

SICS Technical Report  
T92:10

ISRN : SICS-T--92/10-SE  
ISSN : 1100-3154

## **Learning Swedish Morphology with a Neural Network**

by

Martin Eineborg

**Swedish Institute of Computer Science  
Box 1263, SE-164 29 Kista, SWEDEN**

---

ISRN SICS-T--92/10--SE

# **Learning Swedish Morphology with a Neural Network**

by

**Martin Eineborg**

# **Learning Swedish Morphology with a Neural Network**

**A Master of Science Thesis Work  
by  
Martin Eineborg**

**University of Uppsala  
August 1992**

**Funded by and performed at the  
Swedish Institute of Computer Science (SICS)  
Box 1263  
S-164 28 Kista**



## **ABSTRACT**

In this thesis an attempt to derive word classes from word-endings using a neural network is done. The network which uses the back-propagation algorithm was trained with pairs of word-endings and word categories. A brief summation of some important concepts in the field of abstract neural networks is given. Following this the actual task is attacked and its results discussed. The results (the net gives a better result than a pure guess) are indicating that to fully achieve the goal improvements are necessary. Suggestions for these improvements and future works are concluding this paper.

## **ACKNOWLEDGEMENTS**

I would like to thank Björn Gambäck at the Swedish Institute of Computer Science for help and guidance. I would also like to thank Lars Borin at University of Uppsala for his suggestions and patience.



## CONTENTS

1 Introduction .....	1
1.1 The Nervous System .....	1
1.2 History of Artificial Neural Networks .....	2
1.3 Classification of Networks .....	4
1.4 McCulloch and Pitts's Neuron Model .....	5
1.5 Hopfield Net .....	5
1.6 Self-organizing Maps .....	6
1.7 Backpropagation .....	6
2 Examples of Inputs and Outputs .....	7
3 Choice of Net .....	8
3.1 Input-output Function .....	8
3.2 The Chosen Algorithm .....	9
3.3 Network Topology .....	9
3.4 Information Coding .....	10
4 Implementation Issues .....	11
5 Training .....	12
6 Results .....	12
7 Discussion .....	14
8 Improvements .....	15
9 Conclusion .....	16
Appendix A Examples of Actual Net Input and Output .....	17
References .....	20
C-code for a DFA and the Network .....	23



## 1 Introduction

In 1988 Manny Rayner and Åsa Hugosson [Rayner *et al* 88] did some work on lexicon learning at SICS. This project was followed up by Sofia Hörmander later in 1988 [Hörmander 88]. A formal grammar was used along with example-sentences to deduce a lexicon. Unfortunately, the exponential explosion that followed from ambiguities in the grammar caused the system to be very slow. One way to attack this problem would be to let a neural network suggest restrictions on the possible word-classes of the unknown word derived from its morphology.

Thus, this project is based on the idea that some relationship exists between the grammatical category of a word and its last letters a neural network trained with a sufficient amount of word-endings should be able to discover this.

### 1.1 The Nervous System

In the body there is a wide variety of neurons but certain features are the same. The main body of the neuron is called the *soma*. Extending from the soma are a set of short branches called *dendrites*. Also attached to the body is a long fibre called *axon*. The axon divides and each branch ends in a *terminal button*. Contact with other neurons are made as the terminal button meets the dendrite of another neuron. This neuron-to-neuron contact is called a *synapse*. When certain conditions are met the neuron fires, i.e. an impulse is propagated down the axon. There are two kinds of synapses, excitatory and inhibitory. An input from an *excitatory* synapse will increase the probability of the neuron firing and an input from an *inhibitory* will decrease the probability. Synapses are created or destroyed when something is learned. A neuron can receive from 5000 to 15000 impulses from other neurons. In an average human brain there are about  $10^{11}$  neurons. These are arranged in very complex networks.

## 1.2 History of Artificial Neural Networks

Several researchers around 1940 suggested that a more brain-like machine should be created. The first step towards such a model was taken by McCulloch and Pitts [McCulloch & Pitts 43] when they proposed their neuron model (see section 1.4). A second step was taken in 1949 by Donald Hebb [Hebb 49] when he gave the biological basis for a learning rule. He suggested that if a neurons input fires consistently at the time that the neuron itself is firing, then the weight for that synapse will be strengthened.

Around 1960 Bernard Widrow and Marcian E. Hoff [Widrow 62] came up with a learning rule called the Widrow-Hoff rule or the *delta rule*. It can be expressed as:

$$w_k(t+1) = w_k(t) + \alpha \delta_k(t) x_k(t)$$

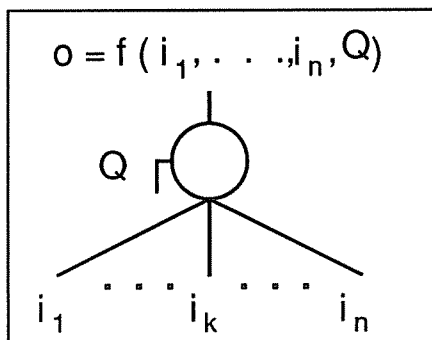
where  $\alpha$  is a constant (gain term) typically  $0.01 \leq \alpha \leq 10$

$w_k(t)$  is the value of weight  $k$  at time  $t$

$\delta_k(t)$  is the error of neuron  $k$  at time  $t$

$x_k(t)$  is the incoming value from neuron  $k$  at time  $t$ .

It was shown by Rosenblatt in 1962 that the delta rule causes the weights to converge [Rosenblatt 62]. Rosenblatt also developed the perceptron (see figure 1) which basically is a McCulloch and Pitts neuron that can classify a continuous valued or binary valued input into one of two classes.



$i_k$  is the input  
 $Q$  is a threshold value  
 $o$  is the output which is calculated by the function  $f$

fig. 1

A serious blow against neural science came in 1969 when Minsky and Papert [Minsky & Papert 69] exposed some of the limitations of the perceptron. They showed that some problems can not be solved with a perceptron neural network consisting of only one layer, e.g. it is not possible to perform the XOR-function. A collection of points in space is said to be linearly separable if there exist a line that can separate the classes. In higher dimensions it is called hyperplanar separability when a hyperplane is used to separate classes. A single layer perceptron network can only perform a linear separable function.

The behaviour of the net can be viewed graphically where the classes are separated by a hyperplane. The picture below is showing where two classes are separated by a plane.

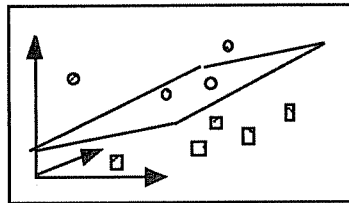


fig. 2

The hyperplane specifies *decision regions*. A decision region specifies to which class an input value is assigned.

In order to handle nonlinear functions a hidden layer has to be included in the net. A *hidden neuron* receives input from other neurons and transmits output to other neurons. A *hidden layer* consists only of hidden neurons. In 1986 Rumelhart, Hinton, and Williams [Rumelhart *et al* 86] came up with a network that could handle hidden layers. The method is called backpropagation and uses supervised learning (see section 1.7). The network usually is a feed-forward net, although other topologies are possible. For example, some interesting research on how to use the back-propagation algorithm in recurrent nets, i.e. networks with cycles, have been done [Rumelhart *et al* 86, Pineda 87].

It has been shown that arbitrary decision regions can be arbitrarily well approximated by feedforward neural networks with one hidden layer [Cybenko 89].

In 1982 an influential network, called the Hopfield network, was created by John Hopfield, professor of Biology and Chemistry at the California Institute of Technology. Hopfield's idea was to use the network as an associative memory [Hopfield 82].

Another important model was created by Kohonen. The Kohonen network differs from the previous in that it organizes the input data by itself. He used maps and unsupervised training [Kohonen 84, Kohonen 88].

In the following sections these network models will be addressed in greater detail.

### 1.3 Classification of Networks

Over the years several learning laws have been developed. There are now several kinds of networks. Neural networks can roughly be characterized by

- summation- and outputfunction

How is the total incoming value calculated? Is it the inner product of the incoming values and the weights?

How is an output formed? Through what function is the total incoming value passed? Is it a sigmoid function or a hard limiter?

- network topologies

How many layers are there and how many neurons per layer are there? Is it a fully connected net, ie a neuron in layer  $k$  receives input from all neurons in layer  $k-1$ ? Is it a partially connected net, ie a neuron in layer  $k$  receives input only from some neurons in layer  $k-1$ ?

- supervised or unsupervised learning.

See section 1.6

## 1.4 McCulloch and Pitts's Neuron Model

In 1943 the neurophysiologist Warren McCulloch and the logician Walter Pitts proposed a model of a neuron [McCulloch & Pitts 43]. Just like the biological neuron, McCulloch and Pitts's artificial neuron takes several inputs and produces one output. The changes in synapses are simulated by *weight* variables. A weight has two features: the sign of the weight determines if the incoming impulse is excitatory or inhibitory and the absolute value of the weight determines to what degree notice should be taken of the incoming impulse. When the incoming values are above a certain level (the threshold) the neuron fires. This is handled by the *firing rule* of the neuron. In the McCulloch and Pitts model the firing rule can be expressed by the following simple mathematical formula:

the neuron fires iff  $\sum_k x_k w_k > t$

where  $x_i$  is the value received from neuron  $i$   
 $w_i$  is the weight associated with input from neuron  $i$   
 $t$  is the threshold

In McCulloch and Pitts's neuron only two values are allowed as output, 1 or 0. An output of 1 means that the neuron fires and an output of 0 means no firing at the neuron. Modification of the weights is handled by a *learning rule* which is used when the net is to learn something.

This simplified model of a neuron is still a basis of many current neural networks.

## 1.5 Hopfield Net

The Hopfield model is a fully associative network consisting of only one layer. A fully associative network means that the outputs of all neurons are connected to some other neurons. The purpose of this model is that the network, when given some input, converge to a stable state that can be seen as the network's response to the given input pattern. The model uses the same kind of neurons and firing rule as McCulloch and Pitts.

Each neuron fires individually and at an equal probability. As soon as one neuron fires the state of the net is changed. The point of the training is to make certain states stable.

## 1.6 Self-organizing Maps

Teuvo Kohonen has developed a network that uses unsupervised learning.

When training a net with *supervised learning* both the input pattern and the correct class of that pattern are presented to the network.

When training a net with *unsupervised learning* only the input pattern is presented to the network. The net has to find suitable classes for the material by itself.

A Kohonen network consists of a number of neurons organized in a two-dimensional plane, this organization is called a map.

The input pattern is given to all neurons at the same time. All neurons are output neurons. The neuron for which the Euclidean distance between the input-vector and the weight-vector is a minimum is selected as being the response of the given pattern. The neighbours of the response neuron are selected and their weights are altered to decrease the distance to the input pattern. Kohonen lets the neighbourhood area decrease with time. The amount of which the distance is decreased is also decreased by time.

This way the net learns to respond in a different place on the map for input patterns that are different and the map is said to become an ordered map.

## 1.7 Backpropagation

Backpropagation uses a two-phase learning cycle. During the first phase, the input pattern is propagated through the network. Some sort of distance, usually the Euclidean distance, is calculated between the actual output and the desired output of the net. This distance is the error of the net.

The second phase starts with the error being propagated backwards through the net, adjusting the weights along its way. Then the next pattern can be processed. This cycle continues until the net satisfactory has learnt all patterns. The neurons used differs from those used by McCulloch and Pitts in that real values are used as weights, thresholds, and outputs.

The output of the neuron is given by:

$$o_m = 1 / (1 + e^{-a(m)}) \text{ where } a_m \text{ is the activation of the } m\text{:th neuron.}$$

The activation  $a_m$  of neuron  $m$  is:

$$a_i = \sum_{vj} (w_{ij} * x_{ij}) + t_i$$

where  $w_{ij}$  is the  $j$ :th weight of neuron  $i$   
 $x_{ij}$  is the  $j$ :th input to neuron  $i$   
 $t_i$  is the threshold of neuron  $i$ .

There are two weight adjustment rules:

$$\begin{aligned} \text{for output neurons the error } \delta_{pj} &= (t_{pj} - o_{pj}) o_{pj} (1 - o_{pj}) \\ \text{for hidden neurons the error } \delta_{pj} &= (\sum_{vk} \delta_{pk} w_{kj}) o_{pj} (1 - o_{pj}) \end{aligned}$$

When the training of the net is over, the weights are frozen and need not be altered. The training phase is the most time consuming part of using a network.

## 2 Examples of Inputs and Outputs

The network should be given enough information to be able to discover the relationship between the last letters of a word and its grammatical category. In reply to a word-ending the network should be able to give enough information to remove some of the ambiguities that occur in the grammar.

Examples of what could be input and output of the net:

word-ending	answer
-ande	verb in present participle (eg hoppande)
-ade	verb in imperfect or preteritum (hoppade)
-are	adjective in comparative (vackrare)
-ast	adjective in superlative (vackrast)
-arna	noun in plural determinate form (bilarna)
-en	noun in singular determinate form (bilen)

More specifically, the net should be able to determine if a word is

- a noun and if so, has it a definite or indefinite article and is it the genitive case.
- is it an adjective and if so, is it in comparative case or superlative
- is it a verb and if so, is it in present, preteritum, imperative, infinitive, or supine.

As almost all suffixes in Swedish consists of four letters or less, a net with four or perhaps even three letters should be sufficient. A smaller net leads to shorter training time but unfortunately it might also lead to decrease in performance.

### 3 Choice of Net

In choosing a network several facts have to be considered: the input-output function, the training algorithm, the network topology, information coding. These issues will now be addressed in turn.

#### 3.1 Input-Output Function

The net is supposed to perform some kind of function between two domains  $f: X \rightarrow Y$ . For every element  $x \in X$  do we know which element  $y \in Y$  it should be related to? If not a unsupervised neural network

should be considered, eg Kohonen's selforganizing networks. How many tuples  $\langle x,y \rangle$  can we find? Are there enough for the training and testing of the network?

In this case several thousands of tuples existed. All necessary information came from a text [Teleman 74]. The only thing that had to be done was to choose a word along with its grammatical category and extract the last three letters. This was done by a DFA (Deterministic Finite Automata) written in C (see appendix).

### 3.2 The Chosen Algorithm

The material suggested that some sort of supervised learning was suitable. The previously described backpropagation algorithm is a reliable, easy to implement and probably the most often used algorithm. Therefore the backpropagation algorithm was chosen.

### 3.3 Network Topology

Determining the topology of the network is the most difficult part of using a neural network.

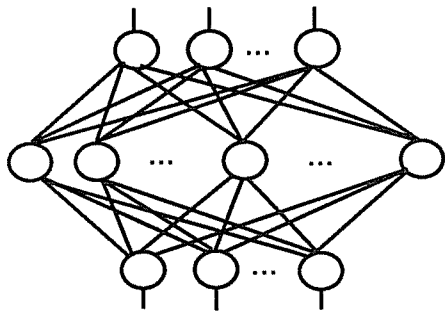
Two main topics have to be considered, how many layers should there be and how many neurons should there be in each layer?

This is not easy to answer. If the network is to perform a linear separable function, only one layer is enough. If the network is to perform a nonlinear function one or more hidden layers may be needed. A net with too few neurons will have problems classifying the material. A network with too many neurons leads to memorization of the training material and thus will not perform correct when presented with new unseen material.

In order to ensure that the network was capable of performing the desired function a three layer network with one hidden layer was chosen. Two sizes of the hidden layer were tried. The first had 87 neurons and the second had 20 neurons.

So the result was two three layer architectures. One with 87 input neurons, 87 hidden neurons, and 13 output neurons. And one with 87 input neurons, 20 hidden neurons, and 13 output neurons.

Pictures of the two networks can be seen below.

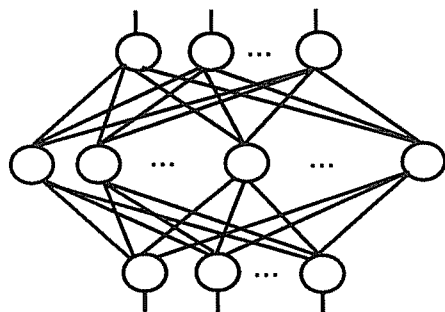


13 output neurons

87 hidden neurons

87 input neurons

fig. 3 showing net 1



13 output neurons

20 hidden neurons

87 input neurons

fig. 4 showing net 2

### 3.4 Information Coding

The information can be encoded in several ways before it is passed to the input layer of the network.

For example the information can be in

- localized form      This means that each value has a position of its own in the input layer. An example of this coding the numbers 1, 2 and 3 as 100, 010 and 001.
  
- distributed form    This means that each value is spread all over the input layer. An example of this is coding the numbers 1, 2 and 3 as 110, 101 and 111.

The data can also be represented as a real number, an integer, or as a binary number.

Several forms of representations were tested (see results below section 10), but the main focus was on a localized binary representation.

Since the Swedish alphabet consists of 29 letters, a vector of 29 binary numbers was used for each letter. This means, as only the last three letters were used, that a vector of 87 binary numbers was needed. In this vector the digits 1 and 0 were used to indicate if a letter is present or not. What happens if a word has less than three letters? For example, this can occur if the net is given the word "ge" (give). It was solved by representing the missing letter with a nullvector, ie a vector with 29 zeros. The outputs of the net consisted of 13 real numbers. The net has output values for noun, adjective, verb, determinate article, genitive, comparativ, superlativ, present, preteritum, imperative, infinitive, supine and passive. A high output value means that the network shows a high degree of certainty.

#### **4 Implementation Issues**

SICS had a general neural network package [Hewetson 1990] in which a network could have been developed. Unfortunately this was a beta version and thus not very reliable. Most of the features were not yet implemented and the program crashed several times. So a specialized network had to be built.

The developed network package consists of two programs, one used for training and one for using the trained network. When the training is done all weights are saved on a file. This file is used when the network is to be tested. The weights are not altered during the testing phase. The programs were written in C.

## 5 Training

The network has been trained with a text consisting of 500 words. Although more words were available the training took too much time. The training of the network is suspiciously slow, but I have not found any particular reason for this. As the training took so much time, the training continued for a given number of cycles and not until an error function was minimized.

## 6 Results

Unfortunately the results are not what I hoped for. Neither network topology did well in learning.

The following figures are very uncertain and does only consider if the net has correctly characterized the word categories noun, verb, or adjective. The net has been trained using 500 word-endings which has been processed 1000 times.

A text with previously unseen words was presented to the two different topologies. This shows that the network with 87 neurons in the hidden layer predicts the right category in 59% of the previously unseen words.

The network with 20 hidden neurons did not perform any better since it only could give a correct categorization in 49% of the words.

This is however better than a pure guess which would result in a 33% chance of getting the answer right.

It is reasonable that we only consider answers when the net output has reached a sufficiently high level of confidence. This was accomplished by setting up a constraint saying that the value of an answer has to be greater than 0.5 to be considered correct.

It is also reasonable that there should be only one winner among mutually exclusive classes. This is the case with noun, adjective and verb. This is accomplished by demanding a difference between the answer and next most likely answer to be greater than 0.1. A test with the two constraints showed a better result than before.

The 87-87-13 net gave the best result with a correct answer in 63% of the words.

The 87-20-13 network gave a correct answer in 59% of the words.

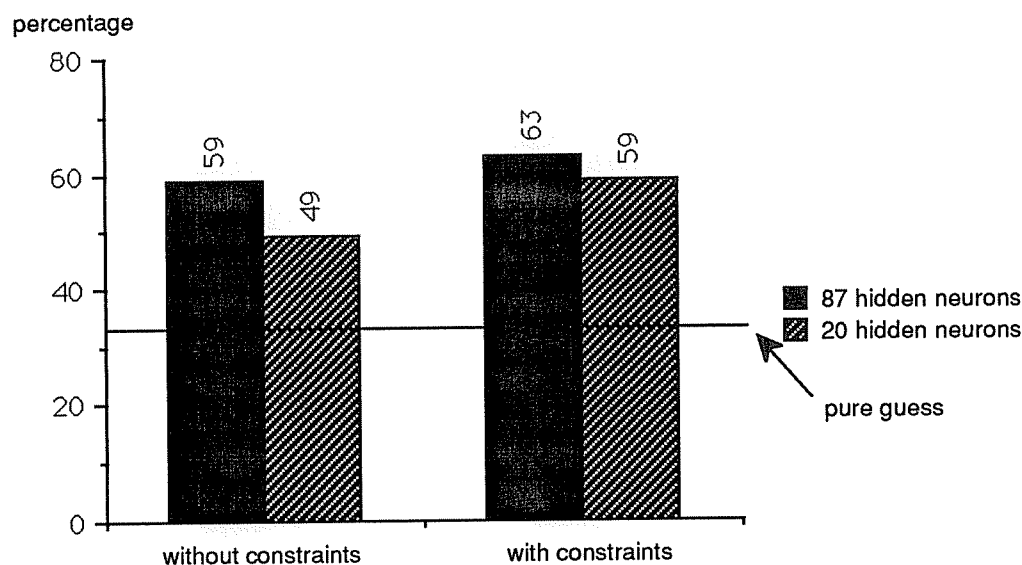


fig. 5 showing results

A network with 15 input neurons, 15 hidden neurons and 13 output neurons was also constructed. This net was used to test a distributed representation of the letters. A letter was represented by 5 binary numbers. For example, the letter A had representation 00001, B had repr. 00010, C had repr. 00011 and so on. As before, a missing letter was represented by zeros, ie 00000. The last three letters were used and this meant that only 15 binary numbers were needed. The net had a very difficult time trying to learn. It was only able to learn very few examples and as soon as the number of examples increased it's performance decreased rapidly.

## 7 Discussion

The main problem lies in deciding how many neurons to use in the hidden layers. Testing different number of neurons would seem the obvious solution. This is however not possible since the training takes a small eternity.

Training on a small net consisting of 40 inputs, 20 neurons in the first hidden layer, 4 neurons in the second hidden layer, and 4 output neurons shows that the machine needs approximately  $10^{-5}$  seconds for each weight. This means that it takes about 4 seconds for a net of 2 inputs, 2 weights in the first hidden layer, 1 weight in the second layer, and 1 output neuron to learn the XOR-function.

This would also imply that a net of size 87 inputs, 87 neurons in the first hidden layer, 87 neurons in the second layer, and 13 output neurons would need 119 hours or nearly 5 days in order to process 500 examples 5000 times.

The network performs well when it is to learn no more than a few input patterns since then the scale of the network can be held in reason. But when increasing the number of input patterns the network obviously has to include more neurons or even a second hidden layer. This slows down the training process enormously.

Not knowing whether it is the size of the network or not enough training that is to blame for the results makes it very difficult indeed.

What is needed is a different approach to the problem. A different training algorithm. Perhaps the problem can be solved with the suggested improvements (seen below in section 8) to the backpropagation algorithm or a completely different type of network, eg Kohrens network, might be another solution.

## 8 Improvements

There are several possible explanations for the low recognition rate. In the training of the net only 500 word endings was used. Several words occurred more than one time in the training text. Removing the duplicates would leave only about 220 word endings. This is probably all too few for the net to detect the necessary pattern. The net should be trained on larger texts containing several thousands of words.

In determining the grammatical categories of the words only the last three letters were used. This is probably too few letters. Since several affixes in Swedish consists of four letters it seems reasonable that the last four letters of a word should be considered by the net. The training time for a net with  $4 \cdot 29 = 116$  input neurons will, of course, increase. A possible compromise is feeding the net with the last three letters along with information telling whether the fourth last letter is a consonant, vowel or nothing at all. To realise this, only 3 more neurons would be needed instead of adding the full 29. Not using all information possible from the fourth last letter is based on the assumption that the influence of the letters, in determining the category of the word, is decreasing by the distance from the end of the word.

Another possible improvement is to include the length of the word along with the last letters of the word.

Instead of just using the last letters of a word we could include information from the surrounding words. For example, if we knew which grammatical categories the two preceding words belonged to we could use this information to determine the category of the present word.

Nakamura, Maruyama, Kawabata, and Shikano [Nakamura *et al* 90] have been investigating word category prediction with neural networks. They use the grammatical categories of the preceding words to predict the category of the next word. This information could also be included in our net along with the last letters of the present word.

Improvements to the standard backpropagation algorithm exists and could be used to decrease the training time [Fukumi, Omatu 91]. This would perhaps make it possible to realise some of the above suggestions.

And, of course, a completely different neural network algorithm could be used.

## 9 Conclusions

An attempt to derive word classes from word-endings using a neural network has been done. The network which uses the back-propagation algorithm were trained with pairs of word-endings and word categories.

The results are better than what a pure guess would be. However, to increase performance some improvements should be made, both to the input data and to the actual training algorithm.





0.0013	imperative
0.0087	infinitive
0.0118	supine
0.6211	passive

The high value at the third output indicates that the net believes that the word-ending belongs to a verb.

The net with 20 hidden neurons shows less confidence when given the same input.

0.2672	noun
0.1143	adjective
0.5317	verb
0.3357	determinate article
0.0332	genetive
0.0024	comparativ
0.0085	superlativ
0.3810	present
0.0029	preteritum
0.0278	imperative
0.1315	infinitive
0.0443	supine

## References

- Aleksander I., Morton H.,  
*An Introduction to Neural Computing*,  
Chapman & Hall, London, 1990.
- Cybenko, G.,  
"Approximation by Superpositions of a Sigmoidal Function",  
*Mathematical Control Systems Signals*, 2, 1989.
- Fukumi, Minoru, Sigeru, Omatu,  
"A New Back-Propagation Algorithm with Coupled Neuron",  
*IEEE Transactions on Neural Networks*, September 1991.
- Hebb, D.,  
*The Organization of Behaviour*,  
Wiley, New York, 1949.
- Hecht-Nielsen, R.,  
*Neurocomputing*,  
Addison-Wesley, 1990.
- Hewetson, M.,  
*Pygmalion Neural Network Programming Environment*,  
Esprit Project 2059,  
Department of Computer Science, University College London,  
1990.
- Hopfield, J.,  
"Neural Networks and Physical Systems with emergent  
collective properties",  
*Proceedings of the National Academy of Science USA*,  
79: 2554-2558, 1982.
- Hörmander, S.,  
The Problems of Learning a Lexicon with a Formal Grammar  
SICS Research Report - R88019, Stockholm, Sweden, 1988.
- Khanna, T.,  
*Foundations of Neural Networks*,  
Addison-Wesley, 1990.

- Kohonen, T.  
*Self-Organization and Associative Memory*  
Springer-Verlag, Heidelberg, 1984/1988.
- McCulloch, W. S., Pitts, W. H.,  
"A Logical Calculus of the Ideas Imminent in Nervous  
Activity",  
*Bulletin of Mathematical Biophysics*, 5: 115-133, 1943.
- Minsky, M., Papert, S.  
*Perceptrons: an Introduction to Computational Geometry*,  
MIT Press, Massachusetts, 1969.
- Nakamura, M., Maruyama, K., Kawabata, T., Kiyohiro, S.,  
"Neural Network Approach to Word Category Prediction for  
English Texts"  
*13th International Conference on Computational Linguistics*,  
Vol. 3, Helsinki, 1990.
- Pineda, F. J.,  
"Generalization of back-propagation to recurrent neural  
networks"  
*Physical Review Letters*, vol. 59, pp. 2229-2232, 1987
- Rayner, M., Hugosson, Å., Hagert, G.,  
Using a Logic Grammar to Learn a Lexicon  
SICS Research Report - R88001, Stockholm, Sweden, 1988.
- Rosenblatt, F.,  
*Principles of Neurodynamics: Perceptrons and the Theory of  
Brain Mechanism*,  
Spartan Books, New York, 1962.
- Rumelhart, D. E., Hinton, G. E., Williams, R. J.  
"Learning internal representations by error propagation",  
*Parallel Distributed Processing vol 1 and 2*  
MIT Press, Massachusetts, 1986.
- Teleman, U.,  
*Manual för grammatisk beskrivning av talad och skriven  
svenska (in Swedish)*,  
Studentlitteratur, Lund, 1974.

Widrow, B.,  
Generalization and information storage in networks of  
ADALINE neurons, *Self-Organizing Systems*,  
Spartan Books, New York, 1962.

## C-code

The following pages contain C-code examples for the implementation of a Deterministic Finite-state Acceptor (DFA) for finding word-endings in a corpus and for the actual neural network:

DFA for finding suitable word-endings in a corpus. ...	24
Declarations for the neural net.....	30
Primitives.....	31
Main functions for the forward phase.....	36
Main functions for the backward phase.....	38
Program to train the network.....	42
Program to initialize the net and process words.....	45

```

/*
   This is a DFA for finding suitable word-endings in corpus.
   A word and its category are represented as XYZ>ABC in corpus
   where XYZ is the word and ABC its grammatical category.
*/

#include <stdio.h>
#include "declarations.c"          /* contains some constants */

#define noOfWords 10000          /* words to be scanned */
#define wordLength 60           /* max word length */

/* void show_example(int end, char scanned[], int cat[]) displays word- ending and category
of the scanned word. */

void show_example(int end, char scanned[], int cat[])
{
    int i, extended[noOfInputs] = {0};

    extend(end, scanned, extended);

    printf("in: ");
    for(i = 0; i < noOfInputs; i++)
        printf("%d ", extended[i]);

    printf(" out: ");
    for(i = 0; i < noOfDesiredOutputs; i++)
        printf("%d ", cat[i]);

    printf("\n");
}

/* void extend(int end, char scanned[], int extended[]) transforms the three last letters into a
representation suitable for the net. */

void extend(int end, char scanned[], int extended[])
{
    int pos;

    if ((scanned[end] >= 65) && (scanned[end] <= 93)) {
        pos = scanned[end] - 65 + 29 + 29 + 29;
        extended[pos] = 1;
    }

    if ((scanned[end - 1] >= 65) && (scanned[end - 1] <= 93)) {
        pos = scanned[end - 1] - 65 + 29 + 29;
        extended[pos] = 1;
    }

    if ((scanned[end - 2] >= 65) && (scanned[end - 2] <= 93)) {
        pos = scanned[end - 2] - 65 + 29;
        extended[pos] = 1;
    }
}

```

```

        if ((scaned[end - 3] >= 65) && (scaned[end - 3] <= 93)) {
            pos = scanned[end - 3] - 65;
            extended[pos] = 1;
        }
    }
}

/* void reset(int cat[noOfDesiredOutputs]) resets the present word category. */
void reset(int cat[noOfDesiredOutputs])
{
    int j;

    for(j = 0; j < noOfDesiredOutputs; j++)
        cat[j] = 0;
}

main()
{
    int state = 0, i = 0, word = 0, found = 0, c,
    cat[noOfDesiredOutputs] = {0};
    char scanned[wordLength] = {0};

    while ((c = getchar()) != EOF) && (found < noOfWords) {

        if (state == 0) {
            state = 1;
            scanned[0] = 32;
            scanned[1] = c;
            i = 2;
            reset(cat);
        }

        else if (state == 1 && c == '>')
            state = 2;
        else if (state == 1 && c != '>') {
            scanned[i] = c;
            i++;
        }

        else if (state == 2 && c == 'N')
            state = 10;
        else if (state == 2 && c == 'A')
            state = 20;
        else if (state == 2 && c == 'V')
            state = 30;
        else if (state == 2 && c != 'A' && c != 'N' && c != 'V')
            state = 0;
    }
}

```

```

else if (state == 10 && c == 'N') {
    cat[0] = 1; /* sets a flag for noun in target */
    state = 11;
}
else if (state == 10 && c != 'N')
    state = 0;

```

/\* if the character found is a space we have found a acceptable word-ending \*/

```

else if (state == 11 && c == ' ') {
    show_example(i-1, scanned, cat);
    reset(cat);
    state = 1;
    scanned[0] = c;
    i = 1;
    found++;
}

```

```

else if (state == 11 && c == 'D')
    state = 12;
else if (state == 11 && c == 'G')
    state = 13;
else if (state == 11 && c != ' ' && c != 'D' && c != 'G')
    state = 0;

```

```

else if (state == 12 && c == 'D') {
    cat[3] = 1; /* sets a flag for determinate article in target */
    state = 14;
}

```

```

else if (state == 12 && c != 'D')
    state = 0;

```

```

else if (state == 13 && c == 'G') {
    cat[4] = 1; /* sets a flag for genitive in target */
    state = 14;
}

```

```

else if (state == 13 && c != 'G')
    state = 0;

```

```

else if (state == 14 && c == ' ') {
    show_example(i-1, scanned, cat);
    reset(cat);
    state = 1;
    scanned[0] = c;
    i = 1;
    found++;
}

```

```

else if (state == 14 && c != ' ')
    state = 0;

```

```

else if (state == 20 && c == 'J') {
    cat[1] = 1; /* sets a flag for adjective in target */
}

```

```

        state = 21;
    }
    else if (state == 20 && c == 'N')
        cat[0] = 1; /* sets a flag for noun in target */
        state = 11;
    else if (state == 20 && c != 'J' && != 'N')
        state = 0;

    else if (state == 21 && c == ' ') {
        show_example(i-1, scanned, cat);
        reset(cat);
        state = 1;
        scanned[0] = c;
        i = 1;
        found++;
    }
    else if (state == 21 && c == 'K')
        state = 22;
    else if (state == 21 && c == 'S')
        state = 23;
    else if (state == 21 && c != ' ' && c != 'K' && c != 'S')
        state = 0;

    else if (state == 22 && c == 'P') {
        cat[5] = 1; /* sets a flag for comparative in target */
        state = 24;
    }
    else if (state == 22 && c != 'P')
        state = 0;

    else if (state == 23 && c == 'U') {
        cat[6] = 1; /* sets a flag for superlative in target */
        state = 24;
    }
    else if (state == 23 && c != 'U')
        state = 0;

    else if (state == 24 && c == ' ') {
        show_example(i-1, scanned, cat);
        reset(cat);
        state = 1;
        scanned[0] = c;
        i = 1;
        found++;
    }

    else if (state == 30 && c == 'V') {
        cat[2] = 1; /* sets a flag for verb in target */
        state = 31;
    }
    else if (state == 30 && c == 'N') {

```

```

        cat[0] = 1; /* sets a flag for noun in target */
        state = 11;
    }
    else if (state == 30 && c != 'V' && c != 'N')
        state = 0;

    else if (state == 31 && c == ' ') {
        show_example(i-1, scanned, cat);
        reset(cat);
        state = 1;
        scanned[0] = c;
        i = 1;
        found++;
    }
    else if (state == 31 && c == 'P')
        state = 32;
    else if (state == 31 && c == 'I')
        state = 33;
    else if (state == 31 && c == 'S')
        state = 34;
    else if (state == 31 && c != ' ' && c != 'P' && c != 'I' && c != 'S')
        state = 0;

    else if (state == 32 && c == 'S') {
        cat[7] = 1; /* sets a flag for present in target */
        state = 35;
    }
    else if (state == 32 && c == 'T') {
        cat[8] = 1; /* sets a flag for preteritum in target */
        state = 35;
    }
    else if (state == 32 && c != 'S' && c != 'T')
        state = 0;

    else if (state == 33 && c == 'P') {
        cat[9] = 1; /* sets a flag for imperative in target */
        state = 35;
    }
    else if (state == 33 && c == 'V') {
        cat[10] = 1; /* sets a flag for infinitive in target */
        state = 35;
    }
    else if (state == 33 && c != 'P' && c != 'V')
        state = 0;

    else if (state == 34 && c == 'N') {
        cat[11] = 1; /* sets a flag for supine in target */
        state = 35;
    }
    else if (state == 34 && c != 'N')
        state = 0;

```

```
else if (state == 35 && c == ' '){
    show=example(i-1, scanned, cat);
    reset(cat);
    state = 1;
    scanned[0] = c;
    i = 1;
    found++;
}
else if (state == 35 && c != ' '){
    state = 0;
}
}
```

```

/*
    Declarations for neural net
*/

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

#define weightsInHiddenLayer1      87
#define weightsInHiddenLayer2      20
#define weightsInOutputLayer       0

#define noOfInputs                  87
#define neuronsInHiddenLayer1      20
#define neuronsInHiddenLayer2      13
#define neuronsInOutputLayer        0

#define noOfLayers                  3
#define lastLayer                   noOfLayers - 1
#define largestNoOfNeurons          87
#define largestNoOfWeights          87

#define noOfDesiredOutputs          13

typedef double weightType[noOfLayers][largestNoOfNeurons]
                        [largestNoOfWeights];
typedef double outputType[noOfLayers][largestNoOfNeurons];
typedef double thresholdType[noOfLayers][largestNoOfNeurons];
typedef double desiredOutputType[noOfDesiredOutputs];
typedef double errorType[noOfLayers][largestNoOfWeights];

```

```

/*
    Primitives
*/

/* int neurons_per_layer(int layer) returns the number of neurons in the given layer */

int neurons_per_layer(int layer)
{
    switch(layer) {
        case 0:
            return(noOfInputs);
            break;
        case 1:
            return(neuronsInHiddenLayer1);
            break;
        case 2:
            return(neuronsInHiddenLayer2);
            break;
        case 3:
            return(neuronsInHiddenLayer3);
            break;
    }
}

/* int weights_per_neuron(int layer) returns the number of weights a neuron in the given
layer has */

int weights_per_neuron(int layer)
{
    if (layer == 0)
        return(0);
    else
        return(neurons_per_layer(layer - 1));
}

/* double nth_weight(int layer, int neuron, int n, weightType weightNet) returns the n:th
weight of the given neuron */

double nth_weight(int layer, int neuron, int n, weightType weightNet)
{
    return(weightNet[layer][neuron][n]);
}

/* double nth_input(int layer, int n, outputType outputNet) returns the n:th input of the given
layer */

double nth_input(int layer, int n, outputType outputNet)
{
    return(outputNet[layer-1][n]);
}

```

```
/* double nth_output(int layer, int neuron, outputType output) returns the output of the given neuron */
```

```
double nth_output(int layer, int neuron, outputType output)
{
    return(output[layer][neuron]);
}
```

```
/* double nth_target(int n, desiredOutputType target) returns the n:th desired output value */
```

```
double nth_target(int n, desiredOutputType target)
{
    return(target[n]);
}
```

```
/* double nth_error(int layer, int neuron, errorType errors) returns the error of the given neuron */
```

```
double nth_error(int layer, int neuron, errorType errors)
{
    return(errors[layer][neuron]);
}
```

```
/* double get_threshold(int layer, int neuron, thresholdType thresholdNet) returns the threshold of the given neuron */
```

```
double get_threshold(int layer, int neuron, thresholdType thresholdNet)
{
    return(thresholdNet[layer][neuron]);
}
```

```
/* void store_nth_weight(double value, int layer, int neuron, int n, weightType weightNet) changes the n:th weight of the given neuron to value */
```

```
void store_nth_weight(double value, int layer, int neuron, int n,
                      weightType weightNet)
{
    weightNet[layer][neuron][n] = value;
}
```

```
/* void store_nth_threshold(double value, int layer, int neuron, thresholdType thresholdNet) changes the threshold of the neuron to value */
```

```
void store_nth_threshold(double value, int layer, int neuron,
                        thresholdType thresholdNet)
{
    thresholdNet[layer][neuron] = value;
}
```

```

/* void store_output(double value, int layer, int neuron, outputType outputNet) sets the
output value of the given neuron to value */

void store_output(double value, int layer, int neuron,
                  outputType outputNet)
{
    outputNet[layer][neuron] = value;
}

/* void store_error(double value, int layer, int neuron, errorType errors) sets the error at
the given position to value */

void store_error(double value, int layer, int neuron, errorType errors)
{
    errors[layer][neuron] = value;
}

/* void set_thresholds(thresholdType thresholds, int value) sets all thresholds to value */

void set_thresholds(thresholdType thresholds, int value)
{
    int layer, neuron;

    for(layer = 0; layer < noOfLayers; layer++)
        for(neuron = 0; neuron < neurons_per_layer(layer); neuron++)
            thresholds[layer][neuron] = value;
}

/* void insert_inputs(double word[noOfInputs], outputType outputs) feed the last letters to
the network */

void insert_inputs(double word[noOfInputs], outputType outputs)
{
    int i;

    for(i = 0; i < noOfInputs; i++)
        store_output((double)word[i], 0, i, outputs);
}

/* void reset_outputs(outputType outputs) sets all outputs to 0 */

void reset_outputs(outputType outputs)
{
    int layer, neuron;

    for(layer = 0; layer <= lastLayer; layer++)
        for(neuron = 0; neuron < neurons_per_layer(layer); neuron++)
            store_output(0, layer, neuron, outputs);
}

```

```

/* int last_layer(int layer) is a predicate which is true if the given layer is the last one,
false otherwise */

int last_layer(int layer)
{
    return(layer == lastLayer);
}

/*
Some functions for showing answer, output, threshold, and weight values.
*/

/* void show_outputs(outputType outputs) displays the output of all neurons */

void show_outputs(outputType outputs)
{
    int layer , neuron;

    printf("outputs\n");
    for(layer = 0; layer <= lastLayer; layer++)
        for(neuron = 0; neuron < neurons_per_layer(layer); neuron++)
            printf("output (%d%d): %f ", layer, neuron,
                nth_output(layer, neuron, outputs));
    printf("\n\n");
}

/* void show_weights(weightType weights) displays the weights of all neurons */

void show_weights(weightType weights)
{
    int layer, neuron, i;

    printf("weights\n");
    for(layer = 0; layer <= lastLayer; layer++)
        for(neuron = 0; neuron < neurons_per_layer(layer); neuron++)
            for(i = 0; i < weights_per_neuron(layer); i++)
                printf("weight (%d%d%d): %f ", layer, neuron, i,
                    nth_weight(layer, neuron, i, weights));
    printf("\n\n");
}

/* void show_thresholds(thresholdType thresholds) displays the threshold of all neurons */

void show_thresholds(thresholdType thresholds)
{
    int layer , neuron;

    printf("thresholds\n");
    for(layer = 1; layer <= lastLayer; layer++)

```

```

        for(neuron = 0; neuron < neurons_per_layer(layer); neuron++)
            printf("threshold (%d%d): %f ", layer, neuron,
                get_threshold(layer, neuron, thresholds));
    printf("\n\n");
}

/* void show_result(outputType outputs) displays the answer the net is giving */
void show_result(outputType outputs)
{
    int neuron;

    for(neuron = 0; neuron < neurons_per_layer(lastLayer); neuron++)
        printf("\answer = %.20f ",
            nth_output(lastLayer, neuron, outputs));
    printf("\n\n");
}

```

```

/*
  Main functions for the forward phase.
*/

/* double total_incoming_value(int layer, int neuron, weightType weightNet, outputType
outputNet) returns the weighted sum of a given neuron */
double total_incoming_value(int layer, int neuron,
                           weightType weightNet,
                           outputType outputNet)
{
  int i;
  double sum = 0;

  for(i = 0; i < weights_per_neuron(layer); i++)
    sum += nth_weight(layer, neuron, i, weightNet) *
           nth_input(layer, i, outputNet);

  return(sum);
}

/* double activation(int layer, int neuron, weightType weightNet, outputType outputNet,
thresholdType thresholdNet) returns the activity of a given neuron */
double activation(int layer, int neuron, weightType weightNet,
                 outputType outputNet, thresholdType thresholdNet)
{
  return(total_incoming_value(layer, neuron, weightNet, outputNet)
         + get_threshold(layer, neuron, thresholdNet));
}

/* double output_function(int layer, int neuron, weightType weightNet, outputType
outputNet, thresholdType thresholdNet) returns the output of a given neuron */
double output_function(int layer, int neuron, weightType weightNet,
                      outputType outputNet, thresholdType thresholdNet)
{
  return(1 / (1 + exp(-activation(layer, neuron, weightNet, outputNet,
                                thresholdNet))));
}

/* void forward(weightType weightNet, outputType outputNet, thresholdType thresholdNet)
does one forward propagation throu the net, changing the output values of all neurons */
void forward(weightType weightNet, outputType outputNet,
            thresholdType thresholdNet)
{
  int layer, neuron;

  for(layer = 1; layer <= lastLayer; layer++)
    for(neuron = 0; neuron < neurons_per_layer(layer); neuron++)

```

```
store_output(output_function(layer, neuron, weightNet,  
                             outputNet, thresholdNet),  
             layer, neuron, outputNet);  
}
```

```

/*
  Main functions for the backward phase.
*/

/* double output_delta(int layer, int neuron, outputType outputs, desiredOutputType targets)
returns the error of the answer of the net */

double output_delta(int layer, int neuron, outputType outputs,
                    desiredOutputType targets)
{
  double out;

  out = nth_output(layer, neuron, outputs);
  return((nth_target(neuron, targets) - out) * out * (1 - out));
}

/* double hidden_delta(int layer, int neuron, outputType outputs, weightType weights,
errorType errors) returns the error of a neurons weight in a hidden layer */

double hidden_delta(int layer, int neuron, outputType outputs,
                    weightType weights, errorType errors)
{
  int i;
  double sum = 0, out;

  for(i = 0; i < neurons_per_layer(layer + 1); i++)
    sum += nth_error(layer + 1, i, errors) *
          nth_weight(layer + 1, i, neuron, weights);

  out = nth_output(layer, neuron, outputs);
  return(sum * out * (1 - out));
}

/* void adjust_thresholds(double beta, int layer, outputType outputs, weightType weights,
thresholdType thresholds, thresholdType newThresholds, desiredOutputType targets,
errorType errors) calculates the new threshold of a given neuron and places the result in
newThresholds */

void adjust_thresholds(double beta, int layer, outputType outputs,
                      weightType weights, thresholdType thresholds,
                      thresholdType newThresholds,
                      desiredOutputType targets, errorType errors)
{
  int neuron;
  double error;

  for(neurons = 0; neuron < neuron_per_layer(layer); neuron++) {
    store_error(error = hidden_delta(layer, neuron, outputs, weights,
                                     errors), layer, neuron, errors);
    store_nth_threshold(get_threshold(layer, neuron, thresholds) +
                       beta * error * 1, layer, neuron, newThresholds);
  }
}

```

```

}

/* void adjust_weights(double beta, int layer, int neuron, outputType outputs, weightType
weights, weightType newWeights, desiredOutputType targets, errorType errors) calculates
the error and the new weights of a given neuron and places the results in newWeights and
errors */

void adjust_weights(double beta, int layer, int neuron,
                   outputType outputs, weightType weights,
                   weightType newWeights, desiredOutputType targets,
                   errorType errors)
{
    int weight;

    for(weight = 0; weight < weights_per_neuron(layer); weight++)
        store_nth_weight(nth_weight(layer, neuron, weight, weights) +
                        beta * nth_error(layer, neuron, errors) *
                        nth_output(layer - 1, neuron, outputs), layer,
                        neuron, weight, newWeights);
}

/* void adjust_output_weights(double beta, int neuron, outputType outputs, weightType
weights, weightType newWeights, desiredOutputType targets, errorType errors) calculates
the error and new weights of a given neuron in the output layer and places the results in
newWeights and errors */

void adjust_output_weights(double beta, int neuron, outputType outputs,
                          weightType weights, weightType newWeights,
                          desiredOutputType targets,
                          errorType errors)
{
    int weight;
    double error;

    for(weight = 0; weight < weights_per_neuron(lastLayer); weight++) {
        store_error(error = output_delta(lastLayer, neuron, outputs,
                                         targets), lastLayer, neuron, errors);
        store_nth_weight(nth_weight(lastLayer, neuron, weight, weights) +
                        beta * error * nth_output(lastLayer - 1, weight,
                                                  outputs), lastLayer, neuron, weight, newWeights);
    }
}

/* void adjust_output_thresholds(double beta, outputType outputs, thresholdType thresholds,
thresholdType newThresholds, errorType errors) calculates the new threshold of a neuron in
the output layer */

void adjust_output_thresholds(double beta, outputType outputs,
                             thresholdType thresholds,
                             thresholdType newThresholds,
                             errorType errors)
{

```

```

int neuron;

for(neuron = 0; neuron < neurons_per_layer(lastLayer); neuron++)
    store_nth_threshold(get_threshold(lastLayer, neuron, thresholds)
        + beta * nth_error(lastLayer, neuron, errors) *
        1, lastLayer, neuron, newThresholds);
}

/* void back(double beta, outputType outputs, weightType weights, weightType newWeights,
thresholdType thresholds, thresholdType newThresholds, desiredOutputType targets)
calculates new weights and threshold for all neurons and places the result in newWeights and
newThresholds */

void back(double beta, outputType outputs, weightType weights,
    weightType newWeights, thresholdType thresholds,
    thresholdType newThresholds, desiredOutputType targets)
{
    int layer, neuron;
    errorType errors = {0};

    for(neuron = 0; neuron < neurons_per_layer(lastLayer); neuron++) {
        adjust_output_weights(beta, neuron, outputs, weights, newWeights,
            targets, errors);
        adjust_output_thresholds(beta, outputs, thresholds, newThresholds,
            errors);
    }

    for(layer = lastLayer - 1; layer > 0; layer --)
        for(neuron = 0; neuron < neurons_per_layer(layer); neuron++) {
            adjust_thresholds(beta, layer, outputs, weights, thresholds,
                newThresholds, targets, errors);
            adjust_weights(beta, layer, neuron, outputs, weights,
                newWeights, targets, errors);
        }
}

/* void backward(double beta, outputType outputs, weightType weights, thresholdType
thresholds, desiredOutputType targets) calculates the new weights and thresholds of the net
*/

void backward(double beta, outputType outputs, weightType weights,
    thresholdType thresholds, desiredOutputType targets)
{
    int layer, neuron, weight;
    weightType newWeights = {0};
    thresholdType newThresholds = {0};

    back(beta, outputs, weights, newWeights, thresholds, newThresholds,
        targets);

    for(layer = 0; layer < noOfLayers; layer++)
        for(neuron = 0; neuron < neurons_per_layer(layer); neuron++) {
            store_nth_threshold(get_threshold(layer, neuron

```

```
        newThresholds), layer, neuron, thresholds);  
    for(weight = 0; weight < weights_per_neuron(layer); weight++)  
        store_nth_weight(nth_weight(layer, neuron, weight,  
            newWeights), layer, neuron, weight, weights);  
    }  
}
```

```

/*
  This program trains the network.

  gcc -o train.out train.c -lm -O and run with
  train.out training_data_file output_file

  The program saves all coefficients of the network in the
  output_file.

  The training_data_file is created by the DFA.
*/

#include <stdio.h>
#include <stdlib.h>
#include "declarations.c"
#include "primitives.c"
#include "forward.c"
#include "backward.c"

#define noOfWords2000

typedef int wordType[noOfWords][noOfInputs];
typedef int categoryType[noOfWords][noOfDesiredOutputs];

/* void store_target(double value, int pos, desiredOutputType targets) sets the target value
at pos to value */

void store_target(double value, int pos, desiredOutputType targets)
{
    targets[pos] = value;
}

/* void insert_targets(int cat[noOfDesiredOutputs], desiredOutputType targets) inserts all
target values */

void insert_targets(int cat[noOfDesiredOutputs],
                   desiredOutputType targets)
{
    int i;

    for(i = 0; i < noOfDesiredOutputs; i++)
        store_target((double)cat[i], i, targets);
}

/* void init_weights(weightType weights) initializes the weights of the network to random
numbers */

void init_weights(weightType weights)
{
    int layer, neuron, weight;

    for(layer = 0; layer <= lastLayer; layer++)

```

```

        for(neuron = 0; neuron < neurons_per_layer(layer); neuron++)
            for(weight = 0; weight < weights_per_neuron(layer); weight++)
                store_nth_weight((1 / (1 + (double)(rand() % 10))), layer,
                                neuron, weight, weights);
    }

/* void show_weights_for_file(FILE *file, weightType weights) displays the weights of all
neurons so they can be saved for the testing phase */

void show_weights_for_file(FILE *file, weightType weights)
{
    int layer, neuron, weight;

    for(layer = 0; layer <= lastLayer; layer++)
        for(neuron = 0; neuron < neurons_per_layer(layer); neuron++)
            for(weight = 0; weight < weights_per_neuron(layer); weight++)
                fprintf(file, "w(%d %d %d): %.20f ", layer, neuron, weight,
                        nth_weight(layer, neuron, weight, weights));
    fprintf(file, "\n\n");
}

/* void show_thresholds_for_file(FILE *file, thresholdType thresholds) displays the
threshold of all neurons so they can be saved for the testing phase */

void show_thresholds_for_file(FILE *file, thresholdType thresholds)
{
    int layer, neuron;

    for(layer = 1; layer <= lastLayer; layer++)
        for(neuron = 0; neuron < neurons_per_layer(layer); neuron++)
            fprintf(file, "t(%d %d): %.20f ", layer, neuron,
                    get_threshold(layer, neuron, thresholds));
    fprintf(file, "\n\n");
}

/* int examples(char *file, wordType words, categoryType cats) gets all examples from the
file named file */

int examples(char *file, wordType words, categoryType cats)
{
    int i, sum = 0;
    FILE *ifp = fopen(file, "r");

    while (fscanf(ifp, "in: %d", &words[sum][0]) == 1) {
        for(i = 1; i < noOfInputs; i++)
            fscanf(ifp, "%d", &words[sum][i]);
        fscanf(ifp, " out: %d", &cats[sum][0]);
        for(i = 1; i < noOfDesiredOutputs; i++)
            fscanf(ifp, "%d", &cats[sum][i]);
        fscanf(ifp, "\n");
        sum++;
    }
}

```

```

    fclose(ifp);
    return(sum);
}

void main(int argc, char *argv[])
{
    int sum = 0, i, j, cycle;
    double beta = 0.9;
    FILE *ifp, *ofp;
    wordType words = {0};
    categoryType cats = {0};
    weightType weights = {0};
    outputType outputs = {0};
    thresholdType thresholds = {0};
    desiredOutputType targets = {0};

    ofp = fopen(argv[2], "w");

    init_weights(weights);
    set_thresholds(thresholds, 1);

    sum = examples(argv[1], words, cats);

    printf("\n%d examples have been read\n cycles: ", sum);
    scanf("%d", &cycle);
    printf("\n");

    for(j = 0; j < cycle; j++)
        for(i = 0; i < sum; i++) {
            reset_outputs(outputs);
            insert_inputs(words[i], outputs);
            insert_targets(cats[i], targets);

            forward(weights, outputs, thresholds);
            backward(beta, outputs, weights, thresholds, targets);
        }

    show_weights_for_file(ofp, weights);
    show_thresholds_for_file(ofp, thresholds);

    fclose(ofp);
}

```

```

/*
  This program initializes a neural network with data from a file
  and processes a number of words to be categorized.

  gcc -o examine.out examine.c -lm -O and run with
  examine.out net_state_file unknown_words

  where net_state_file is created by the training program.

  The unknown words should be in the following format:

  in: X  where X is a 87 digit binary number.
*/

#include <stdio.h>
#include <stdlib.h>
#include "declarations.c"
#include "primitives.c"
#include "forward.c"
#include "backward.c"

typedef int unknownType[500][noOfInputs];

/* void setup_network(char *filename, weightType weights, thresholdType thresholds)
  opens the file pointed to by fileName and sets the weights and thresholds of the net according
  to the values specified in the file */
void setup_network(char *filename, weightType weights,
                  thresholdType thresholds)
{
  int layer, neuron, weight;
  double value;
  FILE *ifp;

  ifp = fopen(fileName, "r");

  while (fscanf(ifp, "w(%d %d %d): %lf ", &layer, &neuron, &weight,
               &value) == 4)
    store_nth_weight(value, layer, neuron, weight, weights);

  while (fscanf(ifp, "t(%d %d): %lf ", &layer, &neuron, &value) == 3)
    store_nth_threshold(value, layer, neuron, thresholds);

  fclose(ifp);
}

/* int get_unknows(char *fileName, unknownType word) reads the file pointed to by
  fileName. The words (or rather word-endings) found are stored in words. The function
  returns the number of word-endings read. */
int get_unknows(char *fileName, unknownType word)

```

```

{
    int sum = 0, i;
    FILE *ifp = fopen(fileName, "r");

    while(fscanf(ifp, "in: %d", &word[sum][0]) == 1) {
        for(i = 1; i < noOfInputs; i++)
            fscanf(ifp, " %d", &words[sum][i]);
        fscanf(ifp, "\n");
        sum++;
    }

    fclose(ifp);
    return(sum);
}

```

```

void main(int argc, char *argv[])
{
    int nr, i, j;
    unknownType words = {0};
    weightType weights = {0};
    outputType outputs = {0};
    thresholdType thresholds = {0};

    setup_network(argv[1], weights, thresholds);
    nr = get_unknowns(argv[2], words);

    for(i = 0; i < nr; i++) {
        reset_outputs(outputs);
        insert_inputs(words[i], outputs);
        for(j = 0; j < noOfInputs; j++)
            printf("%d ", words[i][j]);
        forward(weights, outputs, thresholds);
        show_result(outputs);
    }
}

```