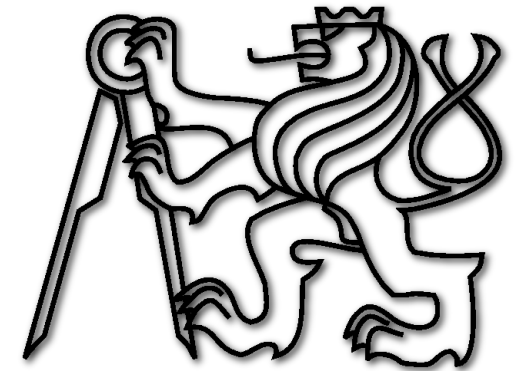# Artificial Neural Networks
MLP, Backpropagation

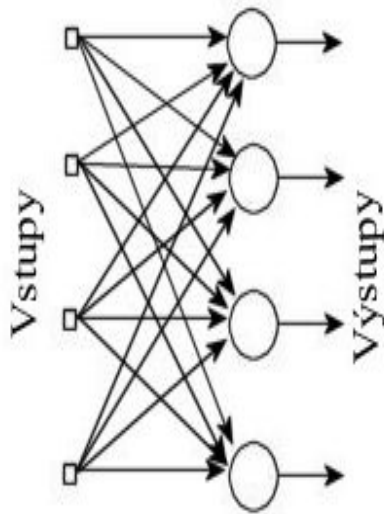*Jan Drchal*

`drchajan@fel.cvut.cz`

Computational Intelligence Group
Department of Computer Science and Engineering
Faculty of Electrical Engineering
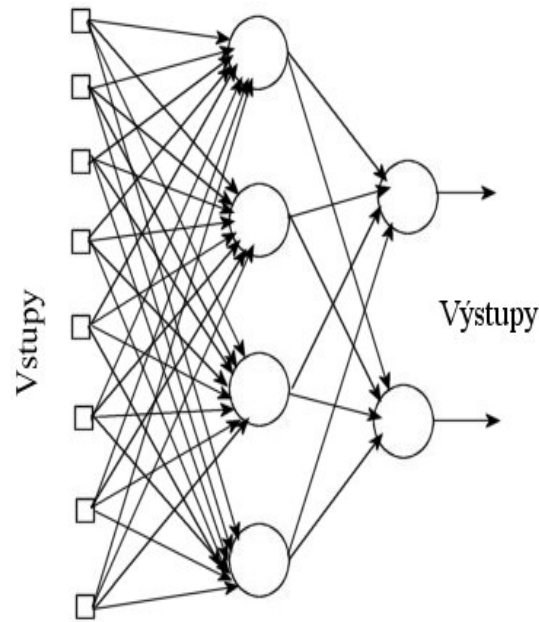*Czech Technical University in Prague*

Outline

- MultiLayer Perceptron (MLP).

- How many layers and neurons?

- How to train ANNs?

- Backpropagation.
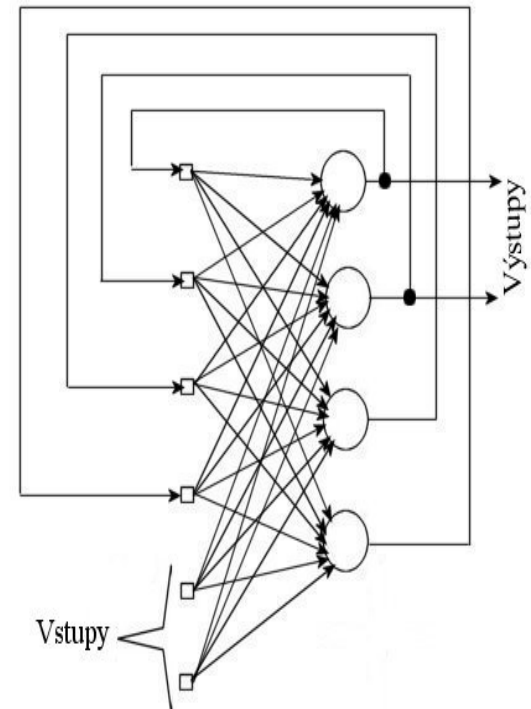
- Derived and other methods.

Layered ANNs

single layer

two layers

feed-forward networks

recurrent network

COMPUTATIONAL
INTELLIGENCE
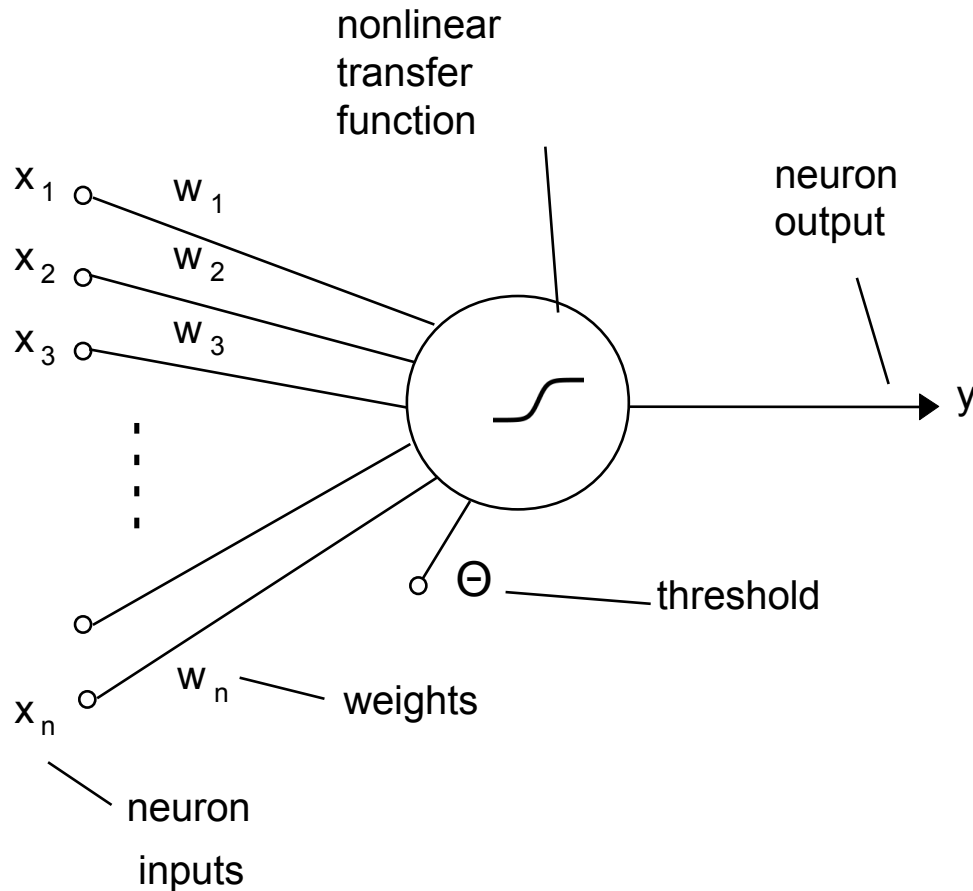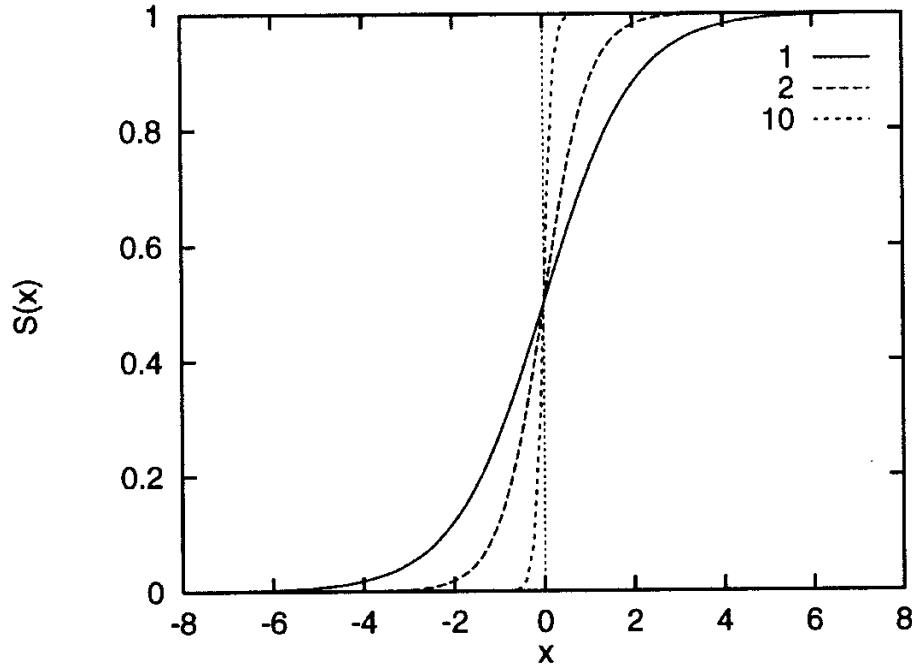GROUP

MultiLayer Perceptron (MLP)
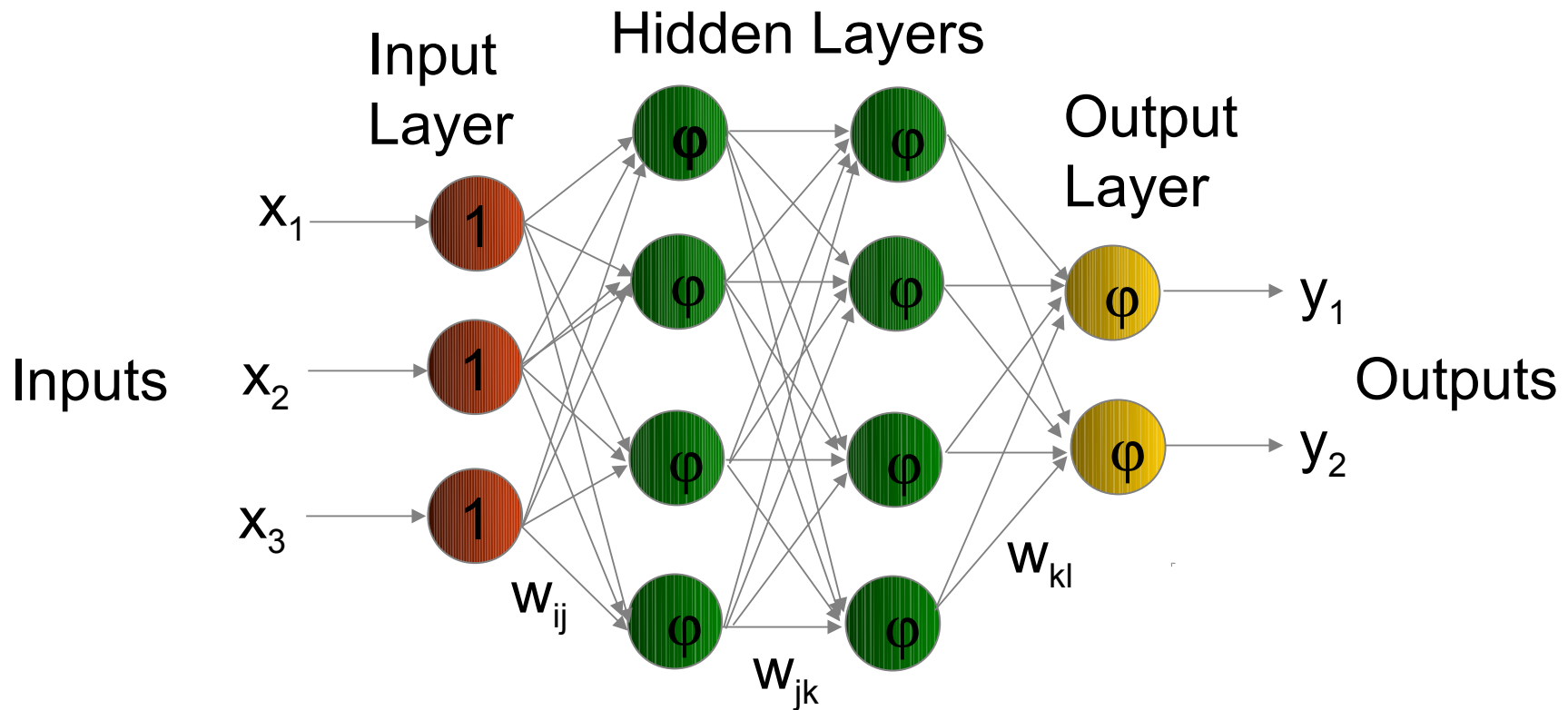
Neurons in MLPs

McCulloch-Pitts perceptron.

Logistic Sigmoid Function

Sigmoid for different
gain/slope parameter γ.
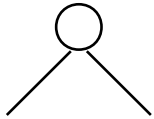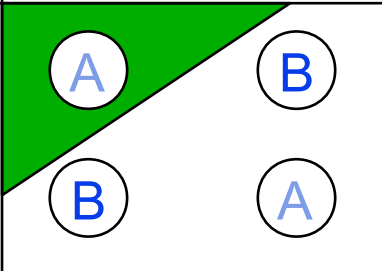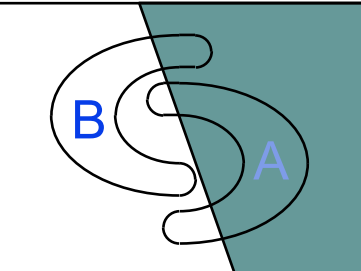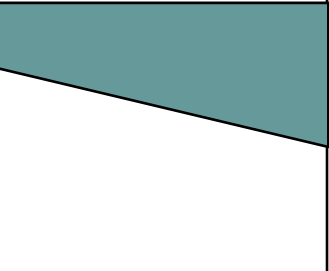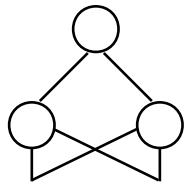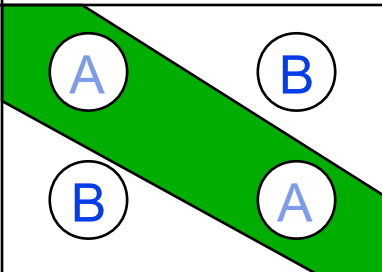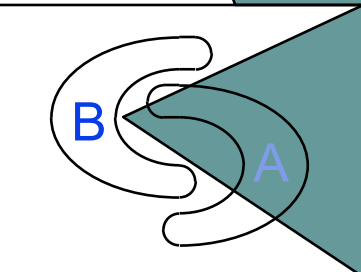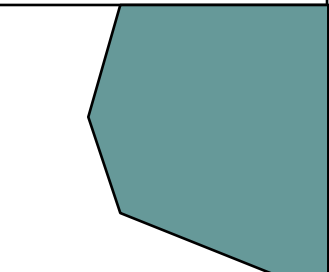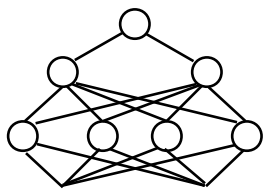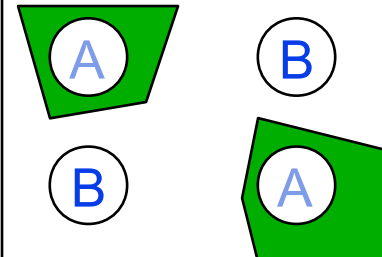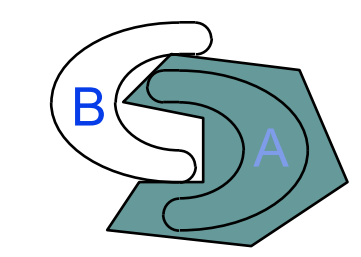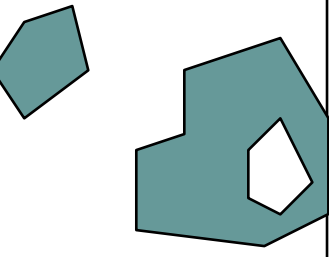
$$S(s) = \frac{1}{1 + e^{-\gamma s}}$$

- But also
  many other
  (non)-linear
  functions...

COMPUTATIONAL
INTELLIGENCE
GROUP

How Many Hidden Layers?
MLPs with Discrete Activation Functions

see ftp://ftp.sas.com/pub/neural/FAQ3.html#A_hl for overview

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)			

How Many Hidden Layers? Continuous MLPs

- **Universal Approximation** property.

- Kurt Hornik: "For MLP using **continuous, bounded, and non-constant** activation functions a **single hidden layer is enough** to approximate any function."

- **Q**: what about linear activation?

How Many Hidden Layers? Continuous MLPs

- **Universal Approximation** property.

- Kurt Hornik: "For MLP using **continuous, bounded, and non-constant** activation functions a **single hidden layer is enough** to approximate any function."

- **Q**: what about linear activation?

- **A**: it is continuous, non-constant but not bounded!

- We will get back to this topic later...

Continuous MLPs

- Although one hidden layer is enough for a continuous MLP:

  - we don't know how many neurons to use,

  - **fewer neurons are often sufficient for ANN architectures with two (or more) hidden layers.**

    See ftp://ftp.sas.com/pub/neural/FAQ3.html#A_hl for example.

How Many Neurons?

No one knows :(we have only rough estimates (upper bounds):

ANN with a **single hidden layer**:

$$N_{hid} = \sqrt{N_{in} \cdot N_{out}} \, ,$$

ANN with two **hidden layers**:

$$N_{hid-1} = N_{out} \cdot \left(\sqrt[3]{\frac{N_{in}}{N_{out}}} \right)^2 , \quad N_{hid-2} = N_{out} \cdot \left(\sqrt[3]{\frac{N_{in}}{N_{out}}} \right) .$$

You have to experiment.

COMPUTATIONAL
INTELLIGENCE
GROUP

Generalization vs. Overfitting

- When training ANNs we typically want them to perform accurately on new previously unseen data.

- This ability is known as the **generalization**.

- When ANN rather memorizes the training data while giving bad results on new data, we talk about **overfitting (overtraining)**.

COMPUTATIONAL
INTELLIGENCE
GROUP

Training/Testing Sets

Training set	Testing set

Random samples.

~70%   ~30%
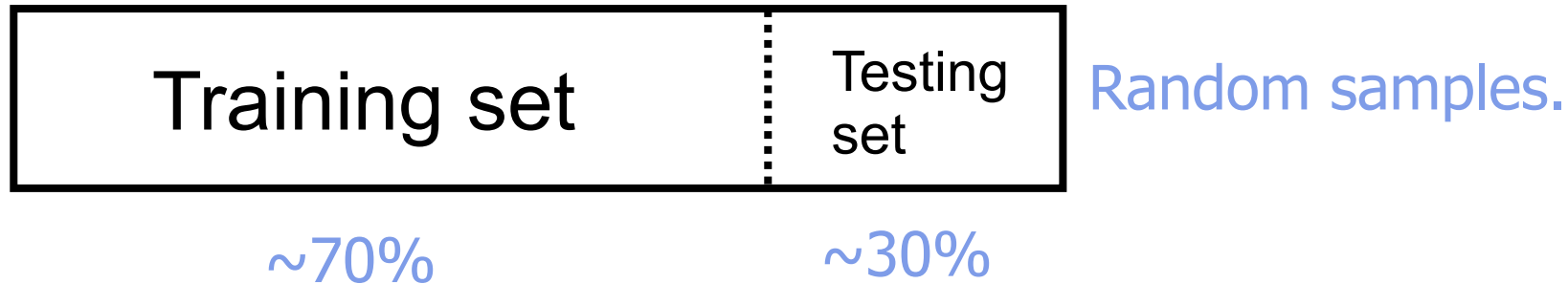
Used to train ANN model.

Testing set error
→ generalization

Overfitting Example

here, we should stop training

testing error

training error
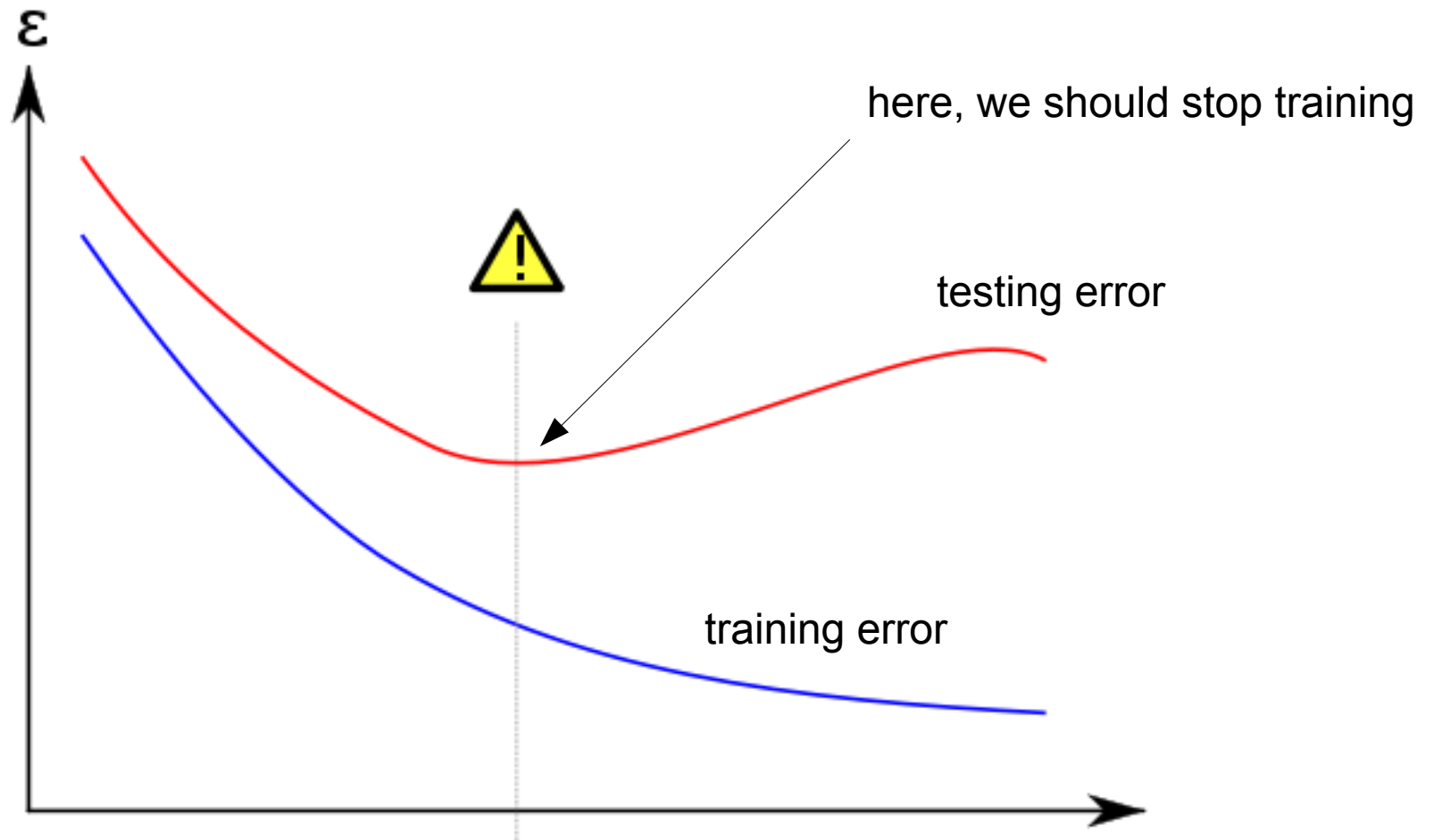
http://upload.wikimedia.org/wikipedia/commons/1/1f/Overfitting_svg.svg

Example: Bad Choice of A Training Set
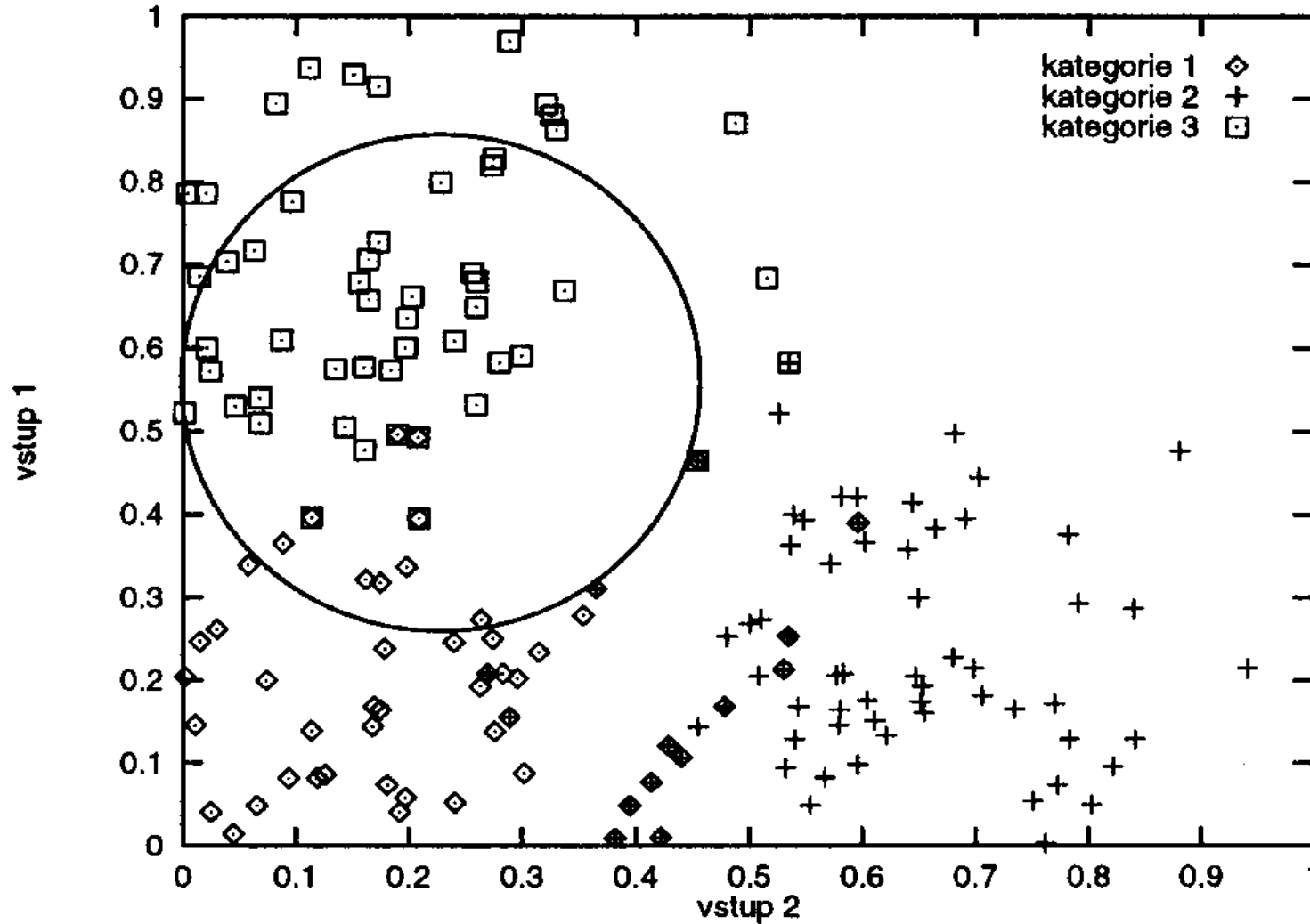
Training/Validation/Testing Sets

- Ripley "Pattern Recognition and Neural Networks", 1996:

  - **Training set**: A set of examples used for learning, that is to fit the parameters [i.e., weights] of the ANN.

  - **Validation set**: A set of examples used to tune the parameters [i.e., architecture, not weights] of an ANN, for example to choose the number of hidden units.

  - **Test set**: A set of examples used only to assess the performance (generalization) of a fully-specified ANN.

- Separated: ~60%, ~20%, ~20%.

- Note: meaning of the validation and test sets is often reversed in literature (machine-learning vs. statistics).

For example see Priddy, Keller: Artificial neural networks: an introduction (Google books, p. 44)

Training/Validation/Testing Sets II
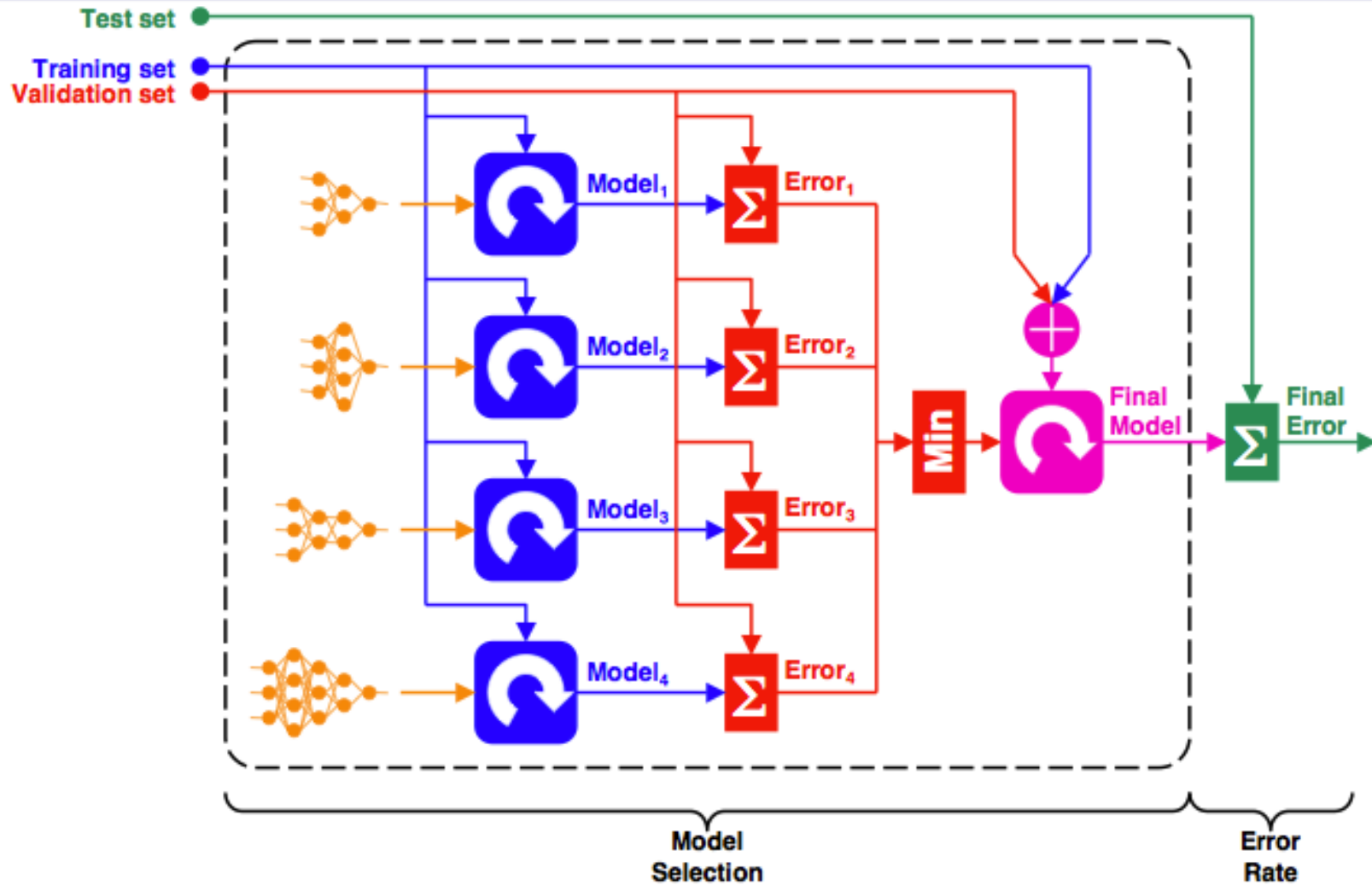
•Taken from Ricardo Gutierrez-Osuna's slides: http://courses.cs.tamu.edu/rgutier/ceg499_s02/l13.pdf

k-fold Cross-validation

Example 10-fold cross-validation:

Split training data to 10 folds of equal size.
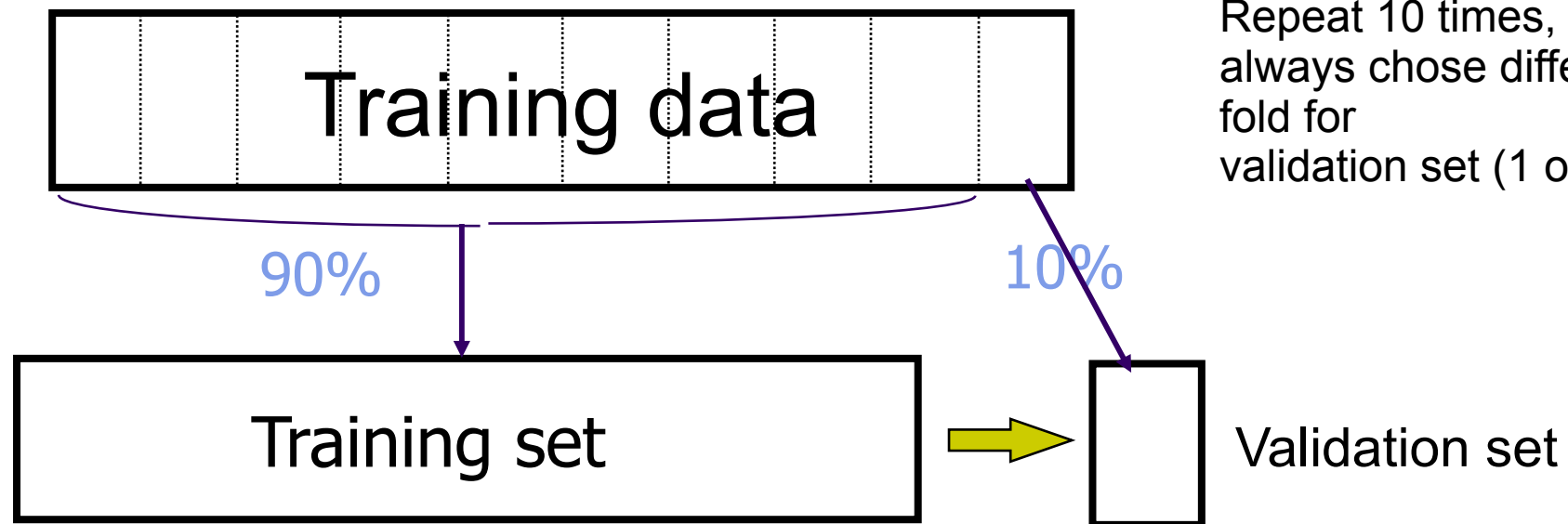
Repeat 10 times, always chose different fold for validation set (1 of 10).

90%   10%

Training set   →   Validation set

Create 10 ANN models.
Choose the best for testing data.

The cross-validation error is the average over all (10) validation sets.

Suitable for small datasets, reduces the problems caused by random selection of training/testing sets

"Cleaning" Dataset

- Imputing missing values.
- Outlier identification:
  - the instance of the data distant from the rest of the data
- Smoothing-out the noisy data.

Data Reduction

- Often needed for large data sets.

- The reduced dataset should be representative sample of the original dataset.

- The simplest algorithm: randomly remove data instances.

COMPUTATIONAL
INTELLIGENCE
GROUP

Data Transformation

- Normalization:
  - scaling/shifting values to fit given interval..

- Aggregation:
  - i.e. "binning" - discretization (continuous values to classes).

Learning Process Notes

- We don't have to choose instances sequentially $\rightarrow$ random selection.

- We can apply certain instances more frequently than others.

- We need often hundreds to thousands epochs to train the network.

- Good strategy might speed things up.

COMPUTATIONAL
INTELLIGENCE
GROUP

Backpropagation (BP)

- Paul Werbos,

- 1974, Harvard, PhD thesis.

- Still popular method,

- many modifications.

- **BP is a learning method for MLP:**

  - **continuous, differentiable activation functions!**

COMPUTATIONAL
INTELLIGENCE
GROUP

BP Overview

random weight initialization
repeat
  repeat *// epoch*
    choose an instance from the training set
    apply it to the network
    evaluate network outputs
    compare outputs to desired values
    modify the weights
  until all instances selected from the training set
until global error < criterion

ANN Energy

Backpropagation is based on a minimalization of ANN *energy* (= error). Energy is a measure describing how the network is trained on given data. For BP we define the energy function:

$$E_{TOTAL} = \sum_p E_p$$

The total sum computed over all patterns of the training set.

where

$$E_p = \frac{1}{2} \sum_{i=1}^{N_o} (d_i^o - y_i^o)^2$$

we will omit "p" in following slides

Note, ½ – only for convenience – we will see later...

COMPUTATIONAL
INTELLIGENCE
GROUP

ANN Energy II

The energy is a function of:

$$ E = f\left(\vec{X}, \vec{W} \right) $$

$\vec{W}$ weights (thresholds) → **variable**,

$\vec{X}$ inputs → **fixed (for given pattern)**.

Backpropagation Keynote

- For given values at network inputs we obtain an energy value.

- Our task is to minimize this value.

- The minimization is done via modification of weights and thresholds.

- We have to identify how the energy changes when a certain weight is changed by Δw.

- This corresponds to partial derivatives $\dfrac{\partial E}{\partial w}$.

- We employ a **gradient method**.

Gradient Descent in Energy Landscape

Energy/Error Landscape
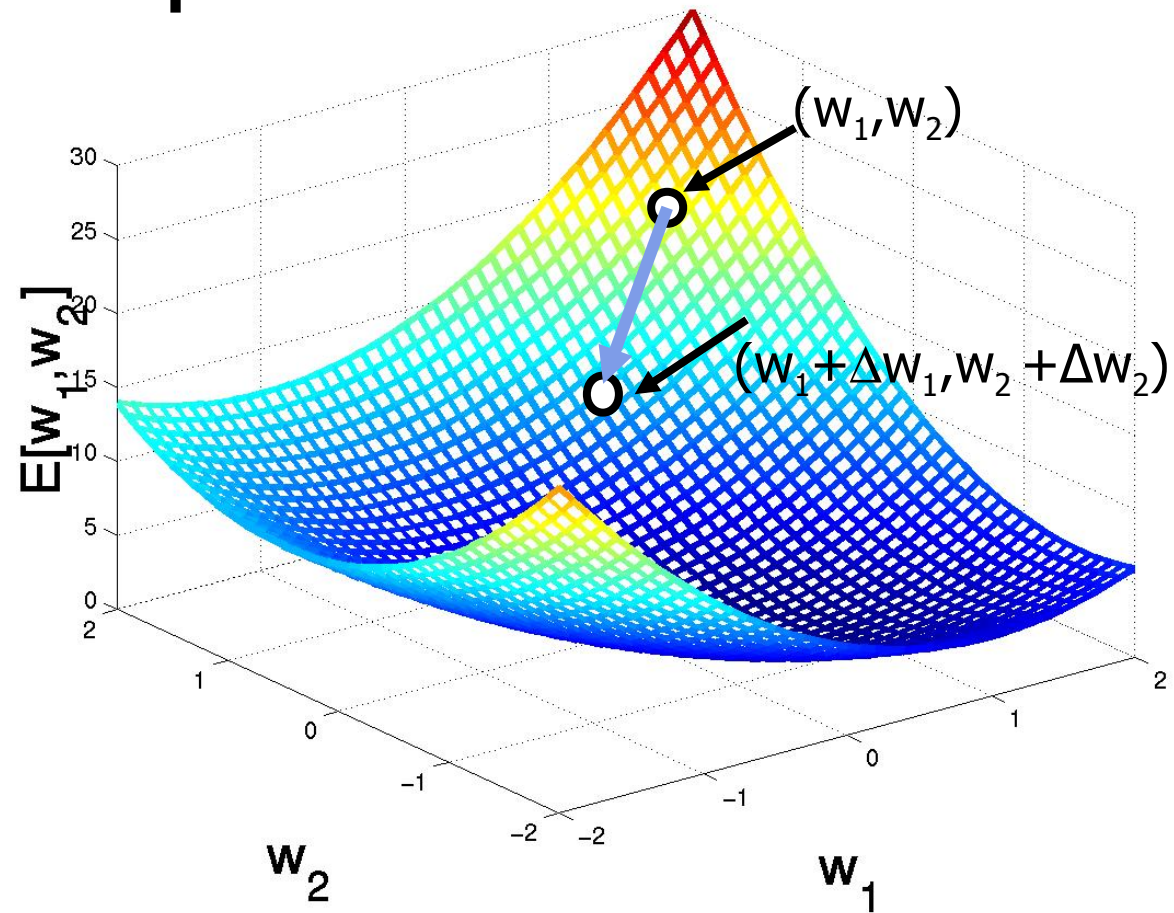
Weight Update

- We want to update weights in opposite direction to the gradient:

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

weight
"delta"

learning
rate

Note: gradient of energy function is a vector which contains partial derivatives for all weights (thresholds)

Notation

w_{jk} weight of connection from neuron j to neuron k

w_{0k}^{m} threshold of neuron k in layer m

w_{jk}^{m} weight of connection from layer $m\text{-}1$ to m

s_{k}^{m} inner potential of neuron k in layer m

y_{k}^{m} output of neuron k in layer m

x_{k} k-th input

N_{i}, N_{h}, N_{o} number of neurons in input, hidden and output layers

COMPUTATIONAL
INTELLIGENCE
GROUP
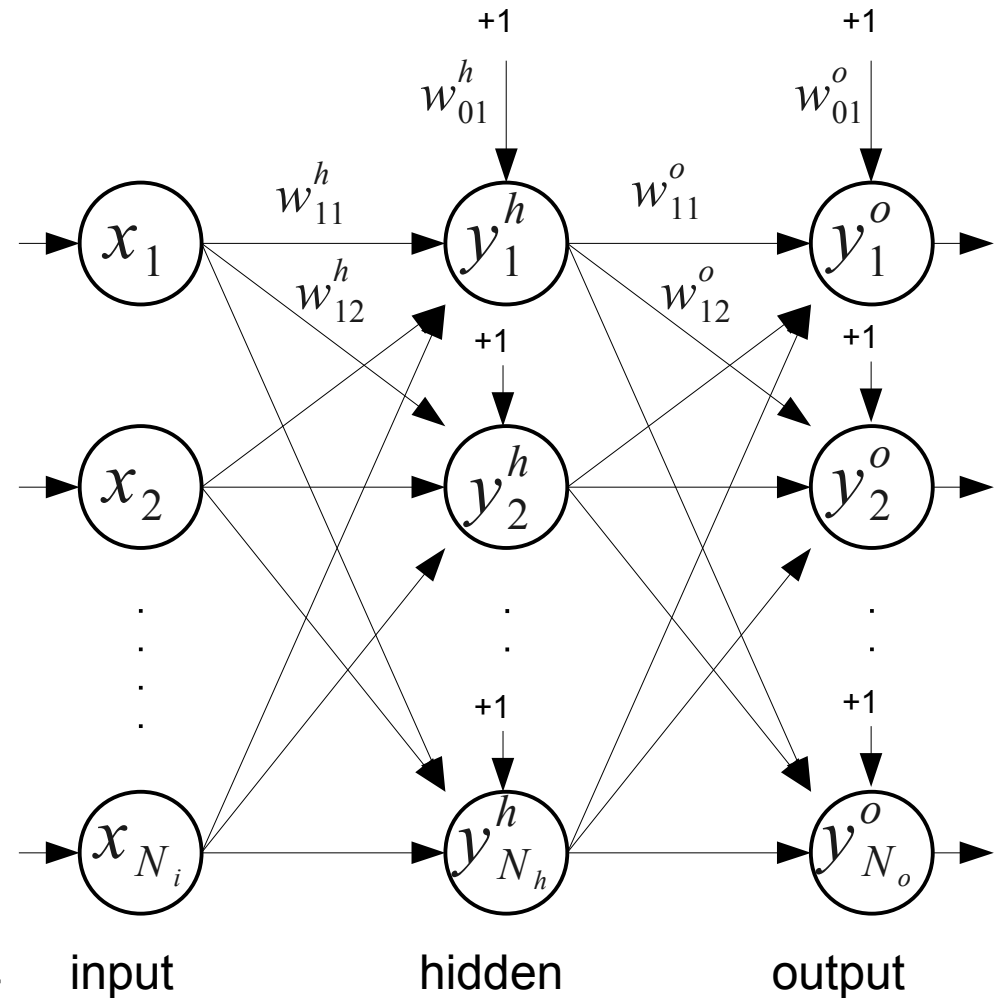
Energy as a Function Composition

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial s_k} \frac{\partial s_k}{\partial w_{jk}}$$

Energy as a Function Composition

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial s_k} \frac{\partial s_k}{\partial w_{jk}}$$

use $\longrightarrow$ $s_k = \sum_j w_{jk} y_j$

$$\frac{\partial s_k}{\partial w_{jk}} = y_j$$

Energy as a Function Composition

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial s_k} \frac{\partial s_k}{\partial w_{jk}}$$

$$s_k = \sum_j w_{jk} y_j$$

use

denote

$$\frac{\partial s_k}{\partial w_{jk}} = y_j$$

$$\delta_k = -\frac{\partial E}{\partial s_k}$$

Energy as a Function Composition

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial s_k} \frac{\partial s_k}{\partial w_{jk}}$$

$$s_k = \sum_j w_{jk} y_j$$

use

denote

$$\frac{\partial s_k}{\partial w_{jk}} = y_j$$

$$\delta_k = -\frac{\partial E}{\partial s_k}$$

$$\Delta w_{jk} = \eta \, \delta_k \, y_j$$

Remember the delta rule?

Output Layer

$$\delta_k = -\frac{\partial E}{\partial s_k} \qquad \xrightarrow{\text{output layer}} \qquad \delta_k^o = -\frac{\partial E}{\partial s_k^o}$$

Output Layer

$$\delta_k = -\frac{\partial E}{\partial s_k} \qquad \xrightarrow{\text{output layer}} \qquad \delta_k^o = -\frac{\partial E}{\partial s_k^o}$$

$$\frac{\partial E}{\partial s_k^o} = \frac{\partial E}{\partial y_k^o} \frac{\partial y_k^o}{\partial s_k^o}$$

Output Layer

$$\delta_k = -\frac{\partial E}{\partial s_k}$$

output layer →

$$\delta_k^o = -\frac{\partial E}{\partial s_k^o}$$

$$\frac{\partial E}{\partial s_k^o} = \frac{\partial E}{\partial y_k^o} \frac{\partial y_k^o}{\partial s_k^o}$$

derivate of activation function

$$\frac{\partial y_k^o}{\partial s_k^o} = S'(s_k^o)$$

Output Layer

$$\delta_k = -\frac{\partial E}{\partial s_k} \xrightarrow{\text{output layer}} \delta_k^o = -\frac{\partial E}{\partial s_k^o}$$

$$E = \frac{1}{2}\sum_{i=1}^{N_o}(d_i^o - y_i^o)^2$$

$$\frac{\partial E}{\partial s_k^o} = \frac{\partial E}{\partial y_k^o}\frac{\partial y_k^o}{\partial s_k^o}$$

use

derivate of activation function

$$\frac{\partial E}{\partial y_k^o} = -(d_k^o - y_k^o)$$

$$\frac{\partial y_k^o}{\partial s_k^o} = S'(s_k^o)$$

Output Layer

$$\delta_k = -\frac{\partial E}{\partial s_k} \qquad \xrightarrow{\text{output layer}} \qquad \delta_k^o = -\frac{\partial E}{\partial s_k^o}$$

$$E = \frac{1}{2}\sum_{i=1}^{N_o} (d_i^o - y_i^o)^2 \qquad \frac{\partial E}{\partial s_k^o} = \frac{\partial E}{\partial y_k^o}\frac{\partial y_k^o}{\partial s_k^o}$$

use

dependency
of energy
on a network
output

$$\frac{\partial E}{\partial y_k^o} = -(d_k^o - y_k^o)$$

derivate of
activation
function

$$\frac{\partial y_k^o}{\partial s_k^o} = S'(s_k^o)$$

Output Layer

$$\delta_k = -\frac{\partial E}{\partial s_k} \xrightarrow{\text{output layer}} \delta_k^o = -\frac{\partial E}{\partial s_k^o}$$

$$E = \frac{1}{2}\sum_{i=1}^{N_o}(d_i^o - y_i^o)^2 \qquad \frac{\partial E}{\partial s_k^o} = \frac{\partial E}{\partial y_k^o}\frac{\partial y_k^o}{\partial s_k^o}$$

use

dependency
of energy
on a network
output

$$\frac{\partial E}{\partial y_k^o} = -(d_k^o - y_k^o)$$

That is why we used 1/2.

derivate of
activation
function

$$\frac{\partial y_k^o}{\partial s_k^o} = S'(s_k^o)$$

Output Layer

$$\delta_k = -\frac{\partial E}{\partial s_k} \qquad \xrightarrow{\text{output layer}} \qquad \delta_k^o = -\frac{\partial E}{\partial s_k^o}$$

$$E = \frac{1}{2} \sum_{i=1}^{N_o} (d_i^o - y_i^o)^2 \qquad \frac{\partial E}{\partial s_k^o} = \frac{\partial E}{\partial y_k^o} \frac{\partial y_k^o}{\partial s_k^o}$$

use

dependency of energy on a network output

$$\frac{\partial E}{\partial y_k^o} = -(d_k^o - y_k^o)$$

Again, remember the delta rule?

That is why we used 1/2.

derivate of activation function

$$\frac{\partial y_k^o}{\partial s_k^o} = S'(s_k^o)$$

Output Layer

$$\delta_k = -\frac{\partial E}{\partial s_k} \qquad \xrightarrow{\text{output layer}} \qquad \delta_k^o = -\frac{\partial E}{\partial s_k^o}$$

$$E = \frac{1}{2} \sum_{i=1}^{N_o} (d_i^o - y_i^o)^2 \qquad \frac{\partial E}{\partial s_k^o} = \frac{\partial E}{\partial y_k^o} \frac{\partial y_k^o}{\partial s_k^o}$$

use

dependency
of energy
on a network
output

$$\frac{\partial E}{\partial y_k^o} = -(d_k^o - y_k^o)$$

Again, remember
the delta rule?

That is why we used 1/2.

derivate of
activation
function

$$\frac{\partial y_k^o}{\partial s_k^o} = S'(s_k^o)$$

$$\Delta w_{jk}^o = \eta \, \delta_k^o \, y_j^h = \eta (d_k - y_k^o) S'(s_k^o) \, y_j^h$$

Hidden Layer

$$\delta_k = -\frac{\partial E}{\partial s_k} \qquad \xrightarrow{\text{hidden layer}} \qquad \delta_k^h = -\frac{\partial E}{\partial s_k^h}$$

Hidden Layer

$$\delta_k = -\frac{\partial E}{\partial s_k} \qquad \xrightarrow{\text{hidden layer}} \qquad \delta_k^h = -\frac{\partial E}{\partial s_k^h}$$

$$\frac{\partial E}{\partial s_k^h} = \frac{\partial E}{\partial y_k^h} \frac{\partial y_k^h}{\partial s_k^h}$$

Hidden Layer

$$\delta_k = -\frac{\partial E}{\partial s_k}$$

hidden layer $\longrightarrow$

$$\delta_k^h = -\frac{\partial E}{\partial s_k^h}$$

$$\frac{\partial E}{\partial s_k^h} = \frac{\partial E}{\partial y_k^h} \frac{\partial y_k^h}{\partial s_k^h}$$

$$\frac{\partial y_k^h}{\partial s_k^h} = S'(s_k^h)$$

Same as output layer.

Hidden Layer

$$\delta_k = -\frac{\partial E}{\partial s_k} \qquad \xrightarrow{\text{hidden layer}} \qquad \delta_k^h = -\frac{\partial E}{\partial s_k^h}$$

$$\frac{\partial E}{\partial s_k^h} = \frac{\partial E}{\partial y_k^h} \frac{\partial y_k^h}{\partial s_k^h}$$

Note, this is output
of a **hidden** neuron.

$$\frac{\partial y_k^h}{\partial s_k^h} = S'(s_k^h)$$

Same as output layer.

COMPUTATIONAL
INTELLIGENCE
GROUP

Hidden Layer

$$\delta_k = -\frac{\partial E}{\partial s_k}$$

hidden layer →

$$\delta_k^h = -\frac{\partial E}{\partial s_k^h}$$

$$\frac{\partial E}{\partial s_k^h} = \frac{\partial E}{\partial y_k^h} \frac{\partial y_k^h}{\partial s_k^h}$$

Note, this is output of a **hidden** neuron.
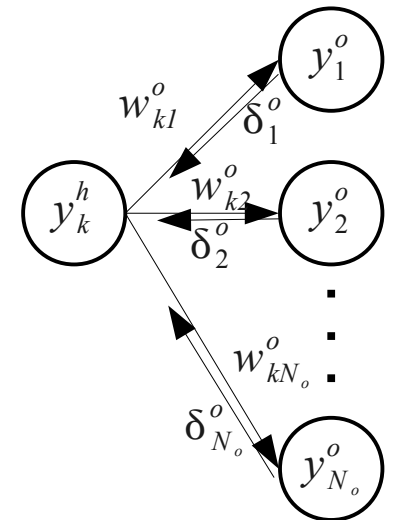
$$\frac{\partial y_k^h}{\partial s_k^h} = S'(s_k^h)$$

? let's look at this partial derivation

Same as output layer.

Hidden Layer II

$$\frac{\partial E}{\partial y_k^h} = \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} \frac{\partial s_l^o}{\partial y_k^h} = \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} \frac{\partial}{\partial y_k^h} \left(\sum_{i=1}^{N_h} w_{il}^o y_i^h \right) =$$

Apply the chain rule
(http://en.wikipedia.org/wiki/Chain_rule).

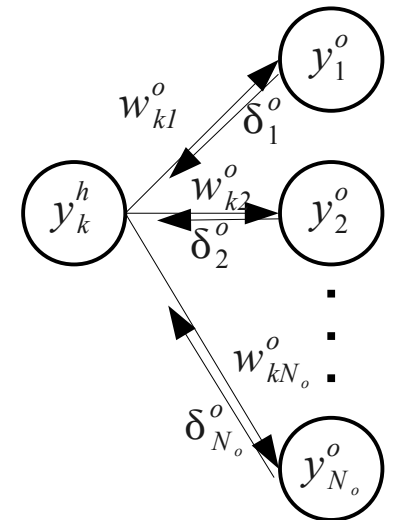COMPUTATIONAL
INTELLIGENCE
GROUP

Hidden Layer II

$$\frac{\partial E}{\partial y_k^h} = \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} \frac{\partial s_l^o}{\partial y_k^h} = \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} \frac{\partial}{\partial y_k^h} \left(\sum_{i=1}^{N_h} w_{il}^o y_i^h \right) =$$

Apply the chain rule
(http://en.wikipedia.org/wiki/Chain_rule).

$$= \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} w_{kl}^o = -\sum_{l=1}^{N_o} \delta_l^o w_{kl}^o$$
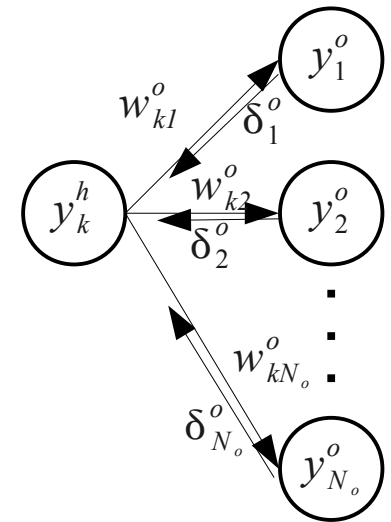
Hidden Layer II

$$\frac{\partial E}{\partial y_k^h} = \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} \frac{\partial s_l^o}{\partial y_k^h} = \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} \frac{\partial}{\partial y_k^h} \left(\sum_{i=1}^{N_h} w_{il}^o y_i^h \right) =$$

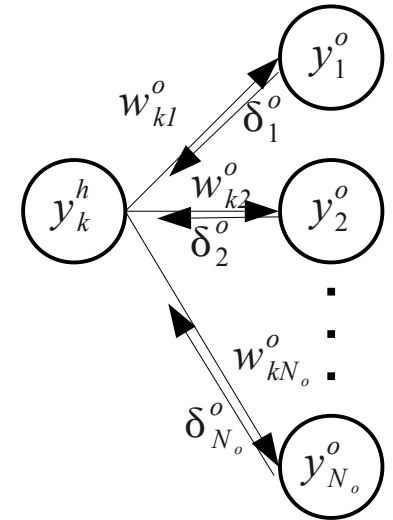Apply the chain rule
(http://en.wikipedia.org/wiki/Chain_rule).

$$= \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} w_{kl}^o = -\sum_{l=1}^{N_o} \delta_l^o w_{kl}^o$$

But we know
this already.

$$\delta_k^o = -\frac{\partial E}{\partial s_k^o}$$

Hidden Layer II

$$\frac{\partial E}{\partial y_k^h} = \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} \frac{\partial s_l^o}{\partial y_k^h} = \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} \frac{\partial}{\partial y_k^h} \left(\sum_{i=1}^{N_h} w_{il}^o y_i^h \right) =$$

Apply the chain rule
(http://en.wikipedia.org/wiki/Chain_rule).

$$= \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} w_{kl}^o = -\sum_{l=1}^{N_o} \delta_l^o w_{kl}^o$$

But we know
this already.

Take the error of
the output neuron

and multiply it by
the input weight.

$$\delta_k^o = -\frac{\partial E}{\partial s_k^o}$$

Hidden Layer II

$$\frac{\partial E}{\partial y_k^h} = \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} \frac{\partial s_l^o}{\partial y_k^h} = \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} \frac{\partial}{\partial y_k^h} \left(\sum_{i=1}^{N_h} w_{il}^o y_i^h \right) =$$

Apply the chain rule
(http://en.wikipedia.org/wiki/Chain_rule).

$$= \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} w_{kl}^o = - \sum_{l=1}^{N_o} \delta_l^o w_{kl}^o$$
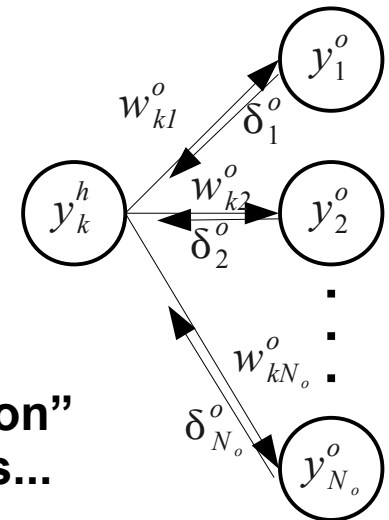
Here the "back-propagation" actually happens...

But we know this already.

Take the error of the output neuron

and multiply it by the input weight.

$$\delta_k^o = - \frac{\partial E}{\partial s_k^o}$$

Hidden Layer III

Now, let's put it all together!

$$\frac{\partial E}{\partial s_k^h} = \frac{\partial E}{\partial y_k^h} \frac{\partial y_k^h}{\partial s_k^h} = -\left(\sum_{l=1}^{N_o} \delta_k^o w_{kl}^o\right) S'(s_k^h)$$

COMPUTATIONAL
INTELLIGENCE
GROUP

Hidden Layer III

Now, let's put it all together!

$$\frac{\partial E}{\partial s_k^h} = \frac{\partial E}{\partial y_k^h} \frac{\partial y_k^h}{\partial s_k^h} = -\left(\sum_{l=1}^{N_o} \delta_k^o w_{kl}^o \right) S'(s_k^h)$$

$$\delta_k^h = -\frac{\partial E}{\partial s_k^h} = \left(\sum_{l=1}^{N_o} \delta_k^o w_{kl}^o \right) S'(s_k^h)$$

COMPUTATIONAL
INTELLIGENCE
GROUP

Hidden Layer III

Now, let's put it all together!

$$\frac{\partial E}{\partial s_k^h} = \frac{\partial E}{\partial y_k^h} \frac{\partial y_k^h}{\partial s_k^h} = -\left(\sum_{l=1}^{N_o} \delta_k^o w_{kl}^o\right) S'(s_k^h)$$

$$\delta_k^h = -\frac{\partial E}{\partial s_k^h} = \left(\sum_{l=1}^{N_o} \delta_k^o w_{kl}^o\right) S'(s_k^h)$$

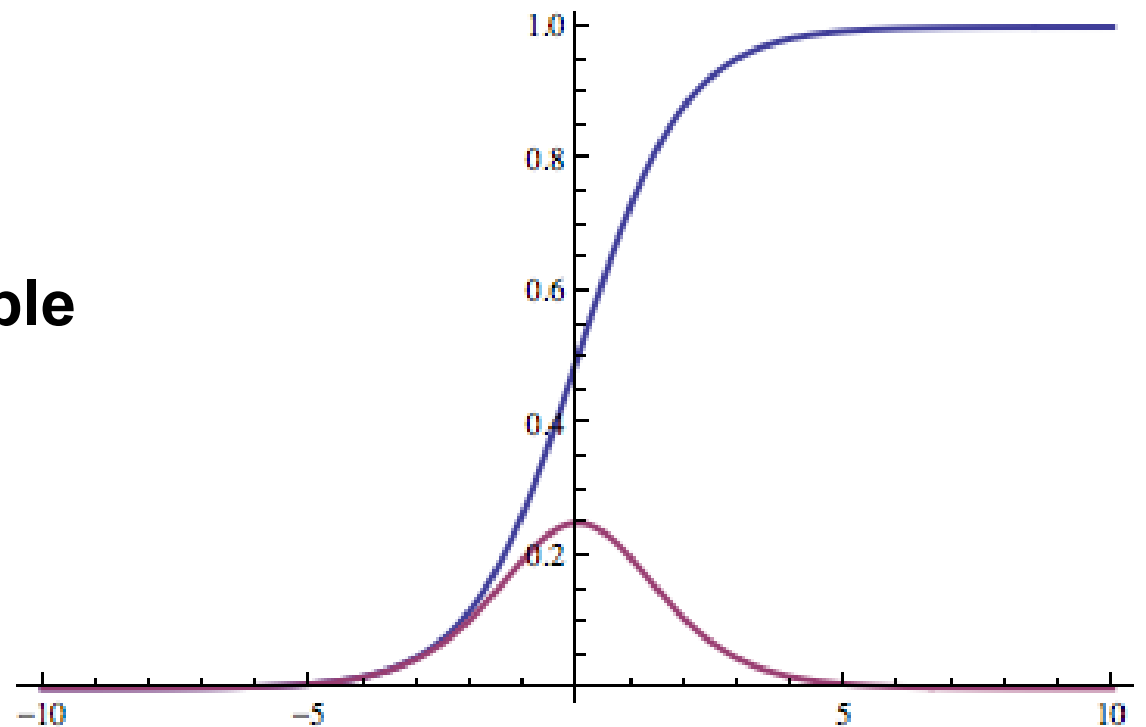The derivation of the activation function is the last thing to deal with!

$$\Delta w_{jk}^h = \eta\, \delta_k^h x_j = \eta\left(\sum_{l=1}^{N_o} \delta_k^o w_{kl}^o\right) S'(s_k^h) x_j$$
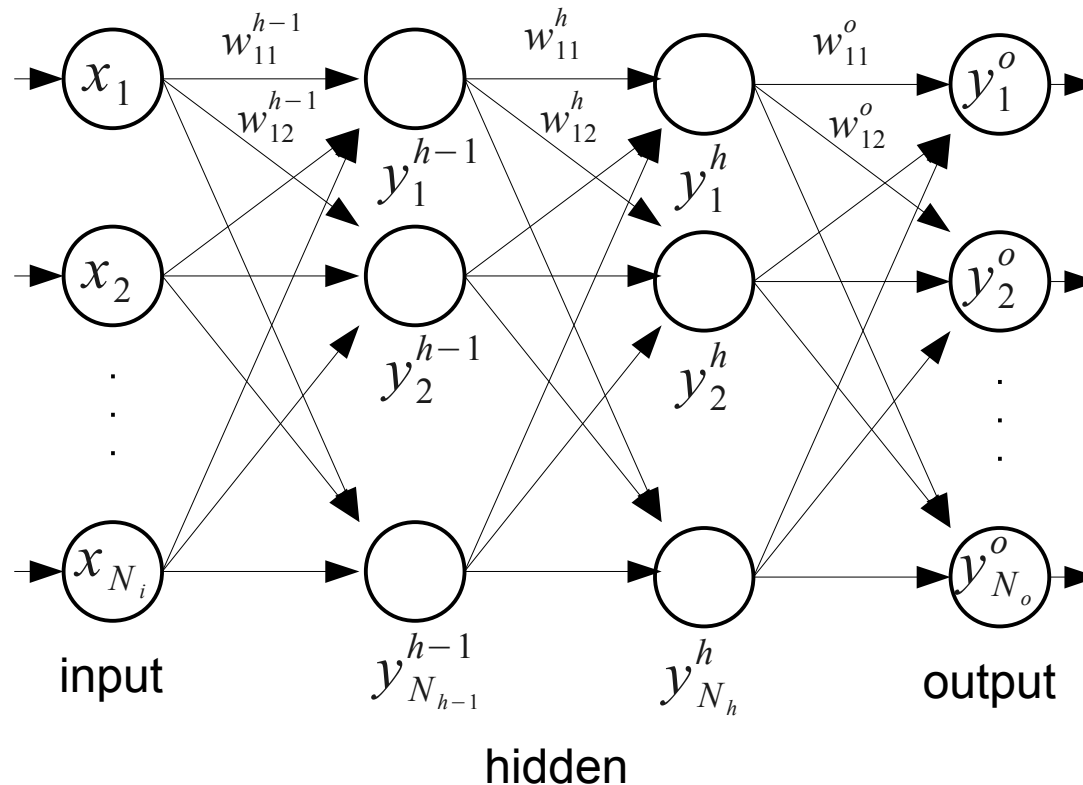
COMPUTATIONAL INTELLIGENCE GROUP

Sigmoid Derivation

$$S'(s_k) = \left(\frac{1}{1+e^{\gamma s_k}} \right)' = \frac{\gamma}{1+e^{\gamma s_k}} \frac{e^{\gamma s_k}}{1+e^{\gamma s_k}} = \gamma \, y_k (1 - y_k)$$

That is why we needed continuous & differentiable activation functions!

How About General Case?

- Arbitrary number of hidden layers?

- It's the same: for layer *h-1* use δ_k^h.

BP Put All Together

Output layer:

$$\Delta w_{jk}^{o} = \eta \, \gamma \, y_k^{o} (1 - y_k^{o})(d_k - y_k^{o}) y_j^{h}$$

This is equal to x_j when we get to inputs.

Hidden layer *m* (note that *h+1 = o*):

$$\Delta w_{jk}^{m} = \eta \, \delta_k^{m} y_j^{m-1} = \eta \, \gamma \, y_k^{m}(1 - y_k^{m}) \left(\sum_{l=1}^{N_{m+1}} \delta_k^{m+1} w_{kl}^{m+1} \right) y_j^{m-1}$$

Weight (threshold) updates:

$$w_{jk}(t+1) = w_{jk}(t) + \Delta w_{jk}(t)$$

COMPUTATIONAL INTELLIGENCE GROUP

Potential Problems

- High dimension of weight (threshold) space.

- Complexity of energy function.

- Different shape of energy function for different input vectors.

Modifications to Standard BP

- Momentum

- Simple, but greatly helps when avoiding local minima:

$$\Delta w_{\mathrm{ij}}(t) = \eta \, \delta_j(t) \, y_i(t) + \alpha \, \Delta w_{\mathrm{ij}}(t-1)$$

momentum constant: $\alpha \in [0,1)$

Modifications to Standard BP II

- Normalized cumulated delta,

    - all changes applied together,

- Delta-bar-delta rule,

    - using also previous gradient

- Extended,

    - both above together,

- Quickprop, Rprop: second order methods (approximate Hessian).

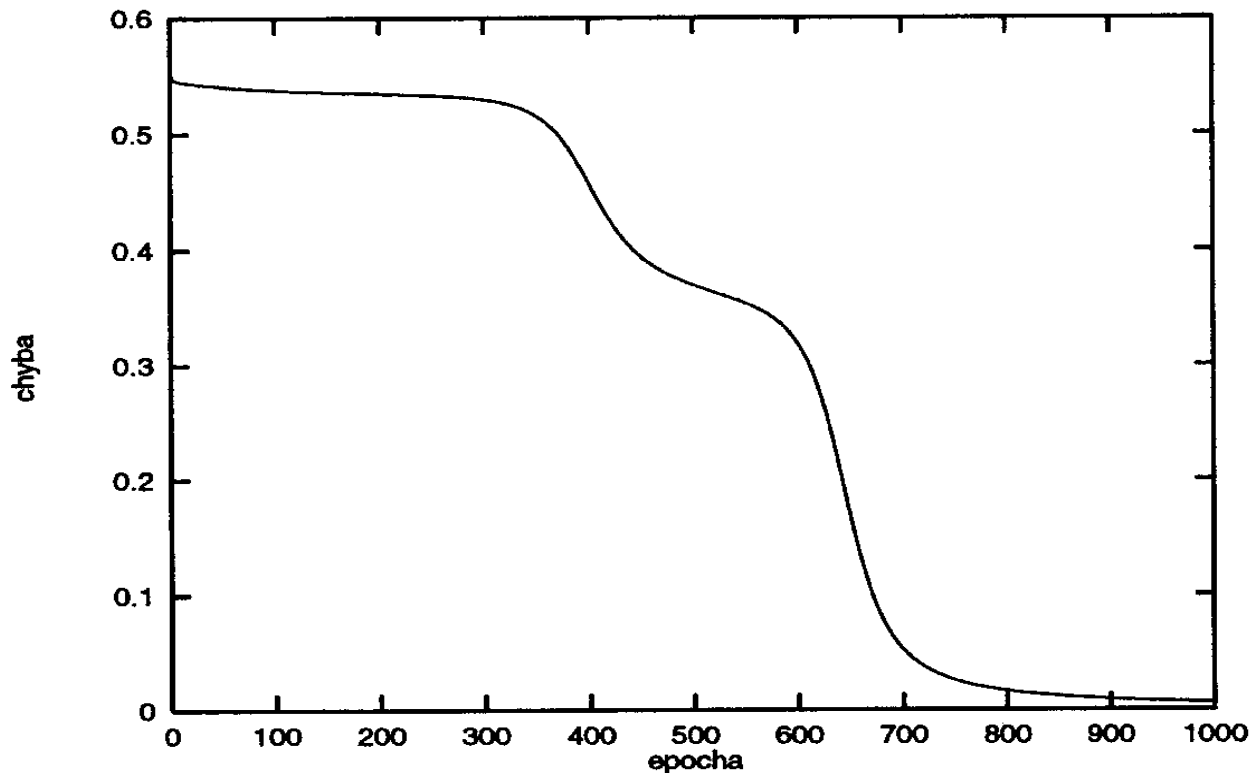# Other Approaches Based on Numerical Optimization

- Compute partial derivatives over the total energy:
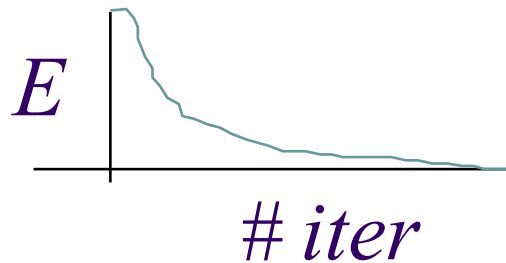
$$\frac{\partial E_{TOTAL}}{\partial w_{jk}}$$

and use **any** numerical optimization method, i.e.:

  – Conjugated gradients,

  – Quasi-Newton methods,
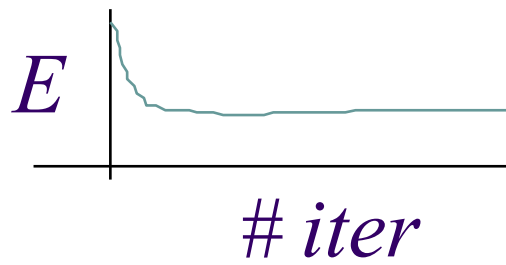
  – Levenberg-Marquardt

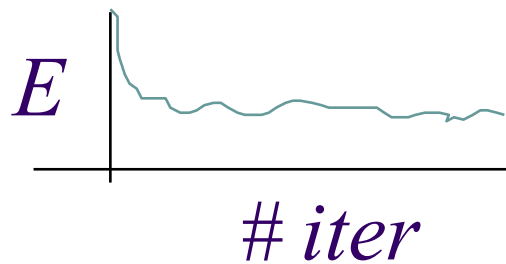# Typical Energy Behaviour During Learning

Typical Learning Runs

This is best! Constant and fast decrease of error.

Seems like local minimum.
Change learning rate, momentum,
ANN architecture, run again
(random weight initialization)...

Noisy data.

Next Lecture

- Unsupervised learning, SOM etc.

COMPUTATIONAL
INTELLIGENCE
GROUP