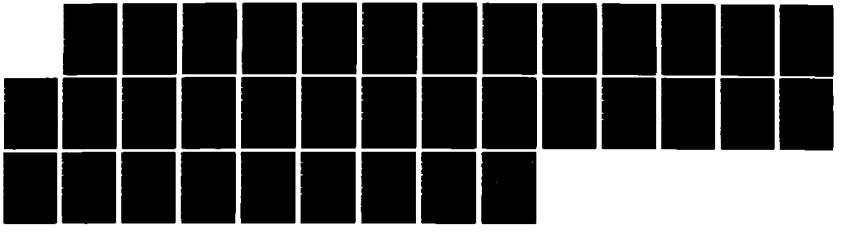
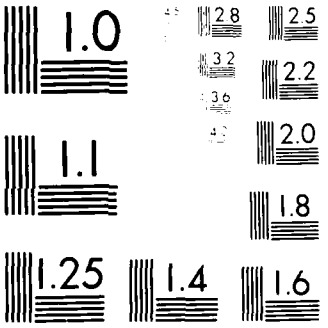
AD-A170 349 CONTEXT-FREE PARSING IN CONNECTIONIST NETWORKS(U) 1/1
ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE M FANTY
30 NOV 85 TR-174 N00014-84-K-0655

UNCLASSIFIED F/G 9/2 NL





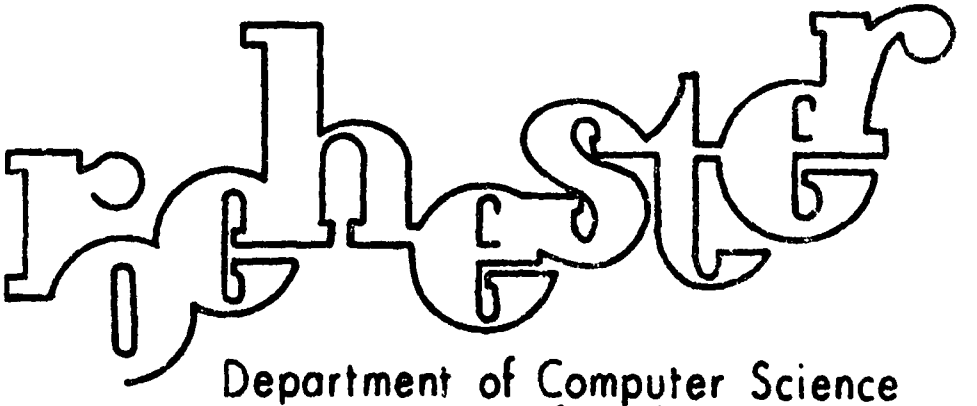
(12)

AD-A170 349

Context Free Parsing in Connection Networks

Mark Fanty
Computer Science Department
The University of Rochester
Rochester, NY 14627

TR174
November 30, 1985



Department of Computer Science
University of Rochester
Rochester, New York 14627

(12)

AD-A170 349

Context Free Parsing in Connectionist Networks

Mark Fanty
Computer Science Department
The University of Rochester
Rochester, NY 14627

TR174
November 30, 1985

Department of Computer Science
University of Rochester
Rochester, New York 14627

Context-Free Parsing in Connectionist Networks

Mark Fanty
Computer Science Department
The University of Rochester
Rochester, NY 14627

TR174
November 30, 1985

Abstract

This paper presents a simple algorithm which converts any context-free grammar (without ε-productions) into a connectionist network which parses strings (of arbitrary but fixed maximum length) in the language defined by that grammar. The network is fast and deterministic. Some modifications of the network are also explored, including parsing near misses, disambiguating and learning new productions dynamically.

This work was supported by an Office of Naval Research grant number N00014-84-K-0655

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR174	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Context-Free Parsing in Connectionist Networks		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Mark Fanty		8. CONTRACT OR GRANT NUMBER(s) N00014-84-K-0655
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department University of Rochester Rochester, NY 14627		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA/1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE November 30, 1985 13. NUMBER OF PAGES 30
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

parallel parsing
connectionist networks
context-free grammars

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper presents a simple algorithm which converts any context-free grammar (without ε-productions) into a connectionist network which parses strings (of arbitrary but fixed maximum length) in the language defined by that grammar. The network is fast and deterministic. Some modifications of the network are also explored, including parsing near misses, disambiguating and learning new productions dynamically.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

1. Introduction

My goal in designing a connectionist[1] parser was to be able to build, in a systematic way, for any context-free grammar, a network which will parse strings in that grammar (within a length restriction). The network must represent the parse tree for the input when finished, and must clearly indicate when there is no parse. Most important, I wanted the network to be deterministic and completely general. My goal is not cognitive modeling per se, but to provide a powerful technique which could prove useful for natural language understanding and other connectionist applications. Context-free grammars have proven very useful to Computer Scientists. The existence of a fast, simple and relatively efficient connectionist parser may well be of some importance to Cognitive Scientists working with connectionist models.

Several other connectionist parsing schemes have appeared recently. Certain ideas can be found in all of them (including mine). They all parse context-free grammars, using individual units to stand for the terminal and nonterminal symbols. Given a production such as **S → NP VP**, there are excitatory connections from **NP** and **VP** nodes to **S** nodes which can provide bottom-up evidence for the presence of an **S**, as well as excitatory connections from **S** to **NP** and **VP** providing top-down feedback. When a parse completes, the active units and their connections form a structure isomorphic to the parse tree for the input. I will at times refer to units in a parsing network as parents or children accordingly.

The work of Cottrell (1985) and Waltz and Pollack (1985) encompasses natural language understanding in a much broader sense than syntactic structure alone. Waltz and Pollack do not even attempt to build a general purpose parsing network. Their network is custom built for each input. Cottrell's networks are more general. He even has a program to build the networks from an input grammar. Unfortunately, the network does not always find the correct parse.[2]

In both models, contradictory interpretations of the input are mutually inhibitory. Only one should be on when the network stabalizes. Which one depends on how much semantic and syntactic support each receives from the rest of the network. A parse proceeds by activating the input, e.g. the node for "the" in position one, the node for "man" in position two, etc., and letting the network run until it settles into a consistent configuration representing the parse of the input with all ambiguities resolved.

The work of Selman and Hirst (1985) is nearer my own in ambition: their goal is a general purpose parsing scheme which will perform correctly for any input. They do not consider outside influences, such as semantics. They use mutually inhibitory binder nodes to connect a nonterminal node to all the subtrees it might dominate, each binder representing a different

[1] For an introduction to the connectionist paradigm, see Feldman and Ballard (1982).

[2] This can be an advantage from the viewpoint of cognitive modeling if the errors made resemble those made by humans.

A-1

production of the nonterminal. Likewise, binder nodes connect a node to all nodes which might dominate it. They use a variation of the Boltzmann machine (Hinton and Sejnowski, 1983) computational scheme. The units representing the input are clamped on. The others execute asynchronously, turning on or off probabilistically based on how much excitation and inhibition they are receiving. Simulated annealing (Kirkpatrick, Gelatt, and Vecchi, 1983) is used to settle the network into a state where the active units are mutually reinforcing to a large degree. The best possible state is one which represents a legal parse. In order to reach this optimal state with high probability, the network must settle gradually. Selman and Hirst used 24,000 updates for each unit.

My network is deterministic, fast, guaranteed to work for all inputs of any context-free grammar, and conceptually very simple. It is a simple, exact connectionist solution to the computational problem of parsing Contradictory parses do not inhibit each other (but see section three); all possible parses proceed in parallel. This requires a large number of units – typically tens or hundreds of thousands (see below).

The remainder of this paper is organized into four parts Section two describes the network in detail, giving an algorithm for generating it from a given CFG. An exact analysis of the network's complexity is presented, along with some ways of trimming its size. Section three touches on some methods of disambiguating, i.e. choosing a unique parse tree for ambiguous input. Section four gives a cursory account of parsing near-miss input, i.e. input which is almost grammatical. Section five gives a detailed description of how productions can be learned dynamically in some circumstances. These latter sections are intended to explore the flexibility of the parser as well as test some connectionist learning techniques. They do not provide an account of language acquisition.

All networks have been implemented using the Rochester Connectionist Simulator (Fanty & Goddard, 1986). Simulation traces are included below.

2. The Network

2.1 Structure and function

The strategy used closely parallels that of the CYK parser (Hopcroft & Ullman, 1979). The network contains units representing the terminals and nonterminals of the grammar (several for each, in fact) as well as match units which will be explained below. The units are best thought of as organized into a table, with the columns representing starting positions in the input string, and the rows representing lengths. There is a unit for each terminal symbol in each position of row one. There is a unit for each nonterminal at every position in the table (potentially; see section 2.2). Terminal units are activated by some outside source and represent the input to the parser. A nonterminal unit will become active if other units representing the right-hand side of one of its productions, and having appropriate starting positions and lengths, are active. The parse proceeds in a bottom-up fashion. Figure 1 illustrates the parsing of the string **aabbb** for the grammar shown. The terminal symbols are activated as input. The **A** unit in the first row becomes active because of input from the **a** unit in the same position and the production **A**→**a** The **A** unit in (2,1) – row two, column one – becomes active

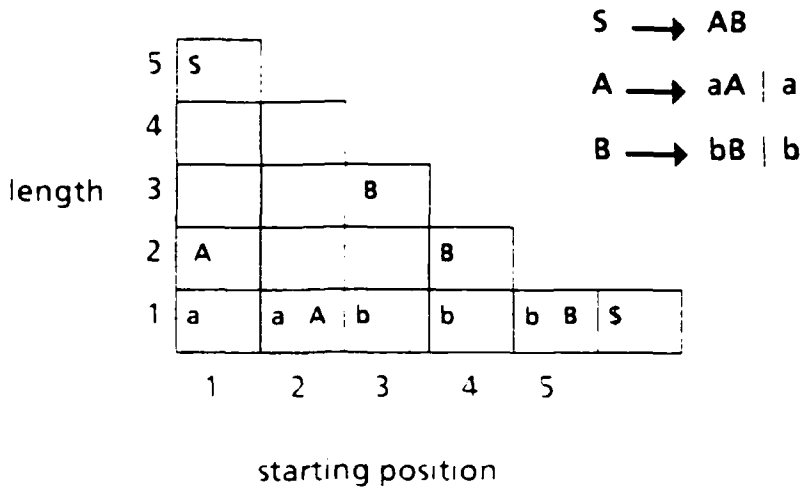


Figure 1
Parsing the input **aabbb**. Only the relevant terminal and nonterminal units are shown

because of input from both the **a** unit in (1,1) and the **A** unit in (1,2) and the production **A→aA**. Similarly for the **B** units. The **S** unit becomes active because of input from **A** in (2,1) and **B** in (3,3). Because there is an active start symbol in column one whose length is the same as the length of the input, the input is accepted. In order to mark the end of the input, a special **S** unit becomes active at the end of the input string.

The active units represent the parse tree. Of course, there will in general be many units which become active but don't represent a final parse tree. In the above parse, there will be an active **A** unit in (1,1), for example. This will not affect the ability of the network to recognize, but a second, top-down, pass of activity is necessary in order to pick out only those units which participate in a complete parse – if the string is ambiguous, more than one parse will stay active (see section three for possible modifications). The top-down pass works as follows: when an active unit representing the start symbol beginning in position one and of length n receives input from a **S** symbol in position n + 1, it becomes hyperactive. This unit is the root node of all parse trees. It passes this hyperactivity down to all units which form part of one of its completely recognized productions They pass the activity down in turn until it reaches the bottom. Only the hyperactive units represent a parse. In order to better distinguish the two levels of activity, a unit activated during the first bottom-up pass will be called *primed*, and a unit active after the the final pass will be called *on*. In our simulations, *primed* units have a potential and output of 5; *on* units have a potential and output of 10.

A more detailed account of how the network works follows. There are three kinds of units: nonterminal units, terminal units and match units. Each unit has two sites, or locations, where an incoming connection might be made. One is for bottom-up input. Enough input to this site will prime the unit. The other site is for top-down input. Every pair of units with a bottom-

up link between them also has a top-down link. If a *primed* match or nonterminal unit receives input to the top-down site from an *on* unit, it will turn *on*. The site functions are described below, and are given exactly in appendix two.

The terminal units are the simplest. They are *primed* by some outside source and represent the input to the parser. They are all on row one because they must be of length one. One per starting position should be *primed* in order to represent an input, although more than one might be activated in the case of input ambiguity (see section three). The **S** units are a special kind of terminal unit. One should be turned *on* in the position following the input.

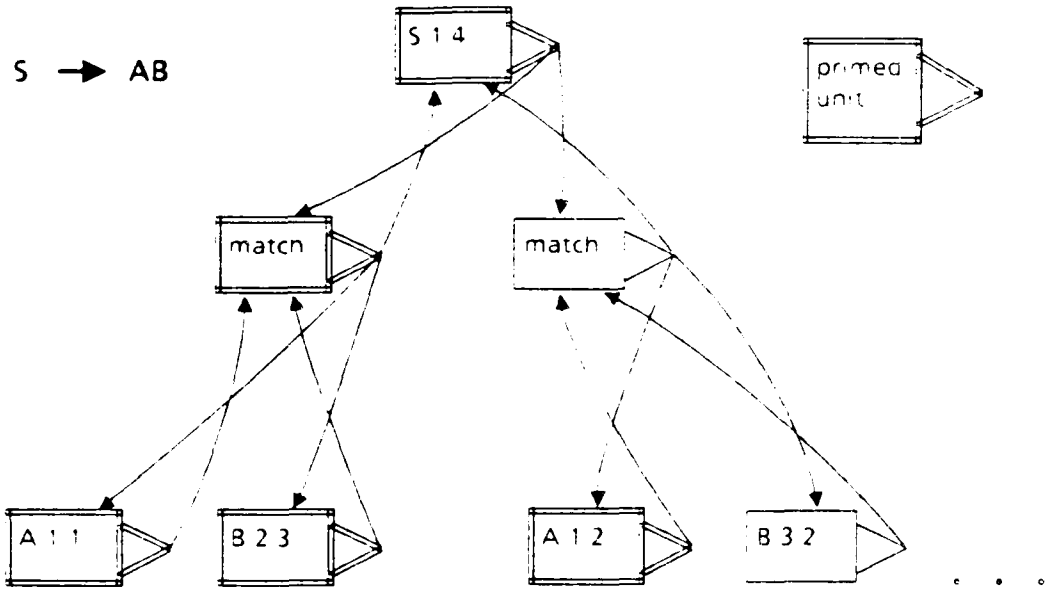


Figure 2
A 1 2 is the nonterminal unit representing **A**, starting at position 1, and of length 2. There is a match unit for each combination of lengths for each production.

The match units are used to represent the various instances of productions for nonterminal units. They receive bottom-up inputs from units representing the symbols on the r h s of their production. The starting positions and lengths of these units must be consistent with the nonterminal unit being served and with each other. For each nonterminal unit there is one match unit for each allowable combination of lengths for each production (see Figure 2). The bottom-up links to the match units are weighted so that all the connected units must be *primed* before the match unit becomes *primed*. The bottom-up inputs to a match unit are processed with a filtered-sum function. The sum of the inputs is taken, but each input is

allowed to contribute at most some fixed amount. This seems the most natural and flexible way of fixing the following problem.

Suppose match unit X represents a production with three nonterminals on the r.h.s. Two of the three nonterminal units are *primed*, giving an input to the unit of (with weights of 2/3 on the links)

$$5 * \frac{2}{3} + 5 * \frac{2}{3} + 0 * \frac{2}{3} = 6.67$$

which is below the threshold of ten, as desired. Now suppose the first nonterminal unit turns *on* because of top-down activation from somewhere else. The input to the match unit is now

$$10 * \frac{2}{3} + 5 * \frac{2}{3} + 0 * \frac{2}{3} = 10$$

which causes it to become *primed* falsely. This is prevented by limiting each bottom-up input's influence to 5 * weight.

A nonterminal unit receives bottom-up input from its match units. If any of them become *primed*, this means one of the productions has been realized, so the nonterminal unit becomes *primed*. A nonterminal unit responds very simply to bottom-up input from its match units: if any are *primed*, it becomes *primed* and provides bottom-up input to match units above it.

If the priming eventually reaches a start-symbol nonterminal unit in column one, row *n* and the input is of length *n*, then there is a parse of the input. This root unit will be receiving "top-down" *on* input from the **S** unit in row *n* + 1 used to mark the end of the input. This causes it to turn *on*. It now provides top-down *on* input to its match units. Any which were *primed* turn *on*, and provide top-down *on* feedback to all connected units. In this way the parse tree(s) will be turned *on* from the top down. In order to achieve correct behavior, the units must respond to top-down input in the following way. If the unit is already *primed* and the top-down input is at the *on* level, then the unit turns on. If the unit were not required to first be *primed*, then an *on* nonterminal unit would turn *on* all its match units, even though the production instance they represent may be quiet. Match units have only one input to their top-down site, so they can simply respond to it in a thresholded manner. Nonterminal units will typically have many inputs to their top-down site. Simply summing these inputs would result in incorrect behavior, as input from two *primed* match units will have the same sum as input from a single *on* match unit. To differentiate the two cases, nonterminal units take the maximum input as the value of the top-down site. This must be *on* to have any effect.

When *primed* terminal units receive *on* input at their top-down site, they turn *on* as well. It is possible for the network to detect when the parse is complete by sensing the presence of an *on* terminal symbol in every column up to the length of the input. If we exclude productions with a r.h.s. of

length one, the parse will complete in at most 4*input-length steps. The network could reject input by timing out.

The network can be turned off by taking away the external input to the terminal units (including **S**). This will remove bottom-up and top-down activation from the network: the source of all bottom-up activation is the terminal units, the source of all top-down activation is the **S** unit acting through the root node.

2.2 Network construction

A program to build a network for a given context-free grammar has been written in LISP and tested using the Rochester Connectionist Simulator (Fanty, *forthcoming*). The algorithm appears in the appendix in a pseudo language. The strategy is to work bottom up, first placing each terminal in each position of row one. Since the productions of a unit on row *n* depend only *on* units in rows $< n$, it is possible to build the network in a single bottom-up pass through the table. For each combination of lengths of each production, the appropriate units are looked up in the table. If they exist, a match unit for that production is created and the appropriate links made. If none of the productions of a nonterminal in some location are possible, the nonterminal unit is not made. If, for example, the nonterminal **B** could only generate strings of length three or more, then no nonterminal units of the form B.*n*.1 or B.*n*.2 would be created, and there would be no match units in charge of productions looking for such units.

ε-productions are not allowed, which is not too limiting, as a grammar with ε-productions can always be translated to one without. They could easily be added if desired. If there are productions of the form $X \rightarrow Y$, where Y is a single nonterminal, then Y units must be processed before X units on each row.

2.3 Complexity of the network

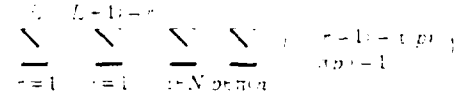
In order to facilitate the discussion, I introduce the following.

L = maximum length of input string
T = set of terminals in the grammar
N = set of nonterminals in the grammar
$n(n)$ = set of productions of nonterminal n
$v(p)$ = list of nonterminals in production p (may be repeats)
$t(p)$ = list of terminals in production p (may be repeats)

The number of nonterminal and terminal units is reasonable. In the worst case there are

$$\frac{L(L+1)}{2} N + LT$$

of them. The number of match units is significantly larger, however. In the worst case there are



match units. The sum represents, for each row, for each column, for each nonterminal at that location, for all productions of that nonterminal, all the possible combinations of constituent lengths in an expansion of that production. The quantity $(r\text{-}1)$-$|\pi(p)|$ *choose* $|v(p)|$-1 represents the number of different assignments of lengths to constituents which will sum to the desired length. The reasoning is as follows. How many ways can three constituents sum to ten? There are two boundaries between constituents to be chosen, and nine different locations to choose from: (10-1) *choose* (3-1). The term $|\pi(p)|$ appears in the first term because terminals reduce by one the space into which the nonterminals may expand.

For example, if $L = 15$, $N = 10$, $|\pi(n)| = 5$ and $|v(p)| = 3$ for all nonterminals and ignoring terminals, then the network will have at most 1,200 nonterminal units and 153,750 match units. This figure can best be improved be keeping $v(p)$ small (by putting the grammar in Chomsky normal form, for example). If $|v(p)| = 2$ in the above example, the number of match units will be at most 34,000. The size of the network is $O(n^{m+1})$, where n is the length of the network and m is the number of nonterminals on the right-hand side of the productions. Thus, it may be desirable in practice to limit the number of nonterminals on the right-hand side of a production to two. If this is done, the size of the network will be $O(n^3)$.

For the following grammar, taken from Selman and Hirst (1985), for inputs of length up to 15, the number of nonterminal units is 570 and the number of match units is 2,040.

S → NP VP	NP → determiner NP2
S → VP	NP → NP2
VP → verb	NP → NP PP
VP → verb NP	NP2 → noun
VP → VP PP	NP2 → adjective NP2
PP → preposition NP	

The number of units may be significantly smaller than the worst case if several nonterminal units cannot generate strings of all lengths (especially short lengths). The network described above would have 30 more nonterminal units and 210 more match units if **PP**s could generate strings of length one. A better example of the potential savings is provided be the following grammar which has the same number of nonterminals and productions as the preceeding grammar, but no nonterminal can generate strings of length one.

S → D B	D → c A
S → B D	D → A
B → a D	D → D C
B → B C	A → e d
C → b D	A → e A

For inputs of length up to 15, the number of nonterminal units is 458 and the number of match units is 1561. If nonterminal units for each nonterminal were placed at each location of the table, the number of nonterminal units would be 600 and the number of match units would be 2794.

The simulations run in $O(n)$ time, where n is the length of the input. Multiplying the execution time by the number of units (with the length of the r.h.s. of productions limited to two), we get a total of $O(n^4)$ computation steps. The serial execution time for parsing is $O(n^3)$ for straightforward parsing algorithms and about $O(n^{2.5})$ for the asymptotically best algorithm so far. This means that the network is bigger/slower than the best we could expect by about a factor of n. This is because the algorithm is not completely parallelizable. The activity must work its way up from the bottom serially.

2.4 Implementation using two-state linear threshold units

It is possible to build an equivalent network using only simple, single-site, linear threshold units which have only two levels of activity: *on* and off (1 and 0). The transformation is simple. Replace every unit in the original network with a pair of units – one for the bottom-up pass, and one for to the top-down pass (see Figure 3). The bottom-up unit being active corresponds to the

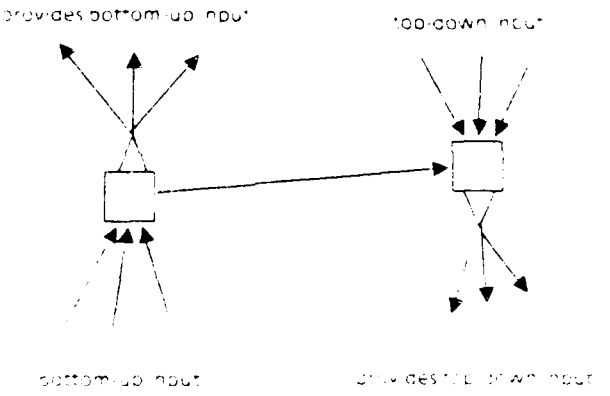


Figure 3
Two simpler units can do the job of one more complex unit

original unit being *primed*; the top-down unit being active corresponds to the original unit being on. The bottom-up inputs to match unit pairs will be weighted as before to require them all to be on. They no longer need to be filtered as they only come from other bottom-up units. The top-down unit does not need to take the maximum of its inputs as it only receives input from other top-down nodes. In order to require that the pair be *primed* before turning on, the weight on the link from the bottom-up unit to the top-down unit needs to equal the sum of the weights on the other inputs to the top-down unit, and the threshold on the top-down unit must be set so that it must receive input from the bottom-up unit and at least one top-down input.

When the parse completes, the active top-down units represent the parse tree.

2.5 Informal proof of correctness

In order to establish the correctness of a network as described above, we need only show three things. (Assume the grammar has no ε-productions. All units begin off except those terminal units representing the input, which are *primed*.)

(1) *A nonterminal unit* A.m.n *will become* primed *if and only if other terminal and nonterminal units spanning* m *through* m + n *in exactly the order of one of* A*'s productions are* primed.

We will say that these other terminal and nonterminal units satisfy one of *A*'s productions. *A* will become *primed* if and only if one of its match units becomes *primed*. These are the only units with connections to its bottom site, and only input to this site will cause an off unit to become *primed*. Each match unit corresponds to one of *A*'s productions. It has an input from a unit corresponding to each symbol in the production in such a way that positions *m* through *m + n* are spanned exactly. The weights on the inputs to the match units are such that they must all be *primed* in order for the input to exceed the threshold. Each input is filtered so that no single input can contribute more its share. Every possible satisfaction of each of *A*'s productions has a corresponding match unit.

(2) *When the input is of length* n, *the first nonterminal unit to turn* on *(if any do) will be* **S.1.n**, *and it will turn* on *only if it is first primed. (***S** *is the start symbol.)*

In order for a nonterminal unit to turn *on* it must first be *primed* and then receive *on* input to the top-down site. Since input from an *on* unit is required to turn another unit on, the first unit to turn *on* must receive top-down input from **S.*n* + 1.1**, the only unit *on* from the beginning. The only unit with such a connection is **S.1.*n***.

(3) *A nonterminal unit – other than the first to turn on, as in (2) – will turn on if and only if it was first primed and it is one of the units satisfying a nonterminal unit which turned on previously.*

Following (2), we need only show that a *primed* nonterminal unit will receive input to its top-down site from an *on* unit if and only if it helps satisfy some production of an *on* nonterminal unit. Except for input from the *on* **S** unit, which has just one connection to the root node, nonterminal units receive input to their top-down site only from match units to which they contribute. A match unit will turn *on* just when it was first *primed* (production satisfied) and its parent nonterminal unit turns on. All inputs to a *primed* match unit contribute to the satisfaction of its production instance.

2.6 Simulation results

A network for the grammar in section 2.3 was built and simulated with the following input: *det noun verb det adj adj noun*, which corresponds to

sentences such as *The man kissed the tall attractive woman*. The results of the simulation are given in table 1. Only units which have non-zero potential are shown. Match units are omitted in order to make the table more readable. After 26 steps, the network is stable. Those units with a potential of 10 represent the (unique) correct parse.

Table 1
Simulation of *det noun verb det adj adj noun*

Unit name	Potential after 1 step	Potential after 13 steps	Potential after 26 steps
det 1	5	5	10
noun 2	5	5	10
verb 3	5	5	10
det 4	5	5	10
adj 5	5	5	10
adj 6	5	5	10
noun 7	5	5	10
S 8	5	5	5
NP2 2 1	0	5	10
NP 2 1	0	5	5
VP 3 1	0	5	5
S 3 1	0	5	5
NP2 7 1	0	5	10
NP 7 1	0	5	5
NP 1 2	0	5	10
S 2 2	0	5	5
NP2 6 2	0	5	10
NP 6 2	0	5	5
S 1 3	0	5	5
NP2 5 3	0	5	10
NP 5 3	0	5	5
NP 4 4	0	5	10
VP 3 5	0	5	10
S 3 5	0	5	5
S 2 6	0	5	5
S 1 7	0	5	10

Table 2 shows the results of simulating *noun verb det noun prep noun prep det noun*, e g *John hit the man with Tom with a hammer*. This sentence is ambiguous in many ways. Notice the overlapping constituents, such as the **PP** from 5 to 9 (PP 5 5) and the **NP** from 3 to 6 (NP 3 4). The match nodes provide enough information to distinguish the various parses, but if the match nodes are invisible externally, the state of the network does not make

sense. In any case, it may be desirable to select only one parse. This is the topic of the next section.

Table 2.
Simulation of *noun verb det noun prep noun prep det noun*
(terminals not shown)

Unit name	Potential after 13 steps	Potential after 26 steps	Potential after 26 steps with disambiguation (section three)
NP2.1.1	5	10	10
NP.1.1	5	10	10
VP.2.1	5	5	5
S.2.1	5	5	5
NP2.4.1	5	10	10
NP.4.1	5	5	5
NP2.6.1	5	10	10
NP.6.1	5	10	10
NP2.9.1	5	10	10
NP.9.2	5	5	5
S.1.2	5	5	5
NP.3.2	5	10	10
PP.5.2	5	10	10
NP.8.2	5	10	10
VP.2.3	5	10	10
S.2.3	5	5	5
NP.4.3	5	5	5
PP.7.3	5	10	10
S.1.4	5	5	5
NP.3.4	5	10	5
NP.6.4	5	10	5
VP.2.5	5	10	10
S.2.5	5	5	5
PP.5.5	5	10	5
S.1.6	5	5	5
NP.4.6	5	5	5
NP.3.7	5	10	5
VP.2.8	5	10	10
S.2.8	5	5	5
S.1.9	5	10	10

3. Disambiguating

A lot of local disambiguating happens naturally because some interpretations do not participate in a complete parse tree and, thus, never turn *on*. This could account for word sense disambiguation in many cases. For example, to parse the sentence *The man walked on the deck*, both **noun** and **verb** would be *primed* in position six, but only **noun** would turn on, as there is no complete parse using the other interpretation

When the input is truly syntactically ambiguous, however, more than one parse tree will be *on* simultaneously. The parse can be made unambiguous by allowing only one match node (i.e. production) per nonterminal unit to remain active. The simplest way to do this is to order the match units of each nonterminal and add inhibiting links from each to those of lesser rank. Any inhibiting input from a superior match unit would be enough to prevent activation. Only the highest ranking *primed* unit would stay *primed*. The match units would need to be ranked not only according to which production of the grammar they represent, but also according to the combination of lengths of their production. The ranking would not need to be consistent throughout the network. One production could dominate another only for short lengths or towards the beginning,for example

The above scheme was not implemented; however, a different scheme was. In this scheme, each match unit inhibits all the other match units belonging to the same nonterminal node. An *off* match unit receiving inhibiting input will not become primed. Thus, this scheme prefers shallower parse trees.

Because the simulator used is synchronous, multiple match units will prime simultaneously whenever the subtrees have equal depth. When this happens, the inhibition must be gradual, or the match units will turn each other off. This will allow them all to come back on the following step and cycle in this manner indefinitely. The following behavior reliably yields a single winner. The inhibiting weights between the match units vary randomly between -0.5 and -1.0 exclusive. A primed unit receiving inhibition will lower its potential by an amount equal to the strongest inhibiting input. When the potential gets to zero, it turns off. When only one match unit is left, the lack of inhibition allows it to gain its full primed potential of five. For example, suppose two match units become primed at the same time and that match unit M1 inhibits M2 with weight −0.6 and M2 inhibits M1 with weight −0.65. The following is a trace of their behavior in one step increments (with minor arithmetic errors):

M1	5.0	1.75	0.45	0.0	0.0
M2	5.0	1.99	0.94	0.67	5.0

M2 has a higher potential after the first round of inhibition because it is more weakly inhibited. Now M2 is receiving inhibition equal to −0.6 * 1.75 and M1 is receiving inhibition equal to −0.65 * 1.99. M2's domination of M1 is increasing. There is no way for M1 to push M2 under 0 since its potential is less than M2's and its inhibition is less than its potential. With more than two match units on the interactions are more complicated. If the inhibiting inputs were summed, it would be possible for every match unit to go off after one

step and oscillate as described above. This is why the total inhibition is equal to the strongest single input.

If the inhibitory weights between match units were adjusted dynamically according to how often the match unit came on, then the network could learn to prefer more common interpretations. It might also be possible to use nonsyntactic information to affect the preferred parse with external contributions to the inhibition. This was not implemented.

The last column of Table two shows a simulation of the ambiguous sentence from the previous section using the disambiguation scheme just described. A single unambiguous parse tree results. No match units battled, as the ambiguities involved parse trees of different depths; the first on won by default. For example, in deciding between the two productions **VP → verb NP** and **VP → VP PP**, the latter always wins because its parse tree is shallower. I make no claims about the adequacy of this scheme. I present it as a demonstration of how disambiguation could be done in this model. Other strategies are possible as well.

4. Parsing near-miss input

It is sometimes desirable to give a reasonable parse of input strings which are not in the language defined by the grammar – ungrammatical but still understandable natural language utterances, for example. The parsing network described above is too rigid to do this effectively if some of the input is missing or there is extra input. But if the source of the ungrammaticality is simply the substitution of incorrect input of the same length as some correct input, then it can be made to do this by having match units become partially *primed* to an extent reflecting the closeness of the match between the input string and the production.

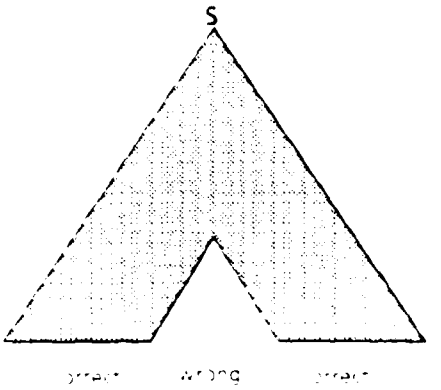


Figure 4
In a near-miss parse, the portion of the input which is grammatical is parsed

I implemented a network that extends the disambiguating network described previously. Match units whose productions are partially satisfied

may have a potential between 0 and 2. This range was chosen so that inhibition from a *partially-primed* match unit will always be less than inhibition from a *primed* match unit. The weights on inhibiting connections range from −.5 to −1.0. A *primed* match unit will always deliver inhibition of −2.5 or less; a *partially-primed* match unit will always deliver inhibition of −2.0 or more. *Primed* match units always inhibit *partially-primed* match units. *Partially-primed* match units compete with each other much as *primed* match units do (see section three), except the inhibition from a *partially-primed* match unit is not strong enough to prevent the *priming* of other match units. This is necessary, as a match unit may become *partially-primed* before one of its brothers becomes fully *primed*. The strongest *partially-primed* match unit may be beaten out by weaker *partially-primed* match units because of the random variation in the strength of the inhibiting weights.

Because inhibition from a *partially-primed* match unit does not prevent other match units from becoming *partially-primed*, care must be taken to prevent the re-*priming* of a *partially-primed* match unit which has just been defeated. Unless it subsequently receives additional input and becomes fully *primed*, a defeated *partially-primed* match unit will stay off.

The permissiveness of the network can be adjusted by setting a minimum input required for partial priming. In the simulations run, the cutoff was 3 (an input of 10 represented complete satisfaction of the production). The potential of a *partially-primed* match unit is equal to its total input divided by 5. Some of the key steps in a simulation of the ungrammatical input *det det noun verb noun* are given in Table three. The grammar used was the one

Table 3
Near-miss parse of *det det noun verb noun*
(Only relevant units shown.)

Unit	Potential after step							
	0	3	7	10	14	15	17	23 and after
det 1	5.0	5.0	5	5.0	5.0	5.0	5.0	10.0
det 2	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
noun 3	5.0	5.0	5.0	5.0	5.0	5.0	5.0	10.0
verb 4	5.0	5.0	5.0	5.0	5.0	5.0	5.0	10.0
noun 5	5.0	5.0	5.0	5.0	5.0	5.0	5.0	10.0
$ 6	5.0	5.0	5.00	5.0	10.0	10.0	10.0	10.0
NP2 3 1	0.0	5.0	5.0	5.0	5.0	5.0	5.0	10.0
NP2 5 1	0.0	5.0	5.0	5.0	5.0	5.0	5.0	10.0
NP 5 1	0.0	0.0	5.0	5.0	5.0	5.0	5.0	10.0
NP2 2 2	0.0	0.0	1.05	1.05	1.05	1.05	1.05	10.0
VP 4 2	0.0	1.05	5.0	5.0	5.0	5.0	10.0	10.0
NP 1 3	0.0	1.05	1.27	1.27	1.27	1.27	10.0	10.0
S 1 5	0.0	0.0	0.0	1.32	1.32	10.0	10.0	10.0

in section 2.3. Only the relevant units are included. As before, many units not in a complete parse will become *primed*. Even more become *partially-primed*. All the non-match units which eventually turn *on* are shown. The *partial-priming* of NP.1.3 after step 3 is due to input from det.1; after step 7, it (actually, its match unit) is also receiving some input from the *partially-primed* NP2.2.2 so its potential increases to 1.27. Eventually, the partial priming reaches the root node S.1.5. What happens next differs from previous networks. In this network, the end-marker, $.6, must have potential equal to 10 in order to provide sufficient top-down feedback to turn on the root node. It requires five steps to reach that level. This is to give complete parses a chance to reach the top. Once a *partially-primed* match node turns *on*, it is too late for a fully *primed* match node to inhibit it. When the parse completes, the section of the input which does not fit into the parse remains primed.

In a simulation with input *det det det verb det det det*, no parse completed. NP1.3 and VP4.4 were both partially primed, but their combined input was less than the threshold for S.1.7's match unit.

5. Learning new productions dynamically

The near-miss network described above has been modified to learn new productions dynamically. The circumstances under which it is capable of learning are depicted in Figure 5. After a near-miss parse, there will be a gap

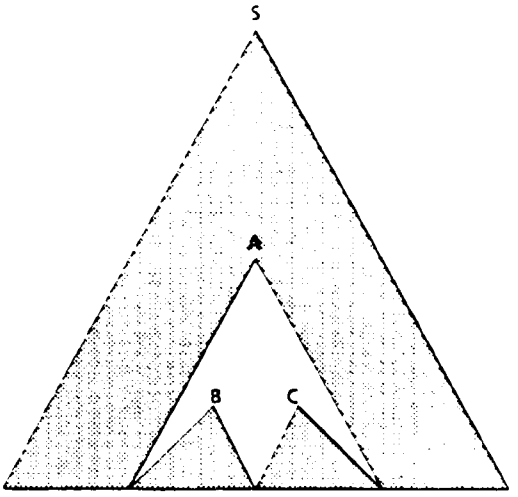


Figure 5
When the production **A → B C** is learned, the parse will be complete

in the parse tree where some constituent was "expected" but not found. If the gap can be parsed as one or two constituents, then a match node representing the new production will be recruited (Feldman, 1982) (**A → B C** in Figure 5).

This mechanism cannot explain the acquisition of a grammar from raw data. For one thing, no new nonterminals are learned. This is especially limiting given the restriction on production length (see below). It can sometimes account for new rules composed of known constituents. The real purpose of this section is to explore the flexibility of the parsing network. I do not claim to have an adequate mechanism for grammar acquisition.

5.1 Local learning

This section describes the recruitment of a match node to represent a production instance. The production will not be recognized elsewhere in the network. In order to make the network more tractable, the right-hand side of productions must be of length one or two. I will call such productions type one and two respectively. For any context-free grammar, there is a weakly equivalent grammar satisfying this restriction. Each bottom-up input to a match unit now goes to its own site. Accompanying each nonterminal unit is a single learn unit and some free match units which do not yet represent a production. Fixed match units represent production instances as before except for the presence of two bottom-up sites for type two productions. A nonterminal unit being turned *on* from above without having first been primed means an instance of the nonterminal is expected but not found. This turns the learn unit *on*, which enables the free match units. If the input in question can be parsed as one or two constituents, then some match unit will be recruited to represent this production instance.

The additional bottom-up sites are to enable free match units to detect when they are receiving bottom-up input from a potential production. Each free match unit can learn only production instances with some fixed combination of constituent lengths or division. For example, the free match unit in Figure 6 can learn productions with two constituents, the first of length four and the second of length six. Notice that any combination of one input to the bottom-right site and one input to the bottom-left site constitutes a legal production.

Figure 6 depicts the setup of a free match unit just before learning occurs. The free match unit is receiving bottom-up input to each site from exactly one unit. It has not yet primed because it is in a *free* state; it requires additional input from the learn unit before responding. The nonterminal unit B 3 10 is *on* but no match unit is primed. This will cause the learn unit to come *on*. The learn unit requires *on* input from B 3 10 and is inhibited by the match units. Once the learn unit comes *on*, the free match unit will become highly active briefly. I will call this state *excited*. It now inhibits the learn unit, whose job is done. If more than one input per bottom-up site had been active, the free match unit would not have responded.

Three changes now occur which transform the free match unit into a fixed match unit. First, the unit no longer enters the *free* state in which input from the learn unit is required. Second, the weights on all inactive bottom-up links are zeroed. The match unit now responds only to the pair learned. Third, the top-down links to the learned constituents are given positive weights. It is the need for this weight change which necessitates the special *excited* activity of a match unit which has just learned a production. Weight change occurs at the destination unit. The only way for B 3 4 and C 7 6 (see Figure 6) to know to increase the weight on the top-down links from the match unit is from this

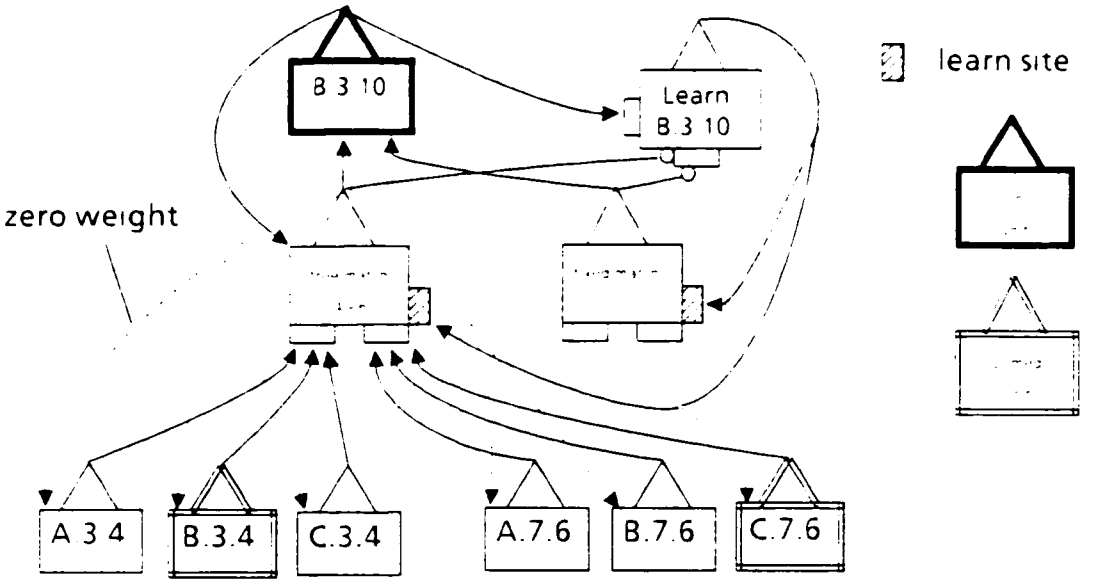


Figure 6.
Configuration of free match unit just before learning. The Learn unit is about to turn on.

level of activity. The other, *non-primed* units receiving top-down excited input do not change the weights. Because of the excited input, B.3.10, B.3.4 and C.7.6 become momentarily excited as well. Soon, all excited units settle to an *on* state and the parse completes. The situation after learning is shown in Figure 7.

It is possible for free match nodes in separate divisions to each learn a production at the same time. This ability is necessary for global learning in the next section. Since all match nodes inhibit each other, it is necessary to suppress this inhibition during learning. Because of this, all interpretations of ambiguous input will be learned if they are in different divisions and none will be learned if they are in the same division. If there is more than one match node representing a division, only one should learn a new production This can be achieved by ordering the free match nodes of a division and putting strong inhibitory connections from the earlier match nodes to the learn site of the later ones.

Table four shows a simulation of a local-learning network for the simple grammar

S → A B **A → a a** **B → b**

The production instance being learned is B.3.1 → b.1. After step 13, the near-miss parse has activated B.3.1 even though it was never primed. The learn unit and the free match unit take a couple of steps to come on. B 3 1 and, after step 15, the learn unit are decaying. If learning does not happen, they will turn off eventually. Notice that a.3 becomes excited two steps after the match unit does. The delay is caused by the need to increase the weight on

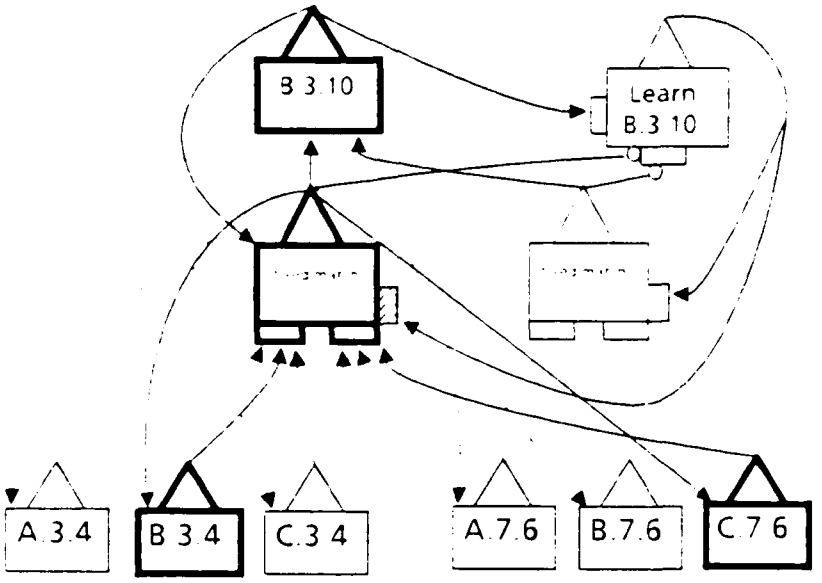


Figure 7
Configuration of match unit just after learning

Table 4
Demonstration of local learning.
(Only relevant units shown.)

Unit	Potential after step								
	12	13	14	15	16	17	18	19	20
a.1	5	5	5	10	10	10	10	10	10
a 2	5	5	5	10	10	10	10	10	10
a.3	5	5	5	5	5	5	15	15	10
$ 4	10	10	10	10	10	10	10	10	10
B 3 1	0	10	9	8	8	15	15	15	10
learnB 3 1	0	0	8	15	14	0	0	0	0
match	0	0	0	3	15	15	15	10	10
A 1 2	5	10	10	10	10	10	10	10	10
S 1.3	10	10	10	10	10	10	10	10	10

Table 5
Same input as in Table 4, but after learning
(Only relevant units shown.)

Unit	Potential after step				
	6	7	9	10	11
a.1	5	5	5	5	10
a.2	5	5	5	5	10
a.3	5	5	5	5	10
$.4	10	10	10	10	10
B.3.1	5	5	10	10	10
learnB.3.1	0	0	0	0	0
match	5	5	5	10	10
A.1.2	5	5	10	10	10
S.1.3	5	10	10	10	10

the link from the match node to a.3. Table five highlights a parse of the same input after learning.

5.2 Global learning

This section describes an extension to the above scheme which distributes a production instance learned locally throughout the whole network. Although the exact mechanism is different, the idea of using a central template to program other representations was inspired by McClelland's (1985) Connection Information Distributer. I will first give a high-level description of what happens, then provide a more detailed description of the implemented network. There is a single, global representation of each production. When learning occurs locally, the production learned is noted in the global store. After the network calms down, it enters a special learn state, which limits the spreading activation. When global learning occurs, the units involved in the production are activated throughout the network by the global representation in such a way that the local learning mechanism of the previous section burns in all the production instances.

When every instance of some production in the network is active and learning simultaneously, a great deal of care must be taken to insure cross-talk does not occur. A single unit, such as NP.3.4, may be both the parent and son for the same production. In order to distinguish the various roles a unit may play, a separate unit is used for each. To facilitate this, the grammar must be in Chomsky normal form, which means that all productions have the form
X → **Y Z** or **X** → **a**, where **X**, **Y** and **Z** are nonterminals and **a** is a terminal. Any context-free grammar can be converted to a weakly equivalent grammar in Chomsky normal form (Hopcroft & Ullman, 1979). For such a grammar, there are only three roles a nonterminal can play: parent, left-son and right-son. Accordingly, the job previously done by nonterminal units is now done by a

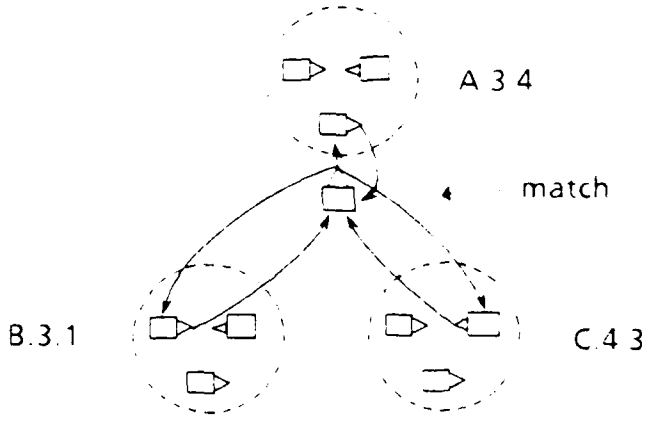


Figure 8
Production instance A.3.4 → B.3.1 C.4.3 with relevant links shown

trio of units, one for each role of the nonterminal. Figure 8 shows the setup for the production instance A.3.4 → B.3.1 C.4.3.

Normally, the three nonterminal units pass on activation. Bottom-up activation is spread from the match unit to the parent unit to the two son units which are connected to other match units. Top-down activation comes from one of the son units and goes through the parent unit to the match unit. However, the spread of activation between parent and son units for a nonterminal group is blocked when the network is in a global learn state. In the networks implemented, this is done by having the activation go through pass units which are inhibited by a GlobalLearnUnit (see Figure 10). When the network is not in a global learn state, it behaves just like the one described in section 5.1, disambiguation, near-miss parsing and local learning work the same way. The network has a central representation of productions comprised of four pools of units (Figure 9). The top pool has one unit for every nonterminal and represents that nonterminal as a parent. The bottom left (bottom right) pool also has one unit for every nonterminal and represents that nonterminal when it is the left-son (right-son). The terminal pool has one unit per terminal and represents that terminal as a son. To represent the production **A → B C**, a production unit will have two-way excitatory links to the **A** unit in the top pool, the **B** unit in the bottom left pool, and the **C** unit in the bottom right pool.

The global units are connected to nonterminal groups throughout the parsing network in the following way (see Figure 10). Each unit in the top pool is connected to the top site of the parent unit in all the nonterminal groups for that unit. When a unit in the top pool turns *on*, the bottom units in all the corresponding nonterminal groups turn *on*. Each unit in the bottom left pool is connected to the bottom site of the left son unit in all the nonterminal groups for that unit. When a unit in the bottom left pool turns *on*, the top left units in all the corresponding nonterminal groups become *primed*. There are similar connections from the bottom right pool to the right-son unit of each corresponding nonterminal group, and from the terminal pool to the terminal units.

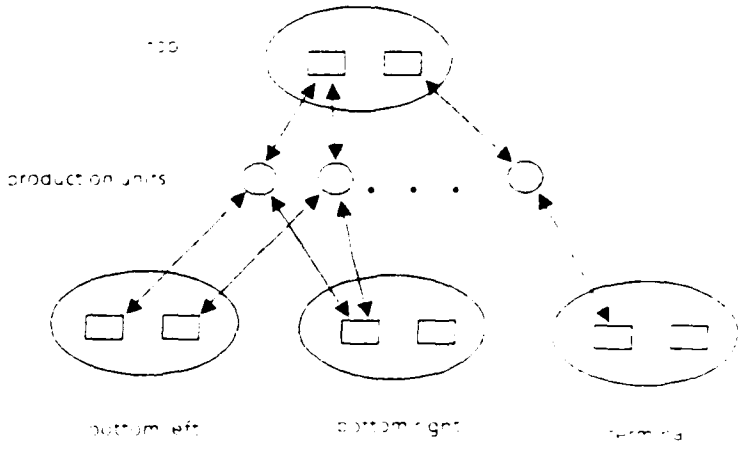


Figure 9
Global production templates

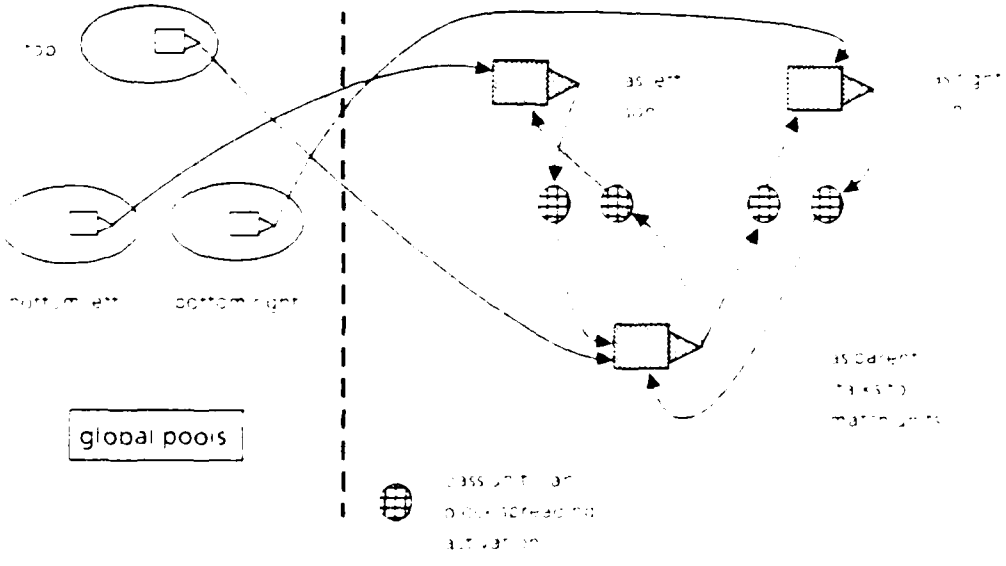


Figure 10
Nonterminal unit replaced by seven units in a global learning network.
Sample connections from global units are shown.

For each connection described in the preceding paragraph, there is a reciprocal connection to the units in the central pools. If a unit in a nonterminal group becomes *excited*, it will cause the corresponding unit in the central pool to become *primed*. If exactly one unit in the top pool and one in each bottom or one in the terminal pool are *primed*, a unique production is represented by the pattern of activation. This is how the global

production representation knows when local learning has occurred and what has been learned.

Here is what happens when learning occurs. The local learning mechanism causes some production to be learned. The parent and two children of the production instance learned become excited. In the parent nonterminal group, only the bottom unit becomes excited because the pass units in a nonterminal group do not pass on excitation. Similarly, in the left son nonterminal group, only the upper left unit becomes excited, and likewise for the right son. This causes the corresponding units in the central pools to become *primed*. (This does not seriously affect the parsing network.) The global production unit representing this production is now receiving activation from all three components and will become *primed* after two simulation steps. If more than one global production unit attempts to become *primed* at the same time, mutual inhibition will force them all to zero potential and global learning of these productions will not occur. This prevents the cross talk which would occur if two productions were turned *on* in the central template simultaneously. Once a global production unit becomes *primed*, it inhibits all other global production units. The components of the production in the global nonterminal and terminal pools become inactive as soon as they stop receiving *excited* input from the network. The *primed* feedback from the global production unit is not enough to keep them active. Self feedback keeps the production unit *primed*. It will not turn *on* until the parse is finished. It requires excitation from the GlobalLearnUnit, which does not turn *on* until the network has finished a parse (during simulations, this unit is turned on by hand).

When the parse completes, the terminal units are turned off and the network calms down. The GlobalLearnUnit is turned *on*. This causes the still *primed* production unit to turn on, which turns *on* the global units comprising the production. For example, if the production to be learned is **A** → **B C**, then the **A** unit in the top pool, the **B** unit in the lower left pool, and the **C** unit in the lower right pool will be *on*. This causes all bottom units in **A** nonterminal groups to turn *on* and all top left units in **B** nonterminal groups and all top right units in **C** nonterminal groups to become *primed*. Because the GlobalLearnUnit is inhibiting the pass units, activation will not spread within nonterminal groups.

The situation is just what is needed for local learning to occur all over the network. The bottom unit of the **B** nonterminal groups is *on*, but no fixed match unit is *primed*. For each division, there will be one free match unit receiving input from **B** as left son and **C** as right son. These productions will be learned. (This will not affect the global pools of units in their current state.) After a few steps, the global units become exhausted and turn off. They will remain quiescent for a few steps longer - enough to let the network calm down. When the network units lose input from these global units, they quickly die down.

In the network described so far global learning would not occur at the nonterminal group which learned the production locally. For example, suppose the local production instance A 4 3 → B 4 1 C 5 2 is learned. A little while later global learning takes place. B 4 2 and C 6 1 are *primed* and A 4 3 is on, but A 4 3 will not learn this production instance because B 4 1 and C 5 2 are also *primed*, which *primes* the now fixed match unit for this

production, which inhibits the local learn unit for A.4.3. This is avoided by having the GlobalLearnUnit inhibit the inhibition of local learn units. It has a strong positive connection to all local learn units at the inhibit site. We now must worry about redundant match units being learned. This is prevented by having fixed match units inhibit free match units with the same division. If the production being learned is already known for some division, then that fixed match unit will become *primed*, and no new match unit will be fixed. This is an extension of a mechanism already in place: free match units already inhibit other free match units (ranked lower) within the same division. These inhibiting connections remain even after the unit becomes fixed. All we need to add are inhibiting connections from match units which begin life fixed. With this addition, the new production can truly be learned globally. It also makes the network more robust, as described in the next section.

The network builder was programmed to make two unfixed match nodes per division per nonterminal group in order to test that no redundant productions were learned. Simulations for the following grammar worked correctly (the results are somewhat big for a table, so they will be summarized):

S → A B	**A → a**	**C → c**
B → B C	**B → b**	

First, the network was run with the input **b c a b**. This resulted in a near-miss parse (actually the miss was not too near). The production instance A.1.3→B.1.2 A.3.1 was learned. This caused the three units A.1.3.parent, B.1.2.lson and A.3.1.rson to become *excited*, which *primed* the global units A.Parent, B.Lson and A.Rson, which *primed* A->B.A, the global production unit for **A → B C**. A couple of steps later the *excited* network units calmed down to *on*. This caused all the global units except A->B.A to turn *off*. When the parse completed, the terminal units were turned *off* and the network was run until only A->B.A was left *primed* (about 12 steps). The GlobalLearnUnit was turned *on* by hand, which turned *on* A->B.A, which turned *on* A.Parent, B.Lson and A.Rson. *On* input from these units turned *on* all A.x.y.parent units and *primed* all B.x.y.lson and A.x.y.rson units. Learn units for all the A.x.y groups came *on*. The ones with length greater than one had free match units get excited, one per division, and learn their new productions.

The A.1.3 group had its fixed match unit for A.1.3→B.1.2 A.3.1 *prime* because of the production instance it had just learned. This did not prevent the learn unit from coming *on* because the GlobalLearnUnit was *on*. It did prevent the other free match unit with the division 2 + 1 from activating. One free match unit for the division 1 + 2 became excited and learned the production instance A.1.3→B.1.1 A.2.2. After a few steps, the global nonterminal units (but not the GlobalLearnUnit) become exhausted as do all the learn units. The network quickly calms down.

After learning, the network was tested with the input **b b a a b**, which requires requires the use of two new instances of the production **A → B A**. The parse completed successfully. The following units were *on* when the network stabilized

S 1 5 par

A 1.4 lson
A 1.4.par

A 2.3 rson
A 2 3 par

B 2 2 lson
B 2 2 par

B.1.1.lson	B 2 1 lson	C 3 1.rson	A 4.1 rson	B 5 1 rson	
B.1.1 par	B 2.1 par	C 3 1 par	A 4 1 par	B 5.1 par	
b 1	b 2	a.3	a 4	b 5	$ 6

5.3 Deferred learning

The global learning described in the previous section would fail to distribute productions learned locally if more than one is learned during a parse. It is possible for the network to take advantage of periods of quiet to catch up on unlearned productions. The mechanism I propose (but have not implemented) is spontaneous *excited* activity of fixed match units when the network is quiet. The probability of a match unit becoming *excited* would be inversely proportional to the length of time it has been fixed, so that recently fixed match units would be more likely to become *excited*. The *excited* match unit would *excite* the parent and children of the production, which would start global learning as in the previous section. More than one match unit activating simultaneously would do no harm. Global learning of already learned productions will have no effect as described above

One way to accomplish this would be to run a link from the global learn unit (which could just as easily be a network of units) to match units. The link would be to a new site. The global learn unit is *on* when parsing is not taking place, so if the match unit responded to input to this site by becoming *excited* probabilistically, we would have the desired effect. The weight on this link could be made non-zero when the unit becomes fixed and gradually decay after that. Recently fixed match units would get the most activation and, thus be more likely to fire.

Rehearsing productions also makes the network more robust. If a match unit were to "die", then another would be recruited to take its place the next time its production was rehearsed.

6. Discussion

I consider the major advantage of my parser to be its generality and its quick, sure results. The major difference between it and other parsing schemes is the way it maintains all possible parses in parallel, eliminating the need for search (i.e. relaxation). One of the goals of connectionism is to account for the solution of complex tasks in a few computational steps using massive parallelism. My network does exactly that.

The major disadvantage of my parser is its rigid structure and fixed length. Because the length of the network is fixed, the set of strings parsed is finite. Of course, this is true of any implemented parsing mechanism, but one might hope for an extendable structure. It would be nice if the network could parse longer strings by acquiring more resources (i.e. units) on the fly. McClelland's (1985) CID mechanism may prove useful in this capacity (McClelland and Kawamoto, 1986), though the resource requirements of CID are substantial. I would very much like to be able to efficiently acquire and release general purpose working units as needed. Nevertheless, I think that fixed length structures can prove useful and may point the way to flexible structures which perform the same task.

The parsing algorithm upon which my network is based is an example of dynamic programming (Aho, Hopcroft & Ullman, 1974). Other problems with efficient dynamic programming solutions may have similar parallel implementations. One such problem is finding the number of edit operations (*insert, replace* and *delete*) required to convert one string to another. A fast implementation of this algorithm could be very useful in cognitive tasks requiring pattern matching. The values passed during a computation are the number of edit steps so far. This means that a potentially large amount of information must be comunicated, unlike the parsing network in which only the existence of constituents is communicated. While this does not stand in the way of a fast parallel implementation, it may present complications for a connectionist model, where the output values of units are not meant to carry much information (Feldman and Ballard, 1982).

The learning mechanism described is certainly inadequate as a theory of language acquisition, but in keeping with the purpose of the paper, it does demonstrate some techniques which may prove useful. In connectionist models, multiple copies of subnetworks which do the same task are common. Learning in such a way that all copies are kept consistent is a difficult problem. My solution does this with a minimum of overhead. It may even be possible to use a similar mechanism to learn new syntactic categories dynamically. The learning results also demonstrate that the network can be flexible.

There are several directions future work might take. One would be to make the length restriction more flexible. Another would be to improve the learning algorithm. The results given only scratch the surface of what is needed. Another we have partially worked out would be to efficiently extend the network to handle augmented grammars (Gazdar et al., 1985) in which the grammar symbols have properties. Rules are applicable only if the properties meet the accompanying restrictions, e g. that the NP and VP have the same number.

Acknowledgments

I would like to thank Garrison Cottrell and my advisor Jerome Feldman for many helpful suggestions and the taxpayers of the United States for funding fundamental research.

References

Aho, A.V., J.E. Hopcroft and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Reading, Mass.: Addison-Wesley, 1974.

Cottrell, G.W. *A connectionist approach to word sense disambiguation* Doctoral dissertation, Computer Science Department, University of Rochester, April 1985.

Fanty, M.A. and N. Goddard. "The Rochester Connectionist Simulator " *forthcoming.*

Feldman, J A. "Dynamic connections in neural networks," *Biological Cybernetics, 46,* 27-39, 1982.

Feldman, J.A. and D H. Ballard. "Connectionist models and their properties " *Cognitive Science, 6,* 205-254, 1982.

Gazdar, G., E. Klein, G. Pullum, and I. Sag. *Generalized Phrase Structure Grammar.* Oxford: Basil Blackwell Publisher Ltd, 1985

Hopcroft, J E. and J.D. Ullman. *Introduction to automata theory, languages, and computation.* Reading, Mass.: Addison-Wesley, 1979.

Kirkpatrick, S., C.D Gelatt and M.P Vecchi. "Optimization by simulated annealing " *Science, 220,* no. 4598, 671-680, 1983

McClelland, J.L. "Putting knowledge in its place: a scheme for programming parallel processing structures on the fly." *Cognitive Science, 9,* 113-146, 1985.

McClelland, J.L. and A.H. Kawamoto. "Mechanisms of sentence processing: Assigning roles to constituents of sentences " in D E. Rumelhart and J L. McClelland (Eds.) *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 2.* Cambridge, MA: Bradford Books/MIT Press.

Selman, B. and G Hirst. "A rule-based connectionist parsing system." *Proceedings of the seventh Annual Conference of the Cognitive Science Society,* Irvine, Cal., 212-221, 1985.

Waltz, D L. and J.B. Pollack. "Massively parallel parsing " *Cognitive Science, 9,* 51-74, 1985.

Appendix One

Network Building Algorithm

```
length is the maximum length of the input string
nonterminals is the set of nonterminals
terminals is the set of terminals
start-symbol is the start symbol of the grammar

for i ← 1 to length                    /* make the terminal units */
    for-each term in terminals
        MakeUnit( type ← Terminal, name ← term.i.1)
    endfor-each
    MakeUnit(type ← Terminal, name ← $.i.1)
endfor

for row ← 1 to length
    for col ← 1 to 1 + (length – row)
        for-each nt in nonterminals
            for-each prod in Productions-of(nt)
                if (row ≥ LengthOf(prod)) then  /* possibly room *
                    DoProduction(row,col,nt,prod)
                endif
            endfor-each
        endfor-each
    endfor
endfor


/* connect end markers with start symbol units */
for i ← 1 to length – 1
    if (Exists(start-symbol.1.i )) then
        MakeLink(from ← $ i + 1.1, to ← start-symbol 1 i,
                    weight ← 2, site ← top)
    endif
endfor

if (Exists(start-symbol.1.length )) then
        MakeUnit(type ← Terminal, name ← $ length + 1 1)
        MakeLink(from ← $ length + 1.1, to ← start-symbol 1 length,
                    weight ← 2, site ← top)
endif

DoProduction(length,start,nt,prod) ←

/*      The vectors ntlen and sum are used to generate every possible  *
/*      combination of lengths of symbols in this production  ntlen[i] is  *
/*      the length of the i'th nonterminal in prod  sum[i] is the sum of  *
/*      the lengths of the first i – 1 nonterminals in prod               *

        * set up initial configuration */

        weight ← 2 0  LengthOf(prod)
        pieces ← NumberOfNonterminals(prod)
```

```
nttot ← length – NumberOfTerminals(prod)   /* length of all nt's */

/* check for an all-terminal production of the wrong length */
if pieces = 0 and length = LengthOf(prod) then return

for i ← 1 to pieces – 1
    ntlen[i] ← 1
    sum[i] ← i – 1
endfor
sum[pieces] ← pieces – 1
ntlen[pieces] ← nttot – sum[pieces]

/* loop once for each configuration */

loop   * until break */

    /* test configuration */

    where ← start   /* where the next symbol must begin */
    whichnt ← 1     /* which nonterminal is next in the prod */
    for-each symbol in prod
        if IsNonterminal(symbol) then
            if Exists(symbol.where.ntlen[whichnt]) then
                whichnt ← whichnt + 1
                where ← where + ntlen[which]
            else goto next      /* this configuration fails */
            endif
        else /* symbol is a terminal *
            if Exists(symbol.where.1) then
                where ← where + 1
            else goto next      /* this configuration fails *
            endif
        endif
    endfor-each



/*  a configuration with all subordinate units in existence     */
/*  has been found.                                             */

if Not(Exists(nt.start.length)) then
    MakeUnit(type ← Nonterminal, name ← nt.start.length)
endif
/* match units are not named, so the index of the made         */
/* unit is saved in a variable for future reference            */
match ← MakeUnit(type ← Match)
where ← start     * where the next symbol must begin *
whichnt ← 1       * which nonterminal is next in the prod *
 * make links between match and subordinate units *
for-each symbol in prod
    if IsNonterminal(symbol) then
        MakeLink(from ← symbol.where.ntlen[whichnt],
                 to ← match, site ← bottom, weight ← weight)
        MakeLink(to ← symbol.where.ntlen[whichnt],
```

```
                        from ← match, site ← top, weight ← 1)
                whichnt ← whichnt + 1
                where ← where + ntlen[which]
            else /* symbol is a terminal */
                MakeLink(from ← symbol.where.ntlen[whichnt],
                        to ← match, site ← bottom, weight ← weight)
                MakeLink(to ← symbol.where.ntlen[whichnt],
                        from ← match, site ← top, weight ← 1)
                where ← where + 1
            endif
        endfor
        /* make links between match and nt */
        MakeLink(from ← match, to ← nt.start.length, site ← bottom,
                weight ← 1)
        MakeLink(to ← match,from ← nt.start.length, site ← top,
                weight ← 1)

        next:    /* make next configuration */

        change ← pieces - 1
        while(change > 0 and
              ntlen[change] + sum[change] ≥ nttot - (pieces - change))
            change ← change - 1
        endwhile
        if(change < 1)then exitloop    /* all done - no more */
                                       /* configurations     */
        endif
        ntlen[change] ← ntlen[change] + 1
        for i ← change + 1 to pieces
            ntlen[i] ← 1
            sum ← sum[i - 1] + ntlen[i - 1]
        endfor
        ntlen[pieces] ← nttot - sum[pieces]    /* last must take up slack */
endloop
```

Appendix Two

Unit Functions for Simple Network

```
/* site functions - these return a value for each site of the unit  These values
are used by the other functions below */

/* SFsum is used by bottom site of nonterminal units */
SFsum(input-list)
    sum = 0.0
    for-each input in input-list
        sum = sum + (Value(input)*Weight(input))
    endfor-each
    return sum
end SFsum

/* SFfilterSum is used by bottom site of match units */
SFfilterSum(input-list)
    sum = 0.0
    for-each input in input-list
        sum = sum + Min(Value(input), 5.0) * Weight(input))
    endfor-each
    return sum
end SFfilterSum

/* SFmax is used by top site of nonterminal and terminal units */
SFmax(input-list)
    hold = 0.0
    for-each input in input-list
        if(hold < (Value(input)*Weight(input))) then
            hold = (Value(input)*Weight(input))
        endif
    endfor-each
    return sum
end SFsum

/* These functions are called to set unit parameters after the site functions
have been called.  The same one can be used for all units if external primed
input is provided for the input string */

UFparse(unit)
    if(unit state = off and SiteValue(unit,"bottom")    10.0) then
            unit state = primed
            unit potential = 5.0
            unit output = 5
    else if(unit state = primed and SiteValue(unit,"top")    *0.0) then
            unit state = on
            unit potential = 10.0
            unit output = 10
    endif  /* else leave it alone */
end UFparse
```

END

DTIC

9-86