Pattern Recognition Lecture
"Nonlinear Classifiers"

Prof. Dr. Marcin Grzegorzek

Research Group for Pattern Recognition
www.pr.informatik.uni-siegen.de

Institute for Vision and Graphics
University of Siegen, Germany

Overview

Introduction

- In the previous lecture we dealt with linear classifiers described by linear discriminant functions (hyperplanes) $g(\mathbf{x})$.
- For two linearly separable classes, the weights of the linear function can be determined by, e. g., the perceptron algorithm.
- For two nonlinearly separable classes, linear classifiers can be optimally designed by, e. g., minimising the sum of error squares.
- **Lecture 4 deals with nonlinearly separable problems for which the design of a linear classifier does not lead to satisfactory performance.**

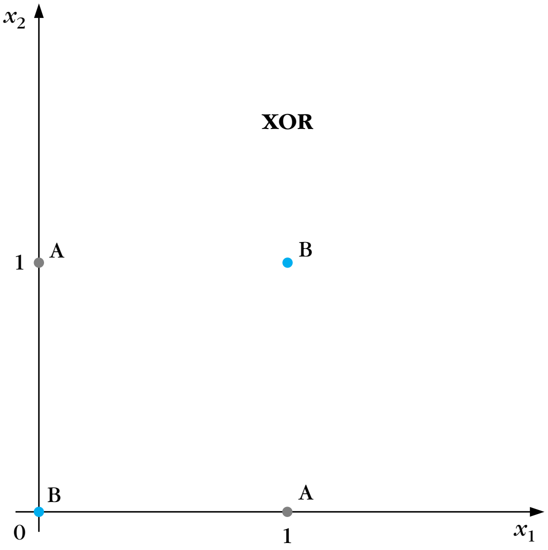
Overview

The XOR Problem - Introduction

- Boolean functions can be interpreted as two-class classification tasks.
- Depending on the input data $\mathbf{x}$, the output is either 0 or 1, and $\mathbf{x}$ is classified into one of the two classes $A(1)$ or $B(0)$.
- The truth table for the XOR problem looks like follows:

x_1	x_2	XOR	Class
0	0	0	B
0	1	1	A
1	0	1	A
1	1	0	B

The XOR Problem - Visualisation



The XOR Problem - Linear Solution

- What would be the output of the minimising the sum of error squares?
- The minimisation is given by $(X^{\mathrm{T}}X)\mathbf{w} = X^{\mathrm{T}}\mathbf{y}$
- For the XOR problem

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix}$$

$$\Downarrow$$

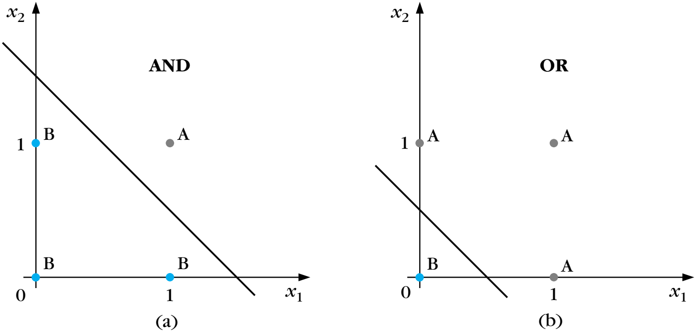
$$\begin{bmatrix} w_1 \\ w_2 \\ w_0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The AND and OR Problems - The Truth Table

- The truth table for the AND and OR problems looks like follows:

x_1	x_2	AND	Class	OR	Class
0	0	0	B	0	B
0	1	0	B	1	A
1	0	0	B	1	A
1	1	1	A	1	A

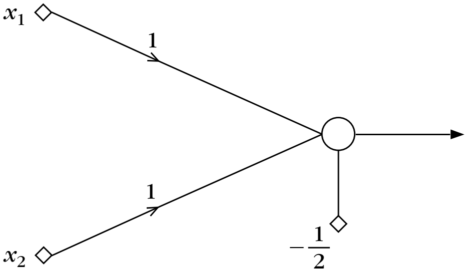
The AND and OR Problems - Visualisation



$$(X^{\mathrm{T}}X)\mathbf{w} = X^{\mathrm{T}}\mathbf{y}$$
$$\Downarrow$$
$$\mathbf{w} = [\tfrac{3}{2}, \tfrac{3}{2}, -\tfrac{9}{4}]^{\mathrm{T}}$$

$$(X^{\mathrm{T}}X)\mathbf{w} = X^{\mathrm{T}}\mathbf{y}$$
$$\Downarrow$$
$$\mathbf{w} = [1, 1, -\tfrac{1}{2}]^{\mathrm{T}}$$

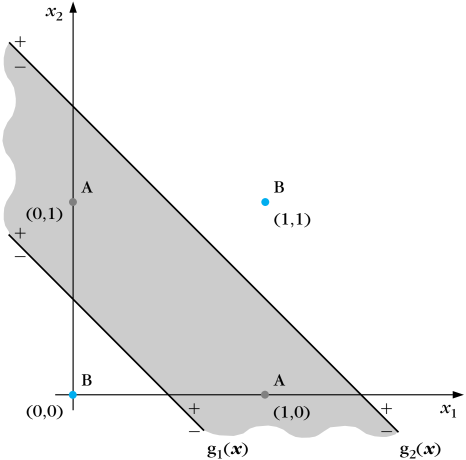
OR Gate Realised as a Perceptron



Overview

Two-Layer Perceptron for the XOR Problem (1)

- An intuitive solution to the XOR problem would be two lines instead of a single separating line



Two-Layer Perceptron for the XOR Problem (2)

The classification problem for a pattern **x** has to be solved here in two successive phases:

1. The position of **x** is calculated with respect to each of the decision lines separately.
2. The results from the first phase are combined and the position of **x** is determined with respect to both lines.

The realisation of the two decision lines $g_1(\cdot)$ and $g_2(\cdot)$ during the first phase is achieved by the adoption of two perceptrons with inputs x_1 and x_2 and appropriate synaptic weights.

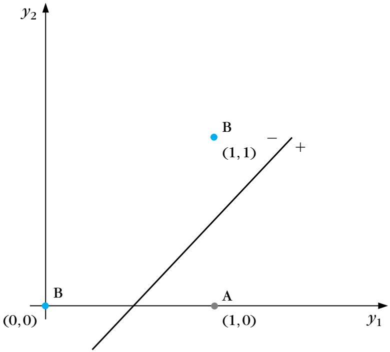
Two-Layer Perceptron for the XOR Problem (3)

- Inputs x_1 and x_2 have corresponding outputs given by $y_i = f(g_i(\mathbf{x})), i = 1, 2$ where the activation function $f(\cdot)$ is the step function with levels 0 and 1.
- The truth table for the two computation phases of the XOR problem looks like follows

1st Phase				**2nd Phase**
x_1	x_2	y_1	y_2	
0	0	$0(-)$	$0(-)$	B(0)
0	1	$1(+)$	$0(-)$	A(1)
1	0	$1(+)$	$0(-)$	A(1)
1	1	$1(+)$	$1(+)$	B(0)

Two-Layer Perceptron for the XOR Problem (4)

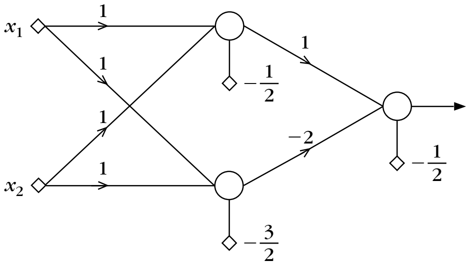
- The 1st phase maps the input vector $\mathbf{x}$ to a new one $\mathbf{y} = [y_1, y_2]^{\mathrm{T}}$.
- The decision in the 2nd phase is taken based on the transformed data



- The mapping of the 1st phase transforms the nonlinearly separable problem to a linearly separable one.

Two-Layer Perceptron Solving the XOR Problem

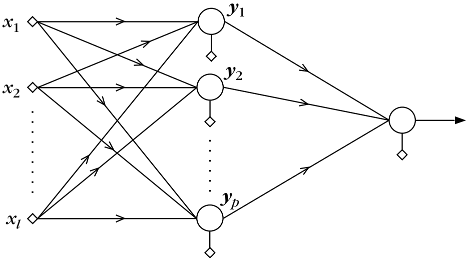
- Each of the three lines is realised via a neuron with appropriate synaptic weights



- It is called also a two-layer neural network.
- The nodes are called neurons.
- The nodes of the 1st phase constitute the hidden layer.
- The node of the 2nd phase constitutes the output layer.

Two-Layer Perceptron for l-Dimensions (1)

- The two-layer perceptron can be generalised for l-dimensional feature vectors $\mathbf{x} \in \mathbb{R}^l$



- The number of output neurons may also be greater than one.

Two-Layer Perceptron for l-Dimensions (2)

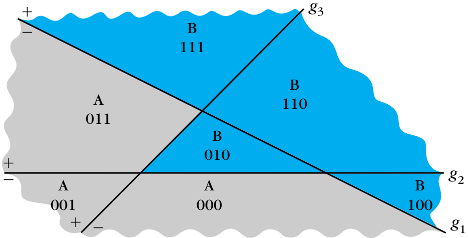
- As for the XOR problem, employing the step activation function maps the input space onto the vertices of the hypercube of unit side length in the p-dimensional space denoted by

$$H_p = \{[y_1, \ldots, y_p]^{\mathrm{T}} \in \mathbb{R}^p, \quad y_i \in \{0, 1\}\}$$

- The mapping of the input space onto the vertices of the hypercube is achieved via the creation of p hyperplanes.
- Each of the hyperplanes is created by a neuron in the hidden layer.
- The output of each neuron is 0 or 1, depending on the relevant position of the input vector with respect to the corresponding hyperplane.

Two-Layer Perceptron Example for 3-Dimensions

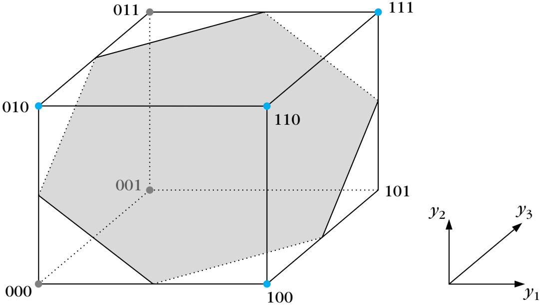
- The outputs of the hidden layer are three planes.
- They divide the space into seven regions, which in 2D looks like follows



- Each region corresponds a vertex of the unit 3D cube (see next slide).

Two-Layer Perceptron Example for 3-Dimensions

- The neurons of the first hidden layer map an input vector onto one of the vertices of a unit cube.
- The output neuron realises a plane to separate vertices according to their class label



- The output plane is: $y_1 - y_2 + y_3 + 0.5 = 0$

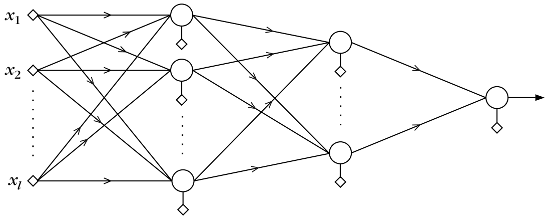
Two-Layer Perceptron - Limitations

- The classification on the previous slide is possible, because vertices of class A (000, 001, 011) lie on the one side, and vertices of class B (010, 100, 110, 111) on the other.
- If A consists of the union $000 \cup 111 \cup 110$ and B of the rest, it is not possible to construct a single plane that separates class A from class B.
- Concluding, two-layers of neurons are not enough to perform classification in this case.

Overview

Three-Layer Perceptrons (1)

- The problem mentioned on the previous slide springs from the fact that the output neuron can realise only a single hyperplane.
- The difficulty can be overcome by including an additional hidden layer.



- The three-layer perceptron can separate classes resulting from any union of regions.

Overview

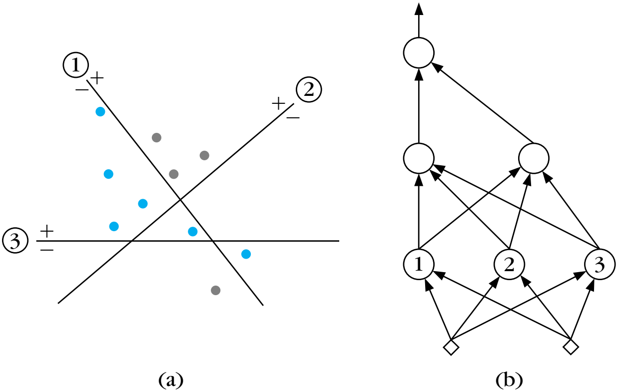
General Idea

- The starting point is a small architecture unable to solve the problem at hand.
- It is successively augmented until the correct classification of all N feature vectors from the training set X is achieved.
- Some algorithms expand the architecture in terms of layers, others expand the number of nodes.
- The problem is decomposed into smaller ones that are easier to handle.

The Tiling Algorithm (1)

- It constructs architectures with many layers.
- For the two-class (A and B) case, the algorithm starts with a single node $n(X)$ in the first layer which is called the master unit of this layer.
- This node is trained and divides the training data set X into two subsets X^+ and X^-
- If X^+ (X^-) contains feature vectors from both classes, we introduce an additional node $n(X^+)$ ($n(X^-)$) which is called the ancillary unit.
- This node is trained using only the feature vectors in X^+ (X^-).
- If one of the X^{++}, X^{+-} (X^{-+}, X^{--}) contain vectors from both classes, more ancillary nodes are added.

The Tiling Algorithm (2)



The Tiling Algorithm (3)

- Let $X_1 = \{\mathbf{y} : \mathbf{y} = f_1(\mathbf{x}), \mathbf{x} \in X\}$, where f_1 is the mapping implemented by the first layer.
- Applying the procedure described by the previous slide to the set X_1, we construct the second layer of the architecture and so on.
- It has been shown that the newly added master unit classifies correctly all the vectors correctly classified by the master unit of the previous layer, plus at least one more.
- Thus, the tiling algorithm produces an architecture that classifies correctly all patterns of X in a finite number of steps

Overview

Introduction

- Fixing the architecture and computing its synaptic weights for a particular tasks can be solved by minimising an appropriate cost function of its output.

- A serious difficulty to this approach is the discontinuity of the step (activation) function which prohibits differentiation with respect to the unknown parameters (synaptic weights).

- How to overcome this difficulty?

Approximation of the Activation Function

- The activation function of a neuron is, e. g.

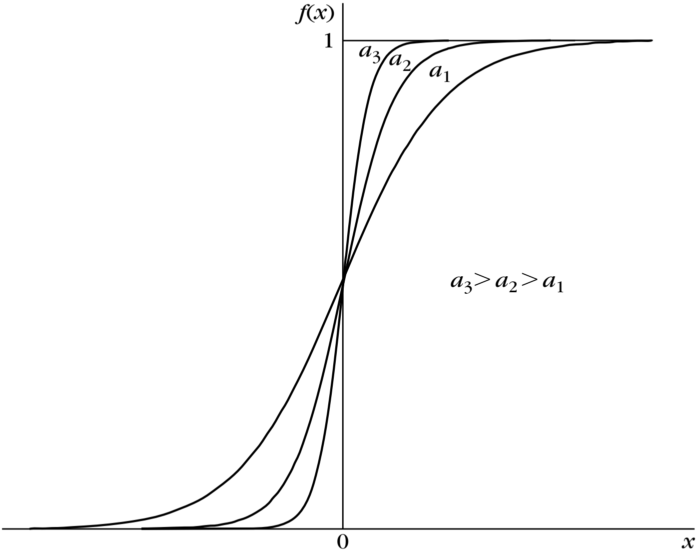
$$f(x) = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}$$

- An approximation of this function is the logistic function

$$f(x) = \frac{1}{1 + e^{-ax}}$$

- This function belongs to the family of sigmoid functions and is differentiable, in contrast to the activation function itself.

The Logistic Function



Variations of the Logistic Function

- Sometimes an antisymmetric variation of the logistic function is employed

$$f(x) = \frac{2}{1 + e^{-ax}} - 1$$

- It varies between 1 and -1 and belongs to the family of hyperbolic tangent functions.

$$f(x) = c\frac{1 - e^{-ax}}{1 + e^{-ax}} = c\tanh(\frac{ax}{2})$$

- In the following, we will adopt multilayer neural architectures and assume an activation function of the type given on this slide.

Problem Statement

The goal is to derive an iterative training algorithm that computes the synaptic weights of the network so that an appropriately chosen cost function is minimised.

Assumptions

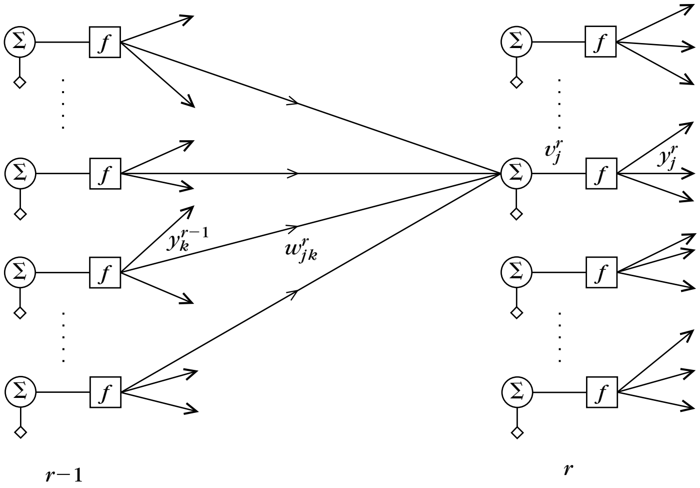
- The network consists of L layers of neurons.
- We have $k_0 = l$ nodes in the input layer and k_r neurons in the layer no. r, $r = 1, \ldots, L$.
- All the neurons employ the same sigmoid activation function.
- N training pairs are available $(\mathbf{y}(i), \mathbf{x}(i))$.
- The output is a k_L-dimensional vector

$$\mathbf{y}(i) = [y_1(i), \ldots, y_{k_L}(i)]^{\mathrm{T}}$$

- The input feature vectors are k_0-dimensional

$$\mathbf{x}(i) = [x_1(i), \ldots, x_{k_0}(i)]^{\mathrm{T}}$$

Variables Involved in the Backpropagation Problem



Approach in General (1)

- During training, when vector $\mathbf{x}(i)$ is applied to the input, the output of the network will be $\widehat{\mathbf{y}}(i)$ which may be different from the desired value $\mathbf{y}(i)$.
- The synaptic weights are computed so that an appropriate (for each problem) cost function J, which is dependent on the values $\mathbf{y}(i)$ and $\widehat{\mathbf{y}}(i)$ $(i = 1, \ldots, N)$ is minimised.
- The solution is based on the gradient descent scheme.

Approach in General (2)

- Let $\mathbf{w}_j^r$ be the weight vector (including the threshold) of the neuron no. j in the layer no. r

$$\mathbf{w}_j^r = [w_{j0}^r, w_{j1}^r, \ldots, w_{jk_{r-1}}^r]^{\mathrm{T}}$$

- The basic iteration step will be of the form

$$\mathbf{w}_j^r(\mathrm{new}) = \mathbf{w}_j^r(\mathrm{old}) + \Delta\mathbf{w}_j^r$$

with

$$\Delta\mathbf{w}_j^r = -\mu \frac{\partial J}{\partial \mathbf{w}_j^r}$$

Cost Function (1)

- We will focus our attention on cost functions of the form

$$J = \sum_{i=1}^{N} \varepsilon(i)$$

- ε is an appropriately defined function depending on $\widehat{\mathbf{y}}(i)$ and $\mathbf{y}(i)$ $(i = 1, \ldots, N)$.

- J is expressed as a sum of the N values that function ε takes for each of the training pairs $(\mathbf{y}(i), \mathbf{x}(i))$.

Cost Function (2)

- For example, $\varepsilon(i)$ can be the sum of squared errors in the output neurons

$$\varepsilon(i) = \frac{1}{2} \sum_{m=1}^{k_L} (y_m(i) - \widehat{y}_m(i))^2, \quad i = 1, \ldots, N$$

- For the computation of the correction term, the gradient of the cost function J with respect to the weights is required and, consequently, the evaluation of $\partial\varepsilon(i)/\partial\mathbf{w}_j^r$.

Computation of the Gradients (1)

- Let $y_k^{r-1}(i)$ be the output of the neuron no. k in the layer no. $r-1$ for the training pair no. i
- w_{jk}^r is the current estimate of the corresponding weight leading to the neuron no. j in the layer no. r.
- Thus, the argument of the activation function $f(\cdot)$ of the latter neuron will be

$$v_j^r(i) = \sum_{k=1}^{k_r-1} w_{jk}^r y_k^{r-1} + w_{j0}^r \equiv \sum_{k=0}^{k_r-1} w_{jk}^r y_k^{r-1}$$

where

$$\forall r, i \quad y_0^r(i) \equiv 0$$

Computation of the Gradients (2)

- The dependence of $\varepsilon(i)$ on $\mathbf{w}_j^r$ passes trough $v_j^r(i)$.

- Therefore, by the chain rule in differentiation, we have

$$\frac{\partial \varepsilon(i)}{\partial \mathbf{w}_j^r} = \frac{\partial \varepsilon(i)}{\partial v_j^r(i)} \frac{\partial v_j^r(i)}{\partial \mathbf{w}_j^r}$$

Computation of the Gradients (3)

- Using the expression for $v_j^r(i)$ from slide 21, we have

$$\frac{\partial v_j^r(i)}{\partial \mathbf{w}_j^r} = \begin{bmatrix} \frac{\partial v_j^r(i)}{\partial w_{j0}^r} \\ \vdots \\ \frac{\partial v_j^r(i)}{\partial w_{jk_{r-1}}^r} \end{bmatrix} = \mathbf{y}^{r-1}(i) = \begin{bmatrix} 1 \\ y_1^{r-1}(i) \\ \vdots \\ y_{k_{r-1}}^{r-1}(i) \end{bmatrix}$$

Computation of the Gradients (4)

- Let us define
$$\frac{\partial \varepsilon(i)}{\partial v_j^r(i)} \equiv \delta_j^r(i)$$
- Then, the correction term can be computed by
$$\Delta \mathbf{w}_j^r = -\mu \sum_{i=1}^{N} \delta_j^r(i) \mathbf{y}^{r-1}(i)$$
- $\delta_j^r(i)$ is computed for the cost function on Slide 20 starting from $r = L$ and propagating backward for $r = L - 1$, $L - 2$, ..., 1.

The Backpropagation Algorithm (1)

1. **Initialisation**
   - Initialise all the weights with small random values from a pseudorandom sequence generator
2. **Forward Computations**
   - For each of the training feature vectors $\mathbf{x}(i)$ $(i = 1, \ldots, N)$ compute all the $v_j^r(i)$ and $y_j^r(i) = f(v_j^r(i))$ $(j = 1, \ldots, k_r$ and $r = 1, \ldots, L)$.
   - Compute the cost function for the current estimate of weights.

The Backpropagation Algorithm (2)

3. Backward Computations

- For each $i = 1, \dots, N$ and $j = 1, \dots, k_L$ compute $\delta_j^L(i)$
- Keep computing $\delta_j^{r-1}(i)$ for $r = L, L-1, \dots, 2$ and $j = 1, \dots, k_r$.

4. Update the Weights

- For $r = 1, \dots, L$ and $j = 1, \dots, k_r$

$$\mathbf{w}_j^r(\text{new}) = \mathbf{w}_j^r(\text{old}) + \Delta \mathbf{w}_j^r$$

with

$$\Delta \mathbf{w}_j^r = -\mu \sum_{i=1}^{N} \delta_j^r(i) \mathbf{y}^{r-1}(i)$$