Confluent Preorder Parsing

CS-TR-95-03

HO, Kei Shiu Edward and CHAN, Lai Wan

Department of Computer Science
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong

email : ho052@cs.cuhk.hk and lwchan@cs.cuhk.hk

KEYWORDS: Neural Networks, Connectionist Syntactic Parsing, RAAM, Holistic Transformation, Confluent Preorder Parser, Linearization of a Hierarchical Parse Tree, Parsing Erroneous Sentences, Syntactic Disambiguation

Abstract

In this paper, syntactic parsing is discussed in the context of connectionism. A new model - the Confluent Preorder Parser (CPP), is proposed which exemplifies the holistic parsing paradigm. Holistic parsing has the advantage that little assumption has to be made concerning the detailed parsing algorithm, which is often unknown or debatable, especially when human language understanding is concerned. In the CPP, syntactic parsing is achieved by transforming in a one-shot manner, from the connectionist representation of the sentence to the connectionist representation of the preorder traversal of its parse tree, instead of the parse tree itself. As revealed by the simulation experiments, generalization performance is excellent (as high as 90%). Besides, the CPP is also capable of parsing erroneous sentences and resolving syntactic ambiguities. A systematic study is conducted to explore the range of factors which can affect the effectiveness of it. This error-recovery capability is especially useful in natural language processing when incomplete or even ungrammatical sentences are to be dealt with.

1. Introduction

In language processing, syntactic parsing addresses the problem of finding out the hierarchical relationship (e.g. constituency) that exists between the terminals (or more specifically, words) in a sentence. Traditionally, this relationship is manifested in the form of a tree - the parse tree. Using classical AI methodologies, sentences and their parse trees are represented symbolically (for example, as character strings, linked-lists, etc.) and the parsing problem is treated as a rational inference process in which the internal structure of the sentence is broken down and analyzed step-by-step (which corresponds to, say, reading the sentence from left-to-right), with the parse tree being built up incrementally at the same time (see Allen, 1995).

In this paper, syntactic parsing is discussed in the setting of connectionism, sentences and parse trees are represented subsymbolically as vectors of real-valued units (subsymbols) and syntactic parsing is achieved via holistic inference (Blank et. al., 1992) - a process in which the connectionist representation of a sentence is transformed to the connectionist representation of its parse tree directly without first decoding the subsymbolic representation back to the sentence it encodes nor building the parse tree incrementally (see Figure 1).

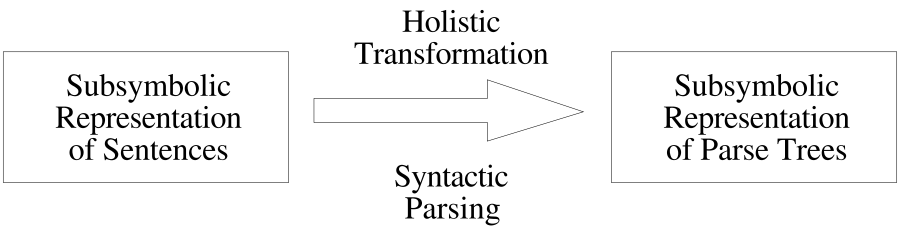


Figure 1. Holistic Parsing Paradigm

In this paper, we are concerned with :

(i) **deriving connectionist representations for the sentences and their parse trees which are efficient in preserving the most sequential and hierarchical information inherent in the respective data structure relevant to the task of syntactic parsing, and**

(ii) designing a holistic parsing transformation which is robust and which can generalize satisfactorily to novel sentence structures.

Compared to the classical techniques, holistic parsing is unique in the way that little knowledge will be assumed for the detailed parsing mechanism, which is often unknown or debatable, especially when human language processing is concerned. Effectively, the actual parsing process is treated as a "black box". During training, the parser will adapt itself to acquire the necessary grammatical regularity underlying the set of training examples. Connectionist parsing with its robust processing characteristics has the appeal of its potential capability to handle incomplete and even ungrammatical sentences. In addition, it can generalize well to novel sentences with exposure to a limited subset of possible sentences only. By analyzing the underlying mechanism of the holistic parsing inference process after training, we may obtain an idea of a model which is more psychologically plausible and which can give a better explanation to the real behavioral data in language usage.

In succeeding discussions, two connectionist models will be discussed, namely, the **Backpropagation Parsing Network (BPN)** and the **Confluent Preorder Parser (CPP)**. Both of them follow the idea of connectionist syntactic parsing as depicted in Figure 1. Common to them, sequential sentences and their corresponding hierarchical parse trees are represented using **Recursive Auto-Associative Memory (RAAM)** and/or **Sequential Recursive Auto-Associative Memory (SRAAM)**. So in Section 2, the basic principles of RAAM and SRAAM will be covered first.

Then we will briefly describe in Section 3 the Backpropagation Parsing Network (BPN) - a holistic connectionist parser first proposed in (Ho & Chan, 1994), and discuss its limitations. Rectifying these limitations, a new model is introduced in Section 4. This new model - the Confluent Preorder Parser (CPP), differs from the BPN in two major aspects. Firstly, the equivalence between a parse tree and its preorder traversal is exploited. The original hierarchical parse tree is transformed via a 1-to-1 symbolic mapping into a sequence listing the nodes and leaves that are come across when the tree is visited in a preorder-like fashion. Two separate SRAAMs are trained to encode the two types of sequences - the original sentences and the preorder traversals, respectively. Syntactic parsing is achieved by a sequence-to-sequence transformation - from the SRAAM representation for the original sentence sequence to the SRAAM representation for the preorder traversal sequence of its corresponding parse tree, instead of the parse tree itself.

Secondly, to further enhance the performance of the CPP, the idea of confluent inference (Chrisman, 1991) is applied. The representation of a complete sentence and that of the preorder traversal of its corresponding parse tree will no longer be developed independently as in the BPN. Instead, they are forced to be approximately equal (or so called confluent) during training. In this way, the holistic parsing function depicted in Figure 1 is effectively implemented as an identity function. Simulations were performed and the results will be presented in Section 5.

In Section 6, we will evaluate the robustness of the CPP by systematically studying its error-recovery capability in processing erroneous sentences. Four types of errors have been examined (erroneous sentences with one wrong element, one extra element, one missing element, or two elements swapped). Different factors - the type of the error, the location of the error and the length of the sentence, which affect the effectiveness of this error-correction capability will be evaluated. Taking one step further, we will show in Section 7 how this robust processing characteristic of the CPP can help in achieving syntactic disambiguation.

Finally, a conclusion will be given in Section 8.

2. Recursive Auto-Associative Memory (RAAM)

How can symbolic data structures be represented in a neural network? Pollack has offered an efficient scheme - the Recursive Auto-Associative Memory (hereafter referred as RAAM or structured RAAM) (Pollack, 1990). Stated in its original form, a RAAM is basically a km-m-km feedforward encoder network (where $k \geq 2$ can be considered as the degree of compression) capable of deriving distributed representations for recursive symbolic data structures, e.g. trees, lists, etc. Upon successful training, say, by using the back-propagation algorithm (Rumelhart, Hinton & Williams, 1986), the hidden layer activation is the RAAM representation of the composite structure with its k components appearing in the input layer. The fact that the representations for all the components as well as their composite structure are of the same length m enables the latter to appear as a part of another more complicated structure via recursive composition (see Figure 2).

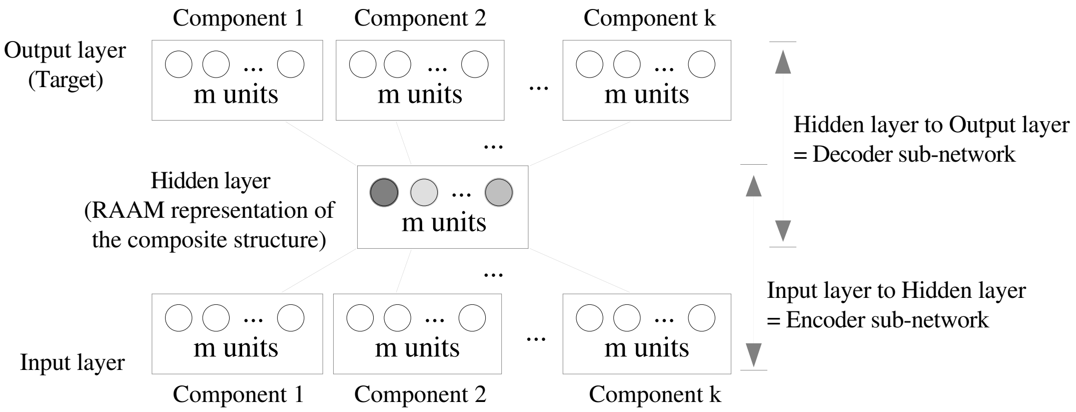


Figure 2. Recursive Auto-Associative Memory (RAAM)

2.1 Representing Parse Trees

In principle, any symbolic data structure representable as a k-branching tree can be encoded using RAAM. Consider the binary tree in Figure 3 which can also be schematically represented by the Lisp-like structure ((D N) (V ((D N) (P (D N)))))[1] where N, V, P and D are atomic symbols (also known as leaves or terminals) while s, np1, np2, vp and pp are composite structures (also called internal nodes or non-terminals). If we think of D as standing for determiner, N for noun, V for verb, P for preposition, np1 and np2 for noun phrases, vp for verb phrase, pp for prepositional phrase and s for sentence (in other words, they are all grammatical categories), the binary tree shown in Figure 3 can be perceived as representing the parse tree for the sentence pattern <D N V D N P D N>[2].

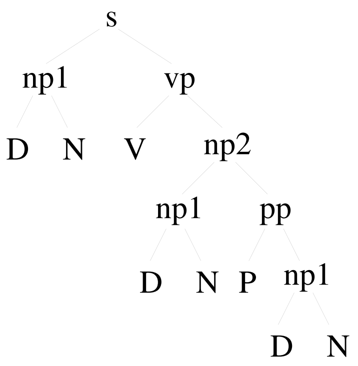


Figure 3. A binary tree

To encode s, we train a RAAM network with k=2. A representation is first assigned to each atomic symbol, which is usually a binary vector. The encoding process starts from the leaves of the tree and works bottom-up, first combining terminals to produce simpler structures (e.g. np1) which can then be further composed with other terminals or structures to give rise to more complicated structures (e.g. np1 + pp ⇒ np2). This process repeats recursively until the top of the tree s is reached. The set of training patterns in Table I are obtained.

Input layer		*Hidden layer*	*Output layer (Target)*	
Component 1	*Component 2*	*Composite structure*	*Component 1*	*Component 2*
D	N	np1 = (D N)	D	N
P	np1	pp = (P (D N))	P	np1
np1	pp	np2 = ((D N) (P (D N)))	np1	pp
V	np2	vp = (V ((D N) (P (D N))))	V	np2
np1	vp	s = ((D N) (V ((D N) (P (D N)))))	np1	vp

Table I. Training patterns to encode the parse tree ((D N) (V ((D N) (P (D N)))))

A point to note is that the representations for the internal nodes np1, np2, pp and vp which are supposed to be evolving during training can also appear in the input layer and the output layer (as training target). In other words, we are facing a moving-target training problem.

Our experience on the experimental simulations reveals that a smaller rate and a larger momentum are preferred in order to have the best performance. More importantly, the real-time weight update mode should be used instead of batch update (sometimes, the batch update mode simply doesn't work). This should sound intuitively natural since larger, more complicated structures (e.g. vp and s) are defined out of simpler structures (such as np1 and pp). This also dictates that the above training patterns should be fed to the RAAM in the strict top-down order as presented.

When training succeeds, the weights from the input layer to the hidden layer define a RAAM encoder while the weights from the hidden layer to the output layer effectively implement a RAAM decoder. By "clamping" the representations of the components in the input layer, the RAAM representation for the composite structure will appear in the hidden layer.

On the other hand, by "clamping" the RAAM coding of a structure in the hidden layer, the representations for its k components will be obtained in the output layer. For each of these components, we first test for its binary-ness. If it passes, it is a terminal and we proceed to find out its actual identity. Otherwise, it is a non-terminal and we have to decompose it further using the RAAM decoder (by "clamping" its RAAM coding in the hidden layer again) to extract its immediate components and apply our decoding rule recursively to each of them. This process continues until all non-terminals have been fully decoded to their constituent terminals, or the depth of recursion has exceeded the maximum level of nesting of all the structures concerned. In the latter case, we simply fail.

2.2 Representing Sentences in SRAAM

There is a special kind of binary trees, known as left-branching trees. With a little imagination, the sentence <D N V D N P D N> (which corresponds to the parse tree in Figure 3) can be perceived as being represented by the left-branching tree ((((((((# D) N) V) D) N) P) D) N) shown in Figure 4 (where # is an extra terminal used to denote an empty sentence/sequence) (Pollack, 1990). In this sense, the notion of "structure" and "component" (or "sub-structure") will be replaced by "sentence" and "prefix" respectively, while the bottom-up encoding (by RAAM) and the top-down decoding of the left-branching tree effectively corresponds to the left-to-right encoding and the right-to-left decoding of the sentence respectively (see Figure 4).

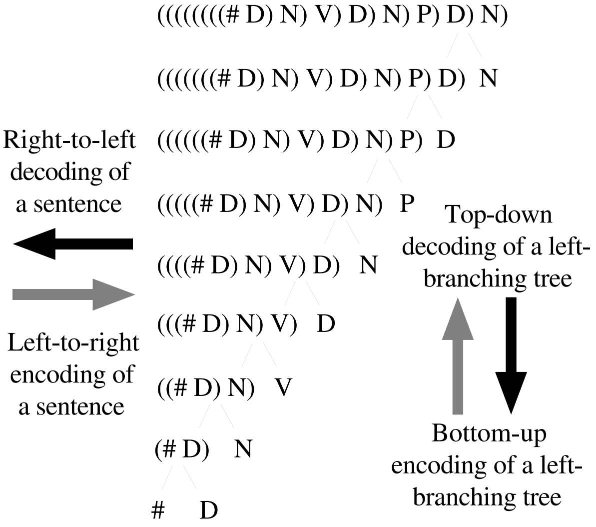


Figure 4. The sentence <D N V D N P D N> represented as
a left-branching tree ((((((((# D) N) V) D) N) P) D) N)

In a left-branching tree, as its name suggests, only the left branch expands in every level and the right child of each internal node is always a terminal. When represented using RAAM, this structural characteristic can help to simplify the network architecture. Firstly, the compression factor k can be fixed at 2 since left-branching trees are binary trees by definition. As a result, we now only have "component 1" and "component 2" in the input layer and the output layer (cf. Figure 2), and we shall rename them as "prefix <1..i-1>" (meaning the SRAAM representation of the prefix from the 1-st terminal to the (i-1)-th terminal) and “i-th terminal” respectively (see Figure 5).

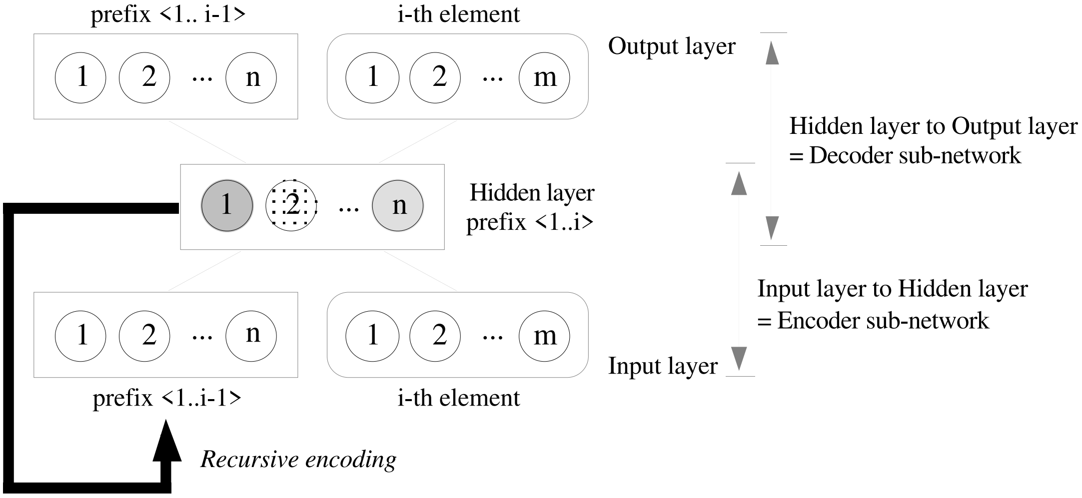


Figure 5. Sequential RAAM (SRAAM)

In addition, during the recursive left-to-right encoding (and training) of a sentence, the representations of the sentence itself and its prefixes will appear in “prefix <1..i-1>“ of the input layer (and the output layer) only, but never in “i-th terminal”, in which we can have a terminal only - the i-th (next) terminal in the sentence sequence. Observing this, we can use a sufficiently large number of units in the hidden layer for encoding sentences and their prefixes without increasing the length of the representations of terminals at the same time. In this way, more structures can be encoded without scaling up the training time too much. More importantly, the decoding logic now becomes simpler. What is obtained in “i-th terminal” during each decoding step should be a terminal and we only have to check if the representation in “prefix <1..i-1>“ is to be decoded/expanded further or not.

To exploit these advantages, the original RAAM is adapted accordingly, giving rise to the Sequential RAAM (SRAAM) architecture in Figure 5 (Pollack, 1990). As shown, the choice of the value of n can now be made independently of the value of m. As an illustration, to encode the sentence <D N V D N P D N>, the set of training patterns in Table II will be used.

Input layer		*Hidden layer*	*Output layer (Target)*	
prefix <1..i-1>	*i-th terminal*	*prefix <1..i>*	*prefix <1..i-1>*	*i-th terminal*
#	D	<D>	#	D
<D>	N	<D N>	<D>	N
<D N>	V	<D N V>	<D N>	V
<D N V>	D	<D N V D>	<D N V>	D
<D N V D>	N	<D N V D N>	<D N V D>	N
<D N V D N>	P	<D N V D N P>	<D N V D N>	P
<D N V D N P>	D	<D N V D N P D>	<D N V D N P>	D
<D N V D N P D>	N	<D N V D N P D N>	<D N V D N P D>	N

Table II. Training patterns to encode the sentence <D N V D N P D N>

This completes our discussion on the theory of RAAM. To conclude, a data structure representable as a hierarchical tree can now be encoded using a (structured) RAAM, whereas a data structure representable as a sequence can be encoded using a SRAAM. Effectively, most data structures can be catered for. In subsequent discussions, we will attempt the parsing problem using RAAM as the representation tool.

3. Backpropagation Parsing Network (BPN)

3.1 Syntactic Parsing via SRAAM-to-RAAM Transformation

Ho and Chan realize the holistic syntactic transformation in Figure 1 by using a fully-connected 3-layered feedforward neural net (Ho & Chan, 1994). A set of sentences and their respective parse trees are first encoded by training a SRAAM and a (structured) RAAM respectively and independently. Then, the Parsing Network is trained to approximate the mapping from the SRAAM code representing a sentence to the RAAM code representing its corresponding parse tree (see Figure 6).

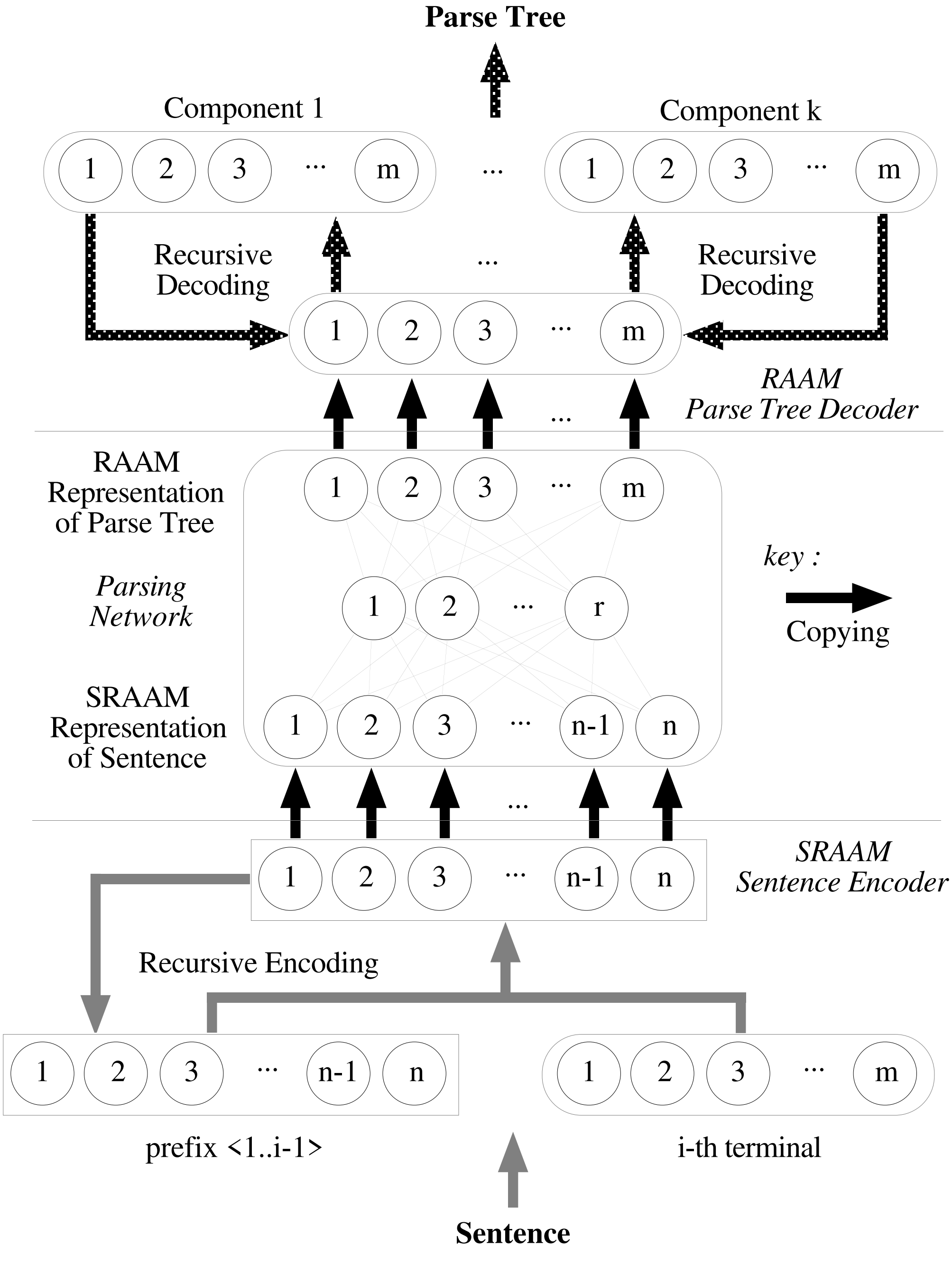


Figure 6. Backpropagation Parsing Network (BPN)

To parse a sentence, its SRAAM code is first obtained by recursive encoding via the SRAAM encoder sub-network, which is then transformed by the Parsing Network into a RAAM code. This RAAM code is supposed to represent the parse tree of the original sentence. Upon recursive decoding by the RAAM decoder sub-network, the symbolic form of the parse tree can be reconstructed.

Preliminary experiments reveal that the performance of this approach is unsatisfactory. Generalization is as low as 40%. Yet, because of the limited reconstruction capability of the SRAAM network, the system can generate probable parses for damaged input sequences.

3.2 Drawbacks of the BPN

There is no denying that the overall generalization performance of the BPN will very much depend on the generalization capability of the Parsing Network. As a feed-forward net, it generalizes on the basis of the similarity between the novel pattern and (one of) the training patterns. Intuitively, the inference process of the parse tree of an unseen sentence resembles an interpolation or extrapolation in some sense. To have the best performance, we should at least require sentences of similar structures (thus having similar parse trees) to be represented by similar SRAAM and RAAM codes. In other words, the overall generalization will also depend on the generalization capability of the representation schemes/mechanisms for the sentences and their parse trees which in the BPN are the SRAAM and RAAM networks respectively.

However, a significant drawback of the BPN is that the SRAAM representations of the sentences and the RAAM representations of their parse trees are being learned/developed separately and independently. Thus, there is no guarantee that a meaningful correlation can be established between them. Even worse, the unavoidable inexactness incurred in training the Parsing Network simply sacrifices further the generalization capability of the overall parsing model.

4. Confluent Preorder Parser (CPP)

In view of this, a new model - the Confluent Preorder Parser (CPP), is proposed (see Figure 7). Basically, it is a dual-ported SRAAM (Chrisman, 1991), with one single hidden layer (shaded in Figure 7) apparently shared between two SRAAM networks - the SRAAM Sentence Encoder and the SRAAM Preorder Traversal Encoder. Two major modifications have been made to the BPN :

(i) Firstly, a parse tree is no longer perceived as a hierarchical data structure. Instead, it is "linearized" into a sequence which lists the nodes and leaves that are come across when the parse tree is visited in a preorder-like fashion (see Section 4.1). Parsing is thus implemented as a transformation from one type of sequences - the sentences, to another type of sequences - the preorder traversals of the parse trees. SRAAM representations are derived for each type of sequences.

(ii) Secondly, the syntactic transformation originally implemented by the Parsing Network is now replaced by a 1-to-1 mapping (that is, an identity function). In other words, the SRAAM representation for a sentence will now be directly decoded to produce the preorder traversal sequence desired, from which the appropriate parse tree is constructed. As a result, the error incurred by the Parsing Network can be avoided. To achieve this, identical representation will have to be developed for each sentence and the preorder traversal of its parse tree during training. Through confluent inference (Chrisman, 1991), the two types of representations will no longer evolve independently as in the BPN (see Section 4.2).

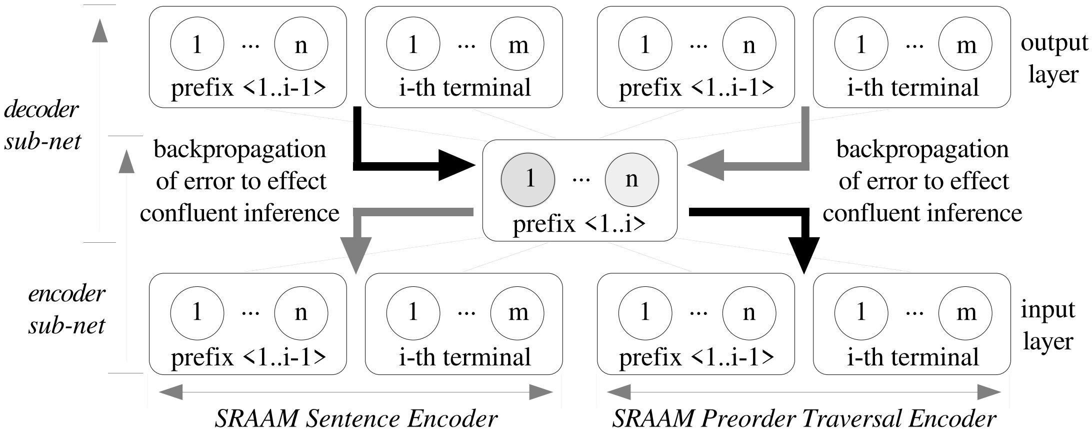


Figure 7. Confluent Preorder Parser (CPP)

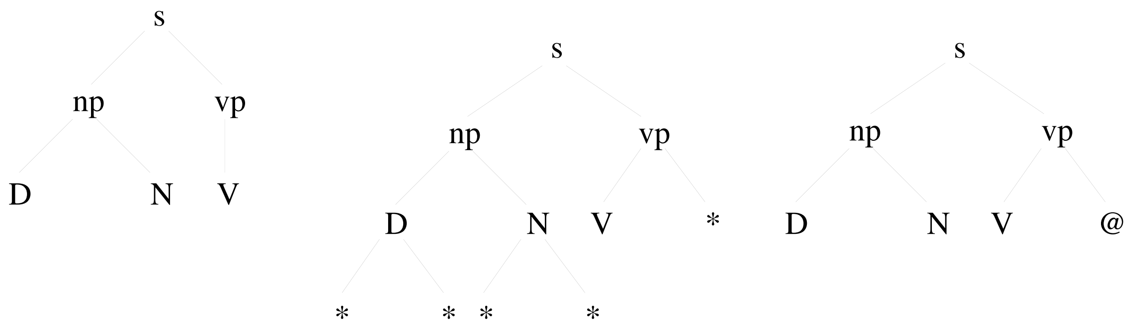
4.1 Mapping Hierarchical Parse Trees to Preorder Traversal Sequences

It is known that a full binary tree can be uniquely defined by listing its nodes and leaves in preorder (in case it is not full, we need to know the inorder traversal as well). For example, given the binary tree ((D N) (V ((D N) (P (D N))))) shown in Figure 3, its preorder traversal will give rise to the sequence <s np D N vp V np np D N pp P np D N>. Alternatively, we can choose to ignore the actual identities of the internal nodes (such as np and vp) and represent them uniquely by a special terminal $. In this way, the preorder traversal will be <$ $ D N $ V $ $ D N $ P $ D N>.

We should emphasize that the notion of preorder traversal should not be confined to binary trees only. Given a general tree of valence k (where k ≥ 2), for each internal node having q < k children only, (k-q) leaves will be added (on the right hand side of its q children), each being a special terminal @[3]. Effectively, the tree becomes regularly structured with each node being either a leaf or an internal node having k children exactly. The resulting full k-tree is then traversed in a preorder-like fashion based on the following algorithm :

```
preorder_traversal (t : tree_of_valence_k)
{
if (t is a leaf) then
    display (t)
else
    if (t is an internal node) then
        {
        display (t)                /* or display ($) if the identities of the internal nodes are not distinguished */
        for each $t_i$ of the k children of t do
            preorder_traversal ($t_i$)
        }
}
```

Kwasny and Kalman first applied this type of 1-to-1 mapping between a tree and its preorder traversal to facilitate the encoding of a set of trees using a SRAAM network (Kwasny & Kalman, 1993). The major difference between their approach and that of the CPP is that in theirs, null pointers are explicitly represented for the leaves of the trees and those internal nodes with number of branches less than the valence of the tree (intuitively, they are not "full"). Consequently, the preorder traversal of the binary parse tree in Figure 8(a) will be represented as in Figure 8(b) and preorder traversal of it will give rise to the sequence of terminals <s np D * * N * * vp v * * *>, where s, np and vp represent internal nodes and * denotes a null pointer. Whereas in the CPP, it is represented as in Figure 8(c). As shown, the special terminal @ has been inserted as the right child of vp in order to make it (and the whole tree) full. Its preorder traversal will give rise to the sequence <s np D N vp V @>.


(a) A binary parse tree

(b) Representation of the binary tree in (a) in the model of Kwasny and Kalman

(c) Representation of the binary tree in (a) in the CPP

Figure 8. Different representations of the binary parse tree ((D N) (V)) of the sentence <D N V>

A point to note is that in the simulations which are to be described in Section 5, all grammar rules are of valence 2. In other words, they are of the form X → Y Z (where X is a non-terminal while Y and Z can be a terminal or a non-terminal). Therefore, only full binary trees will be generated and we don't need the terminal @ at all. But in case there is any rule of the form X → Y, the extra terminal @ will be assumed to fill up trees with empty branches.

4.2 Training via Confluent Inference

Given a set of training sentences, we generate another set of preorder traversals for their corresponding parse trees. With reference to Figure 7, the training sentences as sequences will be encoded by the SRAAM Sentence Encoder on the left of the CPP, while the preorder traversals will be encoded by the SRAAM Preorder Traversal Encoder on the right. Syntactic parsing then operates by transforming from the SRAAM representation of the original sentence to the SRAAM representation of the preorder traversal of its parse tree.

Instead of simply encoding the two types of sequences independently and then having them paired arbitrarily as in the BPN, the SRAAM representation for each complete sentence and the SRAAM representation for the preorder traversal of its corresponding parse tree will be made identical by backpropagating (during training) the error from the SRAAM Sentence Encoder network to the SRAAM Preorder Traversal Encoder network and vice versa. This idea conforms with the original spirit of confluent inference as advocated by Chrisman (Chrisman, 1991) - making close the representations for the problem space, which in our case are the SRAAM codes for the sentences, and the representations for the solution space, which are the SRAAM codes for the preorder traversals of the parse trees.

Let's illustrate this idea with an example. Consider the sentence <D N V D N> with its parse tree shown in Figure 9. Preorder traversal of it gives rise to the sequence <s np D N vp V np D N> .

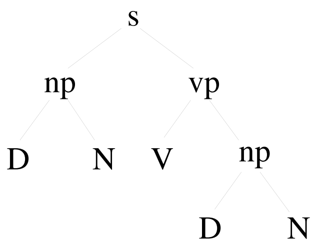


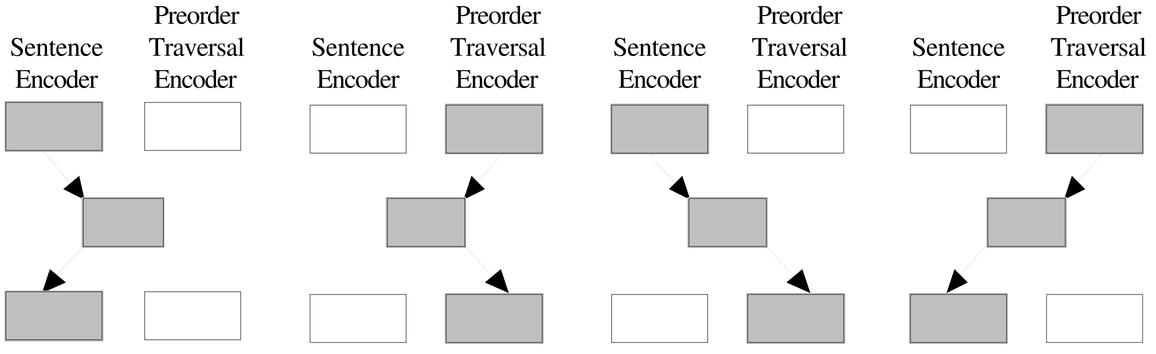
Figure 9. The parse tree of the sentence <D N V D N>

To train the CPP to parse the sentence, we need three types of training patterns :

(i) Firstly, SRAAM representations are obtained for the sentence <D N V D N> and its four prefixes <D>, <D N>, <D N V> and <D N V D> by encoding them using the following five training patterns :

Input layer			*Output layer (Target)*	
Sentence Encoder	Preorder Traversal Encoder	*Hidden layer*	Sentence Encoder	Preorder Traversal Encoder
# + D	Don't care	<D>	# + D	Don't care
<D> + N	Don't care	<D N>	<D> + N	Don't care
<D N> + V	Don't care	<D N V>	<D N> + V	Don't care
<D N V> + D	Don't care	<D N V D>	<D N V> + D	Don't care
<D N V D> + N	Don't care	<D N V D N>	<D N V D> + N	Don't care

The null-context # (which represents an empty sequence) is used for initializing "prefix <1..i-1>" (see Figure 7) at the start of every sentence. By "don't care"-ing some of the input neurons of the CPP and their corresponding output neurons, the weights joining these "don't care"-d units and the hidden neurons are effectively fixed and will not be adapted during this part of training. In other words, only the SRAAM Sentence Encoder of the total CPP will be involved for these training patterns, as depicted by the shaded portion in Figure 10a. In this respect, it simply functions as an ordinary SRAAM encoder.



(a) Encoding sentences (b) Encoding preorder traversal sequences (c) Confluent inference (d) Confluent inference

Figure 10. Different portions (shaded) of the CPP are adapted for different training patterns.
The arrows show the backpropagation of error and the white blocks refer to "Don't care"-d inputs/outputs.

(ii) Similarly, the preorder traversal sequence <s np D N vp V np D N> and its prefixes (<s>, <s np>, <s np D>, <s np D N>, <s np D N vp>, <s np D N vp V>, <s np D N vp V np> and <s np D N vp V np D>) are encoded using the following set of training patterns :

Input layer			*Output layer (Target)*	
Sentence Encoder	Preorder Traversal Encoder	*Hidden layer*	Sentence Encoder	Preorder Traversal Encoder
Don't care	# + s	<s>	Don't care	# + s
Don't care	<s> + np	<s np>	Don't care	<s> + np
Don't care	<s np> + D	<s np D>	Don't care	<s np> + D
Don't care	<s np D> + N	<s np D N>	Don't care	<s np D> + N
Don't care	<s np D N> + vp	<s np D N vp>	Don't care	<s np D N> + vp
Don't care	<s np D N vp> + V	<s np D N vp V>	Don't care	<s np D N vp> + V
Don't care	<s np D N vp V> + np	<s np D N vp V np>	Don't care	<s np D N vp V> + np
Don't care	<s np D N vp V np> + D	<s np D N vp V np D>	Don't care	<s np D N vp V np> + D
Don't care	<s np D N vp V np D> + N	<s np D N vp V np D N>	Don't care	<s np D N vp V np D> + N

This time, only the weights of the SRAAM Preorder Traversal Encoder network are affected (see Figure 10b).

(iii) Finally, to make confluent inference work, two more patterns have to be added to place extra constrains on the evolution of the SRAAM representation for the complete sentence and that for the preorder traversal of its respective parse tree :

Input layer			*Output layer (Target)*	
Sentence Encoder	Preorder Traversal Encoder	*Hidden layer*	Sentence Encoder	Preorder Traversal Encoder
<D N V D>+N	Don't care	<D N V D N>	Don't care	<s np D N vp V np D>+N
Don't care	<s np D N vp V np D>+N	<s np D N vp V np D N>	<D N V D>+N	Don't care

Error is backpropagated from the output layer of the SRAAM Preorder Traversal Encoder to the input layer of the SRAAM Sentence Encoder, and from the output layer of the SRAAM Sentence Encoder to the input layer of the SRAAM Preorder Traversal Encoder, as illustrated by Figures 10c and 10d respectively. All we want is to develop identical representation for <D N V D N> and <s np D N vp V np D N>.

In this way, not only the SRAAM code for each complete sentence and the SRAAM code for the preorder traversal of its corresponding parse tree can be brought closer together, but the intermediate representations for their sub-sentences such as <D N V> and sub-sequences of the preorder traversals such as <s np D N vp V> will also interact with one another to affect their own final codings, since they are bound by the part-whole relationship that exists between a sentence and its prefixes as well as between a preorder traversal and its prefixes.

A natural consequence is that they will be correlated in a regular and reasonable manner so that sentences having similar syntactic structures will share some fragments in their preorder traversal sequences (because of linearization), and thus are more likely to be represented by similar SRAAM codes. More importantly, both types of representations can be better adapted to the characteristics of the task at hand, namely syntactic parsing.

As a final remark, training the network in this way immediately gives rise to two useful functions, the first one being parsing. In addition, given the preorder traversal of a parse tree, we can encode it using the encoder sub-net of the SRAAM Preorder Traversal Encoder which can then be recursively decoded using the decoder sub-net of the SRAAM Sentence Encoder on the left to recover the original sentence (effectively, the brackets in the Lisp-like representation of the tree are removed). This is indeed the inverse function of parsing which may be useful for error-checking and error-correction.

4.3 Confluent Preorder Parsing after Training

With confluent inference, the holistic parsing transformation as depicted in Figure 1 is effectively implemented by a 1-to-1 mapping function. Upon successful training, syntactic parsing proceeds as follows. With reference to Figure 11, treating a sentence as a sequence of terminals, its SRAAM representation is first derived by recursive encoding it from left to right using the encoder sub-net of the SRAAM Sentence Encoder (step 1).

The SRAAM code thus obtained (after the whole sentence has been scanned) will be fed directly to the decoder sub-net of the SRAAM Preorder Traversal Encoder (step 2). Through recursive decoding, another sequence of terminals will be obtained which corresponds to the preorder traversal of the parse tree of interest (step 3). By making use of the equivalence mapping between this preorder traversal and the parse tree it is derived from, the latter can be re-constructed.

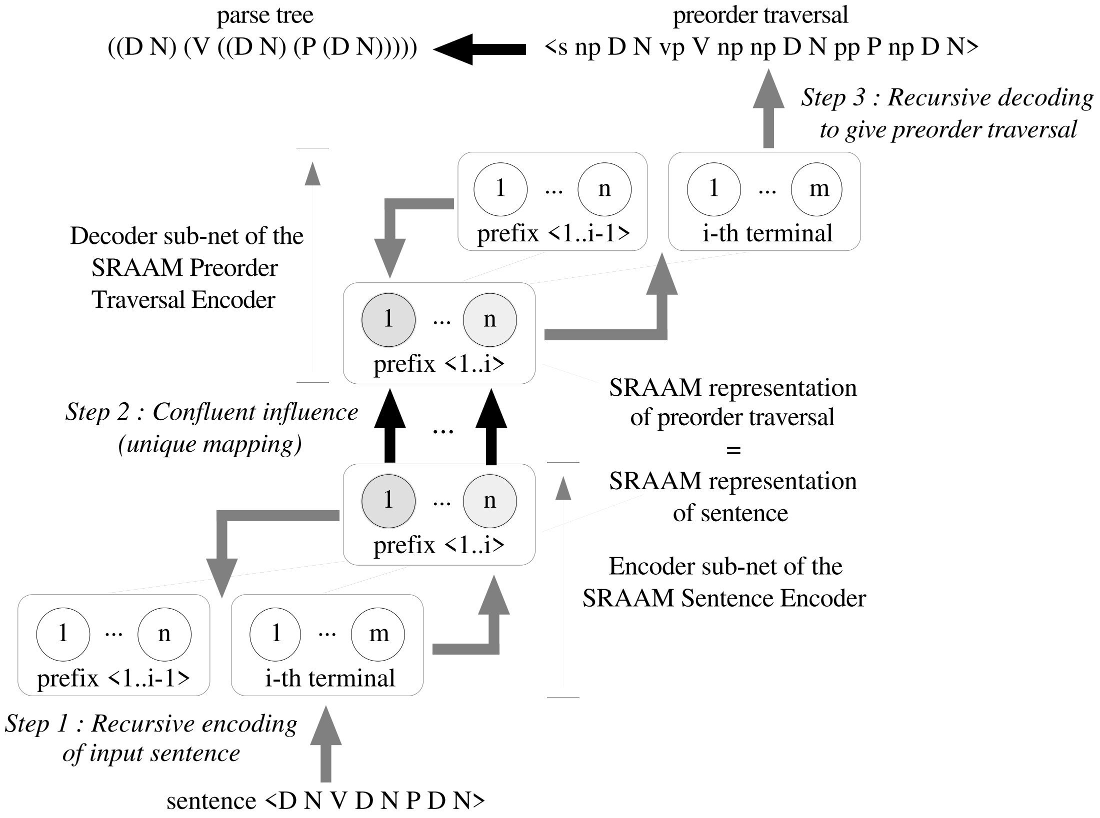


Figure 11. Parsing the sentence <D N V D N P D N> to obtain the parse tree ((D N) (V ((D N) (P (D N)))))

4.4 Advantages of the Confluent Preorder Parser

Two advantages can be expected from this new parsing model :

(i) Firstly, experience tells us that it is far more difficult to train a structured RAAM to encode trees than to train a SRAAM to encode sequences (provided that the number of parameters in both cases are comparable). As depicted in Figure 5, the decision for the length of the SRAAM codes for non-terminals (that is, the value of n) can be made independently of the choice of the value of m which is the length of the codings for terminals. In other words, we can increase the storage capacity of the hidden layer (so as to encode more sentences and parse trees) by increasing the value of n without affecting m at all. When n is sufficiently larger than m, the network effectively reduces to an encoder with a compression ratio of 2 to 1.

(ii) More importantly, the structure of a parse tree is explicitly represented via the use of the internal nodes such as np. Consequently, it can be expected that similar sentences having similar parse trees will share some sub-sequences in their preorder traversal sequences. For example, the sentences <D A N V D A N>, <D A N V D N> and <D N V D A N> will correspond to the preorder traversals <s np D ap A N vp V np D ap A N>, <s np D ap A N vp V np D N> and <s np D N vp V np D ap A N> respectively.

What is significant is that this type of similarity will be manifested at the surface level, as different from the BPN, in which the structure of a parse tree is only implicitly reflected by the way (and the order) the different terminals and components are “amalgamated” together by the RAAM to form the parse tree. Since neural networks generalize by interpolating/extrapolating between the novel sentence pattern and the training patterns, this surface similarity in the representations of the preorder traversals of the parse trees will undoubtedly promote the overall generalization performance of the parser. Moreover, through confluent inference, there will be a high probability that the SRAAM codings for the two types of sequences (sentences and the preorder traversals of their corresponding parse trees) will correlate in a meaningful and regular way from which generalization can be further benefited.

4.5 Extension : Learning to Parse Phrases

So far in our discussion, the CPP needs to learn parsing complete sentences only. This agrees with the basic philosophy of the holistic parsing paradigm that the minimum grammatical knowledge should be assumed. The CPP has access to the ultimate input (the sentence) and the ultimate output (the parse tree) of parsing only. It has no idea of their internal structure, like the fact that meaningful syntactic groupings can be formed among the terminals in a sentence, each of them being directly associated with certain sub-tree of the total parse tree.

Relaxing this constraint slightly, the CPP can be taught to parse phrases also in addition to complete sentences. Phrases are generated using internal nodes other than s as the root (e.g. np). Take for an instance, the sentence s=<D N V D N> (which is considered in Section 4.2) is in fact made up of two phrases - a noun phrase np=<D N> and a verb phrase vp=<V D N>. They correspond to the sub-parse-trees (D N) and (V (D N)), with preorder traversals <np D N> and <vp V np D N> respectively. To acquire this knowledge, the CPP has to learn the association between the SRAAM representation of np=<D N> and that of <np D N>, and also the association between the SRAAM representation of vp=<V D N> and that of <vp V np D N>. Specifically, an identical representation has to be developed for np=<D N> and <np D N>, as well as for vp=<V D N> and <vp V np D N>. Confluent inference is thus applied again, as in the case of a complete sentence and its corresponding parse tree.

In order to achieve this, the set of training patterns mentioned in Section 4.2 has to be augmented. To begin with, training patterns have to be added for deriving representations for the phrases np=<D N> and vp=<V D N> by the SRAAM Sentence Encoder. Since np=<D N> has already been encoded (as <D N> is a prefix of <D N V D N>), we need to cater for vp=<V D N> only :

Input layer			*Output layer (Target)*	
Sentence Encoder	Preorder Traversal Encoder	*Hidden layer*	Sentence Encoder	Preorder Traversal Encoder
# + V	Don’t care	<V>	# + V	Don’t care
<V> + D	Don’t care	<V D>	<V> + D	Don’t care
<V D> + N	Don’t care	<V D N>	<V D> + N	Don’t care

Next, the preorder traversals <np D N> and <vp V np D N> which correspond to the two sub-trees (D N) and (V (D N)) respectively are encoded by the SRAAM Preorder Traversal Encoder :

Input layer			*Output layer (Target)*	
Sentence Encoder	Preorder Traversal Encoder	*Hidden layer*	Sentence Encoder	Preorder Traversal Encoder
Don’t care	# + np	<np>	Don’t care	# + np
Don’t care	<np> + D	<np D>	Don’t care	<np> + D
Don’t care	<np D> + N	<np D N>	Don’t care	<np D> + N
Don’t care	# + vp	<vp>	Don’t care	# + vp
Don’t care	<vp> + V	<vp V>	Don’t care	<vp> + V
Don’t care	<vp V> + np	<vp V np>	Don’t care	<vp V> + np
Don’t care	<vp V np> + D	<vp V np D>	Don’t care	<vp V np> + D
Don’t care	<vp V np D> + N	<vp V np D N>	Don’t care	<vp V np D> + N

Finally, the following training patterns help to enforce the necessary constraints of confluent inference :

Input layer			*Output layer (Target)*	
Sentence Encoder	Preorder Traversal Encoder	*Hidden layer*	Sentence Encoder	Preorder Traversal Encoder
<D> + N	Don’t care	<D N>	Don’t care	<np D> + N
Don’t care	<np D> + N	<np D N>	<D> + N	Don’t care
<V D> + N	Don’t care	<V D N>	Don’t care	<vp V np D> + N
Don’t care	<vp V np D> + N	<vp V np D N>	<V D> + N	Don’t care

As revealed by our simulations to be reported in Section 5, the generalization performance of the CPP can be substantially promoted with this extension. But doubtless, this is realized at the expense of using more training patterns (given the same number of complete training sentences).

5. Experimental Comparisons

In this section, we will report on two simulations carried out to evaluate the performance of the CPP. A hypothetical grammar is assumed in each case from which a set of syntactically well-formed sentences are obtained. This set of sentences is divided into two subsets - the training set and the testing set. The parser is trained using the back-propagation algorithm on the training set. Training is declared to be successful (and which is then ceased) when all sentences in the training set have been learned and can be successfully parsed. Generalization capability of the parser is then measured by evaluating its performance on the testing set.

5.1 Experiment 1

As a preliminary evaluation, the performance of the two syntactic parsers, BPN and CPP, are compared using the example quoted in (Ho & Chan, 1994). There are totally 34 training examples and 5 testing cases. The simulation parameters and results have been summarized in Table III.

	Length of representations			*Training parameters*[4]		*Successful Parse %*	
	Terminal	*SRAAM*	*RAAM*	*Rate*	*Momentum*	*Training*	*Testing*
BPN[5]	m = 21	n = 31	m = 21	RAAM : 0.005	0.75	100%	40%
				SRAAM : 0.0025	0.75		
CPP	m = 22	n = 125	N/A	0.0025	0.95	100%	80%

Table III. The simulation parameters and results of experiment 1

In both cases, terminals are represented by orthogonal binary strings in order to minimize the mutual-interference between their codings. The symbols m and n are interpreted as in the corresponding Figures 6 and 7. The number of hidden layer units r in the Parsing Network of the BPN is 15 (see Figure 6). For the CPP, all internal nodes are represented by the special terminal \$. A point to note is that far more units are dedicated for the SRAAM representations in the CPP since there are two types of sequences to encode - the sentences and the preorder traversals of the parse trees.

5.2 Experiment 2

In experiment 2, we shall focus our attention on the CPP only. Training sentences and testing sentences are generated using the context-free grammar below which is being adopted from (Pollack, 1990) :

Sentence	*Noun Phrase*	*Verb Phrase*	*Prepositional Phrase*	*Adjectival Phrase*
s → np vp	np → D ap	vp → V np	pp → P np	ap → A ap
s → np V	np → D N	vp → V pp		ap → A N
	np → np pp			

Each syntactically well-formed sentence is generated using s as the root. By limiting the depth of recursive applications of rules to 5 levels only, a total of 112 sentences are obtained. Among them, 80 sentences are randomly selected for training while the remaining 32 sentences are reserved for the testing phrase. As before, the symbols can be thought of representing syntactic categories - N for noun, V for verb, P for preposition, A for adjective, D for determiner, np for noun phrase, vp for verb phrase, pp for prepositional phrase, ap for adjectival phrase and s for sentence respectively. In other words, the sentences thus obtained represent generic patterns of grammatical sentences (with respect to the grammar above).

Three versions of the CPP have been implemented. In CPP1, all internal nodes of the parse trees are uniquely represented by the special terminal symbol \$ while in CPP2 and CPP3, the actual identities of the internal nodes are distinguished and a separate representation is assigned to each. On the other hand, in CPP1 and CPP2, the network needs to learn parsing complete sentences only, but in CPP3, the extension as considered in Section 4.5 is implemented. In addition to complete sentences, the parser is also trained to parse phrases to produce the corresponding sub-parse-trees. A point to note is that, from CPP1 to CPP3, there is a gradual increase in the amount (and quality) of linguistic knowledge assumed.

The representations of the terminals (including the internal nodes and $) have been shown in Table IV. Again, orthogonal binary strings are used. The error injection method as proposed by Stolcke and Wu (Stolcke & Wu, 1992) has been adopted. An extra bit is attached as the leftmost bit of each coding which is initialized to 0 for a terminal while for a non-terminal (that is, a SRAAM code), its value is forced to 1 during training. Summarized in Table V are the training parameters and the results. As shown, the generalization performance of the CPP improves as the amount of linguistic knowledge incorporated increases.

	Length = 19 units
N	0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
V	0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
P	0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0
D	0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0
A	0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0
$	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1

(a) CPP1

	Length = 23 units
N	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
V	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
P	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0
D	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0
A	0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0
s	0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0
np	0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
vp	0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
pp	0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ap	0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

(b) CPP2 and CPP3

Table IV. The representations of the terminals used in experiment 2

	Length of representations (cf. figure 7)		*Training parameters*		*Successful parse %*	
	SRAAM	*Terminal*	*Rate*	*Momentum*	*Training*	*Testing*
CPP1	n = 200	m = 19	0.00125	0.9	100%	75%
CPP2	n = 220	m = 23	0.00225	0.9	100%	81.25%
CPP3	n = 230	m = 23	0.00225	0.9	100%	93.75%

Table V. The training parameters and results of experiment 2.
The symbols m and n (for lengths of representations) are interpreted as in Figure 7.

6. Parsing Erroneous Sentences

Recall at the beginning of this paper, we have claimed that one advantage of connectionist parsing over the classical AI approaches is that neural network processing is inherently robust. Therefore, we can expect a connectionist parser such as the CPP will exhibit such error-recovery capability by generating probable parses for erroneous or damaged sentences. This characteristic is especially appealing when natural language is being dealt with. Incomplete or ungrammatical sentences are occurring so often (especially in spoken conversation and as psycholinguistic phenomena) that every successful parser implementation should be able to tackle in order to be practical and useful.

What we mean by a probable parse is that the parse tree generated is syntactically well-formed, that is, it can be derived (step-by-step from the starting non-terminal, which is s in our case) using the production rules of the grammar. Doubtless, being a probable parse should not be the whole story. When a sentence is only "trivially disturbed" (implying that the errors committed are insignificant, for example, a certain word is missing, a pair of neighboring words are swapped or a word is wrongly spelled) which intuitively should not totally prevent the perception of its meaning, the parser would have to generate not only a probable parse, but a "sensible" parse.

As an illustration, let's consider a sentence S = <D A N V D N> which corresponds to the parse tree ((D (A N)) (V (D N))). Suppose the erroneous sentence S' = <D A V D N> is resulted by deleting the third terminal N from S. Then, only parse tree (ii) of the following two probable parses for S' can be a sensible parse, based on the argument that the sentence <D A N V D N> corresponding to the parse tree (ii) is closer (or more similar) to S' than the sentence <D N V D A N> corresponding to the parse tree (i) :

(i) ((D N) (V (D (A N))))
(ii) ((D (A N)) (V (D N)))

The error-recovery capability of the BPN has only been briefly investigated in (Ho & Chan, 1994). Here, a systematic study will be performed to evaluate the effectiveness of the CPP in tackling with erroneous sentences. To begin with, let S_TRAIN be the set of the 80 training sentences used in experiment 2 and denote by TERMINAL the set of all terminals/leaves {D, N, V, P, A}. A set of erroneous sentences S_ERROR will then be obtained which is the union of the following four subsets :

(i) S_WRONG_ELEMENT (SW) - erroneous sentences with one wrong element
For every S_k=<s_1 s_2 ... s_p> in S_TRAIN, produce one $SW_{k,i}$ for each s_i (i=1, 2, ..., p) by replacing s_i by a terminal t randomly selected from TERMINAL where $s_i \neq t$. As a result, p SW's are obtained where p is the length of S_k. The set S_WRONG_ELEMENT collects all $SW_{k,i}$'s produced in this way.

(ii) S_EXTRA_ELEMENT (SE) - erroneous sentences with one extra element
For every S_k=<s_1 s_2 ... s_p> in S_TRAIN, produce one $SE_{k,i}$ by inserting between s_i and s_{i+1} (i=1, 2, ..., p-1) one terminal t randomly selected from TERMINAL. Also, produce $SE_{k,0}$ by attaching one $t \in$ TERMINAL at the beginning of S_k and $SE_{k,p}$ by appending one $t \in$ TERMINAL to the end of S_k. Totally, (p+1) $SE_{k,i}$'s are generated for S_k in this way. All of them will be collected in S_EXTRA_ELEMENT.

(iii) S_MISSING_ELEMENT (SM) - erroneous sentences with one missing element
For each S_k=<s_1 s_2 ... s_p> in S_TRAIN, produce one $SM_{k,i}$ by deleting s_i (i=1, 2, ..., p) from S_k. Altogether p $SM_{k,i}$'s will be given rise for each S_k which are contained in the set S_MISSING_ELEMENT.

(iv) S_SWAP_ELEMENT (SS) - erroneous sentences with two neighboring elements swapped
For each S_k=<s_1 s_2 ... s_p> in S_TRAIN, produce one $SS_{k,i}$ by swapping the two elements s_i and s_{i+1} (i=1, 2, .., p-1) which are neighboring to each other. A total of (p-1) SS's are thus produced for each S_k. The set S_SWAP_ELEMENT is constructed by collecting all these $SS_{k,i}$'s.

Thus each subset is constructed by systematically injecting one kind of noise to different positions of each original sentence in S_TRAIN. In this way, we can have a better understanding of the sensitivity of the CPP to different kinds of errors. For instance, the syntactically well-formed sentence S_k=<D N V> may give rise to the following erroneous sentences :

Error Type	*Erroneous Sentences*
SW	$SW_{k,1}$=<N N V> $SW_{k,2}$=<D V V> $SW_{k,3}$=<D N P>
SE	$SE_{k,0}$=<N D N V> $SE_{k,1}$=<D D N V> $SE_{k,2}$=<D N A V> $SE_{k,3}$=<D N V V>
SM	$SM_{k,1}$=<N V> $SM_{k,2}$=<D V> $SM_{k,3}$=<D N>
SS	$SS_{k,1}$=<N D V> $SS_{k,2}$=<D V N>

The erroneous sentences in S_ERROR will then be processed again by each of CPP2 and CPP3 respectively (as trained in experiment 2). As mentioned earlier, an erroneous sentence $S' \in$ S_ERROR is said to be successfully tackled if a sensible parse tree is obtained such that the sentence S^0 corresponding to the parse tree obtained should "look very much alike" S'. But to realize this rather abstract criteria of minimum distortion, a mechanistic decision procedure should be established.

Represent by Length(S') the number of elements in the erroneous sentence S', and Length(S^0) the number of elements in the sentence S^0 corresponding to the parse tree T^0 obtained, we first require the difference between Length(S') and Length(S^0) be at most 1 only. In this way, the parse tree T^0 obtained and its corresponding sentence S^0 cannot deviate much from the original erroneous sentence S'.

Next, a dummy element (abbreviated by "?") will be added to the shorter sentence (if S' and S^0 do differ in length) in order to bring the two sentences to the same length before they are compared/matched in an element-by-element manner. Note that there can be more than one way to add the extra element. Every possible combination will be enumerated and the matching process is repeated for each. For example, if S'=<D N A V D N> and S^0=<D N V D N>, then Length(S')-Length(S^0)=1 and one "?" will be added to S^0 in the following different ways :

S^0_1 = <? D N V D N>

S^{0}_{2} = <D ? N V D N>
S^{0}_{3} = <D N ? V D N>
S^{0}_{4} = <D N V ? D N>
S^{0}_{5} = <D N V D ? N>
S^{0}_{6} = <D N V D N ?>

In a certain sense, this process effectively aligns S^{0} with S’ in all possible ways and each of them will be compared with S’ in an element-by-element style. If any one of S^{0}_{i} matches S’, the parse tree T^{0} obtained is said to be sensible and will be accepted.

A critical parameter in this matching process is the maximum number of mismatches allowed between a S^{0}_{i} and S’ (denoted by max_mismatches). Doubtless, a parse tree will be more likely to be accepted as sensible if a larger value is assumed for max_mismatches. In our experiments, two cases have been considered which correspond to max_mismatches=1 and max_mismatches=2 respectively. A point to note is that for S’ $\in$ S_SWAP_ELEMENT, we should account for the case when S^{0} and T^{0} correct S’ by bringing the two swapped elements in S’ back to their original order (in other words, S^{0} is the original sentence S from which S’ is derived). The number of mismatches in that case will be 2.

The result for each of the four types of errors has been summarized in Table VI. As shown, CPP seems to be most proficient in correcting erroneous sentences with one wrong element. Among the four types of errors, we claim that in natural language, S_WRONG_ELEMENTS and S_MISSING_ELEMENTS should occur more frequently than the others. They correspond to the cases when words are wrongly spelled and words are missed (especially the determiner D like “the”, “a” and “an”) respectively. On the other hand, CPP3 which has learned to parse phrases in addition to complete sentences is more robust when erroneous sentences are concerned.

		max_mismatches = 1				max_mismatches = 2			
		No Loop Back		Loop Back Once		No Loop Back		Loop Back Once	
Error type	Total	CPP2	CPP3	CPP2	CPP3	CPP2	CPP3	CPP2	CPP3
SW	853	90.15%	94.61%	90.27%	94.61%	91.21%	94.72%	91.21%	94.72%
SM	933	54.51%	54.63%	54.51%	54.63%	69.64%	70.93%	69.75%	70.93%
SE	853	32.79%	33.44%	32.79%	33.44%	46.62%	50.80%	46.84%	50.80%
SS	773	N/A	N/A	N/A	N/A	63.65%	66.88%	66.88%	67.01%

Table VI. Percentage of erroneous sentences which can be sensibly parsed/recovered

Just as a remark, for some erroneous sentences, more than one probable parse can be considered as sensible. Take for instance, the erroneous sentence S’=<D A N A V D N> which is (originally) obtained by swapping the second and third element of the sentence S=<D A A N V D N> can have the two sensible parses ((D (A (A N))) (V (D N))) and ((D (A N)) (V (D N))). While the former is the desired parse tree corresponding to the original sentence S, the latter simply treats the fourth element A of S’ as superfluous which when deleted, will recover another “legitimate” sentence S"=<D A N V D N> corresponding to the parse tree ((D (A N)) (V (D N))).

The error-recovery capability can be further enhanced. Given an erroneous sentence S', a parse tree T^{0} is obtained which corresponds to the sentence S^{0}. If S^{0} is rejected by the matching process mentioned above, it will be fed back (looped back) to the sentence encoder side of the CPP again and the parsing process is re-attempted. In some sense, S^{0} can be considered as the first approximation of S’ and the original syntactically well-formed sentence S from which S’ is derived from. Another parse tree T^{1} is then generated and the sentence S^{1} thus obtained is subject to the matching procedure again. By repeating this process further, we can obtain T^{2} and S^{2}, T^{3} and S^{3}, etc. The rationale is that the desired parse tree T which corresponds to the original sentence S can be approached through this iterative correction procedure.

Doubtless, in same stage during this iterative process, improvement may cease and further looping back is simply a waste of effort. From our experience, looping back once (to obtain T^{1} and S^{1}) will be enough since further improvement will be trivial. With reference to Table VI, “No Loop Back” and “Loop Back Once” correspond to S^{0} and S^{1} respectively.

To better understand the error-correction capability, we will take one step further to classify each erroneous sentence in S_ERROR by the location at which the error occurs. Denote by Length(S) the number of elements in the original sentence S (with no error) and that the error involves the p-th element[6] of S, the precise classification will be :

if $0 \le \frac{p}{\text{Length(S)}} < 0.25$, the error is said to be located in the 1st quarter of S

if $0.25 \le \frac{p}{\text{Length(S)}} < 0.5$, the error is said to be located in the 2nd quarter of S

if $0.5 \le \frac{p}{\text{Length(S)}} < 0.75$, the error is said to be located in the 3rd quarter of S

if $0.75 \le \frac{p}{\text{Length(S)}} \le 1$, the error is said to be located in the 4th quarter of S

The result has been shown in Figure 12. In almost all cases, the effect of the location of the error on the error-recovery capability will be more profound when it is nearer to the end of the sentence. While this result can be expected naturally since the sentence is read (and encoded) by the SRAAM from left to right and that confluent inference is only enforced after the whole sentence has been scanned, the position effect thus reflected is unrealistic and undesirable. We will return to this issue again in Section 8.

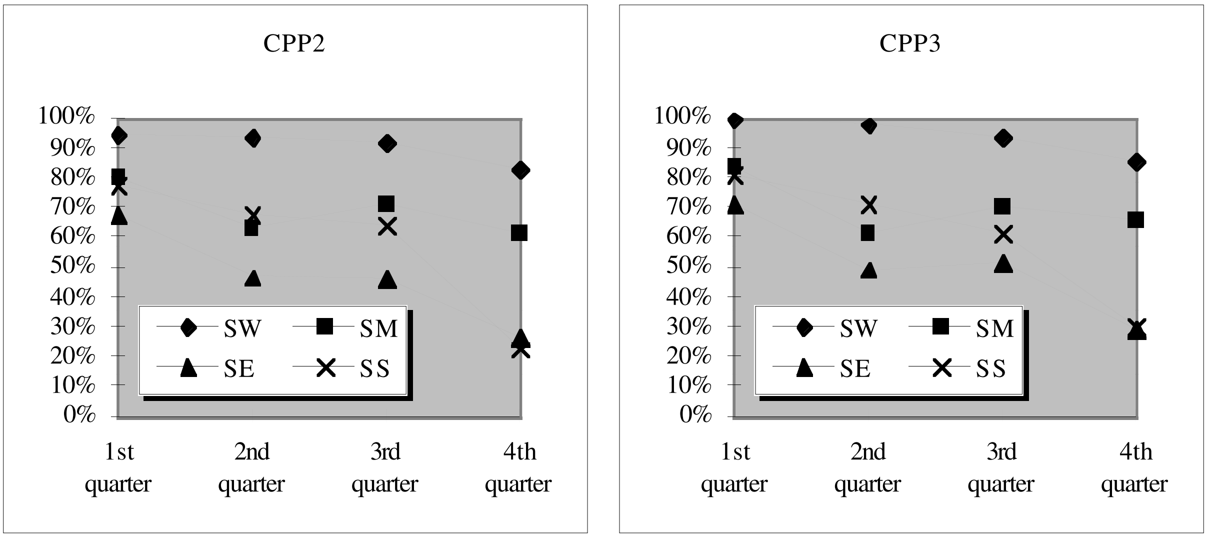


Figure 12. The effect of the location of the error on the error-recovery capability of the CPP
(x-axis: the location of the error; y-axis: the percentage of erroneous sentences successfully parsed)

Still one more factor which influences the error-recovery capability is worth of further exploration - the length of the sentence. The result has been summarized in Figure 13. As shown, no unique trend can be spotted. In some cases (such as the SS's), the robustness seems to increase with the length of the erroneous sentence, while in the others, the longer the sentence, the less likely the error can be successfully corrected.

These two apparently contradictory phenomena can be explained as follows. On the one hand, the longer the sentence involved, the less significant the effect of the error has on parsing the sentence, in the sense that the longer the sentence, the greater proportion of it is correct. This helps to explain the case of the SS sentences. On the other hand, since training cannot be exact, error thus seems to be unavoidable. During parsing, a sentence is recursively encoded and then recursively decoded to give its parse tree. A natural consequence is that, if error exists, as it really does when there is noise in the original input sentence, it will be accumulated during these two recursive processes. If the length of the sentence happens to be too long, this imprecision simply becomes too significant to be tolerable because of the depth of the recursion and avoids the proper interpretation and parsing of the sentence. For the other three types of errors, this latter factor seems to dominate.

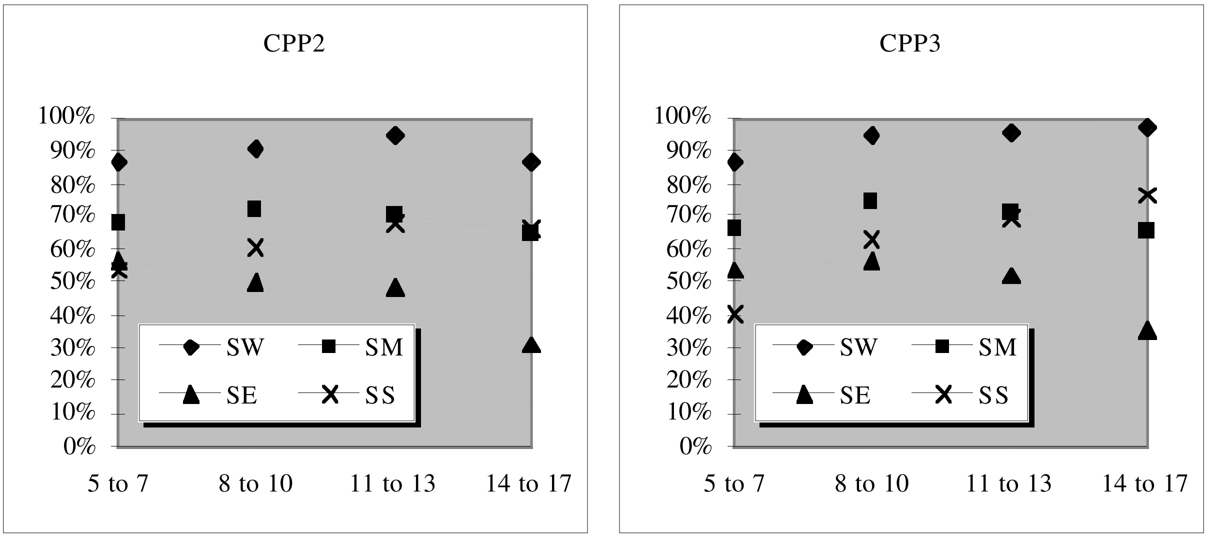


Figure 13. The effect of the length of the erroneous sentence on the error-recovery capability of the CPP (x-axis: the length of the erroneous sentence; y-axis: the percentage of erroneous sentences successfully parsed)

7. Syntactic Disambiguation

A word W occurring in a sentence S is said to be ambiguous if it can be interpreted in multiple ways. In general, these different interpretations correspond to different possible senses (meanings) of W. For example, the word "star" in the sentence "the astronomer married the star" can be perceived in two ways : an astronomical body, or a popular, famous person (e.g. singer, actress, etc.) in the show business (Pollack & Waltz, 1985).

Occasionally, each of the possible senses may correspond to a different syntactic interpretation of the ambiguous word W. For example, the syntactic category of the word "duck" is different for its two occurrences in "they eat a duck" and "they duck their heads". Ambiguity of this type can be resolved by considering the syntactic categories of those neighboring words collocated with the ambiguous word W in the sentence S they co-occur (e.g. "a" in the first sentence and "they" in the second). Precisely, they help to restrict the legitimate syntactic class of W as permitted by the grammar from which S is derived. Having the right syntactic class determined can then fix a single meaning of W appropriate to its usage or interpretation in S. Therefore, the word "duck" in the sentence "they eat a duck" should be interpreted as a noun since it follows the determiner "a". In other words, it refers to an animal rather than an action.

Since this resolution process exploits only the syntactic knowledge of the sentence and the grammar, it is specifically referred as syntactic disambiguation. This is different from the case of the sentence "an astronomer married the star" in which the two possible senses of "star" both interpret the word as a noun. Therefore, it cannot be disambiguated using syntactic relationship. Instead, semantic constraint must be called for. As in this sentence, the fact that the verb "married" assumes a human direct object effectively eliminates the first possibility.

A point to note is that in the syntactic disambiguation mentioned above, only words within the same sentence as the ambiguous word take part in the resolution process. In fact, very often, only neighboring words of it will be significant. From another point of view, only local information is involved.

We claim that the same syntactic disambiguation capability is possessed by the CPP. To demonstrate this, an experiment has been designed. As in the previous section, we inject noises to the training sentences in S_TRAIN of experiment 2 to create another set of sentences S_AMBIGUOUS for testing. Denote again by TERMINAL the set of terminal symbols {D, N, V, A, P}. We first consider the case when there is only one ambiguity embedded. Here comes the precise construction of S_AMBIGUOUS. For every training sentence $S_k=<s_1\ s_2\ \ldots\ s_p>$ in S_TRAIN, we produce a set of ambiguous sentences S_k_AMBIGUOUS=$\{S_{k,\{i\}}\}$ where i=1..p. Each sentence $S_{k,\{i\}}$ is constructed by replacing the i-th terminal s_i in S_k with a new terminal s_i/t where t is a terminal randomly selected from the set TERMINAL with $s_i \neq t$. This s_i/t then represents the dual identity (or dual syntactic category) of an ambiguous word W which appears as the i-th word (or terminal) of the sentence S_k. Altogether, p ambiguous $S_{k,\{i\}}$ sentences are given rise for each S_k (where p is the length of S_k).

As an illustration, suppose S_k=<D N V D N> and it happens that t=V, then an ambiguous sentence $S_{k,\{2\}}$=<D N/V V D N> will be included in S_k_AMBIGUOUS. Finally, S_AMBIGUOUS is defined by :

$$S\_AMBIGUOUS = \bigcup_{k=1}^{card(S\_TRAIN)} S_k\_AMBIGUOUS$$

where card(S_TRAIN) denotes the cardinality of the set S_TRAIN. Representations of such ambiguous terminals as s_i/t are obtained by OR-ing the corresponding representations of the terminals s_i and t. For example, assuming the codings of the terminals defined for the CPP in experiment 2 (see Table IV), if s_i=N and t=V, s_i/t=N/V will be defined as :

$$\begin{array}{rl} s_i\ (N) = & 0\ 111\ 000\ 000\ 000\ 000\ 000 \\ t\ (V) = & 0\ 000\ 111\ 000\ 000\ 000\ 000 \\ \hline s_i/t\ (N/V) = N\ OR\ V = & 0\ 111\ 111\ 000\ 000\ 000\ 000 \end{array}$$

As before, CPP2 and CPP3 are called upon to process the sentences in S_AMBIGUOUS to resolve the ambiguities involved. From another point of view, each $S_{k,\{i\}}$ can be perceived as an erroneous sentence. But this time, we need a more strict criterion of successful recovery. For each $S_{k,\{i\}}$ in S_AMBIGUOUS, the sentence $S^0_{k,\{i\}}$ corresponding to the parse tree $T^0_{k,\{i\}}$ produced should be of the same length as $S_{k,\{i\}}$ and that they should differ in one terminal only - the ambiguous one. Moreover, to be claimed successful, the correct identity of the ambiguous terminal should be resolved. In other words, no mismatch is allowed. With reference to our earlier example in which $S_{k,\{2\}}$=<D N/V V D N>, the parser should be able to produce the parse tree $T^0_{k,\{2\}}$=((D N) (V (D N))) which corresponds to the sentence $S^0_{k,\{2\}}$=<D N V D N>. Effectively, the ambiguity involved in the terminal/word N/V is resolved to give the correct identity N.

To obtain a clearer picture, the disambiguation capability is further evaluated by parsing sentences of two ambiguities $S_{k,\{i_1,i_2\}}$ (where $i_1 \neq i_2$) and sentences of three ambiguities $S_{k,\{i_1,i_2,i_3\}}$ (i_1, i_2 and i_3 being all different) also. Still one more factor is worth of consideration. Originally, each ambiguity involves two alternatives only (e.g. D/N). As a comparison, another set of experiments are performed in which three alternatives are generated for each ambiguity (e.g. D/N/V). In the extreme case, if each ambiguity is offered the total range of possible alternatives, which is equal to the number of terminals in the set TERMINAL (in our case, it is equal to 5), then the problem is effectively reduced into one of filling in the missing word(s).

The results of all the different cases have been summarized in Table VII. As shown, the performance is satisfactory. A simple observation should reveal that this syntactic disambiguation process is basically very similar to the error-recovery task of sentences with wrong terminal symbols. Thus the good performance is natural and can be expected.

		1 ambiguity		*2 ambiguities*		*3 ambiguities*	
	Total	CPP2	CPP3	CPP2	CPP3	CPP2	CPP3
2 alternatives	853	98.24%	99.65%	95.90%	98.36%	91.89%	97.31%
3 alternatives	4395	96.60%	98.83%	89.19%	95.31%	72.70%	91.37%
5 alternatives	14545	94.26%	96.13%	71.26%	90.17%	40.18%	77.34%

Table VII. Syntactic disambiguation by the CPP

8. Discussions and Conclusions

In this paper, we have proposed a new connectionist parsing model - the Confluent Preorder Parser (CPP). Compared to the traditional approaches, there are two major innovations.

Firstly, according to the classical view, the parsing problem is tackled as a transformational process from a sequential sentence to its corresponding hierarchical parse tree. However, the CPP takes a different perspective. Stemming on the equivalence between a symbolic parse tree and its preorder traversal sequence, parsing is achieved by transforming from a sentence sequence to the preorder traversal sequence of its corresponding parse tree, instead of the parse tree itself.

With preorder traversals, the structure of a parse tree has been made explicit via the use of the internal nodes or \$. Parse trees corresponding to similar sentences (and thus having some common sub-trees) will share certain fragments in their preorder traversal sequences. What is significant is that this type of similarity will be manifested at the surface level, as different from the BPN in which the structure of a parse tree is only implicitly reflected by the way (and the order) the different terminals and components are "amalgamated" together by the RAAM to form the parse tree. Since neural networks generalize by interpolating or extrapolating between the novel sentence pattern and the training patterns, this surface similarity in the representations of the preorder traversals of the parse trees will undoubtedly promote the overall generalization performance of the parser.

But how important does this factor actually (and quantitatively) mean to the overall performance? To answer this question, a control experiment has been set up. This time, parse trees instead of their preorder traversals will be dealt with directly and they are encoded using a structured RAAM network as in the BPN. Parsing is achieved by holistically transforming from the SRAAM representation of the input sentence to the RAAM representation of its corresponding parse tree. Again, confluent inference has been applied as in the CPP. The same problem set in experiment 2 is then re-attempted using this adapted parser. Training now takes less time to converge. Yet, generalization performance drops to 28.125% only while the original CPP scores at least 75% in the same test. In other words, by exploiting the equivalence between a parse tree and its preorder traversal, generalization is at least boosted by a factor of 2.67 apparently.

The second innovation of the CPP is that the output/answer of the parsing process is not produced incrementally. Instead, a separate SRAAM network is trained for encoding the sentence sequences and preorder traversal sequences respectively, and parsing is implemented as a holistic mapping function from the SRAAM representation of the input sentence to the SRAAM representation of the output preorder traversal sequence desired, **directly and in a "one-shot" manner**. Apparently, the trained parser network operates without involving any step-by-step rational inference.

A point to emphasize is that actually, there is a continuum of different techniques for parsing (and in fact for natural language understanding in general). On one extreme, we find techniques which are characterized by having a well-defined algorithm guiding the derivation of the parse tree. Techniques of this type are considered as incremental, in the sense that it usually takes multiple steps to obtain the final answer. What is important is that these steps are supposed to be followed in a strict order and very little exception is usually allowed. An intermediate data structure (e.g. an intermediate parse tree) will be built up in every step which serves as input to the next step. More importantly, there is a strong linguistic assumption behind these language processing algorithms, for example, the Chomskyan generative grammar, the case grammar, etc. Almost all symbolic AI parsing systems (e.g. chart parser, see Allen, 1995) are inherently incremental.

Disadvantages of the incremental approaches are evident. Their parsing algorithms may not be psychologically plausible in the sense that they are incapable of reflecting the real processing mechanism underlying actual language usage. And the point is, very little has been known about this mechanism in fact. But it seems that natural language perception cannot be a totally conscious and rational process. Rule-based incremental techniques simply cannot account for the speed, generative capability and extensive flexibility of language usage. The linguistic assumptions which the incremental approaches are based on are originally postulated for explaining the competence of natural language only, but not to account for its performance. Consequently, they may not be suitable and efficient for implementation as a von Neumann processing system (see Kwasny & Faisal, 1992, for example). In addition, many of the incremental systems treat the different dimensions or modalities of language separately and independently - syntax processing is done first, then semantics, and finally pragmatics. Doubtless, language perception is an integrated process and that any separation of information sources may lead to information loss.

On the other extreme of the continuum, we have holistic parsing techniques as exemplified by the CPP. They avoid proposing any model or explanation about the actual mechanism underlying language perception. Instead, it is specified as a "black-box" with input and output. The minimum of linguistic knowledge is assumed. Often, behaviorally-related data (which may even be naturally observable) will be used as the input and the output. As in the CPP, a sentence is used as the input and a parse tree is specified as the output. The exact specification of the actual transformation from the former to the latter is omitted. This is exactly what Figure 1 has postulated. The grammatical knowledge and regularity relevant to the parsing (or language understanding) process will be "learned" by exposure to samples of pairs of sentences and parse trees. The underlying belief of this kind of models is that language usage is conventional and can be guided by experience.

Between these two extremes, there is much room to accommodate different design alternatives. Some of these hybrid approaches apply connectionist techniques to implement some or all of the components of the incremental parser. Examples include the PARSEC model (Jain & Waibel, 1990), the SPEC model

(Miikkulainen, 1993), and the connectionist deterministic parser proposed by Kwasny and Faisal (Kwasny & Faisal, 1992).

Simulation results have shown that the CPP can generalize well to novel sentences. Moreover, it exhibits some error-recovery capability when presented with noisy inputs and we have tailored an experiment to show how this capability can be further extended to achieve syntactic disambiguation. A systematic study has been conducted to study the factors that can affect the effectiveness of it. This robustness is unique to neural network processing and what is important is that it has been achieved as a by-product and **without extra effort**. For problems that involve representing and/or manipulating hierarchical data structure(s), which used to be "hard" for neural networks, the success of the CPP has demonstrated a viable alternative. The combination of the two techniques - confluent inference and the linearization of hierarchical data structure, may be capable of offering an efficient connectionist solution.

Several fruitful extensions are worth of further consideration for the CPP. Firstly, the position effect as mentioned in Section 6 is both undesirable and psychologically unrealistic. To deal with it, we can append to each sentence a terminator marker which is implemented as a special terminal symbol (in much the way as $ and @). Effectively, it plays the role of a period/full-stop. The CPP maps the resulting SRAAM representation of the complete sentence to the SRAAM representation of the preorder traversal only after the terminator marker (that is, the full-stop) has been read in and processed.

Secondly, a simple sentence such as <D A N V> can be a prefix (or fragment) of another longer sentence such as <D A N V D N>, and that the parse tree of the latter - ((D (A N)) (V (D N))), can be constructed from the parse tree of the former - ((D (A N)) V), by replacing the terminal V with the sub-tree (V (D N)). It thus seems to be intuitively natural and computationally desirable to have the CPP to learn to parse the simpler sentences first. Only after the CPP has managed to process this subset of shorter and simpler sentences (say when training is 100% successful) can learning proceed to include the more complex sentences also. This suggests the use of the incremental learning technique (Elman, 1991).

Finally, although we have emphasized that parsing by the CPP is implemented as a holistic transformation between SRAAM representations, it can also be analyzed as a step-by-step rational inference process by perceiving its operations as governed by a finite automata. Based on this formalism, we can have a clearer picture of how robust parsing and generalization are achieved. On the other hand, it simply places a theoretical limit on the representation power of the Confluent Preorder Parser and other similar models.

Notes

1. Occasionally, we shall adopt the Lisp-like notation as a compact representation for trees.
2. A possible filler for the sentence pattern <D N V D N P D N> may be "The boy hits the man with a hat."
3. The special terminal @ will have the same status as other terminals such as D or N.
4. The rate and momentum in each case are fixed during training. This is also true for experiment 2.
5. The data for the BPN as quoted in Table III are adopted from the original paper (Ho & Chan, 1994).
6. For S_WRONG_ELEMENT, it means that the p-th element is wrong and for S_MISSING_ELEMENT, it implies that the p-th element is deleted. Concerning S_SWAP_ELEMENT, it implies that the p-th element and the (p+1)-th element of the original sentence S are swapped where $1 \le p \le \text{Length}(S)-1$. The situation is a little bit complicated for S_EXTRA_ELEMENT. If the extra element is inserted before the p-th element of the original correct sentence S (where $1 \le p \le \text{Length}(S)$), the error is said to occur at the p-th position. In case the extra element is inserted at the end, the error is considered to be located in the 4th quarter also.

References

Allen, J. (1995) *Natural Language Understanding*. The Benjamin/Cummings Publishing Company, Inc.

Blank, D. S., Meeden, L. A. & Marshall, J. B. (1992) Exploring the Symbolic/Subsymbolic Continuum: A Case Study of RAAM. In J. Dinsmore (Ed.), *The Symbolic and Connectionist Paradigms: Closing the Gap*, Hillsdale, NJ: Lawrence Erlbaum Associates, pp.113-148.

Chalmers, D. J. (1992) Syntactic Transformations of Distributed Representations. In N. Sharkey (Ed.), *Connectionist Natural Language Processing*, Intellect Books, pp.46-55.

Chrisman, L. (1991) Learning recursive distributed representations for holistic computation. *Connection Science*, 3, 345-366.

Elman, J. L. (1991) *Incremental learning, or The importance of Starting Small.* CRL Technical Report 9101, Center for Research in Language, University of California, San Diego.

Ho, K. S. Edward & Chan, L. W. (1994) Representing Sentence Structures in Neural Networks. *Proceedings of the International Conference in Neural Information Processing Systems*, Vol. 3, pp.1462-1467.

Jain, A. N. & Waibel, A. H. (1990) Incremental Parsing by Modular Recurrent Connectionist Networks. In D. S. Touretzky (Ed.), *Advances in Neural Information Processing Systems 2*, San Mateo, CA: Morgan Kaufmann, pp. 364-371.

Kwasny, S. C. & Faisal, K. A. (1992) Symbolic Parsing via Subsymbolic Rules. In J. Dinsmore (Ed.), *The Symbolic and Connectionist Paradigms: Closing the Gap*, Hillsdale, NJ: Lawrence Erlbaum Associates, pp.209-236.

Kwasny, S. C. & Kalman, B. L. (1993) *Tail-Recursive Distributed Representations and Simple Recurrent Networks*. Technical Report WUCS-93-52, Department of Computer Science, Washington University.

Maskara, A. & Noetzel, A. (1993) Forced Simple Recurrent Neural Networks and Grammatical Inference. *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*, Hillsdale, NJ: Lawrence Erlbaum Associates, pp.420-425.

Miikkulainen, R. (1993) *Subsymbolic Case-Role Analysis of Sentences with Embedded Clauses*. Technical Report AI93-202, Department of Computer Sciences, the University of Texas at Austin.

Pollack, J. B. (1990) Recursive Distributed Representations. *Artificial Intelligence*, 46, 77-105.

Pollack, J. & Waltz, D. (1985) Massively parallel parsing: A strongly interactive model of natural language interpretation. *Cognitive Science*, 9, 51-74.

Rumelhart, D.E., Hinton, G. E., & Williams, R. J. (1986) Learning internal representations through error propagation. In D. E. Rumelhart, J. L. McClelland, & The PDP Research Group (Eds.), *Parallel distributed processing: Experiments in the microstructure of cognition*, Vol. 1, Cambridge: MIT Press, pp.318-362.

Stolcke, A. & Wu, D. *Tree matching with recursive distributed representations*. Technical Report TR-92-025, International Computer Science Institute, Berkeley.