A Connectionist Approach to High-Level Cognitive Modeling

Rainer Goebel
Department of Psychology, Gutenbergstr. 18
3550 Marburg, West-Germany

Abstract—In this paper a connectionist framework is outlined which combines the advantages of symbolic and parallel distributed processing. With regard to the acquisition of cognitive skills of adult humans, symbolic computation is stronger related to the early stages of performance whereas parallel distributed processing is related to later, highly practiced, performance. In order to model skill acquisition, two interacting connectionist systems are developed. The first system is able to implement symbolic data structures: it reliably stores and retrieves distributed activity patterns. It also can be used to match in parallel one activity pattern to all other stored patterns. This leads to an efficient solution of the variable binding problem and to parallel rule matching. A disadvantage of this system is that it can only focus on a fixed amount of knowledge at each moment in time.
The second system – consisting of recurrent back-propagation networks – can be trained to process and to produce sequences of elements. After sufficient training with examples it possesses all advantages of parallel distributed processing, e. g., the direct application of knowledge without interpreting mechanisms. In contrast to the first system, these networks can learn to hold sequentially presented information of varying length simultaneously active in a highly distributed (superimposed) manner. In earlier systems – like the model of past-tense learning by Rumelhart and McClelland – such forms of encodings had to be done "by hand" with much human effort. These networks are also compared with the tensor product representation used by Smolensky.

1 Introduction

Parallel distributed processing seems especially well suited to model automatic, subconscious information processing. Symbolic computations for example variable binding, explicit rule following and deliberate planning are much harder to achieve.
I think that for modeling high-level cognitive processing both styles of computation are necessary. Considering the *temporal dimension* of learning a cognitive skill reveals that they are even intimately related: early in acquisition, symbolic processing is very important because instructions (facts and rules) must be stored and processed deliberately. The application of knowledge during this stage is controlled, slow, effortful and generally serial but it has the great advantage that approximate performance is achieved immediately through general *interpreting* procedures.
In later stages of practice the relevance of connectionist computation gradually increases: the explicit, declarative knowledge is converted to more flexible, more robust and richer procedural knowledge. The application of this knowledge is for the most part *automatic*, requires few attention, is fast and more parallel: it constitutes "expert knowledge". This kind of *directly* applied knowledge gradually emerges because subskills are automated: elementary processing steps are associated together and activated as whole units.

According to this view, I propose a connectionist framework which considers the acquisition of cognitive skills in terms of two interacting components (similar to Norman, 1986): one component – capable of symbol processing – trains or 'programs' the other component. The first component consists of 'symbol-modules' which efficiently implement essential aspects of conventional (von Neumann) computers. The second component consists of 'PDP-modules' – recurrent back-propagation networks – which possess all advantages of parallel distributed processing.

I will explain the basic features of the proposed system using a simple illustration task.

2 The Illustration Task

The proposed framework addresses skill acquisition of adults or at least older children who sufficiently master their native language. To explain the basic ideas of the approach, I use a very simple illustration task: 'Pig Latin' (Harvey, 1985). Children often learn to speak Pig Latin as a "secret" language. A word is translated into Pig Latin according to the following rule: take any consonants at the beginning (up to the first vowel) and move them to the end. Then add "ay" to the end. So CAT becomes ATCAY, TRUST becomes USTTRAY and IS becomes ISAY. To make things a little bit more difficult, I added the following rule: if the word begins and ends with a vowel then add "way" to the end. Thus EASY doesn't become EASYAY but EASYWAY which also sounds better. Note that Pig Latin is spoken, thus pronounciation is relevant for applying the rules. In the following however the rules are applied to strings of letters. Despite the simplicity of these rules, it requires some practice until one speak Pig Latin fluently!

After general remarks about the used networks the basic features of symbol-modules are explained. Then it is outlined how a symbol module can explicitly follow the Pig Latin rules. Thereafter PDP-modules are described and it is shown how a special module learns by example to behave as following the rules. Finally it is outlined how both modules are integrated in such a way that the symbol module trains the PDP-module to follow the rules.

3 Recurrent Networks: Handling Temporal Data

Cognitive processes evolve in time: sequences of elements (for example letters, phonemes or words) must be processed and produced as output. Therefore both modules consist of (multilayered) *recurrent* networks. Recurrent networks are well suited to handle temporal data because they provide a kind of short term memory for retaining activity states over time. The system operates in discrete time steps; the update scheme is as follows: layers are updated synchronously in ascending order (as in feedforward nets). If unit i is updated at time step t and there are recurrent connections (from the same or higher layers) to unit i, then the activity of the corresponding units at time step $t-1$ is used. Both modules use linear and semi-linear units.
PDP-modules are trained using the generalized delta rule. Back-propagation allows networks to find themselves interesting internal representations, but is designed for feedforward nets only. Rumelhart et. al. (1986) describe a technique which allows the application of back-propagation to arbitrary connected networks: Every recurrent network operating in (discrete) time events can be "*unfolded in time*" to a feedforward network with identical behavior. However this technique has some disadvantages. Therefore I made some modifications yielding a system with high efficiency (see Goebel, 1990)

4 Symbol-Modules: Explicit Rule Following

Symbol-modules are special architectures designed to store and retrieve distributed activation patterns (symbols) reliably and immediately.
A basic assumption, also valid for PDP-modules, is that the activity of some units can serve as target values for nearby output units. This allows one module to send teaching signals to other modules.
In its simplest form a symbol module is composed of two parts (see figure 1). One part — consisting of two equal width layers — acts as a 'gate' for symbols. Through this 'symbol gate' the module can receive symbols from other modules or from input units. It also can send symbols through this gate to other modules or to output units. At each time step one module can handle one symbol.
The other part consists of a (large) layer that acts as a sequence of 'memory cells'. These 'local memory units' are totally connected to the output units.

4.1 Location-Addressed Retrieval

Assume that a sequence of symbols, for example the string TIME is to be stored in the symbol-module of figure 1. Each (linear) local memory unit has a connection to its right neighbor with weight 1. In this example the number of units N_s that constitute a symbol is 5.

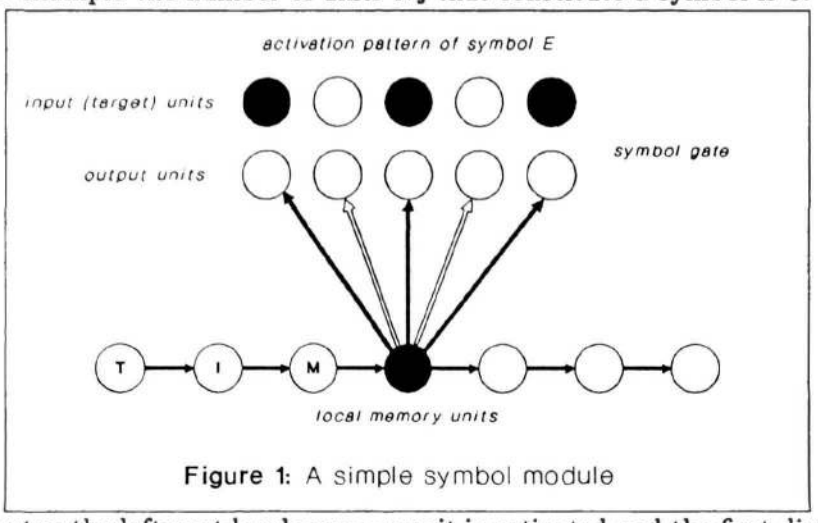

Figure 1: A simple symbol module

At the first time step the leftmost local memory unit is activated and the first distributed pattern – representing T – appears at the target units of the symbol gate. Now learning takes place according to the delta rule. The result is that the activation values of the target units are mapped to the weights coming from the active local memory unit to the output units.
At the next time step the second local memory unit is active and the second symbol (I) appears on the target units. Now these activation values are mapped to the corresponding weights etc.
Consider retrieving the sequence. Assume all local memory units are turned off. Now the leftmost local unit is turned on and activation flows through the learned weights to the output units evoking the first stored symbol T. At the next time step the next local unit is active evoking the second stored symbol I and so on. This is *location-addressed* retrieval.
The delta rule changes the weights as follows:

$$w_{ol}(t+1) = w_{ol}(t) + \epsilon a_l(a_t - a_o)$$

The value of learning rate ϵ is 1. (The index l stands for a local unit, o for an output unit and t for a target (input) unit.) Consider the weights between a non-active local unit and the output units:

$$w_{ol}(t+1) = w_{ol}(t) + 0(a_t - a_o) = w_{ol}(t)$$

This says that the weights between non-active local units and the output units are not affected by the learning process. For the weights between the active local unit and the output units results:

$$w_{ol}(t+1) = w_{ol}(t) + 1(a_t - a_o) = w_{ol}(t) + a_t - a_o$$

Note that a_o is the value of a linear output unit: $a_o = a_l w_{ol}(t)$. Because in the considered case $a_l = 1$ we have

$$w_{ol}(t+1) = w_{ol}(t) + a_t - w_{ol}(t) = a_t$$

This means: regardless of the current weight between the active local unit and an output unit, the new weight equals exactly the activation value of the corresponding target unit.

According to this learning scheme each distributed symbol is stored at a distinct location (has its own weights) and is therefore not affected by other symbols in memory. The result is a dualistic representation scheme: each symbol is represented both in a local and a distributed form.

I have improved this simple symbol module in many ways, for example introducing 'pointers' to control the activation of local memory units. This allows to build symbolic data structures like stacks and lists. (The first larger system I have build wiht symbol modules is a simple LISP interpreter.)

4.2 Content-Addressed Retrieval

Suppose that the layer of local memory units and the layer of output units are totally connected in *both* directions. Symbols consist of $1, -1$ patterns. The storing mechanism is as before but each dictated weight change now operates in both directions yielding symmetrical connections: $w_{ol} = w_{lo}$.
Such symbol modules can be used from two sides to retrieve stored symbols. In addition to location-addressed retrieval (from the local units to the output units), *content-addressed* retrieval from the output units (acting as input units) to the local memory units is possible.
Content-addressed retrieval works as follows: If symbol s_i, for example M, is presented to the output units, activation flows down to the local units. Because the weights to each local unit represent a stored symbol, the current activation pattern s_i is compared in parallel to all stored symbols! The net input to local unit l_j reflects the strength of match between s_i and s_j.
If each local unit has a threshold of $N_s - 2$ the symbol(s) which *exactly matches* s_i become active. If the local units are designed as a Winner-Take-All (WTA) network, the *best matching* symbol is retrieved. Best match also allows *pattern completion* if some output units are clamped to 0.

The combination of content- and location-addressed retrieval can be used to bind values to variables. Suppose the sequence 'X,3,Y,5,Z,7' is stored in a symbol module. Now to retrieve the value of variable 'Y', one just has to present its pattern at the output units. This activates its corresponding local memory unit. At the next time step its right neighbor is active producing the corresponding value. Thus retrieving the value of a variable operates in constant time, independent of the number of stored variables. I have enhanced this variable binding mechanism to bind sequences of elements and to allow to retrieve the value of the value of a variable.

4.2.1 Parallel Rule Matching

On the basis of the described features of symbol modules, I have developed a simple production system which processes the Pig Latin rules (see figure 2).

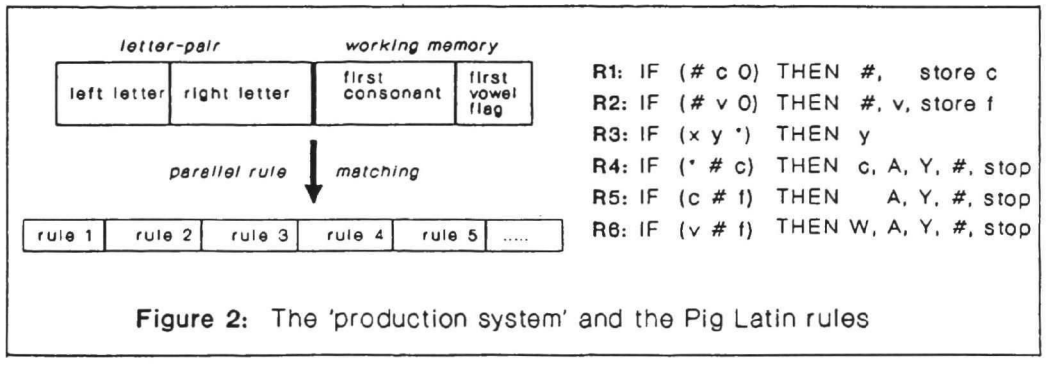

Figure 2: The 'production system' and the Pig Latin rules

This special symbol module has a larger symbol gate to represent two letters and some further information (working memory). A condition is represented through one local unit. In this version exact match is used. No (local) variables are yet established. If a local unit becomes active, it activates its right neigbors constituting the action part. Actions can send symbols to a special output module or store values in working memory. There is no conflict resolution yet implemented, thus the rules must be defined such that the conditions exclude each other.
There are several ways to define the Pig Latin rules within the scope of this production system.

Since no local variables are yet allowed, I have restricted the rules in the following way: only three consonants (B, M, T) and two vowels (E, I) are used. Additionally, if a word begins with a consonant, the next letter must be a vowel. For reasons which will become clear in the next section a string is not presented as a sequence of letters, but in small chunks, consisting of letter-pairs. Word-edges are indicated with the special symbol '#'. Thus the string #TIME# is actually presented as the following sequence: #T, TI, IM, ME, E#.
The rules for the production system are shown in figure 2. The 'c' stands for a consonant, 'v' for a vowel and 'x' and 'y' for a vowel or a consonant but not #. The letter f represents a flag in working memory indicating that the first letter was a vowel. The '*' indicates that the short term memory may have an arbitrary value. Note that each rule is actually repeated several times with local variables replaced with letters. Thus an actual rule as an instance of the first rule is: 'IF (# B 0) THEN #, store B'.
Consider as an example how the string #TIME# is processed. At first, #T is presented to the output units. Now all conditons are compared in parallel against the data. The first rule matches, sending # to the output module and stores T in the short term memory. Now the next pair TI is presented and rule 3 matches, sending I to the output module. The next pairs IM and ME are treated the same. Then the last pair, E#, is presented. Rule 4 matches, finally sending T, A, Y and # to the output module.

5 PDP-Modules: Learning by Example

PDP-modules are ordinary (recurrent) back-propagation networks, thus possessing their main advantages: simultaneous consideration of many pieces of information, learning from experience and generalization to novel situations. To exploit these features of parallel distributed processing, a different approach to learn the Pig Latin rules is reasonable: present a whole input string at once to an input layer and compute in one step, in parallel, the correct output string.

But there is a problem: how are strings – entities of varying length – represented with a fixed number of units? One solution might be to use a buffer large enough to hold the longest string: the input layer and output layer are divided in many parts ('slots'), each representing one element (letter) in successive order. But there are some problems with this *position dependent* representation (Mozer, 1988). The most important problem with regard to the Pig Latin task is that this representation does not support proper generalization because it cannot handle word edges – the most critical information – in isolation, independant from the length of the string.
Rumelhart and McClelland (1986) solved this problem within the scope of the past tense model using a *context-sensitive* representation: each element (phoneme) is represented together with its predecessor and its successor constituting a "Wickelphone". To represent the word 'represent', it is decomposed into the following set of triplets: #Re, rEp, ePr, pRe, rEs, eSe, sEn, eNt, nT#. Now each Wickelphone represents a 'slot' (one input unit) and all words ending in the same phonemes have the same slot active! This allows to extract the relevant regularities.

According to this representation scheme I have implemented a simple two layer network to process strings according to the Pig Latin rules. To keep things simple only 'Wickel-pairs' are used. Of course, letter-pairs aren't well suited to represent strings distinctly (Also Wickelphones cannot represent every string distinctly; see Pinker & Prince (1988) for a thorough criticism of the past tense model). Therefore I have defined the additional constraint that no letter may appear twice. This constraint together with those of the previous section lead to 39 letter-pairs (39 input and 39 output units) and 71 distinct strings.
The network learns the Pig Latin task quickly. After training with 50% of the strings it generalizes well to almost all remaining strings (only three strings are processed incorrect).
This solution however has two serious drawbacks. First, if more letters are allowed to build strings, there are quickly too many letter-pairs (this was the main reason to use 'Wickelfeatures' – a more compact, distributed representation – in the past tense model). Second, a simple feedforward network cannot handle sequentially presented data.

What one would like is a network which combines elegantly the strengths of both parallel and serial processing. Therefore I have developed a PDP-module which can sequentially process letter-pairs and can hold the presented information simultaneously active in a highly distributed (superimposed) manner.

In the following, the basic idea is described using local representations of the pairs in a pre-wired simple network. Then it is outlined how the network can learn itself proper representations.

Figure 3 shows a small illustration network. The input layer and the output layer consist of two 'slots' each representing one of three elements (A, B or C). The hidden layer consists of six local units representing letter-pairs. Now suppose the elements B and C are presented to the input layer. They activate the pair unit BC which in turn activates the elements B and C of the output layer (see figure 3). Each (linear) pair unit is self connected with a value of +0.8. Thus, the activity of the active pair unit slowly decays as time proceeds and consequently the activity of the output units, too. But this does not really happen because of recurrent connections from the left three output units to the pair units. These output units have positive connections to pairs which 'match' the output unit with its left side. The output unit representing B, for example, has positive connections to all pairs Bx and zero weights to all other pairs.

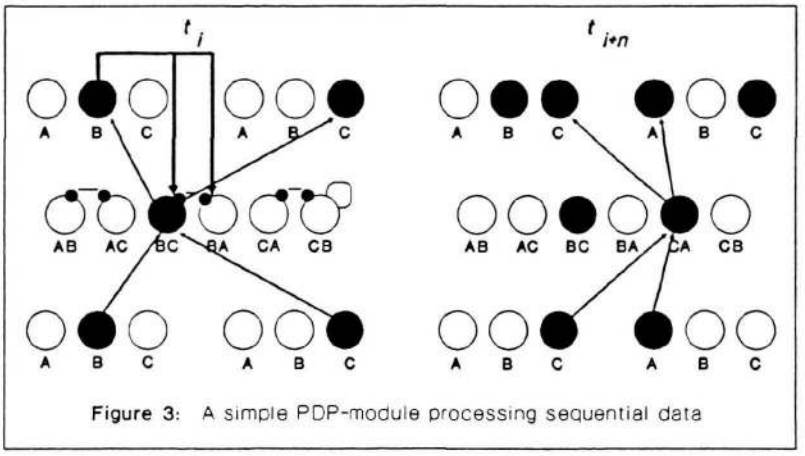

Figure 3: A simple PDP-module processing sequential data

Thus at the next time step the active output unit B activates the pair units BA and BC. This prevents the pair unit BC from decaying to zero but BA gets some activity, too. However each pair unit has inhibitory connections to other pair units with the same left element. This allows BC to prevent BA from becoming active. In summary, the activity state of the pair units at time step $t+1$ is the same as before. The active pair unit and the active output unit mutually reinforce each other as time proceeds creating a stable 'resonant state' (Grossberg, 1987).

Why not using self connections of value +1? In this case no recurrent connections from the left output units would be necessary. The basic trick however is that the same left output unit can activate different units of the right output units depending on the stored pair units. Thus the system acts as a 'dynamic pattern associator': the mapping function from left to right is modulated by the activity of the pair units!

Now suppose the next letters C, A are presented; they turn on the CA pair unit which itself activates the corresponding output units (see figure 3). Through resonance both pair units remain active. Note, that from looking only to the output units one cannot decide whether B goes with C and C with A or whether B goes with A and C with C! This is essentially the variable binding problem (Smolensky, 1987). The right information is present in the hidden layer of pair units. Retrieving that knowledge operates as follows: all left output units are clamped to 0 except the unit in question. If for example A and C are clamped to zero, only the pair BC gets support from B, CA starts to decay. Consequently the activity of the A-unit on the right side also decreases. Thus only the left B and the right C remains active!

This small network was entirely "hand-wired" to explain the basic idea. Is it possible to make a network learn the right weights on its own? The answer is yes, if the network structure is restricted in some ways. PDP networks do generalize to novel inputs but there are often too many generalization functions. A possible solution to this problem is to build some a priori ('innate')

knowledge into the network, guided from the solution space the network might discover.
I have replaced the fixed weights in the described network with random weights but imposed the following constraints: only positive connections are allowed from lower to higher layers and also from the left output units to the hidden layer. Within the hidden layer only inhibitory connections are allowed except the self-connections which are fixed at 0.8.
The network is trained with all pairs. At the first time step, two letters are presented to the network and also used as target values for the output units. At the next time step, the hidden units gets its own previous activations as target values. According to this training regime, the network tries to find distributed representations 'rLR' in the hidden layer which must suffice three conditions: L + R → rLR (input to hidden); rLR → L + R + r'LR (hidden to output and hidden to hidden); L + r'AB → rAB (left output and (decayed) hidden to the same hidden as before). Note that only single pairs are used to train the network but the imposed constraints strongly suggest generalization to hold more than one pair active.

This network finds interesting distributed representations for the letter pairs. If strings are presented sequentially the distributed representations of the letter-pairs are superimposed upon each other. The network has no fixed capacity to hold letter pairs but saturates gracefully with the number of stored pairs. The capacity can be determined through the number of used hidden units (the capacity depends upon the alphabet size and the number of units, see Rosenfeld & Touretzky, 1988). This is in contrast to the variable binding mechanism – the tensor product representation – used by Smolensky (1987) which has a fixed capacity.

In summary, the final PDP-module processes sequences of letter pairs constructing a highly distributed representation which produces a corresponding representation of the output string. This representation in turn can be decomposed to the elemantary letters. Thus it can handle sequential input and output like the symbol module.

6 The Synthesis: Rules and Practice

In the last sections it was shown how the Pig Latin rules can be taught to symbol-modules and PDP-modules. Both modules solve the problem exploiting their special advantages. To combine the virtues of each module, they are finally integrated into a larger system which operates as follows (see figure 4):

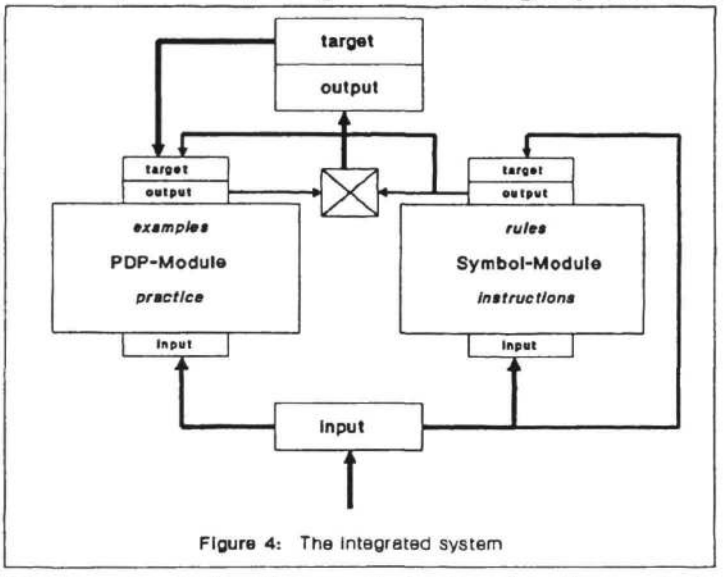

Figure 4: The integrated system

First, the Pig Latin rules are presented and successively stored in the symbol module. The system is now able to process strings according to the rules. The resulting strings obtained by the symbol module are send to the target units of the PDP-module. Therefore the PDP-module gradually learns to mimic the behavior of the symbol module and extracting the underlying regularities.
Consider a more interesting case: The Pig Latin rules are presented to the symbol module but not the special rule which handles strings both beginning and ending with a vowel. This case resembles

more closely to real life situations where instructions cover only the most strong regularities but not all subtle details. Now the symbol module operates correctly most of the time. Only if a string beginning and ending with a vowel is to be processed the symbol module produces an incorrect output string. In this case it is assumed that the environment provides an external teaching input which is transferred to the PDP-module. Thus the PDP-module is trained from two sources: most of the time from the symbol module (if it produces correct output) and sometimes from an external teacher yealding finally to more accurate and fluent performance than the symbol module.

7 Conclusions

This paper presents a suggestive approach to high-level cognitive modeling: a connectionist system learning both like traditional AI-systems, emphasizing rule-like knowledge, and also like PDP models, emphasizing learning through experience.
The concrete realization of this idea poses many new questions about the interactions of the proposed modules. The first results obtained are preliminary in nature but they promise that more complex systems will lead to attractive psychological models of skill acquisition and also offer new ways to escape the 'brittleness' of many conventional AI-systems.

Acknowledgements

Many thanks to Hans-Henning Schulze for helpful conversations during the writing of this paper. Thanks too to Thomas Göttsche, Klaus Hahn, Dieter Heim, Dirk Vorberg and Kai-Uwe Wagner.

References

Goebel, R. (1990). Learning Symbol Processing with Recurrent Networks. In R. Eckmiller (Ed.), Proceedings of the International Conference on Parallel Processing in Neural Systems and Computers (ICNC), Düsseldorf, F.R.G., 19-21 March, 1990.

Grossberg, S. (1987). Competitive Learning: From Interactive Activation to Adaptive Resonance. Cognitive Science, **11**, 23-63.

Harvey, B. (1985). Computer Science Logo Style. Volume I, MIT Press, Cambridge, MA.

Mozer, M.C. (1988). A Focused Back-Propagation Algorithm for Temporal Pattern Recognition (Tech. Rep.). University of Toronto, Departments of Psychology and Computer Science

Norman, D. A. (1986). Reflections on Cognition and Parallel Distributed Processing. In D. E. Rumelhart and J. L. McClelland (Eds.), Parallel Distributed Processing. Volume II, MIT Press, Cambridge, MA.

Pinker, S. & Prince, A. (1988). On Language and Connectionism: Analysis of a Parallel Distributed Processing Model of Language Acquisition. Cognition, **28**, 73-193.

Rosenfeld, R. & Touretzky, D. S. (1988). A Survey of Coarse-Coded Symbol Memories. In D. S. Touretzky, G. Hinton and T. Sejnowski (Eds), Proceedings of the 1988 Connectionist Models Summer School.

Rumelhart, D.E., Hinton, G.E. & Williams, R.J. (1986). Learning Internal Representations by Error Propagation. In D. E. Rumelhart and J. L. McClelland (Eds.), Parallel Distributed Processing. Volume I, MIT Press, Cambridge, MA.

Rumelhart, D.E., & McClelland, J.L. (1986). On Learning the Past Tenses of English Verbs. In D. E. Rumelhart and J. L. McClelland (Eds.), Parallel Distributed Processing. Volume II, MIT Press, Cambridge, MA.

Smolensky, P. (1987). On Variable Binding and the Representation of Symbolic Structures in Connectionist Systems (Tech. Rep.). University of Colorado at Boulder, Department of Computer Science.