A Sequential View of AND-parallelism Through Partial AND-Processes

Bernd Schend
BASF AG
ZXT-Informatik-Technologie
6700 Ludwigshaien, West Ocrmany

Abstract

Most implementations of AND-parallelism tackle the shared variable problem by running literals in parallel only if they have no variables in common and thus are independent from each other. But even in its independent form, AND-parallel clause execution remains an intricate matter with numerous problems to be solved. First to mention, a carefully worked out synchronization mechanism is required to control process activities like dependency checking, literal ordering, message passing and intelligent back-tracking. Due to the complex nature of these problems, implementation of AND-parallelism often results in a rather opaque interplay of processes. Therefore, based on a new kind of process, this paper presents a sequential - Prolog like - view of AND-parallelism for making it more understandable on one hand and easier to implement on the other. Processes of the new type will be called *partial AND-processes.*

1 Introduction

In process based systems AND-parallelism is usually realized by introducing producer/consumer relationships among clause literals and their corresponding OR-processes. Therewith independent literals act as parallel producers of variable bindings, which will be consumed by dependent literals in subsequent steps. Dividing clause literals into producers and consumers shows the advantage of avoiding binding conflicts since shared variables can have only one producer at a time. Realizing the producer/consumer approach to AND-parallelism, however, requires a sophisticated synchronization scheme to guarantee correct process interaction. Especially in the backward phase of clause evaluation [Chang, 1985, Conery, 1987], a lot of activities have to be controlled, ranging over message passing, reordering literals, cancellation and creation of processes. Moreover, to complicate things, the whole synchronization job is settled on a single AND-process in most systems. But loading a process with so much work may seriously affect its message passing capacity and thus turn it into a communication bottleneck. With all these drawbacks in mind one might question, whether producer/consumer parallelism can actually be realized in a better arranged manner. Obviuosly it would help a system designer if he could view parallel clause execution in terms of a well known and easy to understand sequential system. Offering such a view through *partial AND-processes,* a new kind of process, will be the subject of this paper.

The paper starts by introducing some notational framework. After that, the principles of process based program execution will be illustrated by the hand of AND/OR-trees. Two fundamental search strategies for such trees will be discussed then, before the special strategy giving AND-parallelism a sequential touch is explained in section four. Finally we pick up the task of producing output substitutions through intelligent backtracking in order to demonstrate how partial AND-processes solve a typical implementation problem of AND-parallelism.

2 Notation

For the rest of the paper the reader is assumed to be familiar with the basic terminology of logic programming as introduced by Lloyd [1984], for example. Some supplementary notation used subsequently is given below.

Variables will be named by letters x, y, z, u, and v. Letters a, b, c, and d denote constants, and f, g, h function symbols. An *expression* can be a term or literal. If e represents an expression, Var(e) is the set of all variables occuring in e. Two expressions e_1 and e_2 are called *dependent* if $Var(e_1) \cap Var(e_2) = 0$. Substitutions are named by Greek letters like 9, a, and y; a substitution $9= \{x_1/t_1,\ldots,x_n/t_n\}$, n>O, *binds* variable X to

the term t_i, $1 \le i \le n$. *Composition* $\theta_1 \cdot \theta_2$ of two substitutions is defined as $\theta_1 \cdot \theta_2(e) = \theta_1(\theta_2(e))$ for every expression e. Finally, the number of elements of a finite set M will be denoted by |M|.

3 Program Execution by AND/OR-processes

Every logic program - started with some goal - implicitly defines an AND/OR-tree [Kowalski 1979], which makes program execution a matter of searching all the answers to the user's query. In this connection an execution strategy for logic programs naturally turns into a *search strategy*, which tells a system how to traverse the tree under consideration. To illustrate the effects of different search strategies, we now describe the behaviour of some fictitious process systems when executing the first clause from figure 1.

(1) p(x,y) ← q(x), r(y), s(x)
(2) q(a)← (4) r(c)← (6) s(a)←
(3) q(b)← (5) r(d)← (7) s(b)←

Figure 1 Example program

Based on AND/OR-trees, processes created during program execution constitute an *AND/OR-process tree*. By the way, viewing processes as nodes of a tree allows us, for instance, to talk of the AND-father of a process, just saying the father is an AND-process. Conventions for drawing process trees are shown in figure 2.

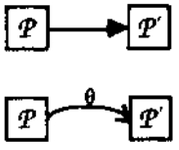

Figure 2 Parts of a process tree

As the special strategy coming up with partial AND-processes will be both, sequential and parallel in nature, the single components of this mixture will be considered first.

Sequential Search Strategy

The sequential component within our mixed strategy is a left to right one comparable to that used in standard Prolog systems. Figure 3 mirrors its effect on execution of clause p(x,y) ← q(x), r(y), s(x), which - for the sake of simplicity - is assumed to be called with the empty input $\theta_0 = \emptyset$.

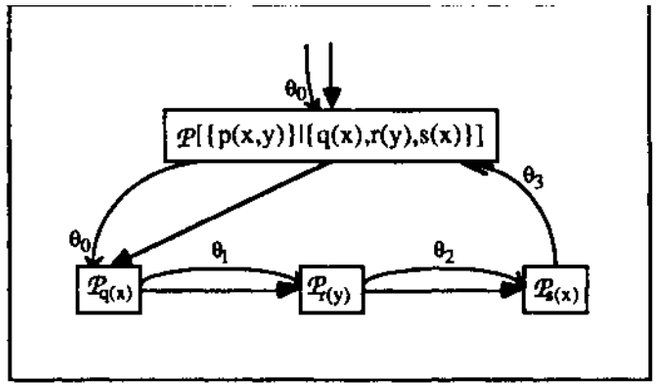

Figure 3 Sequential execution of p(x,y) ← q(x), r(y), s(x)

The process tree resulting from our left to right strategy shows an AND-process, named $\mathcal{P}[\{p(x,y)\}|\{q(x),r(y),s(x)\}]$, which is created to execute our example clause. In order to solve this task, the AND-process initially starts a descendant OR-process $\mathcal{P}_{q(x)}$ to compute solutions for $\theta_0 q(x) = q(x)$. Having found the answer $\gamma_1 = \{x/a\}$, process $\mathcal{P}_{q(x)}$ produces its output substitution $\theta_1 = \gamma_1 \cdot \theta_0 = \{x/a\}$ and then creates a descendant OR-process $\mathcal{P}_{r(y)}$ to solve literal $\theta_1 r(y) = r(y)$. On the basis of our example program, $\mathcal{P}_{r(y)}$ subsequently finds the answer $\gamma_2 = \{y/c\}$ and computes the output $\theta_2 = \gamma_2 \cdot \theta_1 = \{x/a, y/c\}$. Next, OR-process $\mathcal{P}_{s(x)}$, created for $\theta_2 s(x) = s(a)$, will find $\gamma_3 = \emptyset$ as a solution and send substitution $\theta_3 = \gamma_3 \cdot \theta_2 = \{x/a,y/c\}$ to the AND-process above. Finally, θ_3 represents a solution for the whole program clause.

Parallel Search Strategy

In contrast with a sequential strategy, using a parallel one causes some body literals to be executed in parallel. Figure 4 shows the effect of a somehow extremely parallel strategy since it solves all body literals simultaneously.

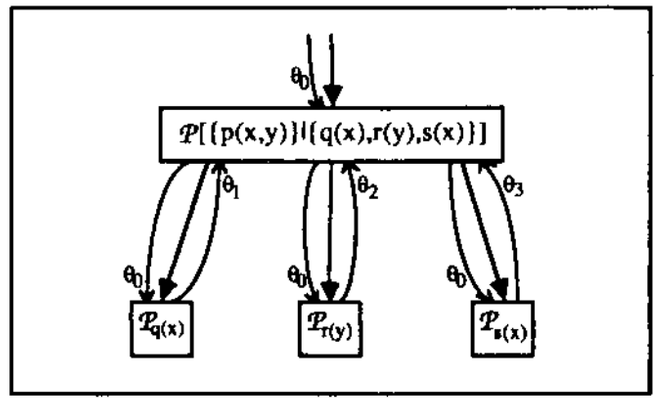

Figure 4 Parallel execution of p(x,y) ← q(x), r(y), s(x)

According to the parallel strategy, all body literals are executed independently by corresponding OR-processes. Process $\mathcal{P}_{q(x)}$, for instance, works on $\theta_0 q(x) = q(x)$ and produces the output substitution $\theta_1 = \{x/a\}$. In the same way, processes $\mathcal{P}_{r(y)}$ and $\mathcal{P}_{s(x)}$ compute answers for $\theta_0 r(y)$ and $\theta_0 s(x)$, which we assume to be $\theta_2 = \{y/c\}$ and $\theta_3 = \{x/b\}$, respectively. Yet, from the point of the AND-process above, θ_1, θ_2 and θ_3 represent only partial solutions for its own problem, which is to find answers for whole program clause. Therefore, having received the single solutions, an AND-process has to join them to a complete one, provided values of shared variables are consistent with each other. Consistency of variable bindings, however, is not guaranteed. Just look at processes $\mathcal{P}_{q(x)}$ and $\mathcal{P}_{s(x)}$ from above, which bind variable x to non unifiable values.

To get around the consistency problem, most systems [Chang, 1985, Conery, 1987, DeGroot, 1984] synchronize OR-processes by making them producers or consumers of variable bindings. Since shared variables can have only one producer at a time, no conflicting bindings will arise. But as mentioned before, producer/consumer parallelism often leads to a rather intricate interplay of processes. The intention of this paper, therefore, is to give a more perspicuous view of AND-parallelism and - by doing so - convey new ideas of how typical implementation problems can be solved in a more understandable and efficient manner. A new type of process, called partial AND-process, will play a central role thereby.

4 Partial AND-processes

As their name already indicates, partial AND-processes bear some likeness to *normal* AND-processes, but differ in the scope of the problem to deal with. Whereas an AND-process has to solve all the literals of a clause body, a partial AND-process works on a subset of body literals only. Such a subset just corresponds to a group of independent literals selected for parallel execution. Whereas literals within a group will be solved in parallel, group execution is purely sequential in nature. Consequently, we get a Prolog like clause execution on the level of p(artial)AND-processes and their corresponding literal groups (see upper half of figure 5).

In regard of a more detailed description of what happens in figure 5, we first of all claim that special routines for dependency checking and literal grouping are available to AND- and pAND-processes. Due to lack of space, we simply view these routines as black boxes and describe their behaviour only superficially. Again, for the execution of our program clause we assume its input substitution 6_0 being empty.

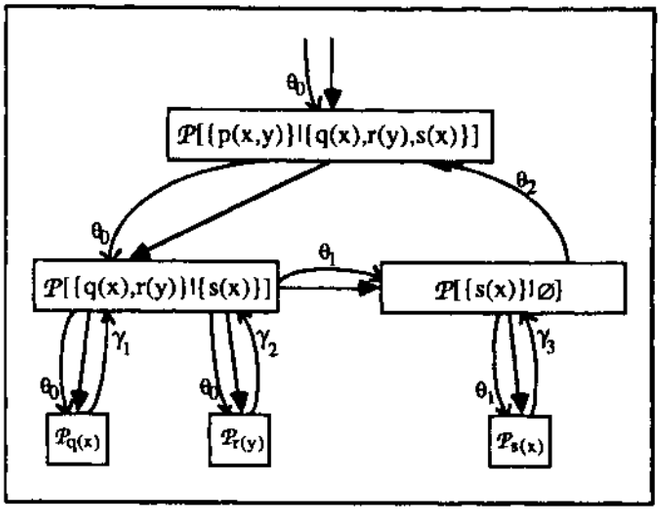

Figure 5 pAND-parallel execution of
$p(x,y) \leftarrow q(x), r(y), s(x)$

The AND-process $\mathcal{P}[\{p(x,y)\}|\{q(x),r(y),s(x)\}]$ starts clause execution by determining literal dependencies within the environment θ_0. This practically means testing, whether $Var(\theta_0 L) \cap Var(\theta_0 L') \neq \emptyset$ for all pairs (L,L') of distinct body literals. As a result we get two alternatives for grouping independent literals, namely $G_1 = \{q(x),r(y)\}$ and $G_2 = \{s(x),r(y)\}$. Next, the AND-process decides to group q(x) and r(y) together, and upon that, has them executed by a pAND-process $\mathcal{P}[\{q(x),r(y)\}|\{s(x)\}]$. This one starts OR-processes $\mathcal{P}_{q(x)}$ and $\mathcal{P}_{r(x)}$ to solve literals $\theta_0 q(x)$ and $\theta_0 r(y)$ in parallel. On the basis of our example program from figure 1 the answers $\gamma_1 = \{x/a\}$ and $\gamma_2 = \{y/c\}$ will be computed and send to $\mathcal{P}[\{q(x),r(y)\}|\{s(x)\}]$. Having received both substitutions, the pAND-process produces the output substitution $\theta_1 = (\gamma_1 \cup \gamma_2) \cdot \theta_0 = \{x/a, y/c\}$ and immediately creates its son $\mathcal{P}[\{s(x)\}|\emptyset]$ with input substitution θ_1. In order to solve $\theta_1 s(x) = s(a)$ the pAND-son starts an OR-process $\mathcal{P}_{s(x)}$, from which it receives the answer substitution $\gamma_3 = \emptyset$. Finally, $\mathcal{P}[\{s(x)\}|\emptyset]$ computes its output $\theta_2 = \gamma_3 \cdot \theta_0 = \{x/a,y/c\}$ and transfers it via message to the AND-process above.

The last example showed partial AND-processes acting as producers and consumers of variable bindings. Yet, unlike Conery's model [Conery, 1987], bindings are not produced for single literals, but for whole groups of literals. To identify the literal group a pAND-process is executing we have used the notation $\mathcal{P}[\Gamma|\Pi]$ with Γ and Π being sets of clause literals. Separation by a stroke characterizes Γ as the the literal group actually executed by the process, whereas literals in Π have to

be solved by other pAND-processes in subsequent steps. Due to this separation, literals in T get an active and literals in Π a passive character from the point of $\mathcal{P}[\Gamma|\Pi]$.

Definition 4.1

Suppose *P* is an AND- or pAND-process named $\mathcal{P}[\Gamma|\Pi]$. Then *AL(T)*= r is the set of *active* and $\mathcal{PL}(\mathcal{P}) = \Pi$ the set of *passive literals* of P.

Active literals and their corresponding OR-processes can be regarded as producers of variable bindings, which are gathered by a pAND-process to built its own output substitutions. However, that there are several OR-processes contributing to an output, is completely hidden to the pAND-son, which consumes it as an input substitution. From the point of a consuming son, its pAND-father together with all its active literals simply appears as a single module just producing output substitutions.

5 Producing Output Substitutions

Solutions produced by pAND-processes will be sent as output substitutions to the consuming process, which itself receives them as input substitutions. In general, sending of solutions is demand driven, since new output/input substitutions are only produced when explicitly requested by the consumer. Output substitutions generated by pAND-processes are characterized below.

Definition 5.1

Let $\mathcal{P}$ be a pAND-process with input substitution θ_{in} and its active literal set $\mathcal{AL}(\mathcal{P}) = \{L_1,...,L_n\}$, $n \geq 1$. Further, let $S_1,..,S_n$ be the sets of answer substitutions computed by descendant OR-processes, i.e. S_j contains all answers found for $\theta_{in}L_j$, $1 \leq j \leq n$.

Then $OS(\mathcal{P}, \theta_{in}) = \{\text{Join}(\gamma_1,...,\gamma_n)\cdot\theta_{in} | \gamma_j \in S_j,\ 1 \leq j \leq n\}$ is the set of *output substitutions* of $\mathcal{P}$, where $\text{Join}(\gamma_1,...,\gamma_n) = \gamma_1 \cup ... \cup \gamma_n$.

The definition above points out that every output substitution produced by a pAND-process consists of a constant and variable part. The first one corresponds to the input substitution θ_{in} and the latter to a tuple $(\gamma_1,...,\gamma_n)$ of answers computed by descendant OR-processes. Therefore, given a fixed input substitution θ_{in}, each output substitution $\theta_{out} = \text{Join}(\gamma_1,...,\gamma_n)\cdot\theta_{in}$ is practically determined by the tuple $(\gamma_1,...,\gamma_n)$. Moreover every such tuple in its turn may be identified with a tuple of natural numbers. Simply realize that the elements of a set S_j can be numbered in some way, for instance, in the order they are computed by the corresponding OR-process. Conversely, to get an output substitution from a tuple $(m_1,...,m_n)$ of element numbers, just take the m_1-th element from S_1, the m_2-th element from S_2,..., the m_n-th element from S_n, join them together and produce a θ_{out} through composition with θ_{in}.

This strong relationship between number tuples and output substitutions allows us to make production of output substitutions more perspicuous by reducing it to a problem of generating tuples of natural numbers. For solving the generation problem we simply adopt the nested loop model introduced by Conery [1987]. A pAND-process will always start tuple generation with the initial tuple (1,...,1), subsequently changing its components as if they were variables of nested for-loops, i.e. the rightmost components run faster than the ones to the left. Note, if component i, $1 \leq i \leq n$, is increased, all components j, $i < j \leq n$, are reset to 1. As number tuples of the kind mentioned above directly determine output substitutions, they are called *output tuples* from now on.

As said before, production of output tuples and substitutions is demand driven, i.e. a pAND-process produces a new output only when explicitly requested by its pAND-son. Such a request always goes along with a *backtracking between partial AND-processes*. What's behind this type of backtracking will be demonstrated below with a slightly modified version of our example program.

(1) p(x,y) ← q(x), r(y), s(x), t(y)
(2) q(a)← (4) r(c) ← (6) s(b) ← (7) t(u) ←
(3) q(b)← (5) r(d) ←

To execute clause (1) with input substitution $\theta_0 = \emptyset$, an AND-process, named $\mathcal{P}[\{p(x,y)\}|\{q(x),r(y),s(x),t(y)\}]$, will be started initially. Let us assume q(x) and r(y) to be grouped together after dependency checking and executed by a pAND-process $\mathcal{P}[\{q(x),r(y)\}|\{s(x),t(y)\}]$. For solving $\theta_0 q(x)$ and $\theta_0 r(y)$, OR-processes $\mathcal{P}_{q(x)}$ and $\mathcal{P}_{r(y)}$ will be created subsequently, which compute the answers $\gamma_{11} = \{x/a\}$, $\gamma_{12} = \{x/b\}$ for the first, and $\gamma_{21} = \{y/c\}$, $\gamma_{22} = \{y/d\}$ for the second literal. Suppose that the pAND-process receives the substitutions in the order γ_{11}, γ_{12} and γ_{21}, γ_{22}, respectively. According to the nested loop model, the initial tuple (1,1) is used to produce the first output substitution $\theta_{(1,1)} = \text{Join}(\gamma_{11},\gamma_{21})\cdot\theta_{in} = \{x/a,y/c\}$. Next, the process creates its pAND- son $\mathcal{P}[\{s(x),t(y)\}|\emptyset]$ with input $\theta_{(1,1)}$. This son in return starts the OR-processes $\mathcal{P}_{s(x)}$ and $\mathcal{P}_{t(y)}$, the first of which detects that there are no matching clauses for its literal $\theta_{(1,1)}s(x) = s(a)$. Figure 6 shows a snapshot of the process tree

at this moment

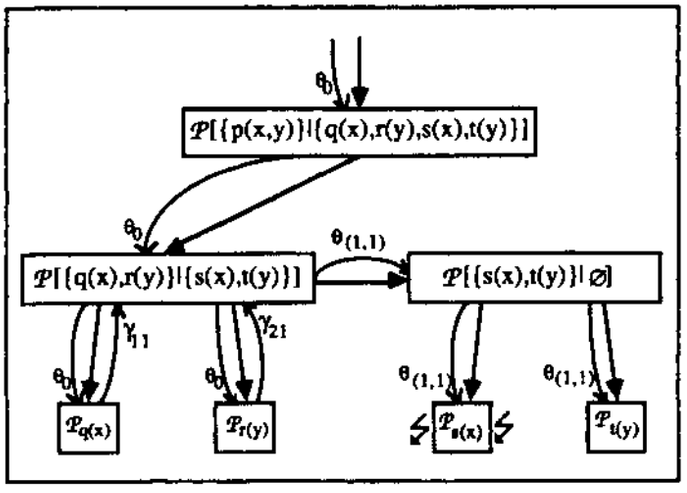

Figure 6 Backtracking situation between pAND-processes

Process $P_{s(x)}$ reports its failure to $\mathcal{P}[\{s(x),t(y)\}|\emptyset]$, which immediately reacts by requesting a new input substitution from its pAND-father $\mathcal{P}[\{q(x),r(y)\}|\{s(x),t(y)\}]$. Based on the nested loop model, the father generates the next output tuple (1,2) and sends the corresponding substitution $\theta_{(1,2)} = Join(\gamma_{11},\gamma_{22}) \cdot \theta_{in} = \{x/a, y/d\}$ to the requesting pAND-son. However, $\theta_{(1,2)}$ will obviously lead to a failure, too, because the value of variable x remaines unchanged by the new substitution. So, our pAND-process generated just another *failure tuple.*

Definition 5.2

Let *P* be a pAND-process with output tuple t and corresponding output substitution θ_t. Tuple t is called a *failure tuple* if $\theta_t L$ is unsolvable for some active literal of the pAND-son of P. Else it will be called a *success tuple.*

To avoid repeated generation of failure tuples - like (1,1) and (1,2) in the example above - a pAND-process must incorporate some kind of intelligent backtracking [Conery, 1987, Woo and Choe, 19861, or equivalently, use some form of *intelligent tuple generation.* This should, for instance, allow process $\mathcal{P}[\{q(x),r(y)\}|\{s(x),t(y)\}]$ of figure 6 to generate tuple (2,1) directly without regarding (1,2). Tuple (2,1) and its corresponding output substitution $\theta_{(2,1)} = Join(\gamma_{12},\gamma_{21}) \cdot \theta_{in} = \{x/b,y/c\}$ bind x to a different value, with which clause execution succeeds finally.

The question now is, how tuple generation can be made intelligent. To achieve this aim, a pAND-process *P* obviously needs some kind of information about what was wrong with its actual output substitution. Within the parallel execution scheme presented here, this information will be supplied by the pAND-son of P by sending all its *failure literals.*

Definition 5.3

Let P be a pAND-process named P[r|n]. An active literal LeT is called a *failure literal* if its corresponding OR-process fails to compute any answer substitution. The set of all failure literals of P is denoted by F*L(T).* For an AND-process P, we set FL*(P)*= *T* by definition.

Our example process $\mathcal{P}[\{s(x),t(y)\}|\emptyset]$ figure 6, for instance, would report s(x) as a failure literal to its pAND-father: $\mathcal{P}[\{q(x),r(y)\}|\{s(x),t(y)\}]$. Now the father process can conclude that it must produce a new binding for variable x, for changing the value of y would definitely not repair the failure and thus lead to another backtracking. As a new value for x can only be achieved by changing the first component of the actual tuple (1,1), we automatically know (1,2) to be another failure tuple. This is confirmed by looking at its corresponding output substitution $\theta_{(1,2)} = \{x/a, y/d\}$, which changes the value of y, but not that of x.

In order to determine all its failure literals a pAND-process proceeds as follows: After descendant OR-processes have been created for every active literal, it enters a kind of *test state.* There the process waits until all OR-sons have acknowledged by sending a success or failure message. The literal of a failing OR-process will be recorded as a failure literal thereby. Note, a pAND-process will wait forever if one of its OR-sons is caught in an infinite loop. By the way, waiting for all OR-processes to acknowledge allows us to handle so called multiple failures [Conery, 1987, Woo and Choe, 1986] in a rather simple manner. Further, collecting all failure literals before backtracking results in a more intelligent version since information about failures gets more comprehensive. This information will be used to repair the failure by generating a new output tuple, the corresponding substitution of which binds at least one variable in each failure literal to another value. Such tuples and substitutions will be *called failure relevant* from now on.

In terms of our nested loop model the whole problem of generating tuples intelligently may now be stated as: What loop variable, or equivalently, what tuple component should be changed in order to make its corresponding output substitution relevant to the failure ? To answer this question we first of all introduce a successor relation on AND- and pAND-processes. Definition of the relation is based on the process tree of figure 7, which shows an AND-process $\mathcal{P}[\Gamma_0|\Pi_0]$ with descendant pAND-processes $\mathcal{P}[\Gamma_1|\Pi_1],...,\mathcal{P}[\Gamma_n|\Pi_n]$. Note, $\Pi_n = \emptyset$.

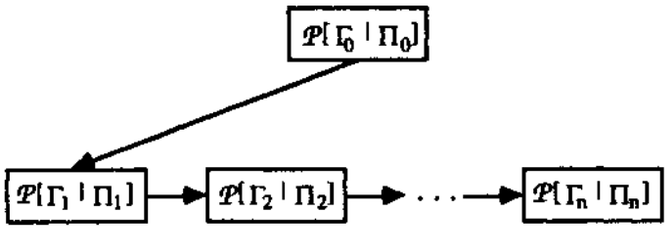

Figure 7 AND-process $\mathcal{P}[\Gamma_0|\Pi_0]$ with descendant pAND-processes

Definition 5.4

Let $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n$, $n \geq 1$, be the processes of figure 7 with $\mathcal{P}_i = \mathcal{P}[\Gamma_i|\Pi_i]$ and $0 \leq i \leq n$. The *successor* of $\mathcal{P}_i$ is defined by $succ(\mathcal{P}_i) = \mathcal{P}_j$ with $j = i+1 \bmod n+1$. Similarly we define the *predecessor* of $\mathcal{P}_j$ by $pred(\mathcal{P}_j) = \mathcal{P}_i$ iff $succ(\mathcal{P}_i) = \mathcal{P}_j$. Instead of $succ(\mathcal{P})$ and $pred(\mathcal{P})$, we also use the notations $\mathcal{P}_{succ}$ and $\mathcal{P}_{pred}$, respectively.

Next, we relate active literals of a pAND-process to active literals of its successor. Literals of the first will be treated as producers, literals of the second as consumers of variable bindings. In addition, a special function is introduced for mapping active literals to tuple components and vice versa.

Definition 5.5

Let $\mathcal{P}$ be a pAND-process with input substitution θ, $L_p \in \mathcal{AL}(\mathcal{P})$ and $L_c \in \mathcal{AL}(\mathcal{P}_{succ})$.

a) $consumer(L_p) = \{L' \in \mathcal{AL}(\mathcal{P}_{succ}) \mid Var(\theta L_p) \cap Var(\theta L') \neq \emptyset\}$.

b) $producer(L_c) = \{L' \in \mathcal{AL}(\mathcal{P}) \mid Var(\theta L_c) \cap Var(\theta L') \neq \emptyset\}$.

c) # is a bijective mapping from $\mathcal{AL}(\mathcal{P})$ to $\{1,\ldots,n\}$, $n = |\mathcal{AL}(\mathcal{P})|$. For a set Γ of literals we define $\#\Gamma = \{\#L \mid L \in \Gamma\}$.

To get an idea of intelligent tuple generation, imagine a pAND-process $\mathcal{P}$ with input substitution $\theta_0 = \emptyset$ and $\mathcal{AL}(\mathcal{P}) = \{p(x), q(y), r(z)\}$, where $\#p(x) = 1$, $\#q(y) = 2$ and $\#r(z) = 3$. In addition, let (2,3,3) be the actual failure tuple of $\mathcal{P}$ and $FL = \{s(x,y), u(z)\}$ the set of failure literals reported by its successor. First of all $\mathcal{P}$ computes producer(L) for every $L \in FL$ resulting in $producer(s(x,y)) = \{p(x), q(y)\}$ and $producer(u(z)) = \{r(z)\}$. Mapping producer literals to tuple indices by the hand of function #, we get $\#producer(s(x,y)) = \{1,2\}$ and $\#producer(u(z)) = \{3\}$. Now, set {1,2} expects us to change the first *or* second component of (2,3,3), in order to produce an output substitution which binds a variable in s(x,y) to another value. Accordingly, set {3} demands the third component to be changed in order to get a new binding for u(z). Alltogether, both requirements can be satisfied by increasing the second tuple component, because this will automatically reset the third component to 1. Hence, the resulting tuple (2,4,1) causes producer literals q(y) and r(z) to contribute to the new output substitution, and all in all, modified variable bindings will be brought to their consumers s(x,y) and u(z).

In general, the simple algorithm below will be used by pAND-processes to determine the tuple component to be changed after backtracking.

Algorithm for Chanting Tuple Components

Let P be a pAND-process P[T\TII and FL the set of failure literals reported by its successor. Further, for a literal $L \in FL$ let $max_L = max\{\#L' \mid L' \in producer(L)\}$.

if $producer(L) \neq \emptyset$ for every $L \in FL$
then $i := min\{max_L \mid L \in FL\}$;
"change the i-th tuple component"
else "stop tuple generation".

In the example above we had FL = (s(x,y), u(z)}. Computing max_L for every failure literal L results in $max_{s(x,y)} = max\{1,2\} = 2$ and $max_{u(z)} = max\{3\} = 3$. Using the algorithm for changing tuples, we get the 2-nd component, 2 = min{2,3), as the one to be changed.

In general, changing the i-th component of a tuple usually means increasing its value by one. However, this may be impossible sometimes because all members of the answer set S_i have already been considered, i.e. the value of the i-th component corresponds to the number of elements in S-. Should this happen, a pAND-process reacts by looking for the first component to the left of i which can be changed. If there is such a component, its value will be increased therewith resetting all components to its right to 1. After having produced the corresponding output substitution, the process enters a kind of forward state, in which dependency checking and literal grouping is done again based upon the new substitution. If the resulting literal group is different from that of the actual pAND-son, the son process will be cancelled and a new one created. Otherwise no cancelling is necessary and the substitution is directly sent to the actual pAND-son. Upon that, all consumer literals receiving a new input substitution will be resolved, i.e. their corresponding OR-processes are initialized and restarted with the new bindings. Finally, having restarted its OR-sons, a pAND-process enters the test state mentioned before, waiting again for success or failure messages.

Over against this, if a pAND-proccss should fail to find an increasable tuple component, no more output tuples can be generated and tuple generation is finfished. A pAND-process, which is unable to produce new output substitutions, initiates backtracking to its own predecessor. In this manner backtracking eventually arrives at the AND-process above, telling it that clause execution is done.

At the end of this section some aspects of our backtracking scheme should be discussed:

The nested loop model for tuple generation shows the advantage of being easy to implement. But its simplicity must be paid for in some sense as tuple generation may become incomplete if OR-processes compute infinite answer sets. One might expect this a serious drawback since many tuples will never be considered. Yet, generating all of infinitely many tuples is impossible, anyway. The only disadvantage is that successful output substitutions are probably not produced if a pAND-process is stuck in an infinite answer set. Fortunately loops of this kind may be stopped through intelligent backtracking.

Requesting a new output/input substitution through backtracking is generally not restricted to pAND-processes, but also used by an AND-process to obtain all solutions for its clause. This is simply done by backtracking to its own predecessor immediately after having received an input substitution from it. The head literal of the clause will be reported as the failure literal thereby.

6 Conclusions

Implementing AND-parallelism in the producer/consumer style requires a sophisticated synchronization scheme for controlling OR-processes, which work on body literals independently. Especially handling failures through intelligent backtracking makes clause execution a rather complicated problem, the solution of which often results in a confusing interplay of processes in practice. To make AND-parallel clause execution more perspicuous and therefore easier to implement, this paper conveyed a sequential view of AND-parallelism. Based on a heterogeneous execution strategy, which is sequential and parallel in nature, independent literals are grouped together and treated as a single module. Literal groups are solved sequentially by partial AND-processes, which in their turn start descendant OR-processes for executing group literals in parallel. The main advantage resulting from the sequential part of the execution strategy is that clause execution may be tackled in a simple Prolog like manner. As an example, the backtracking scheme presented here bears some similarity to the naive right to left method of Prolog, except that it is made intelligent on the basis of producer/consumer relations. Multiple failures of OR-processes can be handled efficiently by partial AND-processes, all the more since information about failing literals is gathered to achieve a more intelligent backtracking version. Apart from this, the load on AND-processes will be eased significantly, allowing them to react to incoming messages promptly.

References

[Chang, 1985] J.H. Chang. *High Performance of Prolog Programs Based on a Static Data Dependency Analysis.* Ph.D. Thesis, 1985.

[Conery, 1987] John S. Conery. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, 1987.

[DeGroot, 1984] Doug DeGroot. Restricted And-Parallelism. In *Proceedings of the International Conference On Fith Generation Computer Systems,* Tokyo, 1984, pp. 471-478.

IKowalski, 1979] Robert A. Kowalski. *Logic for Problem Solving,* Artificial Intelligence Series, Vol.7, Elsevier-North Holland, New York, 1979.

[Lloyd, 1984] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1984.

[Woo and Choe, 1986] N.S. Woo and K.-M. Choe. Selecting the Backtrack Literal in the AND/OR-Process Model. In *IEEE 1986 Symposium on Logic Programming,* Salt Lake City, Utah, pp. 200-210.

Unsupervised Learning by Backward Inhibition

Tomas Hrycej
PCS Computer Systeme GmbH
Pfaelzer-Wald-Str. 36, D-8OO1) Muenchen 90, West Germany

Abstract

Backward inhibition in a two-layer connectionist network can be used as an alternative to, or an enhancement of, the competitive model for unsupervised learning. Two feature discovery algorithms based on backward inhibition are presented. It is shown that they are superior to the competitive feature discovery algorithm in feature independence and controllable grain. Moreover, the representation in the feature layer is distributed, and a certain "classification hierarchy" is defined by the features discovered.

1 Introduction

Connectionist, or neural network, models, i.e., models consisting of networks of simple processing nodes, have been shown to possess cognitive capabilities, in particular, the capabilities of shape recognition [Hinton and Lang, 1985, Fukushima, 1980, or von der Malsburg and Bienenstock, 1986], development of concepts [Dalenoort, 1987], learning [Grossberg, 1982, 1987, Ackley *et al..* 1985, Rumelhart *et al.,* 1986, Le Cun, 1986], storing of patterns [Hopfield, 1982], generalization [Anderson, 1986], etc..

Connectionist models provide some important advantages over symbolic models:

- automatic development of representation,
- correction of noisy input,
- graceful degradation if some cells are erroneous,
- generalization and
- inherent parallelism.

Connectionist models are typically less transparent than symbolic ones. So the preferable (or even the only possible) form of knowledge input is learning by examples.

The most straightforward form of learning is supervised learning, i.e., associating inputs with some known outputs. The best-known algorithms are backpropagation algorithm of Rumelhart *et al.* [1986] and Boltzman machine learning algorithm of Ackley *et al.* [1985]. (Analogical mathematical concepts can be found in discriminant and regression analysis [Kohonen, 1984]. For symbolic supervised learning see, e.g., Winston [1986], Quinlan [1982] or Kodratoff and Ganascia [1986].) However, in some situations, the data available are not sufficient for supervised learning:

1) "Correct" outputs, or responses, are not always known. In some cases, merely an overall criterion of response quality is given (e.g., the survival criterion for living organisms, or reaching a positive biological or emotional state for humans). This has lead to a modification of supervised learning - instead of a set of correct responses, only a reinforcement criterion is given.
2) Even with such a "weaker" reinforcement, direct learning of correct responses may fail because of a high complexity of input. This can be illustrated on the backpropagation model [Rumehart *et al.,* 1986]. It has been shown that only a limited class of input-output encodings can be materialized in a two-layer connectionist system. So more complex, multiple-layer systems and corresponding learning schemes have been developed. However, multiple-layer systems scale poorly - systems with five or more layers seem to be computationally intractable. So there are obvious limitations of the supervised learning: more complex inputs require more layers, but too many layers are, in turn, intractable.

A proposal for solution of the latter problem [Ballard, 1987] is to partition the network in some modular way. A part of the network would find (e.g., by "autoassociation", a form of unsupervised learning, in Ballard's model) a distributed and highly invariant representation of input and supervised learning would be then applied to this discovered representation, instead of the original input.

More general learning models capable of performing unsupervised classification by discovering characteristic features or regularities of input data have been presented by Rumelhart and Zipser [1985] and Grossberg [1987]. They perform essentially some clustering algorithm in a network form (for a symbolic analogy, see, e.g., conceptual clustering of Stepp and Michalski [1986] or the "hybrid" model of

Lebowitz [1985]). However, those models suffer from several deficiencies. As argued in Section 2.2, the model of Rumelhart and Zipser suffers from 1) random and uncontrollable grain of feature discovery, 2) constructing redundant and local feature representation, and 3) difficulties in building feature hierarchies (for more details, see Section 2.2).

This paper presents a novel type of feature discovery model, a backward inhibition model, that does not suffer from the above deficiencies. Mathematically, it is related rather to variance analysis than to cluster analysis.

2 Existing models

In recent past, several models for feature discovery have been presented. Most of them can be classified as "competitive models". A well-known representant of this class is critically examined in Section 2.2.

To introduce some mathematical concepts necessary for explanation of backward inhibition concept, a rudimentary one-feature model is presented in Section 2.1.

2.1 Simple correlation model

The basis for all models treated in this paper is a simple noncompetitive two-layer correlation model, further referred to as "basic adaptive model". All nodes of the input layer are connected to a single node of the feature layer (see Figure 2). A feature *y* is represented by the vector w of weights assigned to connections to a feature node. Activation level of a *y* is given by w'x, x being input vector (activation rule). This model learns by a widely used correlation learning rule, whose continuous form is:

dw = *a*y*x*dt,*

with a < < l a constant.

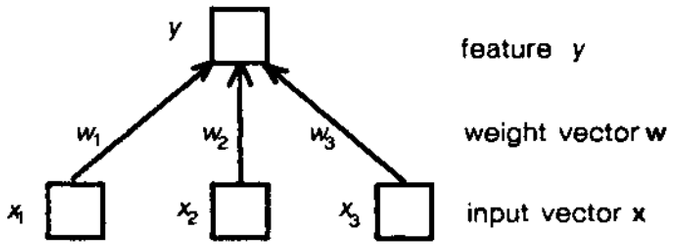

Figure 1. The network for the basic adaptive model.

The weight vector is kept normed, i.e., after each learning step, the weight vector is divided by its length |w| = sqrt(w'*w). The complete learning rule is then

$$w + dw = (w + a*y*x*dt)/|w + a*y*x*dt| =$$
$$= (w + a*y*x*dt)/(1 + a*y*x'*w*dt) =$$
$$= (w + a*y*x*dt)*(1 - a*y*x'*w*dt) =$$
$$= w + a*y*x*dt - a*y^2*w*dt$$

or

$$dw = a*y*x*dt - a*y^2*w*dt.$$

This rule is a special case of adaptation law studied by Kohonen [1984, page 103, Case 5]. As shown by Kohonen, the weight vector converges to the dominant eigenvector, i.e., the eigenvector corresponding to the largest (dominant) eigenvalue, of the input correlation matrix *E(X*X).*

Since this is equally true for all weight vectors, all feature units discover the same feature, the dominant eigenvector. So this model is no serious candidate for feature discovery. However, it can be modified in several ways to perform more satisfactorily.

2.2 Competitive correlation model

One of the possible modifications is the competitive model proposed by Rumelhart and Zipser [1985] (for a more general model, see Grossberg [1987]). The feature layer of this model consists of multiple feature nodes. Instead of activating each feature proportionally to the weighted input into the feature cell, only the feature unit with maximal activation, i.e., with maximal similarity (= inner vector product) of its weight vector with the input vector "fires" and only its own weights are modified.

An obvious interpretation of the learning process in this model is that, for a given input vector, the feature with the maximal value of similarity measure (inner vector product) between the input and its currents weights learns by moving its weights toward the input vector. So competition causes each input vector to "attract" the "nearest" feature, which results in partitioning input into exclusive "clusters". Since, because of competition between features, very different inputs will probably fire different features while similar ones will fire a single common one, the system will converge to a stable state in which each feature corresponds to a cluster of input stimuli. The similarity between members within a single cluster will be significantly higher than the similarity between those of different clusters. A more formal description of this learning process is given in [Rumelhart and Zipser, 1985].

A shortcoming of this model is that some feature cells can remain inactive (i.e., they never fire because of being always outperformed by others.

A direct consequence of this behaviour is that typically very coarse features are found. The only way to refine the grain of discovered features is by increasing number of feature cells. But the more feature cells there are, the higher the probability that many of them remain inactive.

There are two proposals for remedy in [Rumelhart and Zipser, 1985], but our analysis of them has shown that, in general case, they do not exhibit better performance than the basic competitive model [Hrycej, 1987].

There are some additional deficiencies of the competitive model:

1) Features found by the competitive model are disjoint. If their number were substantially higher than two, features would be (almost) independent. But satisfying this condition cannot

always be guaranteed - on the contrary, finding only two features is very frequent (see above and Section 3.4). But then, there is, in fact, only one independent feature - the features are simply logical complements of each other.

2) The representation of input by the feature layer is local since each input activates a single feature node. So the representation found does not possess advantageous properties (error correction, efficient coding, automatic generalization [Hinton *et al.,* 1986]) of distributed representations.

3) None of the features discovered by the competitive model represents a finer partitioning of others. This results directly from their disjointness. So all features are of the same hierarchical level. A hierarchical modification of such a system would be possible if additional layers were added so that clusters discovered by lower levels would be further partitioned by on higher levels. This arrangement requires an a priori structuring of the network (a tree structure). For above reasons, such a structure would be very inefficient: successor nodes of an inactive feature node would always remain inactive, too.

4) An additional drawback of this algorithm is that maximum finding is no parallel operation. (However, there are some parallel alternatives to maximization [Reggia, 1985, Chun *et al.,* 1987].)

Note: The above criticism concerns only feature-discovery properties of the competitive model. There are further valuable properties of competition, e.g., noise reduction or contour enhancement, not covered by the alternative backward-inhibition model below.

3 Backward inhibition

The drawbacks of the above competitive model result from its lack of capability to discover finer features simultaneously with coarse ones. This seems to be the very nature of competition. This section presents a concept of "backward inhibition" which copes with this problem.

3.1 Backward inhibition concept

The basis for my further reflections is the idea that more subtle features can be discovered only if strong features overshadowing them are suppressed in some way.

The implementation of this idea is very simple. After a strong feature y_i has been successfully learned, it is suppressed in the input for some time. The optimal way to suppress a feature in the next input vector is to isolate its component orthogonal to the suppressed feature, i.e., to subtract the orthogonal projection of input vector x on the feature weight vector wj. Since the feature to be suppressed is represented by wj with |wj| = 1, we get

$$\mathbf{xx} = \mathbf{x} - [(\mathbf{w}_j'*\mathbf{x})/|\mathbf{w}_j|]*\mathbf{w}_j = \mathbf{x} - y_j*\mathbf{w}_j$$

with x - original input, xx - corrected input, y_i - feature unit. The suppressed feature unit will not be further activated, since

$$yy_i = \mathbf{xx}'*\mathbf{w}_i = \mathbf{x}'*\mathbf{w}_i - y_j*\mathbf{w}_j'*\mathbf{w}_j = y_i - y_j*|\mathbf{w}_j| = 0.$$

The suppression can be simulated by propagating inhibitory signal (in the size of feature activation) backwards to input units. E.g., if feature node *A* has been activated to activation level 2 via connections of strengths 0.5 and 0.3 to input nodes *B* and *C,* respectively, it inhibits nodes *B* and *C* by amounts -2*0.5=-l and -2*0.3=0.6, respectively.

These properties of backward inhibition will be used in the next two sections for constructing feature discovery algorithms.

3.2 Eigenvectors as features

The adaptive rule of Section 2.1 discovers only the dominant eigenvector of input correlation matrix. We can use backward inhibition principle of the previous section for furhter features to converge to further eigenvectors. So all eigenvectors of input correlation matrix can be found by successively applying backward inhibition.

Eigenvectors of a correlation matrix have some properties which make them intuitively good candidates for meaningful features:

a) They extract highly correlated input lines.

b) If all input lines are mutually independent, eigenvectors are unit vectors, while the dominant eigenvector corresponds to the input line with largest variance, or "the most important input feature".

c) If all input lines are completely correlated, the dominant eigenvector corresponds to the vector of input variances (or to the correlation matrix diagonal).

d) If the correlation matrix can be partitioned to several diagonal submatrices corresponding to mutually independent groups of highly correlated input lines, there are eigenvectors corresponding to each of these groups.

e) Eigenvectors of a correlation matrix (which is always a symmetric matrix) are orthogonal and thus independent.

f) They define a linear transformation which recodes the input as completely as possible, for a given dimensionality. In other words, they explain the variability of the input in the most compact way (in the sense of main components).

So the set of all eigenvectors seems to be an appropriate feature system.

3.3 Latency time algorithm

As stated in the previous section, all eigenvectors of input correlation matrix can be found by successively applying backward inhibition. The simplest method to do this would be to use a sequential algorithm. First, it learns the dominant eigenvector, i.e., it lets the

weight vector of the first feature node converge to the dominant eigenvector. Then, it inhibits this dominant feature in the input (via orthogonal projection, see above) so that the eigenvector with the second largest eigenvalue becomes the dominant eigenvector of such a modified input, and can be in turn discovered by some other feature node, and so on until all eigenvectors have been discovered. It is, in fact, a Gram-Schmidt process for evaluation of orthogonal projections [Kohonen, 1984, page 165]. The ordering of features corresponds to descending ordering of eigenvectors by the absolute value of corresponding eigenvalues.

For all F from 1 to number of features:

learn F-th feature by the basic adaptive rule while backward-inhibiting the input by all features FF<F.

An obvious drawback (and cognitive inadequacy) of this algorithm is its sequentiality. This is, to a certain degree, an intrinsic characteristics of the problem since our adaptation rule always learns only the dominant feature. In other words, small features have to be enabled to come out of the shadow of large features by filtering large features out. This can occur only sequentially. However, control can be made parallel (or better to say local in each feature node) by the following extended activation rule applied simultaneously to all features:

Step 1: Activate feature nodes by the basic adaptive model activation rule.

Step 2: If the weights of a certain feature node have reached a (temporarily) stable state, this feature node performs backward inhibition of input nodes for a certain number of cycles, the latency time.

The learning rule remains identical to that of the basic adaptive model (learning of a latent feature node is suspended during its latency time).

For this algorithm, a criterion of weight stability is needed. A simple test of weight increment 8w for being zero, or very small, is not appropriate since input is inherently varying and thus weights never become completely stable. So an average over recent periods has to be used. I have used an exponentially weighted average of weight increment 8w over past cycles *Edwl Edwabs* (a ratio of average increment and average absolute value of increment) with *Edw* = *p*Edw* + (7-p)*8w and *Edwabs* = *p*Edwabs* + (V-p)*|8w|. This ratio converges to zero if positive increments are periodically traded-off by negative increments while their absolute values may be relatively large (depending on the a-coefficient of the adaptive rule).

This feature discovery model is superior to the competitive *one* in several aspects:

1) Eigenvectors of a correlation matrix are orthogonal (since a correlation matrix is always symmetric) and thus independent.
2) The grain of feature discovery can be controlled by latency time parameter. If latency time is short, strong features are rapidly learned while there is not enough time for weak features to be discovered. If the latency time is long, week features are discovered, too.
3) The representation of input by the feature layer is distributed (see the following example).
4) The eigenvectors being ordered by their eigenvalues, they represent a certain implicit hierarchy, as illustrated by the following example.

Example 1: Let us have 8 input patterns (see Figure 2) represented by 16 binary input lines. (For symmetry, -1 are substituted for zeros.) Although theoretically 16 features could be found, we will be satisfied with 6 features. Features discovered by backward inhibition algorithm are presented in Table 1 as they are contained in the patterns (the activation value of each feature multiplied by 10 is given in each feature column). For easier orientation, an abridged code is given in the middle column. Symbol meanings are: + for feature activation > 0.7, - for feature activation < 0.7, o otherwise.

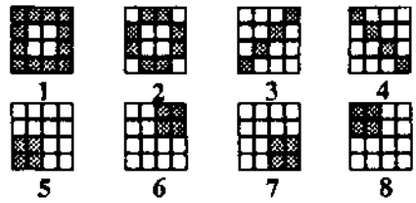

Figure 2. Example: Input patterns and discovered features.

Nr.	Pattern	Code	Feature 1	2	3	4	5	6
1	+++++−++−+++++	o−oo+o	−1	−33	2	2	23	5
2	−++−+−++−−+−++−	−+oo	−10	−26	−10	26	−5	0
3	−+−+−+−+−	o+ooo−	0	28	−2	−2	4	−25
4	+−+−+−+−+	o+ooo+	1	28	−1	−2	6	25
5	−−−++−−++−−−	++o+o−	22	15	−5	22	4	−19
6	−++−++−−−−−−	−+−oo−	−31	15	−8	0	4	−19
7	−−−−−−++−−++	−+++o+	−9	17	20	21	1	19
8	++−−++−−−−−−	o+−oo+	1	12	−32	1	2	20

Table 1. Example 1: Extracted features.

A "classification hierarchy", formed by descending order of eigenvalues is given in Figure 3. As we can see, dominant Feature 2 partitions the paterns to "diagonally oriented" (+) and "margin-oriented" (-). Those diagonally oriented are partitioned by Feature 6 to "bottom right - top left" (+) and "bottom left - top right" (-). Features 1 and 3 measure "balancedness" of diagonal patterns.

This hierarchy is not explicit in the feature vector. However, an implicit encoding is all that is needed for input into a further associative network. Some associations can depend only on strong features like "diagonal" vers, "margin-oriented", encoded by a single feature cell F2 while weaker ones like diagonal

orientation are encoded by F2 and F6 together, yet more special features by adding a further feature cell, etc., so that the actual encoding corresponds to the "path" in the hierarchy tree.

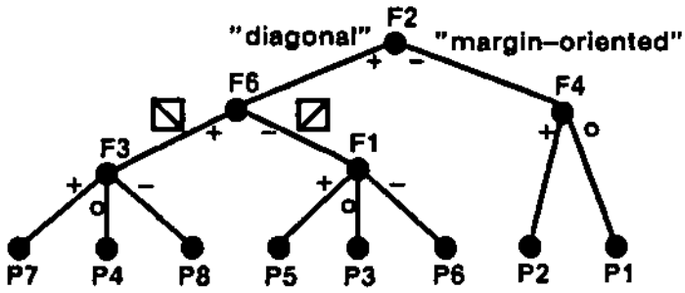

Figure 3. Example 1: Classification hierarchy.

The feature system can be used to classify novel patterns, as illustrated by the following example.

Example 2: Four novel patterns have been experimentally classified by the system (see Figure 4 and the Table 2). Note that Pattern 9 has been classified as "bottom right - top left" and "balanced" but undecided if "margin-oriented" or "diagonal". Pattern 10 is "margin-oriented", Pattern 11 "diagonal" and "balanced" with undecided direction.

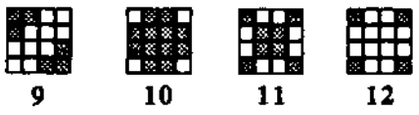

Figure 4. Example 2: Classification of novel patterns.

Nr.	Pattern	Code	Feature 1	2	3	4	5	6
9	++–+–––+–++	ooo+++	–5	–2	–5	13	11	26
10	–++++++++–+–	o–oo–o	–0	–23	0	2	–33	–1
11	+–+–++–++–+–+	+++–oo	10	26	10	–26	5	0
12	+–+–––––+–+	o+oo+o	0	23	–0	–2	33	1

Note: With short latency times, only approximations of eigenvectors are found. Since the "residue" after suppressing "strong" eigenvectors may then substantially differ from the exact one, features corresponding to "weak" eigenvectors may substantially differ from exact eigenvectors, too. However, in all cases tested they showed very high orthogonality, so that good properties of features discovered were preserved.

3.4 Competitive algorithm

Backward inhibition can also be used to improve grain control of the competitive model of Section 2.2. After a feature unit *y* has won the competition and its weights have incrementally learned, the feature represented by its weight vector wj is (partially) suppressed for some time:

xx = x - *b*yi*v/i*

with *0 < b < 1.* Parameter *b* controls the extent of suppression and thus the grain of feature discovery. Because of competition features do not correspond to eigenvectors.

Although this model is difficult to treat mathematically, it can be supposed (and has been verified experimentally, see Example 3) that even weak features get their chance to attract a weight vector so that the grain of feature discovery will be refined.

Example 3: To illustrate grain control, a set of five input vectors has been taken [3,1,0,0,0], [1,3,0,0,0], [0,0,3,1,1], [0,0,1,3,1], and [0,0,1,1,3]. Their correlation matrix is

$$\begin{bmatrix} 10, & 6, & 0, & 0, & 0 \\ 6, & 10, & 0, & 0, & 0 \\ 0, & 0, & 11, & 7, & 7 \\ 0, & 0, & 7, & 11, & 7 \\ 0, & 0, & 7, & 7, & 11 \end{bmatrix}$$

Five learning trials have been made with five and ten feature units and *b = 0.0* to *0.9 (b = 0.0* is equivalent to the original model without backward inhibition). Average numbers of active features are given in the table below.

b	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
5 units	2.8	2.8	2.8	2.8	2.8	3.0	4.2	4.6	4.6	5.0
10 units	3.2	3.2	3.2	3.6	3.8	4.6	4.8	–	–	–

Table 3. Example 3: Average numbers of active features.

The 10-unit case showed poor convergence for *b>0.6,* so corresponding results are not presented. It can be seen that basic competitive algorithm partitioned the patterns typically into three groups, no matter if there were five or ten feature units. The backward-inhibition-based algorithm has found, by contrast, three to five groups (five being the maximum since there are only five different input vectors), depending on the backward inhibition strength *b.*

4 Conclusion

Backward inhibition in a two-layer connectionist network can be used as an alternative to, or an enhancement of, the competitive feature discovery model.

The noncompetitive backward inhibition algorithm discovers features in the form of input correlation matrix eigenvectors and thus satisfies the requirements of independence, controllable grain, and distributed representation. Since the eigenvectors can be ordered by the absolute value of the corresponding eigenvalues, they form a certain "feature hierarchy".

The competitive backward inhibition algorithm operates analogically to basic competitive algorithm, but provides means for controlling the grain of feature discovery by inhibition parameter.

The advantage of backward inhibition over traditional numeric methods of cluster and variance analysis is their parallel implementation in a neural network and immediate construction of distributed representation which can be used for further processing, e.g., in a supervised-learning model.

References

[Ackley *et al,* 1985] D.H. Ackley, G.E. Hinton, and T.J. Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive Science* 9:147-169, 1985 .

[Anderson, 1986] J.A. Anderson. Cognitive capabilities of a parallel system. In E. Bienenstock *et al.* (Eds.), *Disordered Systems and Biological Organization.* Springer-Verlag, Berlin, 1986.

[Ballard, 1987] Ballard, D.H.. Modular learning in neural networks. In *Proceedings of the National Conference on Artificial Intelligence,* pages 279-284, Seattle, 1987.

[Chun *et al,* 1987] H.W. Chun, L.A. Bookman, and N. Afshartous. Network regions: alternative to the winner-take-all structure. In *Proceedings of the Tenth International Conference on Artificial Intelligence,* pages 380-387, Milano, 1987.

[Dalenoort, 1987] G.J. Dalenoort. Development of concepts. *Cognitive Systems* 2(1): 123-140, 1987.

[Fukushima, 1980] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics* 36:193-202, 1980.

[Grossberg, 1982] S. Grossberg. *Studies of Mind and Brain.* D. Reidel Publishing Co.. Dordrecht, 1982

[Grossberg, 1987] S. Grossberg. Competitive learning: From interactive activation to adaptive resonance. *Cognitive Science* 11:23-63, 1987.

[Hinton and Lang, 1985] G.E. Hinton and K.J. Lang. Shape recognition and illusory conjunctions. In *Proceedings of the Ninth International Conference on Artificial Intelligence,* pages 252-259, Los Angeles, 1985.

[Hinton *et al,* 1986] G.E. Hinton, J.L. McClelland and D.E. Rumelhart. Distributed Representations. In D.E. Rumelhart and J.L. McClelland (Eds.), *Parallel Distributed Processing,* Volume 1. MIT Press, Cambridge, 1986.

[Hopfield, 1982] J.J. Hopfield. Neural networks with emergent collective computational abilities. In *Proceedings of the National Academy of Sciences USA,* pages 2554-2558, 1982.

[Hrycej, 1987] T. Hrycej. A mechanism for unsupervised discovery of input features, Technical Report PCS-AI-28, PCS GmbH, Munich, 1987.

[Kodratoff and Ganascia, 1986] Y. Kodratoff and J.-G. Ganascia. Improving the generalization step in learning. In R.S. Michalski *et al* (Eds.), *Machine Learning,* Volume 2. Morgan Kaufmann Publishers, Los Altos, 1986.

[Kohonen, 1984] T. Kohonen. *Self-Organisation and Associative Memory.* Springer-Verlag, New York, 1984.

[Le Cun, 1986] Y. Le Cun. Learning process in an asymetric threshold network. In E. Bienenstock *et al* (Eds.), *Disordered Systems and Biological Organization.* Springer-Verlag, Berlin, 1986.

[Lebowitz, 1985] M. Lebowitz. Classifying numeric information for generalization. *Cognitive Science* 9, 1985.

[Quinlan, 1982] J.R. Quinlan. Semi-autonomous acquisition of pattern-based knowledge. *Machine Intelligence* 10:159-172 , 1982.

[Reggia, 1985] J.A. Reggia. Virtual lateral inhibition in parallel activation models of associative memory. In *Proceedings of the Ninth International Conference on Artificial Intelligence,* pages 244-248, Los Angeles, 1985.

[Rumehart *et al,* 1986] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representation by error propagation. In D.E. Rumelhart and J.L. McClelland (Eds.), *Parallel Distributed Processing,* Volume 1. MIT Press, Cambridge, 1986.

[Rumelhart and Zipser, 1985] D.E. Rumelhart and D. Zipser. Feature discovery by competitive learning. *Cognitive Science* 9:75-112, 1985.

[Stepp and Michalski, 1986] R.E. Stepp and R.S. Michalski. Conceptual clustering: inventing goal-oriented classifications of structured objects. In Michalski, R.S. et al. (Eds.), *Machine Learning,* Volume 2. Morgan Kaufmann Publishers, Los Altos, 1986.

[von der Malsburg and Bienenstock] C. von der Malsburg and E. Bienenstock. Statistical coding and short-term synaptic plasticity: A scheme for knowledge representation in the brain. In E. Bienenstock *et al* (Eds.), *Disordered Systems and Biological Organization.* Springer-Verlag, Berlin, 1986.

[Winston, 1986] P.H. Winston. Learning by augmenting rules and accumulating censors. In Michalski, R.S. et al. (Eds.), *Machine Learning,* Volume 2. Morgan Kaufmann Publishers, Los Altos, 1986.