Health Information Science

University of Victoria

HINF680
Directed Study

Summer Term 2001
Instructor: J. R. Moehr

Decision in Feed Forward Artificial Neural Networks (ANN)

Prepared by

Stefan Valerian Pantazi

Table of Contents

Introduction 4

1. The Basics of Feed-Forward ANNs 4

2. The Simple Perceptron Model 5

A Simple Perceptron Learning the OR Logic Function 9

A Simple Perceptron Learning the AND Logic Function 11

A Simple Perceptron Trying to Learn the XOR Logic Function 14

3. The Multi-layer Perceptron Model (MLP) 16

MLP Learning the XOR Logic Function 17

4. Algorithm for Transforming Nonlinearly Separable Classes in Linearly Separable Classes 19

Conclusions 20

References 20

List of Tables and Figures

Table 1 The possible bipolar input vectors for an ANN made of one neuron with two input connections
Table 2 The truth tables of the **OR** logic function using binary vectors (left) and bipolar vectors (right)
Table 3 The truth tables of the **AND** logic function using binary vectors (left) and bipolar vectors (right)
Table 4 The verification of the algorithm correctness for the perceptron which has learned the **AND** logic function
Table 5 The truth tables of the **XOR** logic function using binary vectors (left) and bipolar vectors (right)
Table 6 The re-coding process of the 4 input vectors performed in the MLP hidden layer
Table 7 The classification process of the three vectors resulted from re-coding and performed in the output layer
Table 8 The re-coding of the 4 input vectors by adding a third common component (x_3) to the vectors in similar classes

Figure 1 The graphic representation of the 2-dimensional input space using bipolar vectors
Figure 2 The graphic representation of the **OR** logic function
Figure 3 The graphic representation of the **AND** logic function
Figure 4 The network state before learning the **OR** logic function
Figure 5 The graphic representation of the network state before learning the **OR** logic function
Figure 6 The graphic representation of the network state evolution when learning the **OR** logic function
Figure 7 The final state of the network when learning the **OR** logic function
Figure 8 The graphic representation of the network final state when learning the **OR** logic function
Figure 9 The network state before learning the **AND** logic function
Figure 10 The graphic representation of the network state before learning the **AND** logic function
Figure 11 The graphic representation of the network state evolution when learning the **AND** logic function
Figure 12 The final state of the network when learning the **AND** logic function
Figure 13 The graphic representation of the network final state when learning the **AND** logic function
Figure 14 The graphic representation of the network state before learning the **AND** logic function
Figure 15 The graphic representation of the network state evolution when learning the **AND** logic function using a non-redundant training set of vectors
Figure 16 The graphic representation of the network final state when learning the **AND** logic function using a non-redundant training set of vectors
Figure 17 The graphic representation of the **XOR** logic functions' decision areas
Figure 18 The graphic representation of the network state before trying to learn the **XOR** logic function
Figure 19 The graphic representation of the network state evolution when trying to learn the **XOR** logic function
Figure 20 The topology of the MLP
Figure 21 The graphic representation of the two non-linearly separable classes defined by the logic **XOR** function
Figure 22 The final state of the multi-layer network after learning the **XOR** logic function
Figure 23 The graphic representations of the decision zones determined by the hidden layer neurons
Figure 24 The graphic representation of the decision zones determined by the output neuron
Figure 25 Graphic representation of the 4 vectors modified by adding a common component (x_3)

Executive Summary

In the introduction of the paper a very short history of the artificial neural network (ANN) research evolution is presented.

In the first chapter the reader is accustomed with the basics of the feed-forward ANNs. Next, the simple perceptron model is introduced presenting the theoretical background and advancing towards more complex issues, using as learning examples the Boolean logic functions like **OR**, **AND** and **XOR**. The end of the chapter makes the transition from the simple perceptron to the multi-layer perceptron (MLP), which is introduced as a ANN model which can learn more complex functions, unlike its predecessor.

Finally a learning algorithm that would enable the simple perceptron model to distinguish between non-linearly separable classes is proposed.

Introduction

First connectionist models date to the ’40s. Although the logical-symbolic approaches dominated the ’60s and the ’70s, they succeeded to model the intelligent behavior of human beings only to limited extent. The connectionist revival of the mid ’80s and ‘90s proved that artificial neural networks are models that can add decision flexibility, robustness (error tolerance) and self-learning capabilities to an artificial intelligence system.

Today, the latest research reports indicate that there is a great interest for hybrid architectures, which combine connectionist models with the logical-symbolic approaches.

1. The Basics of Feed-Forward ANNs

Artificial neural networks are *universal approximators* that can learn any input-output relation. Learning in feed-forward ANN could be seen as the approximation of a function F, defined in the m-dimensional Euclidean space and having values in an n-dimensional Euclidean space.

$$F : R^m \rightarrow R^n$$

A feed-forward ANN is made of simple, interconnected units (neurons). A basic unit (neuron) j has usually the following features:

- Status variable s_j, which stands for the activation of the neuron
- Connections with other units i from previous layers; these connections are characterized by a weight w_{ij} defined usually as a real number
- Threshold value (t_j)
- Activation function g_j, which usually represents the weighted sum of the neuron’s input vector components
- Transfer function f_j which let us calculate the output of the neuron (the resulting signal) and which may be a step function, a sigmoid function or even the identical function

Between all these parameters, the following relations exist:

$$g_j = \sum_{i=1}^{n} w_{ij} s_i$$

$$s_j = f_j(g_j(S_i) - t_j) = f_j(\sum_{i=1}^{n} w_{ij} s_i - t_j)$$

For example, considering as a transfer function the sign function:

$$f(x) : R \rightarrow \{-1, 1\}$$

$$f(x) = \begin{cases} -1 & if \quad x < 0 \\ 1 & if \quad x \geq 0 \end{cases}$$

For a totally connected ANN the formulas above could be interpreted as having the following meaning:

- the activation state of a neuron s_j is -1 (non-activated) if the sign of the difference between the weighted sum of the activations of the previous layer neurons and the neuron's threshold t_j is negative
- the activation state of a neuron s_j is 1 (active) if the sign of the difference between the weighted sum of the activations of the previous layer neurons and the neuron's threshold t_j is positive; in other words, neuron i activates only if the weighted sum of its inputs is greater or equal to its threshold value.

The resemblance of this mathematical model to biological models, in which the neural activation depends as well on the value of a threshold and on the activating or inhibiting synapses realized with other neurons, is now obvious.

When describing an ANN model, besides the characteristics of the units, it is necessary to specify the topology (architecture) of the neural network. Therefore there are ANN with a single layer of neurons (e.g. the simple perceptron) or with more than one layer (e.g. multi-layer perceptron).

Moreover, in order to be capable of learning, one must define a strategy of modifying the ANN parameters (e.g. connections weights and neuron threshold values). This strategy it is called the *learning method* (rule) and because most of the feed-forward ANN' learning algorithms are iterative, these learning rules are usually written as recursive formulas similar to this one:

$$w'_{ij} = w_{ij} + \Delta w_{ij}$$

where w_{ij} and w'_{ij} represents the connection weights between the neurons i and j before and respectively after the adjustment. Δw_{ij} represents the difference between the two weight values of the same connection before and respectively after the adjustment.

2. The Simple Perceptron Model

The single layer perceptron is the simplest model of feed-forward ANN. It may be used for solving simple problems of classification into linearly separable classes. The objects to be classified are defined as m-dimensional vectors (X) whose components (x_1, x_2, ...x_m) have usually one of the discrete values -1 and 1. These vectors are called *bipolar vectors*, unlike *binary vectors* whose components may take one of the values 0 or 1. The input vector space is a multidimensional (m dimensional) space, which contains the vectors to be classified. The output vector space is an n dimensional space, which contains the classes in which the input vectors will be classified.

If, given the input set of vectors, the n classes are linearly separable; the input vector space can be divided into n distinct regions (decision regions) by n hyperplanes whose equations have the following expression:

$$W^T X - T = 0$$

where W^T represents the (n,m) transposed matrix which contains the neurons' connections weights, X is the (m,1) type matrix of the input vectors and T is the (n,1) type matrix of the neuronal threshold values. So, W, and T matrices represent a possible solution of the given classification problem.

To give an example, for a 2-dimensional input space (m=2) and a network made of one neuron, the possible bipolar input vectors are:

	X_1	X_2	X_3	X_4
x_1	-1	-1	1	1
x_2	-1	1	-1	1

Table 1 The possible bipolar input vectors for an ANN made of one neuron with two input connections

Because the network has only one neuron (the output space is uni-dimensional, n=1), the classification problems that it can solve will have only two classes. This limitation is due to the extremely simple topology of the network. Therefore, there will be only one hyperplane (a line actually) which will divide the input space into two areas, each of them corresponding to one of the two classes. The hyperplane (line) equation will be a first-degree equation like of this form,

$$w_1 x_1 + w_2 x_2 - t = 0$$

where w_1, w_2 are the connections' weights and t the neuron threshold.

If we would make a graphic representation of the four possible input vectors (i.e., input space} we may observe that there are seven possibilities of dividing that space into two regions. For example, if the classification criterion imposes that X_1 belongs to one class and the rest of the vectors (i.e., X_2, X_3, X_4} to the other, there is at least one possible line which would correctly divide the space according to this criterion. In this case we say that the two classes are linearly separable.

However, among the seven possible classifications of the four input vectors, there is one that has proved itself to be special, in the way that is not possible to find a line that could divide the input space correctly. This particular case involves nonlinearly separable classes and will be discussed in the next chapter.

If we substitute the bipolar vectors with their binary correspondents (components having values equal to -1 will be set to 0) the decision criteria will become the well-known Boolean logic functions. However, using bipolar vectors is convenient because the perceptron algorithm functionality, which will be discussed later, is based on this type of vectors.

Therefore, because binary logic functions are using zeros and ones only, in order to simulate the learning of these functions by simple perceptrons, we need to convert binary vectors into bipolar vectors. We do this by replacing 0s with –1s. As we can see in the graphic representation, the spatial relations between the vectors are maintained after the transformation, only that the 2-dimensional input space is zero-centered.

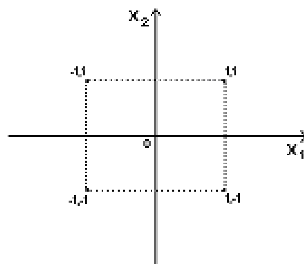

Figure 1 The graphic representation of the 2-dimensional input space using bipolar vectors

For the **OR** logic function the graphic shows very clear the linear divisibility of the classes.

x_1	x_2	x_1 **OR** x_2		x_1	x_2	x_1 **OR** x_2
0	0	0		-1	-1	-1
0	1	1		-1	1	1
1	0	1		1	-1	1
1	1	1		1	1	1

Table 2 The truth tables of the **OR** logic function using binary vectors (left) and bipolar vectors (right)

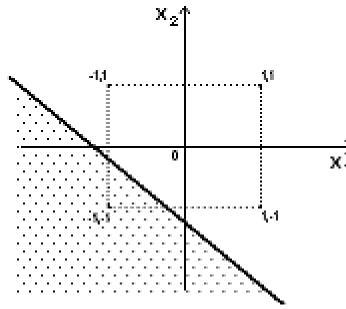

Figure 2 The graphic representation of the **OR** logic function

The dotted zone corresponds to the area where the function is 0 and consequently, to the non-activated state (-1) of the neuron that “learned” the OR function.

In the same manner, the **AND** logic function could be "learned" by a perceptron with a single neuron:

x_1	x_2	x_1 **AND** x_2		x_1	x_2	x_1 **AND** x_2
0	0	0		-1	-1	-1
0	1	0		-1	1	-1
1	0	0		1	-1	-1
1	1	1		1	1	1

Table 3 The truth tables of the **AND** logic function using binary vectors (left) and bipolar vectors (right)

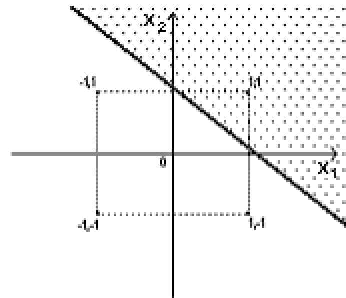

Figure 3 The graphic representation of the **AND** logic function

In Figure 3, the dotted line corresponds to the zone where the **AND** function value is 1 and therefore to the active state (1) of the neuron.

In order to classify the input vectors according to the chosen criterion (function), the perceptron must be trained. The simple perceptron training algorithm is called the perceptron algorithm and consists of an iterative procedure that modifies neurons connection weights and thresholds. Shortly, the algorithm steps are:

1. Initializing connection weights and thresholds with random values; this step is very important because it will have a very serious impact on the generalization capabilities of the network, but it is not an indispensable step; its importance will be discussed separately
2. Presenting an input vector together with its correct classification (output vector); this is why the perceptron algorithm is included in the broader class of supervised learning algorithms
3. The third step has two branches:
    a. If the vector presented in step 2 is correctly classified by the network we may proceed to the next vector by going to the step 2
    b. If the presented vector is incorrectly classified we will modify the connections weights and neuronal thresholds using the following relations and after this we will continue with step 2

$$w'_{ij} = w_{ij} - x_i s_j C, \quad i = 1 \ldots m; \quad j = 1 \ldots n$$

$$t'_j = t_j + s_j C \quad j = 1 \ldots n$$

where:

- m represents the input space dimension
- n represents the output space dimension (i.e., the number of the neurons in the output layer)
- w_{ij}, w'_{ij} are the weights of the connection between the output neuron j and the input neuron i, before and after the adjustment; the input neurons which are m in number, do not actually perform any calculations, their only role is to transmit the m components of the input vector to the next layer which is made of n functional neurons; therefore the simple perceptron has actually two layers of neurons: the input neurons (i.e., neurons that receive the input vector components) and output neurons (i.e., the only functional layer)
- x_i represent the component i of the input vector and will be stored into the i neuron of the input layer
- s_j represents the j neuron' activation
- C is the learning constant
- t_j and t_j' are the threshold values of the j output neuron before and after the adjustment

From the first relation we see that **if the network response is wrong**, the adjustment direction of the weight w_{ij} value (i.e., increase or decrease) depends of the component x_i and the activation state (s_j) of the neuron j like this:

- if s_j<0 (i.e., the inputs' weighted sum does not exceed the t_j threshold value so the neuron is not activated) and x_j>0 (i.e., the input is an "activating" one), Δw_{ij} (i.e., the adjusting value) will be positive (i.e., the weight value will be increased) and if x_i<0 (i.e., the input is an "inhibiting" one), Δw_{ij} will be negative (i.e., the weight value will be decreased); so in other words, when the neuron's weighted sum of the inputs is not sufficient to "beat" the threshold value, the weights of the activating inputs will be increased and those of the inhibiting inputs will be decreased
- if s_j>0 (i.e., the inputs' weighted sum of exceeds the t_j threshold value so the neuron is activated) and x_j>0 (i.e., the input is an "activating" one) Δw_{ij} (i.e., the adjusting value) will be negative (i.e., the weight value will be decreased) and if x_i<0 (i.e., the input is an "inhibiting" one) Δw_{ij} will be positive (i.e., the weight value will be increased); so in other words, when the neuron's weighted sum of the inputs "beats" the threshold value, the weights of the activating inputs will be decreased and those of the inhibiting inputs will be increased

From the second relation we observe that **if the network response is wrong,** the adjustment direction of the neuron's threshold value (i.e., increase or decrease) depends only on the activation state (s_j) of the neuron j like this:

- if s_j<0 (i.e., the inputs' weighted sum does not exceed the t_j threshold value so the neuron is not activated), Δt_j (i.e., the adjusting value) will be negative (i.e., the threshold value will be decreased); so in other words, when the neuron weighted sum of the inputs does not "beat" the threshold value, the neurons' threshold value will be decreased
- if s_j>0 (i.e., the inputs' weighted sum exceeds the t_j threshold value so the neuron is activated), Δt_j will be positive (i.e., the threshold value will be increased); so in other words, when the neuron weighted sum of the inputs "beats" the threshold value, the neurons' threshold value will be increased

By this "natural" way of adjusting the weights and threshold of the neurons, after a variable number of iterations, the network will have the right responses for the input vectors, providing the classes in which they belong are linearly separable.

This procedure continues until all the input vectors are correctly classified or the maximum number of iterations is reached. By representing graphically these iterations, we can see very clearly that the weights and thresholds adjustment direction is done in a way that allows the network to correctly classify as many input vectors as possible. For linearly separable classes it has been proven that the algorithm converges (i.e., a solution is found) in a reasonable number of iterations and that the number depends only by the learning constant value and the initial networks parameters (i.e., weights and thresholds values).

A Simple Perceptron Learning the OR Logic Function

The perceptron has a random initial state:

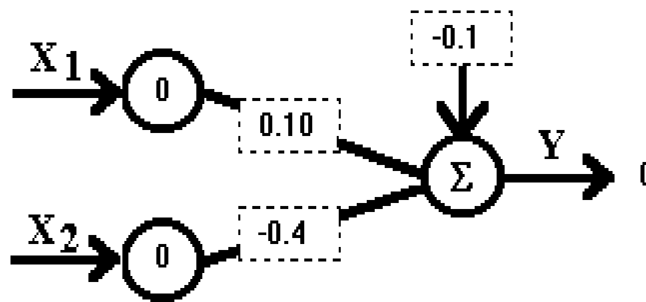

Figure 4 The network state before learning the **OR** logic function; there are three parameters: two weights values (0.10 and –0.4) and a threshold value (-0.1)

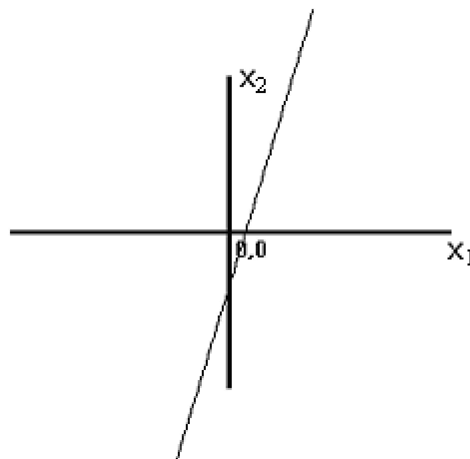

Figure 5 The graphic representation of the network state before learning the **OR** logic function; the initial values of the neuron weights and threshold are the parameters of a first degree equation which is far from any possible solution of the classification problem

After setting the random state, the learning algorithm starts to "move" the system through the solutions space until a solution is found.

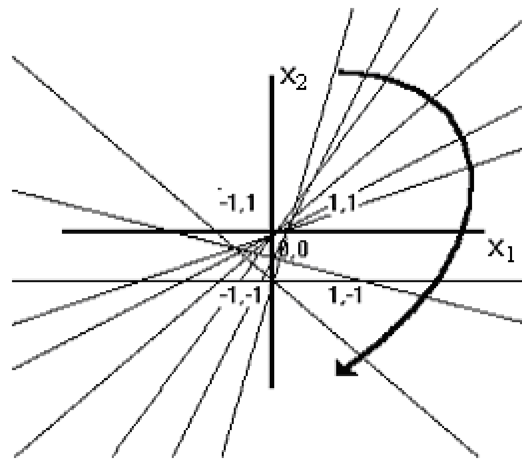

Figure 6 The graphic representation of the network state evolution when learning the **OR** logic function; the direction of evolution is shown by the thick arrow

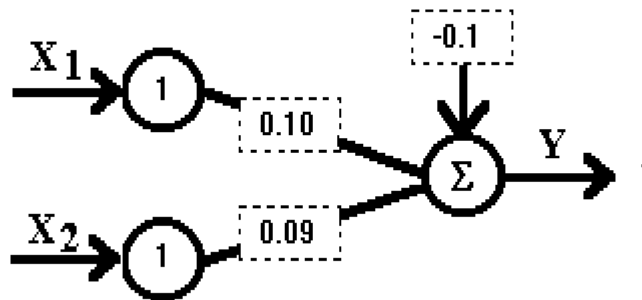

Figure 7 The final state of the network when learning the **OR** logic function

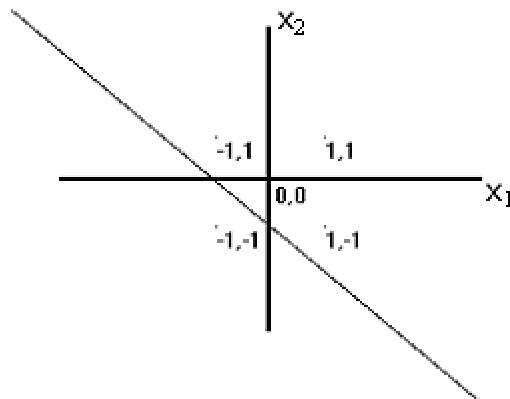

Figure 8 The graphic representation of the network final state when learning the **OR** logic function; the weights and threshold values at the end of training process are a possible solution to the this particular classification problem

A Simple Perceptron Learning the AND Logic Function

For the **AND** logic function the initial random values will be different.

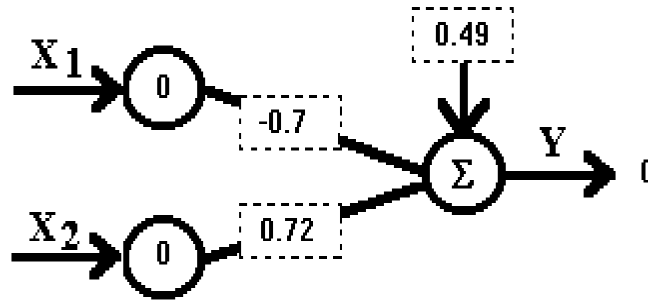

Figure 9 The network state before learning the **AND** logic function

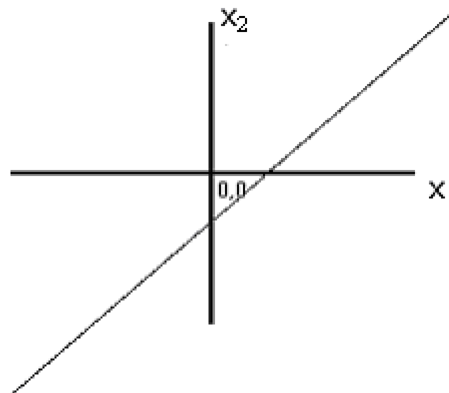

Figure 10 The graphic representation of the network state before learning the **AND** logic function; the initial values of the neuron weights and threshold are far from any possible solution of the problem

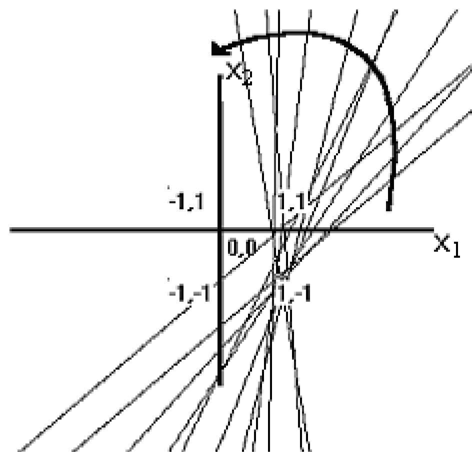

Figure 11 The graphic representation of the network state evolution when learning the **AND** logic function

In this case, the evolution of the learning process has different intermediate states. Looking at the graphic representation we can easily conclude that the number of iterations is strongly influenced by the initial state random values. Sometimes it is even possible that the solution to the classification problem coincides with the initial state.

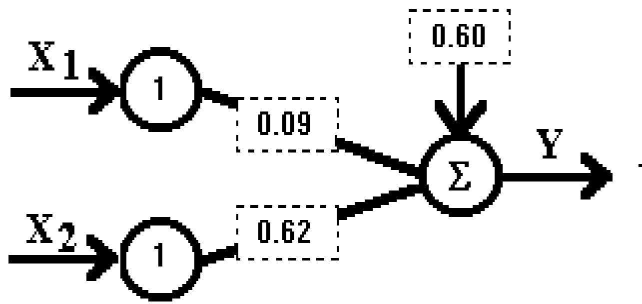

Figure 12 The final state of the network when learning the **AND** logic function

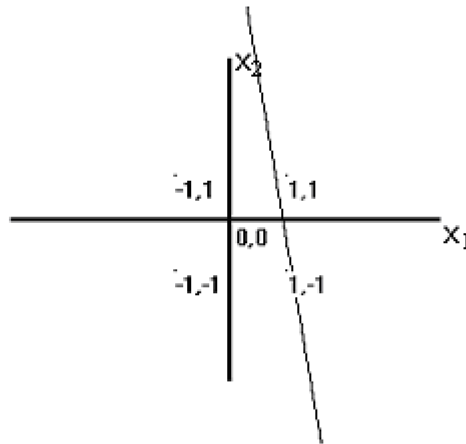

Figure 13 The graphic representation of the network final state when learning the **AND** logic function

If we try to verify the correctness of the algorithm we will have to make calculations ourselves.

x_1	x_2	WSum=w_1x_1+w_2x_2	Out=F(WSum-t)=Sgn(WSum-t)
-1	-1	WSum=-0.09-0.62=-0.71	Out=Sgn(-0.71-0.6)=Sgn(-1.31)=-1
-1	1	WSum=-0.09+0.62=0.53	Out=Sgn(0.53-0.6)=Sgn(-0.07)=-1
1	-1	WSum=0.09-0.62=-0.53	Out=Sgn(-0.53-0.6)=Sgn(-1.13)=-1
1	1	WSum=0.09+0.62=0.71	Out=Sgn(0.71-0.6)=Sgn(0.11)=1

Table 4 The verification of the algorithm correctness for the perceptron which has learned the **AND** logic function; the network parameters are: w_1=0.09, w_2=0.62, t=0.60

The results in the table confirm that after the training process our neuron is capable to classify correctly the four input vectors, or in other words, to reproduce the **AND** logic function response.

If we analyze further the final state of the network, we will observe that the decision line is very near to two of the training vectors (X_3 and X_4). It is appropriate to say that, in this case, the final state corresponds to a solution which is "barely acceptable", unlike the solution obtained earlier for the **OR** logic function, when the decision zones boundary was almost optimal, i.e., evenly distanced from the vectors close to it.

This happens because the neuronal functionality is based on a non-continuous transfer function (i.e., step function) and therefore the system does not have any possibility to evolve towards a "better" solution. To put it in another way, the perceptron algorithm stops immediately a solution is reached, no matter whether the solution is optimal or not. This limitation can be overcome by using continuous activation functions like, for example, sigmoid functions.

Another important aspect refers to the redundancy of the two training vector sets. Indeed, we may see that the vector X_1 from the **AND** logic function training set and the vector X_4 from the **OR** function training set are situated at the longest distances from the decision boundaries. From this very fact we deduce that the two logic functions could be learned by the perceptron even if these particular vectors will be discarded from the training sets, proving that there is a certain degree of redundancy present in the training sets. Moreover, after the redundancy elimination, the fact that the network will have a correct response for these vectors, which no longer belong to the training sets, proves its capacity of generalization.

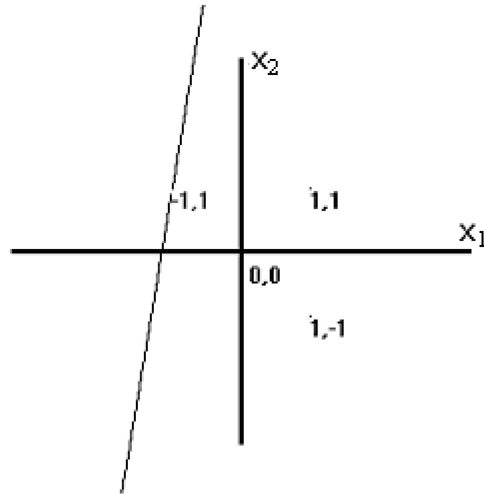

Figure 14 The graphic representation of the network state before learning the **AND** logic function

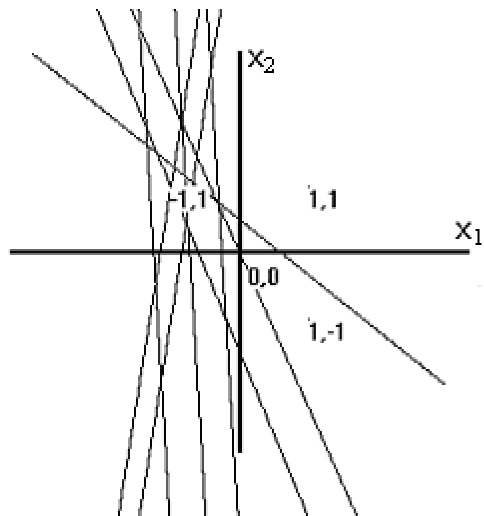

Figure 15 The graphic representation of the network state evolution when learning the **AND** logic function using a non-redundant training set of vectors

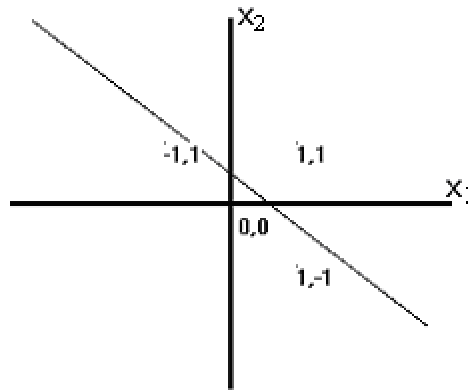

Figure 16 The graphic representation of the network final state when learning the **AND** logic function using a non-redundant training set of vectors

Concluding, we may say as a general principle that when choosing the ANN training vector sets, the most important vectors are those situated close to the decision boundaries and which are making the transition from one decision class to another. The Hamming distance (which is given by the number of bits with opposite values) between these vectors that belong to neighboring classes are therefore minimal. However it is these vectors that assure the correct class discrimination and that provide the perceptron with a powerful generalization capacity.

As we have seen before, the perceptron algorithm iterates until all the vectors in the training set are correctly classified or until a maximum number of iteration has been reached. The maximum number of iterations represents a supplementary condition to stop the learning process because, sometimes, when dealing with high dimensional vectors, the prior determination of linear divisibility of classes may be a difficult task. When trying to train a simple perceptron to classify into non-linearly separable classes the algorithm continues to iterate indefinitely. Hence the necessity to introduce a supplementary stopping condition.

A Simple Perceptron Trying to Learn the XOR Logic Function

The simplest function that involves classification into non-linearly separable classes is the **XOR** logic function, whose truth table is given in Table 5:

x_1	x_2	x_1 **XOR** x_2		x_1	x_2	x_1 **XOR** x_2
0	0	0		-1	-1	-1
0	1	1		-1	1	1
1	0	1		1	-1	1
1	1	0		1	1	-1

Table 5 The truth tables of the **XOR** logic function using binary vectors (left) and bipolar vectors (right)

Representing this function graphically, we easily observe that it is impossible to separate the input space with only one decision line, providing the vectors are to be correctly classified. This is because one of the decision areas - the dotted one - is more complex than those we have seen until now; it actually encloses the other decision region.

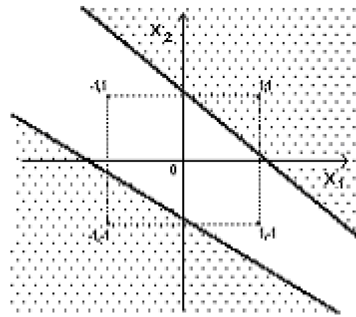

Figure 17 The graphic representation of the **XOR** logic functions' decision areas

If we still want to perform the classification, we should use two lines for separating the two decision areas. This cannot be accomplished using a simple perceptron (i.e., having only one layer) but a network with a more complicated topology like, for example, a two-layered ANN. However if we try to experiment learning the **XOR** function by a simple perceptron we will obtain the following results.

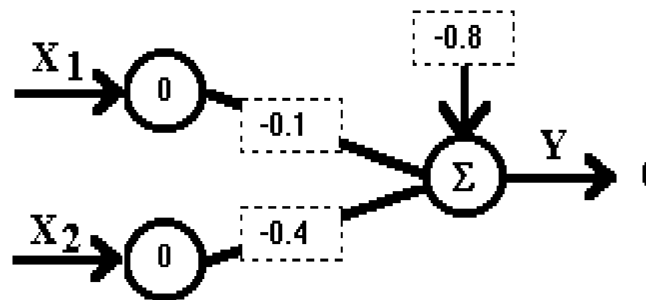

Figure 18 The graphic representation of the network state before trying to learn the **XOR** logic function

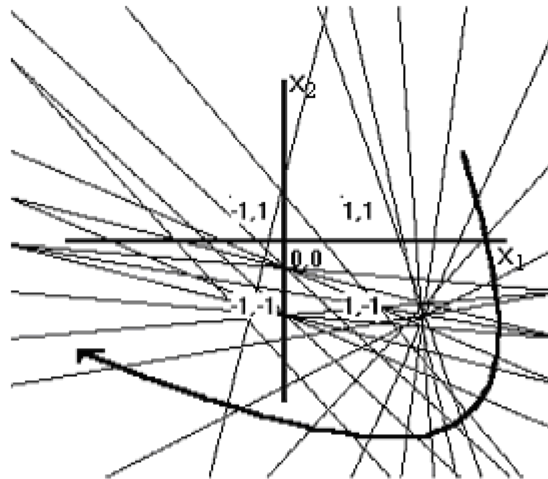

Figure 19 The graphic representation of the network state evolution when trying to learn the **XOR** logic function

The intermediate states of the network cover the whole input space and finally oscillates between some fixed positions. It is important to notice that these fixed positions represent approximate solutions of the classification problem, solutions that ignore (i.e., do not learn) some of the vectors in the training set.

In the **XOR** function case, these approximate solutions are not acceptable because the number of training vectors is very small (i.e., four) and the incorrect classification of one of them cannot be overlooked, but in more complex problems involving vectors with higher dimensionality and probably in a greater number, these approximate solutions may be accepted.

The rationale beyond this comes from the fact that non-linear divisibility of classes may derive from some training sets containing errors introduced by us (e.g. outliers) and which the network cannot learn. In other words, it is possible that, because some errors in the training vector sets, the classes may become nonlinearly separable. In these cases, there are chances that, using an approximate solution, the network learns the good training examples and eliminates the erroneous ones.

3. The Multi-layer Perceptron Model (MLP)

The motivation of developing such an ANN model was that the real world problems do not always employ linearly separable classes and therefore different network topologies and/or algorithms able to deal with nonlinearly separable classes, were needed.

The MLP is made of more than one layer of neurons, completely connected and this feature makes it capable to approximate ("learn") more complex function to accept for training nonlinearly separable classes. Unlike the simple perceptron, in MLP, the decision regions are more complex, being determined by the neurons in the last layer, which processes the signals from the adjacent hidden layer.

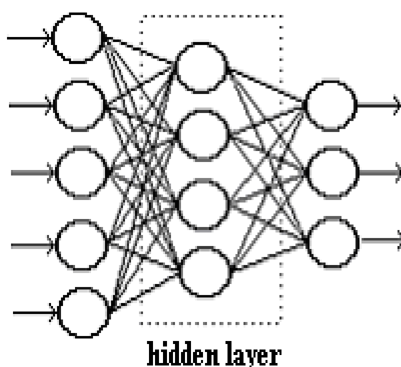

Figure 20 The topology of the MLP

The MLP model in Figure 20 has two functional layers: a hidden layer and the output layer. The input layer is not taken into account because it does not perform any calculation.

The problem that appears in a multi-layered ANN is that the learning algorithm should be different of the simple perceptron algorithm, because there is no possibility of knowing the correct output values of the neurons in the hidden layer and decide if modifying the weights and threshold values is appropriate. However we may consider (although this is not always true) that there is a correlation between the hidden layer's neuron's error and the output layer's neuron's error and therefore, each time when the network does an incorrect classification all layers' weights and threshold values should be changed.

The well-known back-propagation algorithm is based on this very assumption. Its learning strategy consists in back-propagating the network output error from the output layers to the hidden layers and calculating the hidden layers' neurons' errors. Having a value of each neuron's error gives the possibility of adjusting any of the network parameters.

However, in 1986, S. I. Gallant described an alternative to the back-propagation algorithm. This is also called the "pocket algorithm" because its functionality is based on storing ("like in a pocket") the set of parameters for which the network performs the best. When a vector is not correctly classified then the weights of the neurons in all layers (including the hidden ones) are modified using the perceptron algorithm. We have to mention that, for the same training set of vectors, the algorithm results may differ because they are strongly dependent of the initial random state of the network. It is possible that the convergence to a solution will not be achieved in some instances.

MLP Learning the XOR Logic Function

In order to learn the **XOR** logic function a MLP must have at least 2 neurons in the hidden layer and a neuron in the output layer. As we saw in the Table 5, the **XOR** function has identical output values for the vectors X_1 and X_4 and respectively X_2 and X_3. These pairs of vectors belong to same classes although the Hamming distances between them have maximal values (i.e., equal to 2 for 2-dimensional vectors). Vice-versa, the Hamming distances between the vectors that belong to opposite classes are smaller, having a value equal to 1.

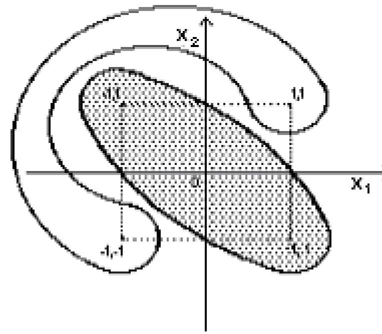

Figure 21 The graphic representation of the two non-linearly separable classes defined by the logic **XOR** function

In the graphic representation of the four vectors we observe how the Hamming distance is correlated perfectly with the spatial distance, i.e., from the spatial point of view, the most distant vectors also have the greatest Hamming distances between them. The reason for the simple perceptron is not discriminating between non-linearly separable classes is that, in a given space, it can learn to classify in same classes only vectors that are close to each other from the Hamming distance point of view.

An ANN that could learn the **XOR** logic function should therefore recode (a process that takes place in the hidden layer) somehow the four input vectors in a way that ameliorates the Hamming distances and makes the newly formed classes (at the output of the hidden layer) linearly separable for the simple perceptron made of the neuron in the output layer.

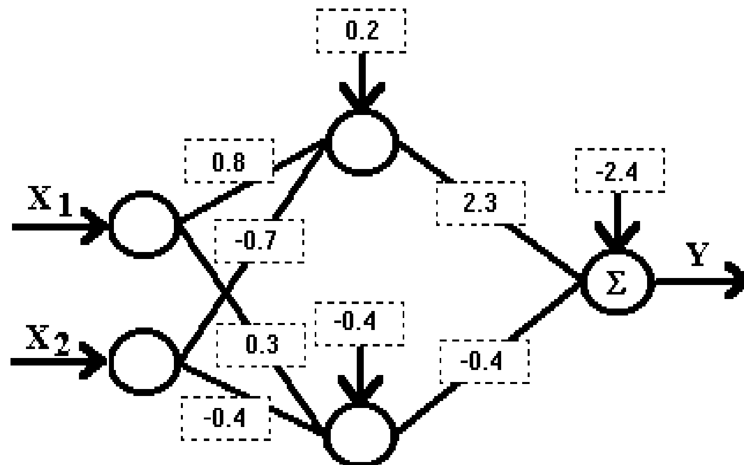

Figure 22 The final state of the multi-layer network after learning the **XOR** logic function

In the MLP in Figure 22, the recoding process is performed in the hidden layer by the two neurons whose activations are shown in Table 6:

x_1	x_2	neuron 1	neuron 2
-1	-1	-1	1
-1	1	-1	-1
1	-1	1	1
1	1	-1	1

Table 6 The re-coding process of the 4 input vectors performed in the MLP hidden layer

After the recoding we observe that 2 of the 4 vectors resulted are identical. Therefore, the simple perceptron in the output layer will actually classify only 3 vectors. The two neurons in the hidden layer learn two functions, dividing the input space into three decision zones. The zone in the middle corresponds to the vectors X_1 and X_4 for which the hidden neurons have different outputs (Table 6).

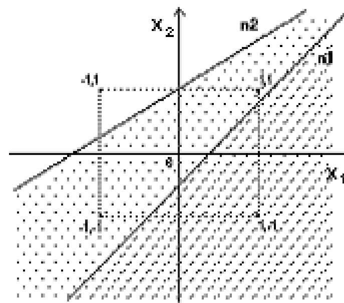

Figure 23 The graphic representations of the decision zones determined by the hidden layer neurons

hidden neuron 1	hidden neuron 2	final neuron
-1	1	-1
-1	-1	1
1	1	1

Table 7 The classification process of the three vectors resulted from recoding and performed in the output layer

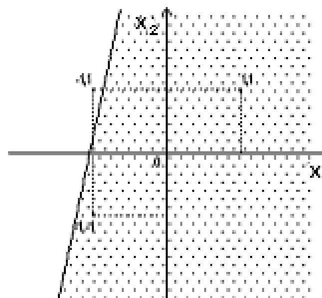

Figure 24 The graphic representation of the decision zones determined by the output neuron

The final neuron splits the newly formed space into two decision zones. The dotted area corresponds to the activated state. So, by recoding the input vectors, nonlinearly separable classes determined by the **XOR** logic function had been internally transformed by the hidden layer into linearly separable ones which could be learned by the simple perceptron in the output layer.

The study of this ANN model learning the **XOR** function could be extended to the study of human reasoning. What we may conclude is that a classification process into linearly separable classes is easy because

the class "prototype extraction process" from the training data sets is easier than that of non-linearly separable classes. This is because the Hamming distances between the vectors in different classes are bigger than those distances between vectors belonging to the same classes. So, in a decision process, a natural way of operating might be to try re-coding the nonlinearly separable classes into linearly separable ones.

4. Algorithm for Transforming Nonlinearly Separable Classes into Linearly Separable Classes

This transformation can be done by adding *common components* to vectors belonging to same class, passing the classification problem in a higher dimensional space. The process of adding common components could extend the perceptron algorithm and may be executed in the eventuality that a simple perceptron cannot learn to discriminate between some classes after a limited number of iterations.

For example, let's take the **XOR** logic function problem. The **XOR** function could be learned by a single layer perceptron if the four 2-dimensional input vectors would be given a third common component (x_3) having the role to decrease the Hamming distances between vectors in same class. In this way the non-linearly separable classes will become linearly separable (i.e., there is a hyperplane separating them). This can be seen in the graphical representation in which the 3-dimensional vectors are shown as corners of a cube.

	x_1	x_2	$\mathbf{x_3}$
X_1	0	0	**1**
X_2	0	1	**0**
X_3	1	0	**0**
X_4	1	1	**1**

Table 8 The re-coding of the 4 input vectors by adding a third common component (x_3) to the vectors in similar classes

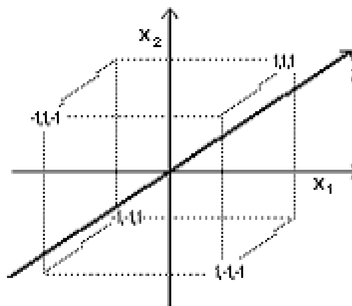

Figure 25 Graphic representation of the 4 vectors modified by adding a common component (x_3)

If we consider the cube edge as the measure unit, the Euclidean distance between two corners will be equal to the squared root of the Hamming distance between the respective vectors.

The process of classification into non-linearly separable classes could resemble a complex decision that looks difficult at first sight. In the medical field such examples are quite easy to find. The nonlinearly separable classes could be associated with pathologic entities whose intricate semiology renders them very difficult to differentiate.

If we want a simple layer perceptron to solve classification problem into non-linearly separable classes we would have to use the following algorithm:

1. execute the training process using the classic perceptron algorithm
2. if the vectors in the training sets could not be learned in a limited number of iterations we go to step 3 otherwise the training process is finished
3. identify the input vectors that could not be learned

4. identify the classes to which these problem vectors belong; for each of the classes identified we execute the following procedure:
    - all the vectors in the class will be given a new component with the same value (we may use the value 1)
    - all the other input vectors will be given a new component with a neutral value (we may use the value 0)

We restart the training process from the step 1, hoping that this time the network will be capable of learning all the examples. In the end, indirectly, we will have a measure of the non-linearity of the classification problem, actually a measure of its complexity; this measure will be given by the number of common components added to the training set of vectors.

Conclusions

Classic logical-symbolic models are successfully used in domains in which knowledge is structured as algorithms, rules or clear principles of solving problems. This knowledge could be enclosed in a computerized knowledge base and will have a decisive contribution to an expert systems' performances.

In contrast, ANN should be mostly used in domains in which the knowledge is rather unclear, the algorithms nonexistent or very difficult to implement and the principles of solving problems unable to cope with all the situations that may appear. The medical field fits perfectly this description.

Further, as far as choosing between different types of ANN is concerned, we must say, and this is very clearly stated in the literature, that the most suitable model is the simplest that could solve the given problem. This is called the *parsimony principle* and was first enounced by the medieval philosopher William of Ockam. So, if we are to choose between a simple perceptron model and a MLP, for a problem of classification in linearly separable classes, we must choose the first one. However the discussion of choosing between models of ANN remains open even when the classes are nonlinearly separable because these classes could be transformed into linearly separable ones simply by passing the problem into a higher dimensional space, as we saw in the previous chapter.

When discussing the ANN we considered that the data in the training sets is not affected by errors or in other words there are no wrongly classified vectors in these sets. In real world problems and especially in the medical field, this rarely happens. Almost in all cases, there is a certain amount of noise affecting the data. This noise is characterized by a certain probability distribution that usually is not known. The classification methods presented not only do not perform well but they will even make wrong classifications due to these data errors (i.e., the structure is not robust).

By definition, a robust system must be immune to vectors in the training set which are not correctly classified. This robustness is obtained by embedding probability information into the ANN models. Consequently these robust models of ANN will need a lot more training examples in order to learn but once they are trained, the better classification results are guaranteed.

References

[1] Eric R. Kandel, James H. Schwartz, **Principles of Neural Science**, Elsevier North-Holland, 1981

[2] Patrick H. Winston, **Artificial Intelligence**, Addison-Wesley Publishing Company, Inc., 1977

[3] P.J. Braspenning, F. Thuijsmann, A.J.M.M. Weijters, **Artificial Neural Networks. An Introduction to ANN Theory and Practice**, Springer

[4] D. Dumitrescu, Hariton Costin, **RETELE NEURONALE. Teorie si aplicatii**, Editura Teora, Bucuresti, 1996

[5] Hojjat Ali, Shih-Lin Hung, **Machine Learning: Neural Networks, Genetic Algorithms and Fuzzy Systems**, John Wiley & Sons, Inc., 1995

[6] **Readings in Artificial Intelligence. The First Decade**, edited by W.J. Clancey and E.H. Shortliffe, Addison-Wesley Publishing Company, Inc., 1984
[7] Adriana Dumitras, **Proiectarea retelelor neurale artificiale**, Editura Odeon, Bucuresti, 1997