LEUPHANA
UNIVERSITÄT LÜNEBURG

INTRODUCTION TO MACHINE LEARNING

05. Neural Networks

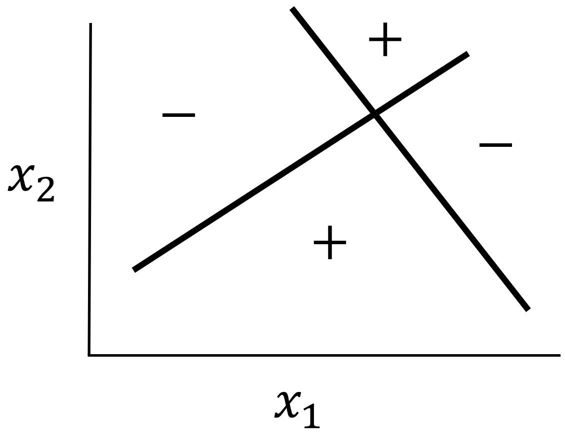
Winter 2020/2021

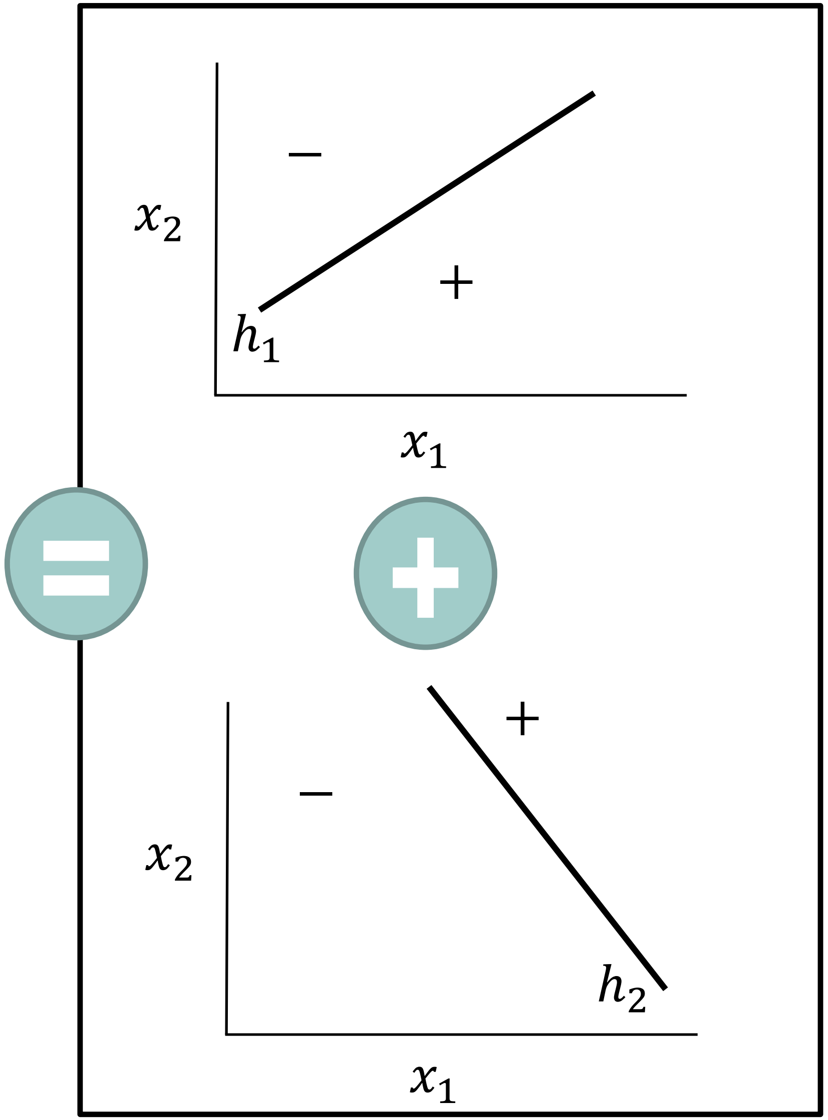
Acknowledgement: Many slides are adopted from Yaser Abu-Mostafa

Agenda

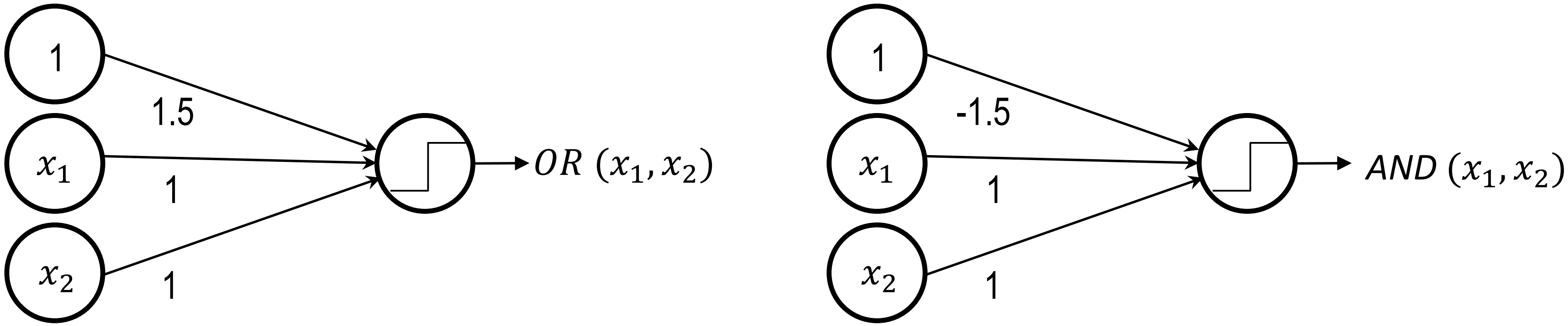
— Introduction
— Learning problem & linear classification
— Linear models: regression & logistic regression
— Non-linear transformation, overfitting & regularization
— Support Vector Machines and kernel learning
— **Neural Networks: shallow [and deep]**
— Theoretical foundation of supervised learning
— Unsupervised learning

The idea behind neural nets

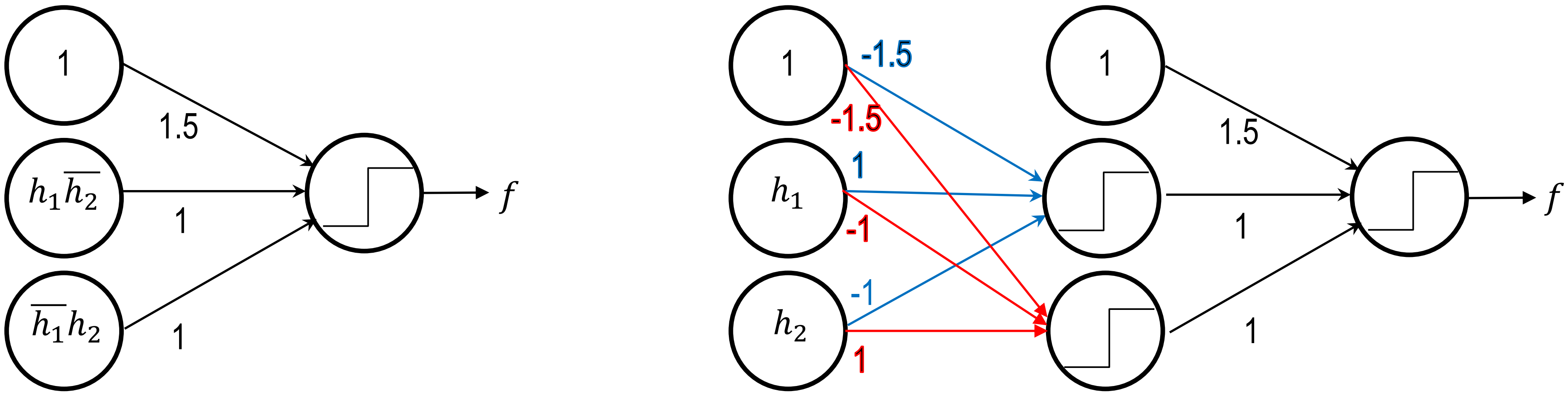




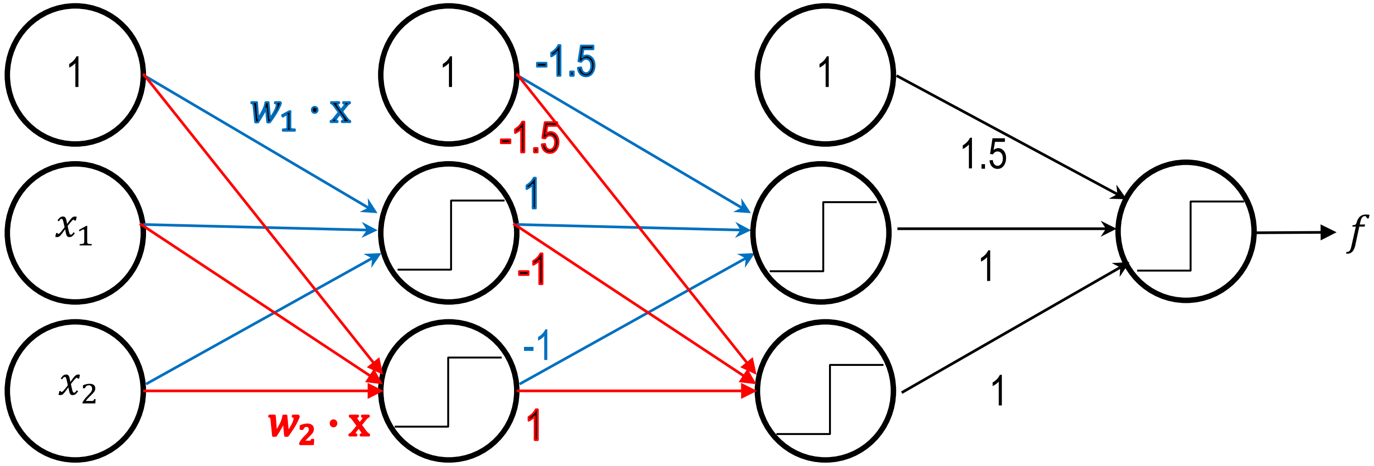
Combining perceptrons to realize Boolean operators



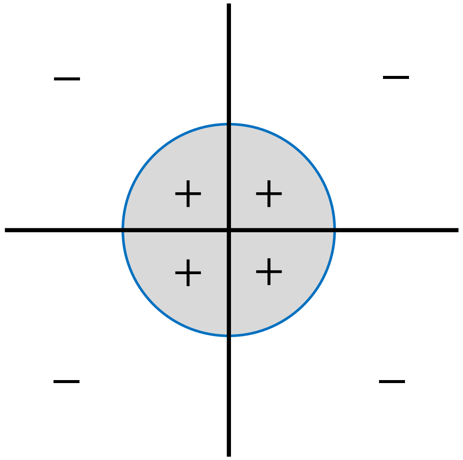
Creating layers



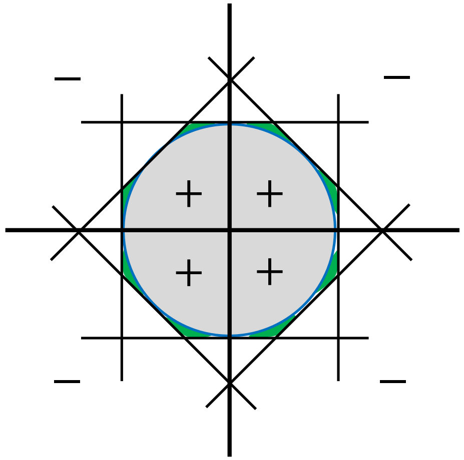
The multilayer perceptron



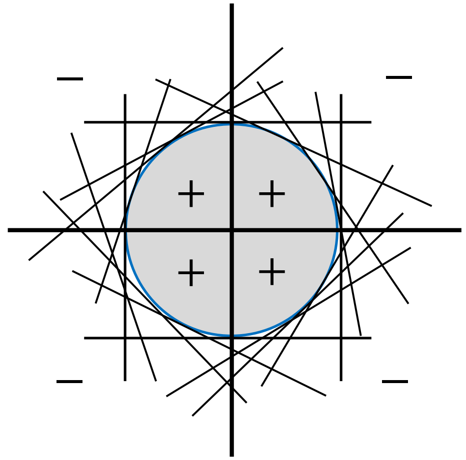
A powerful model



Target

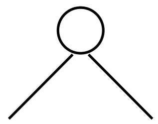
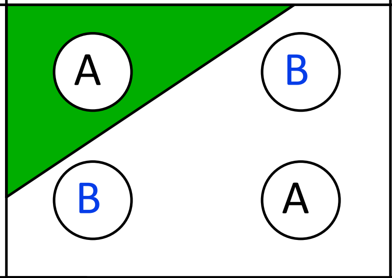
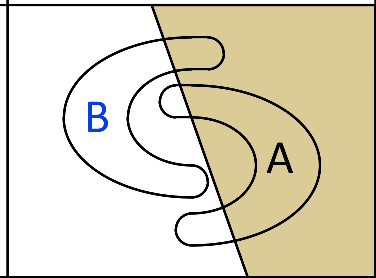
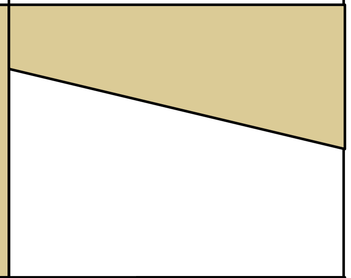
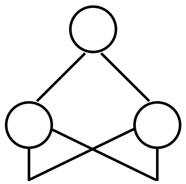
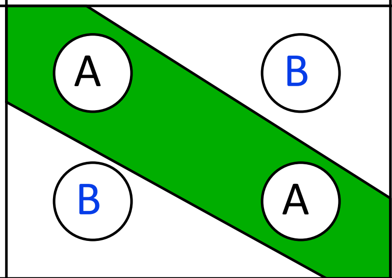
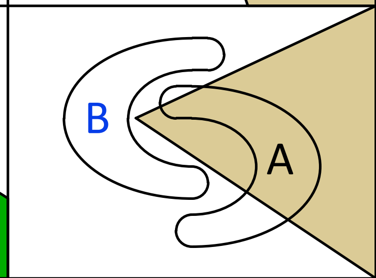
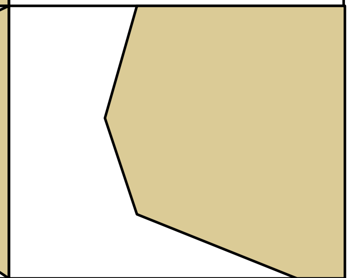
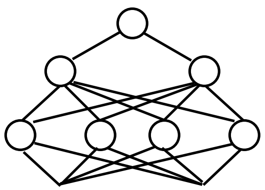
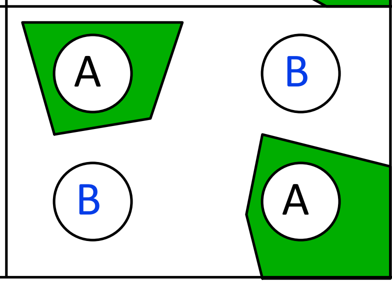
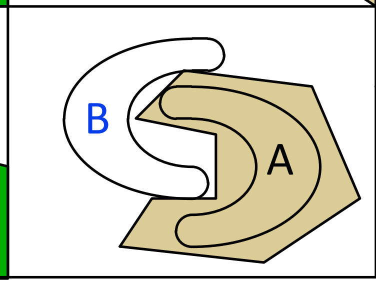
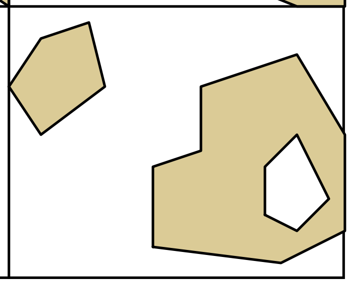


8 perceptrons



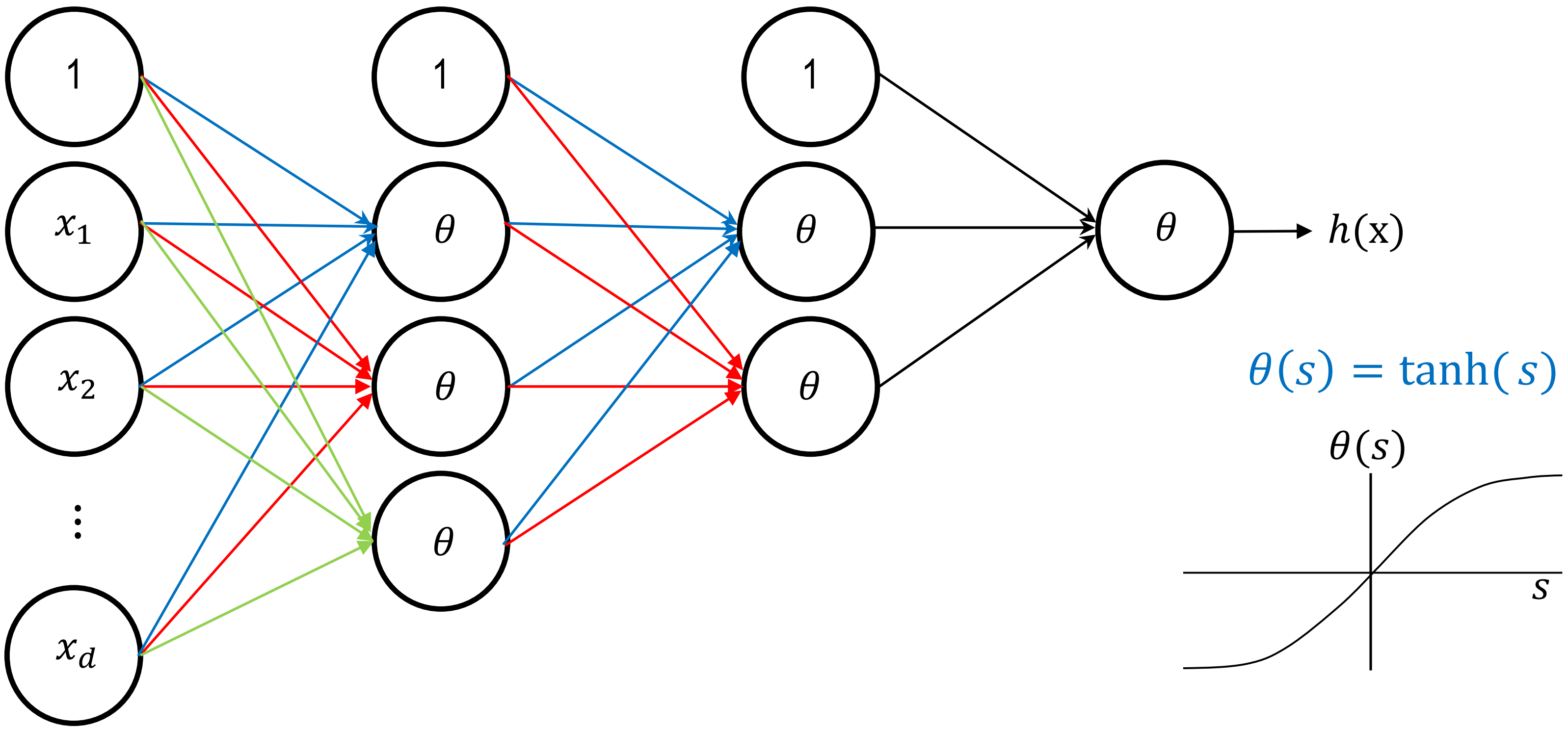
16 perceptrons

The impact of layers

Structure	*Types of Decision Regions*	*Exclusive-OR Problem*	*Classes with Meshed regions*	*Most General Region Shapes*
Single-Layer 	*Half Plane Bounded By Hyperplane*			
Two-Layer 	*Convex Open Or Closed Regions*			
Three-Layer 	Arbitrary (Complexity Limited by No. of Nodes)			

Source: Mark Hoogendoorn (VU Amsterdam)

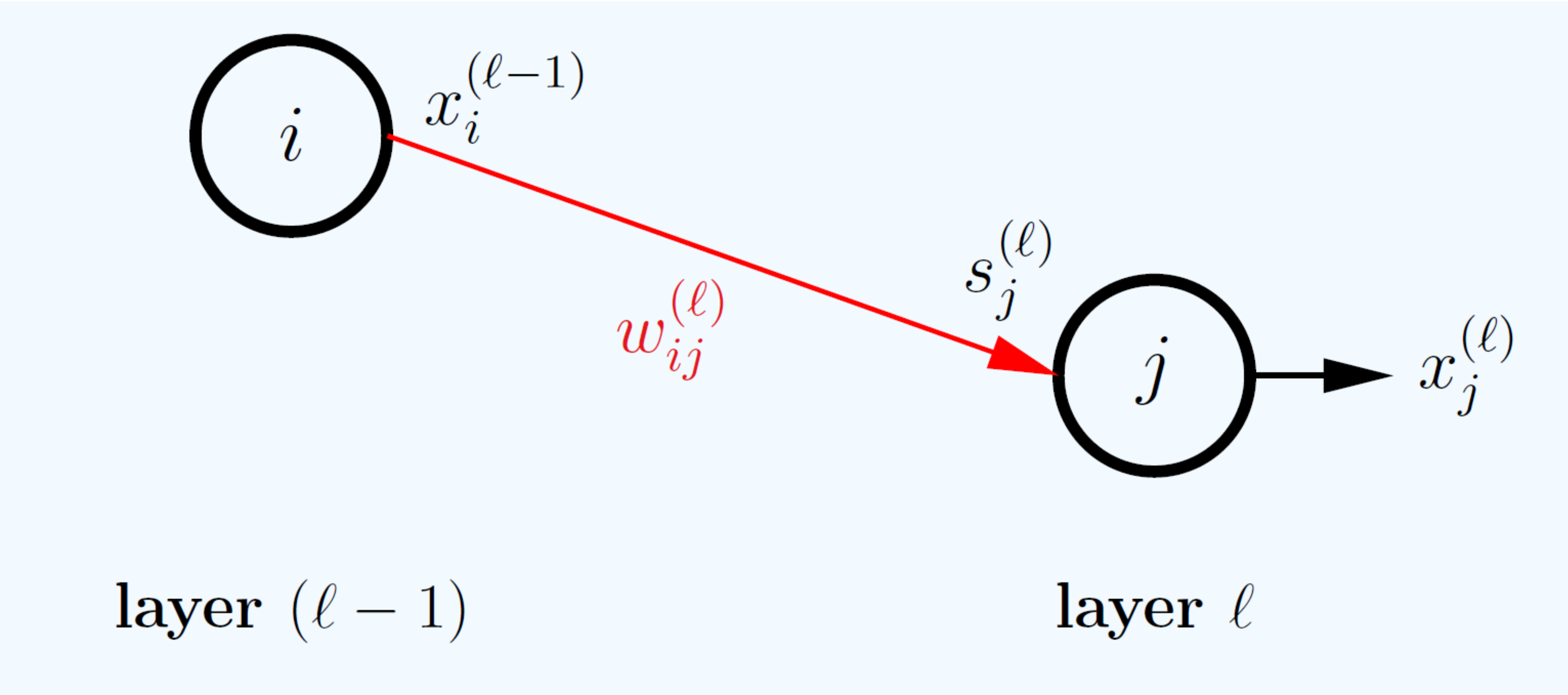
A simple neural network



$$\theta(s) = \tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}}$$

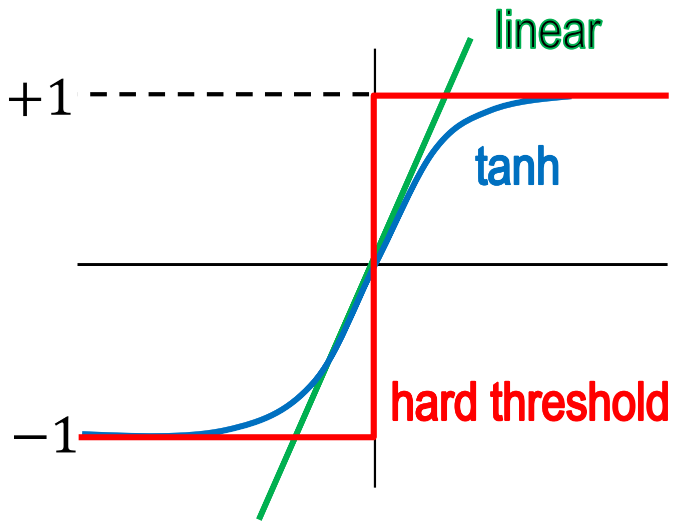
input **x**   hidden layers $1 \leq l < L$ output layer $l = L$

The weight between two nodes



Mathematical form

$$w_{ij}^{(l)} \begin{cases} 1 \le l \le L & \textit{layers} \\ 0 \le i \le d^{(l-1)} & \textit{inputs} \\ 1 \le j \le d^{(l)} & \textit{outputs} \end{cases}$$

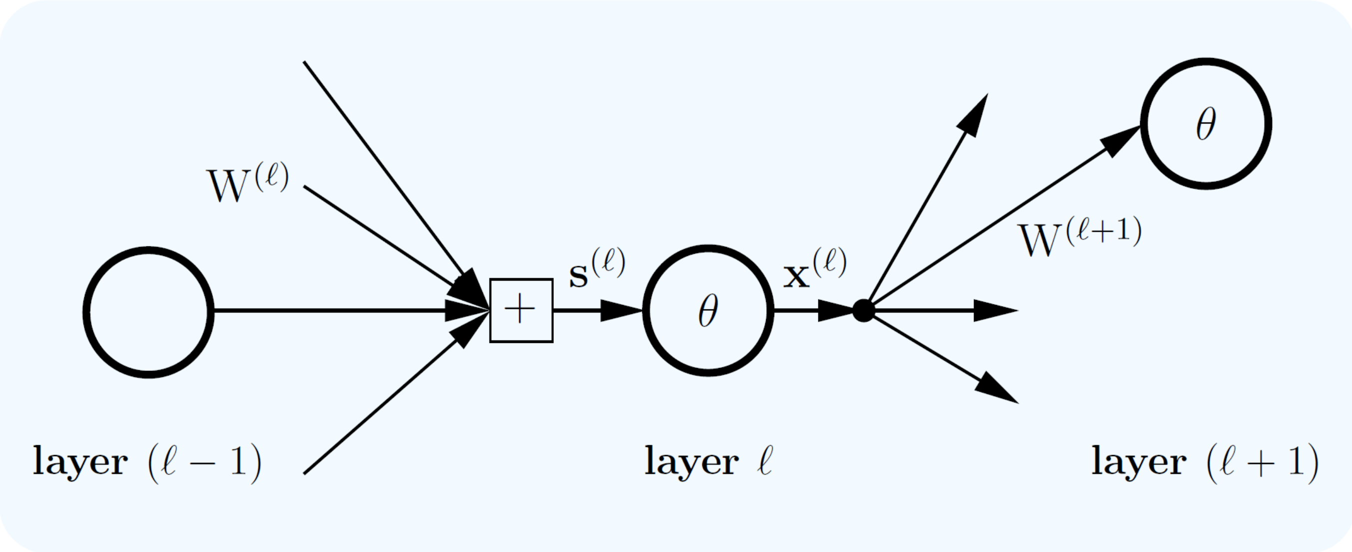


$$x_j^{(l)} = \theta\left(s_j^{(l)}\right) = \theta\left(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}\right)$$

Apply $\mathbf{x}$ to $x_1^{(0)} \cdots x_{d^{(0)}}^{(0)} \rightarrow \rightarrow x_1^L = h(\mathbf{x})$

$$\theta(s) = \tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}}$$

Vectorized version



layer ℓ parameters

signals in	$\mathbf{s}^{(\ell)}$	$d^{(\ell)}$ dimensional input vector
outputs	$\mathbf{x}^{(\ell)}$	$d^{(\ell)} + 1$ dimensional output vector
weights in	$W^{(\ell)}$	$(d^{(\ell-1)} + 1) \times d^{(\ell)}$ dimensional matrix
weights out	$W^{(\ell+1)}$	$(d^{(\ell)} + 1) \times d^{(\ell+1)}$ dimensional matrix

Forward Propagation

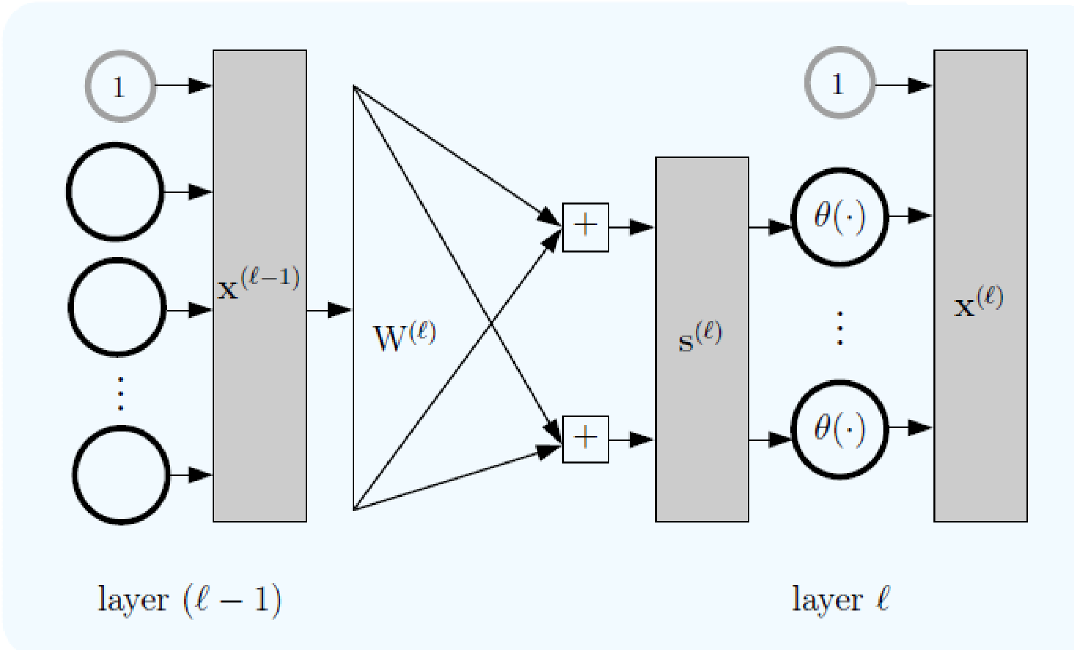
1: $\mathbf{x}^{(0)} \leftarrow \mathbf{x}$ **[Initialization]**

2: **for** $\ell = 1$ to L **do** **[Forward Propagation]**

3: $\quad \mathbf{s}^{(\ell)} \leftarrow (\mathrm{W}^{(\ell)})^{\mathrm{T}}\mathbf{x}^{(\ell-1)}$

4: $\quad \mathbf{x}^{(\ell)} \leftarrow \begin{bmatrix} 1 \\ \theta(\mathbf{s}^{(\ell)}) \end{bmatrix}$

5: $h(\mathbf{x}) = \mathbf{x}^{(L)}$ **[Output]**



Applying stochastic gradient descent

All the weights $\mathbf{w} = \left\{ w_{ij}^{(l)} \right\}$ determine $h(\mathbf{x})$

Error on example (x_n, y_n) is

$$\mathrm{e}\left(h(x_n), y_n\right) = \mathrm{e}(\mathbf{w})$$

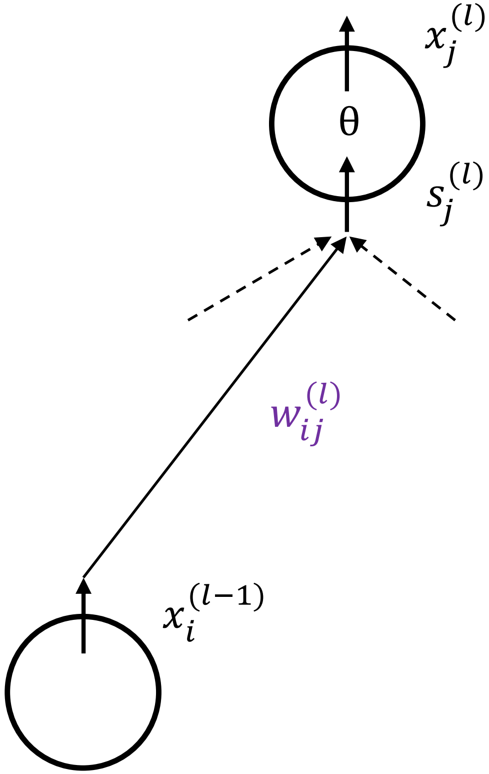
To implement SGD, we need the gradient

$$\nabla\, \mathrm{e}(\mathbf{w}): \quad \frac{\partial \mathrm{e}(\mathbf{w})}{\partial w_{ij}^{(l)}} \quad \text{for all } i, j, l$$

Remember the chain rule?

Computing $\dfrac{\partial\, \mathrm{e}(\mathbf{w})}{\partial\, w_{ij}^{(l)}}$

We can evaluate $\dfrac{\partial\, \mathrm{e}(\mathbf{w})}{\partial\, w_{ij}^{(l)}}$ one by one: analytically or numerically



A trick for efficient computation:

$$\frac{\partial\, \mathrm{e}(\mathbf{w})}{\partial\, w_{ij}^{(l)}} = \frac{\partial\, \mathrm{e}(\mathbf{w})}{\partial\, s_j^{(l)}} \times \frac{\partial\, s_j^{(l)}}{\partial\, w_{ij}^{(l)}}$$

We have $\dfrac{\partial\, s_j^{(l)}}{\partial\, w_{ij}^{(l)}} = x_i^{(l-1)}$ We only need: $\dfrac{\partial\, \mathrm{e}(\mathbf{w})}{\partial\, s_j^{(l)}} = \delta_j^{(l)}$

δ for the final layer

$$\delta_j^{(l)} = \frac{\partial\, \mathrm{e}(\mathbf{w})}{\partial\, s_j^{(l)}}$$

For the final layer $\boldsymbol{l} = \boldsymbol{L}$ and $\mathbf{j} = \mathbf{1}$:

$$\delta_1^{(L)} = \frac{\partial\, \mathrm{e}(\mathbf{w})}{\partial\, s_1^{(L)}}$$

$$\mathrm{e}(\mathbf{w}) := (x_1^{(L)} - y_n)^{\,2}$$

$$x_1^{(L)} = \theta\,(s_1^{(L)})$$

$$\theta'(s) = 1 - \theta^2(s)$$ for the tanh

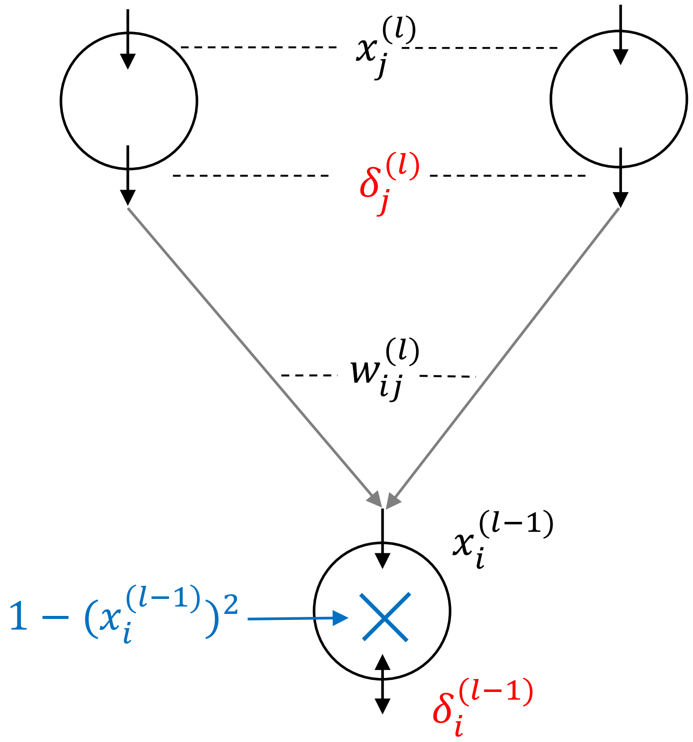
Back propagation δ

$$\delta_i^{(l-1)} = \frac{\partial\, \mathrm{e}(\mathbf{w})}{\partial\, s_i^{(l-1)}}$$

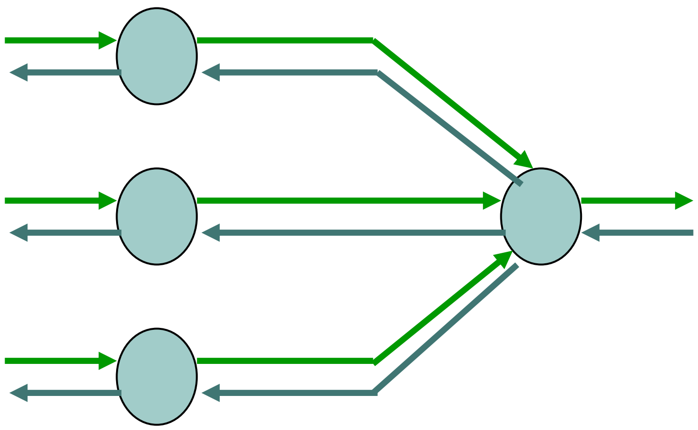
$$= \sum_{j=1}^{d^{(l)}} \frac{\partial\, \mathrm{e}(\mathbf{w})}{\partial\, s_j^{(l)}} \times \frac{\partial\, s_j^{(l)}}{\partial\, x_i^{(l-1)}} \times \frac{\partial\, x_i^{(l-1)}}{\partial\, s_i^{(l-1)}}$$

$$= \sum_{j=1}^{d^{(l)}} \delta_j^{(l)} \times w_{ij}^{(l)} \times \theta'(s_i^{(l-1)})$$

$$\delta_i^{(l-1)} = \left(1 - (x_i^{(l-1)})^2\right) \sum_{j=1}^{d^{(l)}} w_{ij}^{(l)}\ \delta_j^{(l)}$$



Combing feedforward and backpropagation



Network activation
Error computation
Forward Step

Error propagation
Backward Step

Source: Mark Hoogendoorn (VU Amsterdam)

Backpropagation algorithm

Initialize all weights $w_{ij}^{(l)}$ at random

for $\boldsymbol{t} = \mathbf{0}, \mathbf{1}, \mathbf{2}, \dots$ **do**

Pick $\boldsymbol{n} \in \{\mathbf{1}, \mathbf{2}, \dots, \boldsymbol{N}\}$

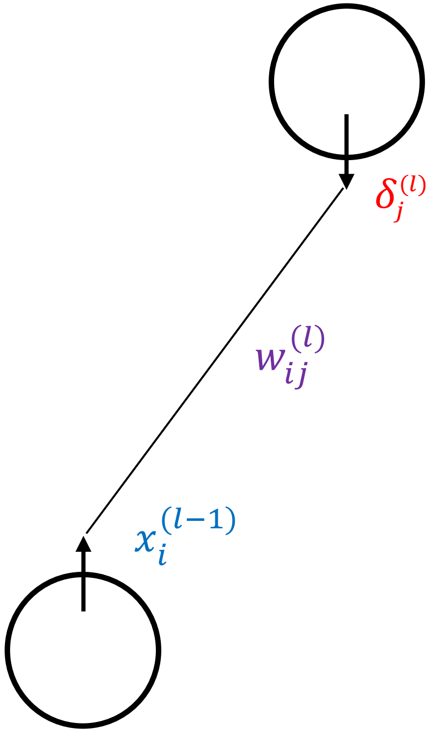
Forward: Compute all $x_j^{(l)}$

Backward: Compute all $\delta_j^{(l)}$

Update the weights: $w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta\, x_i^{(l-1)} \delta_j^{(l)}$

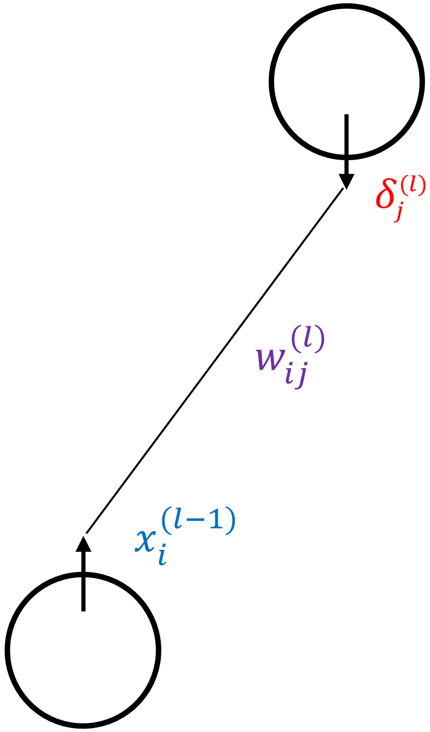
Iterate to the next step until it is time to stop

Return the final weights $w_{ij}^{(l)}$



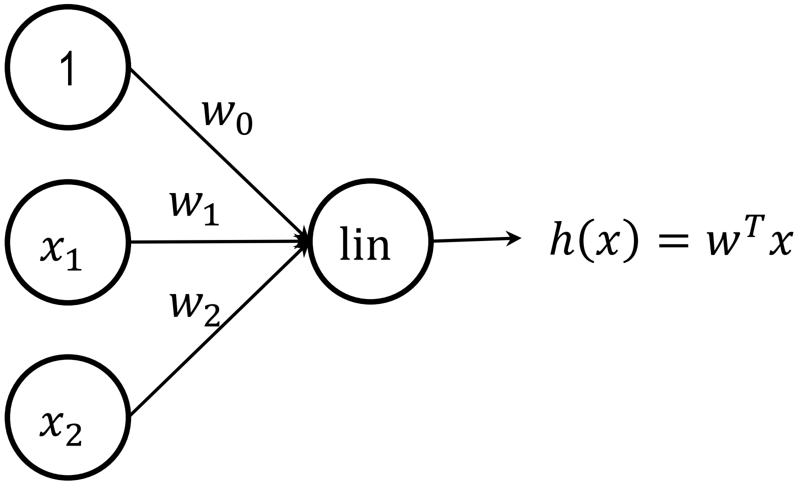
A few words on the Backpropagation Algorithm

- Backpropagation Algorithm (BA) searches for weight values that minimize the total error of the network by iterative application of forward and backward pass
- Convergence is not guaranteed (e.g. learning rate too big or too small)
- If BA convergences it is not guaranteed, that we reached the global minimum



A final word: neural networks for regression

- The backprop algorithm does not heavily depend on the activation function
- If we substitute the $\tanh(s)$ output node just by the linear function (s) something magic happens



- Any continuous function can be approximated by an NN with 2 hidden layers

Deep Learning