# Lecture 5: Neural Networks

## AE4320 System Identification of Aerospace Vehicles

Dr.ir. Coen de Visser
Department of Control & Simulation

TU Delft
Delft University of Technology

# Course Outline

- **Lecture 1: (dr.ir. Coen de Visser)**
  - Course goals and objectives
  - Introduction to System Identification
- **Lecture 2,3: (dr.ir. Daan Pool)**
  - System Identification Experiments
- **Lecture 4,5,6: (dr.ir. Daan Pool)**
  - Kalman filters
  - State estimation & Sensor Fusion
- **Lecture 7,8: (dr.ir. Coen de Visser)**
  - Model structure selection
  - Model parameter estimation

# Course Outline

- **Lecture 9: (dr.ir. Coen de Visser)**
  - Advanced identification approach: Neural networks
- **Lecture 10,11: (dr.ir. Coen de Visser)**
  - Advanced identification approach: Multivariate B-Splines
- **Lecture 12: (dr.ir. Coen de Visser)**
  - Model validation, course conclusion

# Goals of this Lecture

***Questions that will be answered during this lecture:***

1. *What is a neural network?*
2. *What are the similarities and differences of biological and artificial NNs?*
3. *What are the uses of artificial NNs?*
4. *Wat are the advantages and disadvantages of artificial NNs compared to other approaches?*
5. *How do we train an artificial NN?*

# SysID High Level Overview

Where we are now in the System Identification Cycle:

**Experiment phase**

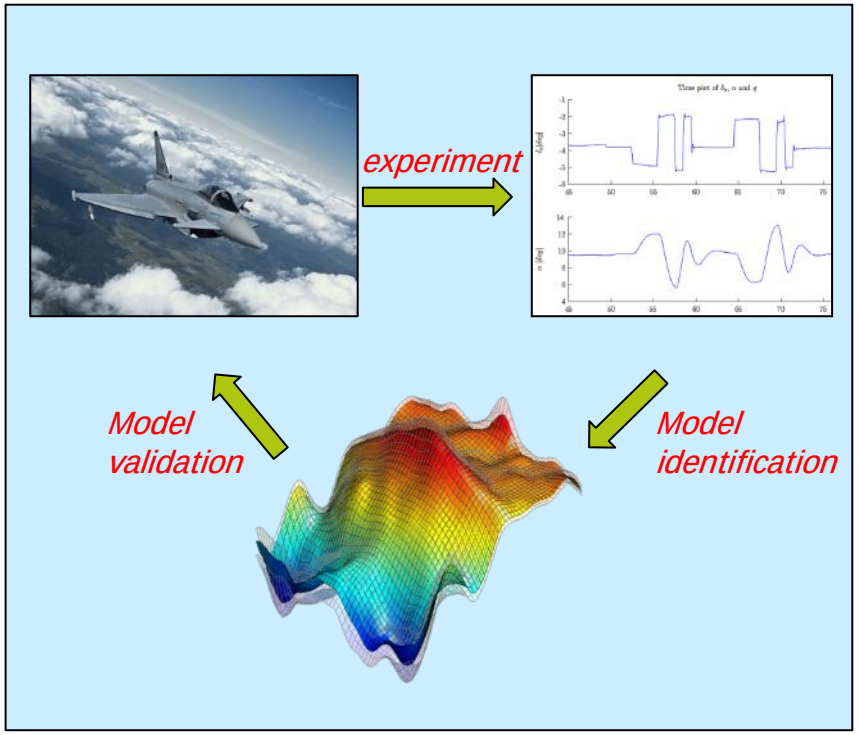
- Plant analysis
- Experiment design and execution
- Data logging and pre-processing

**Model identification phase**

- State estimation
- Model structure definition
- Parameter estimation

**Model validation phase**

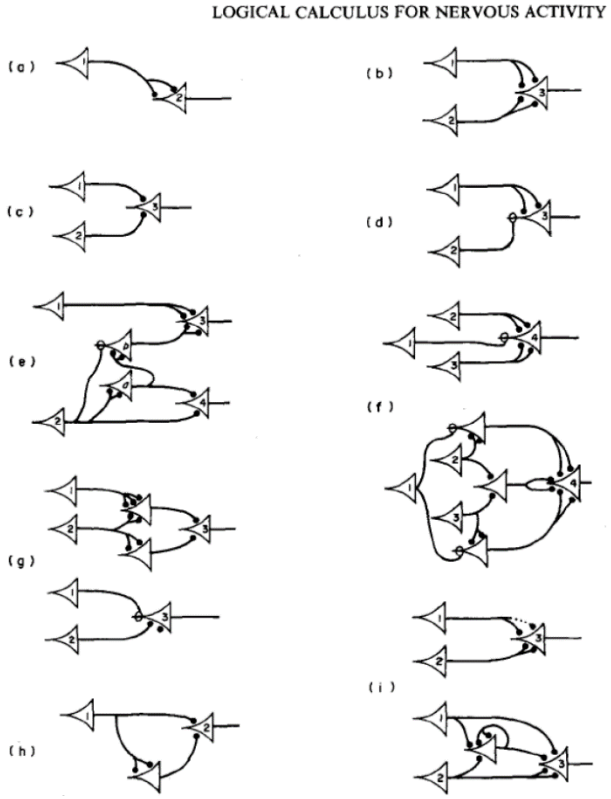
- Model validation



# Neural Networks: Introduction

## History

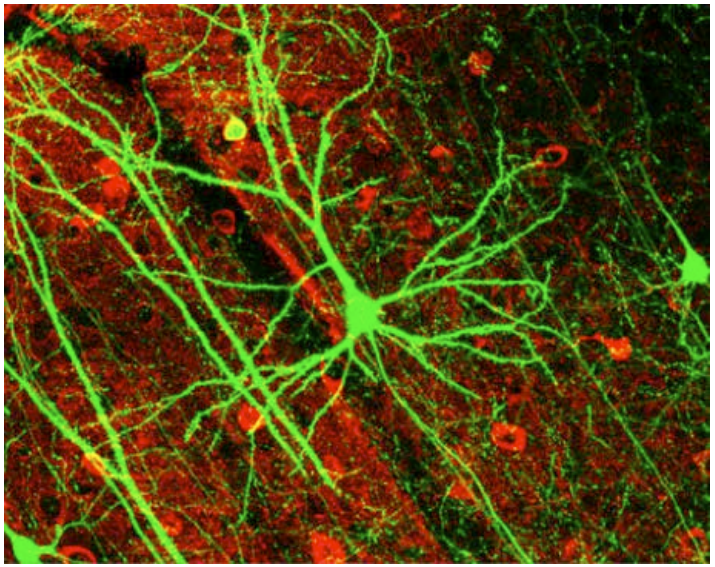
- **1800's**: First appearance of the artificial neural network (ANN)
- **1943**: Modern ANN developed in the work of McCulloch and Pitts.
- **1948**: Hebbian learning based on 'neural plasticity'
- **1960's**: Research stagnated due to lack of computational resources.
- **1975**: Backpropagation enabled training multilayer networks (Werbos, 1975).
- **2006**: Trainable Deep Neural Network (DNN) developed (Hinton, Salakhutdinov, Osindero, Teh).
- **2009**: Hardware acceleration (GPU's) enable practical use of DNNs.
- **2012**: DNN revolution, e.g. DNNs solve the Speech Recognition problem. The hype train starts!

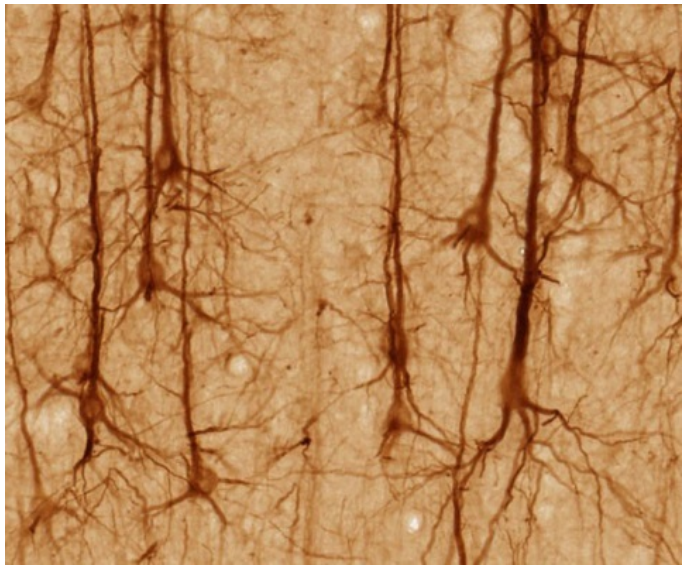


# Neural Networks: Introduction

## What is a Biological Neural Network?

- AKA: the brain.
- A network of neurons that communicate electric signals using specialized connections called synapses.
- All multicellular animals (except sponges) have neural networks.

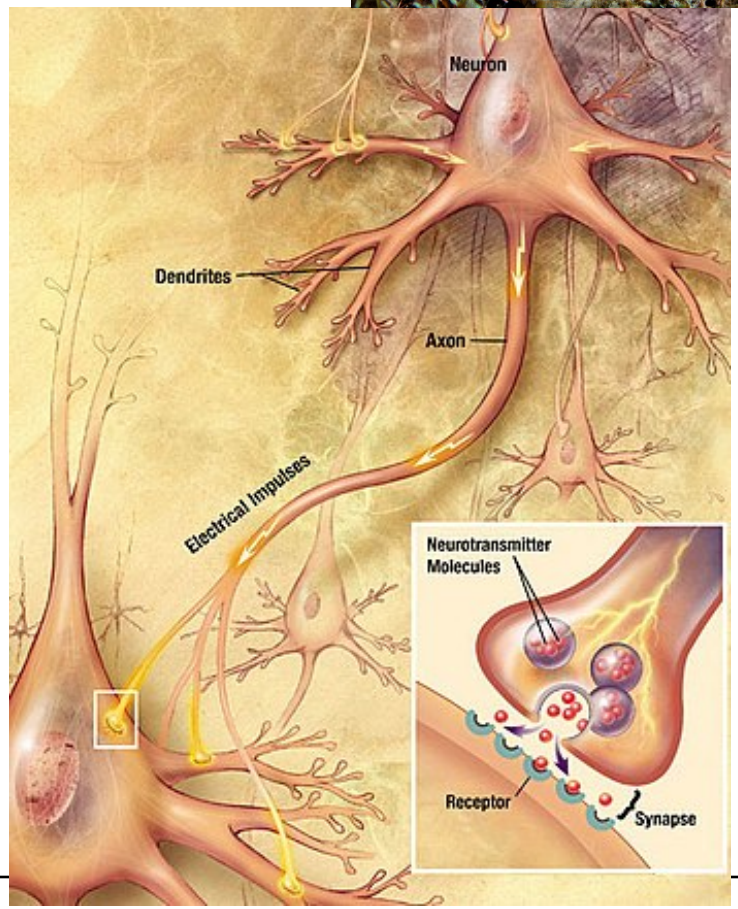




# Neural Networks: Introduction

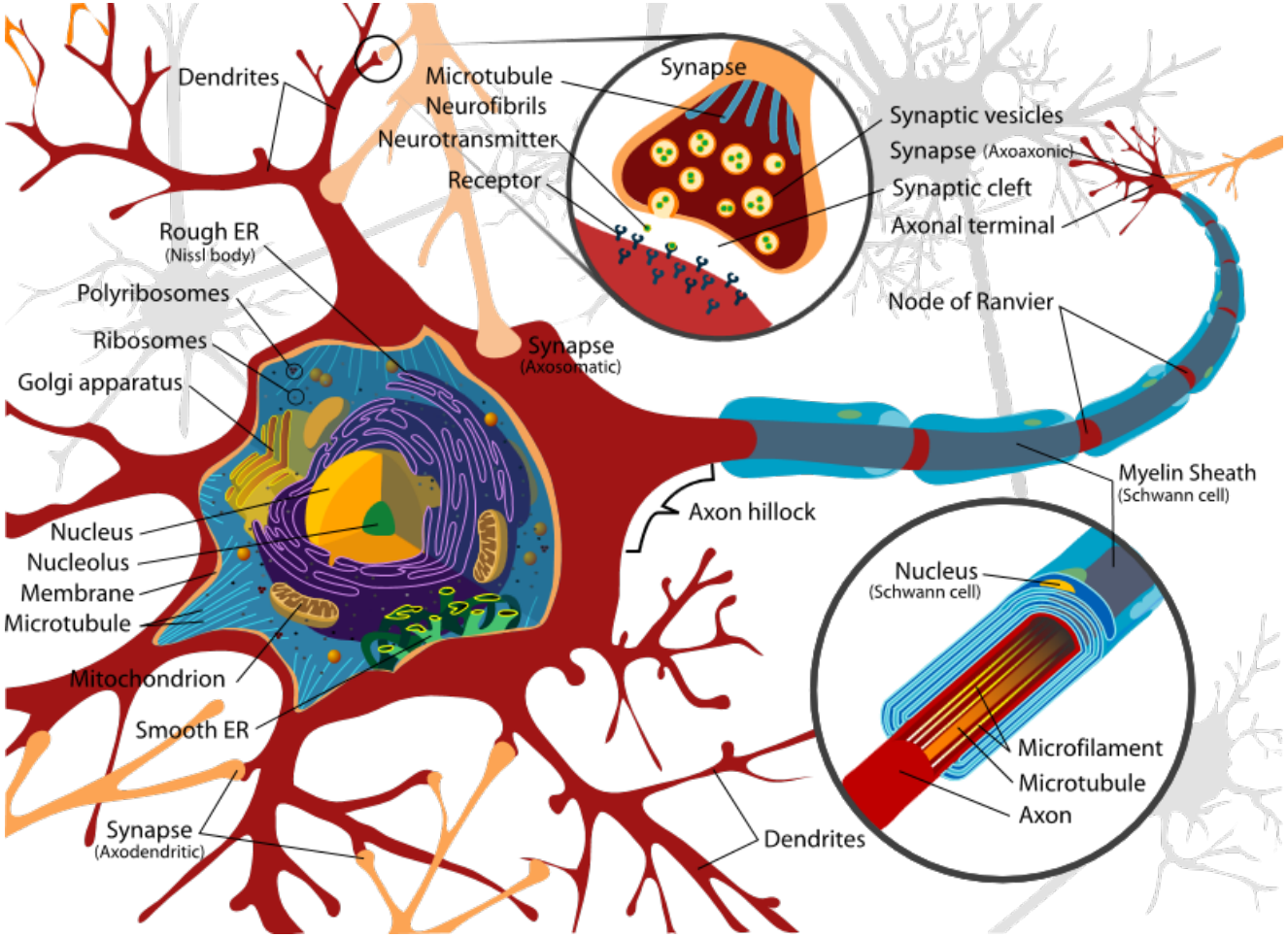
## What is a Biological Neural Network?

- 1 neuron may have 40.000 or more connections to others.
- simplest biological network contains 231 neurons (sea squirt).
- Ant brain has 250.000 neurons.
- Mouse central nervous system has 71 million neurons and 10^12 synapses.
- Dog central nervous system has 2.2 billion neurons.
- **Human** central nervous system has 86 billion neurons and 10^14 synapses.
- African Elephant central nervous system has 257 billion neurons.



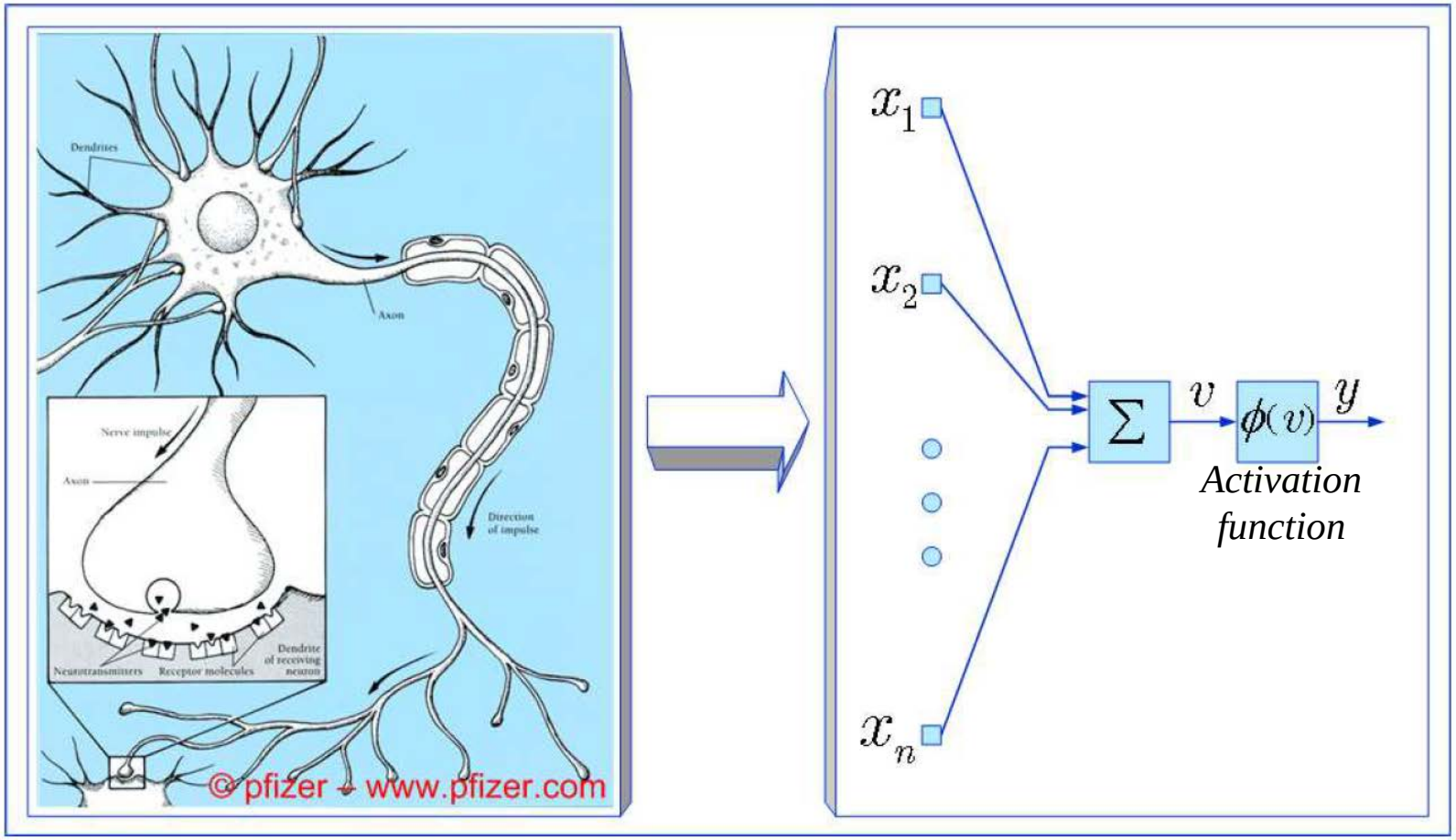
# Neural Networks: Introduction

The Neuron: an extremely complex **analog** computer



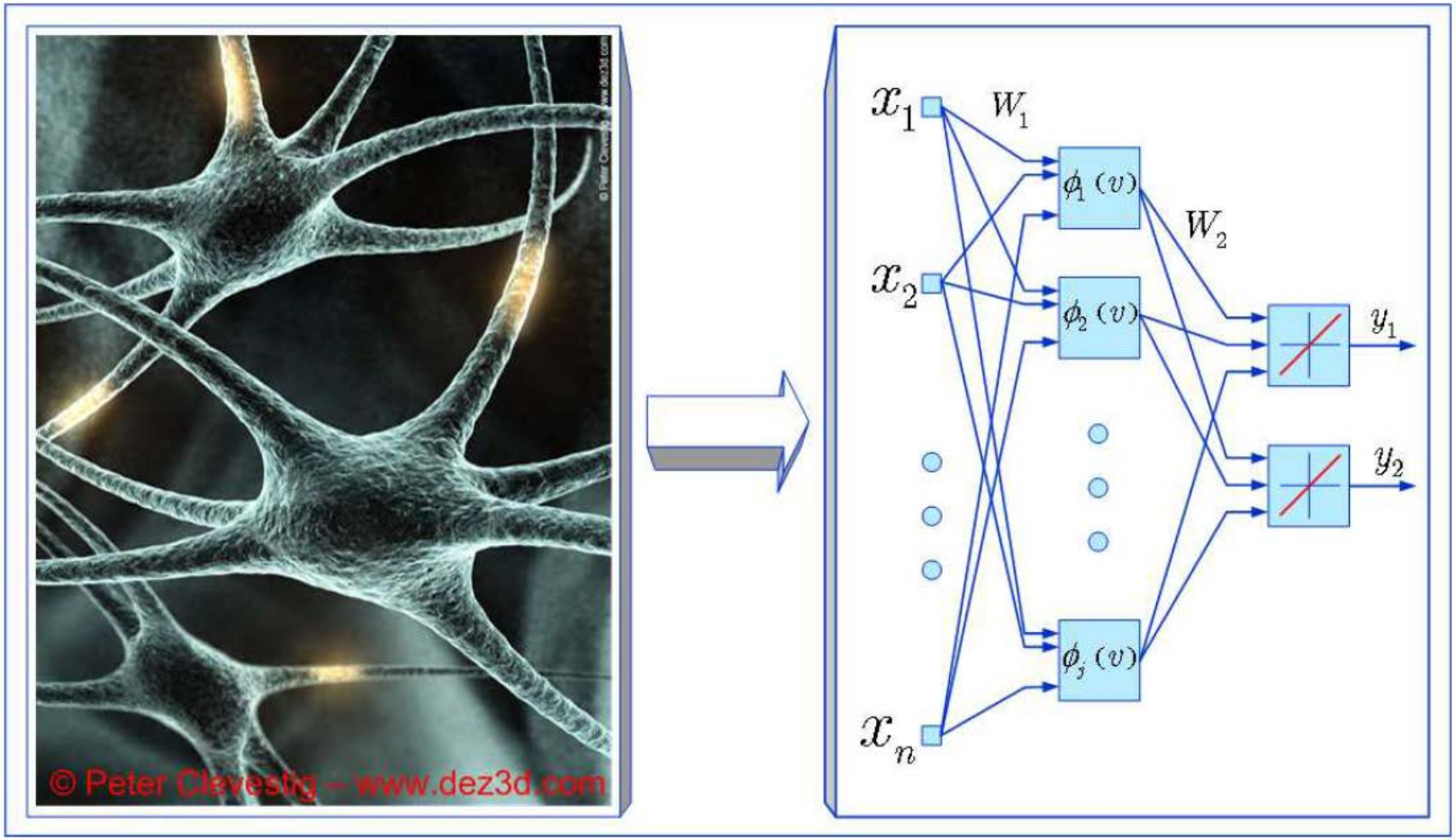
# Neural Networks: Introduction

## Artificial Neuron: simplified model of biological Neuron



# Neural Networks: Introduction

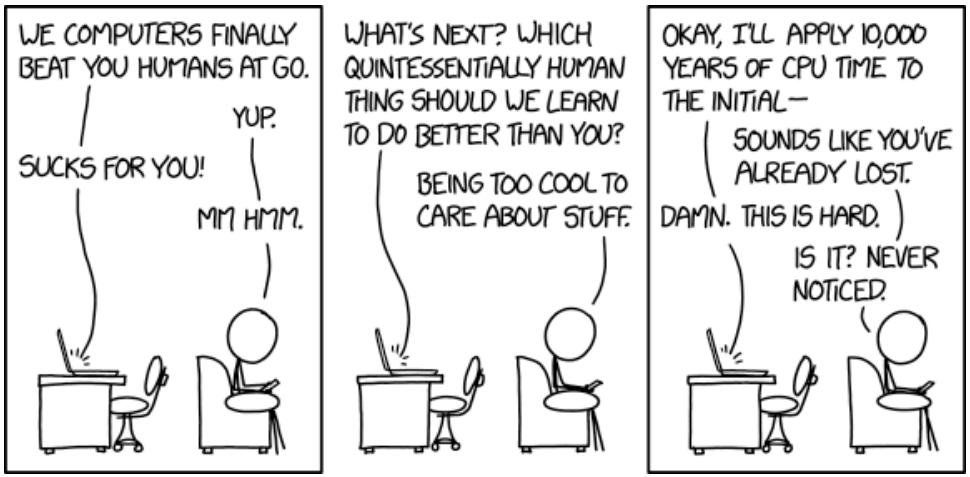
Artificial NN: hugely simplified model of biological NN



# Neural Networks: Introduction

In the last decades ANNs have been used in many different applications:

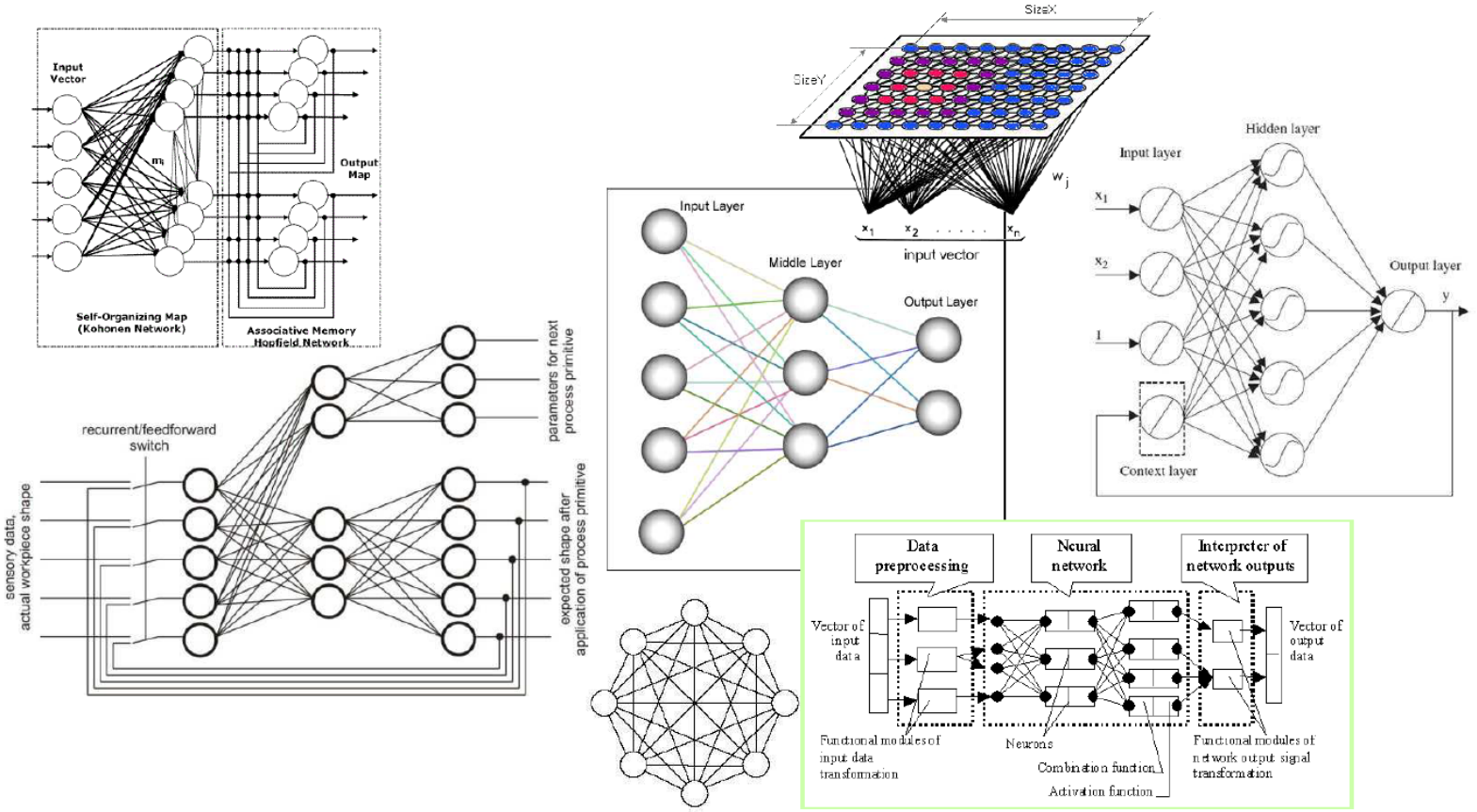
- Pattern recognition
- Speech recognition
- Adaptive control
- Stock market forecasting
- Weather forecasting
- Aerodynamic model identification
- Games: Checkers, Backgammon, Chess, Go, Doom…
- Self-driving cars…




https://www.youtube.com/watch?v=gn4nRCC9TwQ

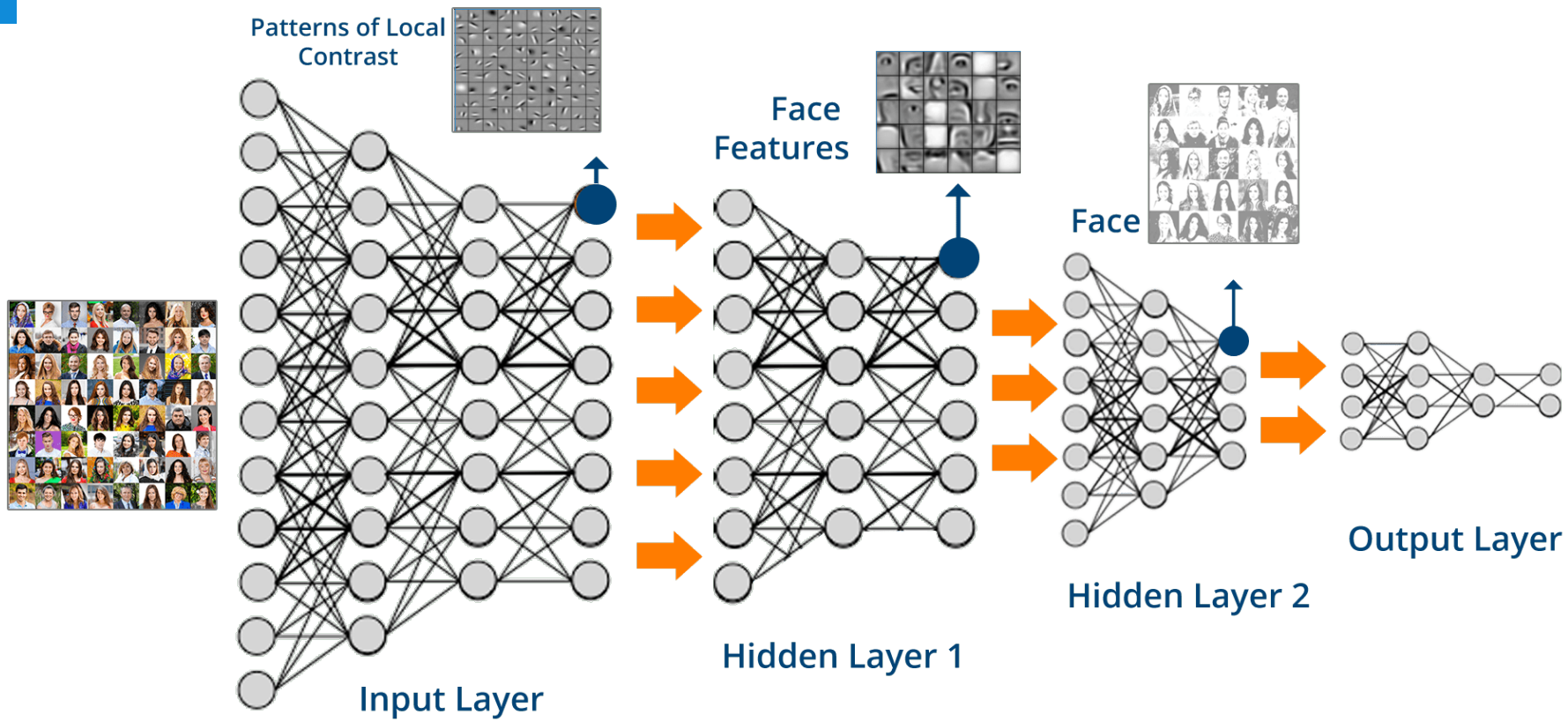
# Neural Networks: Introduction

Many different network structures exist today ranging from simple to very complex networks...



# Neural Networks: Introduction

The latest (hype): deep neural networks...

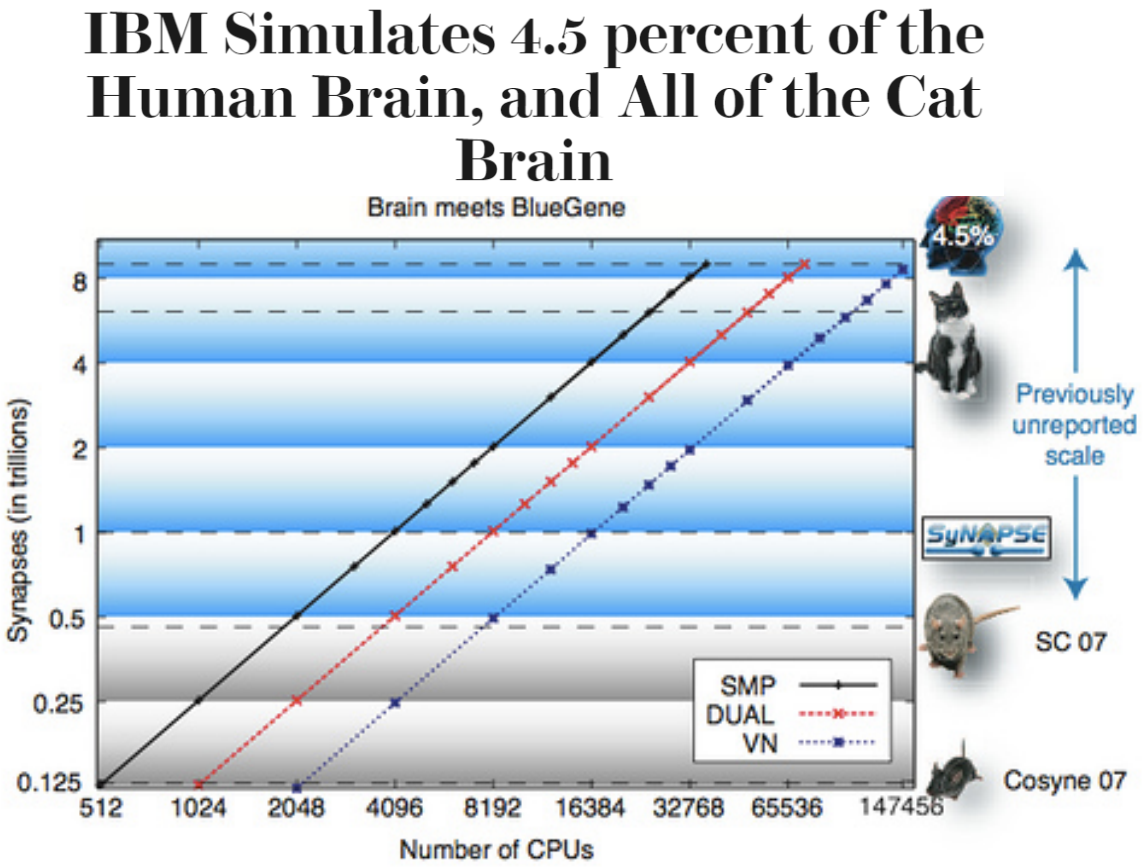


# Neural Networks: Introduction

The latest (hype): deep neural networks...

# Neural Networks: Introduction

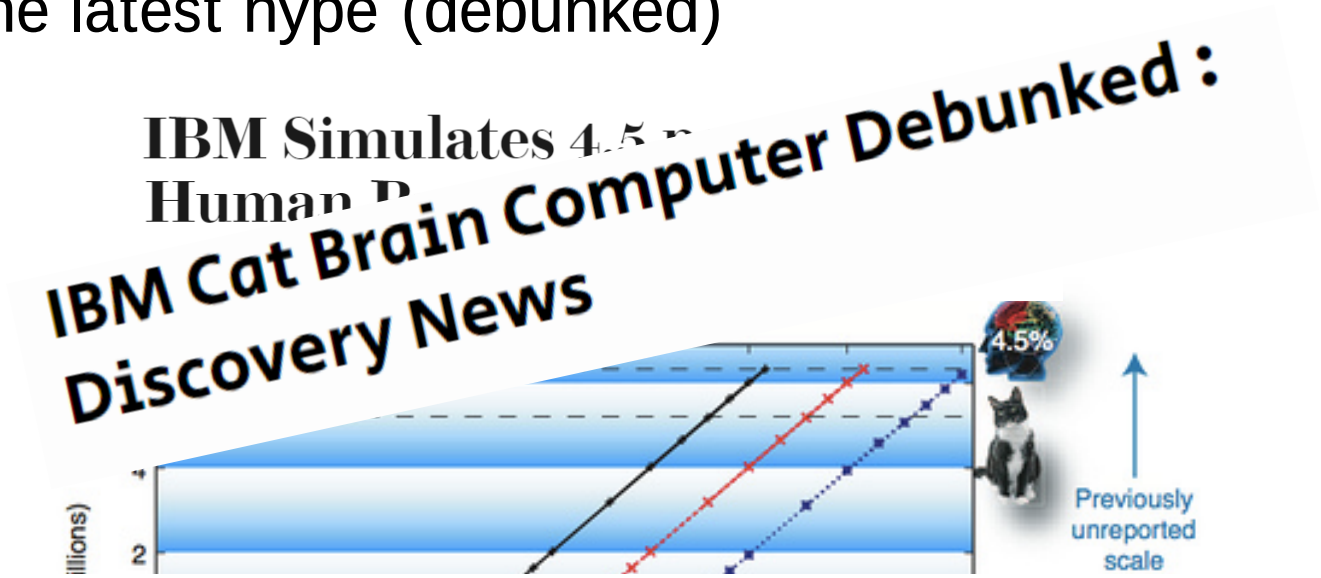
The latest hype: deep neural networks...




Ananthanarayanan et al., *The cat is out of the bag: cortical simulations with $10^9$ neurons, $10^{13}$ synapses*, 2013

# Neural Networks: Introduction

The latest hype (debunked)

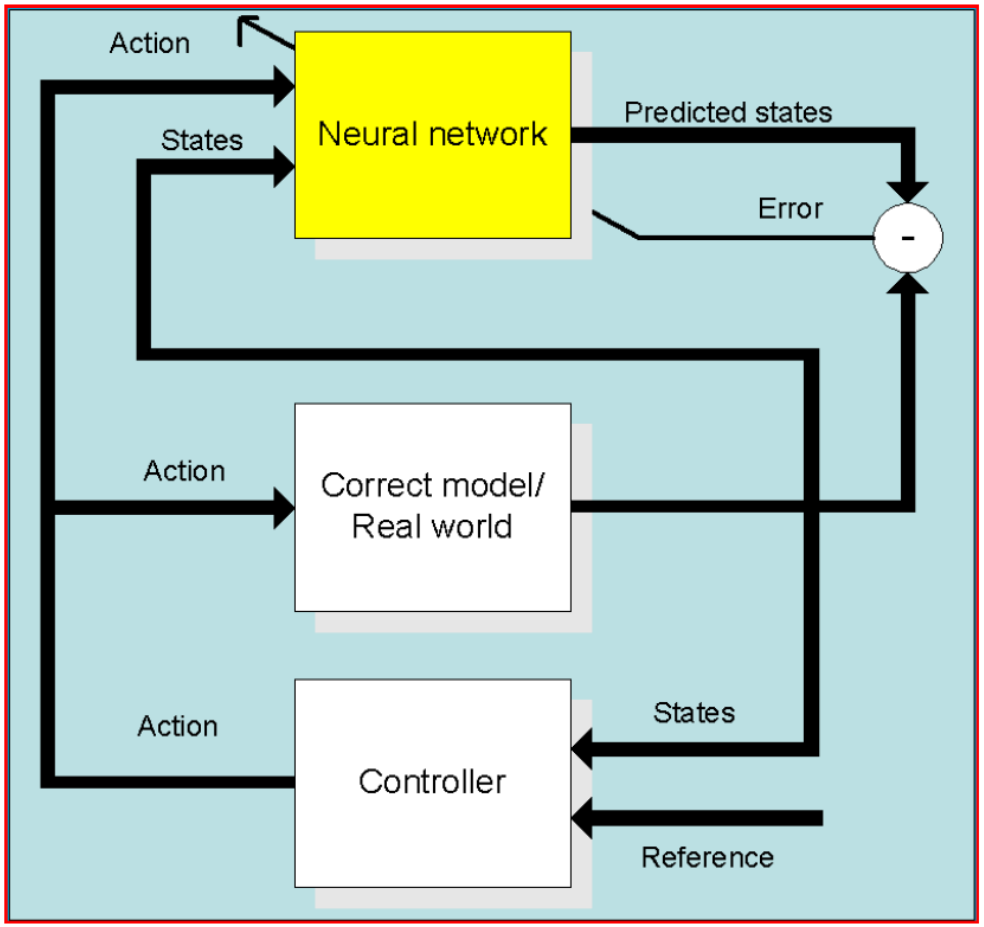



*"These 'points' they simulated and the synapses that they use for communication are literally millions of times simpler than a real cat brain. So they have not even simulated a cat's brain at one millionth of its complexity. It is not even close to an ants brain."* (*Henry Markam, EPFL, 2013*)

# Neural Networks: Introduction

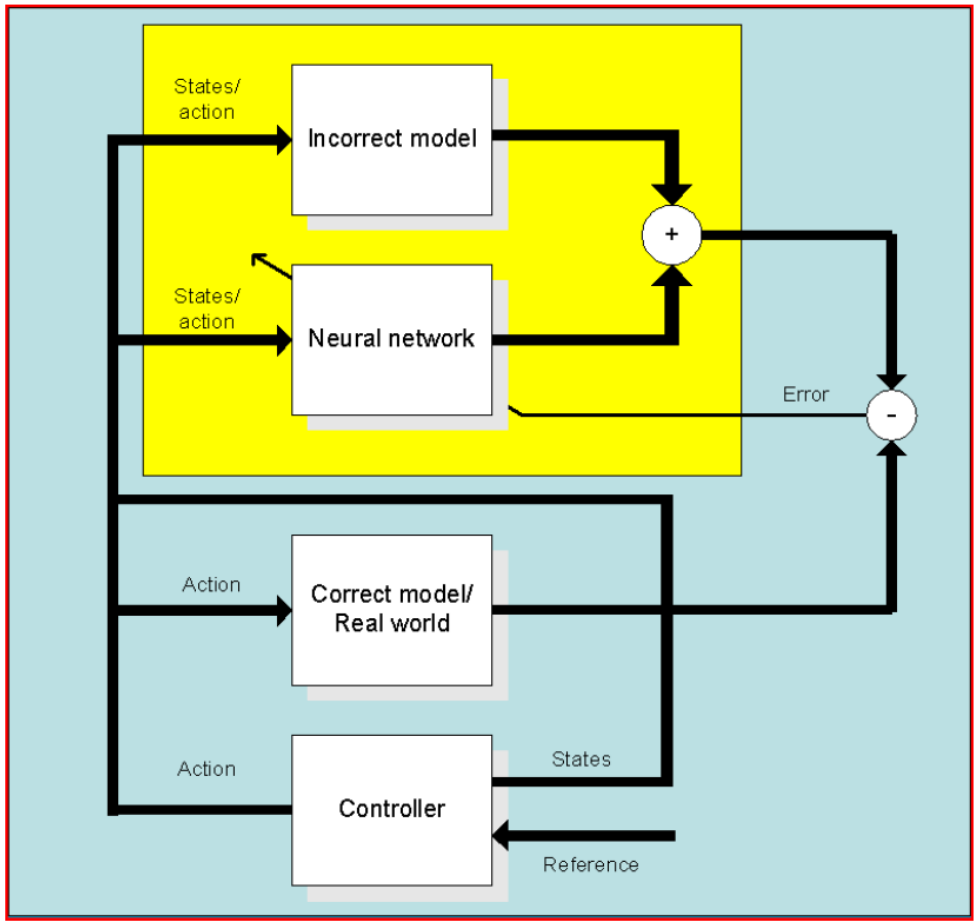
The latest hype...

# Neural Networks: Introduction




Neural network for **complete** system identification

# Neural Networks: Introduction



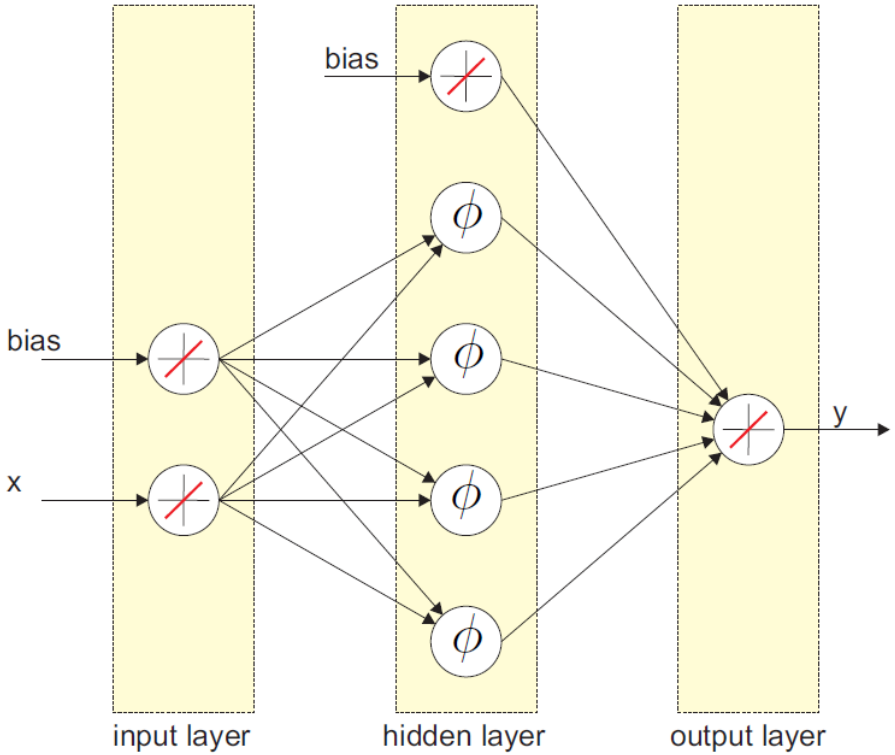

Neural network for **partial** system identification. The neural network is used to update or correct a partial model

# Neural Networks: Introduction

- All neural network types are designed for a specific task: there is not a single network type which is optimal for all tasks.
- If you are given a task ...
  - which type of neural network is the most optimal?
  - what activation function must one use?
  - how many hidden layers are needed?
  - how many neurons are needed in each layer?
  - which input and outputs are required?
  - what training algorithm must be used?
  - how can one collect the required training/testing data?
  - ...

# Visual Inspection of Neural Networks

- Neural network design and optimization is much like puzzling.
- This will be demonstrated for a simple neural network…

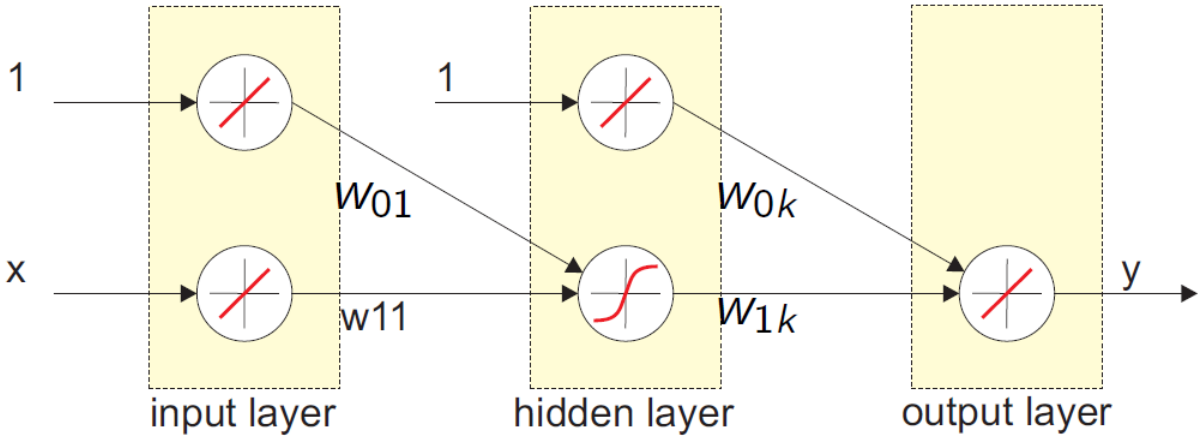



$\varphi$ = *activation function*

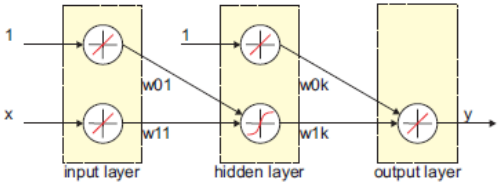
# Visual Inspection of Neural Networks

The input weights $w_{ij}$ and output weights $w_{jk}$ respectively determine the position and the shape of the activation function $\phi_j$:

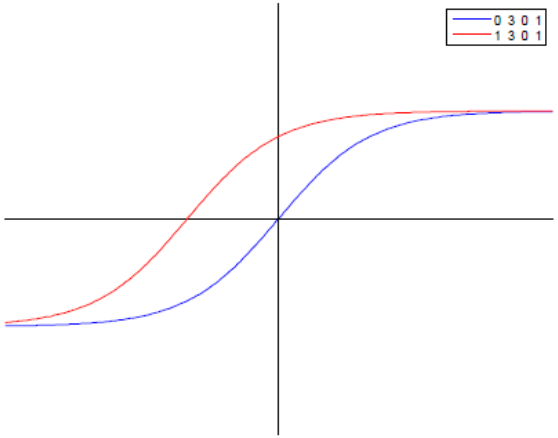
$$y = w_{1k}\phi_j(w_{11}x + w_{01}) + w_{0k}$$



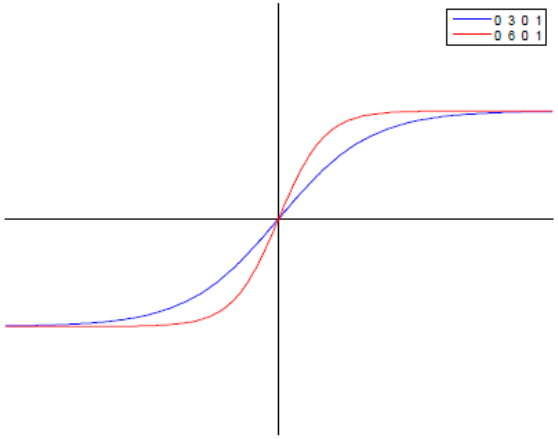
# Visual Inspection of Neural Networks



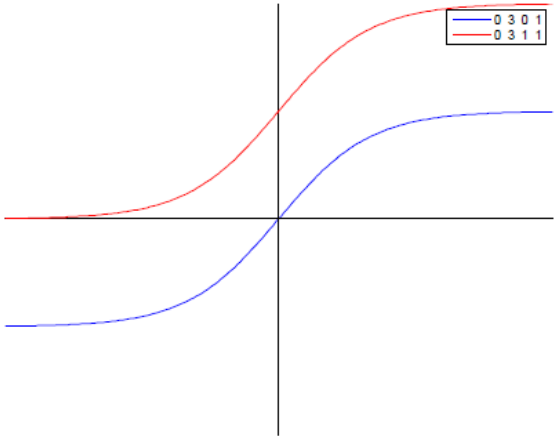

(a) FF structure



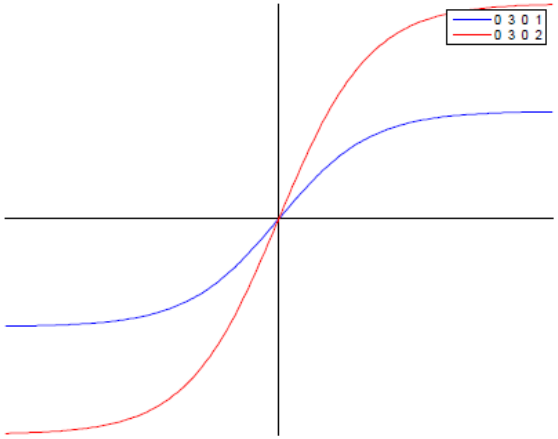

(b) Adaptation of $w_{01}$




(c) Adaptation of $w_{11}$




(d) Adaptation of $w_{0k}$




(e) Adaptation of $w_{1k}$

# Visual Inspection of Neural Networks

## Important Questions

- How can one determine the optimal set of activation functions?
- What can one do if no information is available regarding the IO mapping?

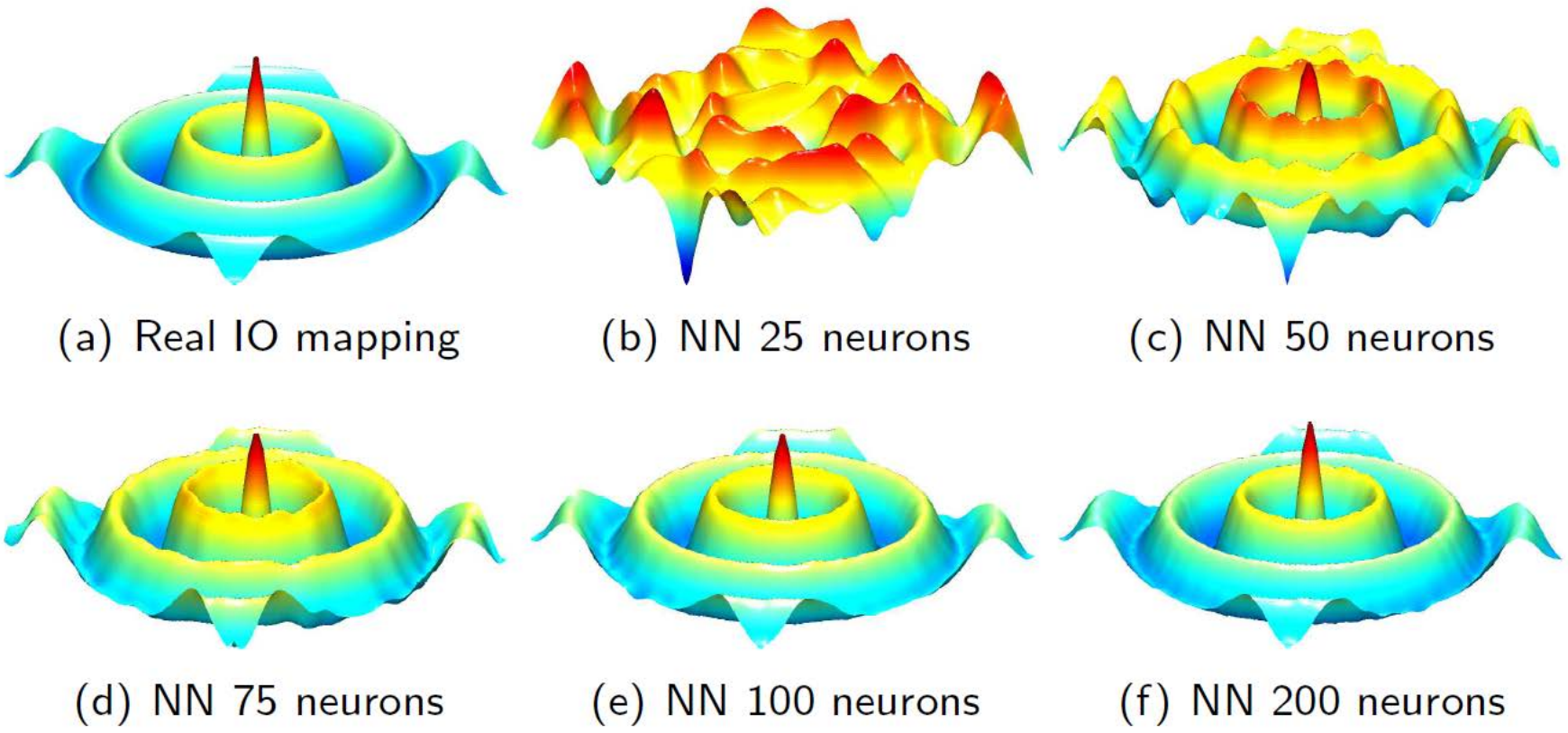
Theorem on Neural Networks:

Cybenko (1989):

*A feedforward neural net with at least one hidden layer with sigmoidal activation functions can approximate any continuous nonlinear function arbitrarily well on a compact set, provided that a sufficient number of hidden neurons are available.*

# Visual Inspection of Neural Networks

Demonstration of approximation power:

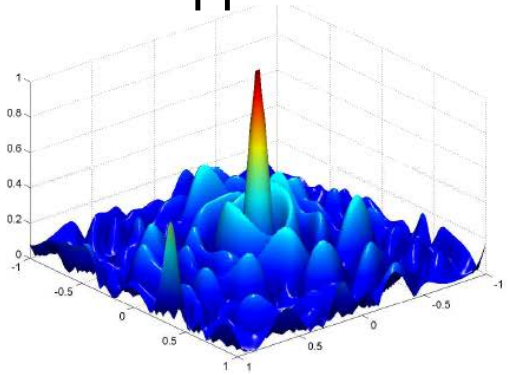


(a) Real IO mapping (b) NN 25 neurons (c) NN 50 neurons

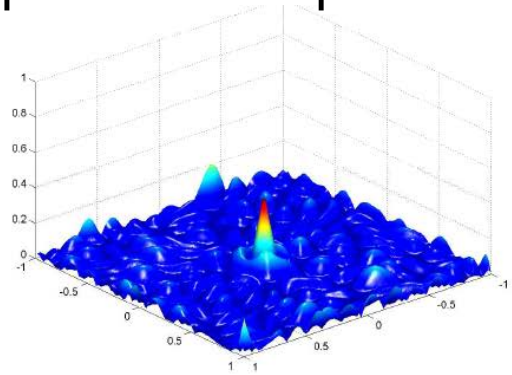
(d) NN 75 neurons (e) NN 100 neurons (f) NN 200 neurons

# Visual Inspection of Neural Networks

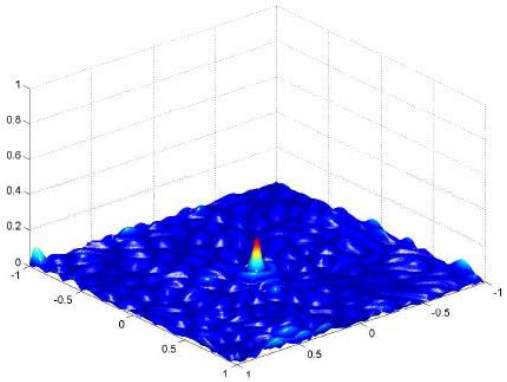
Demonstration of approximation power: Error plots



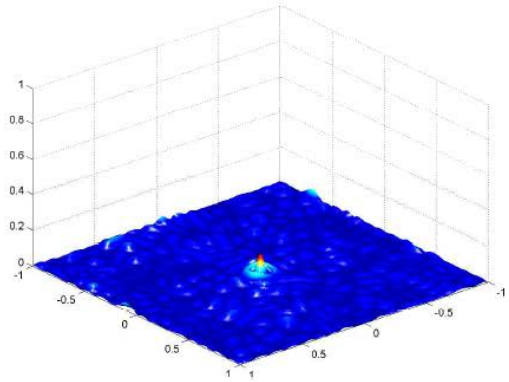
(a) NN 25 neurons



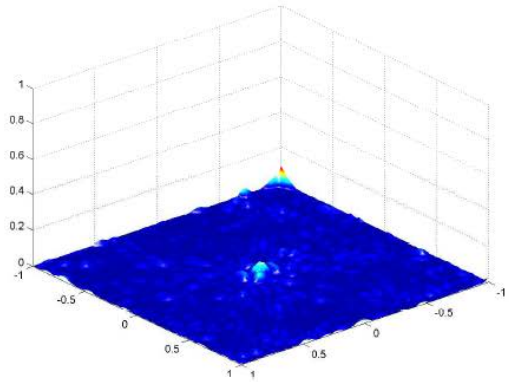
(b) NN 50 neurons



(c) NN 75 neurons



(d) NN 100 neurons



(e) NN 200 neurons

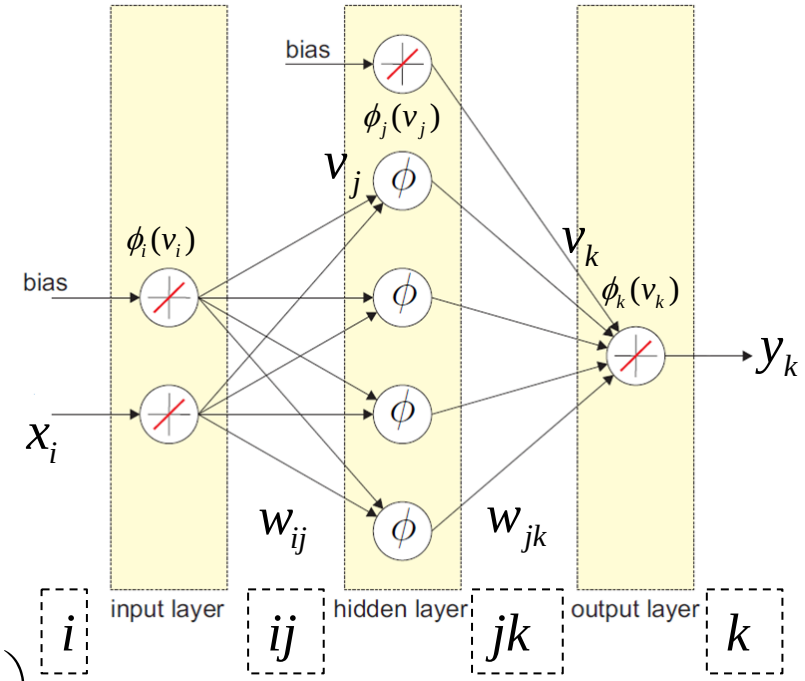
# Mathematical Description

General mathematical description of a single hidden layer feedforward neural network:

$$y_k = \phi_k(v_k)$$

$$v_k = \sum_j w_{jk} y_j = \sum_j w_{jk} \phi_j(v_j)$$

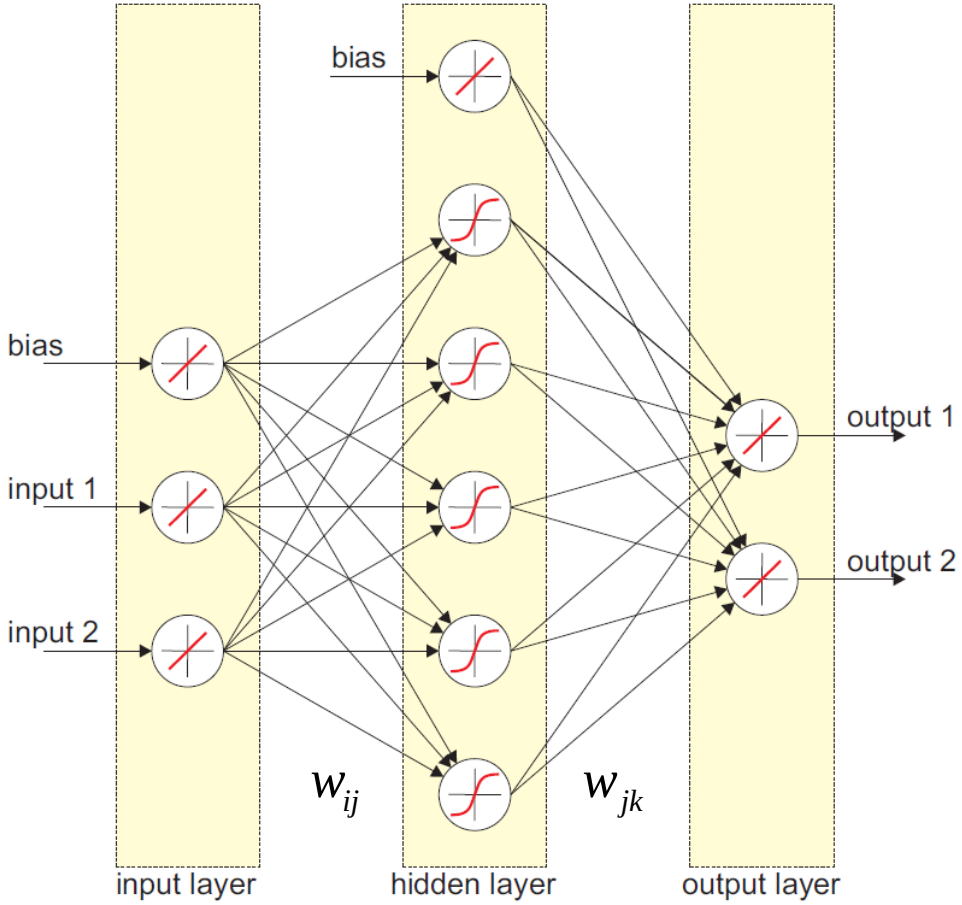
$$v_j = \sum_i w_{ij} y_i = \sum_i w_{ij} \phi_i(v_i)$$




Simplified mathematical description:

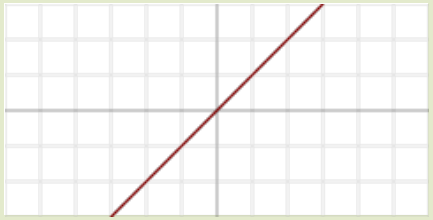
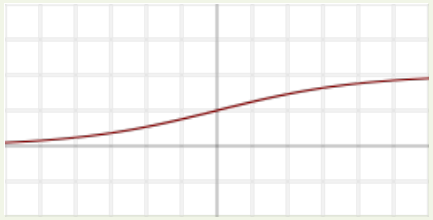
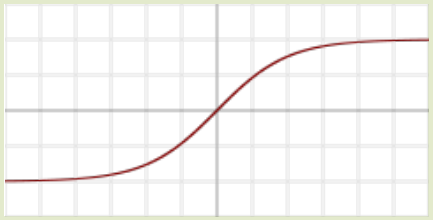
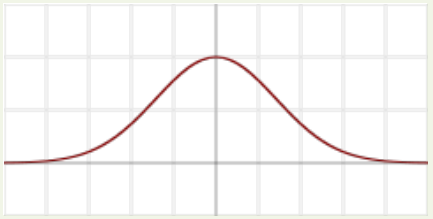
$$\left.\begin{aligned} y_k &= v_k \\ v_k &= \sum_j w_{jk} \phi_j(v_j) \\ v_j &= \sum_i w_{ij} x_i \end{aligned}\right\} \quad y_k = \sum_j w_{jk} \phi_j \left( \sum_i w_{ij} x_i \right)$$

# Visual Inspection of Neural Networks




When we are “training” a neural network, we are actually determining the values of the neural network weights: **the weights are the parameters of the neural network** model structure!

# Mathematical Description

| Activation function | plot | function | derivative |
|---|---|---|---|
| Identity (linear) |  | $\varphi(x) = x$ | $\varphi'(x) = 1$ |
| Sigmoid (logarithmic) |  | $\varphi(x) = \frac{1}{1 + e^{-x}}$ | $\varphi'^{(x)} = \varphi(x)(1 - \varphi(x))$ |
| TanH (hyperbolic Tangent) |  | $\varphi(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$ | $\varphi'(x) = 1 - \varphi(x)^2$ |
| Radial basis function (Gaussian) |  | $\varphi(x) = e^{-x^2}$ | $\varphi'(x) = -2xe^{-x^2}$ |

# Mathematical Description

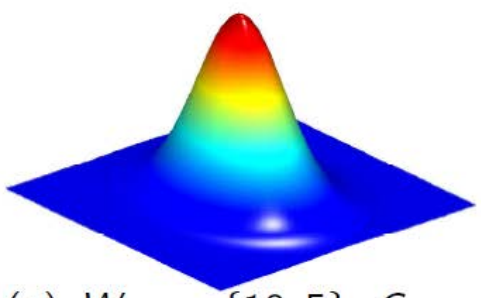
- The feedforward neural networks with **sigmoidal** activation functions $\phi_j(v_j)$ can approximate any function.
- It is one of the most popular networks.
- Another very popular type of network is the radial basis function neural network:

$$\left.\begin{aligned} y_k &= v_k \\ v_k &= \sum_j w_{jk}\phi_j(v_j) \\ v_j &= \sum_i w_{ij}\left(x_i - c_{ij}\right)^2 \end{aligned}\right\} \quad y_k = \sum_j w_{jk}\phi_j\left(\sum_i w_{ij}\left(x_i - c_{ij}\right)^2\right)$$

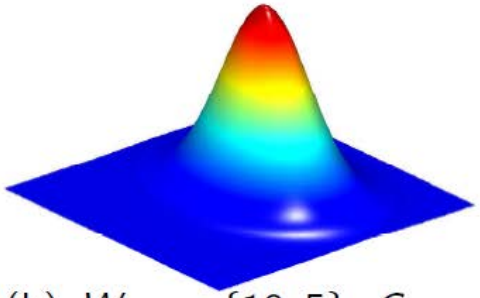
The activation function looks like…

# Mathematical Description

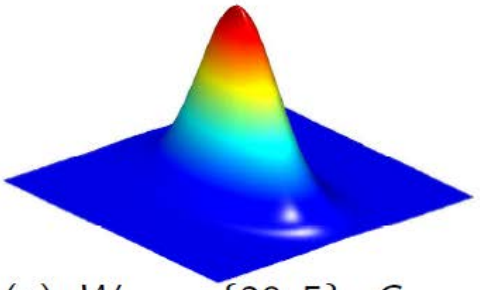
RBF activation functions for different centers and weights



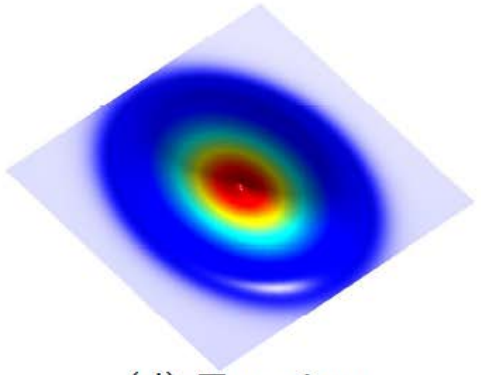
(a) $W_{ij} = \{10, 5\}, C_{ij} = \{0, 0\}$



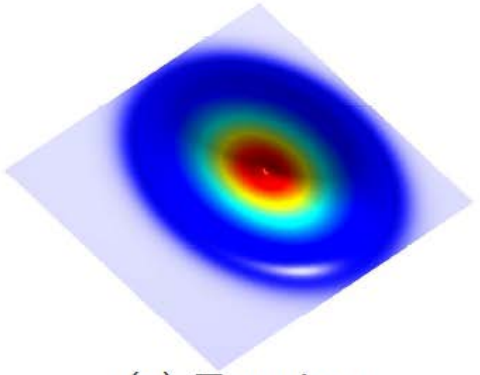
(b) $W_{ij} = \{10, 5\}, C_{ij} = \{1, 0\}$



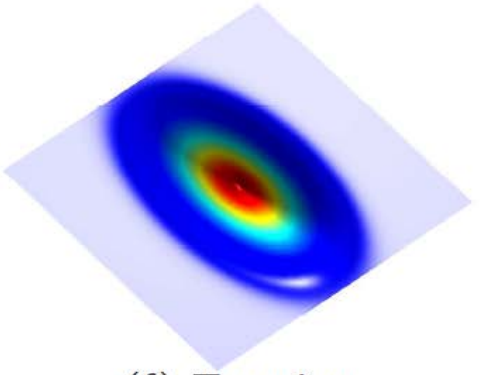
(c) $W_{ij} = \{20, 5\}, C_{ij} = \{0, 0\}$



(d) Top view



(e) Top view



(f) Top view

# Mathematical Description

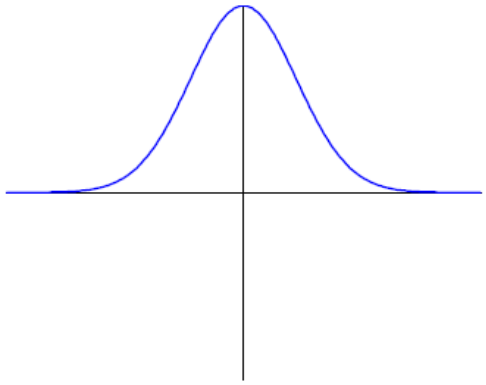
## RBF network

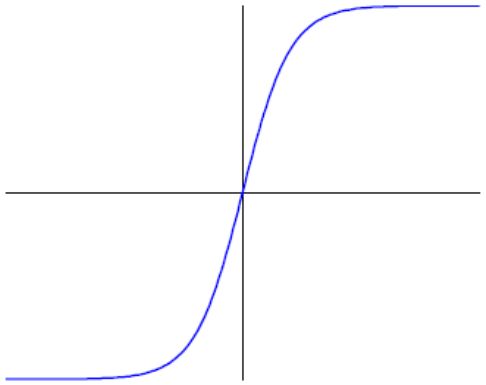
- Local character
- Easy to model local complexity
- Bad generalization?
- High computation load
- Easy optimization

$\Longleftrightarrow$

## FF network

- Global character
- Easy to model global trends
- Good generalization?
- Low computation load
- Difficult optimization





# Mathematical Description

## Summary

- Two commonly used neural networks introduced
- Neural network output is a summation of activation functions
- Characteristics of IO mapping should be reflected by the activation functions
- Shape of the activation functions adjusted by network weights
- Information stored in the network weights
- IO mapping complexity together with choice in activation function determines the required number of neurons.

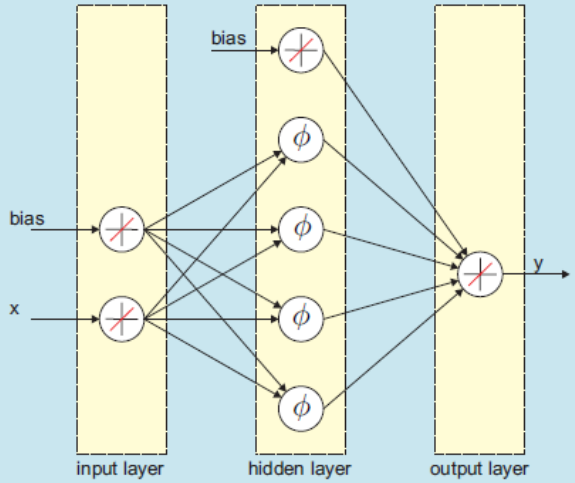
# Training Neural Networks

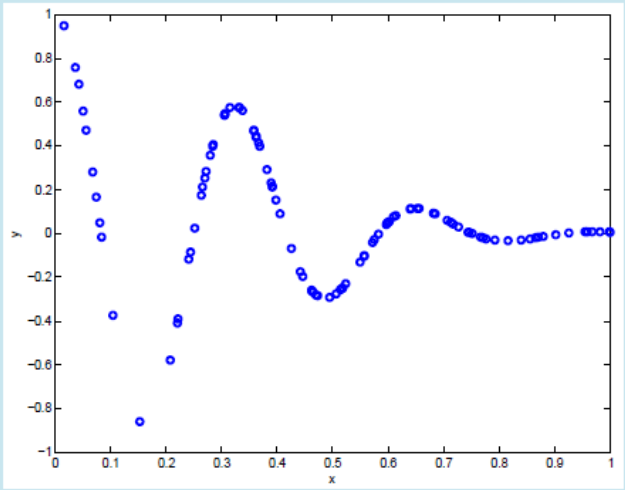
## Start-off point

- Single hidden layer feedforward neural network (fixed structure)

$$y_k = \sum_j w_{jk} \phi_j \left( \sum_i w_{ij} x_i \right)$$

- Static IO mapping, fixed set of data $(x, d)$.



# Training Neural Networks

Aspects of training neural networks:

- Network structure
- Activation function type
- Network weight initialization
- IO data pre-processing
- Adaptation rules
- Stopping conditions

For now consider ...

- Random initialization
- No data pre-processing

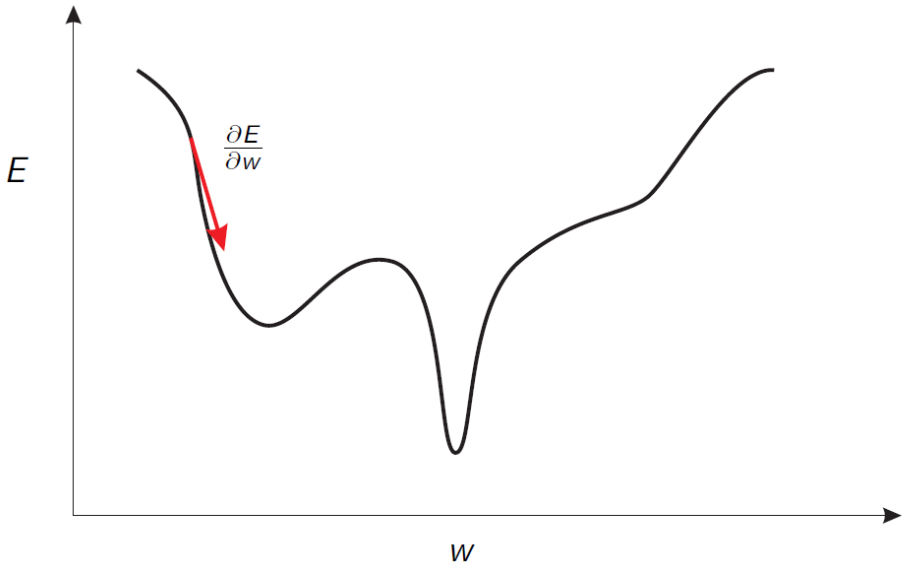
Next: error back propagation...

# Training Neural Networks

Network performance is measured with a squared cost function:

$$E = \frac{1}{2}\sum_{k}(d_k - y_k)^2$$

The error-back propagation approach tries to minimize the cost function by traveling down the slope (gradient method!):



# Training Neural Networks

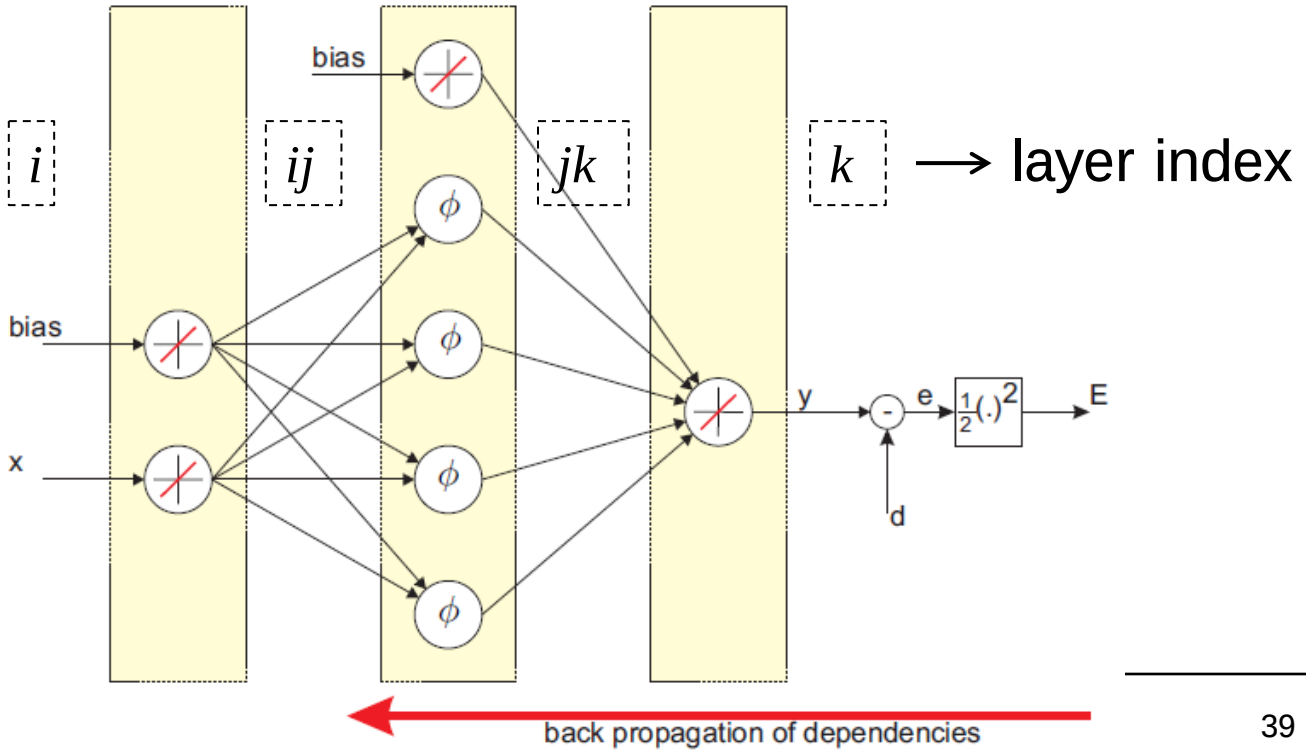
- Updates performed using partial derivatives: step taken in the negative gradient direction.
- Update law for the network weights given by:

$$w_{t+1} = w_t + \Delta w; \qquad \Delta w = -\eta \frac{\partial E}{\partial w_t}$$

- Update law depends on
  1. partial derivatives
  2. learning rate parameter $\eta$
- Partial derivative can be computed in a structured way using the chain rule: **back-propagation** of the errors

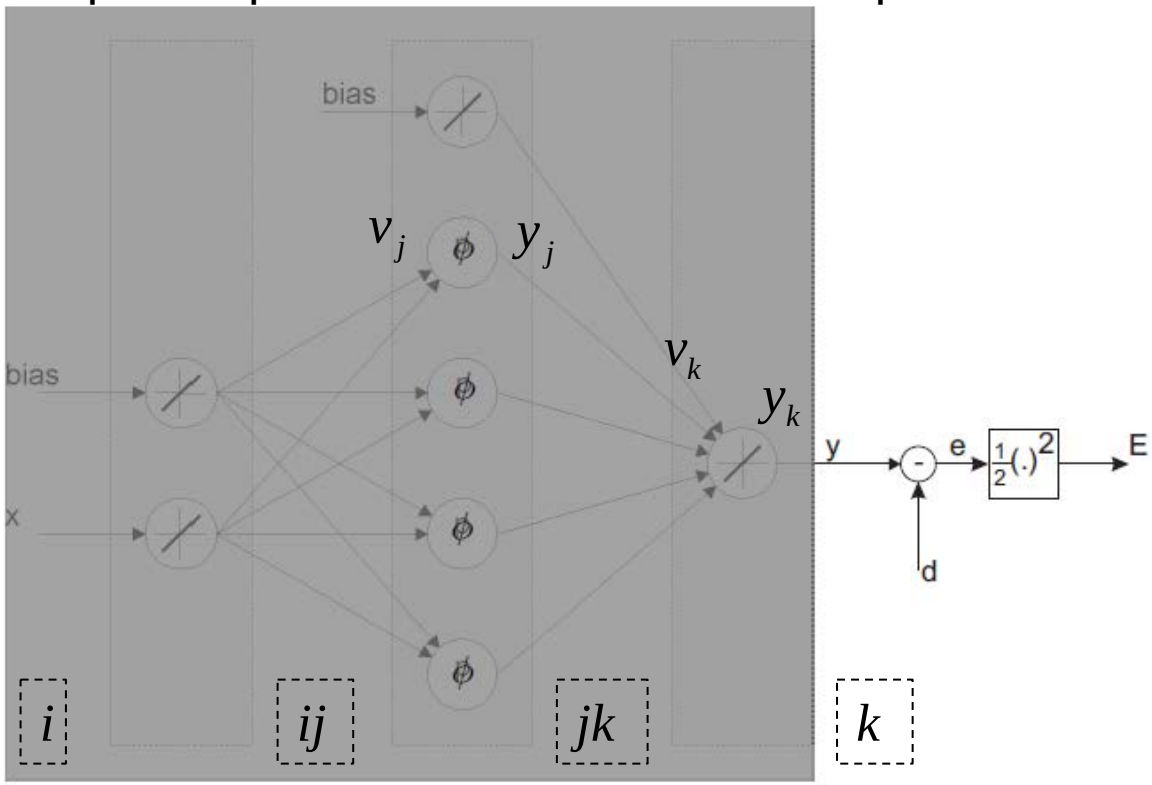
# Error back-propagation

- First a forward computation step is performed to compute the neuron inputs ($v$ ) and outputs ($y$) (consider a single IO data point)
- Then the output errors are computed ($e_k$ )
- Finally the cost function dependencies on the weights are propagated from right to left…



# Error back-propagation

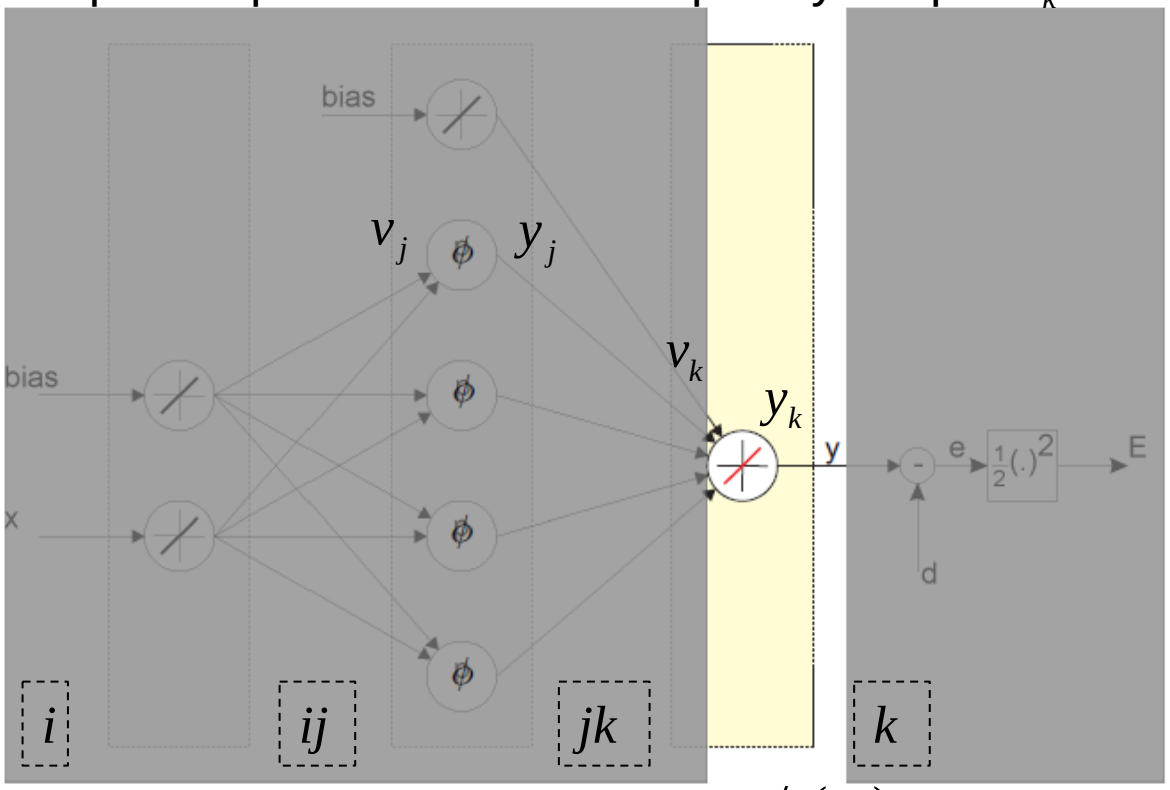
Step 1: Compute dependencies w.r.t. network outputs




$$E = \sum_k \frac{1}{2}\left(d_k - y_k\right)^2 \quad \rightarrow \quad \frac{\partial E}{\partial y_k} = \frac{\partial E}{\partial e_k}\frac{\partial e_k}{\partial y_k} = e_k \cdot -1$$

# Error back-propagation

Step 2: Compute dependencies w.r.t. output layer input $v_k$



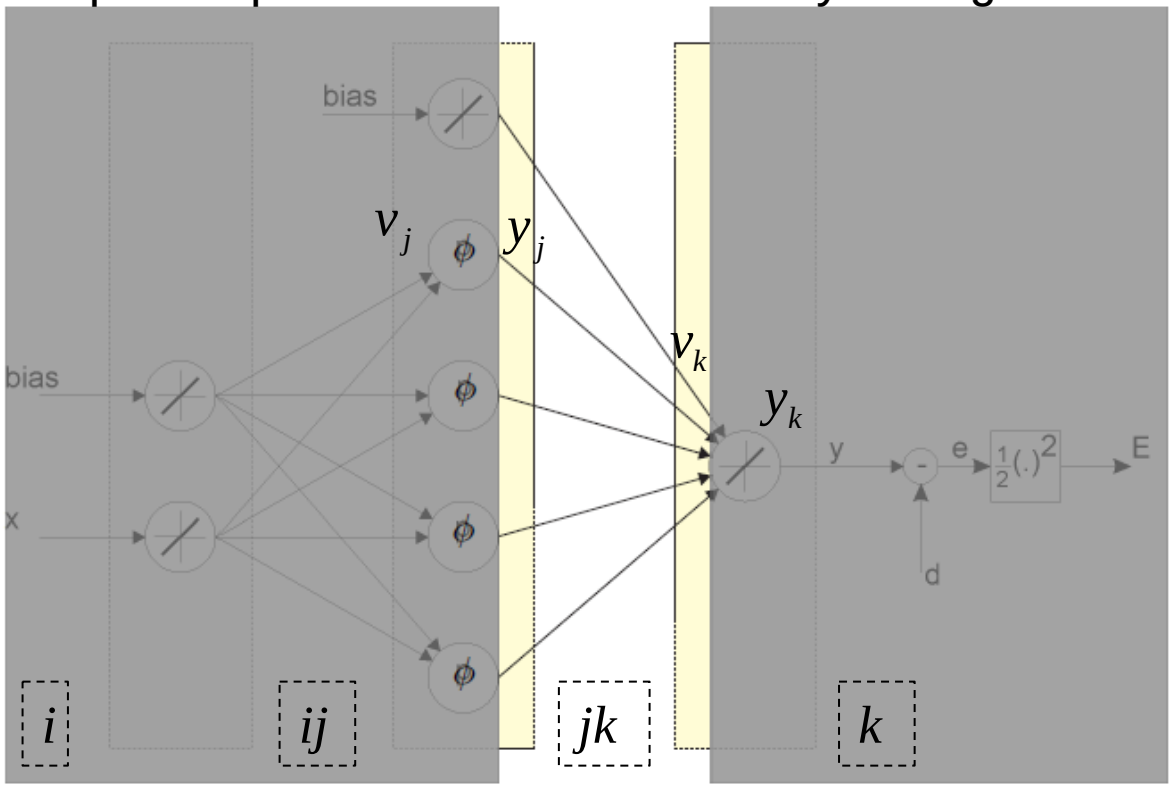

$$y_k = \phi_k(v_k)$$

$$\frac{\partial y_k}{\partial v_k} = \frac{\partial \phi_k(v_k)}{\partial v_k}$$

linear activation function: $\rightarrow \frac{\partial y_k}{\partial v_k} = 1$

# Error back-propagation

Step 3: Compute dependencies w.r.t. hidden layer weights




$$\frac{\partial E}{\partial y_k} = e_k \cdot -1$$

$$\frac{\partial y_k}{\partial v_k} = \frac{\partial \phi_k(v_k)}{\partial v_k}$$

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

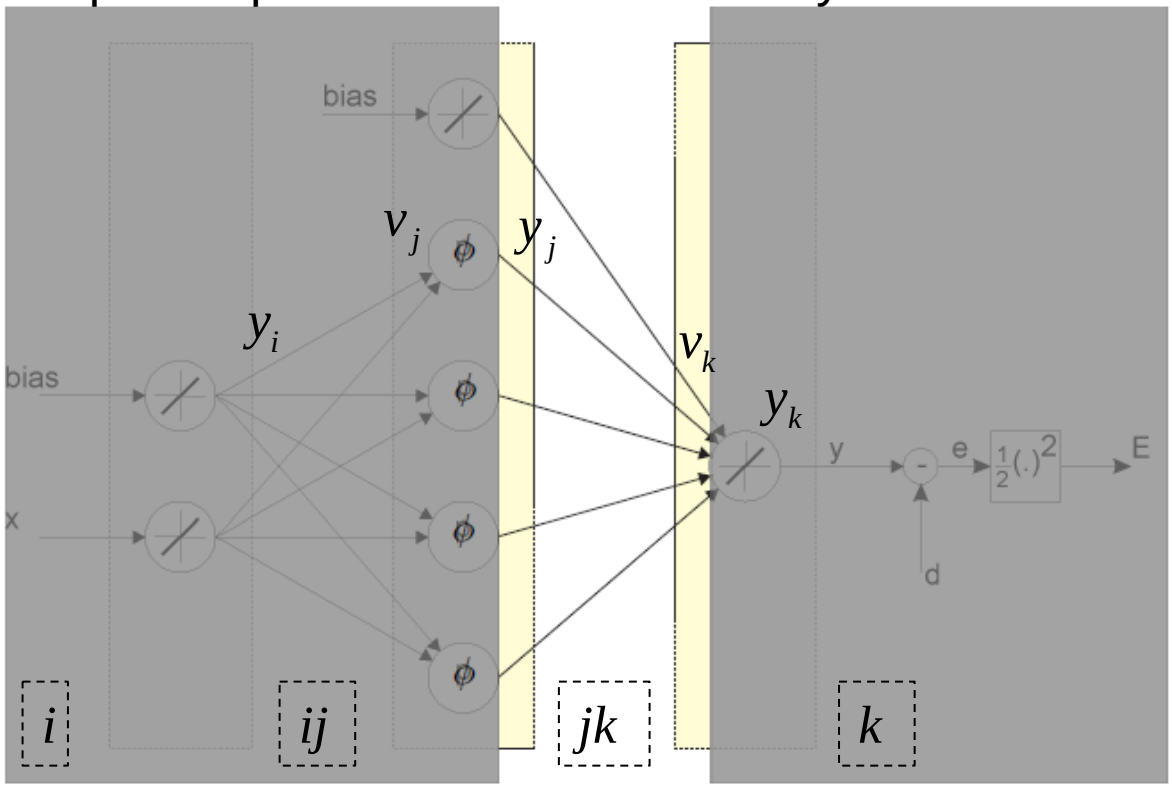
$$v_k = \sum_j w_{jk} y_j$$

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

$$\rightarrow \quad \frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial v_k}\frac{\partial v_k}{\partial w_{jk}} = e_k \cdot -1 \cdot \frac{\partial \phi_k(v_k)}{\partial v_k} y_j$$

# Error back-propagation

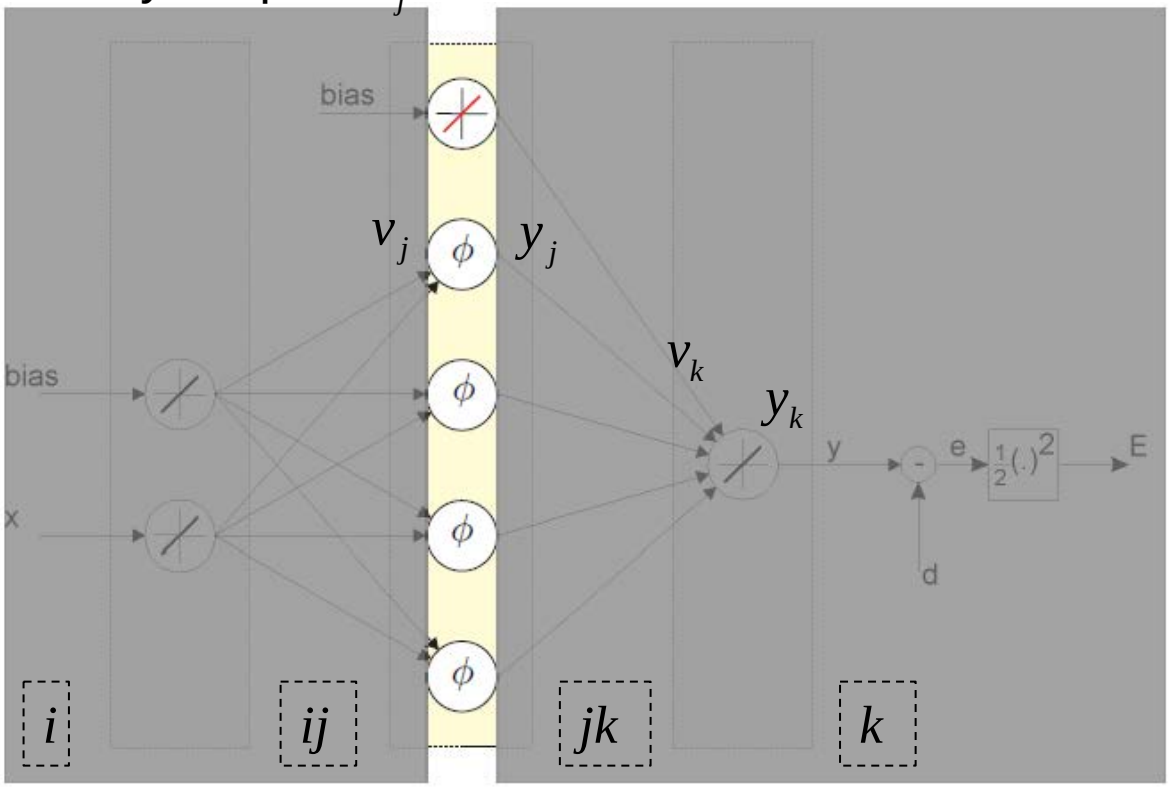
Step 4: Compute dependencies w.r.t. hidden layer activation function output $y_j$




$$v_k = \sum_j w_{jk} y_j \qquad \frac{\partial v_k}{\partial y_j} = w_{jk} \qquad \rightarrow \qquad \frac{\partial E}{\partial y_j} = \sum_k \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial v_k} \frac{\partial v_k}{\partial y_j}$$

# Error back-propagation

Step 5: Compute dependencies of hidden layer activation function outputs $y_j$ w.r.t. hidden layer inputs $v_j$

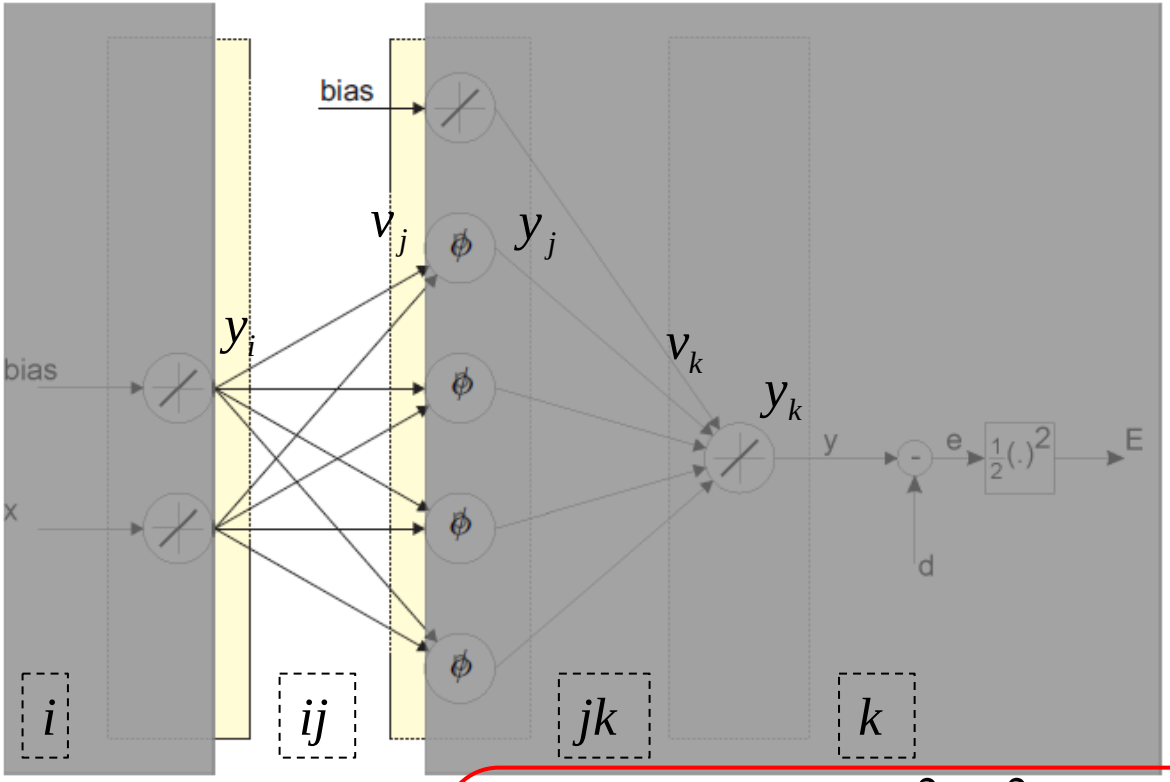



$$y_j = \phi_j(v_j)$$

$$\frac{\partial y_j}{\partial v_j} = \frac{\partial \phi_j(v_j)}{\partial v_j}$$

# Error back-propagation

Step 6: Compute dependencies w.r.t. input weights




$$\frac{\partial E}{\partial y_k} = e_k \cdot -1$$

$$\frac{\partial y_k}{\partial v_k} = \frac{\partial \phi_k(v_k)}{\partial v_k}$$

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

$$\frac{\partial v_j}{\partial w_{ij}} = y_i$$

$$\frac{\partial y_j}{\partial v_j} = \frac{\partial \phi_j(v_j)}{\partial v_j}$$

$$v_j = \sum_i w_{ij} y_i$$

$$\frac{\partial v_j}{\partial w_{ij}} = y_i$$

$$\rightarrow$$

$$\frac{\partial E}{\partial w_{ij}} = \sum_k \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial v_k} \frac{\partial v_k}{\partial y_j} \frac{\partial y_j}{\partial v_j} \frac{\partial v_j}{\partial w_{ij}}$$

$$= \sum_k e_k \cdot -1 \cdot \frac{\partial \phi_k(v_k)}{\partial v_k} \cdot w_{jk} \cdot \frac{\partial \phi_j(v_j)}{\partial v_j} \cdot y_i$$

# Error back-propagation

linear activation function:

$$\frac{\partial y_{k,q}}{\partial v_{k,q}} = \frac{\partial \phi_k(v_{k,q})}{\partial v_{k,q}} = 1$$

Complete partial derivatives (for all 'q' datapoints)

1. **Output weights** (linear activation function):

$$\frac{\partial E}{\partial w_{jk}} = \sum_q \frac{\partial E}{\partial e_{k,q}} \frac{\partial e_{k,q}}{\partial y_{k,q}} \frac{\partial y_{k,q}}{\partial v_{k,q}} \frac{\partial v_{k,q}}{\partial w_{jk}} = \sum_q e_{k,q} \cdot -1 \cdot y_j$$

2. **Input weights** (linear activation function)

$$\frac{\partial E}{\partial w_{ij}} = \sum_q \left[ \sum_k \frac{\partial E}{\partial e_{k,q}} \frac{\partial e_{k,q}}{\partial y_{k,q}} \frac{\partial y_{k,q}}{\partial v_{k,q}} \frac{\partial v_{k,q}}{\partial y_{j,q}} \frac{\partial y_{j,q}}{\partial v_{j,q}} \frac{\partial v_{j,q}}{\partial w_{ij}} \right]$$

$$= \sum_q \left[ \sum_k e_k \cdot -1 \cdot \frac{\partial \phi_k(v_k)}{\partial v_k} \cdot w_{jk} \cdot \frac{\partial \phi_j(v_j)}{\partial v_j} \cdot x_i \right]$$

$$= \sum_q \left[ \sum_k e_k \cdot -1 \cdot w_{jk} \cdot \frac{\partial \phi_j(v_j)}{\partial v_j} \cdot x_i \right]$$

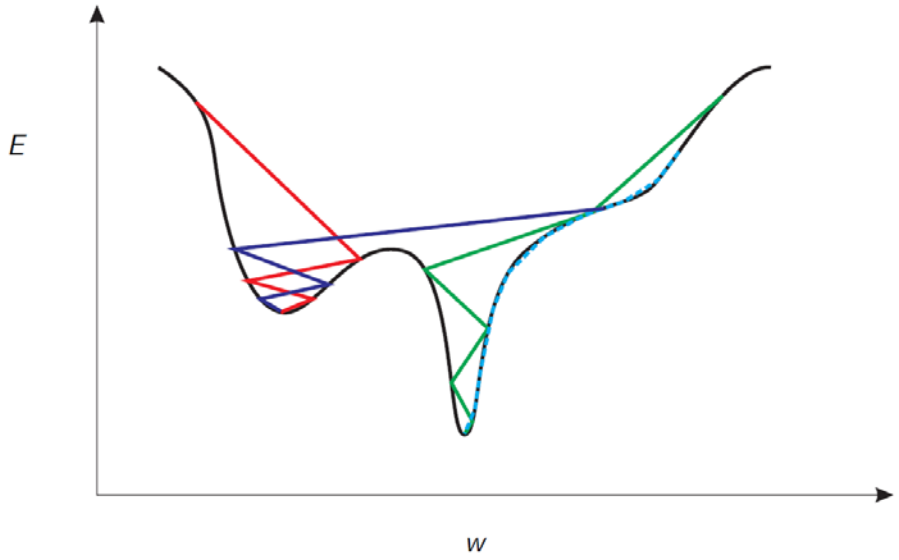
for multiple hidden layers, use the same method!

# Error back-propagation

Now that the derivatives are known the update of the weights can be performed:

$$w_{t+1} = w_t + \Delta w; \qquad \Delta w = -\eta \frac{\partial E}{\partial w_t}$$

- The learning rate $\eta$ is used to scale the weight increment
- Together with the initialization point they determine the final approximation performance:



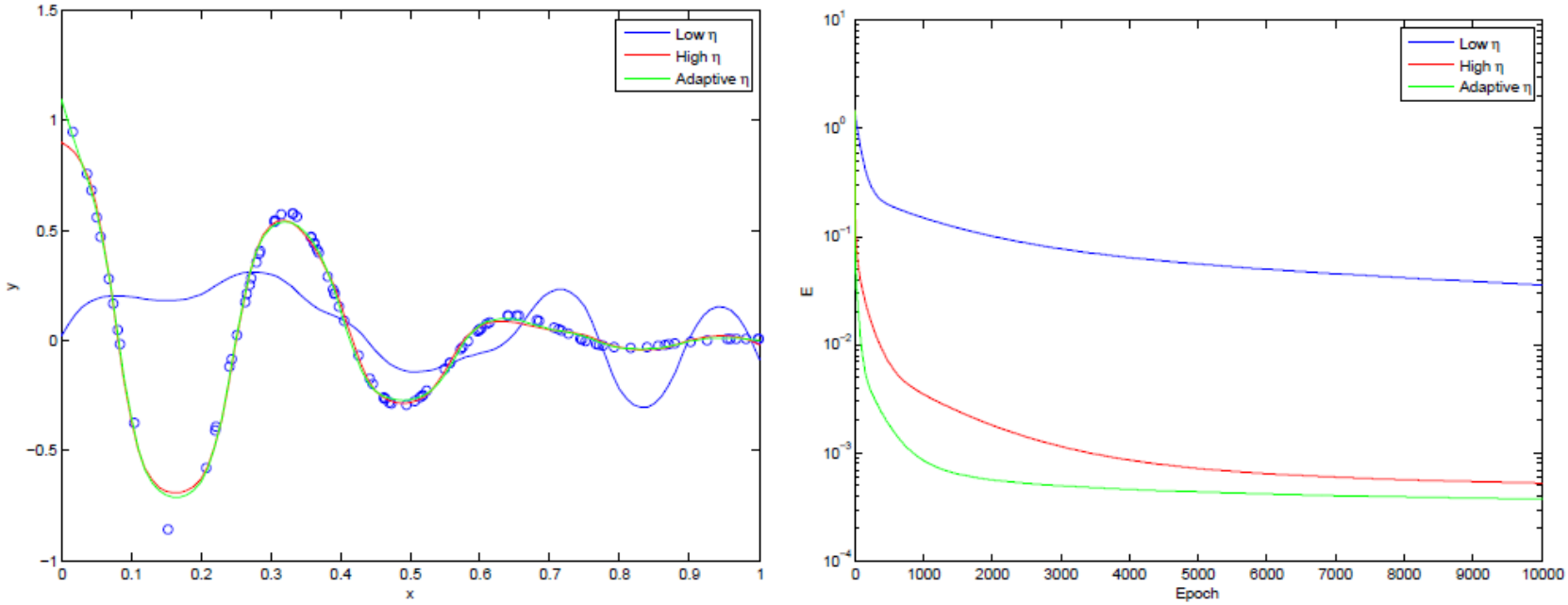
# Error back-propagation

## Basic learning algorithm: fixed learning rate

- Initialization
- LOOP (number of epochs)
  - Compute cost function value $E_t$ and weight update $\Delta W$ for current set of weights $W_t$
  - Perform update of weights and compute new cost function value $E_{t+1}$
  - If $E_{t+1} < E_t$ then accept changes else stop loop
- END LOOP

- Small learning rate: **slow convergence** to local minimum (derivatives zero)
- Large learning rate: **fast convergence**, but can become unstable.

# Error back-propagation




(a) Approximation (b) Performance

Figure: Influence of the learning rate on the performance time history during training

# Error back-propagation

## Learning algorithm: adaptive learning rate

- Initialization
- LOOP (number of epochs)
  - Compute cost function value $E_t$ and weight update $\Delta W$ for current set of weights $W_t$ and $\eta_t$
  - Perform update of weights and compute new cost function value $E_{t+1}$
  - If $E_{t+1} < E_t$ then accept changes and increase learning rate $\eta_{t+1} = \eta_t * \alpha$ else do not accept changes and decrease learning rate $\eta_{t+1} = \eta_t * \alpha^{-1}$.
  - Stop loop if partial derivatives are (nearly) zero.
- END LOOP

# Error back-propagation

## Levenberg-Marquardt

- The previous training algorithms are all first order gradient-descent algorithms (a.k.a. steepest descent)
- There are second order methods available of which the most commonly used is the Levenberg-Marquardt method:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \left(\mathbf{J}^{\mathrm{T}}\mathbf{J} + \mu\mathbf{I}\right)^{-1}\mathbf{J}^{\mathrm{T}}\mathbf{e}$$

  - with $\mu$ an (adaptive) damping parameter
  - with $e = [e(1) \quad e(1) \quad \cdots \quad e(N)]^T$ a (column) vector containing the error
  - if there are $K$ input weights and $M$ output weights, the Jacobian $\mathbf{J}$ is defined as:

$$\mathbf{J} = \frac{\partial \mathbf{e}}{\partial \mathbf{w}_t} = \begin{bmatrix} \frac{\partial \mathbf{e}(1)}{\partial \mathbf{w}_{ij}(1)} & \frac{\partial \mathbf{e}(1)}{\partial \mathbf{w}_{ij}(2)} & \cdots & \frac{\partial \mathbf{e}(1)}{\partial \mathbf{w}_{ij}(K)} & \frac{\partial \mathbf{e}(1)}{\partial \mathbf{w}_{jk}(1)} & \frac{\partial \mathbf{e}(1)}{\partial \mathbf{w}_{jk}(2)} & \cdots & \frac{\partial \mathbf{e}(1)}{\partial \mathbf{w}_{jk}(M)} \\ \frac{\partial \mathbf{e}(2)}{\partial \mathbf{w}_{ij}(1)} & \frac{\partial \mathbf{e}(2)}{\partial \mathbf{w}_{ij}(2)} & \cdots & \frac{\partial \mathbf{e}(2)}{\partial \mathbf{w}_{ij}(K)} & \frac{\partial \mathbf{e}(2)}{\partial \mathbf{w}_{jk}(1)} & \frac{\partial \mathbf{e}(2)}{\partial \mathbf{w}_{jk}(2)} & \cdots & \frac{\partial \mathbf{e}(2)}{\partial \mathbf{w}_{jk}(M)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{e}(N)}{\partial \mathbf{w}_{ij}(1)} & \frac{\partial \mathbf{e}(N)}{\partial \mathbf{w}_{ij}(2)} & \cdots & \frac{\partial \mathbf{e}(N)}{\partial \mathbf{w}_{ij}(K)} & \frac{\partial \mathbf{e}(N)}{\partial \mathbf{w}_{jk}(1)} & \frac{\partial \mathbf{e}(N)}{\partial \mathbf{w}_{jk}(2)} & \cdots & \frac{\partial \mathbf{e}(N)}{\partial \mathbf{w}_{jk}(M)} \end{bmatrix}$$

data samples

input weights

output weights

# Error back-propagation

## Levenberg-Marquardt: tuning

- The LM method can be tuned by changing the value of $\mu$. Starting with the full LM:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \left(\mathbf{J}^{\mathrm{T}}\mathbf{J} + \mu\mathbf{I}\right)^{-1}\mathbf{J}^{\mathrm{T}}\mathbf{e}$$

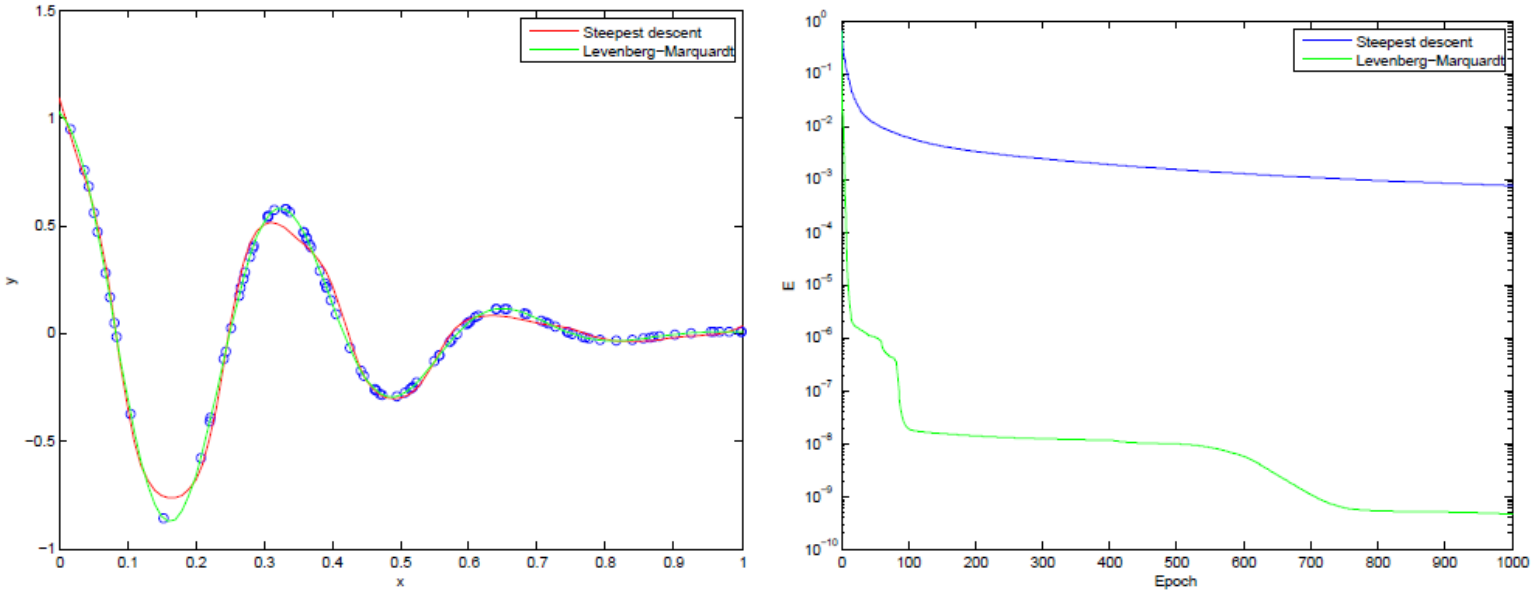
- If $\mu$ is larger then the LM method reduces to a first order steepest-descent method:

$$\mathbf{w}_{t+1} \approx \mathbf{w}_t - \left(\mu\mathbf{I}\right)^{-1}\mathbf{J}^{\mathrm{T}}\mathbf{e} = \mathbf{w}_t - \frac{1}{\mu}\mathbf{J}^{\mathrm{T}}\mathbf{e}$$

- If $\mu$ is small then the LM method reduces to a Newton-like step with $\mathbf{J}^T\mathbf{J}$ as an approximation of the Hessian matrix:

$$\mathbf{w}_{t+1} \approx \mathbf{w}_t - \left(\mathbf{J}^{\mathrm{T}}\mathbf{J}\right)^{-1}\mathbf{J}^{\mathrm{T}}\mathbf{e}$$

# Error back-propagation




(a) Approximation (b) Performance

Figure: Influence of the training algorithm on the performance time history during training

Levenberg-Marquardt is much more efficient and shows faster convergence!

# Error back-propagation

## 1 order method

- Low computation cost
- Numerically stable
- Slower convergence
- Gentle weight updates
- Higher probability of local minimum
- Easy implementation

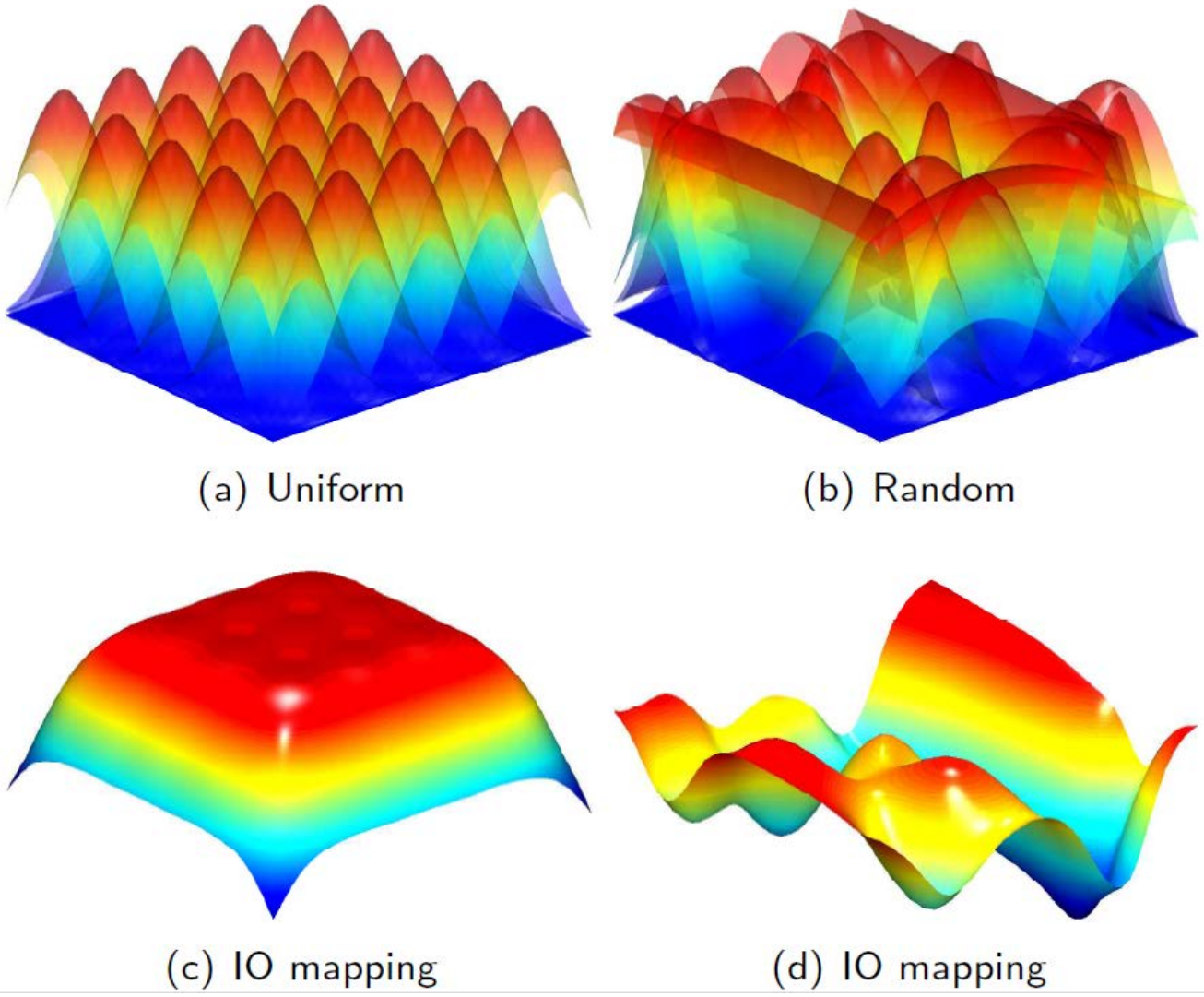
## 2 order method

- Higher computation cost
- Fast convergence
- Aggressive weight updates
- Higher probability of finding global minimum
- More difficult implementation

# Training Neural Networks: Initialization

- As mentioned previously, the initialization of the weights has a large influence.
- For feedforward neural networks with sigmoidal activation function **random initialization** is used: small weight values means that the first order derivative is non-zero in the entire input space.
- For the RBF neural networks one can do the same but also uniform distribution can be performed (see next slide).
- Optimal initialization requires knowledge of the true IO mapping.
- Common practice is to perform the training several time, each time with different initial weights.

# Training Neural Networks: Initialization



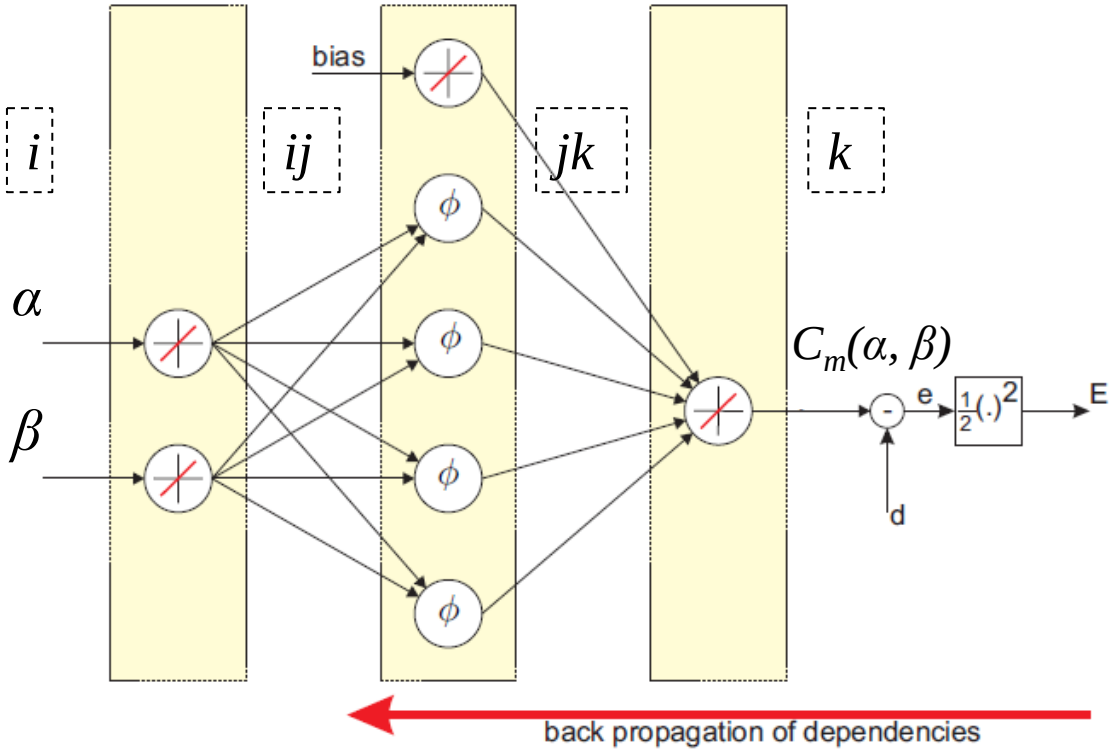
# Training Neural Networks: Summary

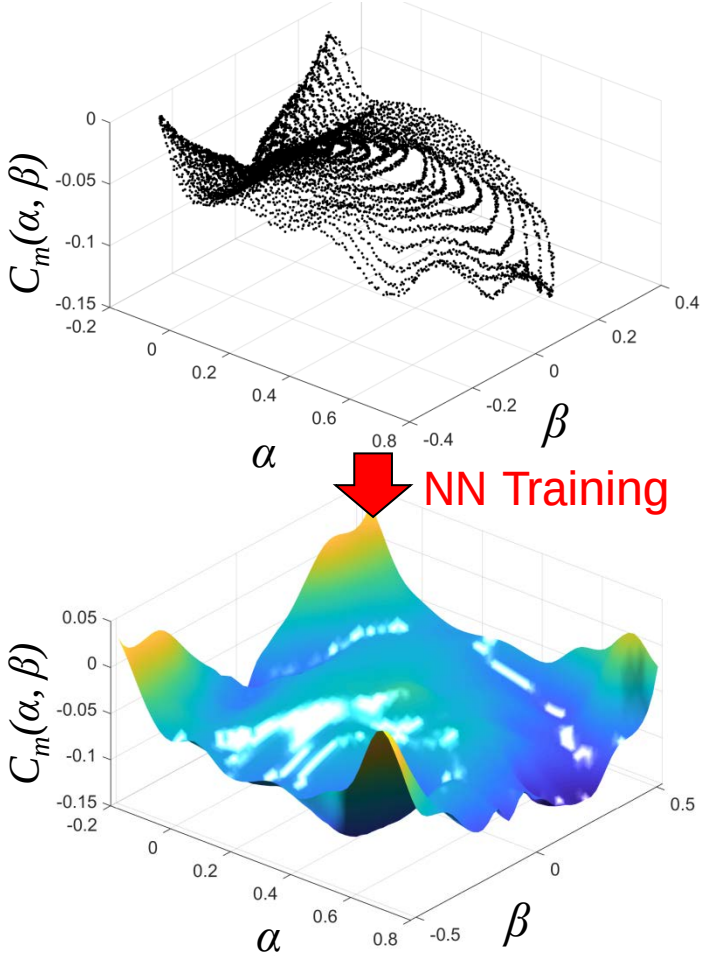
- Several basic training algorithms explained.
- Error-back propagation algorithms most commonly used.
- Error-back propagation using Levenberg-Marquardt method has superior convergence and efficiency.
- Fast convergence & aggressive steps (small $\mu$ ) vs slower convergence & gentle steps (large $\mu$).
- Initialization has a large influence on the final performance.
- High performance hardware accelerated (open-source) software for training NNs exists, e.g. Tensorflow (https://www.tensorflow.org/).

But where to start when defining your own neural network?

# Model Identification with Neural Networks

- Identification relates outputs to inputs through with a model.
- This is not different for neural networks!



# Defining your Network

**Start-off point:**

- Input and output are set
- Activation functions are set
- Static IO mapping
- IO data available

**Question**

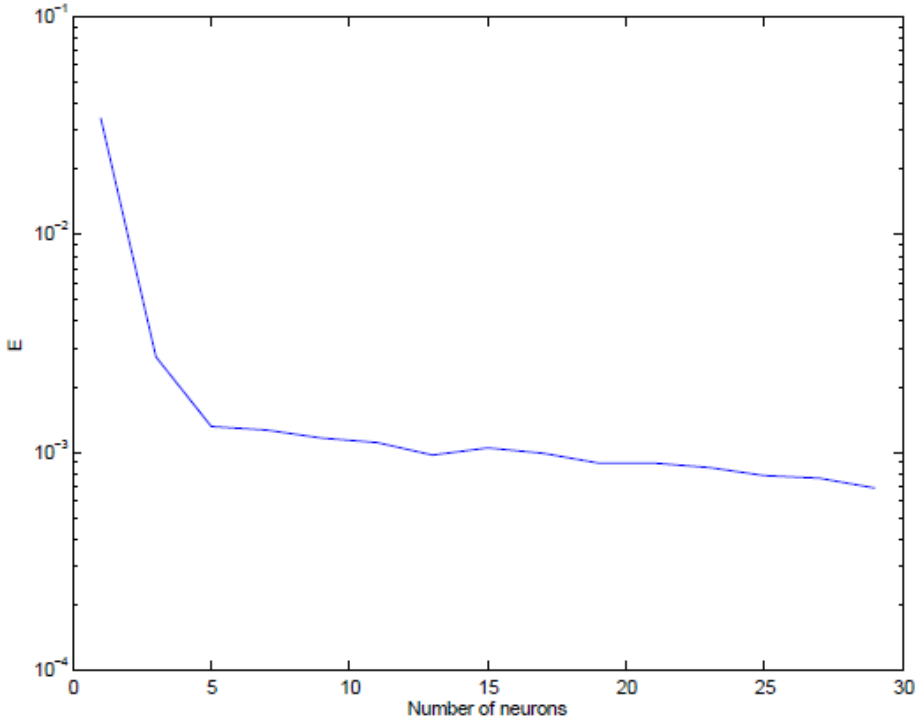
How many neurons do you need for optimal performance?

**Approach:** Iteration based on performance

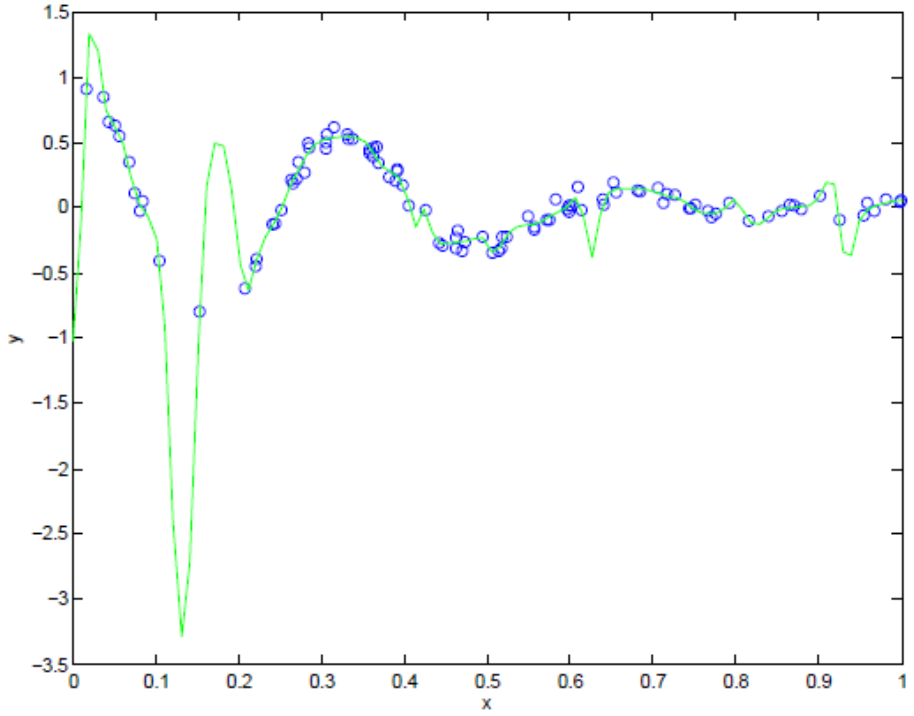
- Perform training phase for a given number of neurons and several weight initializations.
- Increase the number of neurons and if the performance increase is large enough: add more neurons.

Pitfalls: Overfitting, loss of generalization...

# Defining your Network




(a) Performance

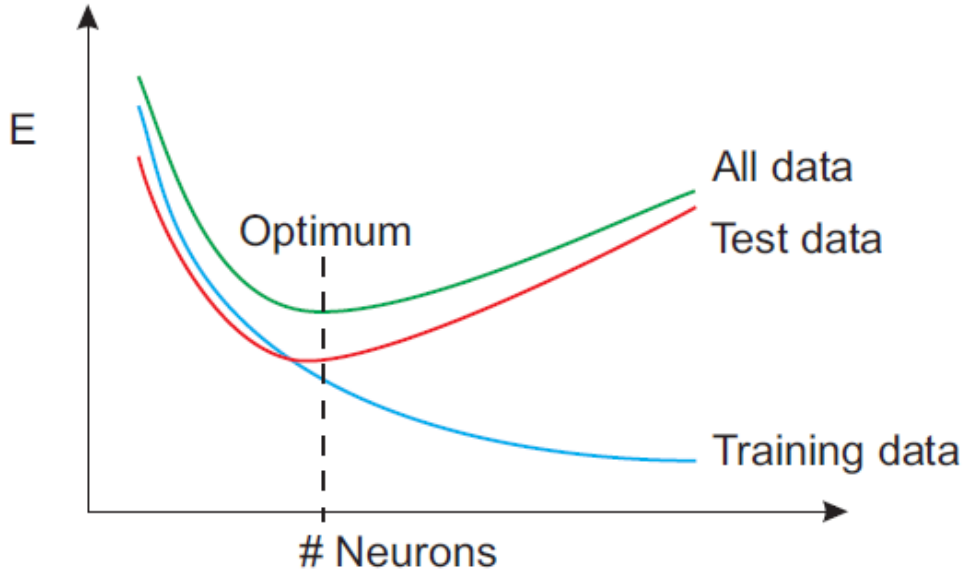



(b) Output for 30 hidden neurons

Figure: Example of the over-fitting problem

# Defining your Network

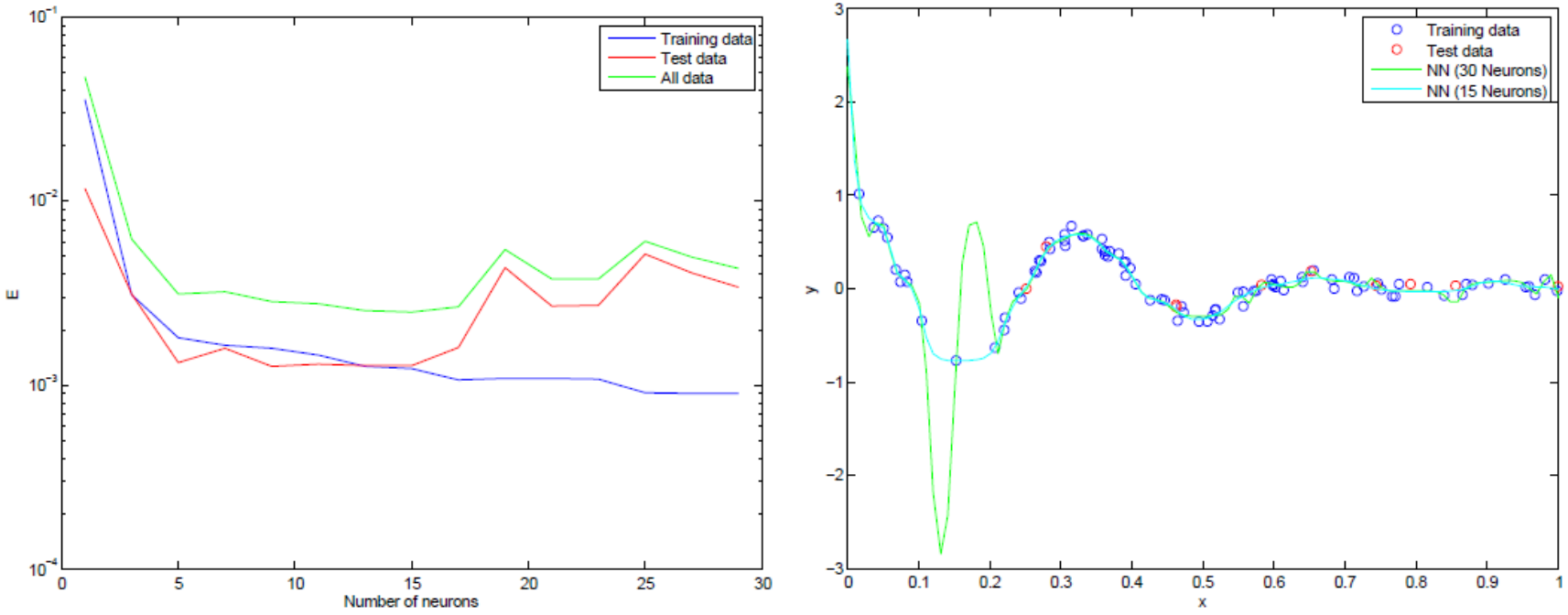
- Common approach to prevent overfitting: use training and test data
- Training algorithm stops if cost function value, using both data sets, is non-decreasing!




- Common ratios between training and test data: 90/10, 80/20.
- The test data will pose an upper limit on the number of neurons!

# Defining your Network

90% training data, 10% test data, per network 10 initializations:




(a) Performance (b) Output for two networks

Figure: Example of using test data to prevent over-fitting

# Defining your Network: Conclusion

- For RBF networks there are more possibilities with optimization since the effects per neuron are local.
- A lot of pruning and allocation methods exist.
- Principle of training/test data still applies.
- Routines are available in the MATLAB neural network toolbox, or open-source packages such as Tensorflow (https://www.tensorflow.org/)

# Neural Networks

## AE4320 System Identification of Aerospace Vehicles

Dr.ir. Coen de Visser
Department of Control & Simulation