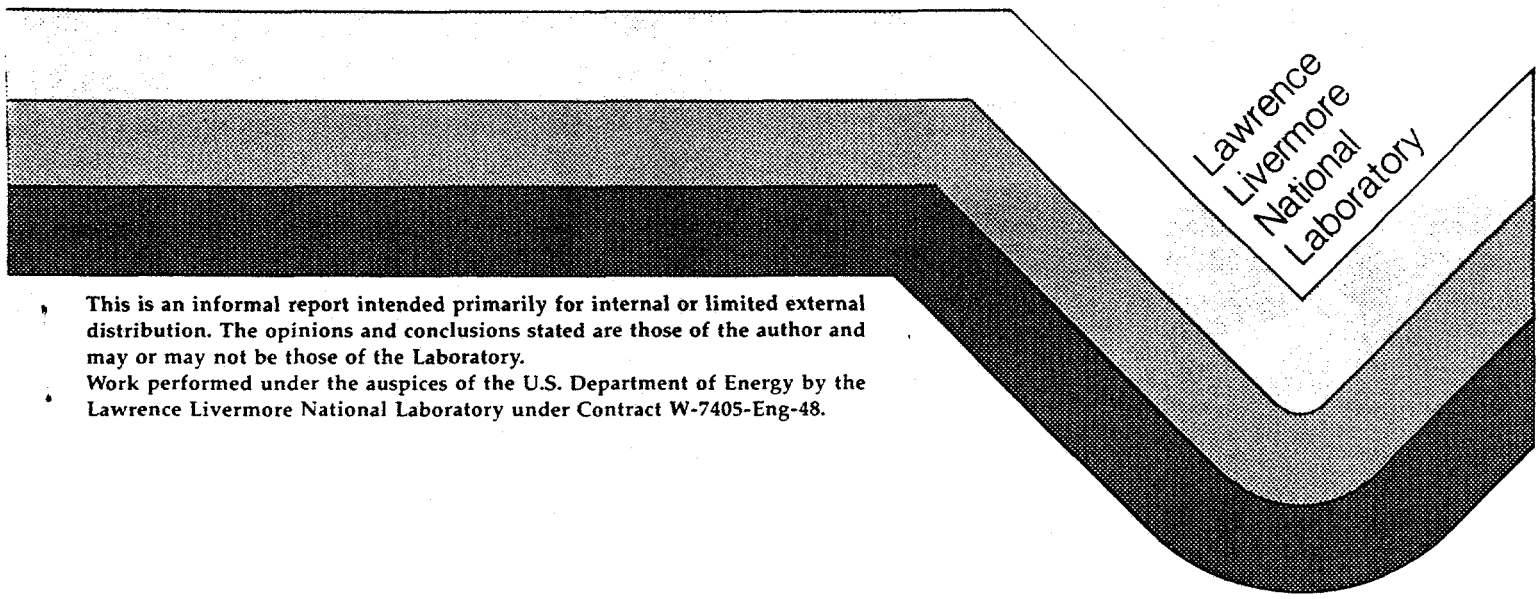
UCRL-ID-121250

"Word Prediction"

D.E. Rumelhart, P. G. Skokowski, B.O. Martin
Lawrence Livermore National Laboratory
Livermore, CA

May, 1995



Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Word Prediction

D.E. Rumelhart, P.G. Skokowski, B.O. Martin

1 May, 1995

1 Abstract

In this project we have developed a language model based on Artificial Neural Networks (ANNs) for use in conjunction with automatic textual search or speech recognition systems. The model can be trained on large corpora of text to produce probability estimates that would improve the ability of systems to identify words in a sentence given partial contextual information. The model uses a gradient-descent learning procedure to develop a metric of similarity among terms in a corpus, based on context. Using lexical categories based on this metric, a network can then be trained to do serial word probability estimation. Such a metric can also be used to improve the performance of topic-based search by allowing retrieval of information that is related to desired topics even if no obvious set of key words unites all the retrieved items.

2 Overview

We have applied Artificial Neural Networks (ANNs) to the problem of continuous text recognition by constructing networks that learn a mapping from a series of words in a text to the probability of occurrence of a large set of possible next words. A machine that can automatically and accurately predict upcoming words based on their written or spoken context can speed up computationally intensive tasks such as automatic speech recognition, handwriting recognition, or translation. Such a system can also be used to improve topic-based search in a large database by allowing retrieval of items that do not contain a key word but do contain a word or words that occur in similar linguistic contexts to a keyword.

We have designed a novel network architecture intended to extract the information needed to determine a conditional probability distribution in the course of learning to map from a group of words to a word likely to occur in that context in a large corpus of text. Since the networks and corpora we have used are large, we have implemented the network architecture on Cray supercomputers to run our simulations. We have developed an ANN architecture that is trained with syntactic information and one that is trained with lexical information. The former is useful for developing word prediction software that would aid a speech recognizer or translation system, while the latter could also be used to aid a topic based search system. Ultimately, the two types of architectures could be combined to produce a system that explicitly incorporates categorical lexical information into a prediction of the likely class of the next word in a string. Such a system could be extremely useful in a variety of tasks including speech recognition, translation, natural language interfaces, topic-based search, and more prosaic domains such as word-processing and document preparation.

3 Stages of the project

The first stage of the project involved producing a network simulation environment on the CRAY supercomputers at NERSC that would allow us to run simulations on extremely large datasets. For this reason we obtained the ASPIRIN network Simulator developed by Russell Leighton at MITRE Corp. This programming environment was installed on the Crays at NERSC to allow us to run our simulations.

The second stage of the project involved obtaining and preprocessing large textual databases. As a concomitant to the project of simulating our networks, we have had to

develop appropriately encoded corpora of text. These data sets were used to train the networks used in our modeling.

Initially we developed a corpus of syntactic training data using the LOB corpus. This corpus contains approximately 1,150,000 words of running text and has been tagged so that each word is accompanied by information about its syntactic category (based on an idiosyncratic syntactic scheme developed by the researchers who accumulated the corpus.) From this source, we produced a file containing 1,151,707 integers in which the nth integer gave the syntactic category (with 131 possible categories) of the nth word of the text.

These data were used to do a pilot study in which a network was trained to map from a string of 5 syntactic categories in the running text of the LOB corpus to a string of 6 categories including a prediction of the category likely to follow the 5 input categories. The description of this network is given below, and the implementation is given in appendix 1.

Our next task was to develop data sufficient to train a network meant to induce a lexical similarity metric for a large corpus. For this task, syntactic information alone was not sufficient. For that reason, we turned to a large (approximately 10 million word) database of articles from the Wall Street Journal, available to us in electronic form.

We have used a one-year portion of the Wall Street Journal corpus (1989) to produce a large body of continuous text in a form appropriate for training the network. This was done by determining the 2000 most frequent words in the corpus and encoding them as integers (using their rank as their code) and coding all other items in a category for low frequency items (all numbers belonged to a single distinct category, and all dollar amounts to still another category.) When the network is trained, these integers are converted on-line to 2001 dimensional input vectors that locally encode the input word. The network is trained using the same codes for input and output items. The internal representation formed by the network provides a compressed (low-dimensional) encoding of the properties of the input and output vectors that can be gleaned from the training procedure. Since the procedure involves a mapping from words to subsequent words, this is the information necessary for determining the probability distribution over subsequent words, given an input environment. The implementation code for this network is given in appendix 2 below.

4 Architecture of Artificial Neural Networks

In the first ANN (see figure 1), we used a simple associative feed-forward network ([?]) and the LOB database encoded as discussed above. The error rule used to train the network was a cross-entropy error function ([?]). The code used to implement this architecture is given in appendix 1.

In order to predict more than just syntactic categorical information in continuous text, and also to develop a lexical similarity metric in the training of a network, we next developed a different architecture, intended to be trained using the large database of running text available from the Wall Street Journal Corpus.

The network architecture we have designed is based upon a traditional associative feed-forward network. words are taken serially from a corpus of text and encoded as binary vectors. The network is then trained to learn a mapping from a word or set of words to the following word in the corpus. This training consists of presenting input and output pairs and then using the error between the prediction of the network and the desired output to update the weights in the network. Such a mapping traditionally employs a network using an input layer, an output layer, and a hidden layer that allows the network to learn

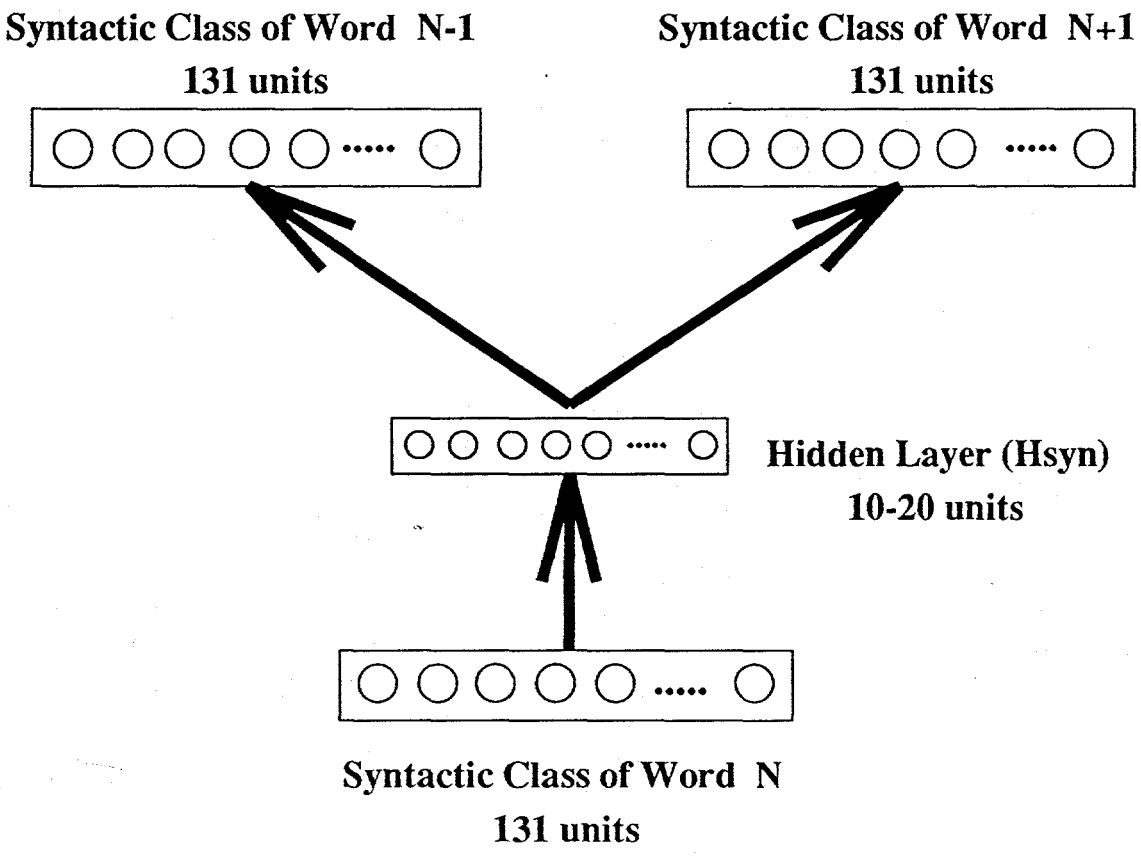


Figure 1: An associative feed-forward network for syntactic class prediction in continuous text.

mappings that involve more than a linear transformation of the input vector to arrive at the desired output.

Our network differs from an classical auto-associator in three major respects.

First, it includes a layer of units between the input units of the network and the sigmoidal hidden layer that associative networks generally employ. This intermediate layer contains units that form categorical distinctions among the input patterns. These categories are constrained by the nature of the prediction task, and by the units' activation function and associated credit-assignment rule. In the course of training these constraints lead to a categorical representation of the input that renders it useful to the prediction task. In this sense, the second layer of the network learns a useful categorical representation of the class of input environments (words or series of words from which predictions are made.)

Second, instead of predicting the next word, we wish to predict the next "category". This has the advantage that categories of words are more predictable than individual words. Thus, we insert another layer of category units prior to the output units. Ultimately it is these category units which we will try to predict.

Finally, whereas many feed-forward associative nets depend upon an updating rule that employ a sum-squared error measure ([?]) we have used an updating rule that minimizes the cross-entropy between output and desired pattern vectors. This insures that even when the number of words encoded grows very large, the network is forced to treat each dimension of the vector as an encoding of important information about the presence or absence of a word. Because we have employed a localist input and output representation and only constructed a distributed representation in the intermediate layers of the network, this is a critical feature of our error function.

The value of the intermediate layers of this network goes beyond their utility in learning the mapping on which they are trained. It is precisely when the network is used to generalize to other corpora or even other mappings that the value of these layer becomes paramount.

5 conclusion

Our results are, thus far, inconclusive. We have developed the required theoretical framework and the tools necessary for developing the required language modelling framework. However, we have not really demonstrated the value of our procedure. The creation of the corpora and its porting of our code to the super computer turned out to require more effort than we anticipated and only preliminary runs have been created. We have every reason to believe that the basic framework will be successful and it is currently being further developed.

6 Appendix 1

6.1 Pattern generator and error function for LOB model

```
/*************************************************************************

syntax/user_init.c: last modified 7/1/93  by Ben Martin

This program generates 5 word inputs and six word outputs for a next
word prediction network.  This version uses priors to condition learning.
```

```
This file is for use with the aspirin/migraines network simulator.
It contains the data generator for the network defined in syntax.aspirin
and it depends on the datafile containing tagnumbers that I made by
running "awk -f /u1/ben/bin/awk/LOB.t2v < $i >> datafile", for each i
in /data/databases/LOB/tags.  Those files were created by using Dave's
script in the LOB directory.

**************************************************************************/

#include "syntax.h"

extern void define_generator(void (*f)(char *item), char *item, char *string);

#define DATAFILE "/data/databases/LOB/tagnums/LOB.tagnum"
#define PRIORSFILE "LOB.priors"
#define NPATTERNS 1151707   /*# of training items in DATAFILE */
#define NTAGS 131    /* # of syntactic categories (tags)*/
#define NINPUTS 5    /* number of tags in input vector */
#define NOUTPUTS 6   /* number of tags in the output */
#define PERIOD 6     /* the tagnum for a period (which is pattern 0) */

/* data: tags */
static int tags[NPATTERNS];

/* input & target buffers */
static float input_vector[NINPUTS * NTAGS], target_vector[NOUTPUTS * NTAGS];

/* priors vector */
static float priors[NTAGS];

/* pattern counter */
static int pattern_counter = NPATTERNS-1;

/*-------------------------------------------------------------*/

/* set input and target to 0's */
static void zero_vectors()
{
  register int counter = NINPUTS * NTAGS;
  while(counter--) {
input_vector[counter] = 0.0;
  }
  counter = NOUTPUTS * NTAGS;
  while(counter--) {
target_vector[counter] = 0.0;
  }
}
```

```
/*-------------------------------------------------------------*/

static void readdata()
{
  FILE *fp, *fopen();
  register int counter;
  int tag;

  printf("\nUsing datafile: %s\n", DATAFILE);

  fp = fopen(DATAFILE, "r");
  if (fp == (FILE *)NULL) {
    fprintf(stderr, "\nUnable to open %s!\n", DATAFILE);
    exit(1);
  }

  /* load the tags */
  for(counter = 0; counter < NPATTERNS; counter++) {
    fscanf(fp,"%d", &tag);
    tags[counter] = tag-1;     /* change from 1-131 to 0-130 indices.*/
  }/* end loop */

  fclose(fp);
  printf("\nDone reading data.\n");
}

/*-------------------------------------------------------------*/

static void set_priors()
{
  FILE *fp, *fopen();
  register int counter;
  float prior;

  printf("\nUsing datafile: %s\n", PRIORSFILE);

  fp = fopen(PRIORSFILE, "r");
  if (fp == (FILE *)NULL) {
    fprintf(stderr, "\nUnable to open %s!\n", PRIORSFILE);
    exit(1);
  }

  /* load the priors */
  for(counter = 0; counter < NTAGS; counter++) {
    fscanf(fp,"%f", &prior);
    priors[counter] = prior;
  }/* end loop */
```

```
  fclose(fp);


  printf("\nDone setting priors.\n");
}

/*-------------------------------------------------------------*/

set_input(/*int pattern_counter*/)
{
  register int position;

  clear_previous_input();
  for(position = 0; position < NINPUTS; position++)
{
 input_vector[tags[(pattern_counter+position)%NPATTERNS]+NTAGS*position]=1.0;
}
}

/*-------------------------------------------------------------*/

set_target(/*int pattern_counter*/)
{
  register int position;

  clear_previous_target();
  for(position = 0; position < NOUTPUTS; position++)
{
target_vector[tags[(pattern_counter+position)%NPATTERNS]+NTAGS*position]=1.0;
}
}

/*-------------------------------------------------------------*/

clear_previous_input()
{
  register int position;

  for(position = 0; position < NINPUTS; position++)
{
input_vector[tags[(pattern_counter+position-1)%NPATTERNS]+NTAGS*position]=0.0;
}
}

/*-------------------------------------------------------------*/

clear_previous_target()
{
```

```
  register int position;

  for(position = 0; position < NOUTPUTS; position++)
{
target_vector[tags[(pattern_counter+position-1)%NPATTERNS]+NTAGS*position]=0.0;
}
}

/*-------------------------------------------------------------*/
/* for every pattern in the file we want to train n,...,n+5 -> n,...,n+6 */

static void make_patterns()
{

int test;

  pattern_counter = ((pattern_counter+1) % NPATTERNS);

  set_input();
  set_target();
}

/*-------------------------------------------------------------*/

void user_init()
{
  zero_vectors();
  set_priors(); /*read the priors from PRIORSFILE into prior[0:NTAGS-1]*/
  readdata();   /* read the data from DATAFILE into tags[]*/

  syntax_set_inertia(0.0);

  syntax_set_input(input_vector);
  outputs_set_target_output(target_vector);

  define_generator(make_patterns, (char *)NULL, "Patterns");
}

/*-------------------------------------------------------------*/

/* Totally irrelevant to the data generator, this section defines
cross entropy error which the syntax.aspirin network specification
calls via "ErrorFunction -> CrossEntropy" */

float CrossEntropy (float *target, float *output, float *credit, int n)
{
  float TotalError = 0.0;
  int i;
```

```
  for(i=0;i<n;i++) {
    credit[i] = (target[i] - output[i])/(output[i]*(1-output[i]));
    TotalError -= target[i] * log(output[i]) + (1-target[i]) * log(1 - output[i]);
  }/* end for */
  printf("error = %f\n",TotalError);
  return( TotalError );
}

/*-------------------------------------------------------------*/

/* This section defines the output filter that takes linear outputs
and priors and makes a net input for sigmoidal units in another BlackBox.*/

void AddPriors (float *output, int width, int height)
{
  int i;

  for(i=0;i<width*height;i++) {
output[i] += priors[i%NOUTPUTS];
  }/* end for */
}
```

6.2 Prior estimator for LOB model

```
/*************************************************************************

syntax/makepriors.c: created 7/1/93  by Ben Martin

This program computes priors to weight the categories for the
syntactic prediction task for the LOB corpus.

*************************************************************************/

#include <stdio.h>
#include <math.h>

#define DATAFILE "LOB.freq"
#define PRIORSFILE "LOB.priors"
#define NPATTERNS 1151707   /*# of training items in DATAFILE */
#define NTAGS 131   /* # of syntactic categories (tags)*/

/*-------------------------------------------------------------*/

main()
{
  FILE *fp1, *fp2, *fopen();
  register int counter;
```

```
  int frequency;
  double P_c;
  double prior;

  printf("\nReading from: %s\n", DATAFILE);
  printf("\nWriting to: %s\n", PRIORSFILE);

  fp1 = fopen(DATAFILE, "r");
  fp2 = fopen(PRIORSFILE, "w");

  if (fp1 == (FILE *)NULL) {
    fprintf(stderr, "\nUnable to open %s!\n", DATAFILE);
    exit(1);
  }
  if (fp2 == (FILE *)NULL) {
    fprintf(stderr, "\nUnable to open %s!\n", PRIORSFILE);
    exit(1);
  }

  /* make the priors */
  for(counter = 0; counter < NTAGS; counter++) {
    fscanf(fp1,"%d", &frequency);
    P_c = (double) (frequency) / (double) (NPATTERNS);
prior = log(P_c/(1-P_c));
    printf("%g\n", prior);
    fprintf(fp2,"%g\n", prior);
  }/* end loop */

  fclose(fp1);
  fclose(fp2);
  printf("\nDone making priors.\n");
}

/*-------------------------------------------------------------*/
```

6.3 ASPIRIN simulator code for LOB network

```
#define N_INPUTS   786
#define N_OUTPUTS  786
#define N_HIDDEN   40

DefineBlackBox syntax
{
    OutputLayer->  Linear_Output
    InputSize->    N_INPUTS
    OutputFilter-> AddPriors
    Components->
    {
```

```
  LinearNode Linear_Output [N_OUTPUTS]
    {
InputsFrom-> Hidden_Layer
    }
PdpNode Hidden_Layer [N_HIDDEN]
    {
InputsFrom-> $INPUTS
    }
    }
}

DefineBlackBox outputs
{
    OutputLayer->  Output_Layer
    ErrorFunction-> CrossEntropy
    Components->
    {
  PdpNode Output_Layer [N_OUTPUTS]
    {
InputsFrom-> !syntax ( with a [1 x 1] Tessellation
using a 0 Xoverlap)
    }
    }
}
```

7 Appendix 2

7.1 Pattern generator and error function for WSJ model

```
/*****************************************************************************

semantics/user_init.c: created 7/23/93  by Ben Martin
last modified: 8/5/93

This program generates word pairs for a next word prediction network.
This version also uses the WSJ corpus.  The top 2000 words are coded as
integers.  This program takes a file containing the integers in their
order of appearance in the corpus and converts them into input/target pairs
for training a network.  This code also defines the exponential and log
units, and cross-entropy error.

This file is for use with the aspirin/migraines network simulator.
It contains the data generator for the network defined in semantics.aspirin.
It draws data from WSJ.lexnums.


*****************************************************************************/
```

```
#include "semantics.h"

extern void define_generator(void (*f)(char *item), char *item, char *string);

#define DATAFILE "/m0/WSJ/lexnums/lexnums"
#define NPATTERNS 6065500   /*# of training items in DATAFILE */
#define NWORDS 2001   /* # of lexical categories */
#define NINPUTS 1   /* number of words in input vector */
#define NOUTPUTS 1  /* number of words in the output */
#define PERIOD 2    /* the lexnum-1 for a period (which is pattern 0) */

/* data: words */
static int words[NPATTERNS];

/* input & target buffers */
static float input_vector[NINPUTS * NWORDS], target_vector[NOUTPUTS * NWORDS];

/* pattern counter */
static int pattern_counter = NPATTERNS-1;

/*-------------------------------------------------------------*/

/* set input and target to 0's */
static void zero_vectors()
{
  register int counter = NINPUTS * NWORDS;
  while(counter--) {
input_vector[counter] = 0.0;
  }
  counter = NOUTPUTS * NWORDS;
  while(counter--) {
target_vector[counter] = 0.0;
  }
}

/*-------------------------------------------------------------*/

static void readdata()
{
  FILE *fp, *fopen();
  register int counter;
  int word;

  printf("\nUsing datafile: %s\n", DATAFILE);

  fp = fopen(DATAFILE, "r");
  if (fp == (FILE *)NULL) {
```

```
    fprintf(stderr, "\nUnable to open %s!\n", DATAFILE);
    exit(1);
  }

  /* load the words */
  for(counter = 0; counter < NPATTERNS; counter++) {
    fscanf(fp,"%d", &word);
    words[counter] = word-1;     /* change from 1-2001 to 0-2000 indices.*/
  }/* end loop */

  fclose(fp);
  printf("\nDone reading data.\n");
}

/*-------------------------------------------------------------*/

set_input(/*int pattern_counter*/)
{
  register int position;

  clear_previous_input();
  for(position = 0; position < NINPUTS; position++)
{
 input_vector[words[(pattern_counter+position)%NPATTERNS]+NWORDS*position]=1.0;
}
}

/*-------------------------------------------------------------*/

set_target(/*int pattern_counter*/)
{
  register int position;

  clear_previous_target();
  for(position = 0; position < NOUTPUTS; position++)
{
target_vector[words[(pattern_counter+position)%NPATTERNS]+NWORDS*position]=1.0;
}
}

/*-------------------------------------------------------------*/

clear_previous_input()
{
  register int position;

  for(position = 0; position < NINPUTS; position++)
{
```

```
input_vector[words[(pattern_counter+position-1)%NPATTERNS]+NWORDS*position]=0.0;
}
}

/*-------------------------------------------------------------*/

clear_previous_target()
{
  register int position;

  for(position = 0; position < NOUTPUTS; position++)
{
target_vector[words[(pattern_counter+position-1)%NPATTERNS]+NWORDS*position]=0.0;
}
}

/*-------------------------------------------------------------*/

static void make_patterns()
{

int test;

  pattern_counter = ((pattern_counter+1) % NPATTERNS);
  set_input();
  set_target();
}

/*-------------------------------------------------------------*/

void user_init()
{
  zero_vectors();
  readdata();   /* read the data from DATAFILE into words[]*/

  semantics_set_inertia(0.0);

  InputBox_set_input(input_vector);
  OutputBox_set_target_output(target_vector);

  define_generator(make_patterns, (char *)NULL, "Patterns");
}

/*-------------------------------------------------------------*/

/* This section defines cross entropy error which the semantics.aspirin
network specification calls via "ErrorFunction -> CrossEntropy" */
```

```
float CrossEntropy (float *target, float *output, float *credit, int n)
{
  float TotalError = 0.0;
  int i;

  for(i=0;i<n;i++) {
if(target[i] != output[i])
  {
if(output[i]!=0.0 && output[i]!=1.0)
  {
credit[i] = (target[i] - output[i])/(output[i]*(1.0-output[i]));
TotalError -= target[i]*log(output[i])+(1.0-target[i])*log(1.0-output[i]);
  }
else
  {
credit[i] = (target[i] - output[i])*100000.0;
TotalError -= 11.513;
  }
  }
else credit[i] = 0.0;
  }/* end for */
  printf("error = %f\n",TotalError);
  return( TotalError );
}

/*-------------------------------------------------------------*/

/* This section defines the exponential units (and derivative!) */

float exp_x ( float x )
{
  return(exp(x));
}

/*-------------------------------------------------------------*/

/* This section defines the log units. */

float log_x ( float x )
{
  if (x==0.0) return(-20.0);
else return(log(x));
}

/* This section defines the derivative of the log units. */

float one_over_x ( float x )
{
```

```
  if (x==0.0) return(10000000.0);
else return(1/x);
}


/*-------------------------------------------------------------*/

/* This section defines the function that sets the filter weights to one*/

float fixed_one(int nx, int ny, int wx, int wy)
{
  return(1.0);
}

float fixed_neg_one(int nx, int ny, int wx, int wy)
{
  return(-1.0);
}
```

7.2 ASPIRIN simulator code for WSJ network

```
#define N_INPUTS   2001
#define N_OUTPUTS  2001
#define N_INCATS   100
#define N_OUTCATS   100
#define N_HIDDEN   50

DefineBlackBox OutputBox
{
    OutputLayer->  output_layer
    ErrorFunction-> CrossEntropy
    Components->
{
  PdpNode output_layer [N_OUTPUTS]
    {
InputsFrom-> !OutcatBox
    }
}
}

DefineBlackBox OutcatBox
{
    OutputLayer->  outcats
    Static->
    Components->
    {
UserNode exp_x exp_x
  outcats [N_OUTCATS]
```

```
    {
InputsFrom-> outcat_norm
( with a Shared [1 x 1] Tessellation
using a 1 Xoverlap
initialized with a fixed_neg_one )
 and !HiddenBox
( with a Shared [1 x 1] Tessellation
using a 0 Xoverlap
initialized with a fixed_one )
    }

UserNode log_x one_over_x
  outcat_norm [1]
    {
InputsFrom-> exp_outcats
( with a Shared [N_OUTCATS x 1] Tessellation
initialized with a fixed_one )
    }

UserNode exp_x exp_x
  exp_outcats [N_OUTCATS x 1]
    {
InputsFrom->  !HiddenBox
( with a Shared [1 x 1] Tessellation
using a 0 Xoverlap
initialized with a fixed_one )
    }
     }
}


DefineBlackBox HiddenBox
{
    OutputLayer->  linear_outcats
    Components->
    {
  LinearNode linear_outcats [N_OUTCATS x 1]
    {
InputsFrom-> hidden_layer
    }

  PdpNode hidden_layer [N_HIDDEN]
    {
InputsFrom-> !IncatBox
    }
    }
}
```

```
DefineBlackBox IncatBox
{
    OutputLayer->  incats
    Static->
    Components->
    {
UserNode exp_x exp_x
  incats [N_INCATS]
    {
InputsFrom-> incat_norm
( with a Shared [1 x 1] Tessellation
using a 1 Xoverlap
initialized with a fixed_neg_one )
and !InputBox
( with a Shared [1 x 1] Tessellation
using a 0 Xoverlap
initialized with a fixed_one )
    }

UserNode log_x one_over_x
  incat_norm [1]
    {
InputsFrom-> exp_incats
( with a [N_INCATS x 1] Tessellation
initialized with a fixed_one )
    }

UserNode exp_x exp_x
  exp_incats [N_INCATS x 1]
    {
InputsFrom->  !InputBox
( with a Shared [1 x 1] Tessellation
using a 0 Xoverlap
initialized with a fixed_one )
    }
     }
}

DefineBlackBox InputBox
{
    OutputLayer->  linear_incats
    InputSize->    N_INPUTS
    Components->
{
  LinearNode linear_incats [N_INCATS x 1]
    {
InputsFrom-> $INPUTS
    }
```

```
    }
}
```