A Hybrid Approach to Parsing Using Neural Network and Shift-reduce Parser

Swaroop S. Vattam
Artificial Intelligence Center
The University of Georgia
Athens, Georgia 30602-7415 U.S.A.

2001 May 04

1 Overview

This paper presents a hybrid approach to parsing English sentences. The approach involves the use of a traditional Shift-reduce parser in conjunction with a Neural Network (NN) to accomplish the task of parsing. The parsing is now a two stage process. Given an input sentence, the Shift-reduce parser is consulted first to do the parsing. If it fails, a well trained neural network takes over from where the Shift-reduce parser had left and tries to complete the parsing. This approach has many advantages. Firstly, the set of rules in the Shift-reduce parser need not be complete, or even very elaborate for that matter. We know that if it fails there is something to follow. Secondly, training the neural network in this domain is very easy, with very little human supervision required. One has to just supply it with a set of correct English sentences. It can also be argued, at least in theory, that this approach generalizes to indefinitely long sentences containing indefinitely many embedded structures.
2 Introduction

Despite decades of research, parsing remains a difficult computation that often results in incomplete, ambiguous structures; and computational grammars for natural languages remain notably incomplete (Simmons and Yu, 1992). Also, most traditional parsers adhere to context-free grammar representation (sometimes augmented with features and transformations) and it is conceded that context-free grammar alone is not powerful enough to account for the syntactic analysis of natural languages (Simmons and Yu, 1992). This work suggests that a solution may be found in the use of a hybrid approach that involves a Shift-reduce parser and a well trained neural network. The Shift-reduce parser encodes context-free rules and the neural network, upon training, accounts for context-sensitivity. Together, the entire system becomes very powerful.

Shift-reduce parsing is sufficiently addressed in NLP literature. On the contrary, Neural Networks seem a little out of place where natural language processing is concerned. There are many issues to be dealt with, like:

- Natural language sentences can be indefinitely long and simple neural networks are fixed-size structures. The number of input, hidden, and output nodes are frozen once a suitable architecture is chosen. It would be unreasonable to create a neural network that would accept sentences up to one hundred words (requiring one hundred input nodes) when the average length of the sentences is only twenty. This is because, the computational complexity of the network increases at least by the square of the number of nodes involved (Simmons and Yu, 1990). So, we are wasting a lot of computational power for handling very long sentences that would be encountered very rarely. And, so far it seems that there will always be a limit on the length of the sentence that can be parsed as we cannot have a network with an infinite number of input nodes. This work addresses this issue and implements a novel method, proposed by Simmons and Yu (1990), that allows us to parse indefinitely long sentences using a fixed-structure neural network.

- Neural networks are sub-symbolic systems and we are working in a symbolic domain. This raises the issue of mapping sentences (symbolic data) into numerical data and vice versa.

- For the neural network to be able to recognize and parse a class of
sentences, it must be shown a lot of sentences which belong to this class. The sentences belonging to a class will have the same syntactic structure. On the surface this seems like a simple issue to resolve. Just feed the neural network with a huge set of syntactically correct sentences. Since we know that all these sentences are correct, the training would require very little human intervention and it doesn’t matter if the training process runs into many days. However, this is not such a simple proposition. Large, real-world documents (the source of sentences) need a lot of pre-processing and any noise in the data gets magnified. In the current implementation of the system, the example sentences have been generated using a small vocabulary and a limited grammar. Therefore this issue is bypassed here.

This approach works well taking advantages of simplicity, tractability and efficiency provided by the Shift-reduce parser and pattern generalization, graceful degradation and learning-by-example provided by the Neural net.

3 The Hybrid parsing system

Figure 1: The Hybrid Parsing System
The Hybrid parsing system consists of five main components. The two parsers, a stack that is used by both the parsers, a scratch pad for information exchange between the two parsers, a set of context-free/context-sensitive rules and a training corpus as shown:

![Flowchart](image)

### 3.1 A Shift-reduce parser

Shift-reduce parsing is the first stage in the whole parsing process. In this type of parsing there is a stack and an input string of syntactic classes, each of which may be indefinitely long. In the initial state the stack is empty. The first step is to shift the first element of the input string onto the stack. In the next and subsequent steps, a grammar is consulted to determine whether the top two elements of the stack reduce to a single symbol. If so, they are replaced by the new symbol; if not, a new element is shifted onto the stack.
The procedure continues until the string is empty and the stack contains only the symbol Sentence. If the parse is unsuccessful at this stage, the input string is passed to the neural net based parser along with other housekeeping information.

### 3.2 A Neural network based parser

The focus of this section is to explain how a fixed-size neural network can be used to parse sentences. The details of the neural network will be discussed in another section that talks about the network architecture and training. This approach to parsing can be best understood with an example. Suppose we are trying to parse the sentence “the young boy from London went home” represented by the string of syntactic classes as (d adj n p n v n). Further, suppose that we have a stack whose length (maximum number of elements that it can hold) is limited by the number of input nodes that we choose for the neural network, say five in this case. We can represent the initial state of the stack and the input string as follows:

\[
\text{stack state} \quad \ast \quad \text{input string} \quad \ast \quad \text{window} \\
( \text{b b b b b} \ast \text{d adj n p n v n})
\]

‘b’ here stands for a blank in the stack position. Applying the operations of the Shift-reduce parser twice produces the following subsequent states of the stack:

\[
\begin{align*}
\text{stack state} & \quad \ast \quad \text{input string} \quad \ast \quad \text{window} \\
( \text{b b b b d} \ast \text{adj n p n v n} \ast \text{n}) \\
( \text{b b b d adj n p n v n} \ast \text{n})
\end{align*}
\]

Defining the rules for context-sensitive parsing as having a left-half initial state mapping to a right-half subsequent state, gives us a basis for the neural network to map each each state of a parser into an immediately succeeding state. In other words, the context-sensitive information in the form of rules, are obtained form the current and the next state of the stack, and it looks like:

\[
Stack_i Input_i \rightarrow Stack_{i+1} Input_{i+1}
\]
We show few of the context-sensitive rules that we obtain from the above three states of the stack \(^1\)

\[(b\ b\ b\ b\ b\ *\ d\ adj\ n\ p\ n) \rightarrow (b\ b\ b\ b\ d\ *\ adj\ n\ p\ n\ v)\]

\[(b\ b\ b\ b\ d\ *\ adj\ n\ p\ n\ v) \rightarrow (b\ b\ b\ d\ adj\ *\ n\ p\ n\ v\ n)\]

\[(b\ b\ b\ d\ adj\ *\ n\ p\ n\ v\ n) \rightarrow (b\ b\ d\ adj\ n\ *\ p\ n\ v\ n\ b)\]

Therefore, one sentence leads to many rules and we can collect a sizeable number of such context-sensitive rules by parsing a number of sentences. The successive states of a parse may be shown for an indefinitely long string. The stack operations can also account for any level of embedding. These rules will be the input/output patterns to train the neural network. The left half of the rule will be the input and the right-half of the rule will be the desired output. We can train a ten input and ten output neural network to learn the patterns contained in the rules shown above. During the training, if neural network sees that the input (b\ b\ b\ d\ adj\ *\ n\ p\ n\ v\ n) has (b\ b\ d\ adj\ n\ *\ p\ n\ v\ n\ b) as the output a lot of times, it learns this pattern. At a later point, if it encounters a (b\ b\ b\ d\ adj\ *\ n\ p\ n\ v\ n) pattern, the probability that it will produce (b\ b\ d\ adj\ n\ *\ p\ n\ v\ n\ b) as the output is high.

So, we can identify two phases in the life-cycle of the neural network based parser. The training phase, and the actual operation phase. During the training phase:

- We collect a large set of valid sentences.
- For each sentence, we run it though the shift-reduce parser recording all the stages that the stack goes though. An important thing to note here is that the rules in the Shift-reduce parser used for this phase need not be complete. Incomplete rules may lead to partial parses. However, the neural network can handle it and still end up with a perfect parse. This is further explained in the section describing the context-free and context-sensitive rules.

\(^1\)The left half of the rule shows the five elements of the stack followed by the next five elements of the input string; the right half of the rule shows the next state.
• We use the states of the stack to arrive at corresponding context-sensitive rules, which form the input/output pairs for training the network.

• Upon feeding the training samples obtained from the above input/output pairs, the neural network learns the patterns latent in the classes of sentences.

During the actual operational phase, when the neural network is already trained:

• We feed the neural network with an initial symbol sequence that would look something like (b b b b b * d adj n p n v n).

• Using this sequence it will predict the next state, i.e.,
  (b b b b d * adj n p n v n).

• The output of the previous stage, (b b b b d * adj n p n v n) becomes the input to the current stage to get the output
  (b b b d adj * n p n v n).

• This process of feeding the output of one stage as the input to the next stage continues till the stopping criteria is reached. The stopping criteria could be that the parsing is complete and the neural network will output (b b b b s * b b b b b b), or, a prespecified number of iterations is reached. Placing a threshold on the number of iterations is important because the process might recurse infinitely.

3.3 Stack

The stack is a very important data structure here. It is used during all phases of the parsing process. It is central to shift-reduce parsing, which is the first stage of parsing in this system. It is also used to generate the input/output pairs for training the neural network as mentioned in the previous section.

3.4 A set of context-free/context-sensitive rules

The hybrid parsing system maintains a repository of context-free and context-sensitive rules. The context-free rules is used by the shift-reduce parser and
the context-sensitive rules are used by the neural network based parser. Typically there would be a small number of context-free rules and large number of context-sensitive rules.

In the current implementation of the hybrid parsing system, the set of context-free rules is very limited. There are about twenty rules that can handle a limited class of sentences. However, this is not a concern as the central theme of this work is based on the premise that no matter how many context-free rules that are incorporated, this set is always incomplete, and there is always a class of sentences that cannot be recognised by the context-free parser. Therefore, it is enough if we show that the class of sentences unrecognizable by the context-free parser can be parsed using the neural network based parser. From the previous section it can be noted that the same limited set of context-free rules are used to generate a set of context-sensitive rules for the neural network based parser. This set of context-sensitive rules helps the neural network based parser to parse the sentences not recognized by the context-free parser. The number of context-sensitive rules generated might be very large in number. But the advantages of having more number of context-sensitive rules as opposed to incorporating a large number of context-free rules into the grammar are:

- the context-sensitive rules are generated as opposed to the manual crafting of the context-free rules.

- there is no overhead involved in maintaining the generated context-sensitive rules. In the case of handcrafted context-free rules, the system of rules is brittle. The addition of a new rule might affect the other existing rules. Extra care has to be taken while adding new rules. The neural network based parser is capable of handling inexactness.

In the current implementation of the hybrid parsing system, the set of context-sensitive rules were generated from the example sentences in the corpus during the training phase. The context-free rules for handling sentences containing the terms ‘who’ and ‘whom’, as in “The dog chased the cat who bit the boy” or “The boy liked the girl whom the dog chased” were deliberately omitted. Then about 256 sentences containing the words ‘who’ and ‘whom’ were generated. These 256 sentences were parsed using the shift-

\footnote{context-free parser refers to a parser that makes use of a finite set of context-free rules to parse a given sentence.}
reduce parser (which did not contain the rules to handle the sentences containing ‘who’ and ‘whom’). Predictably, these resulted in partial parses, i.e., the stack did not contain the single symbol ‘s’ after the input string became empty.

But all the state transitions of the stack during the parsing process were recorded, which are as follows: ³

\[
[[d], [n,v,d,n,whox,v,d,n]]. \\
[[n,d], [v,d,n,whox,v,d,n]]. \\
[[ndash,d], [v,d,n,whox,v,d,n]]. \\
[[np], [v,d,n,whox,v,d,n]]. \\
[[v,np], [d,n,whox,v,d,n]]. \\
[[d,v,np], [n,whox,v,d,n]]. \\
[[n,d,v,np], [whox,v,d,n]]. \\
[[ndash,d,v,np], [whox,v,d,n]]. \\
[[np,v,np], [whox,v,d,n]]. \\
[[vp,np], [whox,v,d,n]]. \\
[[s], [whox,v,d,n]]. \\
[[whox,s], [v,d,n]]. \\
[[v,whox,s], [d,n]]. \\
[[d,v,whox,s], [n]]. \\
[[n,d,v,whox,s], []]. \\
[[ndash,d,v,whox,s], []]. \\
[[np,v,whox,s], []]. \\
[[vp,whox,s], []].
\]

However, we know that the example sentences are complete and correct sentences. Therefore, we append another state

\[[s][]\]

to the set of states recorded above, forcing a parse. Addition of the state

\[[s][]\]

after the state

\[[vp,whox,s],[[]]\]

³ the words ‘who’ and ‘whom’ are put into a separate syntactic category called ‘whox’.
in the above case amounts to adding the context-sensitive rule:

\[(b \ b \ vp \ whox \ s \ * \ b \ b \ b \ b) \rightarrow (b \ b \ b \ b \ s \ * \ b \ b \ b \ b)\]

On the surface this kludge seems equivalent to adding another context-free rule

\[S \rightarrow VP \ Whox \ S.\]

But it is not true. The addition of the context-free rule means that this condition is universally valid, i.e., when ever there is a ‘VP’ followed by a ‘Whox’ followed by an ‘S’, we can replace the entire string by an ‘S’. But the addition of the context-sensitive rule says that only in a certain context we can replace a ‘VP’ followed by a ‘Whox’ followed by an ‘S’ with an ‘S’. This certain context is true when, given an input string, the stack goes through the same set of transitions as mentioned in the above transitions.

For the list of state transitions mentioned above, the corresponding context-sensitive rules generated are as follows:

\[
[[d],[n,v,d,n,whox,v,d,n]] \rightarrow [[n,d],[v,d,n,whox,v,d,n]].
[[n,d],[v,d,n,whox,v,d,n]] \rightarrow [[ndash,d],[v,d,n,whox,v,d,n]].
[[ndash,d],[v,d,n,whox,v,d,n]] \rightarrow [[np],[v,d,n,whox,v,d,n]].
[[np],[v,d,n,whox,v,d,n]] \rightarrow [[v,np],[d,n,whox,v,d,n]].
[[v,np],[d,n,whox,v,d,n]] \rightarrow [[d,v,np],[n,whox,v,d,n]].
[[d,v,np],[n,whox,v,d,n]] \rightarrow [[n,d,v,np],[whox,v,d,n]].
[[n,d,v,np],[whox,v,d,n]] \rightarrow [[ndash,d,v,np],[whox,v,d,n]].
[[ndash,d,v,np],[whox,v,d,n]] \rightarrow [[np,v,np],[whox,v,d,n]].
[[np,v,np],[whox,v,d,n]] \rightarrow [[vp,np],[whox,v,d,n]].
[[vp,np],[whox,v,d,n]] \rightarrow [[s],[whox,v,d,n]].
[[s],[whox,v,d,n]] \rightarrow [[[whox,s],[v,d,n]].
[[whox,s],[v,d,n]] \rightarrow [[v,whox,s],[d,n]].
[[v,whox,s],[d,n]] \rightarrow [[d,v,whox,s],[n]].
[[d,v,whox,s],[n]] \rightarrow [[n,d,v,whox,s],[[]]].
[[n,d,v,whox,s],[[]]] \rightarrow [[ndash,d,v,whox,s],[[]]].
[[ndash,d,v,whox,s],[[]]] \rightarrow [[np,v,whox,s],[[]]].
[[np,v,whox,s],[[]]] \rightarrow [[vp,whox,s],[[]]].
[[vp,whox,s],[[]]] \rightarrow [[s],[[]]].
\]

In all, 3944 context-sensitive rules were extracted from the 256 example sentences.
3.5 Corpus

In the hybrid system, the corpus should contain a large number of documents that are sources of example sentences for training the neural network. The example sentences should cover a wide variety of sentences belonging to various syntactic classes. Also, to learn the rules for a particular syntactic class the corpus should contain a lot of example sentences belonging to that class. Once we have such a corpus, the neural network training can proceed with little human supervision.

In the current implementation of the hybrid system, the corpus contained 256 example sentences that belonged to only one class of sentences, the sentences of the type “The dog chased the cat who bit the boy” or “The boy liked the girl whom the dog chased”. These 256 sentences were generated using a limited vocabulary and small set of rules.

4 Current implementation of Hybrid parsing system

This section discusses the implementation details and the current state of the hybrid parsing system. Details regarding the overall system architecture, the neural network architecture and training are presented. Some of the user interface aspects are also touched upon.

4.1 The overall architecture

The current implementation of the hybrid parsing system consists of two main modules, the Trainer module and Feed-forward module. Initially the entire system comes with a limited shift-reduce parser and a completely untrained neural network based parser. This initial system is equivalent to the shift-reduce parser alone.

Once we have the training corpus ready, we use the trainer module to train the neural network based parser. This parser learns to parse the class of sentences contained in the corpus. Once the neural network is trained, we have the power of the shift-reduce parser and the neural network based parser at our disposal. We can then use the feed-forward module to query the entire system.
4.1.1 Trainer module

The functioning of the trainer module is as follows:

- It looks at the set of sentences in the training corpus. It runs each sentence through the shift-reduce parser, which is already there. It records the state transitions of the stack for each sentence.

- It generates the context-sensitives rules from the state transitions as already discussed.

- It then preprocesses the rules into a format that can be used to train the neural network. This step is necessary because the rules are in symbolic form and neural networks work with only numerical data. Therefore the symbolic rules will have to be mapped into numerical data. This is done by assigning a unique number to each of the part of speech. The numbers assigned in this case were as follows:

\[
\begin{align*}
\text{code(adj,11).} \\
\text{code(adjp,21).} \\
\text{code(comp,31).} \\
\text{code(conj,41).} \\
\text{code(cpoula,51).} \\
\text{code(d,61).} \\
\text{code(deg,71).} \\
\text{code(n,81).} \\
\text{code(name,91).} \\
\text{code(ndash,101).} \\
\text{code(np,111).} \\
\text{code(p,121).} \\
\text{code(part,131).} \\
\text{code(pp,141).} \\
\text{code(pronoun,151).} \\
\text{code(s,161).} \\
\text{code(sbar,171).} \\
\text{code(to,181).} \\
\text{code(v,191).} \\
\text{code(vp,201).} \\
\text{code(whox,211).} \\
\text{code(blank,999).}
\end{align*}
\]
A few examples of prepared rules are as follows:

\[
[[61,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999],
[81,191,61,81,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999]].
[[81,191,61,81,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999]].
\]

The above rules are of the form \([20 \text{ values for the input nodes}][20 \text{ values for the output nodes}]\)

- The entire set of the prepared rules are written to a file. This file is the input to the neural network for training. We have used NeuroShell2\(^4\), a neural network shell to implement the neural network. The input file is exported into NeuroShell2 and a suitable architecture is chosen. Then the training begins.

- Once the network is trained satisfactorily, we generate the C code from NeuroShell2. This C code is integrated into the feed-forward module and compiled. This way we get the trained network into our system.

### 4.1.2 Feed-forward module

The feed-forward module comprises of a shift-reduce parser and a well trained neural network based parser. Most part of the feed-forward module, including the shift-reduce parser, is implemented in Prolog. But the trained network is got from NeuroShell2 in the form of C code. This code has to be compiled using a standard C compiler. The executable obtained is called from within the Prolog based feed-forward module. The information passing between the two happens via the scratchpad, which is basically a file. The functioning of the feed-forward module can be explained as follows:

- It receives an input string to be parsed from the user.

- It tries to parse the input sentence using the shift-reduce parser.

\(^4\)A neural network shell developed by Ward Systems Group, Inc.
• If the parse succeeds, it displays the states of the stack and acknowledges that it is a valid sentence.

• If the parse does not succeed in the first stage, it preprocesses the sentence to obtain its numerical representation. This serves as the input for the neural network based parser. The input is written onto the scratchpad and the control is passed to the neural network based parser.

• The neural network based parser reads the input from the scratchpad and fires the network to obtain the output, again in numerical form. The output is written onto the scratchpad. The control is passed to the main program.

• The main program reads the output of the neural network and converts it back to symbolic form. Most of the time, the output from the neural network do not contain integer values corresponding to parts of speech. A typical output looks like this:


This has to be approximated to get the corresponding symbolic values. Upon approximation and reverse mapping to symbolic data, the above output corresponds to

\[ [\text{ndash, d, v, np, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b}] \]

Amongst these, the first ten elements correspond to the contents of the stack and the rest of the elements correspond to the values in the input string. This output is checked to see if it meets the stopping criteria. One of the stopping criteria is

\[ [[s]] \]

which corresponds to

\[ [s, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b] \]
• If the stopping criterion is encountered, the parse is successful and the system halts. It also displays the state transitions of the stack to arrive at the result.

• If the stopping criteria is not met, the output obtained from the neural network is passed back as the input, by writing it onto the scratchpad. The results are again obtained for the new input and the stopping criteria is checked.

• This process is continued till the sentence is successfully parsed or a predefined number of iterations are reached. It was observed that if a sentence cannot be parsed in ten iterations, it could never be parsed. So the limit was set to ten.

4.2 The Neural network based parser details

As mentioned above, NeuroShell2 was used to implement the neural network based parser. Different configurations of neural networks were tried out. The most suitable architecture for this problem was the Ward nets.

4.2.1 Architecture

The Ward nets architecture:

• had an input layer with 20 neurons and an output layer with 20 neurons.

• had three hidden layers with different activation functions. The first hidden layer had 150 neurons with Gaussian activation function. The second hidden layer had 23 neurons with a tanh activation function. The third hidden layer had 140 neurons with Gaussian complement activation function.

• used a standard back propagation algorithm for learning.

• had a learning rate of 0.1, momentum of 0.2 and initial weights as 0.6 for all the links.
4.2.2 Data set partitioning

The entire data set consisting of 3944 learning patterns were partitioned such that every 6th pattern belonged to the test set and every 10th pattern belonged to the production set and the rest of the patterns were used as the training set. In all, there were 3024 patterns in the training set, 595 patterns in the testing set and 325 patterns in the production set.

4.2.3 Training

A standard back propagation algorithm was used for training (Smith, 1993). The training pattern selection was at random and the calibration interval was set to 50. The weight updates was a function of both the learning rate and momentum. The learning rate and momentum were dynamically changed at the rate of -0.002 and -0.01 respectively (Jacobs, 1988). Missing values, if any, were considered as error conditions allowing the algorithm to take care of it. The stopping criteria was that the number of events since minimum average error during the training should become greater than 200000.

4.3 User interface details

Both the Trainer module and the Feed-forward module has to be accessed from within the Sicstus Prolog environment.

4.3.1 Trainer module

- All the example sentences to be included in the corpus for training purposes are listed a prespecified file.

- Within the Sicstus prolog environment, consult the Prolog source file corresponding to the trainer module.

- Upon querying the train/0 predicate, the following interaction is as shown:

```
|?- train.
Enter file containing example sentences: 'in.dat'.
yes
| ?-  
```
By this time the trainer module has processed all the example sentences and has generated the pattern file. The pattern file contains all the training samples in a form that can be imported into NeuroShell2.

Import the pattern file into NeuroShell2, construct the network based on the architecture mentioned above, and start the training process.

When the training has ended, the neural network is ready to be consulted. Using the code generation option in NeuroShell2, generate the C code of the trained network.

Compile this C code to get the executable and place it in the right path so that the feed-forward module can access it.

4.3.2 Feed-forward module

Within the Sicstus prolog environment, consult the Prolog source file corresponding to the Feed-forward module.

Upon querying the parse/1 predicate, which takes the input sentence in a list format, the following interaction is as shown:

```
| ?- parse([the,dog,chased,the,cat]).
[[d],[n,v,d,n]].
[[n,d],[v,d,n]].
[[ndash,d],[v,d,n]].
[[np],[v,d,n]].
[[v,np],[d,n]].
[[d,v,np],[n]].
[[n,d,v,np],[]].
[[ndash,d,v,np],[[]].
[[np,v,np],[]].
[[vp,np],[[]].
[[s],[[]].
could parse with the regular shift-reduce parser
yes
| ?-
```

In the above case, the sentence was parsed in the first stage itself, using the shift-reduce parser alone.
• If we try to parse the sentence “The boy liked the dog who chased the cat”, the following interaction is as shown:

```
| ?- parse([the, boy, liked, the, dog, who, chased, the, cat]).
[[d], [n, v, d, n, whox, v, d, n]].
[[n, d], [v, d, n, whox, v, d, n]].
[[ndash, d], [v, d, n, whox, v, d, n]].
   . . .
   . . .
[[np, v, n, d], [whox, v, d, n]].
[[vp, n, d], [whox, v, d, n]].
[[vp, n, d], [whox, v, d, n]].
```

Could not parse using traditional parser.
Try with NN-based parser? (y/n) - y.
```
[[d], [n, v, d, n, whox, v, d, n]].
[81.73472, 60.26528, 999.73472, 998.26528, 999.73472, 998.26528, 999.73472, 998.26528, 999.73472, 998.26528, 999.73472, 998.26528, 999.73472, 998.26528].
   . . .
   . . .
[161.87342, 999.16347, 999.64736, 996.34873, 996.88374, 1000.72364, 999.77233, 998.00291, 996.87346, 999.11145, 1002.26355, 999.98566, 998.67673, 998.67673, 999.33376, 995.23245, 998.98574, 999.99876, 1001.87345, 999.35534].
```

Success!! The sentence was successfully parsed
Here are the various states of the stack:
```
[[d], [n, v, d, n, whox, v, d, n]].
[[n, d], [v, d, n, whox, v, d, n]].
[[ndash, d], [v, d, n, whox, v, d, n]].
[[np], [v, d, n, whox, v, d, n]].
[[v, np], [d, n, whox, v, d, n]].
In the above case, the sentence could not be parsed using the shift-reduce parser alone. In the second stage, the neural network based parser was able to parse the sentence.

- Currently, this system assumes that all the words that it encounters is included in the lexicon used by the system. If it is given a sentence that includes a word not contained in its lexicon, the following interaction is as shown:

```
| ?- parse([the, peacock, is, a, beautiful, bird]).
Some of the words entered are not in the vocabulary.
Please try again with a different set of words.
```

```
yes
| ?-
```

The user interface is still crude. But the biggest hindrance is the messy interface between the main system and the NeuroShell2. A lot of manual intervention is required as the integration is not seamless. Also, the main system is implemented using Sicstus Prolog in Unix and NeuroShell2 is a Windows based application. This introduces an additional overhead of transferring files between the two environments.
5 Conclusion and future work

This paper presents a two stage approach to parsing natural language sentences. The first stage uses the traditional shift-reduce parsing technique to parse a given input sentence. If this fails, a neural network based parser takes over from where the shift-reduce parser had failed and tries to complete the parsing. This work also demonstrates that neural networks, which were alien to the natural language processing domain, can be used in a powerful way to parse sentences.

The future work involves improving the system to parse sentences occurring in real world documents. Right now, the capability of the system is limited to parsing a small class of sentences in a toy world. In order to achieve this, more context-sensitive rules need to be incorporated into the system to handle a variety of sentences, the lexicon should be extended, and the corpus should contain real-world documents. On the user interface front, the main system and the neural network needs to be integrated seamlessly.

References


Program Listing

5.1 Trainer Module

% File train.pl
% Part of the trainer module, this file contains the train/0 predicate.
% train/0 predicate compiles the example sentences in the corpus and
% generates neural network training data.
:-use_module(library(system)).
:-ensure_loaded('convert.pl').
:-ensure_loaded('reme.pl').
:-ensure_loaded('nextconvert.pl').

train:-
write('Enter file containing example sentences: '),
read(File),
see(File),
tokenize_file(List),
seen,
pre_process(List,PreprocessedList),
tell('temp1.txt'),
pars_entire(PreprocessedList,[],NewL),
told,
convert_main('temp1.txt','temp2.txt'),
reme('temp2.txt','temp3.txt'),
next_convert_main('temp3.txt','final'),
exec(process,[null,null,null],X).

pars_entire([],X,Y):-!.
pars_entire([.’|Rest],L1,L2):-
reverse(Li,L1),
pars(Ri,Pre),
write_extra(Pre),nl,
pars_entire(Rest,[],NewL),!.
pars_entire([H|T],X,Y):-
reverse(L,H),
pars_entire(T,Y,NewL),!.
write_extra(L):-
\+ L = [s],
write([s],[s]),write('.'),nl,
write([s]),write('.'),nl.
pars(PreprocessedList,Result):-
shift_reduce(PreprocessedList,[],Result).
shift_reduce([],Stack,Result):-
shift([],Stack,Result).
reduce(St,ReducedStack,St1),
shift_reduce(St1,ReducedStack,Result).
shift_reduce([],Result,Result).
shift(X,[(H)|Y],[(H)|Y]).
reduce(St,ReducedStack,St1):-
print_certain(St),
reduce(St,ReducedStack,St1).
print_certain(Stack,St),
reduce(Stack,Stack2),
reduce(Stack2,ReducedStack,St1),
reduce(Stack,Stack1).
print_certain(List,St):-
reverse_concatenate(L, [], L1).
reverse_concatenate([], L, L).
reverse_concatenate([X|L1], L2, L3) :-
reverse_concatenate(L1, [X|L2], L3).
all_pos(List),
write( '['),
write(List),
write( ', '),
get_speech(List),write(NS),
write( ']' ),nl.
print_certain(X,Y).

all_pos([Word|Rest]):-
  pos(Word),all_pos(Rest).
all_pos([]).

got_speech([Word|X], [Cat|Rest]):-
  word(Cat,Word),got_speech(X,Rest).

got_speech([], []).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% s --> np,vp
brule([vp,np|X],[s|X]).
pos(s).
pos(np).
pos(vp).

% np --> pronoun
brule([pronoun|X],[np|X]).
pos(pronoun).

% np --> name
brule([name|X],[np|X]).
pos(name).

% np --> d,ndash,pp,sbar
brule([sbar,pp,ndash,d|X],[np|X]). % pp is prepositional phrase
-pos(sbar).
pos(pp).
pos(ndash).
pos(d).

% np --> d,ndash,sbar
brule([sbar,ndash,d|X],[np|X]).

% np --> d,ndash,pp
brule([pp,ndash,d|X],[np|X]).

% np --> d,ndash
brule([ndash,d|X],[np|X]).

% np --> (del_conj),np,conj,np
brule([np,conj,np|X],[np|X]).
pos(conj).

% sbar --> comp,s
brule([s,comp|X],[sbar|X]). %comp = that, and /' / -- to add into lexicon
-pos(sbar).

% ndash --> adjp,ndash
brule([ndash,adjp|X],[ndash|X]).
pos(adjp).

% ndash --> n
brule([n|X],[ndash|X]).
pos(n).

% adjp --> (del_conj),adjp,conj,adjp

% adj --> deg,adj
brule([adj,deg|X],[adj|X]).
pos(deg).
pos(adj).

% adjp --> adj
brule([adj|X],[adj|X]).

Typ(vp,V,NP,PP,Sbar)}
brule([sbar, pp, np, np, v|X], [vp|X]).
  pos(v).

\% vp(vp(V, NP, sbar))
brule([sbar, np, np, v|X], [vp|X]).

\% vp(vp(V, NP, PP))
brule([pp, np, np, v|X], [vp|X]).

\% vp(vp(V, PP, sbar))
brule([sbar, pp, np, v|X], [vp|X]).

\% vp(vp(V, NP))
brule([np, np, v|X], [vp|X]).

\% vp(vp(V, PP))
brule([pp, np, v|X], [vp|X]).

\% vp(vp(V, sbar))
brule([sbar, np, v|X], [vp|X]).

\% vp(vp(V, PP))
brule([pp, np, v|X], [vp|X]).

\% vp(vp(V, NP))
brule([np, np, v|X], [pp|X]).

\% vp(vp(V, PP))
brule([pp, np, v|X], [pp|X]).

\% vp(vp(V, to, VP))
brule([vp, to, v|X], [vp|X]).
pos(to).

\% vp(vp(V, NP, Part))
brule([part, np, v|X], [vp|X]).
pos(part).

\% vp(vp(V, Part, NP))
brule([np, part, v|X], [vp|X]).

\% vp(vp(VP, Conj, VP1)) \rightarrow \text{del_conj}, vp(VP), conj(Conj), vp(VP1).

\% vp(vp(V, NP))
brule([np, v|X], [vp|X]).

\% vp(vp(V, PP))
brule([pp, v|X], [vp|X]).

\% vp(vp(V, AdjP))
brule([adjp, v|X], [vp|X]).

\% vp(vp(V, sbar))
brule([sbar, v|X], [vp|X]).

\% vp(vp(Copula, AdjP))
brule([adjp, copula|X], [vp|X]).
pos(copula).

\% vp(vp(Copula, NP))
brule([np, copula|X], [vp|X]).

\% vp(vp(V))
brule([v|X], [vp|X]).

\% pp(pp(P, conj, PP1)) \rightarrow \text{del_conj}, pp(P), conj(Conj), pp(PP1).

\% pp(pp(P, PP))
brule([pp, p|X], [pp|X]).
pos(p).

\% v(v(V, Conj, V1)) \rightarrow \text{del_conj}, v(V), conj(Conj), v(V1).

word(whox, whom).
word(conj, and).

word(whox, whox).
word(pronoun,us).
word(pronoun,me).
word(pronoun,you).
word(pronoun,he).
word(pronoun,him).
word(pronoun,she).
word(pronoun,they).
word(pronoun,nobody).
word(pronoun,everyone).
word(pronoun,it).

word(name,max).
word(name,john).
word(name,joe).
word(name,bill).
word(name,jack).
word(name,mary).
word(name,fido).
word(name,felix).

word(adj,silly).
word(adj,gray).
word(adj,young).
word(adj,big).
word(adj,black).
word(adj,hairy).
word(adj,green).
word(adj,fat).
word(adj,unexpected).
word(adj,favorite).
word(adj,amazing).
word(adj,noisy).

%default rule
brule([Word|X],[Cat|X]):=word(Cat,Word).

X------------------ pre_process ----------
pre_process([],[]).
pre_process([H|T],List):-
( H = ',';
 N = '(';
 N = ')'),
pre_process(T,List),!.
pre_process([H|T],List):-
pre_process(T,List),!.

X------------------ get_type ------------
get_type([],[]).
get_type([H|Rest],List):-
get_fos(H,Fos),
assertz(word(H,Fos)),
get_type(Rest,List),!.

X------------------ get_fos -----------
get_fos(Atom,Fos):=
word(Fos,Atom),!.
get_fos(Atom,Fos):-
write('Could not find FOS for '''),write(Atom),write('''. Please enter one:'),nl,
read(Fos),!.

X------------------ char_type -----------
char_type([N,T],[]):
N = 0?7,
N =<122, !,
T =alphabet.
char_type([N,T],[]):
N =<65,
N =<90, !,
T =alphabet.
char_type([N,T],[]):
N =<32, !,
T=blank.
char_type(\, special).

X---------------- append ------------
append([], X, X).
append([A|B], Y, [A|Z]): append(B, Y, Z).

X---------------- read_line ------------
read_line(L): get0(C), read_line_aux(C, L).
read_line_aux(-1, []).!.
read_line_aux(10, []).!.
read_line_aux(13, []).!.
read_line_aux(C, [C|R]): read_line(R).

X----------------- lower_case ------------
lowercase_list([], []).!
lowercase_list([U| LRest], [L| RRest]):-
U >= 65, U <= 90,
L is U+32,
lowercase_list(LRest, Rest).
lowercase_list([L| LRest], [L| RRest]):-
lowercase_list(LRest, Rest).

X---------------- extract_token ------------
extract_token([], [], []).!
extract_token([Blank| X], Y, Z):-
char_type(Blank, blank), !,
extract_token(X, Y, Z).
extract_token([A,A2| Rest], [A| TRest], LRest):-
char_type(A, alphabet), char_type(A2, alphabet), !,
extract_token([A2| Rest], TRest, LRest).
extract_token([A,C| Rest], [A|C| Rest]):-
char_type(A, alphabet),
char_type(C, alphabet), !.
extract_token(S| Rest), [S| Rest].

X---------------- tokenize_list ------------
tokenize_list(List, Tokens):-
extract_token(List, TL, Rest),
tokenize_list_aux(TL, Rest, Tokens).
tokenize_list_aux([], [], []).!.
tokenize_list_aux([T| Rest], [T| Tokens], Tokens).
tokenize_list_aux([T| Tokens], Tokens).

X---------------- tokenize_line ------------
tokenize_line(Tokens):-
read_line(U),
lowercase_list(U, L),
tokenize_list(L, Tokens).

X---------- Tokenize the file ----------
tokenize_file(Tokens):-
tokenize_line(T),
tokenize_file_aux(T, Tokens).
tokenize_file_aux(T, T): at_end_of_stream, !.
tokenize_file_aux(T, Tokens):-
append(T, TT, Tokens),
tokenize_file(TT).

X=======================================

% File convert.pl
% convert_main/2 reads the lists of sentences an does the
% first stage of processing and writes it into a temporary file

code(adj, 11).
code(adjp, 21).
code(comp, 31).
code(conj, 41).
convert_main(Infile, Outfile):-
  see(Infile),
  tell(Outfile),
  repeat,
  rread(Stream),
  Stream = end_of_file, !,
  seen,
  told.

rread(Stream): -
  read(Stream),
  rread_aux(Stream).

rread_aux(end_of_file):- !.
rread_aux(Stream):-
  Stream = [X|[Y]],
  map_values(X, X1),
  mapp(X1, SX1),
  map_values(Y, Y1),
  mapp(Y1, SY1),
  write('['),
  print_values([SX1, SY1]),
  nl.

map_values([], []). map_values([H|T], [NH|NT]):-
  code(H, NH),
  map_values(T, NT).

mapp(X, Y): -
  length(X, L),
  x(L, LS),
  append(X, LS, Y).

print_values([]): -
  write('e').
print_values([H|T]): -
  print_list(H),
  write(' ,'),
  print_values(T).
print_list([H|T]): -
  write(H),
  print_list(T).
print_list([]): -
  write('e').
write(N).

27
write(‘,’),
print_list(T).
% append([],X,X).
% append([H|T],Y,[H|Z]):-
% append(T,Y,Z).
% File reme.pl
% Second stage of preprocessing
reme(Infile,Outfile):-
see(Infile),
tell(Outfile),
repeat,
erread(Stream),
Stream == end_of_file,
!,
seen,
told.
erread(Stream):-
read(Stream),
erread_aux(Stream).
erread_aux(end_of_file):-
!
erread_aux(Stream):-
\ Stream = end_of_file, Stream = X,
removee(X,XE),
write(XE),write(‘.’),
nl.
removee([H|T],H[T]):-
removee(T,T).
erread_aux([H|T],H[T]):-
removee(T,T).
rema(Infile,Outfile):-
see(Infile),
tell(Outfile),
repeat,
erread(Stream),
Stream == end_of_file,
!,
seen,
told.
erread(Stream):-
read(Stream),
erread_aux(Stream).
erread_aux(end_of_file):-
!
erread_aux(Stream):-
\ Stream = end_of_file, Stream = X,
removee(X,XE),
write(XE),write(‘.’),
nl.
removee([H|T],H[T]):-
removee(T,T).
erread_aux([H|T],H[T]):-
removee(T,T).
% File nextconvert.pl
% The final stage of preprocessing.
:-dynamic(meof/1).
next_convert_main(Infile,Outfile):-
see(Infile),
tell(Outfile),
read(New),
write(New),write(‘,’),
repeat,
xrread(Stream),
Stream == end_of_file,
!,
told,
seen.
xrread(Stream):-
\ meof(true),
read(Stream),
xrread_aux(Stream),!.
xrread(end_of_file):-
meof(true).
xrread_aux(end_of_file):-
!
xrread_aux(Stream):-
\ Stream = end_of_file, Stream = X,
write(Stream),write(‘,’),!.
pass(Stream),
5.2 Feed-forward Module

File nnparser.pl

The feed-forward module. Call the predicate parse/1 with input sentence to parse. The sentence should be in list format.

\[\text{:-use_module(library(system)).}\]
\[\text{-ensure_loaded('convert.pl').}\]
\[\text{foreign(c_fire_nlp, p_fire_nlp(+integer)).}\]
\[\text{foreign_resource(nnparser, [c_fire_nlp]).}\]
\[\text{- load_foreign_resource(nnparser).}\]
\[\text{- dynamic(delta/1).}\]
\[\text{all_words_in_vocab([]).}\]
\[\text{all_words_in_vocab([H|T]):-}\]
\[\text{word(_,H),}\]
\[\text{all_words_in_vocab(T).}\]
\[\text{parse(List):-}\]
\[\text{pre_process(List,PreprocessedList),}\]
\[\text{all_words_in_vocab(PreprocessedList),}\]
\[\text{asli_parse(PreprocessedList,Result),!}.
\]
\[\text{parse(List):-}\]
\[\text{\+ all_words_in_vocab(PreprocessedList),}\]
\[\text{write('Some of the words entered are not in the vocabulary.'),nl,}\]
\[\text{write('Please try again with a different set of words.'),nl,!}.
\]
\[\text{asli_parse(PreprocessedList,Result):-}\]
\[\text{shift_reduce(PreprocessedList,[]),Result},\]
\[\text{Result = [s],}\]
\[\text{write('Could parse with the regular shift-reduce parser.'),nl,!}.
\]
\[\text{asli_parse(PreprocessedList,Result):-}\]
\[\text{\+ nl,}\]
\[\text{write('Could not parse using traditional parser.'),nl,}\]
\[\text{write('Try with NN-based parser? (y/n) - '),}\]
\[\text{read(X),}\]
\[\text{process_query(X,PreprocessedList),!}.
\]
\[\text{process_query(n,_):-}\]
\[\text{halt.}\]
\[\text{process_query(y,PreprocessedList):-}\]
\[\text{can_parse(PreprocessedList),}\]
\[\text{shift([],PreprocessedList,ReducedStack,S1),}\]
\[\text{reduce(ReducedStack,ReducedStack,S1),}\]
\[\text{map_values(ReducedStack,Temp),}\]
\[\text{mapp(Temp,Temp1),}\]
\[\text{get_speech(S1,S2),}\]
\[\text{map_values(S2,Temp),}\]
\[\text{mapp(Step,Temp1),}\]
\[\text{\}}]
append(Temp1, Stemp1, Topass),
nparse(PreprocessedList, Topass, []).

print_min([[]],
print_min([(H|T)],-
write(H), nl,
print_min([T],
nparse(List, XStream, XList):-
assertz(delta(5)),
%% the terminating condition is when the NN returns something
%% approximately equal to:
%% 
%% [161,999,999,999,999,999,999,999,999,999,999,999,999,999,
%% 999,999,999,999,999,999,999,999,999,999,999,999,999,999,
%% 999,999,999,999,999,999,999,999,999,999,999,999,999,999,
%% 999,999,999,999,999,999,999,999,999,999,999,999,999,999,
%% approximately_equal(XStream, TComp), nl,
write('Success!! The sentence was successfully parsed'), nl,
write('Here are the various states of the stack: '), nl,
interpret_results(List),
!.

nnparse(PreprocessedList, Topass, List):-
  exec('rm nnin', [null, null, null], X1),
  wait(X1, X11),
  exec('rm nnout', [null, null, null], X2),
  wait(X2, X22),
  tell('minin'),
  print_min(Topass),
  told,
  exec(nlp3, [null, null, null], X),
  wait(X, Y),
  sleep(4),
  see('nnout'),
  read(XStream),
  seen,
  write(XStream), nl,
  append(XStream, List, NewList),
  sleep(1),
  nnparse(PreprocessedList, XStream, NewList),
!.

process_query(y, PreprocessedList):-
  try_different(PreprocessedList, 0), !.
  try_different(PreprocessedList, Count):-
    tell('minin'),
    print_min(Topass),
    told,
    exec(nlp4, [null, null, null], X),
    wait(X, Y),
    sleep(4),
    see('nnout'),
    read(XStream),
    seen,
    write(XStream), nl,
    append(XStream, List, NewList),
    sleep(1),
    Newcount is Count + 1,
    try_different(PreprocessedList, Newcount), !.

try_different(PreprocessedList, 10):-
  write('Sorry!! Could not parse even in second stage.'), nl,
  write('Could be because the sentence is syntactically incorrect or '), nl,
  write('the NN has not been trained on such example sentences.'), !.
  try_different(PreprocessedList, Count):-
    tell('minin'),
    print_min(Topass),
    told,
    exec(nlp4, [null, null, null], X),
    wait(X, Y),
    sleep(4),
    see('nnout'),
    read(XStream),
    seen,
    write(XStream), nl,
    append(XStream, List, NewList),
    sleep(1),
    Newcount is Count + 1,
    try_different(PreprocessedList, Newcount), !.

approximately_equal([], []).
approximately_equal([H|T], [H1|T1]):-
  delta(Delta), !,
  abs(H-H1) <= Delta,
  approximately_equal(T, T1).

print_min([[]],
print_min([(H|T)],-
write(H), nl,
print_min([T],

write(H),nl,
print_min(T).

shift_reduce(S,Stack,Result):-
shift(Stack,\[NewStack\],S1),
reduce(NewStack,ReducedStack,S1),
shift_reduce(S1,ReducedStack,Result).

shift_reduce([],Result,Result).

shift(X,\[Y\],\[X\],Y).

reduce(Stack,ReducedStack,S1):-
print_certain(Stack,S1),
reduce(Stack,ReducedStack,S1),
reduce(Stack,Stack).

reduce(Stack,ReducedStack,S1):-
print_certain(List,S1),
write('['),write(List),write(','),get_speech(S1,NS),write(NS),write('].'),nl,
print_certain(X,Y).

all_pos([Word|Rest]):-
pos(Word),all_pos(Rest).

all_pos([]).

get_speech([Word|X],\[Cat|Rest\]):-
word(Cat,Word),get_speech(X,Rest).

get_speech([],\[]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% s --> np,vp
brule([vp,np|X],\[s|X\]).
pos(s).
pos(np).
pos(vp).

% np --> pronoun
brule([pronoun|X],\[np|X\]).
pos(pronoun).

% np --> name
brule([name|X],\[np|X\]).
pos(name).

% np --> d,ndash,pp,ndash
brule([sbar,pp,ndash,d|X],\[np|X\]). % pp is prepositional phrase
pos(sbar).
pos(pp).
pos(ndash).
pos(d).

% np --> d,ndash,ndash
brule([sbar,ndash,d|X],\[np|X\]).

% np --> d,ndash
brule([pp,ndash,d|X],\[np|X\]).

% np --> d,ndash
brule([ndash,d|X],\[np|X\]).

% np --> (del_conj),np,conj,np
brule([np,conj,np|X],\[np|X\]).
pos(conj).

% sbar --> comp,a
brule([s,comp|X],\[sbar|X\]).%comp = that, and '' -- to add into lexicon
pos(comp).

% ndash --> adjp,ndash
brule([ndaash, adjp|X], [ndaash|X]).
pos(adjp).

% ndaash --> n
brule([n|X], [ndaash|X]).
pos(n).

% adjp --> (del_conj), adjp, conj, adjp
% adjp --> deg, adj
brule([adj, deg|X], [adjp|X]).
pos(deg).
pos(adj).

% adjp --> adj
brule([adj|X], [adjp|X]).

% vp(vp(V, NP, PP, Sbar))
brule([sbar, pp, np, np, v|X], [vp|X]).
pos(v).

% vp(vp(V, NP, Sbar))
brule([sbar, np, np, v|X], [vp|X]).
pos(v).

% vp(vp(V, NP, PP))
brule([pp, np, np, v|X], [vp|X]).
pos(v).

% vp(vp(V, PP, Sbar))
brule([sbar, pp, np, v|X], [vp|X]).
pos(v).

% vp(vp(V, PP))
brule([pp, np, v|X], [vp|X]).
pos(v).

% vp(vp(V, PP))
brule([np, pp, v|X], [vp|X]).
pos(v).

% vp(vp(V, to, NP))
brule([to, np, v|X], [vp|X]).
pos(to).

% vp(vp(V, NP, Part))
brule([part, np, v|X], [vp|X]).
pos(part).

% vp(vp(V, Part, NP))
brule([np, part, v|X], [vp|X]).
pos(NP).

% vp(vp(VP, Conj, VP1)) --> (del_conj), vp(VP), conj(Conj), vp(VP1).

% vp(vp(VP))
brule([np, v|X], [vp|X]).
pos(v).

% vp(vp(VP))
brule([pp, v|X], [vp|X]).
pos(v).

% vp(vp(VP))
brule([adj, v|X], [vp|X]).
pos(v).

% vp(vp(Copula, AdjP))
brule([adjp, copula|X], [vp|X]).
pos(copula).

% vp(vp(Copula, NP))
brule([np, copula|X], [vp|X]).
pos(v).
get_fos(Atom,Fos):-
  word(Fos,Atom),!.
get_fos(Atom,Fos):-
  write('Could not find FOS for '),write(Atom),write('.'),nl,
  read(Fos),!.

X--------------------- char_type ---------------------
char_type(N,T):-
  N=97,  % N is 97
  N=<122,  % N is less than or equal to 122
  T=alphabet.
char_type(N,T):-
  N=65,  % N is 65
  N=<90,  % N is less than or equal to 90
  T=alphabet.
char_type(N,T):-
  N=<32,  % N is 32 or less
  T=blank.
char_type(N,special).

X---------------------- append ----------------------
append([],X,X).
append([A|B],Y,[A|Z]):-append(B,Y,Z).

X------------------ can_parse ------------------
can_parse(PreprocessedList):-
  assertz(brule([vp,whox,s|X],[s])),
  assertz(pos(whox)),
  xshift_reduce(PreprocessedList,[],Result),!,
  Result = [s],
  retract(brule([vp,whox,s|X],[s])),
  retract(pos(whox)).

interpret_results(List):-
  assertz(brule([vp,whox,s|X],[s])),
  assertz(pos(whox)),
  shift_reduce(List,[],Result),!,
  write(Result),
  retract(brule([vp,whox,s|X],[s])),
  retract(pos(whox)).

xshift_reduce(S,Stack,Result):-
  shift(Stack,S,NewStack,S1),
  xreduce(NewStack,ReducedStack,S1),
  xshift_reduce(S1,ReducedStack,Result).

xshift_reduce([],Result,Result).

xreduce(Stack,ReducedStack,S1):-
  brule(Stack,Stack2),
  xreduce(Stack2,ReducedStack,S1).
  xreduce(Stack,Stack,X).

X-------------------read_line-------------------
read_line(L):-get0(C),read_line_aux(C,L).
read_line_aux(-1,[]):!.
read_line_aux(10,[]):!.
read_line_aux(13,[]):!.
read_line_aux(C,[L|Rest]):=read_line(Rest).

X------------ lower_case ------------
lowercase_list([],[]).
lowercase_list([U|LRest],[L|Rest]):-
  U > 65,  % U is greater than 65
  U < 90,  % U is less than 90
  L is U+32,  % L is U plus 32
  lowercase_list(LRest,Rest).
  lowercase_list(LRest,Rest).

X---------------- extract_token ----------------
extract_token([],[],[]).
extract_token([Blank|X],Y,Z):-
  char_type(Blank,blank),!.
5.3 Generated C code for trained neural network

/* This is generated C code. A main has been added later */
/* The program has also been modified so that it reads input from */
/* and writes output to a file */
/* Insert this code into your C program to fire the */
/* #include "math.h"

void Fire_NLP(double *inarray, double *outarray)
{
    double netsum;
    double feature2[76];
    /* inarray[1] is C1 */
    /* inarray[2] is C2 */
    /* inarray[3] is C3 */
    /* inarray[4] is C4 */
    /* inarray[5] is C5 */
    /* inarray[6] is C6 */
    /* inarray[7] is C7 */
    /* inarray[8] is C8 */
    /* inarray[9] is C9 */
    /* inarray[10] is C10 */
    /* inarray[11] is C11 */
    /* inarray[12] is C12 */
    /* inarray[13] is C13 */
    /* inarray[14] is C14 */
    /* inarray[15] is C15 */
    /* inarray[16] is C16 */
}
/* inarray[17] is C17 */
/* inarray[18] is C18 */
/* inarray[19] is C19 */
/* inarray[20] is C20 */
/* outarray[1] is C21 */
/* outarray[2] is C22 */
/* outarray[3] is C23 */
/* outarray[4] is C24 */
/* outarray[5] is C25 */
/* outarray[6] is C26 */
/* outarray[7] is C27 */
/* outarray[8] is C28 */
/* outarray[9] is C29 */
/* outarray[10] is C30 */
/* outarray[11] is C31 */
/* outarray[12] is C32 */
/* outarray[13] is C33 */
/* outarray[14] is C34 */
/* outarray[15] is C35 */
/* outarray[16] is C36 */
/* outarray[17] is C37 */
/* outarray[18] is C38 */
/* outarray[19] is C39 */
/* outarray[20] is C40 */

if (inarray[0]< 16) inarray[0] = 16;
if (inarray[0]> 31) inarray[0] = 31;
inarray[0] = 2 * (inarray[0] - 16) / 15 -1;


if (inarray[7]> 100) inarray[7] = 100;

if (inarray[8]> 100) inarray[8] = 100;

if (inarray[9]> 100) inarray[9] = 100;


if (inarray[16]< 16) inarray[16] = 16;
if (inarray[16]> 99) inarray[16] = 99;
inarray[16] = 2 * (inarray[16] - 16) / 83 -1;

if (inarray[17]< 18) inarray[17] = 18;

if (inarray[18]< 99) inarray[18] = 99;
if (inarray[18]> 100) inarray[18] = 100;
inarray[18] = 2 * (inarray[18] - 99) -1;

netsum = -.3635206;
netsum += inarray[0] * -.3718522;
netsum += inarray[1] * .6671733;
netsum += inarray[2] * -.2726614;
netsum += inarray[4] * -.1348316;
netsum += inarray[5] * 8.235767E-02;
netsum += inarray[6] * .1187233;
netsum += inarray[7] * -.2059524;
netsum += inarray[8] * -.1173071;
netsum += inarray[9] * .1512263;
netsum += inarray[10] * .97;
netsum += inarray[12] * 3.011207E-02;
netsum += inarray[13] * 1.056951;
netsum += inarray[14] * 4.352155E-03;
netsum += inarray[15] * 1.035163;
netsum += inarray[16] * .508994;
netsum += inarray[17] * .4370259;
netsum += inarray[18] * -8.039332E-02;
netsum += inarray[19] * 7.567447E-02;
feature2[0] = 1 / (1 + exp(-netsum));

netsum = -3.362207E-02;
netsum += inarray[0] * 1.756222;
netsum += inarray[1] * .5920472;
netsum += inarray[2] * .1088961;
netsum += inarray[3] * .7034658;
netsum += inarray[4] * .1397584;
netsum += inarray[5] * .1834528;
netsum += inarray[6] * -.1856707;
netsum += inarray[7] * .3656147;
netsum += inarray[8] * -.312692;
netsum += inarray[9] * .2198448;
netsum += inarray[10] * -.1147097;
netsum += inarray[12] * .4035068;
netsum += inarray[13] * -.7062662;
netsum += inarray[14] * .7523631;
netsum += inarray[15] * .322562;
netsum += inarray[16] * .8820039;
netsum += inarray[17] * .972198E-02;
netsum += inarray[18] * -.6.947219E-02;
feature2[1] = 1 / (1 + exp(-netsum));

netsum = -.102641;
netsum += inarray[0] * 1.278583;
netsum += inarray[1] * .5920472;
netsum += inarray[2] * -.7538356;
netsum += inarray[3] * -.487994;
netsum += inarray[4] * 2.841461E-02;
netsum += inarray[5] * -.29629;
netsum += inarray[6] * -.2633515;
netsum += inarray[7] * -.1035347;
netsum += inarray[8] * 9.649734E-02;
netsum += inarray[9] * .1152312;
netsum += inarray[10] * .1503776;
netsum += inarray[12] * .7000844;
netsum += inarray[13] * .4343256;
netsum += inarray[14] * -.5859761;
netsum += inarray[15] * -.3452762;
netsum += inarray[16] * 1.615199;
netsum += inarray[17] * .2536851;
netsum += inarray[18] * -.2447614;
feature2[2] = 1 / (1 + exp(-netsum));

netsum = .3031935;
netsum += inarray[0] * -2.197979;
netsum += inarray[1] * -6.625615;
netsum += inarray[2] * -.1236925;
netsum += inarray[3] * -.792618;
netsum += inarray[4] * -.5423043;
netsum += inarray[5] * -.4.244427E-03;
netsum += inarray[6] * -.3244162;
netsum += inarray[7] * -.1640379;
netsum += inarray[8] * -.1540951;
netsum += inarray[9] * 4.142369E-02;
netsum += inarray[10] * 8.111875E-02;
netsum += inarray[12] * -.2257019;
netsum += inarray[13] * -.2231796;
netsum += inarray[14] * -.240179;
netsum += inarray[15] * -.386572;
netsum += inarray[16] * .1796523;
netsum += inarray[17] * -.7822305;
netsum += inarray[18] * -.3045487;
feature2[3] = 1 / (1 + exp(-netsum));

netsum = .3574204;
netsum += inarray[0] * .21.66646;
netsum += inarray[1] * -.5252303;
netsum += inarray[2] * -.2215231;
netsum += inarray[3] * -.194262;
netsum += inarray[4] * -.4198074;
netsum += inarray[5] * -.1696789;
netsum += inarray[6] * -.2198144;
netsum += inarray[7] * -.1696972;
netsum += inarray[8] * -.9.712861E-02;
netsum += inarray[9] * -.8.326997E-02;
netsum += inarray[10] * -.1.245362;
netsum += inarray[12] * -.2231452;
netsum += inarray[13] * -.3175031;
netsum += inarray[14] * -.4075677;
netsum += inarray[15] * -.4364874;
netsum += inarray[16] * 1.287458;
netsum += inarray[17] * .6687773;
netsum += inarray[18] * -.0890898;
netsum += inarray[19] * -.1440792;
feature2[4] = 1 / (1 + exp(-netsum));

netsum = .2084744;
netsum += inarray[0] * .3211277;
netsum += inarray[1] * -.4.399111E-02;
netsum += inarray[2] * .4.578276;
netsum += inarray[3] * -.7909811;
netsum += inarray[4] * -.4671863;
netsum += inarray[5] * -.5.391194E-02;
netsum += inarray[6] * -.2.479546;
netsum += inarray[7] * -.2.019226;
netsum += inarray[8] * 1.365746E-02;
netsum += inarray[9] * 2.444246E-02;
netsum += inarray[10] * .6898317;
netsum += inarray[12] * -.3409749;
netsum += inarray[13] * .5428051;
netsum += inarray[14] * .1270308;
netsum += inarray[15] * -.5282187;
netsum += inarray[16] * -.5351573;
netsum += inarray[17] * -.0729105;
netsum += inarray[18] * -6.257322E-02;
netsum += inarray[19] * -.456046;

/* The source file has been modified from here */
/* to keep the documentation short */

. . . . .
. . . . .
. . . . .
. . . . .
. . . . .
. . . . .
. . . . .
. . . . .
feature20[118] = 1 / (1 + exp(-netsum));

netsum = -.3317975;
netsum = feature2[0] * -.200726;
netsum = feature2[1] * .2102045;
netsum = feature2[3] * -.191886;
netsum = feature2[4] * -6.799778E-02;
netsum = feature2[5] * -.121053;
netsum = feature2[7] * -4.469411E-03;
netsum = feature2[8] * 8.539068E-02;
netsum = feature2[9] * -6.76638E-02;
netsum = feature2[10] * -1.400709;
netsum = feature2[12] * 2.926992E-02;
netsum = feature2[14] * -1.676571;
netsum = feature2[15] * .1729654;
netsum = feature2[16] * 5.629521E-02;
netsum = feature2[17] * -9.188399E-02;
netsum = feature2[18] * -2.142269;
netsum = feature2[19] * 5.512742E-02;
netsum = feature2[20] * 2.334327E-02;
netsum = feature2[21] * -1.610799E-02;
netsum = feature2[22] * -8.695405E-02;
netsum = feature2[23] * -.092386;
netsum = feature2[24] * -4.142834E-03;
netsum = feature2[25] * -2.762007;
netsum = feature2[26] * .1653398;
netsum = feature2[27] * .1348177;
netsum = feature2[28] * 9.215873E-02;
netsum = feature2[29] * -.1629929;
netsum = feature2[30] * -.3510177;
netsum = feature2[31] * -.2464184;
netsum = feature2[32] * -5.200132E-02;
netsum = feature2[33] * 2.089018;
netsum = feature2[34] * -1.739534E-03;
netsum = feature2[35] * .1439838;
netsum = feature2[36] * -3.606286E-02;
netsum = feature2[37] * -.1349621;
netsum = feature2[38] * -.1943566;
netsum = feature2[39] * -.1579603;
netsum = feature2[40] * -.2414744;
netsum = feature2[41] * .1919391;
netsum = feature2[42] * 6.514727E-02;
netsum = feature2[43] * -.1473741;
netsum = feature2[44] * -2.054458;
netsum = feature2[45] * 8.503885E-03;
netsum = feature2[46] * -.2370402;
netsum = feature2[47] * 1.330654E-04;
netsum = feature2[48] * 3.154456E-02;
netsum = feature2[49] * .2488546;
netsum = feature2[50] * -.118874;
netsum = feature2[51] * -.9.339673E-02;
netsum += feature2[52] * .2872633;
netsum += feature2[53] * -2.764576E-03;
netsum += feature2[54] * -.0576528;
netsum += feature2[55] * -.1129642;
netsum += feature2[56] * 2.196038E-02;
netsum += feature2[57] * .2697708;
netsum += feature2[58] * -8.845692E-02;
netsum += feature2[59] * .2982234;
netsum += feature2[60] * -1.326233E-02;
netsum += feature2[61] * -5.609586E-02;
netsum += feature2[62] * -.284344;
netsum += feature2[63] * -5.833765E-02;
netsum += feature2[64] * -.0294642;
netsum += feature2[65] * -.0627662;
netsum += feature2[66] * -4.507107E-02;
netsum += feature2[67] * -.276912;
netsum += feature2[68] * -3.358168E-02;
netsum += feature2[69] * -3.785118E-03;
netsum += feature2[70] * 8.631478E-02;
netsum += feature2[71] * 2.492307E-02;
netsum += feature2[72] * 9.902146E-02;
netsum += feature2[73] * -6.535082E-02;
netsum += feature2[74] * -9.492006E-02;
netsum += feature2[75] * .2633451;
outarray[19] = 1 / (1 + exp(-netsum));

outarray[0] = 15 * (outarray[0] - .1) / .8 + 16;
if (outarray[0]< 16) outarray[0] = 16;
if (outarray[0]> 31) outarray[0] = 31;
if (outarray[7]> 100) outarray[7] = 100;
outarray[8] = (outarray[8] - .1) / .8 + 99;
if (outarray[8]> 100) outarray[8] = 100;
if (outarray[9]> 100) outarray[9] = 100;
void c_fire_nlp(int i)
{
    FILE *inf, *outf;
    double dout[20];
    int i;
    double darr[20];
    inf = fopen("nnin","r");
    outf = fopen("nnout","w");

    for(i=0;i<20;i++)
    {
        fscanf(inf,"%f",&darr[i]);
    }
    Fire_NLP(darr, dout);
    fprintf(outf,"
[");
    for(i=0;i<20;i++)
    {
        if(i == 19)
            fprintf(outf,"%f,\n",dout[i]);
        else
            fprintf(outf,"%f,"dout[i]);
    }
    fclose(inf);
    fclose(outf);
}

void main(void)
{
    c_fire_nlp(1);
}