How does the Perceptron find a solution?

av

Anna-Karin Hermansson

2013 - No 7

How does the Perceptron find a solution?

Anna-Karin Hermansson

Självständigt arbete i matematik 15 högskolepoäng, Grundnivå

Handledare: Martin Tamm

2013

Contents

1 The Singlelayer Perceptron **3**
1.1 Introduction . . . 3
1.2 Structure . . . 5
1.2.1 The learning rule . . . 6
1.2.2 The Bias input . . . 7
1.3 The learning algorithm . . . 8
1.3.1 An example . . . 9
1.4 Linear separability . . . 11
1.4.1 The exclusive Or (XOR) Function . . . 12
1.5 The Perceptron convergence procedure . . . 14
1.5.1 The Theorem . . . 14
1.5.2 The Proof . . . 17
1.5.3 An example . . . 20
1.6 Limitations . . . 21

2 The Multilayer Perceptron **22**
2.1 Introduction . . . 22
2.1.1 An example . . . 23
2.1.2 Three different types of threshold functions . . . 24
2.2 Hard limiting nonlinear models . . . 25
2.2.1 Decision regions . . . 25
2.2.2 Sequential learning . . . 27
2.3 The Backpropagation algorithm - a continuous nonlinear model . 28
2.3.1 Introduction . . . 28
2.3.2 Going forwards . . . 30
2.3.3 Going backwards: Backpropagation of Error . . . 31
2.3.4 The learning algorithm . . . 38
2.3.5 Initialising the weights . . . 39
2.3.6 Local Minima . . . 39
2.3.7 Generalisation and overfitting . . . 41
2.3.8 Number of hidden layers . . . 41
2.3.9 Decision regions . . . 42
2.3.10 Limitations . . . 43

3 Implementation **43**
3.1 Introduction . . . 43
3.2 Description . . . 44
3.3 Hypothesis, results and comments . . . 46

Acknowledgements

First of all, I wish to thank Pepto systems for giving me the opportunity to realize my degree project at their company. I would like to give a special thanks to my supervisor Tomas Sjögren and the programmer Johan Larsson for their attention.

Further, I express my thanks to Martin Tamm, my supervisor at the university, for his great commitment to this thesis. I would also like to extend my thanks to Jesper Oppelstrup from the Royal Institute of Technology, Stockholm, for his tips concerning the implementation part and my examiner Rickard Bögvad for his valuable critics on an earlier draft of this paper.

Lastly, I would like to thank my family, friends and course mates for their support throughout my time at the university. It has meant a lot to me.

1 The Singlelayer Perceptron

1.1 Introduction

Artificial neural net models are a type of algorithms which have been studied for many years, in the hope of achieving human-like performance in areas such as speech- and image recognition. These models are composed by many nonlinear computational elements working in parallel and arranged in patterns with inspiration from biological neural nets. These computational elements or nodes are interconnected via weights who typically adapt themselves during use in the purpose of improving performance.

In this report, I will focus on the Perceptron, an algorithm created by neural nets which uses a method called Optimum minimum-error in order to classify binary patterns. This algorithm is a highly parallel building block which illustrates neural-net components and demonstrates principles which can be used to form more complex systems [4]. I will start this thesis by, in this first chapter, discussing the theory behind the simplest form of the perceptron, namely the Singlelayer Perceptron. When moving on to the second chapter, I will discuss the Multilayer perceptron which consists of more complex networks. Finally, in the third chapter I will demonstrate an implementation example of a perceptron network used for image recognition. The main question that I will try to answer throughout this report is:

How does the Perceptron find a solution?

Description The perceptron is often described as a highly simplified model of the human brain (or at least parts of it) and was introduced by the psychologist Frank Rosenblatt in 1959, whose definition of it follows as:

"Perceptrons... are simplified networks, designed to permit the study of lawful relationships between the organization of a nerve net, the organization of its environment, and the "psychological" performances of which it is capable. Perceptrons might actually correspond to parts of more extended networks and biological systems; in this case, the results obtained will be directly applicable. More likely they represent extreme simplifications of the central nervous system, in which some properties are exaggerated and others suppressed. In this case, successive perturbations and refinements of the system may yield a closer approximation [5]."

Moreover, in computational terms the perceptron can be described as an algorithm for supervised classification of an input into one of several possible non-binary outputs [17]. This means that a training set of examples with the correct responses (targets) are provided and, based on this training set, the algorithm generalises to respond correctly to all possible inputs. More specifically, the Perceptron decides which of N classes different input vectors belong to, based on the training from examples of each class. The classification problem is discrete meaning that each example belongs to precisely one class, and that the set of classes covers the whole possible output space [1].

The Neuron The most essential component in the perceptron is the neuron, which in biological terms corresponds to nerve cells in the brain. I will here give a short description of how this unit acts in the brain: There are about 10^{11} neurons in the brain and they function as processing units. Their general operation is the following: Transmitter chemicals that exist in the fluid of the brain raise or lower the electrical potential inside the body of the neuron. If this membrane potential reaches a certain threshold, the neuron spikes or fires and a pulse of fixed strength and duration is sent down a [1] nerve fibre called axon [15]. The axons divide into connections to other neurons, by connecting to each of these neurons in a [1] structure called Synapse [16]. The learning in the brain basically occurs by modifying the strength of these synaptic connections between the neurons and by creating new connections. If each neuron were seen as a separate processor, whose computation is about deciding whether or not to fire, then the brain would be a massively parallell computer with many processing elements [1].

In 1962, Rosenblatt demonstrated the capabilities of the Perceptron in his book ”Principles of Neurodynamics”, which brought attention to the area of Neural ”connectionistic” networks. However, his book was not the first work to treat the area. The neuroscientist Warren McCulloch, the logician Walter Pitts and psychologist Donald O. Hebb released work treating Neurological networks already in the 1940s [5].

Hebb's rule In 1949, Hebb introduced a scientific theory, called Hebb's rule [18], which is based on the fact that changes in the strength for synaptic connections are proportional to the correlation to the firing of the two connecting neurons. To explain this further: If two neurons consistently fire simultaneously, then it will affect the strength of the connection in between them in the sence that it will become stronger. However, if two neurons never fire simultaneously the connection in between them will eventually die out. So the idea is that if two neurons both respond to something it would mean that they should be connected.

McCulloch and Pitts neuron In 1943, McCulloch and Pitt constructed a mathematical model in the purpose of capturing the bare essentials of the neuron leaving out all extraneous details. Their neuron was built up by:

- a set of weighted inputs, w_i, corresponding to the synapses

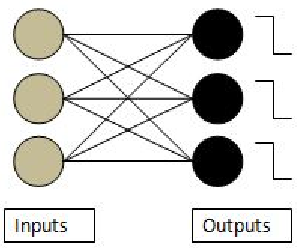


Figure 1: A single layer perceptron

- an adder that summed up all input signals
- an activation function that decided whether or not the neuron should fire, depending on the current inputs

Note that their model is not very realistic, since real neurons are much more complicated, but by building networks of these neurons, a behaviour resembling the action of the brain can be provided.

1.2 Structure

The perceptron is a collection of neurons, inputs and weights that connect the inputs with the neurons. An illustration of this is shown in figure 1. The input nodes together correspond to the input layer of the Perceptron and they are marked out as the greyshaded dots placed on the left in the image. These nodes measure up to the input values that are beeing fed into the network, which in turn correspond to the elements of an input vector. Thus, the number of input values correspond to the dimension of the input vector. The neurons in the perceptron together form the output layer and they are illustrated as the blackcoloured dots to the right in the picture. Their thresholds are marked out as the Z-shaped symbols to the right of them.

The neurons are independent from each other in the sense that the action of one neuron does not influence on the action of other neurons in the perceptron. In order to make the decision whether or not to fire, the sum of the weights connected to the neuron should be multiplied by the inputs and then compared with its own threshold. To explain this further let's denote the elements of the input vector, x_i, where $i = 1, 2, \ldots$ and give every weight the subscript of w_{ij}, such that i is an index that runs over the number of inputs and j correspondingly the number of neurons. For instance, w_{32} would be the weight that connects input number three to neuron number two. Then the value that the jth neuron needs to compare with its threshold in order to make a decision would be $\sum_{i=1} w_{ij} \cdot x_i$. The output of a neuron contains a value that holds the information of whether or not the neuron has fired. In figure 1 the number of inputs and the number of neurons are the same, but that does not have to be the case, in fact the number of inputs and outputs is determined by the data. The purpose of the perceptron is to learn how to reproduce a particular target, which is a given value in the network, by producing a pattern of firing- and non-firing neurons for given inputs. To work out if a neuron should fire or not, the values of the input nodes should be set to the values of the elements in an inputvector followed up by the calculations

$$h = \sum_{i=1}^{m} w_{ij} x_i \tag{1}$$

$$o = g(h) = \begin{cases} 1, & \text{if } h > \theta \\ 0, & \text{if } h \leq \theta \end{cases} \tag{2}$$

where h represents the input to a neuron and o is a threshold function that decides whether or not the neuron should fire [1].

1.2.1 The learning rule

Applying equations (1) and (2) on every neuron would create a pattern of firing and non-firing neurons in a vector of ones and zeros. For instance, a vector looking like (0, 1, 0, 0, 1) would mean that the second and fifth neuron fired and the others did not. This pattern should be compared with the target, i.e. the correct answer, in order to determine which neurons got the right answer and which ones got the wrong answer. The reason why this information is necessary is that some of the weights need to be adjusted and by knowing this the Perceptron knows exactly which weights to change and not. The weights connecting to neurons that calculated the wrong answer need to be adjusted with the purpose of getting closer to the correct answer next time, whereas the values of the weights connecting to neurons with the correct answer should be maintained. The initial values of the weights are unknown. In fact, it is the duty of the network to find a set of values that works and that means producing the correct answer for all neurons.

In order to demonstrate the adjustment procedure of the weights, let's consider a network with a given input vector where exactly one of the neurons get the wrong answer. If the input vector has m elements, there should be m weights connecting to each neuron. Giving the failing neuron the label k would mean that the weights that needs to be changed are the ones denoted w_{ik}, where $i = 1, 2, \ldots, m$, that is the weights connecting the inputs to neuron k. Now, the Perceptron knows which weights to change, but it also need to know how to change them. First of all, it needs to determine if the value of the weights are too high or too low. If the neuron fired when it should not, it means that some of the weights are too big and in the opposite way if it did not fire when i should have, then some of the weights must be too small. To find out in what way the

kth neuron has failed, the Perceptron calculates $t_k - y_k$, where t_k is the target and y_k is the actual answer that it has produced. If this difference is positive, the neuron should have fired when it did not and vice versa if the difference is negative. One thing that needs to be taken in consideration now is that elements of the input vector could be negative, meaning that the value of the weight also needs to be negative if we wanted a non-firing neuron to fire. To overcome this problem, the input value can be multiplied by the difference between the target and the actual output, creating the expression $\Delta w_{ik} = (t_k - y_k) \cdot x_i$. Adding this new product to the old weight value would almost complete the updating process of the weight. There is only one thing left to consider and that is to adding a learning rate, η, to the equation. This parameter determines by how much a weight should be changed and it affects how fast the network will learn. I will discuss this learning rate further in the next part. To sum up, the final rule for updating a weight can be expressed as:

$$w_{ij} + \eta(t_j - y_j) \cdot x_i \rightarrow w_{ij} \tag{3}$$

The process of calculating the activations of the neurons and updating the weights is called the training of the network and it will be repeated until the Perceptron has got all answers correct [1].

The learning Rate The learning rate, η, consequently controls how much the values of the weights should be adjusted. If the learning rate had been skipped, that is giving it the value of one, the weights would have changed a lot whenever the answer was wrong. That could cause the network to be unstable and consequently the weights might never settle down. On the other hand, the cost of having a small learning rate is that the weights would have to visit the inputs very often before they could change significantly. As a result, the network would take a longer time to learn. However, it would make the network more stable and resistant to errors and inaccuracies in the data. Therefore it is preferable to include the learning rate and a common way of setting this value is within the interval $0.1 < \eta < 0.4$ [1].

1.2.2 The Bias input

As mentioned earlier, every neuron has been given a threshold, θ, that determines a value which the neuron has to reach before it can fire. This value should be adjustable and that has to do with the case when all inputs take the value zero. In such cases, the weights would not matter. To demonstrate this, assume that a network has an input layer where all inputs are zero and that the output layer consists of two neurons, one that should fire and one that should not. Let's also assume that the threshold determines the same value all the time. The result would be that the two neurons would act alike, which obviously is a problem. However, there is a way of overcoming this issue, namely by adding an extra input weight and connecting it to each neuron in the network and keeping the value of the input connected to this weight fixed. If this weight is included

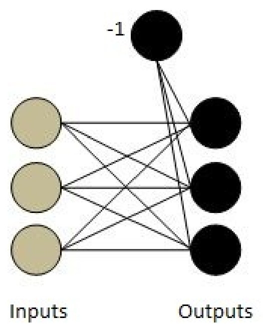


Figure 2: The bias node

in the updating process, its value will change in order to make the neuron fire or not, whichever is correct. This extra fixed input is called a Bias node and its placement in the network is illustrated in figure 2. The value of this Bias node is often set to -1 and its subscribt to 0, such that a weight connecting it with the jth neuron would be denotated as w_{0j} [1].

1.3 The learning algorithm

The perceptron algorithm consists of three phases, namely initialising the weights, training and recognition, where the training correspond to the learning process. It can be described as follows:

- **Initialisation**
    - set all the weights, w_{ij} to random low values.
- **Training**
    - for each iteration:
        - for each input vector
            - In order to determine the error, calculate the activation for each neuron, j, by using the activation function g:

$$y_j = g\left(\sum_{i=1}^{m} w_i x_i\right) = \begin{cases} 1, & \text{if } w_{ij}x_i > 0. \\ 0, & \text{if } w_{ij}x_i \leq 0. \end{cases} \tag{4}$$

            - update the weights by using the learning rule:

$$w_{ij} + \eta(t_j - y_j) \cdot x_i \rightarrow w_{ij}$$

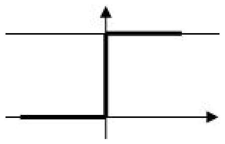
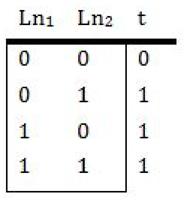


Figure 3: A Hard limiter



Ln1	Ln2	t
0	0	0
0	1	1
1	0	1
1	1	1

Figure 4: The output values computed by different input vectors

- **Recognition**
  - Calculate the activation for each neuron, j, by using equation:

$$y_j = g\left(\sum_{i=1}^{m} w_i x_i\right) = \begin{cases} 1, & \text{if } w_{ij}x_i > 0. \\ 0, & \text{if } w_{ij}x_i \leq 0. \end{cases}$$

[1] There are different types of threshold functions [4]. The function that have been used in this algorithm is called a Hard limiter and is illustrated in figure 3. I will treat different types of threshold functions later on in this report, but in this first chapter I will stick to the hard limiters.

1.3.1 An example

Here follows an example with the Perceptron learning. Consider a network with two input nodes, one bias input and one output neuron, where the values of the inputs and targets are given in the table in figure 4. In figure 5 you can see a plot of the function in the input space where the high outputs are marked as crosses and the low outputs are marked as circles. Figure 6 shows the corresponding perceptron. Denote the inputs by x_0, x_1, x_2 and the corresponding weights by w_0, w_1, w_2. The initial values of the weights is set to $w_0 = -0.05$, $w_1 = -0.02$ and $w_2 = 0.02$ and the fixt value of the bias input x_0 is set to -1. The network starts with the first input vector $(0, 0)$ and calculates the value of the neuron:

$$w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2 = (-0.05) \cdot (-1) + (-0.02) \cdot 0 + (-0.02) \cdot 0 = 0.05$$

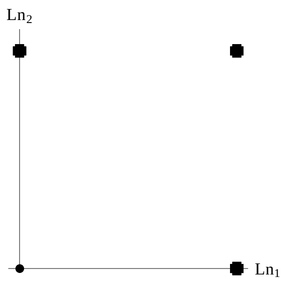


Figure 5: A graph created by the values from the table in figure 4

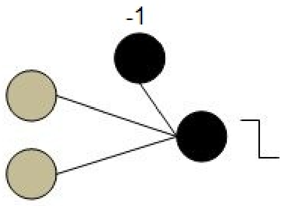


Figure 6: The perceptron network from the example in 1.3.1

As 0.05 is above the threshold value zero, the neuron fires and the output receives the value one. However, this is not the correct answer according to the target which is zero. Therefore the learning rule must be applied in order to adjust the weights. The learning rate used in this algorithm is $\eta = 0.25$.

$$w_0 + \eta(t-y) \cdot x_0 \rightarrow w_0 \Rightarrow (-0.05) + 0.25 \cdot (0-1) \cdot (-1) = 0.2$$
$$w_1 + \eta(t-y) \cdot x_1 \rightarrow w_1 \Rightarrow (-0.02) + 0.25 \cdot (0-1) \cdot 0 = -0.02$$
$$w_2 + \eta(t-y) \cdot x_2 \rightarrow w_2 \Rightarrow 0.02 + 0.25 \cdot (0-1) \cdot 0 = 0.02$$

The next input vector is $(0, 1)$ and computing the value of the output in the same way as with the first input results in a non-firing neuron. From the table in figure 4 one can tell that this is an incorrect answer. As the target for this input vector is one, i.e. that the neuron should fire, the weights need to be updated again:

$$w_0 + \eta(t-y) \cdot x_0 \rightarrow w_0 \Rightarrow 0.2 + 0.25 \cdot (1-0) \cdot -1 = -0.05$$
$$w_1 + \eta(t-y) \cdot x_1 \rightarrow w_1 \Rightarrow -0.02 + 0.25 \cdot (1-0) \cdot 0 = -0.02$$
$$w_2 + \eta(t-y) \cdot x_2 \rightarrow w_2 \Rightarrow 0.02 + 0.25 \cdot (1-0) \cdot 1 = 0.27$$

The next input vectors $(1, 0)$ and $(1, 1)$ get the correct answers which means that the weights do not need to be updated. Now, the perceptron will start from the beginning again with the updated weights and perform the same process until all answers are correct. When this is accomplished, the weights will settled down and the algorithm will be finished. The perceptron has then learnt all the examples correctly. For complete calculations of this algorithm, see appendix.

Note, that it is possible to pick lots of different values for the weights than the ones used in this particular example, in order to get the correct outputs. The weight values that the algorithm finds depends on the learning rate, the inputs and the intial starting values of the weights. The interesting thing here is not the actual values of the weights, but a set of values that actually works, meaning that the network should generalise well to other inputs. In this example the Perceptron converges successfully, meaning that it finds a set of weights that classifies all input vectors correctly [1]. Further, that leads inevitably to the question:

Does the perceptron always reach convergence?

I will discuss this matter in the next chapter.

1.4 Linear separability

Perceptrons with only one output neuron try to separate two different classes from each other, where one of the classes consists of input vectors whose target is a firing neuron and the other class consists of input vectors with a non-firing

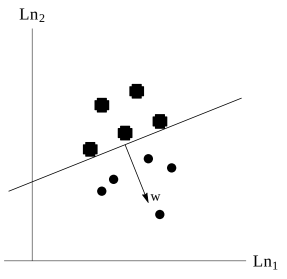


Figure 7: A Decision boundary

neuron as their target. In two dimensions the perceptron tries to find a straight line that separates these two classes, whereas in 3 dimensions this line would correspond to a plane and in higher dimensions a hyperplane. This is called a Decision boundary or a Discriminant function. So, the neuron should only fire on one side of this decision boundary. An example of this in two dimensions is illustrated in figure 7, where the decision boundary is a straight line. The cases where a separating hyperplane exists are called linearly separable cases. The example above demonstrates such a case and by looking at the graph in figure 5, one can see that it is possible to find a straight line that could separate the cross from the circles (where the crosses mean that the neuron fired and the circles mean that it did not). The cases where such a decision boundary exists are sometimes referred to as the OR function. For perceptrons containing more than one output neuron, the weights for each neuron would separately describe such a hyperplane [1]. The discussion above raises the question:

Does the Perceptron always reach convergence in the linearly separable case?

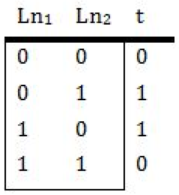
I will try to answer this question in the section 1.5 which treats the perceptron convergence procedure.

1.4.1 The exclusive Or (XOR) Function

Figure 9 demonstrates an example of the XOR function. By looking at the graph created from the values in the table, it can be found out that a straight line that could separate the crosses from the circles does not exist. Thus, the classes are not linearly separable and the perceptron would fail to get the right answer [1]. This raises naturally the question:

Can the Perceptron reach convergence in the non-separable case?
If it can, then how does it do it?

The answer to the first question is yes and the solution is to make the network



Ln_1	Ln_2	t
0	0	0
0	1	1
1	0	1
1	1	0

Figure 8: The output values computed by different input vectors

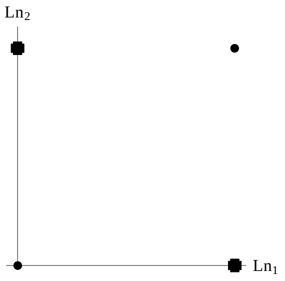


Figure 9: A graph created by the values from the table in figure 8. It illustrates an example of the XOR function.

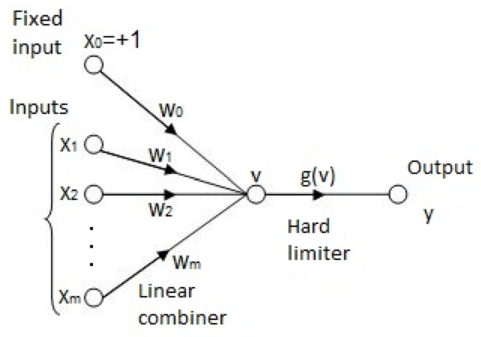


Figure 10: A Linear combiner

more complicated by adding more neurons and by making more complicated connections in between them [1]. I will discuss this further in the second chapter which treats the Multilayer perceptron.

1.5 The Perceptron convergence procedure

At the time when McCulloch and Pitt constructed this perceptron hardly any computers existed and the programming languagues were just above a minimal standard, which might have been a reason for the poor interest in it. Moreover, in the fifties some further developments were made, but at the end of the same decade things became quiet due to the success of the serial von Neumann computer. When Rosenblatt introduced the perceptron it brought attention to an almost forgotten area. The Perceptron, in its simplicity, seemed to be actually capable of learning certain things [5].

The original perceptron convergence procedure had adjusted weights and was found by Rosenblatt. He proved that if inputs belonging to two different classes were separable, the perceptron convergence procedure would converge and find a decision hyperplane that separated the two classes [4].

I will now prove the perceptron convergence using the simplest kind of architecture.

1.5.1 The Theorem

Consider a perceptron with m inputs and a linear combiner that combines them as demonstrated in figure 10. Denote the inputs as $x_1, x_2, \ldots, x_m$ and the associated weights as $w_1, w_2, \ldots, w_m$. Also add a bias input, x_0 , with a fixed value of +1, directly in the linear combiner and a connecting bias weight, w_0. This linear combiner should add up the weights multiplied by the inputs by calculating $\mathbf{w}^\top\mathbf{x}$, where $\mathbf{w}$ and $\mathbf{x}$ are colonn matrices containing the elements

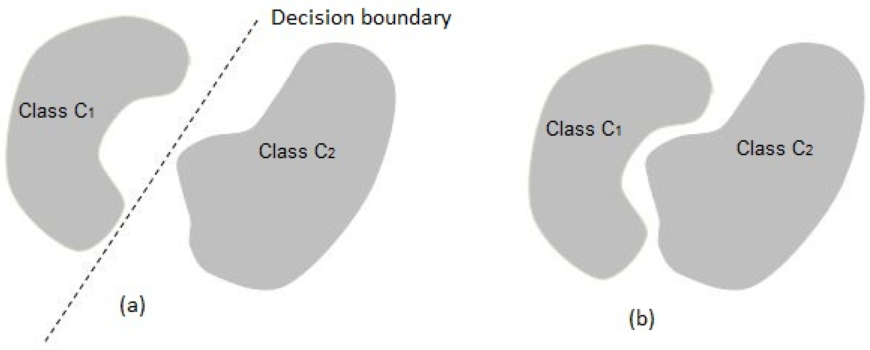


Figure 11: Linear separability

$\mathbf{w} = (w_0, w_1, \ldots, w_m)$ and $\mathbf{x} = (x_0, x_1, \ldots, x_m)$. The result of this calculation is the input to the neuron, v. Thus, after the presentation of the nth pattern, v is written as

$$v(n) = \overrightarrow{\mathbf{w}}^{\top}(n)\overrightarrow{\mathbf{x}}(n) \tag{5}$$

where $\overrightarrow{\mathbf{w}}^{\top}$ is the transpose of the weight-vector, $\overrightarrow{\mathbf{x}}(n)$ is the input-vector for the nth pattern that has been fed into the network and $v(n)$ is the scalar product of $\overrightarrow{\mathbf{w}}(n)$ and $\overrightarrow{\mathbf{x}}(n)$. v is in turn followed up by a hard limiter, creating the output value $y = \varphi(v)$, by transforming it into the value of $+1$ or -1. Thus, I will not use the same threshold function as the one I have been using so far, where the output values zero or one were produced. The reason why I am instead using a hard limiter that produces the values $+1$ or -1 is to simplify the presentation of this proof. The fact that there are only two values that the output of the neuron can produce makes our network a *binary pattern classification.*

Now, this pattern classification problem will be solved as a two class problem. Let the two different classes C_1 and C_2 be linearly separable subsets of R^n. Let also any pattern, i.e any input vector $\overrightarrow{\mathbf{x}}$ from the n-dimensional space, which is beeing fed into the network is either going to belong to C_1 or C_2. Figure 11 demonstrates the sets of the classes in two dimensions, where (a) is a linearly separable case and (b) is not. The end product of the training will be a separating hyperplane (there can be many) such that the following conditions are satisfied:

$$v = \overrightarrow{\mathbf{w}}^{\top} \cdot \overrightarrow{\mathbf{x}} > 0 \text{ when } \overrightarrow{\mathbf{x}} \in C_1 \tag{6}$$

$$v = \overrightarrow{\mathbf{w}}^{\top} \cdot \overrightarrow{\mathbf{x}} \leq 0 \text{ when } \overrightarrow{\mathbf{x}} \in C_2 \tag{7}$$

By sending the input of the neuron, v, into the hard limiter function, the output of the neuron, y, will be computed. This can be expressed as

$$v > 0 \Rightarrow y = +1 \tag{8}$$

$$v \leq 0 \Rightarrow y = -1. \tag{9}$$

This hyperplane corresponds to the decision boundary and its equation is $\overrightarrow{\mathbf{w}}^\top \cdot \overrightarrow{\mathbf{x}} = 0$. In the linearly separable case, shown at image (a) in figure 11, there is a hyperplane that separates the two classes. However, in the non-linearly separable case, shown at image (b) in figure 11, such a hyperplane does not exist.

If $\overrightarrow{\mathbf{w}}^\top \cdot \overrightarrow{\mathbf{x}}$ is greater than zero, after having fed an x-vector from the set C_1 into the network, the classification is correct. However, if an x-vector from the set C_2 is beeing fed into the network and $\overrightarrow{\mathbf{w}}^\top \cdot \overrightarrow{\mathbf{x}}$ still is greater than zero, the classification is incorrect. In the cases when the classification is correctly made, the values of the weights should remain the same and no action needs to be taken. However, if the network has classified incorrectly the weights need to be updated through the learning rule.

- For correct classifications maintain the values of the weights

$$\overrightarrow{\mathbf{w}}(n+1) = \overrightarrow{\mathbf{w}}(n) \text{ if } \overrightarrow{\mathbf{w}}^\top \cdot \overrightarrow{\mathbf{x}}(n) > 0 \text{ and } \overrightarrow{\mathbf{x}}(n) \in C_1 \tag{10}$$

$$\overrightarrow{\mathbf{w}}(n+1) = \overrightarrow{\mathbf{w}}(n) \text{ if } \overrightarrow{\mathbf{w}}^\top \cdot \overrightarrow{\mathbf{x}}(n) \leq 0 \text{ and } \overrightarrow{\mathbf{x}}(n) \in C_2 \tag{11}$$

- For incorrect classifications update the weights by using the learning rule

$$\overrightarrow{\mathbf{w}}(n+1) = \overrightarrow{\mathbf{w}}(n) + \eta\overrightarrow{\mathbf{x}}(n) \text{ if } \overrightarrow{\mathbf{w}}^\top \cdot \overrightarrow{\mathbf{x}}(n) \leq 0 \text{ and } \overrightarrow{\mathbf{x}}(n) \in C_1 \tag{12}$$

$$\overrightarrow{\mathbf{w}}(n+1) = \overrightarrow{\mathbf{w}}(n) - \eta\overrightarrow{\mathbf{x}}(n) \text{ if } \overrightarrow{\mathbf{w}}^\top \cdot \overrightarrow{\mathbf{x}}(n) > 0 \text{ and } \overrightarrow{\mathbf{x}}(n) \in C_2 \tag{13}$$

Note that the learning rule has different signs in the two different cases. The reason why the perceptron makes a wrong classification is that the hyperplane is intersecting at least one of the sets C_1 and C_2. Therefore it needs to move and the sign at the learning rule determines in what direction the hyperplane is moving [7]. I will now prove the following theorem.

Theorem 1. Let C_1 and C_2 be two bounded sets in R^n separated by a hyperplane given by the equation $\overrightarrow{\mathbf{w}_0}^\top \cdot \overrightarrow{\mathbf{x}} = 0$ for some vector $\mathbf{w}_0$ such that $\min |\mathbf{w}_0 \cdot \mathbf{x}| \geq \alpha > 0$, where minimum is taken over all $\mathbf{x} \in C_1 \cup C_2$. Then, $\mathbf{w}(n)$ will converge when using the learning rule.

1.5.2 The Proof

In order to prove the convergence two initial assumptions will be made. The first one is choosing the initial weight vector to be the zero vector, which is a choice that only affects the speed of convergence. The second assumption is choosing the value of the learning rate, η, to be one, which is made to simplify the analysis. The approach of this proof is to find a lower bound and an upper bound for $\mathbf{w}(n)$.

Proof. Suppose that the patterns $\overrightarrow{\mathbf{x}}(n)$, starting with iteration $n = 1, 2, \ldots$, are beeing fed into the network. For some values of n the perceptron will classify incorrectly and after each such presentation the learning rule will be used. These n:s are a part of a series and will be denoted as $n_1, n_2, n_3, \ldots$. In order to prove convergence, it is enough to to prove that the learning rule will stop the updating process after a finite number of steps.

A lower boundary The learning rule will be expressed as

$$\overrightarrow{\mathbf{w}}(n_{k+1}) = \overrightarrow{\mathbf{w}}(n_k) + \overrightarrow{\mathbf{x}}(n_k) \text{ for } \overrightarrow{\mathbf{x}}(n_k) \in C_1 \tag{14}$$

and

$$\overrightarrow{\mathbf{w}}(n_{k+1}) = \overrightarrow{\mathbf{w}}(n_k) - \overrightarrow{\mathbf{x}}(n_k) \text{ for } \overrightarrow{\mathbf{x}}(n_k) \in C_2 \tag{15}$$

where $\eta = 1$. By using the assumption $\overrightarrow{\mathbf{w}}(n_0) = \overrightarrow{0}$ together with the learning rules following expressions can be made:

$$\overrightarrow{\mathbf{w}}(n_1) = \pm\overrightarrow{\mathbf{x}}(n_0) \tag{16}$$

$$\overrightarrow{\mathbf{w}}(n_2) = \pm\overrightarrow{\mathbf{x}}(n_0) + \overrightarrow{\mathbf{x}}(n_1) \text{ for } \overrightarrow{\mathbf{x}}(n_k) \in C_1 \tag{17}$$

$$\overrightarrow{\mathbf{w}}(n_2) = \pm\overrightarrow{\mathbf{x}}(n_0) - \overrightarrow{\mathbf{x}}(n_1) \text{ for } \overrightarrow{\mathbf{x}}(n_k) \in C_2 \tag{18}$$

leading to the general expression:

$$\overrightarrow{\mathbf{w}}(n_{k+1}) = \pm\overrightarrow{\mathbf{x}}(n_0) \pm \overrightarrow{\mathbf{x}}(n_1) \pm \cdots \pm \overrightarrow{\mathbf{x}}(n_k) \tag{19}$$

Taking the inner product of $\overrightarrow{\mathbf{w}_0}^\top$ and all terms in equation (19) results in

$$\overrightarrow{\mathbf{w}_0}^\top \overrightarrow{\mathbf{w}}(n_{k+1}) = \pm\overrightarrow{\mathbf{w}_0}^\top \overrightarrow{\mathbf{x}}(n_0) \pm \overrightarrow{\mathbf{w}_0}^\top \overrightarrow{\mathbf{x}}(n_1) \pm \cdots \pm \overrightarrow{\mathbf{w}_0}^\top \overrightarrow{\mathbf{x}}(n_k) \tag{20}$$

Moreover, the conditions (6) and (7) must hold for $\mathbf{w}_0$, as it is a vector belonging to a separating hyperplane. These conditions together with (14) and (15) imply that whenever $\overrightarrow{\mathbf{w}_0}^\top \overrightarrow{\mathbf{x}}(n_k) < 0$, then there must be a negative sign before the corresponding term in equation (20) and in the opposite way; when $\overrightarrow{\mathbf{w}_0}^\top \overrightarrow{\mathbf{x}}(n_k) > 0$, there must be a positive sign before the corresponding term in equation (20), for $k = 0, 1, 2, \ldots$ Thus, each term in equation (20) is positive and according to *Theorem 1* it should also be greater than or equal to α. These facts lead to the bounding expression

$$\overrightarrow{\mathbf{w}_0}^\top \overrightarrow{\mathbf{w}}(n_{k+1}) \geq k\alpha. \tag{21}$$

Applying Cauchy-Schwartz inequality on the expression $\overrightarrow{\mathbf{w}}(n_{k+1})$ results in:

$$\|\overrightarrow{\mathbf{w}_0}\|^2 \|\overrightarrow{\mathbf{w}}(n_{k+1})\|^2 \geq [\overrightarrow{\mathbf{w}_0}^\top \overrightarrow{\mathbf{w}}(n_{k+1})]^2 \Rightarrow \tag{22}$$

$$\|\overrightarrow{\mathbf{w}_0}\|^2 \|\overrightarrow{\mathbf{w}}(n_{k+1})\|^2 \geq k^2\alpha^2 \Rightarrow \tag{23}$$

$$\|\overrightarrow{\mathbf{w}}(n_{k+1})\|^2 \geq \frac{k^2\alpha^2}{\|\overrightarrow{\mathbf{w}_0}\|^2} \tag{24}$$

An upper boundary In order to find an upper bound, an alternative route can be made. Again, feed the input vectors $\mathbf{x}(n)$, $n = 1, 2, \ldots,$, into the network. As before, the system will classify incorrectly for $n_1, n_2, n_3, \ldots$. Thus, the learning rule will be expressed as:

$$\overrightarrow{\mathbf{w}}(n_{k+1}) = \overrightarrow{\mathbf{w}}(n_k) \pm \overrightarrow{\mathbf{x}}(n_k) \text{ for } k = 1, 2, \ldots \text{ and } \overrightarrow{\mathbf{x}}(n_k) \in C_1 \cup C_2 \tag{25}$$

The first step in this method is to update the squared euclidean norm on both sides of equation (25) as follows:

$$\|\overrightarrow{\mathbf{w}}(n_{k+1})\|^2 = \|\overrightarrow{\mathbf{w}}(n_k)\|^2 + \|\overrightarrow{\mathbf{x}}(n_k)\|^2 \pm 2\overrightarrow{\mathbf{w}}^\top(n_k) \cdot \overrightarrow{\mathbf{x}}(n_k) \tag{26}$$

Now, as the assumption that the Perceptron classifies incorrectly for $k = 1, 2, \ldots$ is made, it would mean that the condition $\overrightarrow{\mathbf{w}}(n_k)^\top \cdot \overrightarrow{\mathbf{x}}(n_k) < 0$ must hold for $\overrightarrow{\mathbf{x}}(n_k) \in C_1$ and $\overrightarrow{\mathbf{w}}(n_k)^\top \cdot \overrightarrow{\mathbf{x}}(n_k) > 0$ for $\overrightarrow{\mathbf{x}}(n_k) \in C_2$. These conditions imply, together with the conditions (14) and (15), the following statements: When $\overrightarrow{\mathbf{w}}(n_k)^\top \cdot \overrightarrow{\mathbf{x}}(n_k) < 0$, then there must be a positive sign in front of the last term of equation (26) and when $\overrightarrow{\mathbf{w}}(n_k)^\top \cdot \overrightarrow{\mathbf{x}}(n_k) > 0$, there must be a negative sign in front of the same term. Consequently, the last term of equation (26) must always be negative, which makes it possible to make the statements

$$\|\overrightarrow{\mathbf{w}}(n_{k+1})\|^2 \leq \|\overrightarrow{\mathbf{w}}(n_k)\|^2 + \|\overrightarrow{\mathbf{x}}(n_k)\|^2 \Rightarrow \tag{27}$$

$$\|\overrightarrow{\mathbf{w}}(n_{k+1})\|^2 - \|\overrightarrow{\mathbf{w}}(n_k)\|^2 \leq \|\overrightarrow{\mathbf{x}}(n_k)\|^2. \tag{28}$$

If equation (28) is applied on $k = 0, 1, \ldots,$ the following condition can be established:

$$\|\overrightarrow{\mathbf{w}}(n_{k+1})\|^2 \leq \sum_{k=1} \|\mathbf{x}(n_k)\|^2 \tag{29}$$

If β is defined as the positive quantity,

$$\beta = \max_{\vec{\mathbf{x}}(n_k) \in C_1 \cup C_2} \|\mathbf{x}(n_k)\|^2$$

then every $\|\mathbf{x}(n_k)\|^2$ for $k = 0, 1 \ldots$ will be less than or equal to β. This fact makes it possible to form the following bounding expression:

$$\|\vec{\mathbf{w}}(n_{k+1})\|^2 \leq k\beta \tag{30}$$

A maximum number of iterations The bound of equation (30) says that as k increases, $\|\vec{\mathbf{w}}(n_{k+1})\|^2$ increases at most linearly. It specifies an upper limit whereas equation (24) specifies a lower limit. As a result, there must be a maximum integer $k_{\max}$ such that both inequalities (24) and (30) will be satisfied. It follows that $k_{\max}$ must satisfy

$$\frac{k_{\max}^2 \alpha^2}{\|\vec{\mathbf{w}_0}\|^2} \leq k_{\max}\beta \tag{31}$$

from where we can obtain the limit of $k_{\max}$ as

$$k_{\max} \leq \frac{\beta \|\vec{\mathbf{w}_0}\|^2}{\alpha^2} \tag{32}$$

This inequality shows that the updating process must stop after a finite number of steps. Hence, the proof is completed [7].

□

The Fixed Increment Convergence Theorem follows as

Theorem 2. Let the subsets of training vectors C_1 and C_2 be linearly separable. Let the inputs presented to the perceptron originate from these two subsets. Then, the perceptron converges after some number of iterations, in the sense that

$$\vec{\mathbf{w}}(m) = \vec{\mathbf{w}}(m+1) = \vec{\mathbf{w}}(m+2) = \ldots$$

are vectors defining the same hyperplane that separates the two subsets, for $m \geq m*$, where $m*$ is a fixed number [9].

However, the main interest of the perceptron is contained in the following theorem:

Theorem 3. If H_1 and H_2 are two finite subsets of C_1 respective C_2 and x_n is a series that feeds all elements in H_1 and H_2 into the network an infinitely number of times, then the perceptron will classify all elements in H_1 and H_2 correctly after a finite number of steps.

This follows directly from *Theorem 1.* since once convergence has been reached all further classifications must be correct.

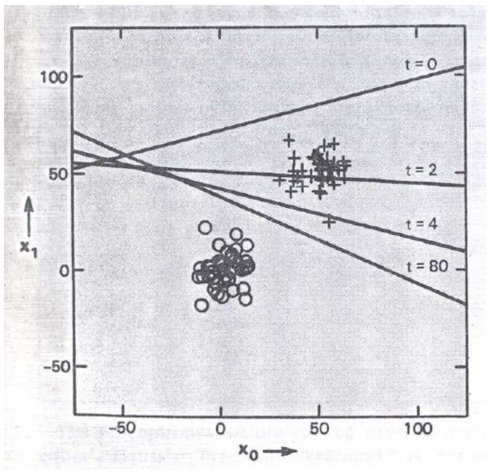


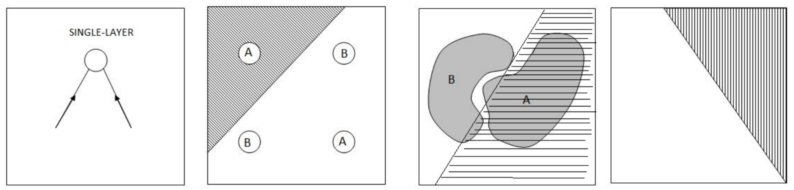
Figure 12: An example which demonstrates the behavior of the decision boundaries during the convergence procedure. This image is taken from [4].

Is the solution unique? It has already been proved, speaking of linearly separable cases, that a solution, $\overrightarrow{\mathbf{w}_0}$, exists such that the hyperplane $\overrightarrow{\mathbf{w}_0}^\top \overrightarrow{\mathbf{x}} = 0$ separates the pattern classes C_1 and C_2 for every input pattern in the training set. However, it does not mean that this hyperplane or solution $\overrightarrow{\mathbf{w}_0}$ is unique. In fact, there may be many such separating hyperplanes or solutions $\overrightarrow{\mathbf{w}_0}$. The point to make here is that a solution, $\overrightarrow{\mathbf{w}_0}$, that is ultimately able to separate the two pattern classes, can be reached and that there is a domain of such $\overrightarrow{\mathbf{w}_0}$:s that would satisfy this condition. To sum up, the idea is not to reach an exact value of $\overrightarrow{\mathbf{w}_0}$, but it is to reach convergence and by that I mean coming to a stage, after having fed patterns into the network, when correct classification is achieved [7].

1.5.3 An example

Figure 12 shows an example of perceptron convergence with two different classes. Class A is marked out with circles and class B with crosses and the different samples have been gone through until 80 inputs have been presented. As can be seen, there are four lines in the image. These lines represent different decision boundaries after the weights have been adjusted, following the errors, on iterations 0, 2, 4 and 80. As can be seen, the classes have been almost separated only after four iterations [4].

Figure 13: Decision regions for a single-layer perceptron



(a) Structure of the network (b) Exclusive or problem (c) Classes with meshed regions (d) Most general region shapes

1.6 Limitations

Rosenblatt's demonstration of the capability of the Perceptron raised a great interest in solving a larger class of problems. Therefore, a lot of research was done with the aim at finding more general methods by extending and refining the training process and building bigger machines. In spite of all this effort, it could be confirmed that there were certain things that the perceptron could not learn [5].

Now, I have proved that the perceptron convergence procedure always works in the cases when a separating hyperplane exists. However, this procedure is not appropriate in cases where classes are not linearly separable, as it would cause the hyperplane to oscillate continuously [4].

Figure 13 demonstrates the types of decision regions a singlelayer perceptron can form, namely a half plane bounded by a hyperplane. It illustrates two non-separable situations at image (b) and (c). The closed contours around the areas labelled A and B show the input distributions of the two classes, when two continuous valued inputs have been fed into the net. The shaded areas correspond to the decision regions. As can be seen in the image, the distributions of the two classes for the exlusive OR problem are disjoint and cannot be separated by a straight line. However, the shaded area at image (b) in figure 13 shows a possible decision region that the perceptron might choose. Neither the second problem, where the input distributions are meshed, can be solved by finding a straight line that would separate the two classes. Image (d) illustrates the shape of general decision regions formed by the singlelayer perceptron.

This problem was used by the cognitive scientist Marvin Lee Minsky and the mathematician and computer scientist Seymour Papert in order to illustrate the weakness of the perceptron [4]. In 1969, they elucidated not only the possibilities but also the restrictions on the perceptron in their book "An introduction to Computational Geometry". The purpose of their mathematical analysis was to advise against looking for methods that would work in every possible situation, by showing in which cases the perceptron performed well and in which cases it

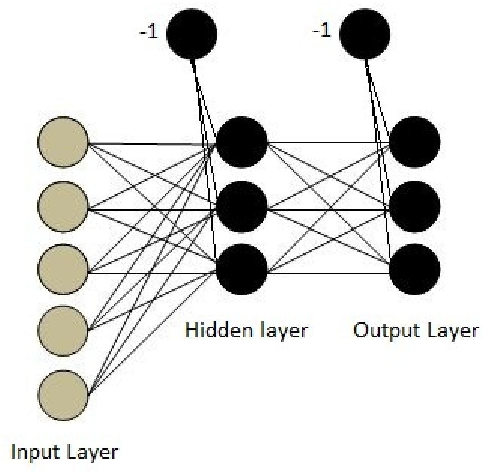


Figure 14: A multilayer perceptron network

performed badly. This publication is often seen as the reason for the diminished interest in the perceptron during the seventies. In 1988, at a republication of the book, Minsky and Papert stated that the early halt of research on neural networks was due to a lack of fundamental theories. According to them, too much effort had been spent researching on the simple Perceptron instead of what was important, namely the *Representation of knowledge.* Moreover, in the seventies the interest and research on the last-mentioned area expanded enormously [5].

2 The Multilayer Perceptron

2.1 Introduction

So far, we have seen that linear models can find separating straight lines, planes or hyperplanes. However, most problems of interest are not linearly separable. In this second chapter, I will be concentrating on making a network more complex in order to solve the classification problem. As concluded, the networks learn through the weights, so to involve more computation more weights should be added into the network. One way of doing so, is by adding more neurons in between the input nodes and the output neurons. This new structure of the network is called the Multilayer perceptron and an example is shown in Figure 14. As with the perceptron, a bias input needs to be connected to every neuron [1].

Multilayer perceptrons are feed-forward nets with one or more layers of nodes between the input- and the output nodes [4]. Feed-forward nets means that each layer of neurons feeds only the very next layer of neurons and receives input only from the immediately preceding layer of neurons. That means that the neurons do not skip layers [11]. (These layers consist of hidden units, or nodes, that does not directly connect to both the input- and output nodes.) Multilayer perceptrons can overcome the limitations that the perceptron has, but were not used in the past because of the lack of effective training algorithms. However, as new training algorithms were developed, it was shown that multilayer perceptrons actually could solve problems of interest.

The work by people like Hopfield, Rumelhart and McClelland, Sejnowski, Feldman and Grossberg amongst other names lead to resurgence within the field of neural networks. The new interest was probably due to the development of new net topologies, new algorithms, new implementation techniques and the growing fascination of the functioning of the human brain [4].

The question now is, how can a Mulitlayer network be trained so that the weights can adapt themselves in order to get the correct answers? At first, the same method as for the perceptron can be used, that is, to compute the error of the output. The next step would be to calculating the difference between the targets and the outputs. The issue to deal with now is the uncertainty of which weights that are wrong. It could be either the ones from the first layer or the second one. Besides, the correct activations for the neurons in the middle layer(s) are also unknown. As it is not possible to examine or correct the values of the neurons that belong to this layer directly, it is called the Hidden layer.

2.1.1 An example

The two-dimensional XOR problem that was demonstrated in figure 9 can not be solved by a linear model like the perceptron. However, the act of adding extra layers of nodes to the network makes it solvable and here is an example that proves it. Take a look at the neural network illustrated in figure 15, where the values of the weights and the names of each node have been marked out. In order to demonstrate that the output neuron produces the correct answers, the inputs will be fed into the network and afterwoods the results will be observed. However, this time the network will be treated as two perceptrons in the sense that the activations of the neurons in the middle, C and D, will be computed first. These will in turn represent the inputs to the output neuron E. The weights connecting the nodes in the input layer to the neurons in the hidden layer will, in this example, be denoted as v_{ij} and w_j will be the weights connecting the hidden layer neurons to the output neuron, where i is an index that runs over the nodes in the input layer and j runs over the neurons in the hidden layer. The bias nodes connecting to the hidden neurons and the output neuron are denoted as b_1 and b_2 and they have both been given the value one. The first input vector to be fed into the network is $(1, 0)$, which means that A=1 and B=0. The calculations of the input to neuron C follow as:

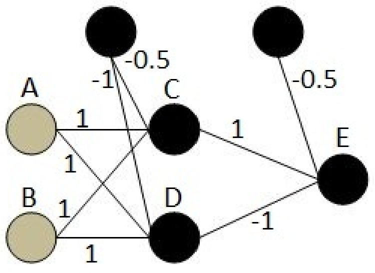


Figure 15: A Multilayer perceptron with weight values which solve the XOR problem

$$b_1 \cdot v_{01} + A \cdot v_{11} + B \cdot v_{21} = (-1) \cdot 0.5 + 1 \cdot 1 + 0 \cdot 1 = 0.5$$

As 0.5 is above the threshold 0, neuron C fires and the value of its output is thus one. The input of neuron D is calculated as:

$$b_1 \cdot v_{02} + A \cdot v_{12} + B \cdot v_{22} = (-1) \cdot 1 + 1 \cdot 1 + 0 \cdot 1 = 0$$

As an input sent to the threshold function has to be *greater than* zero before the neuron can fire, neuron D will not fire and its output value is thus set to zero. Moving on to neuron E, its input will be:

$$b_2 \cdot w_0 + C \cdot w_1 + D \cdot w_2 = (-1) \cdot 0.5 + 1 \cdot 1 + 0 \cdot (-1) = 0.5$$

which means that E fires. Now, by using the same weights for the other inputs (0,0), (0,1) and (1,1) in order to calculate the outputs, the result will be that neuron E fires when A and B have different values and does not fire when they have the same values. These latter calculations can be seen in the appendix. The conclusion to make out of this example is that the XOR function, that was unsolvable for the perceptron, could be solved by adding an extra layer of neurons into the network and thus transforming it to a non-linear model [1].

2.1.2 Three different types of threshold functions

The capabilities of multilayer perceptrons are due to the fact that the computational elements or nodes in these neural net models are nonlinear and analog, meaning that the result of the summed weighted inputs is passed through an internal nonlinear threshold as described before. Until now, hard limiters have been presented as such nonlinearity. However, there are two other types, namely threshold logic elements and sigmoidal nonlinearities. Representatives

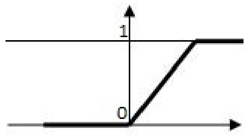


Figure 16: A Threshold logic function

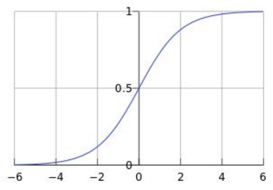


Figure 17: A Sigmoid function. This image is taken from [21].

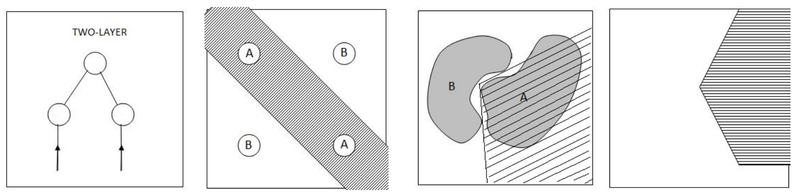
from them are illustrated in figure 16 and 17. In this second chapter I will treat both hard limiting models and a sigmoidal nonlinear model, called the Backpropagation algorithm. I will begin with describing Hard limiting models.

2.2 Hard limiting nonlinear models

2.2.1 Decision regions

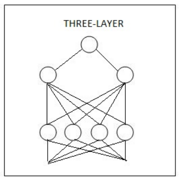
Figures 18 and 19 show the capabilities of perceptrons with two and three layers where hard-limiting nonlinearities have been used. Image (a) shows the structure of the network, images (b) and (c) demonstrate examples of decision regions for the exclusive OR problem and for meshed regions and image (d) examples of general decision regions that the particular network can form.

Figure 18: Decision regions for a two-layer perceptron

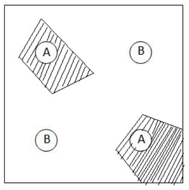


(a) Structure of the network (b) Exclusive or problem (c) Classes with meshed regions (d) Most general region shapes

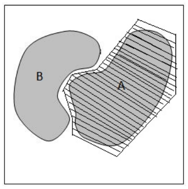
Figure 19: Decision regions for a three-layer perceptron



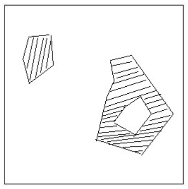
(a) Structure of the network



(b) Exclusive or problem



(c) Classes with meshed regions



(d) Most general region shapes

Two layer perceptrons As discussed before, single-layer perceptrons form half-plane decision regions in the input space. However, a two-layer perceptron instead forms a convex region, including convex hulls and unbounded convex regions, as illustrated at image (b) in figure 18. Such convex regions are formed by intersections of the half-plane region created by each node in the first layer of the multilayer perceptron. Each node in the first layer acts like a single-layer perceptron and places the "high" output points on one side of the hyperplane. The final decision region is created by a logical AND operation in the output node and is the intersection of all half-plane regions from the first layer. Thus, this final decision region formed by a two-layer perceptron is convex and has at most as many sides as there are nodes in the first layer.

This analysis leads to an insight to the problem of choosing the number of nodes in a two-layer perceptron. The number of nodes has to be large enough to form a decision region that is enough complex to solve the problem. Although, it must not be too large because the number of weights required should be reliably estimated from the available training data. An example is illustrated at image (b) of figure 18, where a hidden layer with two nodes solves the exclusive OR problem. However, there are no number of nodes that can solve the problem with the meshed class regions for the two-layer perceptron.

Three layer perceptrons The three-layer perceptron can form arbitrarily complex decision regions (where the complexity is limited by the number of nodes) which are also capable of separating meshed classes. Thus, it can generate disconnected non-convex regions and this is illustrated at figure 19. Therefore, at most three layers is needed to create perceptron-like feed-forward nets.

Moreover, it gives some insight to the problem of choosing the number of nodes in a three-layer perceptron. The number of nodes in the second layer must be greater than one in the cases when the decision regions are disconnected or meshed and cannot be formed by one convex area. In the worst case, the number of second layer nodes must be equal to the number of disconnected regions in the input distributions. The number of nodes in the first layer must generally be sufficient to create three or more edges on each convex area generated by every second-layer node. Typically, there should be more than three times as many nodes in the second layer as in the first [4].

Limitations Feed-forward layered neural networks are perhaps the simplest neuro-computational devices which have the ability of implementing any association between pairs of input-output patterns, provided that enough hidden units are present. Although, such networks, whose learning procedure is about solving a problem given a task using a given architecture, have shown to be computationally prohibitive. However, there is another class of learning procedures that instead focuses on building a network architecture proceeding from a given task and whose approach is to find networks close to the minimal size, but with acceptable learning times. In contrast to the learning procedures of feed-forward layered networks these ones do not focus on the error at all. I will in the next part present one such learning procedure, called Sequential learning.

2.2.2 Sequential learning

Consider a Perceptron with N input units, one output and a yet unknown number of hidden units, that is able to learn from any given set of input-output examples. Sequential learning then means to sequentially separating groups of patterns belonging to the same class from the rest of the patterns. This is done by successively adding hidden units into the network until the patterns that remain all belong to the same class. The internal representations created by these procedures are then linearly separable. I will in this chapter prove the existence of a solution for Sequential learning in Two layer perceptrons, but before I start I will explain the phenomenon of "the grandmother neuron".

The grandmother neuron Consider a one-layer perceptron with N input nodes and an unknown number of output neurons, where each input node either holds the value of 1 or -1. The aim is that exactly one neuron, S, should fire, i.e. produce the output $+1$, after having fed exactly one input vector x_i, $i = 1, 2, 3, \dots$ into the network. For an input vector looking like $(1, -1, -1, -1, 1, 1, -1, 1, \dots)$, the weights w_{ij} should be chosen to have exactly the same pattern $(1, -1, -1, -1, 1, 1, -1, 1, \dots)$. Now, if the signum function $\mathrm{sgn}(\sum w_i x_i - N)$ is used to calculate the output of the chosen neuron S_j it will be $+1$ because the sum consists of N terms that are either 1*1 or $(-1) * (-1)$. Thus, the total sum consists of N 1:s and subtracting N results in $\mathrm{sgn}(0) = 1$. However, if some of the x_i:s does not coincide with the given input then some of the terms in the sum will either be $1 * (-1)$ or $(-1) * 1$. Then $\sum w_i x_i - N < 0$ and $\mathrm{sgn}(\sum w_i x_i - N) = -1$.

Proof of linear separability

Proof. Let M be a finite set consisting of binary vectors and let $D = (x_1, x_2, x_3, \dots, x_k)$ be a subset of M. The aim is to construct a neural network with only one hidden layer that produces the output $+1$ if x_i belongs to D and -1 if x_i does not belong to D. This can be realized by constructing a grandmother neuron S_i for every $x_i \in D$. If we calculate the output-vector in the hidden layer $v = (S_1(x), S_2(x), \ldots, S_k(x))$, for a given input x, and if $x \notin D$, it will consist of k (-1)s: $v = (-1, -1, \ldots, -1)$. On the other hand, if $x \in D$ then $v = (-1, -1, \ldots, 1, \ldots, -1)$, that is one 1 at some position. Therefore, if we consider $\sum S_i(x) + k - 1$ we can see that the sum will be -1 for $x \notin D$ and $+1$ for $x \in D$.

$\square$

Remark: After this is performed, the input space will be partitioned into different regions formed by all patterns, x_i, where each region consists of at least one pattern all with the same target and identified by a single internal representation vector, v. Figure 16 demonstrates an example of a partition in the input space from a sequential algorithm. The circle corresponds to the input space and each hidden unit creates a straight line where the outputs are respresented as $+$ or $-$ on each side of it. Note that in this particular example there are nine internal representations but only five "excluding" clusters.

Limitations Linear separability for Sequential learning in two-layer perceptrons has now been proved. However, this proof does not say anything about the ability of the algorithm to capture the correlations between the presented patterns. In a worst case scenario each region would only contain one pattern, x_i, which means that none of the correlations between the presented patterns will be captured. However, in practice, the purpose with this Sequential learning algorithm is that each region should cluster several patterns. There is, for instance, a method that at each step chooses a weight vector that excludes the maximal number of patterns with the same target [6]. However, the implementation of this algorithm is not something that I will discuss further in this report. Instead, I will focus on the so called Backpropagation algorithm, a feed-forward neural network which uses a different type of threshold function, namely a Sigmoid function.

2.3 The Backpropagation algorithm - a continuous non-linear model

2.3.1 Introduction

As already demonstrated in the perceptron convergence procedure, the decision boundaries would oscillate forever in the case when the inputs are non-separable and the distributions overlap. Thus, it would fail to converge. However, for practical purposes the perceptron algorithm can be modified into the "Least mean square" algorithm or, in short form, the LMS algorithm. This algorithm minimizes the mean square error, determined by the difference between the desired output and the actual output, in a perceptron-like net. An essential difference with Rosenblatt's perceptron and the LMS algorithm is that Rosenblatt used

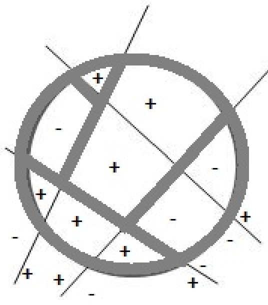


Figure 20: An example of the partition of the input space created by a sequential algorithm. The input space is represented by the large circle and each straight line represents a hidden unit, whose outputs are marked out as a $+$ or $-$ on each side of the line.

a hard limiting nonlinearity, whereas the LMS algorithm makes this hard limiter linear or replaces it by a threshold-logic nonlinearity. The Backpropagation algorithm is a generalisation of the LMS algorithm and it uses a gradient descent technique [4] in order to search the hypothesis space of possible weight vectors to find the weights that would best fit the training examples [2]. This method is also called *Back-propagation of error* and its technique is basically about sending the errors backwards through the network [1]. Instead of adjusting the weights according to the perceptron learning rule, which corresponds to equation (3) [4], this algorithm uses a second training rule called *the delta rule* [2]. This rule is applied after every iteration until the algorithm [4] converges towards a best-fit approximation to the target concept.

Gradient descent In order to explain the gradient descent technique, let us consider the two dimensional case. Consider an error function, $E(w_0, w_1)$, which we want to minimize. Now, the question is how do we find the (w_0, w_1) for which the error is minimal, starting at an arbitrary point, (w_0, w_1)? The answer is by going in the direction of the negated gradient $-(\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1})$. This means changing the weight w_0 by a multiple of $-\frac{\partial E}{\partial w_0}$. An example of the error surface is plotted in figure 21. Its axes w_0 and w_1 represent the values of the two weights in a simple linear unit, that is an unthresholded perceptron. The vertical axes represents the value of E depending on a fixed set of training examples. The arrow in figure 21 shows the negated gradient, i.e. the direction of the steepest decrease of the error, E, at a certain point on the w_0,w_1 plane. The error surface forms a parabolic shape with a single global minimum [2].

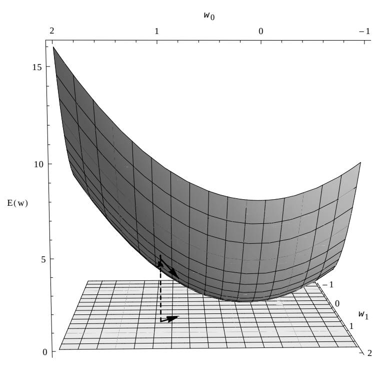


Figure 21: An Error surface in two dimensions

Sejnowski The Backpropagation algorithm has been shown to be capable of solving a number of deterministic problems within areas like speech synthesis and recognition and visual pattern recognition. It has been shown to perform well in most cases by finding good solutions to different problems. One of the first who demonstrated the power of this algorithm was Sejnowski, who trained a two-layer perceptron to form letter to phoneme transcription rules. As the input to the network he used a binary code that indicated the letters in a sliding window with a width corresponding to seven letters that moved over a written transcription of spoken text. The target output corresponded to a binary code that indicated the phonemic transcription of the letter at the center of the window [4].

2.3.2 Going forwards

As with the perceptron, the training of the MLP consists of two parts. The first one is called ”going forwards” and calculates the outputs from the given inputs by using the current weights. Part two is called ”going backwards” and updates the weights according to the output error through a function that computes the difference between the outputs and the targets. Before I start to describe these two phases further, I will list some of the notations that I am going to use throughout this chapter:

i is an index running over the input nodes, j the hidden layer neurons, k the output neurons, whereas v_{ij} denotes the first layer weights and w_{jk} the second layer weights. The activation function (which determines the output of a neuron) used in this algorithm will be denoted as g. Further, the input and output of neuron j in the hidden layer are denoted as h_j respectively a_j, whereas the input and output of neuron k in the output layer are denoted as h_k respectively y_k. In the continuation, when referring to the notation a_j, I will use the word activation instead of output.

The going forwards phase calculates the outputs of the neurons basically in the same way as the perceptron. The only difference is that the calculation has to be performed several times, once for each set of neurons or, in order words, layer by layer. As the MLP works forwards through the network, the activations of one layer of neurons will correspond to the inputs to the next layer.

The Network Output To explain this further, have a look at figure 14 again. The MLP then starts from the left in the figure by feeding the input values, x_i, to the network. These inputs calculate the activations of the hidden layer, a_j, by multiplying them with the first layer of weights, v_{ij}, such that $a_j = g(h_j) = g(\sum_i v_{ij} x_i)$. Moving on to the next step, these activations, a_j, compute the activations of the output layer, y_k, by multiplying them with the next layer of weights, w_{jk}, such that $y_k = g(h_k) = g(\sum_j w_{jk} a_j)$. Thus, the output of the network is a function of the following two variables:

- the current input, $\mathbf{x}$
- the weights of the first layer, $\mathbf{v}$, and of the second layer, $\mathbf{w}$

The values of these computed output neurons, y_k, will in turn be compared to the targets, t_k, in order to determine the error.

2.3.3 Going backwards: Backpropagation of Error

The Error of the Network Thus, the purpose of the learning rule for the MLP is, as for the perceptron, to minimise an error function. However, as more layers have been added to the network it cannot use the same error function as the perceptron, which was $E = \sum_{i=1}(t_i - y_i)$. Now, that there is more than one layer of weights, it has to find out which ones that caused the error; the weights between the input layer and hidden layer or the weights between the hidden layer and the output layer? (In cases with more than one hidden layer it could also be the weights between two such layers.) Another reason why the MLP cannot use the perceptron error function is that the errors for the different neurons may have different signs and thus, summing them up would not result in a realistic value of the total error. To overcome this issue, there is a function called the *Sum-of-squares* that calculates the difference between the target, t, and the output, y, for each node, squares them and adds them together:

$$E(t, y) = \frac{1}{2} \sum_{k=1}^{n} (t_k - y_k)^2. \tag{33}$$

The reason why the term $\frac{1}{2}$ has been added to the function is to make it easier to differentiate, which is exactly what the algorithm will do as it uses the gradient descent method. After having computed the errors, the next step for the algorithm to take is to adjusting the weights, in the purpose of producing a firing or non-firing neuron according to the target. The gradient of the Sum-of-squares function reveals along which direction the error increases and decreases the most and it can be computed by differentiating the function. Since the purpose is to minimise the error, the direction that the algorithm wishes to take is downhill along the graph of the Sum-of-squares function.

The Sum-of-squares function has to be differentiated with respect to a variable and there are two variables that vary in the network during training, namely the weights and the inputs. Although, the only variable that the algorithm have the possibility to vary during training, in order to improve the performance of the network, is the weights. Therefore the function will be differentiated with respect to them and it can be written as $E(\mathbf{v}, \mathbf{w})$. As the weights vary, the output value will change which in turn would change the value of the error. Thus, the sum-of-squares function can be expressed as

$$E(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{N}(t_k - y_k)^2 = \tag{34}$$

$$= \frac{1}{2}\sum_{k}\left[t_k - g\left(\sum_{j} w_{jk}a_j\right)\right]^2 \tag{35}$$

In equation (35), the inputs from the hidden layer neurons, a_j, and the second-layer weights, w_{jk}, are used to decide on the activation of the output neurons, y_k. To explain this further, let's get back to the algorithm of the singlelayer perceptron. Let the activation of a neuron this time be $\sum_j w_{jk}x_j$ instead of one or zero and replace equation (35) with

$$E(\mathbf{w}) = \frac{1}{2}\sum_{k}\left[t_k - \sum_{j} w_{jk}x_j\right]^2, \tag{36}$$

where x_j is an input node. The algorithm adjusts the weights, w_{jk}, in the direction of the gradient of $E(\mathbf{w})$. Differentiating the error function with respect to the weights results in

$$\frac{\partial E}{\partial w_{ik}} = \frac{\partial}{\partial w_{ik}}\left(\frac{1}{2}(t_k - y_k)^2\right) = \tag{37}$$

$$= \frac{1}{2}2(t_k - y_k)\frac{\partial}{\partial w_{ik}}\left(t_k - w_{jk}x_j\right), \tag{38}$$

where $\frac{\partial t_k}{\partial w_{ik}} = 0$, as t_k is not a function of w_{ik}. Thus, the only term which depends on w_{ik} is one corresponding to $i = j$, that is w_{jk} itself, which means that:

$$\frac{\partial E}{\partial w_{ik}} = (t_k - y_k)(-x_i) \tag{39}$$

Now, the aim of the weight updating rule was that the gradient should go downhill. Therefore a negative sign should be added before the gradient, which makes it possible to express the learning rule as

$$w_{ik} - \eta \frac{\partial E}{\partial w_{ik}} \to w_{ik}. \tag{40}$$

The expressions (39) and (40) together completes the expression of the learning rule:

$$w_{ik} + \eta(t_k - y_k)x_i \to w_{ik}. \tag{41}$$

Note that this equation is not identical with equation (3) as the output y_k is computed differently. Until now, the fact that the perceptron uses the none differentiable threshold function have been ignored. However, in order to use the gradient descent method the output must be differentiable and so must the activation function. Thus, the next step would be to finding such an activation function.

A suitable Activation function A suitable activation function should have the three following properties.

- It has to be differentiable, so that its gradient can be computed
- It has to saturate, that is, become constant at the ends of the range so that the neuron could fire or not
- It should change fairly quickly in the middle, in order to reach its saturation values fast

There is a family of S-shaped functions called the sigmoid functions that satisfies these criterions and they are suitable as activation functions, due to their S-shaped forms reminding of the form of the threshold function. The aim with the algorithm is, after all, that the units should act like neurons, that is to fire or not to, but at the same time to vary continuously. An example of a sigmoid curve is illustrated in figure 17 and the most commonly used sigmoid function for the backpropagation algorithm is the one expressed as

$$a = g(h) = \frac{1}{1 + \exp(-\beta h)}, \tag{42}$$

where β is some positive number. Thus, with the sum-of-squares function and a sigmoid function as activation function, it is now possible to differentiate the first one and adjust the weights in order to decrease the error function. An advantage with the sigmoid functions is that its derivatives have a suitable form for the purpose of the backpropagation algorithm. They are computed as

$$g'(h) = \frac{dg}{dh} = \frac{d}{dh}(1 + e^{-\beta h})^{-1} = -1(1 + e^{-\beta h})^{-2} \frac{de^{-\beta h}}{dh} =$$

$$= -1(1+e^{-\beta h})^{-2}(-\beta e^{-\beta h}) = \frac{\beta e^{-\beta h}}{(1+e^{-\beta h})^2} = \beta g(h)(1-g(h)) =$$
$$= \beta a(1-a) \tag{43}$$

For the moment the parameter β will be ignored as it is a matter of scaling. Now that a suitable activation function has been found, the adjustment procedure of the weights needs to be worked out.

Backpropagation of error To sum up, until now the inputs have been fed into the network, the decision of which neurons to fire and not have been made and the errors have been computed, using the sum-of-squared difference between the targets and the outputs. The next step is to find a gradient of the errors that indicates how each weight should be updated. The algorithm starts with the neurons in the output layer and then moves backwards through the network until the input layer has been reached. However, there are two problems;

- The values of the inputs to the output neurons are unknown
- The targets for the hidden neurons are unknown (if there were more than one hidden layer neither their inputs nor their targets would be known)

However, this problem can be solved by using the chain rule of differentiation. In order to find out how the error changes as the network varies the weights, consider how the error changes as the inputs to the weights vary, and multiply this by the change of the input values as the network varies the weights. Differentiating the error function $E(\mathbf{w})$ gives the expression

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial h_k}\frac{\partial h_k}{\partial w_{jk}}, \tag{44}$$

where h_k is the sum of the activations from the hidden layer neurons, a, and the second-layer weights, w, such that $h_k = \sum_l w_{lk} a_l$. h_k is at the same time the *input* to the output layer neuron k. Equation (44) says that it is possible to know how the output error changes as the second-layer weights vary, $\frac{\partial E}{\partial w_{jk}}$, since it is possible to know how the error changes as the input to the output neurons varies, $\frac{\partial E}{\partial h_k}$, and how the input values change as the weights vary, $\frac{\partial h_k}{\partial w_{jk}}$.

Further, by expressing the activations of the output nodes in terms of the activations of the hidden nodes and the output weights, such that $y_k = g(\sum_j a_j w_{jk})$, then the error calculations can be sent back through the network to the hidden layer in order to determine what the target outputs were for those neurons. This can be expressed as

$$\frac{\partial E}{\partial v_{ij}} = \frac{\partial E}{\partial h_j}\frac{\partial h_j}{\partial v_{ij}}, \tag{45}$$

where v_{ij} corresponds to the first layer weights. (Note that exactly the same computations can be performed if the network has extra hidden layers between the inputs and the outputs). Thus, the gradients of the errors can be computed with respect to the weights, so that the weights can be changed in order to decrease the error function. Although, this cannot be done directly, as the differentiation is done with respect to variables that are known. This leads to two different update functions, one for each of the sets of weights, that should be applied backwards through the network starting at the neurons in the output layer and ending up at the input layer. These two learning rules can be expressed as

$$w_{jk} - \eta \frac{\partial E}{\partial w_{jk}} \rightarrow w_{ik} \tag{46}$$

$$v_{ij} - \eta \frac{\partial E}{\partial v_{ij}} \rightarrow v_{ij}. \tag{47}$$

In order to complete the learning rules, the terms $\frac{\partial E}{\partial w_{jk}}$ and $\frac{\partial E}{\partial v_{ij}}$ need to be computed. I will start off with the first one by calculating the second factor of equation (44), leading to

$$\frac{\partial h_k}{\partial w_{jk}} = \frac{\partial \sum_l w_{lk} a_l}{\partial w_{jk}} = \sum_l \frac{\partial w_{lk} a_l}{\partial w_{jk}} = a_j \tag{48}$$

where the condition $\frac{\partial w_{lk}}{\partial w_{jk}} = 0$ for all l except from when $l = j$ has been used. Now, the first factor of equation (44), also called the delta term, $\frac{\partial E}{\partial h_k}$, remains to be computed. It will in the continuation have the notation $\delta_o = \frac{\partial E}{\partial h_k}$, which together with equation (48) gives the expression $\frac{\partial E}{\partial w_{jk}} = \delta_o a_j$. Now, as the input values of a neuron in the output layer are unknown, it is not possible to compute the error of the output directly. Therefore, the chain rule should be applied again, which leads to the expression

$$\delta_o = \frac{\partial E}{\partial h_k} = \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial h_k}, \tag{49}$$

where the output of the k:th neuron in the output layer can be described as

$$y_k = g(h_k^{output}) = g\left(\sum_j w_{jk} a_j^{hidden}\right). \tag{50}$$

To avoid confusion, I've started to mark out whether h refers to the *input* of an output layer neuron or a hidden layer neuron. Further, developing the delta term for the output layer, δ_o, leads to

$$\delta_o = \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial h_k} = \frac{\partial E}{\partial g(h_k^{output})} \frac{\partial g(h_k^{output})}{\partial h_k^{output}} = \tag{51}$$

$$= \frac{\partial E}{\partial g(h_k^{output})} g'(h_k^{output}) = \tag{52}$$

$$= \frac{\partial}{\partial g(h_k^{output})} \left[\frac{1}{2} \sum_k (g(h^{output}) - t_k)^2 \right] g'(h_k^{output}) = \tag{53}$$

$$= (g(h_k^{output}) - t_k) g'(h_k^{output}) = \tag{54}$$

$$= (y_k - t_k) g'(h_k^{output}), \tag{55}$$

where $g'(h_k^{output})$ can be computed as $g'(h_k^{output}) = y'_k = y_k(1 - y_k)$ by using equation (43). Note that in equation (53), the error at the output is replaced by expression (35). Now, the delta term in the output layer can be written as

$$\delta_o = (y_k - t_k) y_k (1 - y_k) \tag{56}$$

Thus, the differentiated error function in the learning rule for the second layer weights (46), may be expressed as

$$\frac{\partial E}{\partial w_{jk}} = \delta_o a_j = (y_k - t_k) y_k (1 - y_k) a_j. \tag{57}$$

Finally, the learning rule for the second layer weights can be expressed as

$$w_{jk} - \eta (y_k - t_k) y_k (1 - y_k) a_j \to w_{jk} \tag{58}$$

or

$$w_{jk} - \eta \delta_o a_j \to w_{jk}. \tag{59}$$

Let's now move on to the first layer of weights, v_{jk}, which connects the input nodes with the hidden nodes. Remember that the algorithm is moving backwards through the network. The first step to take is to compute the delta term of equation (45), $\frac{\partial E}{\partial h_j}$, denoting it as δ_h.

$$\delta_h = \frac{\partial E}{\partial h_j^{hidden}} = \sum_k \frac{\partial E}{\partial h_k^{output}} \frac{\partial h_k^{output}}{\partial h_j^{hidden}} = \tag{60}$$

$$= \sum_k \delta_o \frac{\partial h_k^{output}}{\partial h_j^{hidden}} \tag{61}$$

During the calculation in (60), k runs over the output nodes and in order to reach the step (61), where the notation $\delta_o = \frac{\partial E}{\partial h_k}$ has been used. Before I continue, remember that the inputs of the output layer neurons come from the activations of the hidden layer neurons multiplied by the second layer weights, so that

$$h_k^{output} = \sum_l w_{lk} a_l = \sum_l w_{lk} g(h_l^{hidden}) = g\left(\sum_l w_{lk} h_l^{hidden} \right). \tag{62}$$

This brings about the calculations

$$\frac{\partial h_k^{output}}{\partial h_j^{hidden}} = \frac{\partial g\left(\sum_l w_{lk} h_l^{hidden}\right)}{\partial h_j^{hidden}} = w_{jk} g'(h_j^{hidden}), \tag{63}$$

where the condition $\frac{\partial h_l}{\partial h_j} = 0$, for all l except when $l = j$, has been used in the last step. Further, the computations

$$w_{jk} g'(h_j^{hidden}) = w_{jk} \cdot g(h_j^{hidden})(1 - g(h_j^{hidden})) = w_{jk} a_j (1 - a_j), \tag{64}$$

result in the fact that δ_h could be expressed as

$$\delta_h = a_j(1 - a_j) \sum_k \delta_o w_{jk}. \tag{65}$$

Note that by differentiating the inputs to the hidden neurons with respect to the first layer weights, the result would be the input values:

$$\frac{\partial h_j}{\partial v_{ij}} = \frac{\partial\left(\sum_l v_{lj} x_l\right)}{\partial v_{ij}} = \frac{\sum_l \partial(v_{lj} x_l)}{\partial v_{ij}} = x_i \text{ as } \frac{\partial v_{lj}}{\partial vij} = 0 \text{ for all } l \text{ except } l = j.$$

To sum up, the following conditions have been established:

$$\frac{\partial E}{\partial v_{ij}} = \frac{\partial E}{\partial h_j} \frac{\partial h_j}{\partial v_{ij}}, \ \frac{\partial E}{\partial h_j} = \delta_h \text{ and } \frac{\partial h_j}{\partial v_{ij}} = x_i.$$

Finally, the differentiated error function, $\frac{\partial E}{\partial v_{ij}}$, of the learning rule for the first layer of weights, v_{ij}, can be computed as

$$\frac{\partial E}{\partial v_{ij}} = \delta_h x_i = \left(\sum_k \delta_o w_{jk} a_j (1 - a_j)\right) \cdot x_i = a_j(1 - a_j)\left(\sum_k \delta_o w_{jk}\right) x_i, \tag{66}$$

which means that the learning rule for the first layer of weights (47) can be expressed as

$$v_{ij} - \eta a_j (1 - a_j) \left(\sum_k \delta_o w_{jk}\right) x_i \to v_{ij} \tag{67}$$

or simply

$$v_{ij} - \eta \delta_h x_i \to v_{ij}. \tag{68}$$

Remark:

Note that it is possible to do the same computations with extra hidden layers added between the input layer and the output layer.

2.3.4 The learning algorithm

The MLP learning algorithm can be summed up by dividing it into the different parts Initialisation, Training and Recognition:

- Initialisation
  - Set the values of the weights to small randomly chosen numbers
- Training
  - for each input vector do the following:
    - ∗ Forwards phase:
      - · Calculate the activation of each neuron j in the hidden layer(s) with the following equations:

$$h_j = \sum_i x_i v_{ij} \tag{69}$$

$$a_j = g(h_j) = \frac{1}{1 + exp(-\beta h_j)} \tag{70}$$

      - · Move forward in the network and calculate the activations of the output layer neurons, by using:

$$h_k = \sum_j a_j w_{jk} \tag{71}$$

$$y_k = g(h_k) = \frac{1}{1 + exp(-\beta h_k)} \tag{72}$$

    - ∗ Backwards phase:
      - · Compute the delta term of the output using the equation:

$$\delta_{ok} = (y_k - t_k) y_k (1 - y_k) \tag{73}$$

      - · Compute the delta term of the hidden layer(s) using:

$$\delta_{hj} = a_j (1 - a_j) \sum_k w_{jk} \delta_{ok} \tag{74}$$

      - · update the output layer weights

$$w_{jk} - \eta \delta_{ok} a_j \rightarrow w_{jk} \tag{75}$$

      - · Update the hidden layer weights:

$$v_{ij} - \eta \delta_{hj} x_i \rightarrow v_{ij} \tag{76}$$

    - ∗ Randomize the input vectors into a different order so that they do not get trained in the exactly same order for each iteration

    - continue until the training stops, that is when the error becomes less than its prescribed limits or when a given maximum number of iterations has been reached

- Recognition
    - Use the Forwards phase again

2.3.5 Initialising the weights

Up to this point, I have claimed that the initial weights of the MLP algorithm should be small, randomly chosen, values. This can be explained further by looking at the shape of the sigmoid function in figure 17. If the initial weights were close to 1 or -1, i.e. large in this case, this would cause the sum of the weighted inputs to be close to ± 1 and consequently the sigmoid function will saturate to its maximum or minimum value. Thus, the output of the neuron would receive the value zero or one. On the other hand, if the initial weights were close to zero, it would mean that the sum of the weighted inputs also would be close to zero. Moreover, as the sigmoid function is approximately linear at such values, the output of the neuron would be an approximately linear model. However, it is the duty of the network to decide for itself which way to take and consequently the initial values of the weights should be chosen to be somewhere in between "large" and "close to zero".

2.3.6 Local Minima

As discussed above, the force of the learning rule is to try to minimize the error of the network, by approximating the gradient of the error and following it downhill until a minimum is reached. However, there are no guarantees that the global minimum will be found, but it can also be a local one. The Gradient descent works in the same way for two or more dimensions. One of the problems with this method is that efficient downhill directions are hard to compute locally. The algorithm does not know where the global minimum is or what the error landscape looks like, but it can only compute local features proceeding from the point where it is located at the moment. Thus, the minimum that the algorithm finds depends on where it starts. If it begins near the global minimum, then that is where it is most likely to end up. On the other hand, if it starts near a local minimum it would most probably end up there. How long it will take to find a minimum depends on the exact appearance of the landscape at the current point. This issue is demonstrated in figure 22 [1].

Within the classification area, the problem caused by ending up in a local minima is that the network would cluster two or more disjoint class regions into one [4]. However, despite the fact that convergence towards a global minimum error is not guaranteed, Backpropagation has shown to be a successful approximation method in practice. In practical applications, the problem of getting stuck in a local minima has turned out not to be as severe as one might

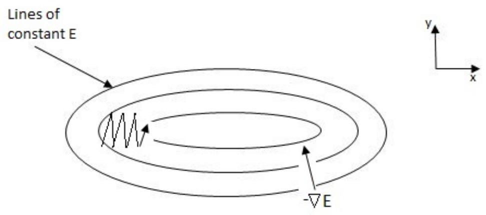


Figure 22: An error landscape in two dimensions which demonstrates different paths that the gradient can take in order to find a minimum. How long it will take to find it depends on the exact appearance of the landscape at the current point. This is illustrated by the two arrows within the landscape.

think for a reason that I will now explain. As each weight corresponds to one dimension, networks with a large number of weights has error surfaces in high dimensional spaces. When the gradient descent finds a local minimum with respect to one of the weights, this point is not neccessarily a local minimum with respect to the other weights. The larger number of weights in the network, the more dimensions are there that can provide an ”escape route” for the gradient descent leading away from a local minimum with respect to a single weight.

Another essential aspect of the local minima is the behavior of the weights as the number of iterations increases. If the weights are initialized to be close to zero, then during the early gradient descent steps the network will form a smooth function which is approximately linear in its inputs, as I have already explained in the previous chapter. Only as the weights have had time to grow, they have come to a stage when they can represent highly nonlinear network functions. The existence of local minimas in the region of the weight space representing these more complex functions does not necessarily have to be a problem, though. The reason for that is that by the time the weights have come to this stage they have hopefully moved close enough to the global minimum, such that also local minimas in this region would be acceptable.

Despite of the reasoning above, gradient descent within complex error surfaces used for artificial neural networks is poorly understood. There are no methods to determine when getting stuck in a local minima will cause difficulties and when it will not. However, there are a some heuristics to avoid local minimas and to make it more likely to find the global minimum [2], like for instance adding more hidden units, lowering the learning rate [4], trying out several different starting points by training several different networks or adding a term to the weight-update rule, called the Momentum term [2].

2.3.7 Generalisation and overfitting

The whole purpose with the neural network is that it should generalise by using training examples from all the inputs. The network must be exposed to enough training before it is able to generalise. However, there is a risk in training the network for too long. As every weight in the network can be varied, the variability of the network is huge and should therefore be trained with caution. Training it for too long would result in overfitting the data, which means that the network would learn about the noise and inaccuracies of the data as well as the actual function. Then the model that has been learnt would be too complicated and it has not generalised well.

2.3.8 Number of hidden layers

There are two things that need to be taken into consideration when choosing the number of weights for the network and that is the number of hidden nodes and hidden layers. These choices are fundamental in succeeding with the application of the Backpropagation algorithm [1]. I will now present some theory treating the area.

Cybenko Consider a network of continuous valued units with the activation function $g(u) = \frac{1}{1+\exp(-u)}$ for the hidden units and $g(u) = u$ for the output units. Let this network implement a set of functions $y_i = F_i[x_k]$ where the input variables are represented by x_k, where $[x_k]$ means $x_1, x_2, \ldots, x_N$, and where y_i represents the output variables. The calculation for such a network with no hidden layer would then be

$$y_i = \sum_i w_{ik} x_k - \theta_i$$

and with one hidden layer

$$y_i = \sum_j W_{ij} g\left(\sum_k w_{jk} x_k - \phi_j\right) - \theta_i,$$

and so on. θ_i and ϕ_j represent the thresholds in this case. Now, the question is: How many hidden layers are needed in order to *approximate* a particular set of functions $F_i[x_k]$ to a given accuracy? The answer is at most two hidden layers with arbitrary accuracy beeing received, given that there is enough units per layer. This was proved by Cybenko in 1988. He also proved, in 1989, that only one hidden layer is enough to approximate any *continuous* function. Although, the utility of these proofs depends on the number of hidden units which is not known in general. However, in many cases this number grows exponentially with the number of input units [3].

Consequently, at most two hidden layers are necessary for the MLP in order to find a solution. As there is no theory which treats the number of hidden

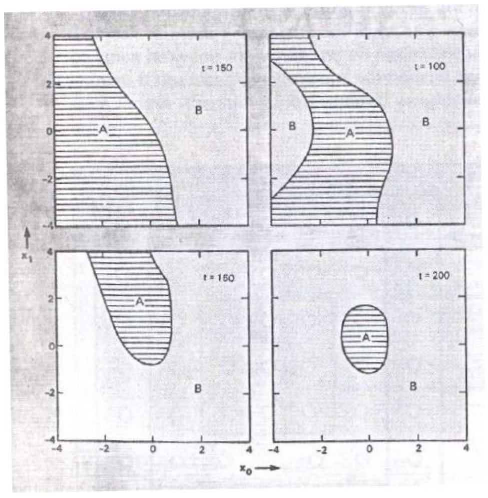


Figure 23: An example of decision regions formed after 50, 100, 150 and 200 iterations created by a two layer perceptron when using the Backpropagation algorithm. This image is taken from [4].

nodes one must experiment with different numbers until a result of satisfaction is achieved [1].

2.3.9 Decision regions

I have already treated the decision regions of multi-layer perceptrons which use hard limiting nonlinearities and only have one output. Their behavior is similiar to multi-layer perceptrons which instead use sigmoidal nonlinearities and have multiple output nodes, and whose way of deciding the class at the recognition part is by choosing the output node with the largest output. However, the behavior of these nets is more complex, because the decision regions are bounded by smooth curves instead of straight line segments. This makes them more difficult to analyse. I will now demonstrate an example of the decision regions formed by a two-layer perceptron.

An example Figure 23 shows an example of how the decision regions can behave when applying a sigmoidal nonlinear model, or more specifically the backpropagation algorithm. The network that has been used here is a two-layer perceptron consisting of two inputs, one hidden layer with eight nodes and an output layer with two nodes. The two different classes that have been classified, denoted as A and B, have been presented on alternate trials and the desired outputs have the value one or zero. The decision region for samples belonging to class A constitutes of a circle with radius one situated at the origin of the inputspace. The initial decision region was a slightly curved hyperplane as can be seen in the picture. It changes gradually as the algorithm runs and eventually, after 200 trials, becomes the circular region that encloses the circular distribution of class A. In this example, 100 patterns from each class have been fed into the network [4].

2.3.10 Limitations

Some limitations that we have seen so far with the Backpropagation algorithm are the risk of overfitting by training the network for too long and the risk of getting stuck in a local minima. Further, the fact that the number of hidden nodes is unknown could also be considered as a limitation as one must experiment with the number until a satisfying result is reached. Another difficulty with the Backpropagation algorithm is that it often requires a very large number of presentations of training data in order to reach convergence which consequently lowers the speed of it [4].

3 Implementation

3.1 Introduction

Today, neural networks are seen as an alternative to "old fashioned" programming [5]. They are often well suited to problems that people are good at solving, but for which traditional computing methods are not [12]. Such methods can solve problems that we have already understood how to solve, whereas artificial neural networks can solve problems that we do not exactly know how to solve [11]. Further, they can solve problems that are too complex for conventional technologies, that is problems that do not have an algorithmic solution or for which an algorithmic solution is too complex to be found [12]. An advantage using neural networks is their ability to adapt, learn, cluster, organize [11] and generalize in making decisions based on imprecise input data [12]. As neural net models use massively parallel nets composed of many computational elements, they can explore many competing hypothesis simultaneously rather than performing a program of instructions sequentially. This is useful in areas such as speech- and image recognition, where many hypothesis are pursued in parallel and consequently, high computation rates are required. They have a great degree of robustness or fault tolerance due to the many processing nodes in the network, where each of them has primarily local connections. A damage on a few nodes or links would not necessarily affect the overall performance of the network [4]. Two other advantages using neural networks within image recognition are the extensibility of the system, that is their ability to recognize more patterns than initially defined, and the fact that the code is simpler when compared to other computing methods.

Training patterns are used in the purpose of teaching the neural network to recognize the images. They consist of two single-dimensional arrays of float numbers - Inputs and Outputs arrays. The Inputs array contains the input data, that is a digitized representation of the character's image [12]. The effort spent in training is to force the neural network to see intrinsic characteristics in the training set. Almost any neural network can be trained so well it might not encounter any errors at all on the training set. But the question is

How well will the neural network perform on patterns that it has never seen before?

This is what the concept of generalisation is about [10].

3.2 Description

Problem formulation The company Pepto systems has an invoicing system where customers print documents via a virtual printer to a server system placed at the company. In this system the printout is interpreted and transformed into a digital format by applying a customer specific template. In order to interpret the print stream which is fed into the system, a computer language called Postscript is used. However, this method has limitations, especially when it comes to special- and unicode signs (where unicode is a computing industry standard for the consistent encoding, representation and handling of text expressed in most of the world's writing systems [19]).

In order to solve this problem, the company gave me the assignment of implementing a method which could recognize signs without the use of char information from Postscript (where char is a character type and enumeration whose values represent Unicode characters [22]). I was asked to create a program which could learn how to recognize digital signs by using the Backpropagation algorithm. The idea for the program was to train itself on signs with a certain font and then try to recognize new signs, that it had never seen before, all with the same font. As their system is developed in Delphi, the solution should be implemented in the same language.

Method To determine the values in the inputlayer, an image with a sign is transformed into a gray scale. Every pixel of this image receives a value depending on the colour of it. If it is black, it receives the value one and if it is white it receives the value zero. These pixels become the values of the inputlayer, by attaching all the rows of the pixels into an inputvector. The initial values of the weights are within the interval $w = [-0.05, 0.05]$, the learning rate that has been picked is $\eta = 0.2$ and the activation function that is beeing used is the sigmoid function $\frac{1}{1+exp(-x)}$. The target is known by navigating all patterns belonging to a particular sign to the same output node. This output node should have the value one when feeding these patterns into the network, while the rest of the nodes are given the target value zero for the same patterns. Thus, the number of nodes in the output must correspond to the number of unique signs.

During training, for each iteration, the error is beeing checked until it reaches the maximum error, which is set to be 1.0. If it never reaches the maximum error the training continues until the maximum iteration 10000 is reached. When the training phase is finished, the updated weights will be saved. At the recognition procedure, new patterns are beeing fed into the network through the forwards phase once by using these saved weights. The output node which computes the highest value, when feeding a specific pattern into the network, corresponds to the answer of the network, that is the sign that it will guess that this pattern belongs to.

Structure The network that I have created has an inputlayer with 6534 nodes, determined in a way I will soon describe. I have chosen to include only one hidden layer in the network architecture. As the patterns have the same font, the variation of them are limited and consequently the network should only need to form decision regions of a simpler kind. The difference in their looks conerns the size and if they have bold or italic styles. When it comes to deciding on the number of hidden nodes, this is something that I will discuss more thoroughly later on. The training set that I have been using has in total 192 different patterns, whereof 60 of them are unique signs, which means that there are 60 neurons in the output layer (as I have explained before, the number of unique signs has to be the same as the number of output nodes). Each sign had at least one and at most five patterns represented in the training set, but with an average of approximately 3 patterns per sign. When choosing the training set I tried to pick signs with as different looks as possible. Therefore, the signs with more varied look had a higher number of representatives in the training set than the ones with very similiar looks.

Adaptive learning There might be cases when the learning rate, η, will be to high, meaning that it might take too big steps each time, as it approaches the minimum, so that it will not be able to get closer and reach it. Therefore, there is a function called adaptive learning, which is optional to use during training, whose purpose is to prevent this from happening. It basically compares the current error with the previous one and adjusts the learning rate according to this comparison.

Assumptions Before training the patterns, there are a few interventions that I have made for different reasons:

- Centering the sign, in order to match as many pixels as possible.
- Erasing as much dirt as possible that should not be there, so that the network will learn only about what is essential which is the sign itself. (This dirt could for example come from the cutting process, when parts of other signs could be included in the image.)

- All the signs that are identical to any sign in the training set are beeing excluded from it. That is done to prevent the network from favoring one particular look of a sign and to improve generalisation.

- To determine the number of nodes in the input layer, the bitmap with the largest height and the one with the largest width is searched in the training set. Five pixels are added to each one of these values and then the two factors are multiplied by each other, such that (largest width + 5) · (largest height + 5) = number of input nodes. All bitmaps in the training set are then magnified to this size, by adding white pixels to the edges. This is done, because all signs need to have the same amount of elements in the input layer for the network to work. The reason why extra pixels are added to the largest height and width, is to prevent from loosing information at the recognition procedure. Imagine, if a bitmap of a pattern that was to be recognized was larger then the trained network, there would not be enough elements in the input vector to fit all information.

- As no patterns containing exclusionary white pixels will be used in this network, no bias unit has been included.

Parts of the code are shown at the appendix.

3.3 Hypothesis, results and comments

Choosing the training set In order to make the network generalize as well as possible, for every sign I have picked images with as many different looks as possible to the training set. If H_1 is training set and subset of the set C_1 of a particular sign, where C_1 represents all possible looks of that sign, then I want to spread out the points in the input space as much as possible. In this case C_1 is finite because the input vectors that are beeing fed into the network are binary vectors consisting of a limited amount of elements. I want the network to form decision regions which are as large as possible, that is as close looking to the set C_1 as possible. When new patterns are beeing fed into the network after the training process in order to be recognized, it is more likely for the network to make the right guesses if the decision region formed at the firing output neuron for this sign is close looking to the set C_1.

Another aspect that I would like to add into the discussion about choosing the training set is the time that it takes to train a network. According to the theory, one limitation of the Backpropagation algorithm is the requirement of large training sets and a thereto a low speed of convergence. Then, my thought is that by using as little amount of patterns in the training set as possible, this problem would be smaller as the network to take less time to train. If, for instance, two patterns are very similiar, than only one of them should be chosen in order to teach the network something new rather than something that it already has seen. From my point of view, this method would make the network more efficient. However, as my program trains and recognizes digital signs with the same font, there are fewer possible looks that each sign could have, compared to for instance all digital signs or handwritten signs. Therefore my network will not require such a large training set as others might do and consequently this limitation will not effect my case that much.

Choosing the number of hidden nodes When it comes to choosing the number of nodes, I have been thinking of the risks and advantages which should be considered when decreasing or increasing the number of nodes.

Using too few hidden nodes A network with few hidden units could have a problem finding an XOR as the decision regions formed by such a network will be less complex compared to a network with a high number of hidden nodes. Consequently, it might not converge at all. My interpretation of the theory is that another disadvantage with using too few hidden units could be that the risk of local minima increases. According to the theory, as the number of weights increases the number of dimensions also increases which means that there are more "escape routes" that can help to avoid getting stuck in a local minima. Thus, decreasing the number of hidden nodes would mean that there are fewer such "escape routes" that helps avoiding local minimas. However, an advantage decreasing the number of hidden nodes should be that the ability to capture more correlations between the patterns, as more patterns will be clustered together. Consequently, the generalisation should improve.

Using too many hidden nodes Using a higher number of hidden nodes should increase the chance of finding an XOR, as such a network should be able to form more complex decision regions. Thus, the chance of reaching convergence should be improved. However, the ability to capture the correlations between the patterns should worsen, as the patterns would be more spread out. Consequently, it could cause the generalisation to worsen. However, the risk of getting stuck in a local minima should decrease as there are more escape routes in a network with higher dimensions. Another disadvantage is that bigger networks take longer to train as there are more calculations to make for each iteration.

I have tried several networks with the same training set, but different architectures, and by this I mean different amounts of nodes in the hidden layer. More specifically I have tried in total 12 different networks with every hundred of hidden nodes within the interval $[100, 1000]$ and another two networks built up by 50 and 75 hidden nodes. For each network I have made the network try to recognize 1100 new (unseen) patterns (with the same font as the ones in the training set). Among these patterns there were at least one representative of each of the 60 signs. I have gathered the results in the table below where the columns (in the order from left to right) represent the number of hidden nodes, if convergence has been reached or not, the number of incorrect guesses, the number of correct guesses and finally a percentage number of correct guesses.

Before I present the results of my research, there is a comment that I would like to make. All networks made the same mix-up between lower case L:s and capital i:s. However, I would not consider these guesses as incorrect because the two signs have exactly the same shape for this particular font and sometimes also the same size. As there are at least two patterns in the training set which are identical for these two signs, it is likely that their decision regions overlap and therefore this behavior was expected. Consequently, as it is the job of the network to cluster patterns with exactly the same shape and size in the same decision region, I will consider these guesses as correctly made.

Results and comments

Hidden nodes	Convergence?	incorrect	correct	share correct
50	no			
75	yes	2	1098	99.80
100	yes	0	1100	100
200	yes	0	1100	100
300	yes	0	1100	100
400	yes	0	1100	100
500	yes	1	1099	99.90
600	yes	0	1100	100
700	yes	1	1099	99.90
800	yes	7	1093	99.40
900	yes	7	1093	99.40
1000	yes	2	1098	99.80

By looking at the table of results above, there are some comments that I would like to make. First of all, I would like to comment on the result of all networks by trying to answer the question: Was this result expected? Well, as the patterns have a limited variation in their looks and the patterns outside the training set do not differ that much from the training set, the training set makes a realistic view of the reality. Moreover, according to the theory almost any neural network could be trained so well that no errors will be encountered on the training set. Thus, my answer to the question if the achieved result was expected would be yes because of the above arguments.

When discussing each network separately, I will presuppose my own discussion treating the risk and advantages when decreasing or increasing the number of hidden units. Beginning with the network using 50 hidden nodes, it appears from the table that it did not reach convergence. I tried to train the network twice and in the second one I included the function Adaptive learning, but no convergence was reached. My conclusion to that is that the decision regions formed were not complex enough to find an XOR.

Increasing the number of hidden nodes to 75 resulted in a network that did reach convergence. However, it did make two mistakes. I would suspect that it is due to the fact that the network got stuck in a local minima rather than failing to capture the correlations in between the patterns, refering to my discussion under the paragraph treating too few number of hidden units.

By looking at the overall achievement, one notable thing is that there is an interval from 100 to 400 nodes where no incorrect guesses have been made. This fact leads naturally to the question if this is a coincidence or if it is a logical pattern? I will discuss this matter later on. Before, I would like to comment on the intervall from 500 to 1000. As can be seen on the table it is more common to make incorrect guesses than not within this interval. More specifically all networks in this interval with the exception of the one with 600 hidden units make incorrect guesses. My first thought is that these networks have been worse at generalising and capturing the correlations between the patterns, referring to my discussion about increasing the number of hidden nodes. Of course, I can not exclude the fact that the incorrect guesses could be due to overfitting of the network, which would be done if the network was trained for too long. During this research, I found that the time increased each time I increased the number of hidden nodes in the network and thus, the network containing 1000 hidden units took the longest to train. However, this behavior is expected as increasing the number of hidden nodes means that there are more computations to perform for each iteration and naturally the training process should take longer time. Therefore, overfitting is not something that I suspect in the first place. Of course, the incorrect guesses could also be caused by local minimas. However, I find this occurence less likely due to my discussion about number of hidden nodes. Consequently, I find it more likely that these networks have generalised badly.

To sum up, when choosing the architecture of the network for this particular font I think that my research can be used as a decision basis and that the good results of networks using 100 to 400 nodes could work as a direction when choosing the amount of nodes in the hidden layer. One way of deciding the amount of hidden nodes could be to choose a number in the middle of this intervall.

Moreover, the conclusion that I make by looking at the result from my research is that the number of hidden nodes did not affect the results in a significant way, meaning that the overall achievement was really good as all networks performed at least 99.4 percent. However, according to the theory, the result also depend on other things like, for instance, the training set.

Even if the result from my test was really good, it might not work as well in the reality. When this program is going to be used for the purpose it is made for, the training set will not be chosen in the same way that I did, that is with great concern, as a computer will make that job. I had the privilege to know all possible looks that each sign could have when choosing my training set, whereas this will not be the case for a computer. Thus, my network might be more robust and generalise better than others. Therefore, I mean that it is still meaningful to suggest interventions for improvements.

Suggestions for improvement According to the theory, the order of the patterns should be changed after each iteration during the training process. This is something that the current version of the algorithm does not do. My speculations and thoughts of why this is important are the following:

A possible explanation that I could think of is preventing the weights from bouncing around. If the weights are focusing on adapting very similar patterns that occur very frequently for a while, and then suddenly start adapting to other similar patterns for a while and so on, rather than seeing a totally different pattern every time, than the weights might bounce around a lot instead of making small adjustments towards different directions. This could lower the speed of convergence (or in worst case never reach convergence), which would give the weights a lot of time to grow and create strong connections between the nodes. Consequently, the risk of overfitting the network would increase.

Eventually, to sum up, my suggestions to improve this program are the following:

- Change the order after each iteration
- When choosing the training set, pick patterns with as different looks as possible so that the network will form decision regions as close to the reality as possible.
- for this particular font that I have been experimenting with, use one hidden layer and somewhere in between 200 and 300 hidden nodes in the network structure.

- In general, when designing networks for other fonts, I believe that one hidden layer is enough, again due to the little variation of the patterns (where only a simple kind of decision region is needed). When it comes to the number of hidden nodes, the only direction that I can give according to the theory and my observations is to find a number which is high enough to reach convergence and avoid local minimas and low enough to generalise well.

References

[1] Machine Learning, Stephen Marsland, Taylor and Francis Group, 2009

[2] Machine Learning, Tom M. Mitchell, The McGraw-Hill Companies, 1997

[3] Introduction to the theory of neural computation, John Hertz, Anders Krogh, Richard G. Palmer, Addison-Wesley Publishing Company, 1991

[4] An introduction to Computing with Neural Nets, Richard P. Lippman, IEEE ASSP Magazine, April 1987

[5] Perceptrons, H.J.M. Peters, Department of Quantitative Economics, University of Limburg, Maastricht (publishing year unknown)

[6] A convergence theorem for sequential learning in two layer perceptrons, Mario Marchan, Mostefa Golea, Department of Physics, University of Ottawa, 34 G. Glinsky, Ottawa, Canada K1N-6N5, Accepted by Europhysics Lett, December 19, 1989

[7] `http://www.youtube.com/watch?v=tRG-OnnQ9g4`, 2013

[8] `http://psych.stanford.edu/~jlm/papers/PDP/Chapter1.pdf`, 2013

[9] `http://www.pearsonhighered.com/assets/hip/us/hip_us_pearsonhighered/samplechapter/0131471392.pdf`, 2013

[10] `http://www.codeproject.com/Articles/16650/Neural-Network-for-Recognition-of-Handwritten-Digi`, 2012

[11] `http://www.codeproject.com/Articles/19323/Image-Recognition-with-Neural-Networks`, 2012

[12] `http://www.codeproject.com/Articles/3907/Creating-Optical-Character-Recognition-OCR-applica`, 2012

[13] `http://en.wikipedia.org/wiki/Massively_parallel_(computing)`, 2013

[14] `http://en.wikipedia.org/wiki/Machine_learning`, 2013

[15] `http://en.wikipedia.org/wiki/Axon`, 2013

[16] `http://en.wikipedia.org/wiki/Synapse`, 2013

[17] `http://en.wikipedia.org/wiki/Perceptron`, 2013

[18] `http://en.wikipedia.org/wiki/Hebbian_theory`, 2013

[19] `http://en.wikipedia.org/wiki/PostScript`, 2013

[20] `http://www.asciitable.com/`, 2013

[21] `http://en.wikipedia.org/wiki/File:Logistic-curve.svg`, Qef, 2013

[22] `http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-Char.html`, 2013

Appendix

In this section I will continue and finish the calculations for the examples in 1.3.1 and 2.1.1 and show parts of the code from the implementation.

Continuation of the calculations in 1.3.1

Round 1 Activation of (0, 1):

$$w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2 = 0.2 \cdot (-1) - 0.02 \cdot 0 + 0.02 \cdot 1 = -0.18$$

(The updated values of these weights have already been calculated as $w_0 = -0.05$, $w_1 = -0.02$ and $w_2 = 0.27$.)

Activation of (1, 0):

$$w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2 = -0.05 \cdot (-1) - 0.02 \cdot 1 + 0.27 \cdot 0 = 0.03$$

Activation of (1, 1):

$$w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2 = -0.05 \cdot (-1) - 0.02 \cdot 1 + 0.27 \cdot 1 = 0.3$$

Round 2 Activation of (0, 0):

$$w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2 = -0.05 \cdot (-1) - 0.02 \cdot 0 + 0.27 \cdot 0 = 0.05$$

Updating of the weights:

$$w_0 + \eta(t-y) \cdot x_0 \to w_0 \Rightarrow -0.05 + 0.25(0-1) \cdot (-1) = 0.2$$
$$w_1 + \eta(t-y) \cdot x_1 \to w_1 \Rightarrow -0.02 + 0.25(0-1) \cdot 0 = -0.02$$
$$w_2 + \eta(t-y) \cdot x_2 \to w_2 \Rightarrow 0.27 + 0.25(0-1) \cdot 0 = 0.27$$

Activation of (0, 1):

$$w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2 = 0.2 \cdot (-1) - 0.02 \cdot 0 + 0.27 \cdot 1 = 0.07$$

Activation of (1, 0):

$$w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2 = 0.2 \cdot (-1) - 0.02 \cdot 1 + 0.27 \cdot 0 = -0.22$$

Updating of the weights:

$$w_0 + \eta(t-y) \cdot x_0 \to w_0 \Rightarrow 0.2 + 0.25(1-0) \cdot (-1) = -0.05$$
$$w_1 + \eta(t-y) \cdot x_1 \to w_1 \Rightarrow -0.02 + 0.25(1-0) \cdot 1 = 0.23$$
$$w_2 + \eta(t-y) \cdot x_2 \to w_2 \Rightarrow 0.27 + 0.25(1-0) \cdot 0 = 0.27$$

Activation of (1, 1):

$$w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2 = -0.05 \cdot (-1) + 0.23 \cdot 1 + 0.27 \cdot 1 = 0.55$$

Round 3 Activation of (0, 0):

$$w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2 = -0.05 \cdot (-1) + 0.23 \cdot 0 + 0.27 \cdot 0 = 0.05$$

Updating of the weights:

$$w_0 + \eta(t-y) \cdot x_0 \to w_0 \Rightarrow -0.05 + 0.25(0-1) \cdot (-1) = 0.2$$
$$w_1 + \eta(t-y) \cdot x_1 \to w_1 \Rightarrow 0.23 + 0.25(0-1) \cdot 0 = 0.23$$
$$w_2 + \eta(t-y) \cdot x_2 \to w_2 \Rightarrow 0.27 + 0.25(0-1) \cdot 0 = 0.27$$

Activation of (0, 1):

$$w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2 = 0.2 \cdot (-1) + 0.23 \cdot 0 + 0.27 \cdot 1 = 0.07$$

Activation of (1, 0):

$$w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2 = 0.2 \cdot (-1) + 0.23 \cdot 1 + 0.27 \cdot 0 = 0.03$$

Activation of (1, 1):

$$w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2 = 0.2 \cdot (-1) + 0.23 \cdot 1 + 0.27 \cdot 1 = 0.3$$

Round 4 Activation of (1, 1):

$$w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2 = 0.2 \cdot (-1) + 0.23 \cdot 0 + 0.27 \cdot 0 = -0.2$$

Now, it has been confirmed that all input vectors compute the right answers for $w_0 = 0.2$, $w_1 = 0.23$ and $w_2 = 0.27$.

Continuation of the calculations in 2.1.1

Input vector (0, 0) Calculation of neuron C:

$b_1 \cdot v_{01} + A \cdot v_{11} + B \cdot v_{21} = (-1) \cdot 0.5 + 0 \cdot 1 + 0 \cdot 1 = -0.5 \Rightarrow$ the output of neuron C is zero

Calculation of neuron D:

$b_1 \cdot v_{02} + A \cdot v_{12} + B \cdot v_{22} = (-1) \cdot 1 + 0 \cdot 1 + 0 \cdot 1 = -1 \Rightarrow$ the output of neuron D is zero

Calculation of neuron E:

$b_2 \cdot w_0 + C \cdot w_1 + D \cdot w_2 = (-1) \cdot 0.5 + 0 \cdot 1 + 0 \cdot (-1) = -0.5 \Rightarrow$ the output of neuron E is zero

Input vector (0, 1) Calculation of neuron C:

$b_1 \cdot v_{01} + A \cdot v_{11} + B \cdot v_{21} = (-1) \cdot 0.5 + 0 \cdot 1 + 1 \cdot 1 = 0.5 \Rightarrow$ the output of neuron C is one

Calculation of neuron D:

$b_1 \cdot v_{02} + A \cdot v_{12} + B \cdot v_{22} = (-1) \cdot 1 + 0 \cdot 1 + 1 \cdot 1 = 0 \Rightarrow$ the output of neuron D is zero

Calculation of neuron E:

$b_2 \cdot w_0 + C \cdot w_1 + D \cdot w_2 = (-1) \cdot 0.5 + 1 \cdot 1 + 0 \cdot (-1) = 0.5 \Rightarrow$ the output of neuron E is one

Input vector (1, 1) Calculation of neuron C:

$b_1 \cdot v_{01} + A \cdot v_{11} + B \cdot v_{21} = (-1) \cdot 0.5 + 1 \cdot 1 + 1 \cdot 1 = 1.5 \Rightarrow$ the output of neuron C is one

Calculation of neuron D:

$b_1 \cdot v_{02} + A \cdot v_{12} + B \cdot v_{22} = (-1) \cdot 1 + 1 \cdot 1 + 1 \cdot 1 = 1 \Rightarrow$ the output of neuron D is one

Calculation of neuron E:

$b_2 \cdot w_0 + C \cdot w_1 + D \cdot w_2 = (-1) \cdot 0.5 + 1 \cdot 1 + 1 \cdot (-1) = -1.5 \Rightarrow$ the output of neuron E is zero

Parts of the Code

The Sum-of-squares function

```
function TBP1Layer.GetError : double;
var
    j : integer;
    total : double;
begin
    total:= 0.0;
    for j:= 0 to OutputNum-1 do
    begin
        total:= total + Power((OutputLayer[j].Target - OutputLayer[j].output), 2) / 2;
    end;
    Result:= total;
end;
```

The Activation function

```
function TBP1Layer.F(x : double) : double;
begin
    result:= 1/(1+exp(-x));
end;
```

The Forwards phase

```
procedure TBP1Layer.ForwardPropagate(pattern: array of double; output : string);
var
    i, j : integer;
    total : double;
begin
    //Apply input to the network
    for i:= 0 to InputNum-1 do
    begin
        InputLayer[i].Value:= pattern[i];
    end;

    //Calculate the hidden layer's inputs and outputs:
    for i:= 0 to Hidden1Num-1 do
    begin
        total:= 0.0;
        for j:= 0 to InputNum-1 do
        begin
            total:= total + InputLayer[j].Value * InputLayer[j].Weights[i];
        end;
        Hidden1Layer[i].InputSum:= total;
        Hidden1Layer[i].Output:= F(total);
    end;

    //Calculate the output layer's inputs, outputs, targets and errors:
    for i:= 0 to OutputNum-1 do
    begin
        total:= 0.0;
        for j:= 0 to Hidden1Num-1 do
        begin
            total:= total + Hidden1Layer[j].Output * Hidden1Layer[j].Weights[i];
        end;
        OutputLayer[i].InputSum:= total;
        OutputLayer[i].output:= F(total);
        if OutputLayer[i].Value = output then
            OutputLayer[i].Target:= 1.0
        else
            OutputLayer[i].Target:= 0.0;
        OutputLayer[i].Error:= (OutputLayer[i].Target - OutputLayer[i].output) *
         (OutputLayer[i].output) * (1 - OutputLayer[i].output);
    end;
end;
```

The Backwards phase: Backpropagation of error

```
procedure TBP1Layer.BackPropagate;
var
    i,j : integer;
    total : double;
begin
     //Calculate hidden layer's error:
    for i:= 0 to Hidden1Num-1 do
    begin
        total:= 0.0;
        for j:= 0 to OutputNum-1 do
        begin
            total:=total + Hidden1Layer[i].Weights[j] * OutputLayer[j].Error;
        end;
        Hidden1Layer[i].Error:=
         Hidden1Layer[i].Output * (1 - Hidden1Layer[i].Output) * total;
    end;
    //Update the first layer weights:
    for i:= 0 to Hidden1Num-1 do
    begin
        for j:= 0 to InputNum-1 do
        begin
            InputLayer[j].Weights[i]:=InputLayer[j].Weights[i] +
                flearningRate * Hidden1Layer[i].Error * InputLayer[j].Value;
        end;
    end;
    //Update the second layer weights:
    for i:= 0 to OutputNum-1 do
    begin
        for j:= 0 to Hidden1Num-1 do
        begin
            Hidden1Layer[j].Weights[i]:=Hidden1Layer[j].Weights[i] +
                flearningRate * OutputLayer[i].Error * Hidden1Layer[j].Output;
        end;
    end;
end;
```

The Training process

```
function TNeuralNetwork.Train(AdaptiveLearn : Boolean) : Boolean;
var
    currentError : double;
    previousError : double;
    currentIteration : integer;
    ix : integer;
    TrainingValue : TTrainingValue;
begin
    currentIteration:= 0;
    previousError:= 0;
    repeat
    begin
        currentError:= 0;
        NeuralNet.SaveOldWeights;
        for ix:=0 to TrainingSet.count-1 do
        begin
            TrainingValue := TrainingSet[ix];
            NeuralNet.ForwardPropagate(TrainingValue.Value, TrainingValue.Key);
            NeuralNet.BackPropagate();
            currentError:= currentError + NeuralNet.GetError();
        end;
        if AdaptiveLearn then
        begin
            if currentIteration<>0 then
            begin
                if previousError<currentError then
                begin
                    NeuralNet.LearningRate:= NeuralNet.LearningRate*0.8;
                end
                else begin
                    NeuralNet.LearningRate:= NeuralNet.LearningRate*1.1;
                end;
            end;
            previousError:= currentError;
        end;
        inc(currentIteration);
    end until ((currentError <= maximumError) or (currentIteration >= maximumIteration));
    Result:= true;
    if currentIteration >= maximumIteration then
        Result:= false;
end;
```

The Recognition process

```
procedure TBP1Layer.Recognize
(Input : array of double; var MatchedHigh : string; var OutputValueHight : double;
var MatchedLow : string; var OutputValueLow : double);

var
    i,j : integer;
    total : double;
    max : double;
begin
    max:= -1;

    for i:= 0 to InputNum-1 do
    begin
        InputLayer[i].Value:= Input[i];
    end;

    if not IsItBlank then
    begin
      for i:= 0 to Hidden1Num-1 do
      begin
          total:= 0.0;
```

```
            for j:= 0 to InputNum-1 do
            begin
                total:= total + InputLayer[j].Value * InputLayer[j].Weights[i];
            end;
            Hidden1Layer[i].InputSum:= total;
            Hidden1Layer[i].Output:= F(total);
        end;

        for i:= 0 to OutputNum-1 do
        begin
            total:= 0.0;
            for j:= 0 to Hidden1Num-1 do
            begin
                total:= total + Hidden1Layer[j].Output * Hidden1Layer[j].Weights[i];
            end;
            OutputLayer[i].InputSum:= total;
            OutputLayer[i].output:= F(total);
            if OutputLayer[i].output > max then
            begin
                MatchedLow:= MatchedHigh;
                OutputValueLow:= max;
                max:= OutputLayer[i].output;
                MatchedHigh:= OutputLayer[i].Value;
                OutputValueHight:= max;
            end;
        end;
      end else
      begin
        MatchedLow:= ' ';
        OutputValueLow:= 1;
        MatchedHigh:= ' ';
        OutputValueHight:= 1;
      end;
  end;
```