Chapter ML:VI (continued)

VI. Neural Networks

- Perceptron Learning
- Gradient Descent
- Multilayer Perceptron
- Radial Basis Functions

Multilayer Perceptron

Definition 1 (Linear Separability)

Two sets of feature vectors, X_0, X_1, of a p-dimensional feature space are called linearly separable, if $p+1$ real numbers, $w_0, w_1, \ldots, w_p$, exist such that holds:

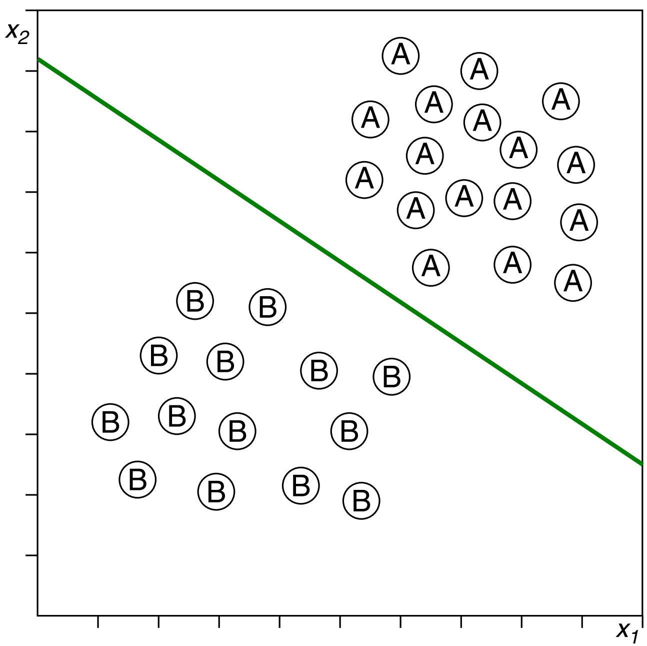
1. $\forall \mathbf{x} \in X_0$: $\sum_{j=0}^{p} w_j x_j < 0$
2. $\forall \mathbf{x} \in X_1$: $\sum_{j=0}^{p} w_j x_j \geq 0$

Multilayer Perceptron

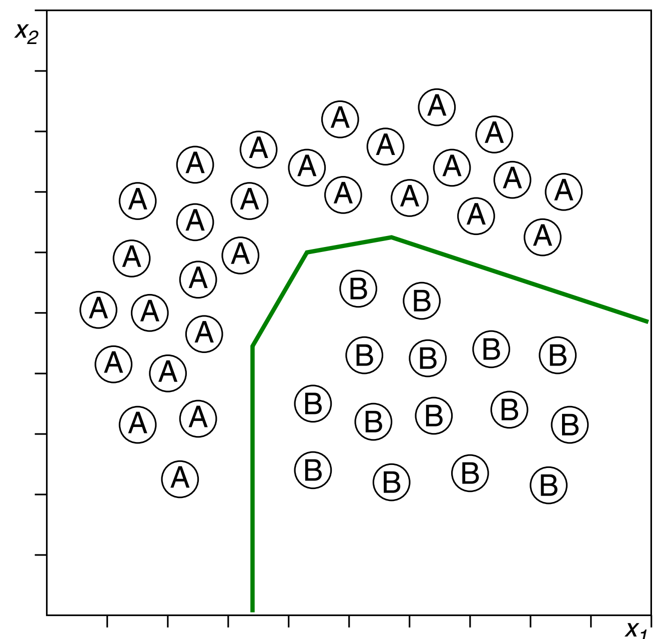
Definition 1 (Linear Separability)

Two sets of feature vectors, X_0, X_1, of a p-dimensional feature space are called linearly separable, if $p+1$ real numbers, $w_0, w_1, \ldots, w_p$, exist such that holds:

1. $\forall \mathbf{x} \in X_0$: $\sum_{j=0}^{p} w_j x_j < 0$
2. $\forall \mathbf{x} \in X_1$: $\sum_{j=0}^{p} w_j x_j \geq 0$



linearly separable



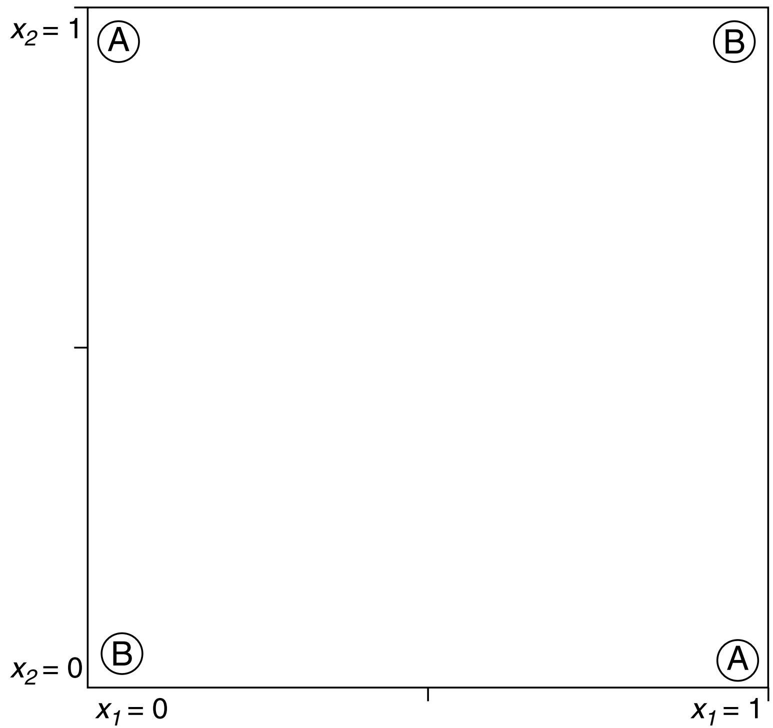
not linearly separable

Multilayer Perceptron

Separability

The *XOR* function defines the smallest example for two not linearly separable sets:

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

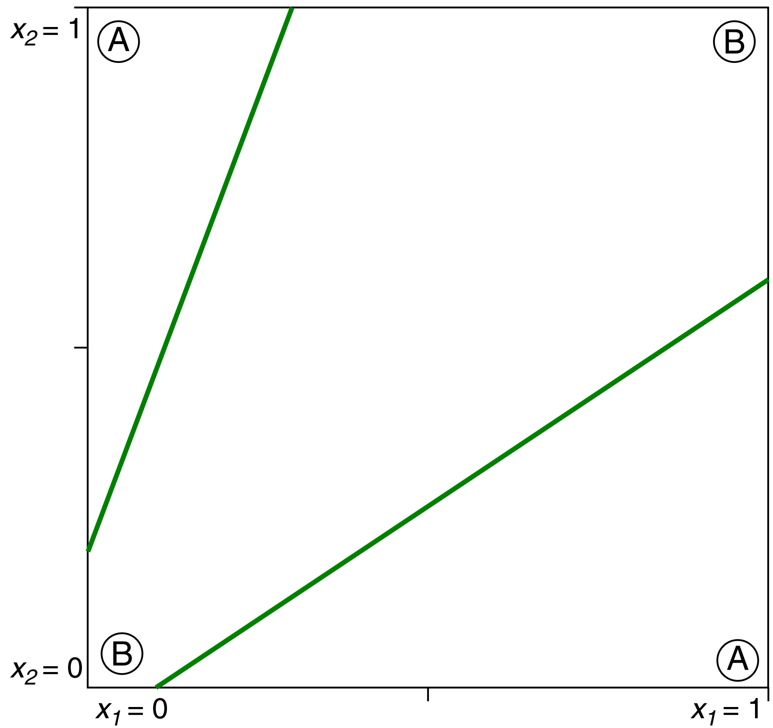


Multilayer Perceptron

Separability (continued)

The *XOR* function defines the smallest example for two not linearly separable sets:

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



➜ specification of several hyperplanes

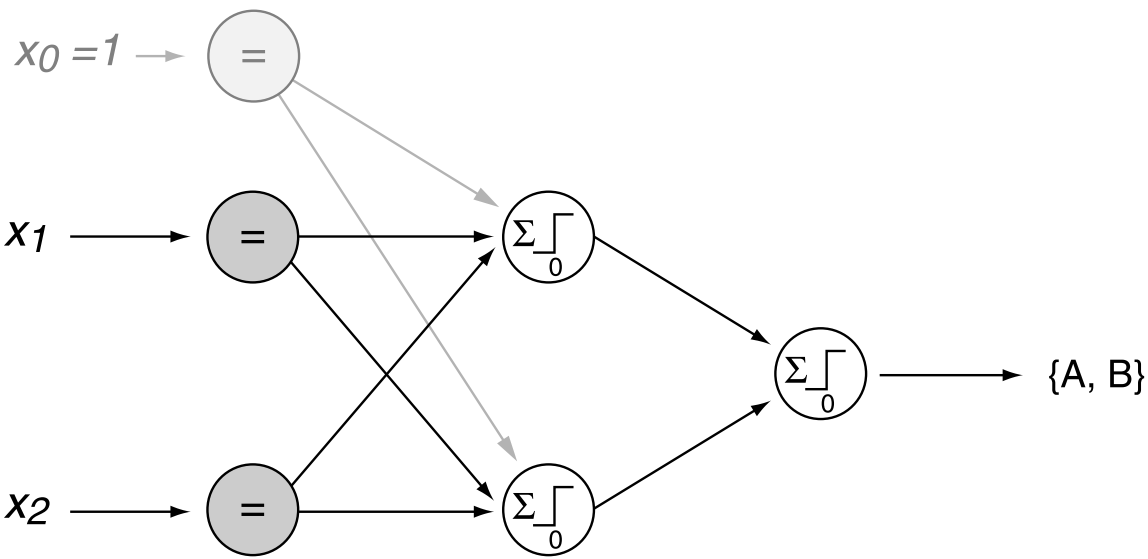
➜ combination of several perceptrons

Multilayer Perceptron

Separability (continued)

Layered combination of several perceptrons: the multilayer perceptron.

Minimum multilayer perceptron that is able to handle the *XOR* problem:



Remarks:

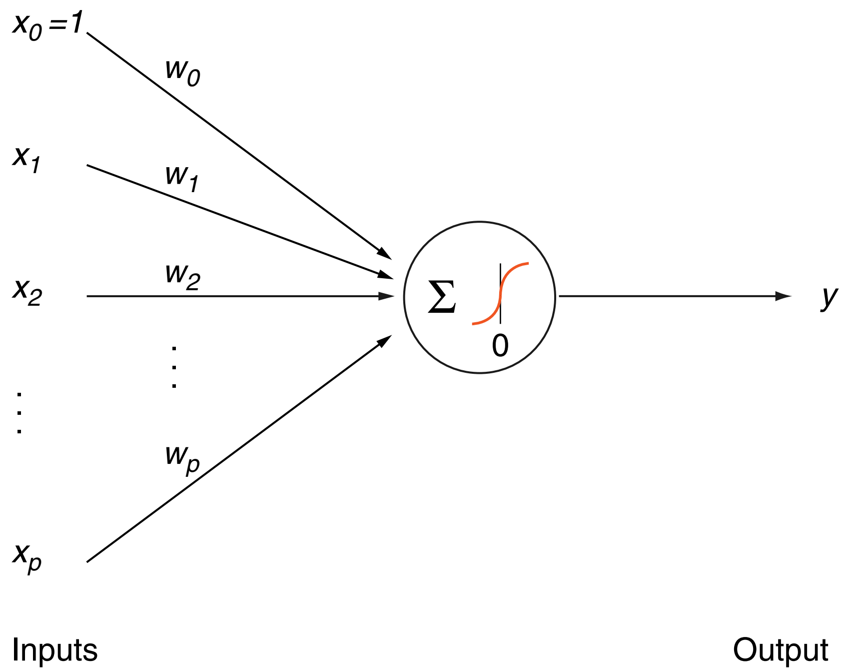
- The multilayer perceptron was presented by Rumelhart and McClelland in 1986. Earlier, but unnoticed, was a similar research work of Werbos and Parker [1974, 1982].
- Compared to a single perceptron, the multilayer perceptron poses a significantly more challenging training (= learning) problem, which requires continuous (and non-linear) threshold functions along wtih sophisticated learning strategies.
- Marvin Minsky and Seymour Papert showed 1969 with the *XOR* problem the limitations of single perceptrons. Moreover, they assumed that extensions of the perceptron architecture (such as the multilayer perceptron) would be similarly limited as a single perceptron. A fatal mistake. In fact, they brought the research in this field to a halt that lasted 17 years. [Berkeley]

[Marvin Minsky: MIT Media Lab, Wikipedia]

Multilayer Perceptron

Computation in the Network [Heaviside]

A perceptron with a continuous and non-linear threshold function:



The sigmoid function $\sigma(z)$ as threshold function:

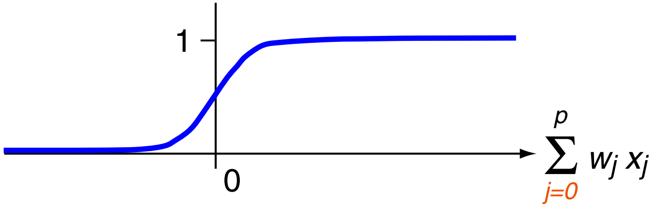
$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad \text{where} \quad \frac{d\sigma(z)}{dz} = \sigma(z) \cdot (1 - \sigma(z))$$

Multilayer Perceptron

Computation in the Network (continued)

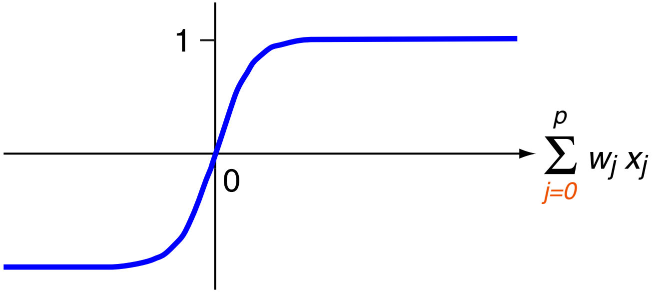
Computation of the perceptron output $y(\mathbf{x})$ via the sigmoid function σ:

$$y(\mathbf{x}) \;=\; \sigma(\mathbf{w}^T\mathbf{x}) \;=\; \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}}$$



An alternative to the sigmoid function is the $\tanh$ function:

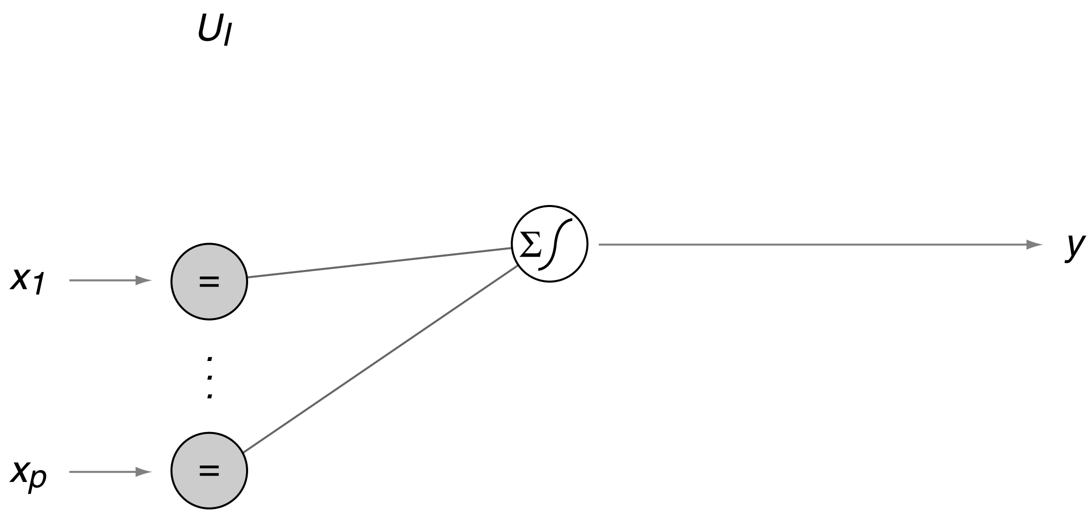
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x}-1}{e^{2x}+1}$$



Multilayer Perceptron

Computation in the Network (continued)

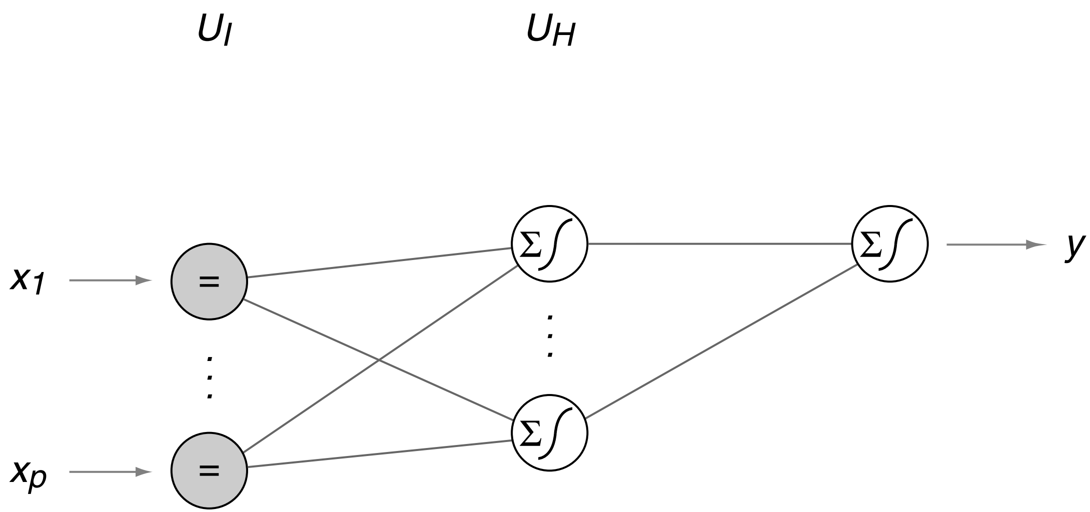
Distinguish units (nodes, perceptrons) of type input, hidden, and output:



Multilayer Perceptron

Computation in the Network (continued)

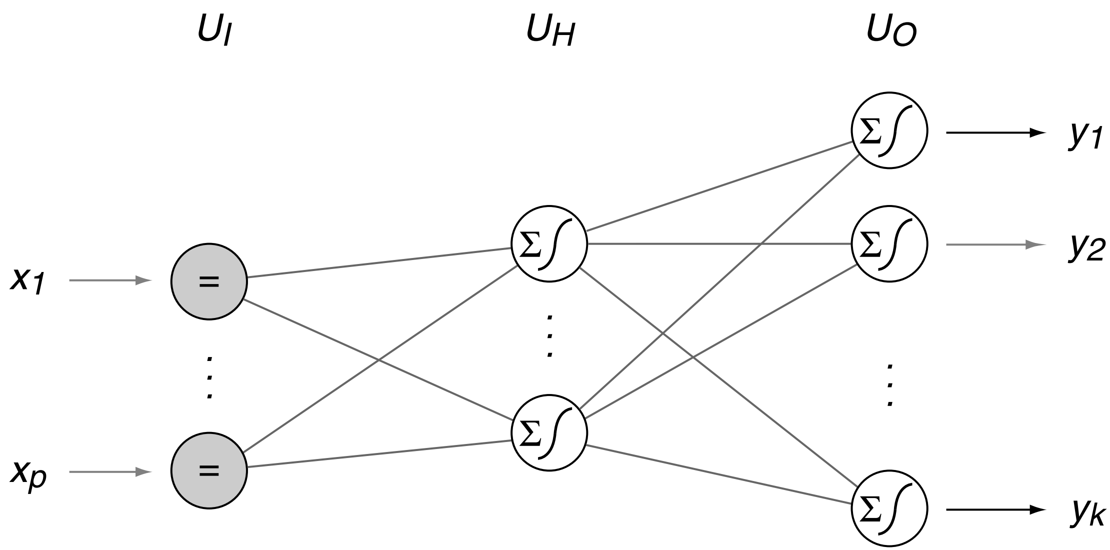
Distinguish units (nodes, perceptrons) of type input, hidden, and output:



Multilayer Perceptron

Computation in the Network (continued)

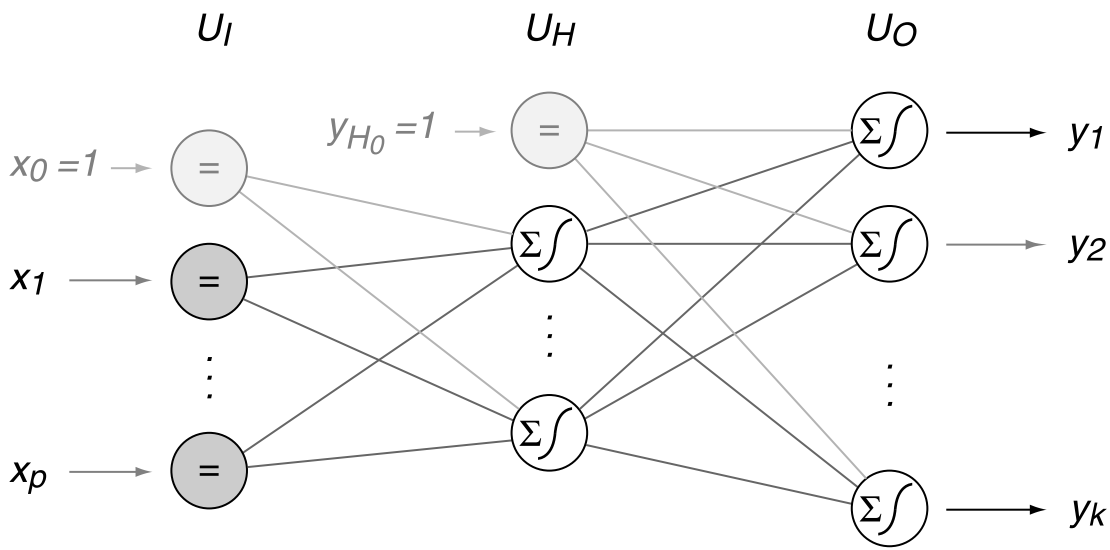
Distinguish units (nodes, perceptrons) of type input, hidden, and output:



Multilayer Perceptron

Computation in the Network (continued)

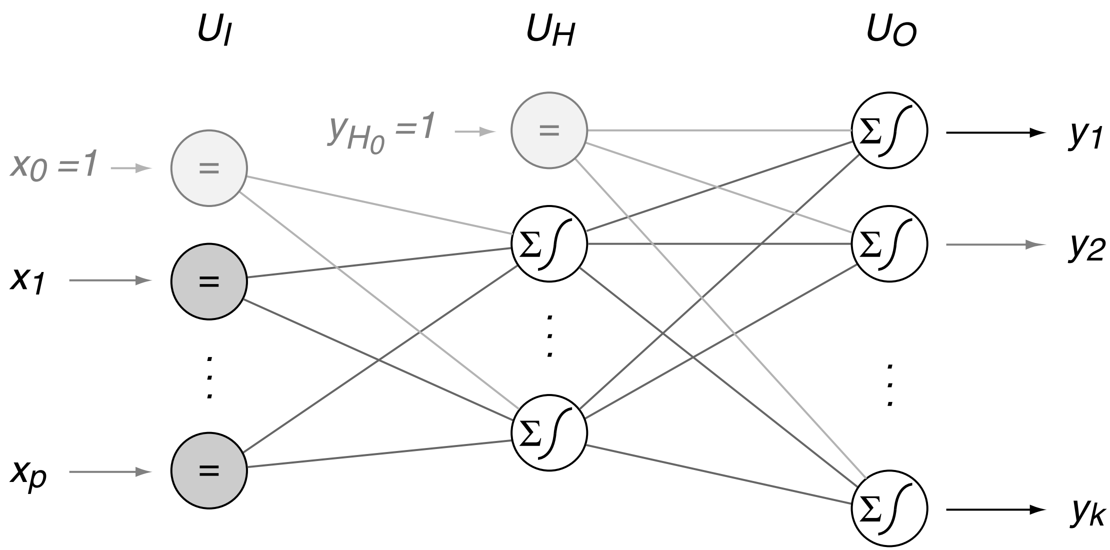
Distinguish units (nodes, perceptrons) of type input, hidden, and output:



Multilayer Perceptron

Computation in the Network (continued)

Distinguish units (nodes, perceptrons) of type input, hidden, and output:



U_I, U_H, U_O	Sets with units of type input, hidden, and output
w_{jk}, Δw_{jk}	Weight and weight adaptation for the edge connecting the units j and k
$x_{j \to k}$	Input value (single incoming edge) for unit k, provided at the output of unit j
y_k, δ_k	Output value and classification error of unit k
$\mathbf{w}_k$	Weight vector (all incoming edges) of unit k
$\mathbf{x}$	Input vector for a unit of the hidden layer
$\mathbf{y}_H$, $\mathbf{y}_O$	Output vector of the hidden layer and the output layer respectively

Remarks:

- The units of the input layer, U_I, perform no computations at all. They distribute the input values to the next layer.
- The network topology corresponds to a complete, bipartite graph between the units in U_I and U_H as well as between the units in U_H and U_O.
- The non-linear characteristic of the sigmoid function allows for networks that approximate every (computable) function. For this capability only three active layers are required, i.e., two layers with hidden units and one layer with output units. Keyword: universal approximator [Kolmogorov Theorem, 1957]
- Multilayer perceptrons are also called multilayer networks or (artificial) neural networks, ANN for short.

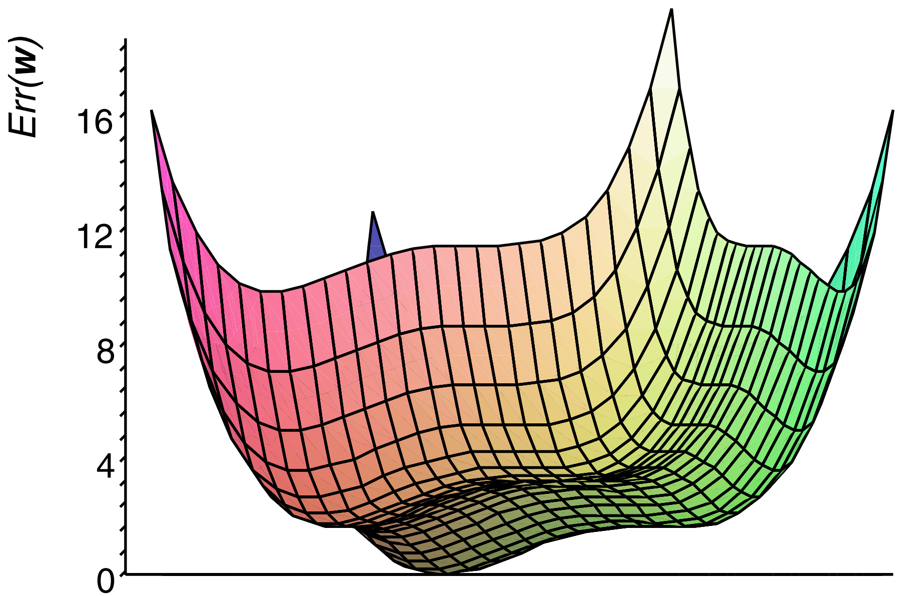
Multilayer Perceptron

Classification Error

The classification error *Err*$(\mathbf{w})$ is computed as sum over the $|U_O| = k$ network outputs:

$$\textit{Err}(\mathbf{w}) = \frac{1}{2} \sum_{(\mathbf{x},\mathbf{c}(\mathbf{x}))\in D} \sum_{v\in U_O} (c_v(\mathbf{x}) - y_v(\mathbf{x}))^2$$

Due its complex form, *Err*$(\mathbf{w})$ may contain various local minima:



Multilayer Perceptron

Weight Adaptation: Incremental Gradient Descent [network]

Algorithm: *MPT* Multilayer Perceptron Training

Input: D Training examples $(\mathbf{x}, c(\mathbf{x}))$ with $|\mathbf{x}| = p+1$, $c(\mathbf{x}) \in \{0,1\}^k$. $(c(\mathbf{x}) \in \{-1,1\}^k)$

η Learning rate, a small positive constant.

Output: $\mathbf{w}$ Weights of the units in U_I, U_H, U_O.

```
initialize_random_weights(U_I, U_H, U_O),  t = 0
REPEAT
  t = t + 1
  FOREACH (x, c(x)) ∈ D DO
    FOREACH u ∈ U_H DO y_u = σ(w_u^T x)   // compute output of layer 1
    FOREACH v ∈ U_O DO y_v = σ(w_v^T y_H)  // compute output of layer 2
  ENDDO
UNTIL(convergence(D, y_O(D)) OR t > t_max)
return(w)
```

Multilayer Perceptron

Weight Adaptation: Incremental Gradient Descent [network]

Algorithm:	*MPT*	Multilayer Perceptron Training
Input:	D	Training examples $(\mathbf{x}, c(\mathbf{x}))$ with $\lvert\mathbf{x}\rvert = p+1$, $c(\mathbf{x}) \in \{0,1\}^k$. $(c(\mathbf{x}) \in \{-1,1\}^k)$
	η	Learning rate, a small positive constant.
Output:	$\mathbf{w}$	Weights of the units in U_I, U_H, U_O.

initialize_random_weights(U_I, U_H, U_O), $t = 0$

REPEAT

  $t = t + 1$

  FOREACH $(\mathbf{x}, \mathbf{c}(\mathbf{x})) \in D$ DO

    FOREACH $u \in U_H$ **DO** $y_u = \sigma(\mathbf{w}_u^T \mathbf{x})$ // compute output of layer 1

    FOREACH $v \in U_O$ **DO** $y_v = \sigma(\mathbf{w}_v^T \mathbf{y}_H)$ // compute output of layer 2

    FOREACH $v \in U_O$ DO $\delta_v = y_v \cdot (1 - y_v) \cdot (\mathbf{c}_v(\mathbf{x}) - y_v)$ // backpropagate layer 2

    FOREACH $u \in U_H$ DO $\delta_u = y_u \cdot (1 - y_u) \cdot \sum_{v \in U_o} w_{uv} \cdot \delta_v$ // backpropagate layer 1

  ENDDO

UNTIL(*convergence*$(D, y_O(D))$ **OR** $t > t_{\text{max}}$)

return$(\mathbf{w})$

Multilayer Perceptron

Weight Adaptation: Incremental Gradient Descent [network]

Algorithm: *MPT* Multilayer Perceptron Training

Input: D Training examples $(\mathbf{x}, c(\mathbf{x}))$ with $|\mathbf{x}| = p+1,\ c(\mathbf{x}) \in \{0,1\}^k$. $(c(\mathbf{x}) \in \{-1,1\}^k)$

η Learning rate, a small positive constant.

Output: $\mathbf{w}$ Weights of the units in U_I, U_H, U_O.

initialize_random_weights(U_I, U_H, U_O), $t = 0$
REPEAT
  $t = t + 1$
  FOREACH $(\mathbf{x}, \mathbf{c}(\mathbf{x})) \in D$ DO
    FOREACH $u \in U_H$ **DO** $y_u = \sigma(\mathbf{w}_u^T \mathbf{x})$ // compute output of layer 1
    FOREACH $v \in U_O$ **DO** $y_v = \sigma(\mathbf{w}_v^T \mathbf{y}_H)$ // compute output of layer 2
    FOREACH $v \in U_O$ DO $\delta_v = y_v \cdot (1 - y_v) \cdot (\mathbf{c}_v(\mathbf{x}) - y_v)$ // backpropagate layer 2
    FOREACH $u \in U_H$ DO $\delta_u = y_u \cdot (1 - y_u) \cdot \sum_{v \in U_o} w_{uv} \cdot \delta_v$ // backpropagate layer 1
    FOREACH w_{jk}, $(j,k) \in (U_I \times U_H) \cup (U_H \times U_O)$ **DO**
      $\Delta w_{jk} = \eta \cdot \delta_k \cdot x_{j \to k}$
      $w_{jk} = w_{jk} + \Delta w_{jk}$
    ENDDO
  ENDDO
UNTIL(*convergence*$(D, y_O(D))$ **OR** $t > t_{\text{max}}$)
return$(\mathbf{w})$

Remarks:

- The generic delta rule (Lines 7 and 8 of the *MPT* algorithm) allows for a backpropagation of the classification error and hence the training of multi-layered networks.
- Gradient descent is based on the classification error of the entire network and hence considers the entire network weight vector.

Multilayer Perceptron

Weight Adaptation: Momentum Term

Momentum idea: a weight adaptation in iteration t considers the adaptation in iteration $t{-}1$:

$$\Delta w_{jk}(t) = \eta \cdot \delta_k \cdot x_{j\to k} \; + \; \alpha \cdot \Delta w_{jk}(t-1)$$

The term α, $0 \leq \alpha < 1$, is called “momentum”.

Multilayer Perceptron

Weight Adaptation: Momentum Term

Momentum idea: a weight adaptation in iteration t considers the adaptation in iteration $t-1$:

$$\Delta w_{jk}(t) = \eta \cdot \delta_k \cdot x_{j \to k} \;+\; \alpha \cdot \Delta w_{jk}(t-1)$$

The term α, $0 \leq \alpha < 1$, is called "momentum".

Effects:

- due the "adaptation inertia" local minima can be overcome
- if the direction of the descent does not change, the adaptation increment and, as a consequence, the speed of convergence is increased.