MULTI-LEVEL NEURAL NETWORK
LANGUAGE TRANSLATOR

James A. Bednar

Dec 11, 1993

1Supervising professor: Risto Miikkulainen
Abstract

This report describes the completion of a project to develop a functioning neural-network-based machine translation system. Our system architecture consists of a sequential neural network to produce a rough unordered translation, and a pair of RAAM encoding networks linked by a backpropagation mapping network to remap the rough translation into the correct sequence. Performance was very good on the training data (99% perfect overall), but was poor on the testing data (only 25% perfect overall). The sequential and mapping networks had poor generalization performance, while the RAAM networks performed well overall. Some possible methods for improving the performance of the sequential and mapping networks are discussed, and if these are implemented the system may become more practically useful.
## Contents

1 Theory and results  
1.1 Introduction .............................................. 3  
1.1.1 About this project and report ......................... 3  
1.1.2 Theory ................................................. 3  
1.2 Architecture .............................................. 4  
1.2.1 Overview ............................................... 4  
1.2.2 Sequential Network ..................................... 4  
1.2.3 RAAM Encode .......................................... 5  
1.2.4 Mapping ............................................... 5  
1.2.5 RAAM Decode .......................................... 6  
1.3 Data set .................................................. 6  
1.4 Results .................................................. 6  
1.4.1 Training dataset ....................................... 7  
1.4.2 Testing Dataset ....................................... 7  
1.5 Further avenues for research .............................. 9  
1.6 Conclusion .............................................. 10  

2 Implementation ............................................. 11  
2.1 Overview of the training and testing process ............ 11  
2.2 Data .................................................. 12  
2.2.1 Dataset ............................................... 12  
2.2.2 Dictionaries .......................................... 13  
2.3 Programs ............................................... 14  
2.3.1 Lisp programs ......................................... 14  
2.3.2 PlaNet networks ....................................... 16  
2.3.3 Csh scripts ........................................... 19  

A Dataset .................................................. 21  

B Dictionaries .............................................. 22  
B.1 Chrisman's dictionary .................................... 22  

C Lisp programs ............................................. 25  

D PlaNet networks .......................................... 26
E Csh scripts

F Training and testing output
  F.1 Sequential network
    F.1.1 5 word input window
    F.1.2 3 word input window
  F.2 RAAM Encode network
    F.2.1 Pure input data
    F.2.2 Actual input data
  F.3 Mapping network
    F.3.1 Pure input data
    F.3.2 Actual input data
  F.4 RAAM Decode network
    F.4.1 Pure input data
    F.4.2 Actual input data
Chapter 1

Theory and results

1.1 Introduction

1.1.1 About this project and report

This report describes final status of a project devoted to the design and implementation of a neural-network architecture for a machine translation system. It currently translates simple sentences from Spanish to English, but the same setup could be used to translate between any other character-based languages given sufficient training data.

The novel feature of this system is the use of a series of neural networks to translate a sentence in a series of steps. Currently, typical programs for machine translation require the labor-intensive development of grammatical rule databases (e.g., [1]). Researchers at IBM have recently had some success using a dramatically different approach that relies only on statistics like word correlation. Our system is a similar project using neural networks instead of explicit statistical analysis.

1.1.2 Theory

There are two possible extremes from which to approach the task of translation. The most obvious is a program based on dictionary lookup and substitution for each word. This could be used with equal success for any sentence. However, such a program is almost useless, since to a great degree, the translation of a word depends on its context in the sentence.

On the other hand, a program that maps entire sentences to one another in one step would produce a high quality translation, but it would not be very useful. This is because it is relatively rare that a particular complex sentence is encountered more than once. A practical system has to fall somewhere in the continuum between these extremes; i.e. it must be both general and high-quality.

Brown et al [2] at IBM attempt to achieve such a synthesis by explicitly modeling a set of transformations. Using statistics culled from a large corpus
of sample sentences, they compute the probability for each of the following, given a particular word:

- whether each other word is an acceptable translation
- whether it will translate as zero, one or more words
- how far it is likely to be offset in the final sentence from its original position

All of these statistics are combined for each word in the sentence to determine the most probable output sentence.

Chrisman [3], on the other hand, demonstrates how a particular form of neural network, the dual-ported RAAM devised by Pollack [6], can be used to perform holistic computation. This involves mapping directly between two different sequences of arbitrary length represented in a fixed size. Such computations represent a capability of neural networks that has no direct equivalent using symbolic computations such as Brown's.

1.2 Architecture

1.2.1 Overview

For this project, we sought to apply some of the compromises used in Brown's system to make Chrisman's approach more practical for real language tasks. Like Brown and Chrisman, we restricted our system to translation of only one sentence at a time, disregarding the surrounding sentences.

We used a two-step approach, first generating a literal translation of each word in the context of the sentence, then mapping that rough translation into a fully ordered one. The basic rationale is that numerous methods exist for using neural networks to map from one sequence to another, but it is relatively difficult to map a sequence into an arbitrary order. Therefore, we have isolated this task so that a network can be optimized for it specifically.

1.2.2 Sequential Network

First, the input sentence is encoded as a sequence of words. The word representations are currently hand-coded using microfeatures (i.e., a binary representation for each word,) but they could also be produced by a network designed to produce optimized representations.

This input sequence is presented to a Jordan [5] recurrent network, which has an input window large enough to hold several word representations. This network is trained to output a series of words in the target language for each input word in the source language. The output sequence should be the most likely translation of the each word given its immediate context. Thus, the
result should be a very rough translation of the original sentence, with the correct words but in the wrong order.

A fundamental limitation of this network is that the window is always finite, and therefore cannot contain every word necessary to determine the correct translation for a word. For instance, given a three-word input window, the following rough translations cannot be correctly realized, since they would entail two different responses to the same input window:

- Usted no esta bien. $\leftrightarrow$ You not are fine.
- Lonnie no esta bien. $\leftrightarrow$ Lonnie not is fine.

I.e., given the window no esta bien the system would have to produce both are and is at different times. This means that even if this network performs perfectly, it will never be able to get all sentences correct. This was done deliberately, since it allows us to see if the other networks in the system will be able to correct the errors left by the first module.

1.2.3 RAAM Encode

The output of the first network is a sequence of arbitrary length that I will call the rough translation. It uses the word representations of the target language, but the sentence structure is not that of the target language. To correct the structure, the sentence has to be examined as a whole, since phrases and words could migrate an arbitrary distance during a perfect translation. We provide this capability by generating a fixed-width, holistic representation of the rough translation sequence using a sequential RAAM network.

The sequential RAAM (Recursive Auto-Associating Memory) is a special case of the general RAAM architecture developed by Pollack [6] for encoding binary trees. The RAAM network is a three-layer backpropagation network that is trained to autoassociate an input word and a set of activations of the hidden layer. For each word in a sequence, that word and the state of the hidden layer after the presentation of the previous word are used as input and targets, thereby forcing the network to develop a representation of these two items on the hidden layer. After the final word in the sequence is presented, the hidden layer activations are a fixed-width representation for the entire sequence.

1.2.4 Mapping

Chrisman used a dual-ported RAAM, which is a pair of RAAM networks which share a common hidden layer. An additional learning step is used with this network to enforce both networks to develop similar or identical internal representations for pairs of sequences. Chrisman uses this capability to perform an implicit mapping between sequences in different languages.
By contrast, we perform similar operations explicitly. The fixed-width representation developed by the RAAM Encode network is mapped by a simple three-layer backpropagation network into a fixed-width representation of the final desired translation. The target representations for this network are determined by the RAAM Decode network (discussed below).

### 1.2.5 RAAM Decode

The fixed-width representation of the target sentence is decoded into the final sequence of words by the RAAM Decode network, which is identical in structure to the RAAM Encode network. The RAAM Decode network is trained to form representations of the final translations. During decoding, it reconstructs (what we hope will be) the desired translation from the representation generated by the mapping network. The final output is an arbitrarily long sequence of microfeature-encoded words.

### 1.3 Data set

To train and test the operation of the system, we developed a complete dataset of sentences based on Chrisman's experiments [3]. (For details, see Section 2.2.1 and the complete listing in A.) The dataset consists of a list of simple sentences in both Spanish and English. Each word in the Spanish sentence has been associated by hand with zero or more English words that represent the literal translation. (Brown describes a system for doing this automatically, and we are assuming that a similar setup would be available for generating our training data.)

A number of interesting phenomena are captured in these sentences, including differences in the number of words and their ordering, distinctions made in one language but not the other, and different translations for a word depending upon context. These complications would make translating word-for-word very awkward. For examples of the various difficulties, consult Chrisman [3] or the full listing of the dataset in Section A.

### 1.4 Results

For all of the percentages cited below, a sentence translation was considered correct only if it matched the target translation exactly. A more precise method of accounting for partially correct sentences could be devised, but this one has the advantages of being both meaningful and easy to check. Since, in general, our system allows a sequence to map to any other sequence, an algorithm for determining how closely sentences match could easily be as complicated as our entire system.

Except where noted, the dataset was divided randomly into two equal groups, here called training and testing. The networks were trained exclusively
on the training dataset until performance no longer improved significantly, usually because the training sentences had been learned perfectly. At this point, the networks were tested for their ability to produce the appropriate mapping for the testing sentences.

The performance on the training dataset indicates the ability of the network to learn a particular mapping given examples. The performance on the testing dataset indicates the degree to which the network picked out features of the training data that generalize well to the entire dataset. For this project, the generalization ability is the most important, since a translation system will need to perform adequately on sentences that it has usually not seen previously.

1.4.1 Training dataset

Using a five-word input window, a fully trained system was able to learn perfect overall translations for the entire dataset. This indicates only that the architecture is capable of performing correctly in the best case.

To make the analysis more realistic, a three-word input window was used for the sequential network for all other experiments. As mentioned earlier, this means that we cannot expect perfect performance from it. For our dataset, the theoretical maximum performance for the 3-word window sequential network is only 77%, assuming that the network will get half of the sentences with contradictory windows correct. The actual result obtained was 74% correct, which corresponds to 96% for particular errors made, see the sample output in Section F.1.2.

Interestingly, the errors made by this network were sufficiently distinct from each other that they could be corrected by the mapping procedure. As shown in Table 1.2, the overall system achieved 99% correct performance for the training dataset (discussed in the next section), even though only 74% of the sentences were initially translated correctly. For sample output, see Section F.4.2. This shows that even with a three-word input window, the system as a whole is capable of nearly perfect performance on the training data.

1.4.2 Testing Dataset

Although the system performed adequately on training data, generalization is more interesting to consider (and was not nearly as successful.) A practical system would need to work for sentences that it has never seen, as well as for those it was trained on.

Performance of the modules

Table 1.1 shows the results for each modules examined independently. That is, to the greatest extent possible, the numbers shown for a module are the percentages the overall system would have gotten perfectly correct had all of
<table>
<thead>
<tr>
<th>Network</th>
<th>Training Dataset</th>
<th>Testing Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq.(3) unnormalized</td>
<td>74%</td>
<td>41%</td>
</tr>
<tr>
<td>Seq.(3) normalized</td>
<td>96%</td>
<td>53%</td>
</tr>
<tr>
<td>RAAM Encode</td>
<td>100%</td>
<td>81%</td>
</tr>
<tr>
<td>Mapping</td>
<td>100%</td>
<td>54%</td>
</tr>
<tr>
<td>RAAM Decode</td>
<td>100%</td>
<td>89%</td>
</tr>
</tbody>
</table>

Table 1.1: Module performance: Individual

the other modules functioned perfectly. The training setups used to determine this data varied; for more information see Section F.

Once the contradictory sentences have been eliminated, the sequential network produced perfect sentences for only 53% of the testing sentences. This indicates fairly poor generalization to new sentences. Considering that many of the input windows (which are all that the sequential network sees) are the same between the training and testing datasets, this result is surprising. In fact, it is possible that this network is not generalizing at all, with the 53% correct being sentences composed only of windows that were in the training dataset. To check this, it would be necessary to examine the sentences by window rather than an entire sentence at a time. This has not yet been performed, and would be an interesting thing to check. This topic is examined further in Section 1.5.

The performance of the individual RAAM networks is much better. They generalized well to new sentences, indicating that their ability to form representations is depends on relevant features of the sentences. Both of them decoded at least 80% of the testing sentences correctly. These results are quite comparable to those obtained by Chrisman [3] in his experiments 2 and 3.

The mapping network achieved only a 54% success rate. This would indicate either that given data with a regular structure, the network was not extracting the structure very well, or that the RAAM representations were not structured in the same way for testing sentences as for those it was trained on. Since the RAAM networks themselves had a better success rate than this for decoding sentences which they had not seen, the mapping network itself is probably at fault.

Chrisman's Experiment 4 was very similar to the individual mapping test. Both of them used two separate RAAM networks, mapping between them with a three-layer backpropagation network. However, he determined that his mapping performance for this configuration was 71%. Though he does not indicate the size of the hidden layer for the mapping network he used, I was unable to obtain such results for a wide range of hidden layer sizes. I have not been able to determine the reason for the discrepancy.

Performance of the system
<table>
<thead>
<tr>
<th>Network</th>
<th>Training Dataset</th>
<th>Testing Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq.(3) normalized</td>
<td>74%</td>
<td>41%</td>
</tr>
<tr>
<td>RAAM Encode</td>
<td>73%</td>
<td>34%</td>
</tr>
<tr>
<td>Mapping</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RAAM Encode</td>
<td>99%</td>
<td>25%</td>
</tr>
</tbody>
</table>

Table 1.2: Module performance: Cumulative

The results for the system as a whole were somewhat surprising, and not altogether encouraging. Table 1.2 indicates the percentage of sentences correct at each point in the pipeline, after they had passed through the previous ones. For the training dataset, the system was able to clean up the sequential output remarkably well, performing almost perfectly on the final translation task (bottom row.)

However, performance on the testing dataset was surprisingly low. Apparently, the poor generalization of the sequential network produced output sequences that were not regularly related to those in the training dataset. Thus, the subsequent networks were unable to clean up the poor translations, resulting in an overall generalization rate of only 25%.

By contrast, Chrisman [3] describes a system that has an overall generalization rate of 75% (Experiment 2) for the same dataset. His system consisted of a single dual-ported RAAM network, with a port for the source sentence and one for the target sentence. We were hoping to have better performance than this, since even 75% correct would result in a nearly unreadable document. However, due at least in part to the factors cited above, our performance was even worse.

1.5 Further avenues for research

The two weak points of this system seem to be the sequential network and the mapping networks. To determine the performance of the sequential network more accurately, it could be trained with windows (instead of sentences) sorted into testing and training datasets. This would allow evaluation of generalization to new windows.

Regardless of the results of this, though, some improvements to the sequential network can be described at this point. Currently, the Jordan network does not make use of any information besides the current input window. Modifying the network to make it an Elman network, which considers both the current input and the previous windows, should have better performance. A network of this type would consider an indefinite amount of context up to a word plus the next few following words.

Chrisman reports much better performance with a dual-ported raam approach, which integrates the encoding and mapping networks into a single network (Experiment 2, 75% correct.) Such an improvement would fit into
our system with relatively little modification. A partial implementation of this is described in Section 2.3.2.

1.6 Conclusion

We were able to develop a functioning neural-network-based machine translation system. However, the performance of this system was less than that of systems that have already been documented, especially [3]. Although some modules of the system performed satisfactorily, the system as a whole is not yet practical for real use. Several possible improvements to the system were listed, and these may provide improved performance.
Chapter 2

Implementation

2.1 Overview of the training and testing process

This project required the development of a suite of Lisp programs, PlaNet networks, data structures, and training scripts. Their use is quite complicated, unfortunately. The steps involved in training and evaluating a network for translation are outlined below.

Create a dataset In order to test and train a neural network for sentence translation, one or more sets of translated sentences are needed. The sentences used in this project are described in Section 2.2.1.

Define networks The structure and behavior of the neural networks needs to be defined. The definitions used in this project are described in Section 2.3.2. The network definition puts a constraint on the size of input and output word representations as described below, so the desired word representation will need to be considered when defining the networks.

Create word representations Each word in the sentence set needs to have a representation that can act as an input to a neural network. For this project, a relatively simple fixed microfeature encoding was used as described in Section 2.2.2.

Generate data for PlaNet The Lisp programs described in Section 2.3.1 are used to assemble the sentences in the dataset into training and testing data in a form suitable for use with the PlaNet neural network package. First, the sentence set is divided into testing and training sets. Then, an input file is created for each type of PlaNet network.

Train networks The input files created by each Lisp program are used to train networks in PlaNet using the csh scripts described in Section 2.3.3. There are many options to set for the learning process, such as the learning rate and number of iterations.
Test networks The Lisp programs described in Section 2.3.1 are used to evaluate the performance achieved by the networks for the training and testing data. Based on these results, the network configuration can be modified if necessary.

2.2 Data

2.2.1 Dataset

Source of sentences

I have expanded the set of 41 examples from the Chrisman paper to include all of the possible simple permutations of the vocabulary. That is, I combined all of the possible subjects with all of the possible verbs and all of the possible predicate endings. I also include the negations of all these sentences. This results in a set of 216 sentences, which is the number that Chrisman states he used. There do not appear to be any other straightforward grammatical sentences that could be formed using this vocabulary and the example sentence structures, so I am reasonably certain that these are exactly the sentences that he or she used. I had the sentences proofread by a much more fluent speaker of Spanish than I, so there should be very few mistranslations or grammatical errors, if any.

Data format

All of the Lisp programs developed manipulate sentences expressed in a simple format that embodies the basic assumptions of this project. In short, the thesis is that it is possible to break down the task of machine translation into two subtasks: first, to generate a rough translation by determining zero or more words that are the simple translation of each word in the context of the source sentence. Secondly, this poorly ordered sequence of words can be mapped to a more natural ordering for the target language.

The format that was used is a simple Lisp construction that allows a translation for each word and a final translation for the sentence to be captured. Elements are each expressed as a list of words, as follows:

Source sentence: \((s_1 \ s_2 \ s_3 \ldots \ s_i)\)

Target sentence: \((t_1 \ t_2 \ t_3 \ldots \ t_m)\)

Translation for source word \(i\): \((tw^i_1 \ tw^i_2 \ tw^i_3 \ldots \ tw^i_{n_i})\)

The format for each sentence is then:

\[
(((t_1 \ t_2 \ t_3 \ldots \ t_n))
\begin{align*}
& (s_1 \ (tw^1_1 \ tw^1_2 \ tw^1_3 \ldots \ tw^1_{n_1})) \\
& (s_2 \ (tw^2_1 \ tw^2_2 \ tw^2_3 \ldots \ tw^2_{n_2}))
\end{align*}
\]
(s₃ \(tw₁^3 \ tw₂^3 \ tw₃^3 \ldots \ twₙ^3\))
\ldots
(s₁ \(tw₁^I \ tw₂^I \ tw₃^I \ldots \ twₙ^I\))

For example, the English translation of the Spanish sentence "No estamos contentos." is "We are not happy.", with each word mapping to one or more words, but in the wrong order. The translations expressed in the above format would be:

((we are not happy)
 (no  (not))
 (estamos  (we are))
 (contentos  (happy)))

2.2.2 Dictionaries

Dictionary format

To create a representation of text that can be used as training data for PlaNet, the Lisp programs replace each word with a numerical representation of itself according to a pre-defined dictionary. This replacement is a simple one-to-one mapping from each word \(w\) to its representation \(f_i\) as follows:

\[
((w_1 \ . \ f_1)
 (w_2 \ . \ f_2)
 (w_3 \ . \ f_3)
 \ldots
 (w_n \ . \ f_n))
\]

An example of a (small) dictionary which maps English words to PlaNet representations spread across 22 units (with either zero or full activations) is:

(((nil  .  "aaaaaaaaaaaaaaaaaa")
 (do  .  "a00000000a0aa00000000")
 (California  .  "00000000a00000000000a"))

Note that the source dictionary needs an entry for --, representing the space between a previous sentence and the current one or the space between the current one and the next. Similarly, the target dictionary needs an entry for nil, which indicates the end of valid data (i.e., the end of the sentence). [The only reason for the different names for these similar constructions is that the -- is more readable in the PlaNet data, while nil is a convenient stopping point for the Lisp parsing routines.]
2.3 Programs

2.3.1 Lisp programs

Programs

The following programs generate training and testing data for networks implemented in the PlaNet neural network system, and evaluate the output from those networks. As input, they use a set of training sentences pair of dictionaries as described above.

create-test.lsp A runnable program (the only one of the set) that randomly splits a set of sentences into testing and training groups according to a given percentage. It is relatively unsophisticated, since it is needed only a few times at the beginning of a training project. Input and output filenames are specified in the program file, which needs to be edited before it is run.

trainseq.lsp, trainseqraam.lsp, traindual.lsp These files contain functions for generating PlaNet training pattern sets from the source translations in the data format discussed above. Each of them reads the set of source translations and produces a binary representation of training sentences for a particular type of network, using the representations for each word from a dictionary file.

Trainseq.lsp, trainseqraam.lsp, and traindual.lsp, produce training data for the Jordan sequential network, a single-ported sequential RAAM network, and a dual-ported sequential RAAM network, respectively.

evalseq.lsp, evalseqraam.lsp These files contain functions for evaluating output data from a PlaNet training or testing session. Evalseq.lsp and evalseqraam.lsp read the data from the sequential or RAAM networks, respectively. They parse the activation patterns into sentences based on a minimum dictionary distance algorithm, and compare the result to the original sentences. Statistics such as percent correct and missed are also computed.

macros.lsp Contains macros for all of the common operations, such as evaluating a directory full of data from different epochs in the training of a network.

library.lsp Contains basic procedures common to several of the create*.lisp and eval*.lisp files.

setup.lsp Initializes the data structures used for all other programs in the set. All of the default parameters that typically will need to be modified are in this file. A typical call would be:

(setup "/u/jbednar/data/Chris/ram/txt" ".300")
Program usage

This section lists the functions that will typically be called by a user. They will be available after a call of the form

\[
\text{(load "/u/jbednar/lisp/macros.lsp")}
\]

has been issued.

Macros (highest-level functions)

\textbf{(train-all (directory))} Given a directory, generate all training files, placing them in the subdirectories \texttt{train} and \texttt{test}.

\textbf{(eval-seq-both (directory epoch-list))} Given a directory and a list of epochs (as strings), evaluates the sequential network output at each epoch for both subdirectories \texttt{train} and \texttt{test}. Example call:

\[
\text{(eval-seq-both "/u/jbednar/data/Chris" ["100" "200")}
\]

\textbf{(eval-seq-train (directory epoch-list))} Given a directory and a list of epochs (as strings), evaluates the sequential output at each epoch for only the subdirectory \texttt{train}. Example call:

\[
\text{(eval-seq-train "/u/jbednar/data/Chris" ["100" "200")}
\]

\textbf{(eval-seq-ram2-both (directory epoch-list))} Given a directory and a list of epochs (as strings), evaluates the two ram outputs at each epoch for both subdirectories \texttt{train} and \texttt{test}. Example call:

\[
\text{(eval-seq-ram2-both "/u/jbednar/data/Chris" ["100" "200")}
\]

\textbf{(eval-seq-ram2-both-train (directory epoch-list))} Given a directory and a list of epochs (as strings), evaluates the two ram outputs at each epoch only for subdirectory \texttt{train}. Example call:

\[
\text{(eval-seq-ram2-both-train "/u/jbednar/data/Chris" ["100" "200")}
\]

\textbf{(eval-seq-ram-real-all (directory epoch-list))} Given a directory and a list of epochs (as strings), evaluates the two ram outputs at each epoch for both subdirectories \texttt{train} and \texttt{test}. Example call:

\[
\text{(eval-seq-ram-real-all "/u/jbednar/data/Chris" ["100" "200")}
\]
High-level functions

(setup (optional path-prefix name-suffix)) Define current working directory, epoch number to be attached after training file names, etc. All of the parameters that change when the dataset, dictionary, or other variables change are updated by this call. Accordingly, all of the functions below require that this function be called beforehand, as shown in the file macros.lsp. The macros are implemented as calls to the following routines after appropriate setup calls. Example of setup information:

(setup "/u/jpeters/data/Chris/raam/test" ".300")

(train-seq nil) Generate moving-window test data for every sentence in the set of pairs Output training and test data for the sequential network

(train-seqraam nil) Generate sets of all sentences, only final sentences, and only rough sentences

(train-dualraam nil) Generate sets of all sentences, only final sentences, and only rough sentences

(eval-seq nil) Evaluate the output from the sequential network

(eval-seqraam nil) Original eval-seqraam: evaluates output from single RAAM network for both the final and rough encodings.

(eval-seqraamreal nil) Evaluates real output from the final RAAM network; i.e. output generated from input from the mapping network, rather than pure training data. Otherwise same as eval-seqraam.

(eval-seqraam-rough nil) Evaluates rough output only from the separate RAAM networks. Otherwise same as eval-seqraam.

(eval-seqraam-final nil) Evaluates final output only from the separate RAAM networks. Otherwise same as eval-seqraam.

(eval-seqraam2 nil) Evaluates output from both the separate RAAM networks, final and rough. Otherwise same as eval-seqraam.

Low-level functions The low-level functions are not meant to be called from the lisp command prompt. They are documented in the code files themselves.

2.3.2 PlaNet networks

The final set of PlaNet networks is listed in Section 2.3.2 and described in the following sections.
Sequential networks

n.ChrisJordan and n.ChrisJordan.bigwindow These network definitions were used for all of the sequential networks used in this project. The only difference between them is the input window size, which is three for n.ChrisJordan and five for n.ChrisJordan.bigwindow.

These files are just macros to define the parameters for n.jordan.mod below. The parameters are:

Nplan: Number of units in the input layer and the plan (context) layer. For this project, this is equal to the number of words in the source window × the width of a source word representation.

Nhid: Number of units in the hidden layer, which can be set smaller to optimize generalization and higher to successfully learn a training set.

Nout: Number of units in the output layer, which for this project is the width of a target word representation.

Tseq: Maximum number of sequence steps in the output for a given input string.

n.jordan.mod This is an straightforward implementation of a Jordan sequential network based on the PlaNet example file example/net/n.jordan. The only changes from that example are the addition of a buffer buffer to hold the output sequence until the network is finished with a particular presentation. The original file did not include any provisions for output of a sequence.

The network reads an input file consisting of an arbitrary number of lines with the following format:

s1s2...sNat0m t1t2...tTseq label

Here, s1s2...sNat0m is a source string Nat0m words long, and t1t2...tTseq is the target output, Tseq words long, for that string. Both Nat0m and Tseq are constant because of limitations in PlaNet: it assumes all input files are to be used directly as input to a set of units, and therefore every unit needs a pattern. For this project, a fixed input window size was used, and thus the fixed source string size was not a problem. However, each word can potentially generate a translation of arbitrary length, so the target string length was chosen as the maximum needed for the translation dataset. For a more general implementation, a more robust input file format would be needed.

Each line of the input file represents one translation to be learned, and is handled by PlaNet as if it were independent of all of the other lines. Thus, for the data created for this project, each source window is presented fully independently of the rest (which is of course not actually true of human translation of a sentence.)

For implementation details, see the source file.
Single-ported sequential RAAM networks
n.ChrisRAAM

This network definition was used for all of the single-ported sequential RAAM networks used in this project. This file is just a macro to define the parameters for n.seqraam below. The parameters are:

Natom: The width of both the source and target word representations, which are identical because a single dictionary is assumed for both input and output.

Nhid: Number of units in the hidden layer, which can be set smaller to optimize generalization and higher to successfully learn a training set.

Tseq: Maximum number of sequence steps in the output for a given input string.

termToler: The tolerance for determining when a nil word has been reached, which is a signal that the end or beginning of a sentence has been reached. This number represents the maximum Euclidean distance from the exact nil representation to an acceptable actual value.

The input and output layers are set to be Natom-Nhid units wide, as in the paper by Chrisman (cited in the final report.)

n.seqraam

This is an implementation of a sequential RAAM network as described in Chrisman’s paper (cited in the final report.) The network reads an input file consisting of an arbitrary number of lines with the following format:

\[ r_1r_2...r_{Tseq} \quad t_1t_2...t_{Tseq} \quad \text{label} \]

Here, \( r_1r_2...r_{Tseq} \) is a rough translation source string Tseq words long, and \( t_1t_2...t_{Tseq} \) is the target output, also Tseq words long, for that string. Tseq is constant because of limitations in PlaNet: it assumes all input files are to be used directly as input to a set of units, and therefore every unit needs a pattern. This parameter was chosen to allow for the maximum sentence length in the translation dataset. The remainder of each sentence was padded with nils.

For training, the network puts each word of the source sentence and encodes it, thereby learning an auto association.

For testing, the network reads each word of the source sentence and encodes it, stopping when a nil has been read.
Dual-ported sequential RAAM networks

n.dual-port

This is an experimental dual-ported sequential RAAM network. It does not appear to function correctly.

2.3.3 Csh scripts

The final set of csh training scripts are described in this section. All of the scripts assume that the current directory at the time they are called contains two subdirectories, entitled train and test. The training scripts call the subprograms trainseq.csh, trainraam.csh, and trainmap.csh, which in turn call trainingloop and testingloop. While most of the work is done in these subprograms, the higher level programs described below determine the behavior relevant for evaluating the training and testing.

Training

learnseq Trains the sequential network using the window patterns in p.SeqData created by the lisp program train-seq. It produces output files with names of the format fp.SeqOutput.XXX, which consist of the sequential outputs from each window at epoch XXX. These files are processed by eval-seq.lsp to produce ASCII versions of the form n1.SeqOutput.XXX. That program also produces a floating point output file in a format readable as input to the RAAM network.

learnraam Trains the sequential RAAM network using the sentence representations in p.SeqRAAMData created by the lisp program train-seqraam. It produces output files with names in the format fp.SeqRAAMOutput.XXX, which consist of the results of autoencoding and decoding the training sentences. These files are processed by eval-seqraam.lsp to produce ASCII versions of the form n1.SeqOutput.XXX.

genmaptraindata Generates the training data for the mapping network by presenting all of the rough translations to the RAAM network and saving the hidden unit representations in files of the form fp.mapHidOut.

learnmap Trains the mapping network using the representations generated by the genmaps program.

Testing

Miscellaneous Most of the module-based testing is handled automatically by the training scripts. Module testing can also be done by testseq, testraam and testmap, which are just versions of the corresponding learn- programs
that instead only present the data. In addition, the following program is used to evaluate the performance of more than one module used together.

testmapping Presents the output from the mapping network fp.mapHidOut.XXX to the sequential RAAM to obtain the final translation of the compressed representation. The output is in the file fp.SRFinalOut.real, which is converted to an ASCII representation by the lisp program eval-seqraamreal.
Appendix A

Dataset
Set of sentences for training to translate
from Spanish to English
[Alistair Chapman 1991]

Each entry contains the information for converting a Spanish sentence to a literal English translation and to a final correct translation.

The format for translating <SOURCE SENTENCES> = <SN1> SN2 ... SNm>
to <FINAL TARGET SENTENCE> is:

(<SNi>) (TW1_1 TW2_1 ... TWm_i)

where TW1_1 ... TWm_i is the literal translation of the source word represented as a sequence of zero or more words.

Note: a '/' following the sentence indicates that it was not actually in the Chapman paper.

(setq SASentences (list

(I am happy)
(esta [I am])
(contento [happy]))

(I am not happy)
(no [not])
(esta [I am])
(contento [happy]))

(you are happy)
(usted [you])
(estas [you])
(contento [happy]))

(you are not happy)
(usted [you])
(no [not])
(estas [you])
(contento [happy]))

(he is happy)
(esta [he is])
(contento [happy]))

(he is not happy)
(no [not])
(esta [he is])
(contento [happy]))

(Lonnie is happy)
(Lonnie [Lonnie])
(esta [is])
(contento [happy]))

(Lonnie is not happy)
(Lonnie [Lonnie])
(no [not])
(esta [is])
(contento [happy]))

(they are happy)
(usted [they are])
(contento [happy]))

(they are not happy)
(no [not])
(usted [they are])
(contento [happy]))

(we are happy)
(esto [we are])
(contento [happy]))

(we are not happy)
(no [not])
(esto [we are])
(contento [happy]))

(they are not happy)
(tiene [they are])
(contento [happy]))

(they are happy)
(tienes [they are])
(contento [happy]))

(we are not happy)
(esta [we are])
(contento [happy]))

(we are happy)
(esta [we are])
(contento [happy]))

(I am angry)
(esta [I am])
(factor [angry])

(I am not angry)
(no [not])
(esta [I am])
(factor [angry])

(you are angry)
(usted [you])
(estas [you])
(factor [angry])

(you are not angry)
(usted [you])
(no [not])
(estas [you])
(factor [angry])

(he is angry)
(esta [he is])
(factor [angry])

(he is not angry)
(no [not])
(esta [he is])
(factor [angry])

(Lonnie is angry)
(Lonnie [Lonnie])
(esta [is])
(factor [angry])

(Lonnie is not angry)
(Lonnie [Lonnie])
(no [not])
(esta [is])
(factor [angry])

(they are angry)
(tienen [they are])
(factor [angry])

(tyhey are not angry)
(no [not])
(tienen [they are])
(factor [angry])

(we are angry)
(esto [we are])
(factor [angry])

(we are not angry)
(no [not])
(esto [we are])
(factor [angry])
(estudiante (a student))

{(they are student)
|{nos
|{estudiantes

{(they are not student)
|{no
|{sos
|{estudiantes

{(we are student)
|{sos
|{estudiantes

{(we are not student)
|{no
|{sos
|{estudiantes

****************************************************************************** want it

{(I want)
|{lo
|{quiero

{(I do not want it)
|{no
|{lo
|{quiero

{(you want it)
|{usted
|{lo
|{quiere

{(you do not want it)
|{no
|{lo
|{quiere

{(he wants it)
|{lo
|{quiere

{(he does not want it)
|{no
|{lo
|{quiere

{(Lonnie wants it)
|{Lonnie
|{lo
|{quiere

{(Lonnie does not want it)
|{no
|{lo
|{quiere

{(they want it)
|{lo
|{quieren

{(they do not want it)
|{no
|{lo
|{quieren

{(we want it)
|{lo

****************************************************************************** want money

{(we do not want it)
|{no
|{lo
|{quieren

****************************************************************************** have it

{(I have it)
|{lo
|{tengo
[no (not)]
[es (he is)]
[de (from)]
[Pittsburgh (Pittsburgh)]

{(Loenie is from Pittsburgh)
 (Loenie (Loenie))
 (es (is))
 (de (from))
 (Pittsburgh (Pittsburgh))
}

{(Loenie is not from Pittsburgh)
 (Loenie (Loenie))
 (no (not))
 (es (is))
 (de (from))
 (Pittsburgh (Pittsburgh))
}

{(they are from Pittsburgh)
 (sonas (they are))
 (de (from))
 (Pittsburgh (Pittsburgh))
}

{(they are not from Pittsburgh)
 (no (not))
 (sonas (they are))
 (de (from))
 (Pittsburgh (Pittsburgh))
}

{(we are from Pittsburgh)
 (sonas (we are))
 (de (from))
 (Pittsburgh (Pittsburgh))
}

{(we are not from Pittsburgh)
 (no (not))
 (sonas (we are))
 (de (from))
 (Pittsburgh (Pittsburgh))
}

-------------------------------------------------------------------------------------------------

{(I am from California)
 (es (I am))
 (de (from))
 (California (California))
}

{(I am not from California)
 (no (not))
 (es (I am))
 (de (from))
 (California (California))
}

{(you are from California)
 (usted (you))
 (es (are))
 (de (from))
 (California (California))
}

{(you are not from California)
 (usted (you))
 (no (not))
 (es (are))
 (de (from))
 (California (California))
}

{(he is from California)
 (es (he is))
 (de (from))
 (California (California))
}

{(he is not from California)
 (no (not))
 (es (he is))
 (de (from))
 (California (California))
}

{(Loenie is from California)
 (Loenie (Loenie))
 (es (is))
 (de (from))
 (California (California))
}

{(Loenie is not from California)
 (Loenie (Loenie))
 (no (not))
 (es (is))
 (de (from))
 (California (California))
}

{(they are from California)
 (sonas (they are))
 (de (from))
 (California (California))
}

{(they are not from California)
 (no (not))
 (sonas (they are))
 (de (from))
 (California (California))
}

{(we are from California)
 (sonas (we are))
 (de (from))
 (California (California))
}

{(we are not from California)
 (no (not))
 (sonas (we are))
 (de (from))
 (California (California))
}
Appendix B

Dictionaries

B.1 Chrisman’s dictionary

The dictionary used in this project was taken from Chrisman [3]. The entries are as follows:

;;; Dictionary of microcoded representations of English words
;;; [From Chrissman 1991]
;;;

(setq engdict
  '((nil . "00000000000000000000000000000000")
    (do . "a0000000000a0a0000000000")
    (does . "a00000000000a0a0000000000")
    (want . "a0000000000a0a0000000000")
    (wants . "a0000000000a0a0000000000")
    (have . "a00000000000a0a0000000000")
    (has . "a00000000000a000000000000")
    (am . "a00000000000a000000000000")
    (is . "a00000000000a000000000000")
    (are . "a00000000000a000000000000")
    (I . "0a0000000000a00000000000")
    (you . "0a00000000000a0000000000")
    (he . "a00000000000a000000000000")
    (Lonnie . "0a00000000000a000000000000")
    (they . "0a00000000000a000000000000")
    (we . "0a00000000000a000000000000")
    (it . "00a000000000000000000000000")
    (from . "00a000000000000000000000000")
    (not . "000a000000000000000000000000")
    (a . "00000000000000000000000000000000")
    (happy . "00000000000000000000000000000000")
    (angry . "00000000000000000000000000000000")
    (fine . "00000000000000000000000000000000")
    (here . "00000000000000000000000000000000")

22
;;; Dictionary of microcoded representations of Spanish words
;;; [From Chrissman 1991]

(setq spadict
  '((-- . "000000000000000000a00000")
    (quiero . "a000000000000000000a00000")
    (quiere . "a000000000000000000a00000")
    (quieren . "a000000000000000000a00000")
    (queremos . "a000000000000000000a00000")
    (tengo . "a000000000000000000a00000")
    (tiene . "a000000000000000000a00000")
    (tienen . "a000000000000000000a00000")
    (tenemos . "a000000000000000000a00000")
    (estoy . "a000000000000000000a00000")
    (esta . "a000000000000000000a00000")
    (estan . "a000000000000000000a00000")
    (estamos . "a000000000000000000a00000")
    (soy . "a000000000000000000a00000")
    (es . "a000000000000000000a00000")
    (son . "a000000000000000000a00000")
    (somos . "a000000000000000000a00000")
    (Usted . "a000000000000000000a00000")
    (Lonnie . "a000000000000000000a00000")
    (lo . "0a000000000000000000a00000")
    (de . "000a0000000000000000a00000")
    (no . "0000a00000000000000000a00000")
    (contento . "0000a00000000000000000a00000")
    (contentos . "0000a00000000000000000a00000")
    (furioso . "0000a00000000000000000a00000")
    (furiosos . "0000a00000000000000000a00000")
    (bien . "0000a00000000000000000a00000")
    (aqui . "0000a00000000000000000a00000")
    (joven . "0000a00000000000000000a00000")
    (jovenes . "0000a00000000000000000a00000"))

23
(viejo . "00000a0000a000000000a")
(viejos . "00000a00000aa00000000a")
(profesor . "000000a00aa00a000000")
(profesores . "0000000a0000a0aa000000")
(estudiante . "000000a00aa000a000000")
(estudiantes . "000000a0000a00a000000")
(Pittsburgh . "00000000a0aaaa00aa000")
(California . "00000000a0aaaa000a00a")
(razon . "0000000a0aaaa000000000")
(sueno . "0000000a0aaaa00000000")
(hambre . "0000000a0aaaa000000000")
(sed . "0000000a0aaaa000000000")
(dinero . "0000000a0aaaa000000000a")}
Appendix C

Lisp programs
(provide 'train-seqram)

;; Generates PlaNet training data for the sequential RAM network
;; J.A. Bednar
;; 04/25/93
;;
;; New program for sequential RAM network, not the general one
;; that came with PlaNet 5.7. Generates a file that can be used directly
;; as a pattern file for training, as is done in train-seq.lisp.
;; Output file is a series of lines, each of which is a sequence of
;; words followed by a stop character (NIL).
;;
;; (in-package "USE")
;; (require 'library)

;; Given a sentence, writes the sequential RAM training pattern for it to the
;; given stream.
;;
;; (defun write-sentence (asentence stream)
;;   (format stream "-a -s -t" 
;;       (sentence-representation asentence *max-sent-length* *target-dict*)
;;       (nullize (format nil "-a" asentence)) ))

;; Generates a training file from a list of sentences
;;
;; (defun gen-raam-train-data [sentence-list filename]
;;   (with-open-file (out-train-stream filename :direction :output)
;;     (dolist (x sentence-list)
;;       (write-sentence x out-train-stream ))
;;     (compress filename) ; Compress output file
;;   )

;; Generate sets of all sentences, only final sentences, and only rough sentences
;; Call this procedure externally after calling setup
;;
;; (defun train-seqram nil
;;   (gen-raam-train-data (elements *align-pairs*)
;;       "seqram-train-filename" )
;;   (gen-raam-train-data (mapcar 'get-final *source-translation-set*)
;;       "seqram-final-train-filename" )
;;   (gen-raam-train-data (mapcar 'get-rough *source-translation-set*)
;;       "seqram-rough-train-filename" )
;;   )
(provide 'train-seq)

;Generates FranNet training data for sequential network
;J.A. Bednar
;2/24/93
;03/04/93 Added spaces between sequential sentence blocks in output to improve readability.
;03/15/93 Changed output sequence max length to 4 to accommodate new Cnriam data
;Changed spaces between sequential sentence blocks to comment char '#'
;Changed ASCII label for seq, window pair to remove spaces
;otherwise FranNet only considers the first word to be the label.
;05/31/93 Changed to make into a callable function

(in-package "USER")
(require 'library)

;; Given a list consisting of two sentences, write a line to the outstream w/
;; the encodings for all the words in the first sentence, concatenated together,
;; the encodings for all the words in the second sentence, concatenated together,
;; and an ASCII representation of the sentence pair.
;; Forces the first sentence to be "window-size" and the second to be "max-lit-length".
;;
;; (defun write-sent-pair (outstream context-lit-out)
;; (format outstream ""*a -a -a -a-"
;; {sentence-representation (first context-lit-out) *window-size* *source-dic*}
;; {sentence-representation (second context-lit-out) *max-lit-length* *target-dic*}
;; {uninitialize (format nil ""*a -a context-lit-out) )

;; Given list representing a Spanish sentence and its word-for-word literal
;; translation, generates moving-window representation of each neighborhood
;; Moving window size is three, literal translation can be of any length
;; Will need to set window size as a parameter eventually...
;;
;; (defun handle-lit-pair (outstream lit-pair)
;; (dotimes (x (- (length lit-pair) 3 *header-window-size*))
;; (let* ((lit-window (subseq lit-pair x (* x window-size*)))
;; {source-window (mapcar 'car lit-window)}
;; {target-window (second [elt lit-window *header-window-size*])})
;; {write-sent-pair outstream (list source-window target-window)})
;; (format outstream ""*4-

;; Generate moving-window test data for every sentence in the set of pairs
;; Call this procedure externally after calling setup
;;
;; (defun train-seq-nil
;; (with-open-file (outstream ""sequential_filename"" :direction :output)
;; (deblit (x *literal-pairs*)
;; (let* ((header-nil) ; add dummy space at beginning and end of sentence
;; {header (dotimes (x *header-window-size* header) (push 'dummylit header))})
;; (handle-lit-pair outstream (append (append header x) header) ))
;; {compress ""sequential-filename"" )} }
(provide 'train-dual)

;; Generates PlanNet training data for the dual-ported sequential RAAM network
;; J.A. Bednar
;; 06/12/93
;; Generates a file that can be used directly as a pattern file for training,
;; as is done in train-seq.lsp.
;; Output file is a series of lines containing a rough translation, a final translation,
;; and an ASCII representation of both. The first two representations consist of
;; a sequence of words followed by a stop character (\n).

(in-package "USER")
(require 'library)

;; Given a sentence, writes the sequential RAAM training pattern for it to the
;; given stream
;;
;; (defun write-align-pair (apsir astream)
;;  (format astream "\"a\" \"a\" \"a\"
;;  (sentence-representation [first apair] *max-sent-length* *target-dict*)
;;  (sentence-representation [second apair] *max-sent-length* *target-dict*)
;;  (utilize (format nil "\"\" a\" apair)) ))

;; Generates a training file from a list of sentences
;;
;; (defun gen-dualraam-train-data (pair-list filenames)
;;  (with-open-file (out-train-stream filenames :direction :output)
;;  (do-list (x pair-list)
;;   (write-align-pair x out-train-stream ))
;;  (compress filename) ; Compress output file
;;)

;; Generate sets of all sentences, only final sentences, and only rough sentences
;; Call this procedure externally after calling setup
;;
;; (defun train-dualtrain nil
;;  (gen-dualraam-train-data *align-pairs* *dualraam-train-filename*) |
;; Create a vector-based target dictionary from the original digit representation
(setq *target-vector-dict* (make-vector (longest (source-translation-set)) :initial-element nil))

;; Create lists of the desired translations
(setq *literal-pairs* (make-vector (cdr *source-translation-set*) :initial-element nil))
(setq *final-pairs* (make-vector (source-translation-set) :initial-element nil))
(setq *align-pairs* (make-vector (source-translation-set) :initial-element nil))

;; Construct filenames

;; Define temporary functions for determination of names
(defun fullname (filename) ; For source training data
  (str+ *path-prefix* filename))

(defun longname (filename) ; For result at a particular epoch
  (str+ *path-prefix* filename *name-suffix*))

;; Set the input and output filenames using the user-defined pathname
(defun seq-train-filename (fullname *p seq data*)
  (form-seq train file name))
(defun seq-train-final-filename (fullname *p seq rawn final*)
  (form-seq train final file name))
(defun seq-rawn-final-filename (fullname *p seq rawn final*)
  (form-seq rawn final file name))
(defun seq-rawn-rough-final-filename (fullname *p seq rawn rough final*)
  (form-seq rawn rough final file name))
(defun seq-rawn-fr-input-filename (fullname *p seq rawn rough fr-input*)
  (form-seq rawn rough fr-input file name))
(defun seq-rawn-fr-1-out-filename (fullname *p seq rawn rough fr-1-out*)
  (form-seq rawn rough fr-1-output file name))
(defun seq-rawn-fr-1-3000-out-filename (fullname *p seq rawn rough fr-1-3000*)
  (form-seq rawn rough fr-1-3000 output file name))

;; Update default path and suffix if specified
(if (= (equal path-prefix all) (str+ *path-prefix* "/"))
  (str+ *path-prefix* "/"))
(if (= (equal name-suffix all) (str+ *name-suffix* "/"))
  (str+ *name-suffix* "/"))

;; Set user-defined parameters for the dataset
(defun seq-train-filename (3) ; moving window width, in words (must be odd)
  (str+ *max-lit-length* 4) ; maximum number of literal words to output
  (str+ *max-sent-length* 6) ; maximum number of words/sentence
)

;; Load source files
;; Word dictionaries: I use Chrisman's microfeature word encodings
(with-open-file (astream "/jbednar/lisp/dictionaries/chris.lisp" :direction :output)
  (defconstant *target-dict* (read astream)))
(defconstant *source-dict* (read astream)))

;; Source translations: Training/testing master sentence set of translations
(let ((filename (str+ *path-prefix* *source-tran-lisp*)))
  (uncompress filename)
  (with-open-file (astream filename :direction :input)
    (defconstant *source-translation-set* (read astream)))
  (compress filename))

;; Compute the dataset-dependent parameters

;; Compute widths of representations (number of units required)
(defun int (length (cdr (first *target-dict*)))
  (length (cdr (first *source-dict*))))

;; Compute size of empty window at beginning and end of sentence
;; (one-half the window, minus the center word)
(defun header-window-size (floor (* window-size* 2))
  (length *source-translation-set*)))

;; Compute number of source translation elements in this dataset
(defun number-of-translation (length *source-translation-set*)))
macros.lsp

[provide 'macros]

;; Various macros to perform multiple actions with one command
;;
;; J.A. Bednar
;;
;; 05/31/93

(in-package "USEF")
(require 'library)
(require 'setup)

;; Given a directory, generate all training files, placing them in the subdirectories 'train' and 'test'.
;; Call this procedure externally
(defun train-all (directory)
  (setup (str+ directory "/train/*")
         (train-seq)
         (train-seqram)
         (setup (str+ directory "/test/*")
                (train-seq)
                (train-seqram)))

;; Given a directory and a list of epochs (as strings), evaluates the sequential
;; output at each epoch for both subdirectories 'train' and 'test'.
;; Call this procedure externally
;; E.g. (eval-seq BOTH "//jbednar/data/risto/Chris.half" "100" "200")
;;
;; (defun eval-seq BOTH (directory epoch-list)
;;  (dolist (this-epoch epoch-list)
;;    (setup (str+ directory "/train/*") (str+ "." this-epoch))
;;    (eval-seq)
;;    (setup (str+ directory "/test/*") (str+ "." this-epoch))
;;    (eval-seq)
;;
;; Given a directory and a list of epochs (as strings), evaluates the sequential
;; output at each epoch for only the subdirectory 'train'.
;; Call this procedure externally
;; E.g. (eval-seq-train "//jbednar/data/risto/Chris.half" "100" "200")
;;
;; (defun eval-seq-train (directory epoch-list)
;;  (dolist (this-epoch epoch-list)
;;    (setup (str+ directory "/train/*") (str+ "." this-epoch))
;;    (eval-seq)
;;    (setup (str+ directory "/test/*") (str+ "." this-epoch))
;;    (eval-seq))

;; Given a directory and a list of epochs (as strings), evaluates the two ram
;; outputs at each epoch for both subdirectories 'train' and 'test'.
;; Call this procedure externally
;; E.g. (eval-seqram2 -both-train "//jbednar/data/risto/Chris.half" "100" "200")
;;
;; (defun eval-seqram2 -both-train (directory epoch-list)
;;  (dolist (this-epoch epoch-list)
;;    (setup (str+ directory "/test/*") (str+ "." this-epoch))
;;    (eval-seqram2))

;; Given a directory and a list of epochs (as strings), evaluates the two ram
;; outputs at each epoch only for subdirectory 'train'.
;;
(defun digit-to-vector (digit-string)
  (let ((y (make-array (length digit-string) :element-type 'short-float)))
    (dotimes (x (length digit-string) y)
      (if (char digit-string x) 1.0)
        (setf (aref y x) 0.0)))
)

;; Given a string representation of a dictionary entry as digits, returns the
;; floating point representation as a vector
(defun dictelem-to-vector (dict elem)
  (concatenate 'vector (digit-to-vector (cdr dict elem))))

;; Given a sentence, returns the representation of it as the concatenation of
;; the sequence of dictionary representations of the words, padded out
;; with FID's until the requested sentence length is reached
(defun sentence-representation (sentence sent-length adictionary)
  (let ((y '()))
    (dotimes (x sent-length y)
      (setf y (str+ y (cdr (assoc (nth x sentence) adictionary))))))

;; Returns the closest vector dictionary entry to the given vector.
(defun lookup (look-vector adictionary)
  (let (closest-word (first adictionary))
    (dolist (current-dist adictionary closest-word)
      (let (current-dist (vec-distance (cdr current-word) look-vector))
        (if (< current-dist closest-dist)
          (setf closest-word current-word))))))
(provide 'library)

;;; Utility procedures for the various planet data generating and analyzing
;;; programs for the sequential and RAM networks.

;;; J.A. Bednar
;;; OS/40/90

;;; Previously part of testgen.lisp

(in-package "USER")

;; Constants
;;
(defun *dummy* (---) ; used at beginning and end of sentence
(defun *dummy* (--- nil) ; actual dummy element
(defun *eq* [con nil nil]) ; end of stream test dummy element

;; GENERAL PURPOSE FUNCTIONS

;; Returns the square of a number
(defun * (x) (* x x))

;; Performs string addition (concatenation) on its arguments from left to right
;; (up to 25 arguments... how could it be made to work for any # of args?)
(defun str (a ... b ... c ...)
  (concatenate 'a (list b) (list c)))

;; Given a list, returns a list consisting of the concatenation
;; of the contents of all first-level sublists.
;; E.g., (elements '((1 2) (3 4 5))) returns (1 2 3 4 5)
(defun elements (alist)
  (let ((y nil)
         (x (append y (first y))))
    y)

;; Given a string, returns the string with spaces replaced by `'
(defun unquote (astring)
  (do ((x (length asting) #t)
       (y #t (char asting x))
       (z #t (char asting x) #_)))
    (y)

;; Given a sequence of floats, returns the sum of all of elements.
(defun sum-total (avector)
  (let ((y nil)
         (x (length avector))
         (setf y (+ y (elt avector x)))))

;; Given the number of units to read from a given stream, returns a vector with that
;; many floating point numbers.
(defun F-read-vector (astream num-units)
  (let ((avector (make-array num-units :element-type 'short-float))
         (x nil)
         (y num-units avector)
         (setf (aref avector x) (read astream)))
    x)

;; Returns the vector distance between two fp vectors of the same length.
(defun vec-distance (vector1 vector2)
  (let ((y 0))
    (do ((x (length vector1) (setq y))
         (setf y (+ y (sqrt (- (aref vector1 x) (aref vector2 x))))))
     ()))

;; Compress an output file
;; (Nonstandard function: If this is not supported by a particular lisp or system, comment the operative statement out below.)
(defun compress (filename)
  (inhibit (let* ((x "compress -f " filename)
               (y "uncompress -f " filename))
               x)
               y)

;; Functions for this accessing this dataset

;; Given a SourceTrans element, returns the Spanish source sentence.
(defun get-source (st-element)
  (mapcar 'car (cdr st-element)))

;; Given a SourceTrans element, returns the rough English translation.
(defun get-rough (st-element)
  (let ((y nil))
    (dolist (x (cdr st-element) y)
      (setq y (append y (elements (cdr x) x))))

;; Given a SourceTrans element, returns the final English translation.
(defun get-final (st-element)
  (car st-element))

;; Given a SourceTrans element, returns a list composed of
;; a list representing the Spanish source sentence and
;; a list representing the desired final English translation.
(defun get-final-pair (st-element)
  (list (get-source-st-element) (get-final-st-element)))

;; Given a SourceTrans element, returns a list composed of
;; a list representing the rough English source sentence and
;; a list representing the desired final English translation.
(defun get-alignment-pair (st-element)
  (list (get-rough-st-element) (get-final-st-element)))

;; Given a string representation of a dictionary entry as digits, returns the
;; floating point representation as a list
(defun dig-to-list (digit-string)
  (let ((y nil))
    (do ((x digit-string y)
         (setf x (aref digit-string x))
         (y (read y))))
    x)
;;; input from the mapping network, rather than pure training data. Otherwise same as
;;; eval-seagram.
;;; Call this procedure externally after calling setup
;;; (defun eval-seagram-nil
   (eval-seagram-base (napcar 'get-final 'source-translation-set*)
   *seagram-real-in-filename*
   *seagram-real-out-filename*)

;;; Evaluates rough output only from the separate RAAM networks. Otherwise same as
;;; eval-seagram.
;;; Call this procedure externally after calling setup
;;; (defun eval-seagram-rough-nil
   (eval-seagram-base (napcar 'get-rough 'source-translation-set*)
   *seagram-rough-in-filename*
   *seagram-rough-out-filename*)

;;; Evaluates final output only from the separate RAAM networks. Otherwise same as
;;; eval-seagram.
;;; Call this procedure externally after calling setup
;;; (defun eval-seagram-final-nil
   (eval-seagram-base (napcar 'get-final 'source-translation-set*)
   *seagram-final-in-filename*
   *seagram-final-out-filename*)

;;; Evaluates output from both the separate RAAM networks, Otherwise same as
;;; eval-seagram.
;;; Call this procedure externally after calling setup
;;; (defun eval-seagram2-nil
   (eval-seagram-rough)
   (eval-seagram-final))
(provide 'eval-segraam)

;; Evaluates the output of the sequential RAM network running in Planet
;; Expects a file consisting of the actual output sequence as lists of floating
;; point numbers terminated by a NIL representation.
;;
;; J.L. Bednar
;; 04/25/93
;; 09/31/93 Changed to allow compressed files and allow calling as a function

(in-package "USER")
(requie 'library)

;; Given a list of vectors, returns a cons consisting of the actual
;; target-language translation and a list of the errors for each word.
;;
;; Comment out the if clause to leave all the NIL's in the output
;
(defun actual-to-result (veclist)
  (let ((words nil) 
        (errors nil))
    (closest-entry nil))
    (dolist (x veclist (cons words errors))
      (uf closest-entry (lookup x 'target-vector-dict*)
        (if (eq (car closest-entry) nil) (return (cons words nil))
            (setf words (cons (car closest-entry) x))
            (setf errors (cons (vec-distance (car closest-entry) x) errors))
        )))
  ;; Gets one line of actual Plateau output data from the floating point file.
  ;; Returns a list of [max-sent-length] * 1 (for period) vectors

(defun fp-segraam-read-actual (asream)
  (let ((y nil))
    (doltemes (x (+ 1 max-sent-length) (reverse y))
      (setf y (cons (fp-read-vector asream 'target-vector-dict* y)))))
  ;; Gets one sentence worth of actual Plateau output data from the floating point file.
  ;; Returns a cons consisting of the closest target translation sentence
  ;; and a list of the errors for each word.

(defun fp-segraam-read-sentence (asream)
  (let ((vectors nil)
        (words nil)
        (result nil)
        (errors nil))
    (setf vectors (fp-segraam-read-actual asream))
    (setf result (actual-to-result vectors))
    (setf words (cons (car result) words))
    (setf errors (cons (cdr result) errors))
    (cons (reverse words) (reverse errors)))))

;; Main procedure
(defun eval-segraam-base (sentence-list in-filename out-filename)
  (uncompress in-filename) ; Uncompress if compressed
  (with-open-file (in-stream in-filename :direction input)
    (format t "Reading from ~A in-filename")
    ; Initialize total statistics for this dataset to 0
    ;
    ;; Needs to be set to have output all on one line
    (setq *print-prety* nil)
    ;; Print header
    (list (header (format nil "~*" * Dist Target RAM encoding/decoding -50T Actual-"~*"
      (prin header)
      (prin header out-stream))
    ;;; Process each source sentence, listing the source, target, and actual sequences
    ;;; along with the number of words wrong and the sum of the error values for the w
    ;;; Also echoes each sentence to a file
    ;
    (dolist (this-element sentence-list "done")
      (let* ((result (fp-segraam-read-sentence in-stream)
                (actual (car result))
                (errors (cdr result))
                (match (equal this-element (remove nil (elements actual))))
                (msg (if match "~*" ~*"))
                (total-error (sum total (elements errors))
                (average-error (/ total-error length errors))
                (display-message
                  (format nil "~*" a -3,3f -a -50T a -a msg average-error this-element
                  (elements actual)))
                (format t "~*" a -4,4f display-message)
                (format out-stream "~*" a -4,4f display-message)
                (inf *sentences-read*
                (if match nil (inf *sentences-missed*))))
      (let* ((sentences-correct (- *sentences-read* *sentences-missed*)))
        (percentage-correct (* 100 (* sentences-correct *sentences-read*)))
        (percentage-missed (* 100 (* sentences-missed *sentences-read*)))
        (total-message
          (strt
            (format nil "% Total sentences read: ~*" *sentences-read")
            (format nil "% Total sentences correct: ~a ~a ~a")
            (sentences-correct percentage-correct)
            (format nil "% Total sentences missed: ~a ~a ~a")
            (sentences-missed percentage-missed)))))
    (prin total-message)
    (prin total-message out-stream)
    (format t "Wrote to ~*" out-filename))
  (compress in-filename ; Recompress input file)
)

;; Original eval-segraam: evaluates output from single RAM network for
;; both the final and rough encodings.

;; Call this procedure externally after calling setup
;;
(defun eval-segraam nil
  (eval-segraam-base (elements 'align-pairs*)
    *segraam-nil-out-filename* ))

;; Evaluates real output from the final RAM network; i.e., output generated from
(average-error (/ total-error (length errors)))
(display-message
  (format nil "\(a \approx 5.3F \approx a -\{5X \approx a\} m\) flag average-error literals actual))

;; Output the .fp file for the next network in the pipeline (the RANM network)
(format fp-outstream "-a-\{4\"
  (io-representation (elements vectors) 'max-sent-length' 'target-vector-dict*))

;; Output the natural language evaluations
(format t "-a-\{4\" display-message)
(format nil-outstream "-a-\{4\" display-message)
(llncf *sentences-read*)
(if match nil (llncf *sentences-missed*) ))

(let* ((sentences-correct (- *sentences-read* *sentences-missed*))
   (percentage-correct (* 100 (/ sentences-correct *sentences-read*)))
   (percentage-missed (* 100 (/ *sentences-missed* *sentences-read*)))
   (totals-message
    (str+ (format nil "\(Total\) sentences read: \(a\) *sentences-read*\)
      (format nil "\(Total\) sentences correct: \(a\) \(-32\)\(\{-4,0\}\)
        sentences-correct percentage-correct\)
      (format nil "\(Total\) sentences missed: \(a\) \(-32\)\(\{-4,0\}\) \(-4\" *sentences-missed* percentage-missed) )))
    (princ totals-message)
    (princ totals-message nil-outstream) )
)
(compress *seq-fp-in-filename*)
(compress *seq-fp-out-filename*)

(format t "Wrote to \(a\) \(-4\" *seq-nil-out-filename*)

(provide 'eval-seq)

;; Evaluates the output of the sequential network running in PlainNet.
;; Expects a file consisting of the actual output sequence as a series
;; of floating point numbers.
;;
;; J.A. Bednar
;; 03/22/93
;; 05/31/93 Changed to make into a callable function
;; (in-package "USER")
;; (require 'library)

;; Gets one line of actual PlainNet output data from the floating point file.
;; Returns a list of "max-lt-length" vectors
;; (defun fp-read-actual (astream)
;; (let ((y nil))
;;   (do ((x (cons (reverse y) (reverse y)))))
;;     (if (and x (fp-read-vector astream "target-dict") y))))

;; Given a list of vectors, returns a cons consisting of the actual
;; target-language translation and a list of the errors for each word.
;;
;;Comment out the if clause to leave the NIL's in the output
;; (defun actual-to-result (vec-list)
;; (let ((words nil) (errors nil)
;;       (closest-entry nil))
;;   (do ((x (cons (reverse words) (reverse errors) (reverse vectors)))
;;       (self closest-entry (lookup x "target-vector-dict")
;;       (if (eq (car closest-entry) nil)
;;         (return (list (reverse words) (reverse errors) (reverse vectors))
;;         (push x vectors)
;;         (push (car closest-entry) words)
;;         (push (vec-distance (cdr closest-entry) x) errors))))

;; Gets one sentence worth of actual PlainNet output data from the floating point file.
;; Returns a cons consisting of the closest target translation sentence
;; and a list of the errors for each word.
;; (defun fp-read-sentence (astream st-element)
;; (let ((newvectors nil)
;;       (words nil) (errors nil) (result nil))
;;   (do ((x (cdr st-element) (list (reverse words) (reverse errors) (reverse vectors)))
;;       (self result (actual-to-result newvectors)
;;       (push (first result) words)
;;       (push (second result) errors)
;;       (push (third result) vectors))))

;; Given a vector, returns the string representation of it as a sequence of floating
;; point numbers
;; (defun vec-to-string (avector)
;;   (do ((x (length avector) y)
;;       (if (and x (fp-read-vector astream "target-dict") y))))

;; Given a sentence, returns the representation of it as the concatenation of
;; the sequence of dictionary representations of the words, padded out
;; with NIL's until the requested sentence length is reached
;; (defun to-representation (vec-list sent-length adictionary)
;; (let ((y nil))
;;   (do ((x (length vec-list) y)
;;       (if (and x (fp-read-vector astream "target-dict") y))
;;     (setf y (str y (vec-to-string this-element))
;;     (setf y (str y (vec-to-string (cdr (assoc nil adictionary)))))))

;; Evaluate the output from the sequential network
;;
;; Call this procedure externally after calling setup
;; (defun eval-seq nil)

;; Open the input file of floating point outputs from sequential network
;; (openfp "seq-fp-in-filename"*)
;; (with-open-file (in-stream "seq-fp-in-filename"*
;;   (format t "Reading from ~a.* seq-fp-in-filename")
;;   ;; Open the output natural language version with error messages
;;   (openfp "nl-outstream" "seq-nl-out-filename"*
;;     (format t "Writing to ~a.* seq-nl-out-filename")
;;       ;; Open the output file of floating point inputs to the ANN network
;;       (openfp "fp-outstream" "seq-fp-out-filename"*
;;         (format t "Writing to ~a.* seq-fp-out-filename")
;;           ;; Needs to be set for output to be all on one line
;;           (setq *princ-pretty* nil)
;;           ;; Initialize total statistics for this dataset to 0
;;           (setq *sentences-read* 0)
;;           (setq *sentences-missed* 0)
;;           (setq *words-read* 0)
;;           (setq *words-missed* 0)
;;           ;; Print text header
;;           (header (format nil ""))
;;           ;; Print header for fp output for next network
;;           (header (format nil "")
;;             (princ header)
;;             (princ header n1-al-outstream)
;;           ;; Print header for fp output for next network
;;           (header (format nil "")
;;             (princ header)
;;             (princ header n1-al-outstream)
;;           ;; Process each source sentence, listing the source, target, and actual sequences
;;           ;; along with the number of words wrong and the sum of the error values for the wor
;;           ;; Also echoes each sentence to a file
;;           (colist (this-element "source-translation-set" "done")
;;             (target (get-rough this-element))
;;             (elements (cdr this-element))
;;             (result (fp-read-sentence instream this-element))
;;             (actual (first result))
;;             (errors (second result))
;;             (vectors (third result))
;;             (match (equal target (remove nil elements actual)))))
;;             (mflag (if match "")
;;               (total-error (sum-total (elements errors)))
;;               (total-error (sum-total (elements errors))))

}
## Translation dictionaries

In this file type, the target dictionary is first, followed by the source dictionary.

### Dictionary of microcoded representations of English words

[From Chrisman 1991]

### Width of English representation: 22 units

<table>
<thead>
<tr>
<th>Word</th>
<th>Microcode</th>
</tr>
</thead>
</table>
| nil | *1aaaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaaa1aaa
(provide 'create-test)

;; Splits a source translation set into training and testing sets according to
;; a specified percentage.
;;
;; J.A. Bednar
;; 04/04/93

;; (in-package "USER")
(defvar 'library

;; Set percent of st-elems to put in the training dataset
;; (the rest are put in the testing dataset)
(defvar *train-percentage* 50)

;; Set the input filename
(defvar *st-file-name* "source-trans.lsp"); Chrisman's full Spanish-English sentence set
(defvar *st-file-name* "source-trans-tiny.lsp"); small testing dataset

;; load source translations
(with-open-file (astream *st-file-name* :direction :input)
  (deconstant *source-translation-set* (read astream))
  (deconstant *st-num-elems* (length *source-translation-set*)))

;; Compute the relative sizes
(let ((train-ratio (* *train-percentage* 100))
      (test-ratio (- 1 (* *train-percentage* 100))))
  (deconstant *train-size* (length (* train-ratio *st-num-elems*)))
  (deconstant *test-size* (length (* test-ratio *st-num-elems*))))

;; Split into two sets according to percentage
(setf *train-st-list* nil)
(setf *test-st-list* nil)

(let ((movep (make-array *st-num-elems* :element-type 'bit))
      (do ((i 0) ((= i *st-num-elems*))
           (let ((an-elem (random (* *st-num-elems* 1))))
             (if (= 0 (elt movep an-elem))
                 (progn (setf (1+ i) (setf (elt movep an-elem) 1)))
                 (do ((k 0) ((= k *st-num-elems*))
                      (let ((an-elem (random (* *st-num-elems* 1))))
                        (if (= 0 (elt movep k))
                            (progn (setf (elt 'source-translation-set* x) *train-st-list*)
                                   (setf (elt *source-translation-set* x) *test-st-list*))))
                               (setf *test-st-list* (reverse *test-st-list*))
                               (setf *train-st-list* (reverse *train-st-list*)))))

;; Outputs list of source-trans elements
(defun at-file-out (at-list size filename)
  (with-open-file (astream filename :direction :output)
    (format astream "<

    (format astream "<

    (format astream "<

    (write at-list astream pretty t)

    (format astream "")

;; Write the representations for the training and testing sets
(at-file-out *train-st-list* *train-size* "source-trans.chris-half-train.lsp")
(at-file-out *test-st-list* *test-size* "source-trans.chris-half-test.lsp")
Appendix D

PlaNet networks
Example net for Jordan's sequential network.
Specify Nplan, Nhid, Nout, and Tseq (no. of patterns in a sequence)

Define ErtMsg via network

IFDEF Nplan; printf ErtMsg; exit; ENDIF
IFDEF Nhid; printf ErtMsg; exit; ENDIF
IFDEF Nout; printf ErtMsg; exit; ENDIF
IFDEF Tseq; printf ErtMsg; exit; ENDIF

layer Eln Nplan
layer Hidden Nhid
layer Output Nout
layer State Nout

connect ElnHidden Plan to Hidden
connect StateHidden State to Hidden
connect HiddenOutput Hidden to Output

Input Nplan
Target Tseq Nout

matrix targetSeq Tseq Nout
vector errorSeq Tseq
vector prevErrorSeq Nout

procedure activate
scalar t # local var. for time step in sequence

targetSeq=target # save target sequence
Output=0; State=0 # clear network states
prevTarg = 0 # present input to Plan

t = 0

while t < Tseq # for each time step in sequence
    call activateStep(targetSeq(t), errorSeq(t)) # activate one step
    t += 1
endwhile
$Error = mean(errorSeq) # average error for the sequence
end

procedure learn
scalar t # local var. for time step in sequence

TargetSeq=target # save target sequence
Output=0; State=0 # clear network states
prevTarg = 0 # present input to Plan

t = 0

while t < Tseq # for each time step in sequence
    call activateStep(targetSeq(t), errorSeq(t)) # activate one step
    call learnStep $ # learn one step
    t += 1
endwhile
$Error = mean(errorSeq) # average error for the sequence
end

scalar decay 0.6
scalar feedbackTarget

procedure activateStep expr targetVec expr error
    clear Output; clear Hidden $ clear net inputs and deltas
    if feedbackTarget==1 then
        State = State * decay + prevTarg
    else
        State = State + decay + Output
    endif
    $ alternative implementation of the above
    $ State = State * decay
end

procedure learnStep
    backward HiddenOutput; delta Hidden
    learn PlanHidden; learn Bias Hidden
    learn StateHidden; learn Bias Output
end

end
**n.seagraam:**

**Sequential version of Jordan Pollack's RAAm architecture for learning binary trees.**
Usage: network n.seagraam

Loosely based on 'example/raam/n.raam' in the style of 'example/raam/n.jordan'

J.A. Bednar
4/25/93

5/4/93 Corrected bug in learning procedure. Deltas were not being computed correctly.

7/17/93 Modified to use external file with procedures common to single and dual RAAm's

Make sure required parameters have been defined beforehand

define ErrMsg /
\n\nFNDSET Natom; printf ErrMsg; exit; ENDIF # size of input representation for one word
FNDREF Nhld; printf ErrMsg; exit; ENDIF # size of hidden layer
FNDREF Tseq; printf Tseq; exit; ENDIF # maximum number of sequence steps
FNDREF termToler; set termToler 0.2; ENDIF # maximum distance to stop char to allow

#define loSize Natom*Nhld # size of input and output layers

#define the network architecture.
layer Input loSize
layer Hidden Nhld
layer Output loSize

connect Input Hidden Input to Hidden
connect Hidden Output Hidden to Output

Input Tseq*Natom
target Tseq*Natom

vector errorSeq Tseq+1 # to store errors for patterns
matrix InputSeq Tseq+1 Natom # to store current input sequence
matrix OutputSeq Tseq+1 Natom # to store current output sequence
vector outComp Nhld # to store current output compressed representation

load /s/jbednar/planet/nat/raam_procedures

// Presents an input sequence to the network. The compressed representation
// can be read from 'outComp' (which is just a copy of the hidden layer), and the
// output sequence can be read from 'OutputSeq'.
//
// Generates compressed representation from input seq.,
// presents this as input for decoding, and
// generates the output sequence from the compressed rep.
procedure activate
InputSeq[0] = nilWord # include stop character
InputSeq[1->Tseq] = input # copy rest of sequence to train from input buffer

end

// Top level training procedure
// Given sequence of words followed by stop character, presents each
// item and propagates to get fixed-width representation, Output can
n.doubleport:

## Dual-port, sequential version of Jordan Pollack's RRAM architecture for
## learning binary trees.
## usage: network n.doubleport
## based on 'n.seagram'
## J.A. Bednar
## 6/12/93
## 7/17/93 Modified to use external file with procedures common to single and dual RRAM

## Make sure required parameters have been defined beforehand

```plaintext
# Define constant macros
define inputSize (Input1|Input2|InputHidden|InputOutput)
define outputSize (Input1|Input2|InputHidden|InputOutput)
define decoderSize (Input1|Input2|InputHidden|InputOutput)
define encoderSize (Hidden|Output)
define RNNSize (Input|Input2|InputHidden|InputHiddenOutput|InputOutput)
define RNNSize2 (Input1|Input2|InputHidden|InputHiddenOutput|InputOutput)
define RNNSize3 (Input1|Input2|InputHidden|InputHiddenOutput|InputOutput)
define RNNSize4 (Input1|Input2|InputHidden|InputHiddenOutput|InputOutput)
```

```plaintext
# define the network architecture.
layer Input1 inputSize
layer Input2 inputSize
layer Hidden inputSize
layer Output1 outputSize
layer Output2 outputSize
layer BothInput inputSize

# Note: Although the following connection and the above two are completely
# separate as far as Planet is concerned, the above connections will need to
# be copied into the following before using it, and copied back to the proper
# connections when confluent training is completed.
connect InputHidden Input1 to Hidden
connect InputHidden Input2 to Hidden
connect HiddenOutput Hidden to Output1
connect HiddenOutput Hidden to Output2
connect BothOutput BothInput to Output1
connect BothOutput BothInput to Output2

# Load common low-level procedures
load ~/joeinar/planset/networks/ssm_procedures
```

```plaintext
# Present an input sequence to the network.
# The output sequence can be read from 'OutputSeq'.
# Generates compressed representation from input seq.,
# presents this as input for decoding,
# and generates the output sequence from the compressed rep.
```
errorSeq[0] = $Error

1:+=1
endwhile

outComp = Xid
$Error = mean(errorSeq)
# save a copy of the compressed representation
# average errors for patterns

end

# Decodes a sequence from the values of the hidden units.
# Output is in the array 'OutputSeq'.

procedures decode layer Hid connect HidOut layer Out

errorSeq = 0
OutputSeq = 1  # initialize with all NUL's just in case

scalar nllDist:
scalar l; i=0
while i < Tseg
    # Begin expansion of
    # call propagateOut(HidOut, Out)
    clear Out
    forward HidOut
    activation Out
    target Out
    # End expansion
    OutputSeq[1] = Out[LFT]
    HidOut = Out[RGT]
    errorSeq[1] = $Error
    nllDist = norm(OutputSeq[1] - nllWord)  # distance to stop character
    if nllOut < termToler) endwhile
    # exit if within tolerance
    $Error = mean(errorSeq)
    # average errors for patterns.

end

# forward propagation of input layer to hidden layer
procedure propagateIn connect InHid layer Hid

clear Hid
forward InHid
activation Hid
end

# forward propagation of hidden layer to output layer
procedure propagateOut connect HidOut layer Out

clear Out
forward HidOut
activation Out
target Out
end

# backward propagation & weight change.
procedure backward connect InHid layer Hid connect HidOut layer Out

backward HidOut
delta Hid
learn InHid
learn HidOut
learnBias Hid
learnBias Out
end

end}

$$$$ end of file raam_procedures


\# n.elman.beta
\# Jeff Elman's sequential network.
\# Based on the Planet/example/met/n.jordan file
\#
\# Modified 3/19 by J.A. Bednar
\#
\# Specify Nplan, NhId, Nout, and Tseq (no. of patterns in a sequence)
\# Nstate is the same as NhId

define ErrMsg \nin\read\with\"network\nplan=..\NhId..\nout=..\seq=\sequence length>\n.\$elan\nState * decay + Hidden

IFNEE[Nplan; printf ErrMsg; exit; ENDIF
IFNEE[NhId; printf ErrMsg; exit; ENDIF
IFNEE[Nout; printf ErrMsg; exit; ENDIF
IFNEE[Tseq; printf ErrMsg; exit; ENDIF

layer Plan Nplan
layer Hidden NhId
layer Output Nout

layer State Nout
layer State NhId

connect PlanHidden Plan to Hidden
connect StateHidden State to Hidden
connect HiddenOutput Hidden to Output

input Nplan

matrix target Tseq Tseq Nout
vector errorSeq Tseq
vector prevTarg Nout

procedure activateStep expr target vec expr error
\# clear net inputs and deltas
\# if feedbackTarget==1 then
\# deleted by J.A.B.
\# State = State * decay + prevTarg
\# else
\# State = State * decay + Output
\# endif

\# target = target vec
\# forward PlanHidden; forward StateHidden
\# activation Hidden
\# forward HiddenOutput; activation Output
\# target Output
\# prevTarg = target
\# error = error

end

procedure learnStep
\# backward HiddenOutput; delta Hidden
\# learn PlanHidden
\# learn StateHidden; learnbias Hidden
\# learn HiddenOutput; learnbias Output

end

scalar decay 0.5
scalar feedback=Target
Appendix E

Csh scripts
Appendix F

Training and testing output

F.1  Sequential network

F.1.1  5 word input window
F.1.2  3 word input window

Training
Testing

F.2  RAAM Encode network

F.2.1  Pure input data
Training
Testing

F.2.2  Actual input data
Training
Testing

F.3  Mapping network

F.3.1  Pure input data

Since the output of the mapping network is a fixed-width internal representation, the accuracy of the mapping cannot be determined without decoding. However, it is difficult to distinguish the performance of the mapping network from the performance of the decoding network. (In contrast, this can be done easily for the RAAM networks because they are trained to autoassociate: their performance is defined by their ability to decode correctly what they have been trained to encode.)
This limitation was circumvented to some degree by the following procedure. Two RAAM networks were trained to form representations of the entire dataset: one for the rough translations, and one for the final translations. Each of them eventually encoded and decoded their representations with perfect accuracy for the entire dataset. At this point, half of the pairs of representations were used to train a mapping network, which was able to map all of them perfectly after training. The generalization ability of the mapping network was then determined by the performance on the other half of the dataset.

Training

Testing

F.3.2 Actual input data

The previous section describes how performance of the mapping network on pure data was obtained. A similar procedure could be used to determine the performance of the mapping network on sentences that have been through all of the previous steps in the pipeline. In this case, all the networks in the system would be trained on half of the dataset except for the final RAAM Decode network, which would be trained on the entire dataset. This would allow the percentage of sentences correctly mapped to be determined independently of the generalization ability of the RAAM Decode network. However, due to time constraints and the fact that I just now figured out how that could be done, this step has not been performed. Thus, only the performance of the system for the pure case has been obtained; the results for the mapping network and the RAAM Decode network have been lumped together in Section F.4.2.

F.4 RAAM Decode network

F.4.1 Pure input data
Training
Testing

F.4.2 Actual input data
Training
Testing
Bibliography


