - 2) + \sigma(2nq - 4) + \cdots + \sigma(2nq - (2n - 2)))))$, $\sigma(\frac{q}{2n} + e)$, and $\sigma(1 - 2nq)$, respectively.

- Stacks are represented similarly to lists. Here Top substitutes Car, Pop substitutes Cdr, Push substitutes Cons, and the predicate Empty substitutes IsNull.
- Scalars are implemented using the same idea of gaps as with lists. Assume a scalar type with n elements $\{0, 1, \ldots (n - 1)\}$. The ith element is represented as scalar $(i, n) \equiv \frac{2i+1}{2n}$. Order operations are implemented as follows:

Operation	Network's emulation
Pred(x)	$\sigma(x - \frac{1}{n})$
Succ(x)	$\sigma(x + \frac{1}{n})$
Ord(x)	$\sigma(xn - \frac{1}{2})$

(4)

- A counter with the value n is represented as $(1 - 2^{-n})$, that is

$$\text{counter}(n) \hookrightarrow .\underbrace{11\ldots1}_{n} \tag{5}$$

The operations on counters **Inc**, **Dec**, and the predicate **IsZero** are implemented by $\sigma(\frac{1}{2}(x+1))$, $\sigma(2x - 1)$, and $\sigma(1 - 2x)$, respectively.

4 Conclusions

In conclusions, we can prove the next theorem.

Theorem 1 *There is a compiler that translates each program in the language NEL into a network. The constants (weights) that appear in the network are the same as those of the program, plus several rational small numbers. Furthermore, the size of the network is O(length) and its running time is O(execution measure). Here, length is the static length of the program, i.e. the number of tokens listed in the source code, and the execution measure is its dynamic length, i.e., the number of atomic commands executed for a given input.*

We may furthermore conclude from previous work described in subsection 1.2 and from the above theorem that all computer algorithms are expressible in NEL using rational constants only, while NEL programs that use real weights are stronger than any digital algorithm.

Acknowledgment

I wish to thank Eduardo Sontag and Jude Shavlik for useful comments.

References

Abu-Mostafa, Y. 1990. Learning from hints in neural networks. *Journal of Complexity* 6:192.

Al-Mashouq, K., and Reed, I. 1991. Including hints in training neural nets. *Neural Computation* 3(3):418–427.

Armstrong, W.; Stein, R.; Kostov, A.; Thomas, M.; Baudin, P.; Gervais, P.; and Popvic, D. 1993. Applications of adaptive logic networks and dynamics to study and control of human movement. In *Proc. Second Interntational Symposium on three-dimensional analysis of human movement*, 81–84.

Balcázar, J. L.; Gavaldà, R.; Siegelmann, H.; and Sontag, E. D. 1993. Some structural complexity aspects of neural computation. In *IEEE Structure in Complexity Theory Conference*, 253–265.

Berenji, H. R. 1991. Refinement of approximate reasoning-based controllers by reinforcement learning. In Birnbaum, L., and Collins, G., eds., *Machine Learning, Proceedings of the Eighth International International Workshop*, 475. San Mateo, CA: Morgan Kaufmann Publishers.

Frasconi, P.; Gori, M.; Maggini, M.; and Soda, G. 1993. Unified integration of explicit rules and learning by example in recurrent networks. *IEEE Transactions on Knowledge and Data Engineering.* Accepted for publication.

Fu, L. M. 1989. Integration of neural heauristics into knowledge-based inference. *Connection Science* 1:325–340.

Giles, C., and Maxwell, T. 1987. Learning, invariance, and generalization in high-order neural networks. *Applied Optics* 26(23):4972–4978.

Ginsberg, A. 1988. Theory revision via prior operationalization. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, 590.

Hertz, J.; Krogh, A.; and Palmer, R. 1991. *Introduction to the Theory of Neural Computation.* Redwood City: Addison-Wesley.

Jordan, M. I. 1992. Forward models: Supervised learning with a distal teacher. *Cognitive Science* 16:307–354.

Kilian, J., and Siegelmann, H. T. 1993. On the power of sigmoid neural networks. In *Proc. Sixth ACM Workshop on Computational Learning Theory.*

Moody, J. 1992. The effective number of parameters: An analysis of generalization and regularization in nonlinear learning systems. In *J.E. Moody, S.J. Hanson, and R.P. Lippmann, editors, Advances in Neural Information Processing Systems*, volume 4, 847–854. San Mateo, CA: Morgan Kaufmann.

Omlin, C., and Giles, C. 1992. Training second-order recurrent neural networks using hints. In Sleeman, D., and Edwards, P., eds., *Proceedings of the Ninth International Conference on Machine Learning*, 363–368. San Mateo, CA: Morgan Kaufmann Publishers.

Omlin, C., and Giles, C. 1993. Rule revision with recurrent neural networks. *IEEE Transactions on Knowledge and Data Engineering.* accepted for publication.

Oursten, D., and Mooney, R. 1990. Changing rules: A comprehensive approach to theory refinement. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 815.

Pazzani, M. 1989. Detecting and correcting errors of omission after explanation-based learning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 713.

Perantonis, S., and Lisboa, P. 1992. Translation, rotation, and scale invariant pattern recognition by higher-order neural networks and moment classifiers. *IEEE Transactions on Neural Networks* 3(2):241.

Pratt, L. 1992. Non-literal transfer of information among inductive learners. In Mammone, R., and Zeevi, Y., eds., *Neural Networks: Theory and Applications II.* Academic Press.

Siegelmann, H. T., and Sontag, E. D. 1991. Turing computability with neural nets. *Appl. Math. Lett.* 4(6):77–80.

Siegelmann, H. T., and Sontag, E. D. 1994a. Analog computation via neural networks. *Theoretical Computer Science.* to appear. A preliminary version in: The second Israel Symposium on Theory of Computing and Systems, Natanya, Israel, June, 1993.

Siegelmann, H. T., and Sontag, E. D. 1994b. On computational power of neural networks. *J. Comp. Syst. Sci.* previous version appeared in *Proc. Fifth ACM Workshop on Computational Learning Theory,* pages 440-449, Pittsburgh, July 1992.

Siegelmann, H. T. 1993. *Foundations of Recurrent Neural Networks.* Ph.D. Dissertation, Rutgers University.

Suddarth, S., and Holden, A. 1991. Symbolic neural systems and the use of hints for developing complex systems. *International Journal of Man-Machine Studies* 34:291–311.

Towell, G.; Craven, M.; and Shavlik, J. 1990. Constructive induction using knowledge-based neural networks. In Birnbaum, L., and Collins, G., eds., *Eighth International Machine Learning Workshop*, 213. San Mateo, CA: Morgan Kaufmann Publishers.

Towell, G.; Shavlik, J.; and Noordewier, M. 1990. Refinement of approximately correct domain theories by knowledge-based neural networks. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 861. San Mateo, CA: Morgan Kaufmann Publishers.