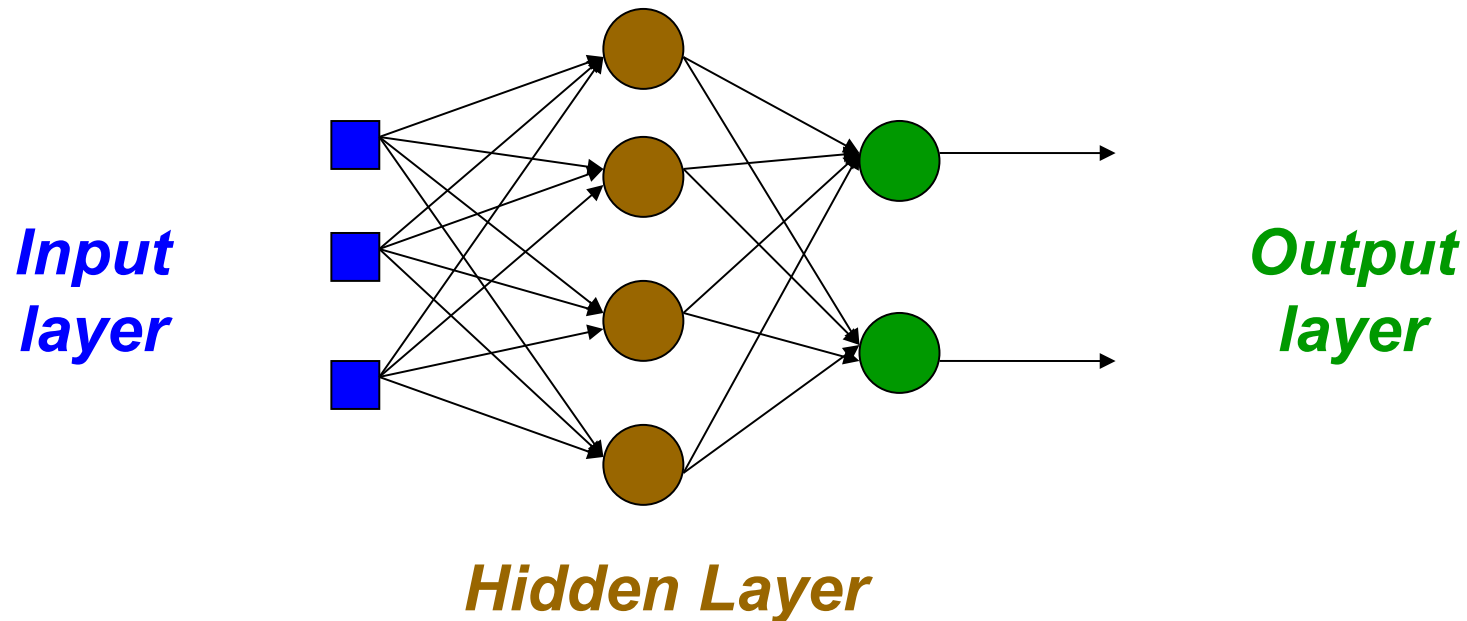


# 2. Multi-layer Perceptron (MLP)

## Gradient Descent and Backpropagation

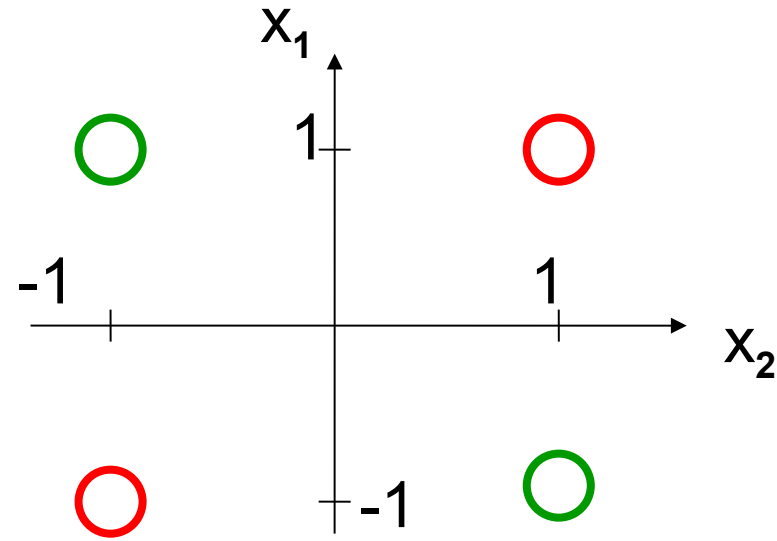
---

- Single perceptrons = linear decision boundary
- The XOR problem cannot be solved by a single perceptron
- MLPs overcome the limitation of single-layer perceptrons
- How to train them ? Gradient Descent => Backpropagation!



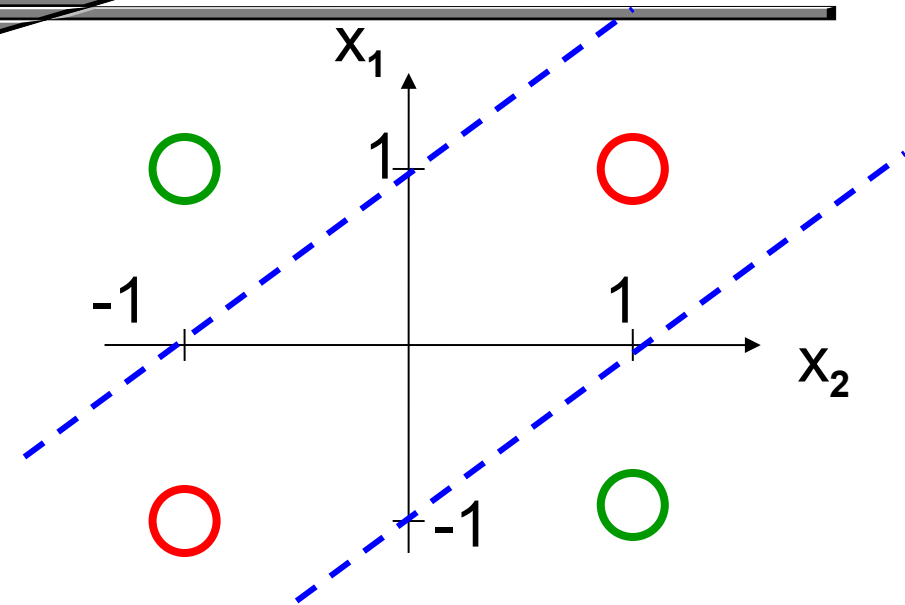
# The XOR problem: not linearly separable

$x_1$	$x_2$	$x_1 \text{ xor } x_2$
-1	-1	-1
-1	1	1
1	-1	1
1	1	-1

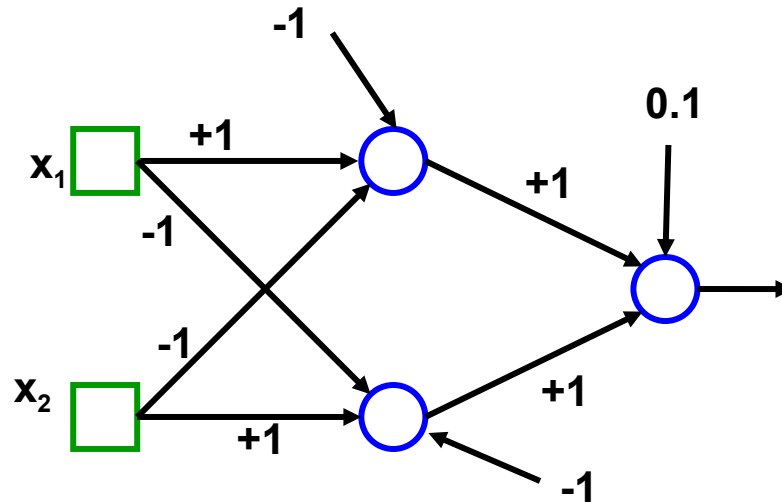


# A solution for the XOR problem

$x_1$	$x_2$	$x_1 \text{ XOR } x_2$
-1	-1	-1
-1	1	1
1	-1	1
1	1	-1



CHECK IT!

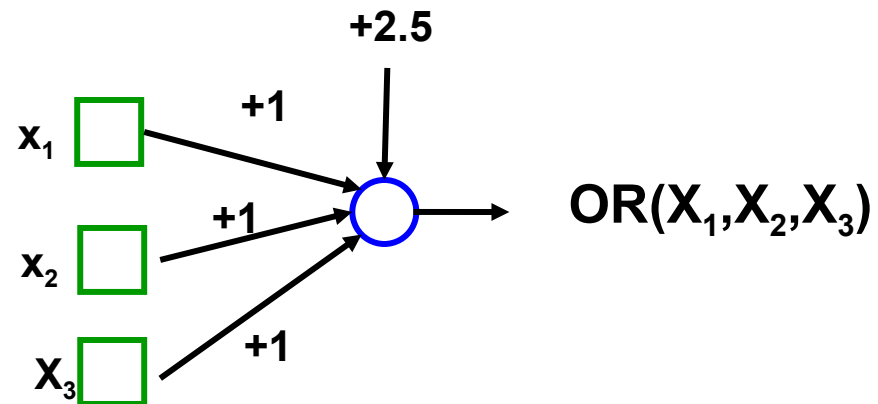
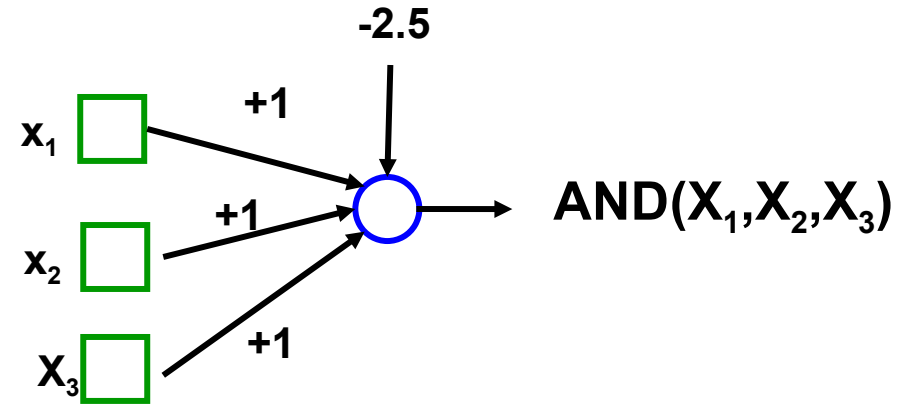


$$\varphi(v) = \begin{cases} 1 & \text{if } v > 0 \\ -1 & \text{if } v \leq 0 \end{cases}$$

$\varphi$  is the sign function.

What about more complicated problems?

# Generalized AND and OR

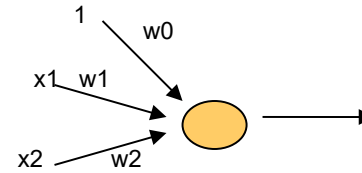


**Generalized AND (OR) can be computed by a single perceptron!**

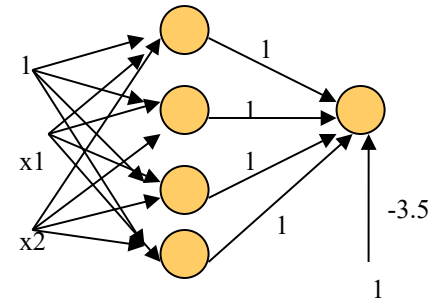
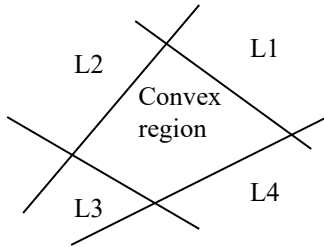
# Types of decision regions

$$w_0 + w_1x_1 + w_2x_2 > 0$$

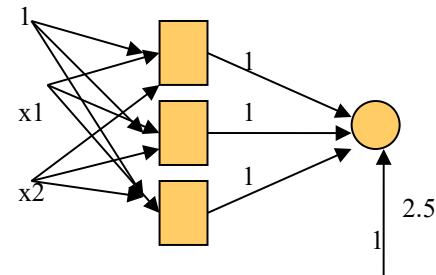
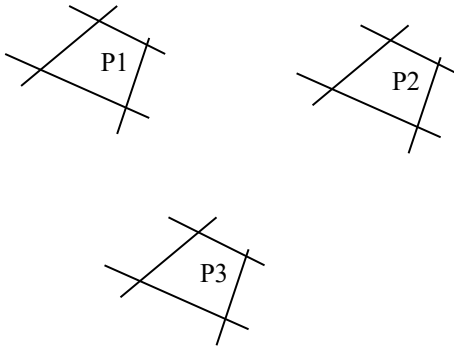
$$w_0 + w_1x_1 + w_2x_2 < 0$$



Network  
with a single  
node

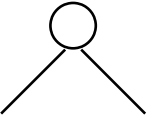
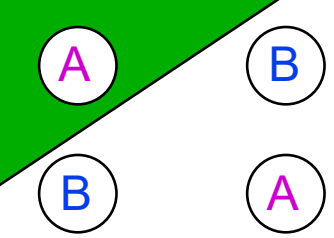
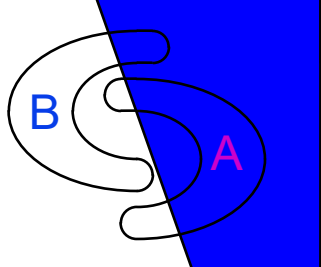
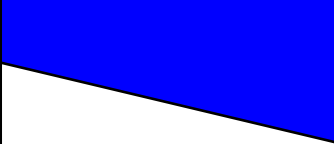
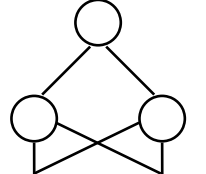
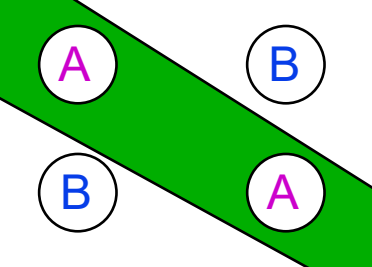
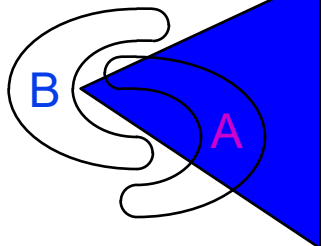
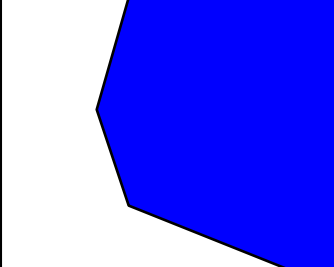
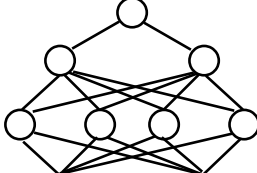
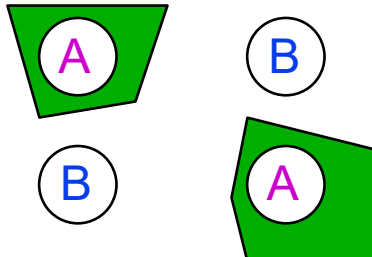
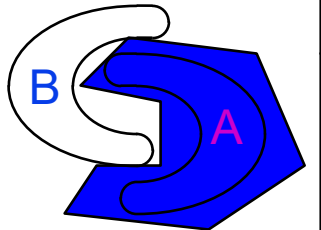
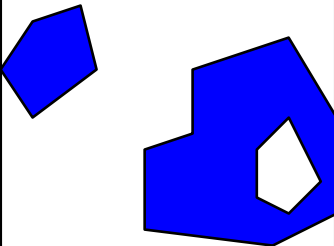


One-hidden layer network that  
realizes the convex region: each  
hidden node realizes one of the  
lines bounding the convex region



two-hidden layer network that  
realizes the union of three convex  
regions: each box represents a one  
hidden layer network realizing  
one convex region

# Different Non-Linearly Separable Problems

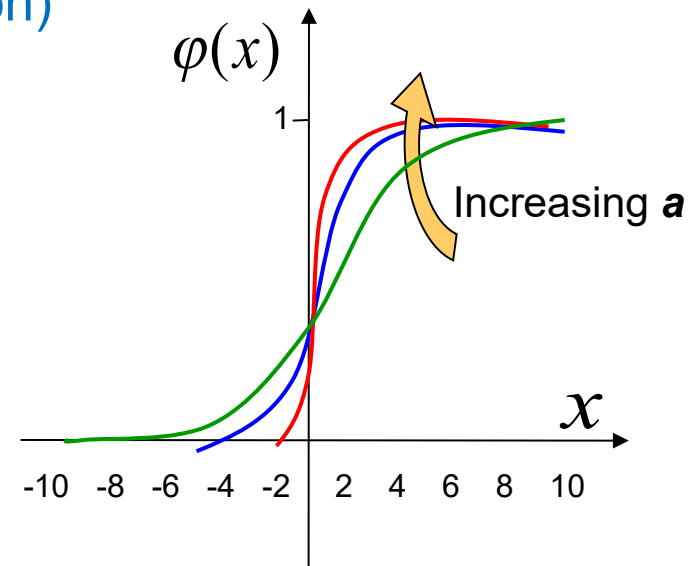
<i>Structure</i>	<i>Types of Decision Regions</i>	<i>Exclusive-OR Problem</i>	<i>Classes with Meshed regions</i>	<i>Most General Region Shapes</i>
<i>Single-Layer</i> 	<i>Half Plane Bounded By Hyperplane</i>			
<i>Two-Layer</i> 	<i>Convex Open Or Closed Regions</i>			
<i>Three-Layer</i> 	<i>Arbitrary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

It also works in more than 2 dimensions!

# How to train multi-layer networks?

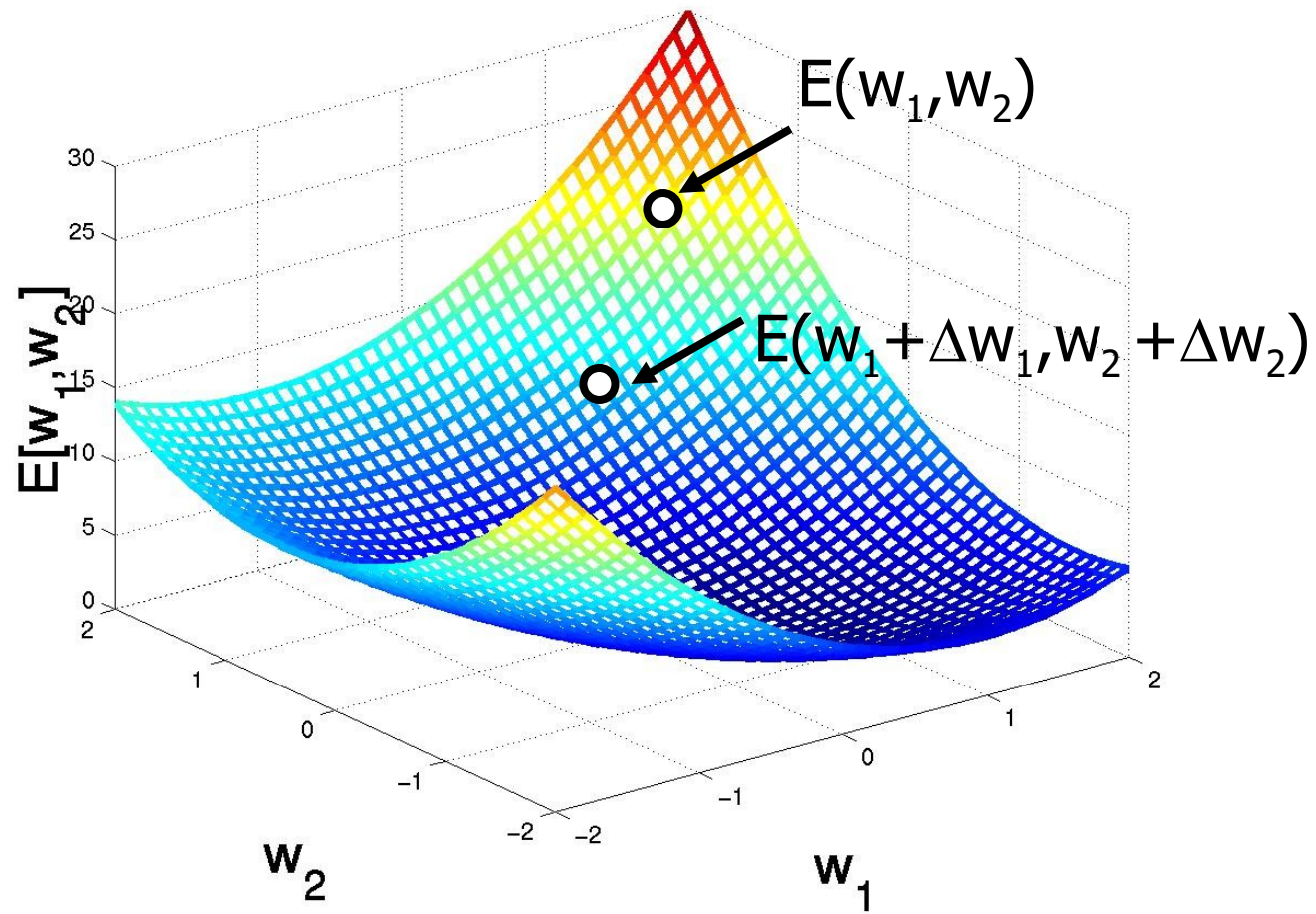
- Main Idea:  
Replace the sign function by its “smooth approximation” and use the gradient descent algorithm to find weights that minimize the error (as with Logistic Regression)

$$\varphi(x) = \frac{1}{1+e^{-ax}} \text{ with } a > 0$$



- The function to be optimized is a complicated one, with many variables and nestings ...
- **FORTUNATELY:** the final update rules are simple !

# Gradient Descent





# Weight Update Rule

---

**gradient descent method:** “walk” in the direction yielding the maximum decrease of the network error  $E$ . This direction is the opposite of the gradient of  $E$ .

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}}$$

*direction (negative gradient)*

$$w_{ji} = w_{ji} + \Delta w_{ji}$$

*update rule*

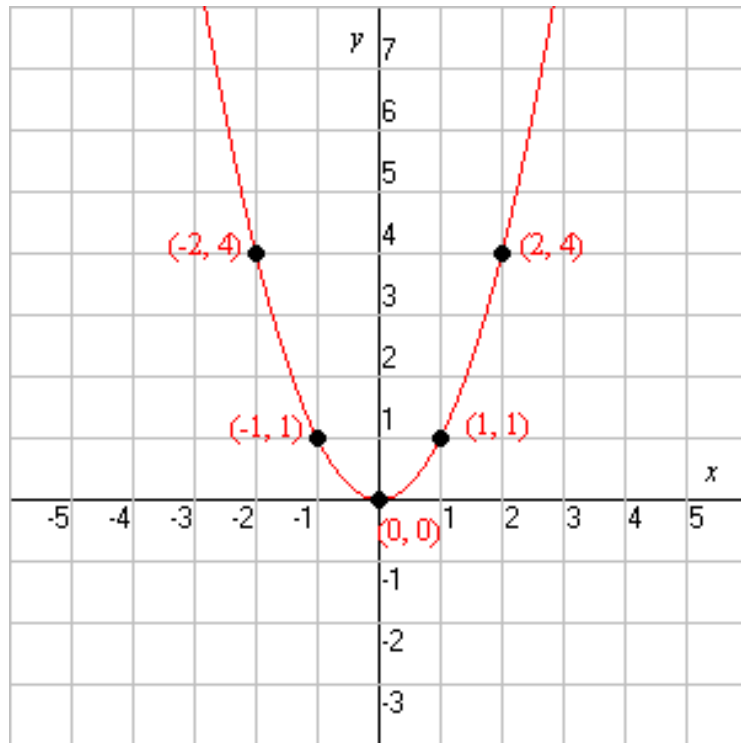
$$w_{ji} = w_{ji} - \eta \frac{\partial E}{\partial w_{ji}}$$

*could be written in one rule!?*

*(in practice we first calculate all gradients and then update all weights)*

**Example:** *find the minimum of  $f(x)=x^2$*

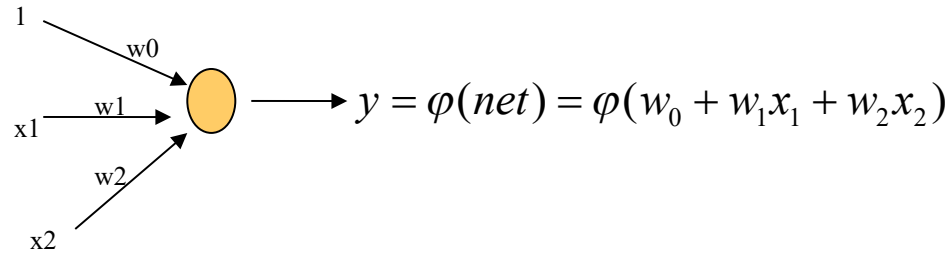
---



- Start at  $x_0=1.0$  (or  $x_0=2.0$ )
- $f'(x)=2x$ , so the update rule:
$$x_{n+1} = x_n - 2x_n$$
- Consider several scenario's:
  - $=1.5 \Rightarrow ?$
  - $=1.0 \Rightarrow ?$
  - $=0.75 \Rightarrow ?$
  - $=0.5 \Rightarrow ?$
  - $=0.25 \Rightarrow ?$

# Delta Rule: one node

- Derive the Delta rule for the following network



$$\varphi(x) = \frac{1}{1+e^{-ax}} \quad \text{with } a > 0$$

$$E(w_0, w_1, w_2) = \frac{1}{2} (y - d)^2 = \frac{1}{2} (\varphi(w_0 + w_1x_1 + w_2x_2) - d)^2$$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \text{ for } i \text{ in } \{0,1,2\}$$

- We need to find  $\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}$

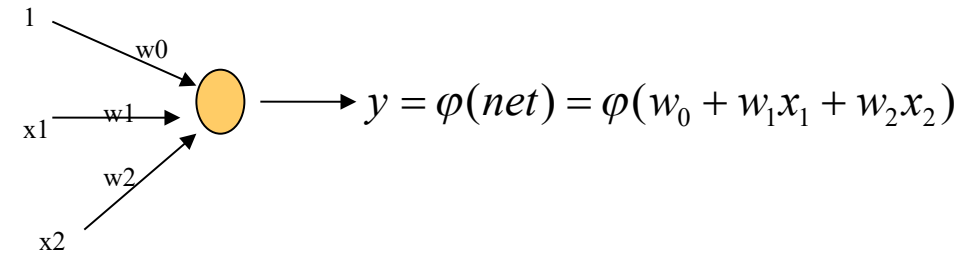
Chain Rule:  
 $f(g(x))' = f'(g(x)) * g'(x)$

## Delta Rule: one node

$$\begin{aligned}\frac{\partial E(w_0, w_1, w_2)}{\partial w_1} &= \frac{1}{2} \frac{\partial (\varphi(w_0 + w_1 x_1 + w_2 x_2) - d)^2}{\partial w_1} \\&= \frac{1}{2} 2(\varphi(w_0 + w_1 x_1 + w_2 x_2) - d) \frac{\partial (\varphi(w_0 + w_1 x_1 + w_2 x_2) - d)}{\partial w_1} \\&= (\varphi(net) - d) \varphi'(net) \frac{\partial (w_0 + w_1 x_1 + w_2 x_2)}{\partial w_1} \\&= (\varphi(net) - d) \varphi'(net) x_1\end{aligned}$$

$$\frac{\partial E(w_0, w_1, w_2)}{\partial w_2} = (\varphi(net) - d) \varphi'(net) x_2$$

$$\frac{\partial E(w_0, w_1, w_2)}{\partial w_0} = (\varphi(net) - d) \varphi'(net)$$



# Delta Rule: one node

---

Thus the update rules for the weights are:

$$\begin{aligned}\Delta w_0 &= \eta(d \overset{\text{negative gradient}}{-} \varphi(\text{net}))\varphi'(\text{net}) \cdot 1 \\ \Delta w_1 &= \eta(d - \varphi(\text{net}))\varphi'(\text{net})x_1 \\ \Delta w_2 &= \eta(\underbrace{d - \varphi(\text{net})}_{\text{delta}})\varphi'(\text{net})\underbrace{x_2}_{\text{input}}\end{aligned}$$

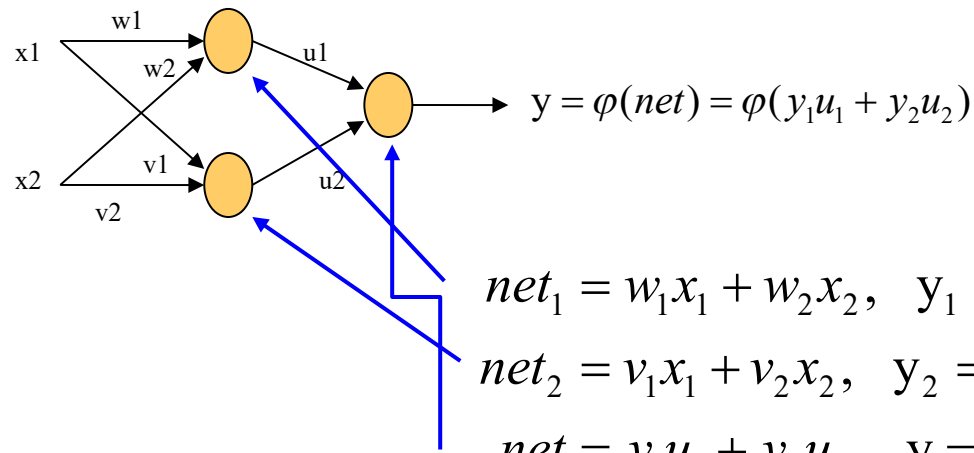
It's handy to know that for the logistic sigmoid function:

$\varphi(x) = 1/(1 + \exp(-x))$  we have:  $\varphi'(x) = \varphi(x)(1 - \varphi(x)) = \text{output}(1 - \text{output})$ ,

so once we know  $\varphi(x)$  we also know  $\varphi'(x)$  !

# Delta Rule: XOR-network

Consider the network (no biases):



$$net_1 = w_1x_1 + w_2x_2, \quad y_1 = \varphi(net_1)$$

$$net_2 = v_1x_1 + v_2x_2, \quad y_2 = \varphi(net_2)$$

$$net = y_1u_1 + y_2u_2, \quad y = \varphi(net)$$

$$= \varphi(y_1u_1 + y_2u_2) =$$

$$= \varphi(\varphi(net_1)u_1 + \varphi(net_2)u_2) =$$

$$= \varphi(\varphi(w_1x_1 + w_2x_2)u_1 + \varphi(v_1x_1 + v_2x_2)u_2)$$

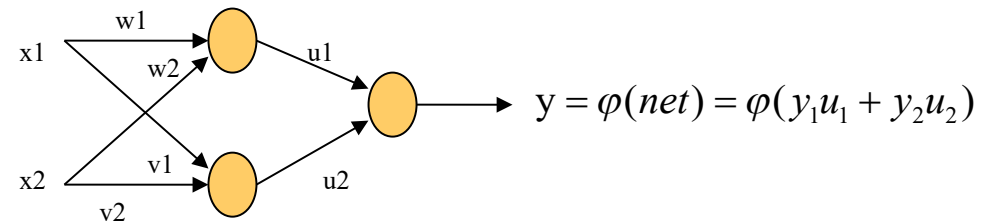
Output as a function of weights

# Derivation of the Delta Rule

$$\begin{aligned}\frac{\partial E}{\partial u_1} &= \frac{1}{2} \frac{\partial (\varphi(y_1 u_1 + y_2 u_2) - d)^2}{\partial u_1} = \frac{1}{2} 2(\varphi(y_1 u_1 + y_2 u_2) - d) \frac{\partial (\varphi(y_1 u_1 + y_2 u_2) - d)}{\partial u_1} \\ &= (y - d) \varphi'(net) \frac{\partial (y_1 u_1 + y_2 u_2)}{\partial u_1} = \underline{(y - d) \varphi'(net) y_1}\end{aligned}$$

From similar calculations we get:  $\frac{\partial E}{\partial u_2} = \underline{(y - d) \varphi'(net) y_2}$

$$\begin{aligned}\frac{\partial E}{\partial w_1} &= \frac{1}{2} \frac{\partial (\varphi(y_1 u_1 + y_2 u_2) - d)^2}{\partial w_1} = \frac{1}{2} 2(\varphi(y_1 u_1 + y_2 u_2) - d) \frac{\partial (\varphi(y_1 u_1 + y_2 u_2) - d)}{\partial w_1} \\ &= (y - d) \varphi'(net) \frac{\partial (y_1 u_1 + y_2 u_2)}{\partial w_1}\end{aligned}$$



# Derivation of the Delta Rule

$$\begin{aligned}\frac{\partial(y_1u_1 + y_2u_2)}{\partial w_1} &= \frac{\partial(y_1u_1)}{\partial w_1} = u_1 \frac{\partial y_1}{\partial w_1} = u_1 \frac{\partial(\varphi(w_1x_1 + w_2x_2))}{\partial w_1} \\ &= u_1\varphi'(net_1) \frac{\partial(w_1x_1)}{\partial w_1} = u_1\varphi'(net_1)x_1\end{aligned}$$

So we obtain:

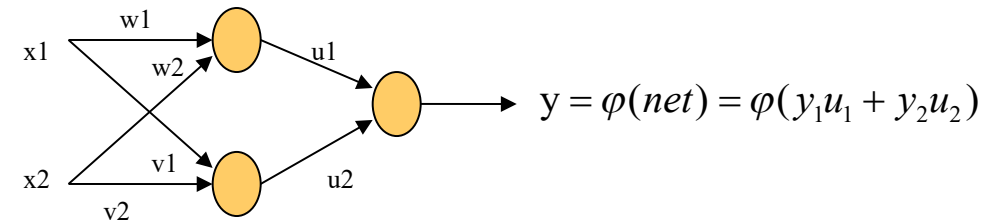
$$\begin{aligned}\frac{\partial E}{\partial w_1} &= (y - d)\varphi'(net) \frac{\partial(\varphi(y_1u_1 + y_2u_2))}{\partial w_1} \\ &= \underline{(y - d)\varphi'(net)u_1\varphi'(net_1)x_1}\end{aligned}$$

From similar calculations we get:

$$\frac{\partial E}{\partial w_2} = \underline{(y - d)\varphi'(net)u_1\varphi'(net_1)x_2}$$

*delta*

$$\frac{\partial E}{\partial v_1} = \underline{(y - d)\varphi'(net)u_2\varphi'(net_2)x_1}, \quad \frac{\partial E}{\partial v_2} = \underline{(y - d)\varphi'(net)u_2\varphi'(net_2)x_2}$$



*Around 2xdepth multiplications => exploding/vanishing gradients problem!* 16



# Key idea behind Backpropagation

---

- The output of a MLP is a composition of two sorts of functions:
  - Linear (weighted) combinations of inputs
  - Activation functions (e.g., sigmoid, logistic, etc.)
  - *Additionally, an error function (loss function) at the output layer*
- Backpropagation is based on an observation that computing partial derivatives of the error function involves multiple application of the chain rule applied to simple functions (linear and sigmoid\*) which leads to a very efficient algorithm that involves 3 phases:
  - Computing the output of the network and the corresponding error (forward pass)
  - Computing the contribution of each weight to the error (backward pass)
  - Adjusting the weights accordingly (to the contribution to error).

\* (or another activation function)

# General case: Generalized Delta Rule

---

$$\Delta w_{ji} = \eta \delta_j y_i, \text{ where:}$$

$$\delta_j = \begin{cases} \varphi'(v_j)(d_j - y_j) & \text{IF } j \text{ output node} \\ \varphi'(v_j) \sum_{\substack{k \text{ of next layer}}} \delta_k w_{kj} & \text{IF } j \text{ hidden node} \end{cases}$$

$w_{ji}$  : weight from node  $j$  to node  $i$  (moving from output to input!)

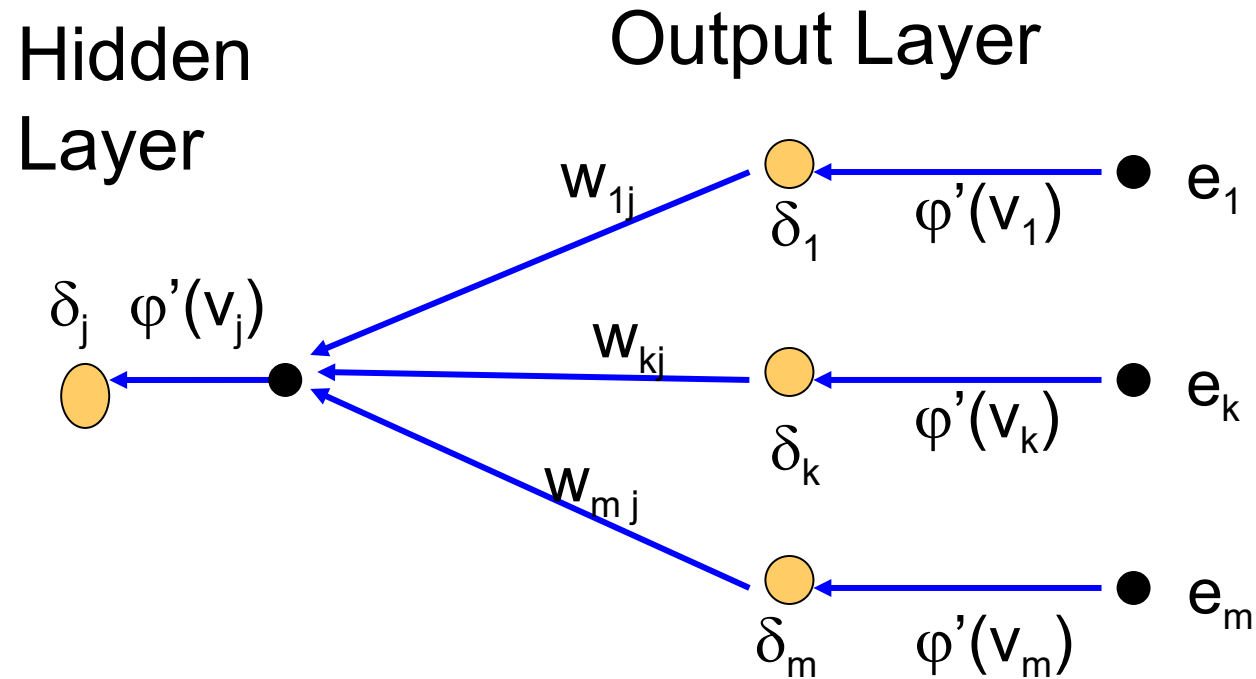
$\eta$  : learning rate (constant)

$y_j$  : output of node  $j$

$v_j$  : activation of node  $j$

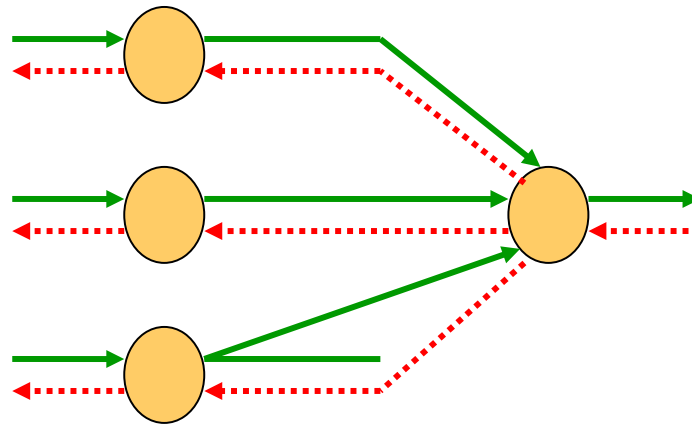
# Error backpropagation

The flow-graph below illustrates how errors are back-propagated from  $m$  output nodes to the hidden neuron  $j$



# Backpropagation: two phases

---



## **Forward Pass:**

→ propagate the input to the output and calculate the error

## **Backward Pass:**

←..... Calculate the required adjustments of weights by propagating the error from the output layer to the input layer; **adjust weights accordingly**

# Backpropagation algorithm

---

- The Backprop algorithm searches for weight values that **minimize the total error of the network**
- Backprop consists of the **repeated** application of the following two passes:
  - **Forward pass**: in this step the network is activated on **one example** and the **error** of each neuron of the output layer is computed. Also **activations** of all hidden nodes are computed.
  - **Backward pass**: in this step the network error is used for updating the weights. Starting at the output layer, **the error is propagated backwards through the network, layer by layer** with help of the generalized delta rule. Finally, **all weights are updated**.
- Weights are initialized at random (how?)
- No guarantees of convergence (why?)  
(when learning rate too big or too small)
- In case of convergence: local (or global) minimum

Instead of processing examples one after another we can process batches !

Error function is the sum\* of errors over training examples, so the gradient of the error function is the sum\* of gradients over all training cases!

## Three update strategies:

expensive

**Full Batch mode** (all inputs at once; conceptually “correct”)

Weights are updated after all the inputs are processed

most popular

**(Mini) Batch mode** (a small, random sample of inputs; “approximate”)

Weights are updated after all a small random sample of inputs is processed (stochastic gradient descent)

???

**On-line mode** (one input at a time: stochastic gradient descent?)

Weights are updated after processing single inputs

# Advantages Stochastic Gradient Descent

- **Additional randomness helps to avoid local minima**
- **Huge savings of the CPU-time**
- **Easy to execute on GPU cards**
- “Approximated gradient” works almost the same as “exact gradient” (almost the same convergence rate)

# Stopping criteria

---

- **total mean squared error change**: Backprop is considered to have converged when the absolute rate of change in the average squared error per epoch is sufficiently small
- **generalization based criterion**: After each epoch the NN is tested for generalization using a separate set of examples (**validation set**). If the generalization performance is adequate then stop. (**Early Stopping**)

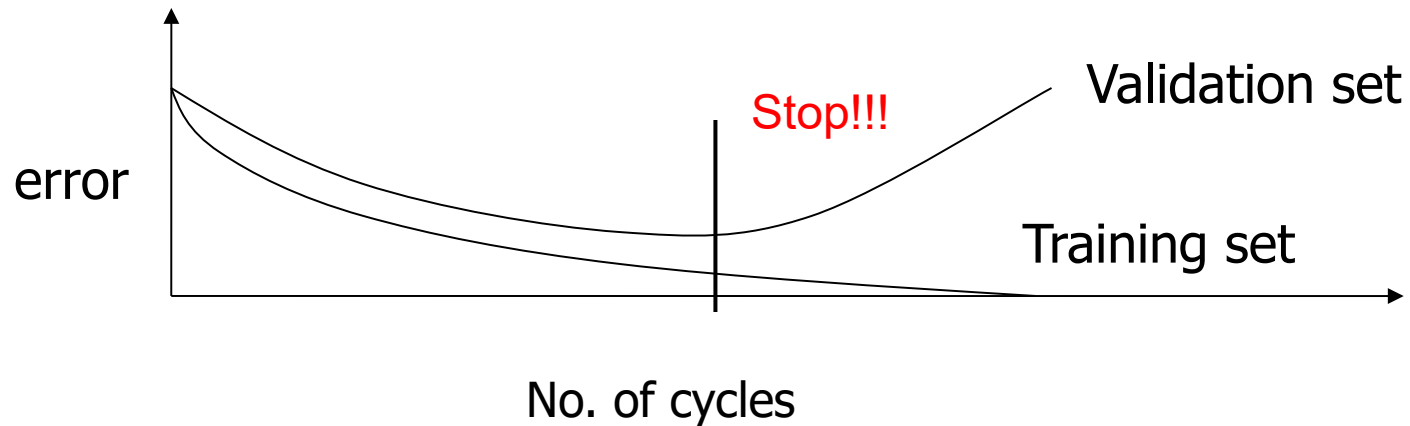
**Epoch**: a single pass through all training cases



# Early Stopping

Training network for too many cycles may lead to **data overfitting** (or “overtraining”)

**Early Stopping:**  
stop training as soon as the error on  
the validation set increases



# Early stopping and 3 sets:

---

- **Training set** – used for training
- **Validation set** – used to decide when to stop training by monitoring the error on it; not used for training!
- **Test set** – used for final estimation of the performance of the trained neural network

# Model Selection by Cross-validation

---

- **Too few hidden units** may prevent the network from learning adequately the data and learning the concept.
- **Too many hidden units** leads to overfitting.
- A **cross-validation scheme** can be used to determine an appropriate number of hidden units by using the optimal test error to select the model with optimal number of hidden layers and nodes.
- N-fold cross-validation:
  1. split your data into N parts (equal size);
  2. develop N networks on all combinations of (N-1) parts;
  3. test each network on the remaining parts (test sets);
  4. average the error over these test sets.

In "DL-practice" too expensive!

# Expressive Power of MLP

---

## Boolean functions:

- **Every boolean function** can be implemented by a network with a **single hidden layer**

*but it might require **exponentially many neurons** ...  
(in the number of inputs)*

Purely  
theoretical  
results!

## Continuous functions:

- **Any bounded continuous function** can be approximated with arbitrarily small error by a network with **two hidden layers**.
- **Stronger: Any bounded continuous function** can be approximated with arbitrarily small error by a network with **one hidden layer**.