Automazione (Laboratorio)

***

Tecniche di Controllo

Reti Neurali e Modelli Fuzzy per L'identificazione, Predizione E Controllo

SILVIO SIMANI

silvio.simani@unife.it

References

Textbook (*suggested*):

• *Neural Networks for Identification, Prediction, and Control*, by Duc Truong Pham and Xing Liu. Springer Verlag; (December 1995). ISBN: 3540199594

• *Nonlinear Identification and Control: A Neural Network Approach*, by G. P. Liu. Springer Verlag; (October 2001). ISBN: 1852333421.

• *Fuzzy Modeling for Control*, by Robert Babuska. Springer; 1st edition (May 1, 1998) ISBN-10: 0792381548, ISBN-13: 978-0792381549.

• *Multi-Objective Optimization using Evolutionary Algorithms*, by Deb Kalyanmoy. John Wiley & Sons, Ltd, Chichester, England, 2001.

Course Overview

1. Introduction
   i. Course introduction
   ii. Introduction to neural network
   iii. Issues in neural network
2. Simple neural network
   i. Perceptron
   ii. Adaline
3. Multilayer Perceptron
   i. Basics
4. Genetic Algorithms: overview
5. Radial basis networks: overview
6. Fuzzy Systems: overview
7. Application examples

Machine Learning

- Improve automatically with experience
- Imitating human learning
  - Human learning
    Fast recognition and classification of complex classes of objects and concepts and fast adaptation
  - Example: neural networks
- Some techniques assume statistical source
  Select a statistical model to model the source
- Other techniques are based on reasoning or inductive inference (e.G. Decision tree)

Machine Learning Definition

A computer program is said to **learn** from *experience* **E** with respect to some class of *tasks* **T** and *performance measure* **P**, if its performance at tasks in **T**, as measured by **P**, improves with experience.

Lecture Notes on Neural Networks and Fuzzy Systems *Silvio Simani*

Examples of Learning Problems

Example 1: handwriting recognition:

- T: recognizing and classifying handwritten words within images.
- P: percentage of words correctly classified.
- E: a database of handwritten words with given classification.

Example 2: learn to play checkers:

- T: play checkers.
- P: percentage of games won in a tournament.
- E: opportunity to play against itself (war games…).

Issues in *Machine Learning*

- What algorithms can approximate functions well and when?
- How does the number of training examples influence accuracy?
- How does the complexity of hypothesis representation impact it?
- How does noisy data influence accuracy?
- *How do you reduce a learning problem to a set of function approximation* ?

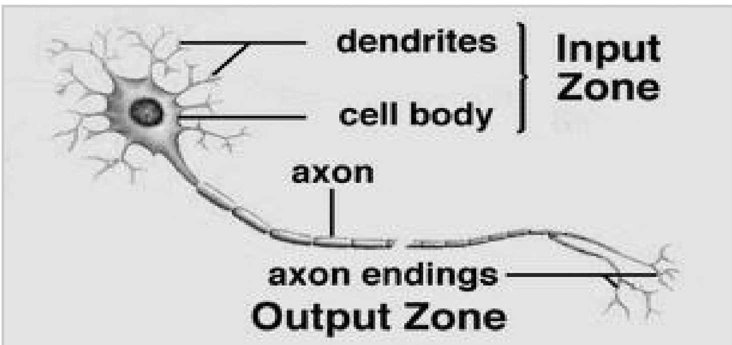
Summary

- *Machine learning* is useful for data mining, poorly understood domain (face recognition) and programs that must dynamically adapt.
- Draws from many diverse disciplines.
- Learning problem needs well-specified task, performance metric and training experience.
- Involve searching space of possible hypotheses. Different learning methods search different hypothesis space, such as numerical functions, *neural networks*, decision trees, symbolic rules.

Introduction to Neural Networks

Brain

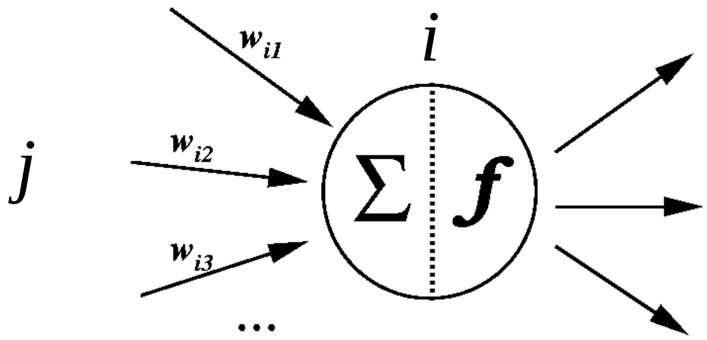
- 10^{11} neurons (processors)
- On average 1000-10000 connections



Artificial Neuron

$$net_i = \sum_j w_{ij} y_j + b$$

bias



$$y_i = f(net_i)$$

Artificial Neuron

- Input/Output Signal may be.
  - Real value.
  - Unipolar {0, 1}.
  - Bipolar {-1, +1}.
- Weight : w_{ij} – strength of connection.

Note that w_{ij} refers to the weight from **unit *j* to unit *i*** (not the other way round).

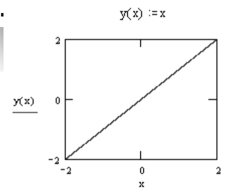
Artificial Neuron

- The bias b is a constant that can be written as $w_{i0}y_0$ with $y_0 = b$ and $w_{i0} = 1$ such that

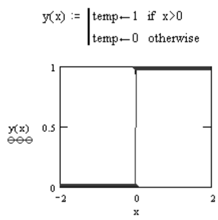
$$net_i = \sum_{j=0}^{n} w_{ij} y_j$$

- The function f is the unit's activation function. In the simplest case, f is the identity function, and the unit's output is just its net input. This is called a *linear unit*.
- Other activation functions are : step function, sigmoid function and Gaussian function.

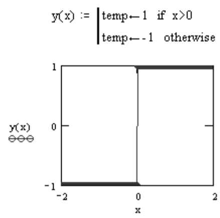
Activation Functions



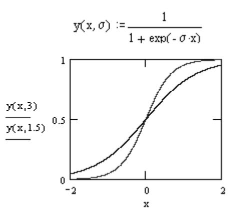
Identity function



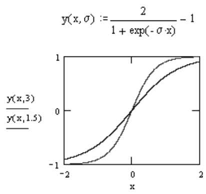
Binary Step function



Bipolar Step function

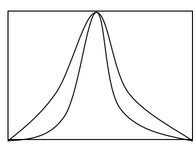


Sigmoid function



Bipolar Sigmoid function

$$y(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$



Gaussian function

When Should ANN Solution Be Considered ?

- The solution to the problem cannot be explicitly described by an algorithm, a set of equations, or a set of rules.
- There is some evidence that an input-output mapping exists between a set of input and output variables.
- There should be a large amount of data available to train the network.

Problems That Can Lead to Poor Performance ?

- The network has to distinguish between very similar cases with a very high degree of accuracy.
- The train data does not represent the ranges of cases that the network will encounter in practice.
- The network has a several hundred inputs.
- The main discriminating factors are not present in the available data, *e.g.* trying to assess the loan application without having knowledge of the applicant's salaries.
- The network is required to implement a very complex function.

Applications of Artificial Neural Networks

- Manufacturing : fault diagnosis, fraud detection.
- Retailing : fraud detection, forecasting, data mining.
- Finance : fraud detection, forecasting, data mining.
- Engineering : fault diagnosis, signal/image processing.
- Production : fault diagnosis, forecasting.
- Sales & marketing : forecasting, data mining.

Data Pre-processing

Neural networks very **rarely** operate on the raw data. An initial **pre-processing** stage is essential. Some examples are as follows:

- Feature extraction of images: for example, the analysis of x-rays requires pre-processing to extract features which may be of interest within a specified region.
- Representing input variables with numbers. For example "+1" is the person is married, "0" if divorced, and "-1" if single. Another example is representing the pixels of an image: 255 = bright white, 0 = black. To ensure the generalization capability of a neural network, the data should be encoded in form which allows for interpolation.

Data Pre-processing

- **CONTINUOUS VARIABLES**
  - A continuous variable can be directly applied to a neural network. However, if the dynamic range of input variables are not approximately the same, it is better to *normalize* all input variables of the neural network.

Simple Neural Networks

Simple Perceptron

Outlines

- The Perceptron
- Linearly separable problem
- Network structure
- Perceptron learning rule
- Convergence of Perceptron

THE PERCEPTRON

- The perceptron was a simple model of ANN introduced by Rosenblatt of MIT in the 1960' with the idea of learning.

- Perceptron is designed to accomplish a simple pattern recognition task: after learning with real value training data

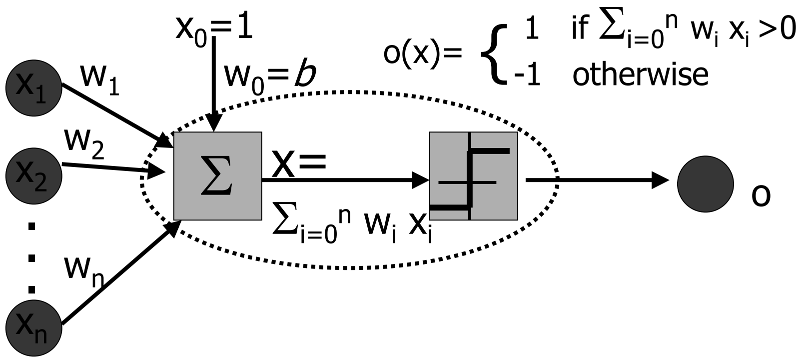
  $\{\underline{x(i)}, d(i),\ i = 1,2,\ldots,p\}$ *where* $d(i) = 1$ *or* -1

- For a new signal (pattern) $\underline{x(i+1)}$, the perceptron is capable of telling you to which class the new signal belongs

$$\underline{x(i+1)} \longrightarrow \boxed{\text{perceptron}} = 1 \text{ or } -1$$

Perceptron

- Linear Threshold Unit (LTU)



Mathematically the Perceptron is

$$\boxed{y = f\left(\sum_{i=1}^{m} w_i x_i + b\right) = f\left(\sum_{i=0}^{m} w_i x_i\right)}$$

We can always treat the bias *b* as another weight with inputs equal 1

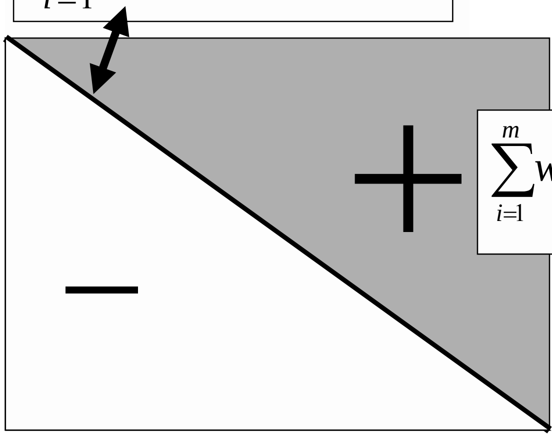
where f is the hard limiter function i.e.

$$y = \begin{cases} 1 \ if \ \sum_{i=1}^{m} w_i x_i + b > 0 \\ -1 \ if \ \sum_{i=1}^{m} w_i x_i + b \le 0 \end{cases}$$

Why is the network capable of solving linearly separable problem ?

$$\sum_{i=1}^{m} w_i x_i + b = 0$$

$$\sum_{i=1}^{m} w_i x_i + b < 0$$

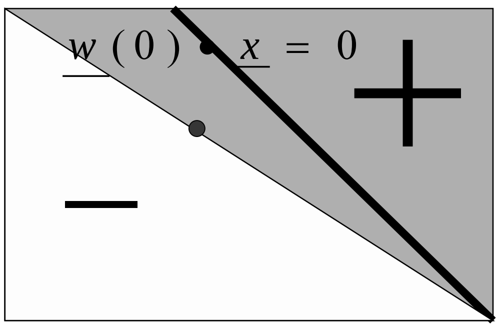


$$\sum_{i=1}^{m} w_i x_i + b > 0$$

Learning rule

An algorithm to update the weights *w* so that finally the input patterns lie on both sides of the line decided by the perceptron

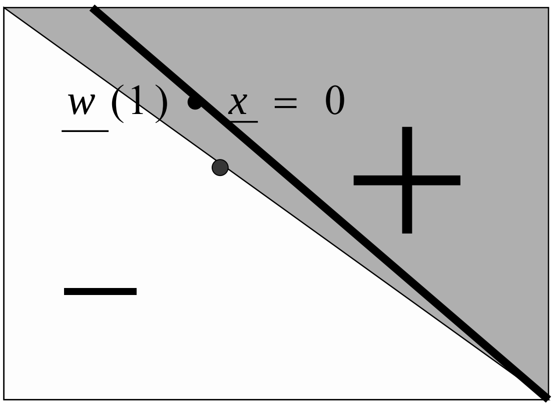
Let *t* be the time, at *t = 0, we have*



Learning rule

An algorithm to update the weights $\underline{w}$ so that finally the input patterns lie on both sides of the line decided by the perceptron

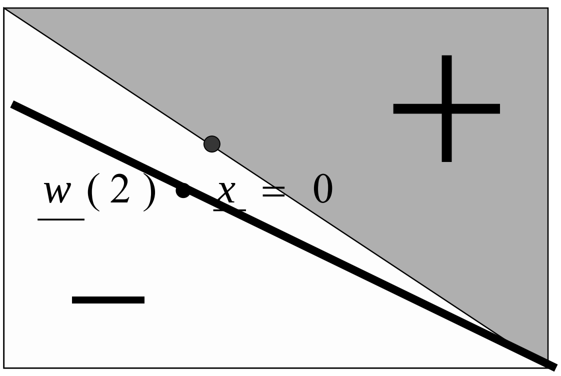
Let t be the time, at $t = 1$



Learning rule

An algorithm to update the weights $\underline{w}$ so that finally the input patterns lie on both sides of the line decided by the perceptron

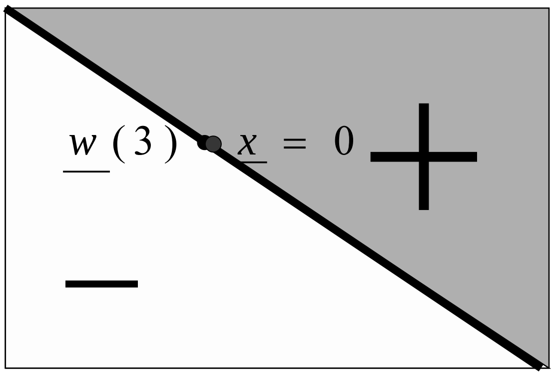
Let t be the time, at $t = 2$



Learning rule

An algorithm to update the weights $\underline{w}$ so that finally the input patterns lie on both sides of the line decided by the perceptron

Let *t* be the time, at *t = 3*



In Math

$$d(t) = \begin{cases} +1 \; if \;\; x(t) \; in \;\; class \quad + \\ -1 \; if \;\; x(t) \; in \;\; class \quad - \end{cases}$$

Perceptron learning rule

$$\underline{w}(t+1) = \underline{w}(t) + \eta(t)[d(t) - sign(\underline{w}(t) \bullet \underline{x}(t))] \, \underline{x}(t)$$

Where $\eta(t)$ is the learning rate >0,

$$sign(x) = \begin{cases} +1 \text{ if } x>0 \\ -1 \text{ if } x<=0, \end{cases}$$ hard limiter function

<u>NB : d(t) is the same as d(i) and x(t) as x(i)</u>

In words:

• If the classification is right, do not update the weights

• If the classification is not correct, update the weight towards the opposite direction so that the output move close to the right directions.

Perceptron convergence theorem (Rosenblatt, 1962)

Let the subsets of training vectors be linearly separable. Then after finite steps of learning we have

lim $\underline{w}(t) = \underline{w}$ which correctly separate the samples.

The idea of proof is that to consider $||\underline{w}(t+1)-\underline{w}||-||\underline{w}(t)-\underline{w}||$ which is a decrease function of t

Summary of Perceptron learning ...

Variables and parameters

$\underline{x}(t) = (m+1)$ *dim.* input vectors at time t
$= (b, x_1(t), x_2(t), \ldots , x_m(t))$

$\underline{w}(t) = (m+1)$ *dim.* weight vectors
$= (1 , w_1(t), \ldots , w_m(t))$

$b =$ bias
$y(t) =$ actual response
$\eta(t)$ = learning rate parameter, a +ve constant < 1
$d(t) =$ desired response

Summary of Perceptron learning ...

Data $\{ (\underline{x}(i), d(i)), i=1,\ldots,p\}$

✓ Present the data to the network once a point

✓ could be cyclic :
$(\underline{x}(1), d(1)), (\underline{x}(2), d(2)),\ldots, (\underline{x}(p), d(p)),$
$(\underline{x}(p+1), d(p+1)),\ldots$

✓ *or randomly*

(Hence we mix time t with i here)

Summary of Perceptron learning (algorithm)

1. Initialisation Set *$\underline{w}(0)=0$*. Then perform the following computation for time step t=1,2,...

2. Activation At time step t, activate the perceptron by applying input vector *$\underline{x}(t)$* and desired response *$d(t)$*

3. Computation of actual response Compute the actual response of the perceptron

$$y(t) = sign\ (\underline{w}(t) \cdot \underline{x}(t))$$

where ***sign*** is the sign function

4. Adaptation of weight vector Update the weight vector of the perceptron

$$\underline{w}(t+1) = \underline{w}(t) + \eta(t)\ [\ d(t) - y(t)\ ]\ \underline{x}(t)$$

5. Continuation

Questions remain

Where or when to stop?

By minimizing the generalization error

For training data $\{(\underline{x}(i), d(i)),\ i=1,\dots p\}$

How to define training error after t steps of learning?

$$E(t) = \sum_{i=1}^{p} [d(i) - sign(\underline{w}(t) \cdot \underline{x}(i)]^2$$

We next turn to ADALINE learning, from which we can understand the learning rule, and more general the Back-Propagation (BP) learning

Simple Neural Network

ADALINE Learning

Outlines

- ADALINE
- Gradient descending learning
- Modes of training

Unhappy Over Perceptron Training

- When a perceptron gives the right answer, no learning takes place
- Anything below the threshold is interpreted as 'no', even it is just below the threshold.
- It might be better to train the neuron based on how far below the threshold it is.

ADALINE

•ADALINE is an acronym for ADAptive LINear Element (or ADAptive LInear NEuron) developed by Bernard Widrow and Marcian Hoff (1960).

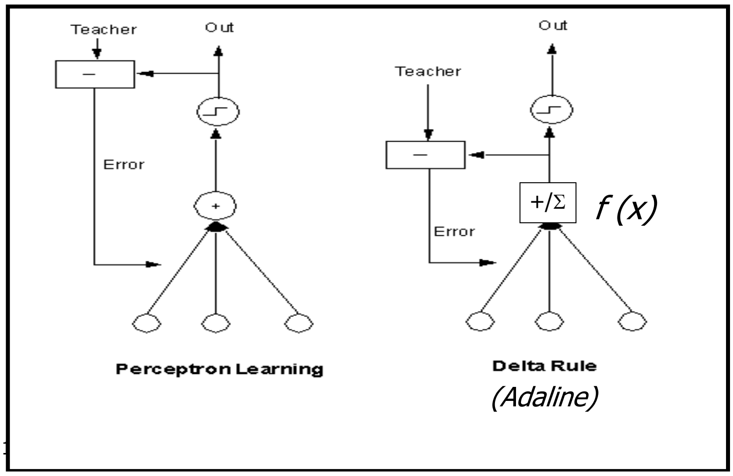
• There are several variations of Adaline. One has threshold same as perceptron and another just a bare linear function.

•The Adaline learning rule is also known as the least-mean-squares (LMS) rule, the delta rule, or the Widrow-Hoff rule.

• It is a training rule that minimizes the output error using (approximate) gradient descent method.

• Replace the step function in the perceptron with a continuous (differentiable) function *f*, e.g the simplest is linear function

• With or without the threshold, the Adaline is trained based on the output of the function *f* rather than the final output.



After each training pattern $\underline{x}(i)$ is presented, the correction to apply to the weights is proportional to the error.

$$E(i,t) = \tfrac{1}{2} [d(i) - f(\underline{w}(t) \cdot \underline{x}(i))]^2 \qquad i=1,\ldots,p$$

N.B. If f is a linear function $f(\underline{w}(t) \cdot \underline{x}(i)) = \underline{w}(t) \cdot \underline{x}(i)$

Summing together, our purpose is to find $\underline{W}$ which minimizes

$$E(t) = \sum_i E(i,t)$$

General Approach gradient descent method

To find g

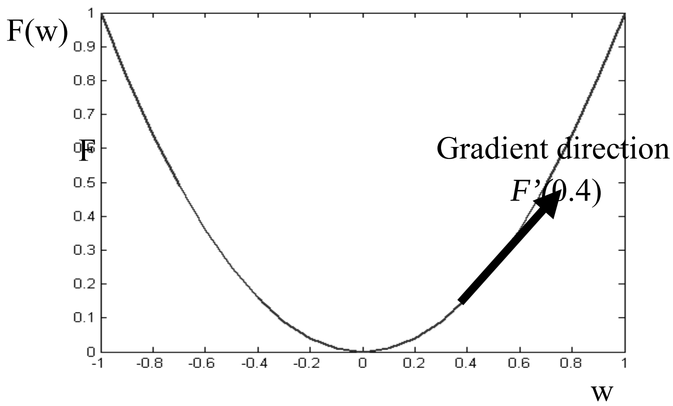
$$\underline{w}(t+1) = \underline{w}(t) + g(E(\underline{w}(t)))$$

so that $\underline{w}$ automatically tends to the global minimum of E(w).

$$\underline{w}(t+1) = \underline{w}(t) - E'(\underline{w}(t))\eta(t)$$

(see figure in the following...)

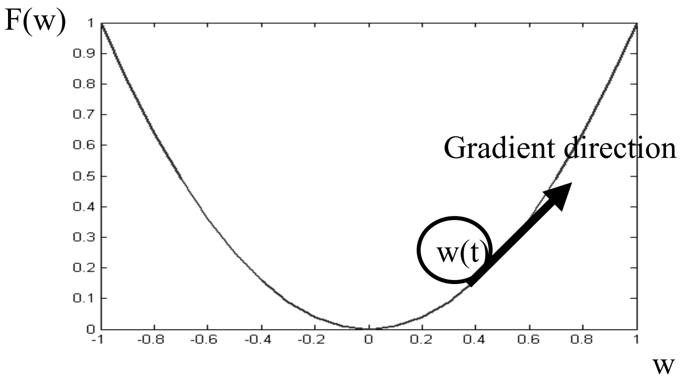
- Gradient direction is the direction of uphill for example, in the Figure, at position 0.4, the gradient is uphill (F is E, consider one dim case)



- In gradient descent algorithm, we have

$$\underline{w}(t+1) = \underline{w}(t) - F'(w(t))\,\eta(\tau)$$

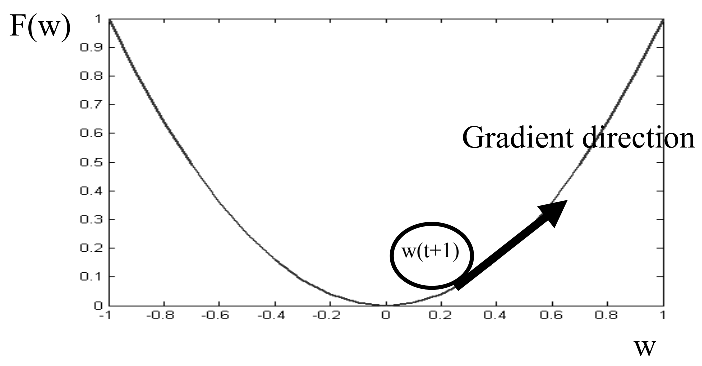
therefore the ball goes downhill since – F'(w(t)) is downhill direction



- In gradient descent algorithm, we have

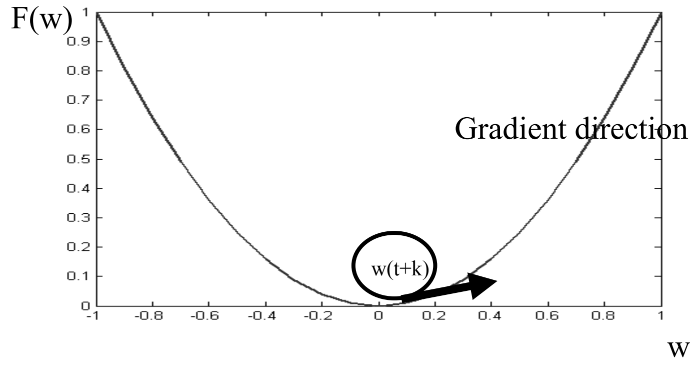
$$w(t+1) = w(t) - F'(w(t))\,\eta(\tau)$$

therefore the ball goes downhill since – F'(w(t)) is downhill direction



Lecture Notes on Neural Networks and Fuzzy Systems *Silvio Simani*

- Gradually the ball will stop at a local minima where the gradient is zero



• In words

Gradient method could be thought of as a ball rolling down from a hill: the ball will roll down and finally stop at the valley

Thus, the weights are adjusted by

$$w_j(t+1) = w_j(t) + \eta(t) \Sigma \ [d(i) - f(\underline{w}(t) \cdot \underline{x}(i))] \, x_j(i) \, f'$$

This corresponds to gradient descent on the quadratic error surface E

<u>When f′ =1, we have the perceptron learning rule (we have in general f′>0 in neural networks). The ball moves in the right direction.</u>

Two types of network training:

Sequential mode (on-line, stochastic, or per-pattern) :
Weights updated after each pattern is presented (Perceptron is in this class)

Batch mode (off-line or per-epoch) :
Weights updated after all patterns are presented

Comparison Perceptron and Gradient Descent Rules

- Perceptron learning rule guaranteed to succeed if
  - Training examples are linearly separable
  - Sufficiently small learning rate η
- Linear unit training rule uses gradient descent guaranteed to converge to hypothesis with minimum squared error given sufficiently small learning rate η
  - Even when training data contains noise
  - Even when training data not separable by hyperplanes

Summary

Perceptron

$\underline{W}(t+1) = \underline{W}(t) + \eta(t)\,[\,d(t) - \text{sign}\,(\underline{w}(t)\,.\,\underline{x})]\,\underline{x}$

Adaline (Gradient descent method)

$\underline{W}(t+1) = \underline{W}(t) + \eta(t)\,[\,d(t) - f(\underline{w}(t)\,.\,\underline{x})]\,\underline{x}\,f'$

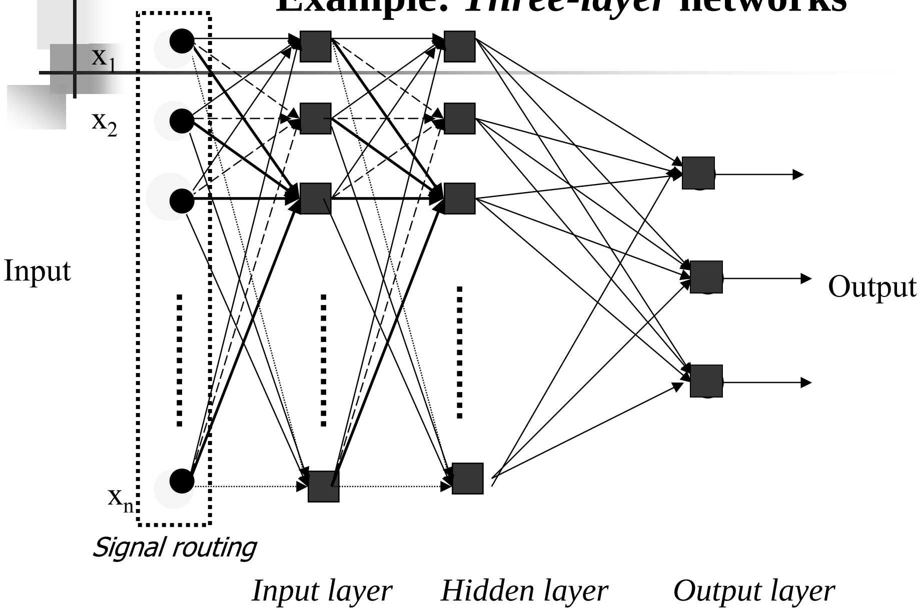
Multi-Layer Perceptron (MLP)

Idea: "Credit assignment problem"

• Problem of assigning 'credit' or 'blame' to individual elements involving in forming overall response of a learning system (hidden units)

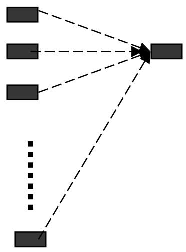
• In neural networks, problem relates to dividing which weights should be altered, by how much and in which direction.

Example: *Three-layer* networks



Properties of architecture

- No connections within a layer
- No direct connections between input and output layers
- Fully connected between layers
- Often more than 2 layers
- Number of output units need not equal number of input units
- Number of hidden units per layer can be more or less than input or output units



Each unit ‘▬’ is a perceptron

$$y_i = f\left(\sum_{j=1}^{m} w_{ij} x_j + b_i\right)$$

BP (Back Propagation)

gradient descent method

+

multilayer networks

MultiLayer Perceptron I

Back Propagating
Learning

BP learning algorithm
Solution to “credit assignment problem” in MLP

Rumelhart, Hinton and Williams (1986)

BP has two phases:

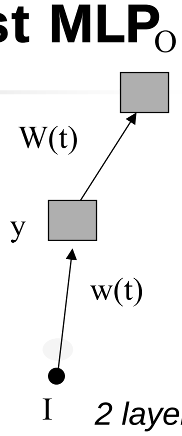
Forward pass phase: computes ‘functional signal’, feedforward propagation of input pattern signals through network

Backward pass phase: computes ‘error signal’, propagation of error (difference between actual and desired output values) backwards through network starting at output units

BP Learning for Simplest MLP

Task : Data {I, d} to minimize

$$E = (d - o)^2 /2$$
$$= [d - f(W(t)y(t))]^2 /2$$
$$= [d - f(W(t)f(w(t)I))]^2 /2$$



2 layers example

Error function at the output unit

Weight at time t is w(t) and W(t), intend to find the weight w and W at time t+1

Where y = f(w(t)I), output of the input unit

Forward pass phase

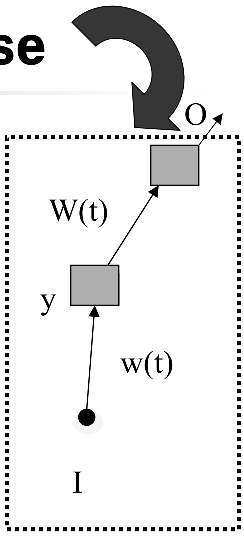
Suppose that we have w(t), W(t) of time t

For given input I, we can calculate

y = f(w(t)I)

and

o = f (W(t) y)
= f (W(t) f(w(t) I))



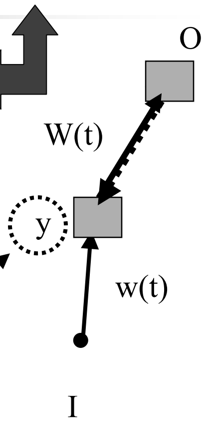
Error function of output unit will be

E = $(d - o)^2 /2$

2 layers example

Backward Pass Phase

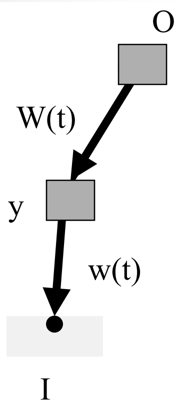
$$
\begin{aligned}
W(t+1) &= W(t) - \eta \frac{dE}{dW(t)} \\
&= W(t) - \eta \frac{dE}{df} \frac{df}{dW(t)} \\
&= W(t) + \eta (d - o) f'(W(t) y) y
\end{aligned}
$$



$E = (d - o)^2 / 2$ $\quad$ $o = f(W(t)\,y)$

Backward pass phase

$$
\begin{aligned}
W(t+1) &= W(t) - \eta \frac{dE}{dW(t)} \\
&= W(t) - \eta \frac{dE}{df} \frac{df}{dW(t)} \\
&= W(t) + \eta (d - o) f'(W(t) y) y \\
&= W(t) + \eta \Delta y
\end{aligned}
$$



where $\Delta = (d - o) f'$

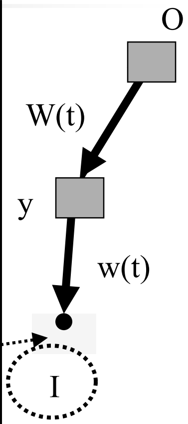
Backward pass phase

$$w(t+1) = w(t) - \eta \frac{dE}{dw(t)}$$

$$= w(t) - \eta \frac{dE}{dy}\frac{dy}{dw(t)}$$

$$= w(t) + \eta (d - o) f'(W(t)y)W(t)\frac{dy}{dw(t)}$$

$$= w(t) + \eta \Delta W(t) f'(w(t)I)I$$



o = f (W(t) y)

= f (W(t) f(w(t) I))

Summary

weight updates are local

$$w_{ji}(t+1) - w_{ji}(t) = \eta \delta_j(t) I_i(t)$$ (input unit)

$$W_{kj}(t+1) - W_{kj}(t) = \eta \Delta_k(t) y_j(t)$$ (output unit)

output unit

$$W_{kj}(t+1) - W_{kj}(t) = \eta \Delta_k(t) y_j(t)$$
$$= \eta (d_k(t) - O_k(t)) f'(Net_k(t)) y_j(t)$$

input unit

$$w_{ji}(t+1) - w_{ji}(t) = \eta \delta_j(t) I_i(t)$$
$$= \eta f'(net_j(t)) \sum_k \Delta_k(t) W_{kj} I_i(t)$$

Once weight changes are computed for all units, weights are updated at same time (bias included as weights here)

We now compute the derivative of the activation function *f*( ).

Activation Functions

- to compute δ_j and Δ_k we need to find the derivative of activation function *f*
- to find derivative the activation function must be smooth

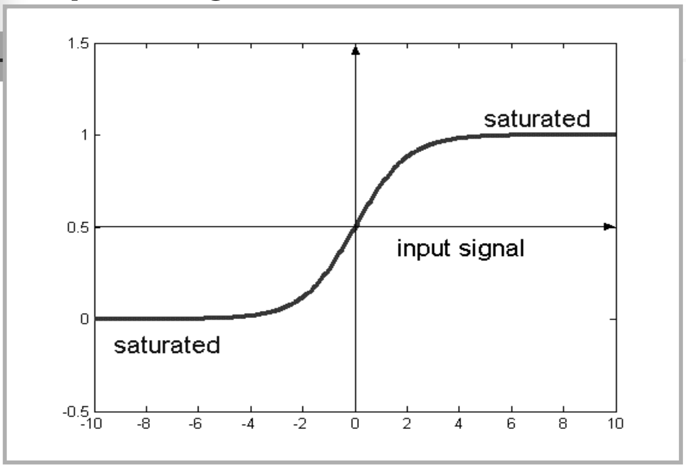
Sigmoidal (logistic) function-common in MLP

$$f\ (net_i(t)) = \frac{1}{1+\exp(-k\,net_i(t))}$$

where k is a positive constant. The sigmoidal function gives value in range of 0 to 1

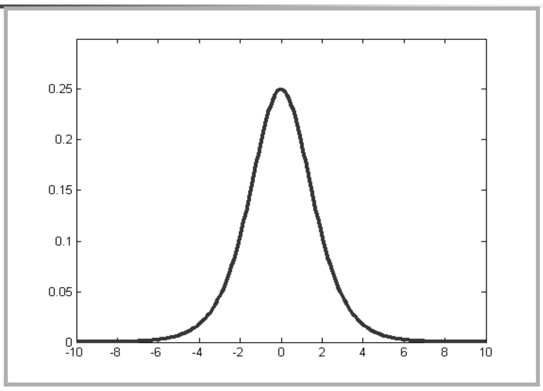
Input-output function of a neuron (rate coding assumption)

Shape of sigmoidal function



Note: when net = 0, f = 0.5

Shape of sigmoidal function derivative



Derivative of sigmoidal function has max at x= 0, is symmetric about this point falling to zero as sigmoidal approaches extreme values

Returning to **local error gradients** in BP algorithm we have for output units

$$\begin{aligned}\Delta_i(t) &= (d_i(t) - O_i(t))\, f'(Net_i(t)) \\ &= (d_i(t) - O_i(t))\, kO_i(t)(1 - O_i(t))\end{aligned}$$

For input units we have

$$\begin{aligned}\delta_i(t) &= f'(net_i(t)) \sum_k \Delta_k(t) W_{ki} \\ &= ky_i(t)(1 - y_i(t)) \sum_k \Delta_k(t) W_{ki}\end{aligned}$$

Since degree of weight change is proportional to derivative of activation function, weight changes will be greatest when units receives mid-range functional signal than at extremes

Network training:

- ❖ Training set shown repeatedly until stopping criteria are met
- ❖ Each full presentation of all patterns = 'epoch'
- ❖ Randomise order of training patterns presented for each epoch in order to avoid correlation between consecutive training pairs being learnt (order effects)

Two types of network training:

- ➢ **Sequential mode** (on-line, stochastic, or per-pattern) Weights updated after each pattern is presented
- ➢ **Batch mode** (off-line or per -epoch)

Advantages and disadvantages of different modes

Sequential mode:

- Less storage for each weighted connection
- Random order of presentation and updating per pattern means search of weight space is stochastic-reducing risk of local minima able to take advantage of any redundancy in training set (*i.e.* same pattern occurs more than once in training set, esp. for large training sets)
- Simpler to implement

Batch mode:

- Faster learning than sequential mode

MultiLayer Perceptron II

Dynamics of MultiLayer Perceptron

Summary of Network Training

Forward phase: $\underline{I}(t), \underline{w}(t), \underline{net}(t), \underline{y}(t), \underline{W}(t), \underline{Net}(t), \underline{O}(t)$

Backward phase:

Output unit

$$W_{kj}(t+1) - W_{kj}(t) = \eta \Delta_k(t) y_j(t)$$
$$= \eta (d_k(t) - O_k(t)) f'(Net_k(t)) y_j(t)$$

Input unit

$$w_{ji}(t+1) - w_{ij}(t) = \eta \delta_j(t) I_i(t)$$
$$= \eta f'(net_j(t)) \sum_k \Delta_k(t) W_{kj}(t) I_i(t)$$

Network training:

Training set shown repeatedly until stopping criteria are met. Possible convergence criteria are

- Euclidean norm of the gradient vector reaches a sufficiently small denoted as θ.
- When the absolute rate of change in the average squared error per epoch is sufficiently small denoted as θ.
- Validation for generalization performance : stop when generalization reaching the peak (illustrate in this lecture)

Goals of Neural Network Training

To give the correct output for input training vector (Learning)

To give good responses to new unseen input patterns (Generalization)

Training and Testing Problems

- **Stuck neurons**: Degree of weight change is proportional to derivative of activation function, weight changes will be greatest when units receives mid-range functional signal than at extremes neuron. To avoid stuck neurons weights initialization should give outputs of all neurons approximate 0.5
- **Insufficient number of training patterns**: In this case, the training patterns will be learnt instead of the underlying relationship between inputs and output, i.e. network just memorizing the patterns.
- **Too few hidden neurons**: network will not produce a good model of the problem.
- **Over-fitting**: the training patterns will be learnt instead of the underlying function between inputs and output because of too many of hidden neurons. This means that the network will have a poor generalization capability.

Dynamics of BP learning

Aim is to minimise an error function over all training patterns by adapting weights in MLP

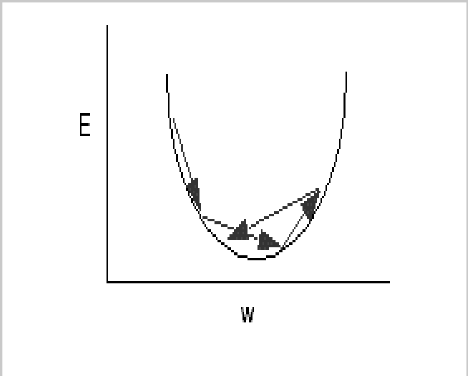
Recalling the typical error function is the mean squared error as follows

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

The idea is to reduce E(t) to global minimum point.

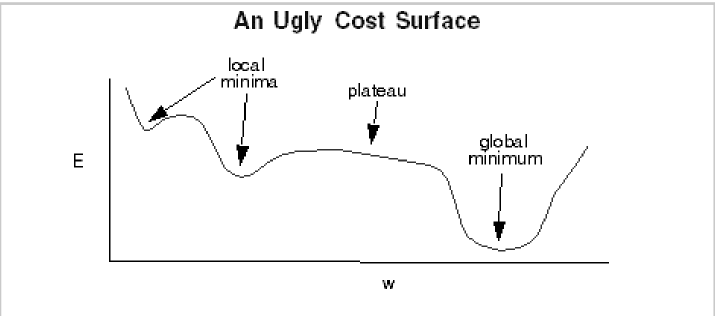
Dynamics of BP learning

In single layer perceptron with linear activation functions, the error function is simple, described by a smooth parabolic surface with a single minimum



Dynamics of BP learning

MLP with non-linear activation functions have complex error surfaces (e.g. plateaus, long valleys etc.) with no single minimum



For complex error surfaces the problem is learning rate must keep small to prevent divergence. Adding momentum term is a simple approach dealing with this problem.

Momentum

• Reducing problems of instability while increasing the rate of convergence
• Adding term to weight update equation can effectively holds as exponentially weight history of previous weights changed

Modified weight update equation is

$$w_{ij}(n+1) - w_{ij}(n) = \eta \delta_j(n) y_i(n) + \\ + \alpha [w_{ij}(n) - w_{ij}(n-1)]$$

Effect of momentum term

➢ If weight changes tend to have same sign, momentum term increases and gradient decrease speed up convergence on shallow gradient
➢ If weight changes tend have opposing signs, momentum term decreases and gradient descent slows to reduce oscillations (stabilizes)
➢ Can help escape being trapped in local minima

Selecting Initial Weight Values

- Choice of initial weight values is important as this decides starting position in weight space. That is, how far away from global minimum
- Aim is to select weight values which produce midrange function signals
- Select weight values randomly from uniform probability distribution
- Normalise weight values so number of weighted connections per unit produces midrange function signal

Convergence of Backprop

Avoid local minumum with fast convergence:

- Add momentum
- Stochastic gradient descent
- Train multiple nets with different initial weights

Nature of convergence

- Initialize weights 'near zero' or initial networks near-linear
- Increasingly non-linear functions possible as training progresses

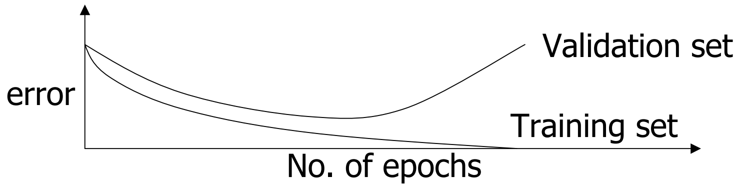
Use of Available Data Set for Training

The available data set is normally split into three sets as follows:

- Training set – use to update the weights. Patterns in this set are repeatedly in random order. The weight update equation are applied after a certain number of patterns.
- Validation set – use to decide when to stop training only by monitoring the error.
- Test set – Use to test the performance of the neural network. It should not be used as part of the neural network development cycle.

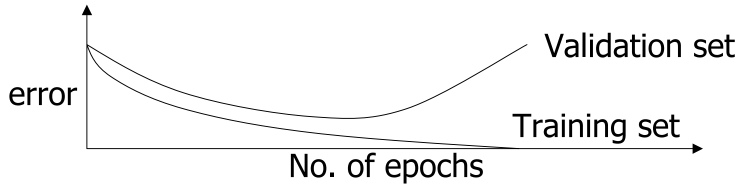
Earlier Stopping - Good Generalization

- Running too many epochs may overtrain the network and result in overfitting and perform poorly in generalization.
- Keep a hold-out validation set and test accuracy after every epoch. Maintain weights for best performing network on the validation set and stop training when error increases increases beyond this.



Model Selection by Cross-validation

- Too few hidden units prevent the network from learning adequately fitting the data and learning the concept (more than two layer networks).
- Too many hidden units leads to overfitting.
- Similar cross-validation methods 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.



Alternative Training Algorithm

Genetic Algorithms

History Background

- Idea of evolutionary computing was introduced in the 1960s by I. **Rechenberg** in his work "***Evolution strategies***" (*Evolutionsstrategie* in original). His idea was then developed by other researchers. **Genetic Algorithms** (GAs) were invented by John **Holland** and developed by him and his students and colleagues. This lead to Holland's book "*Adaption in Natural and Artificial Systems*" published in 1975.
- In 1992 John **Koza** has used genetic algorithm to evolve programs to perform certain tasks. He called his method "**Genetic Programming**" (GP). LISP programs were used, because programs in this language can expressed in the form of a "parse tree", which is the object the GA works on.

Biological Background

Chromosome.

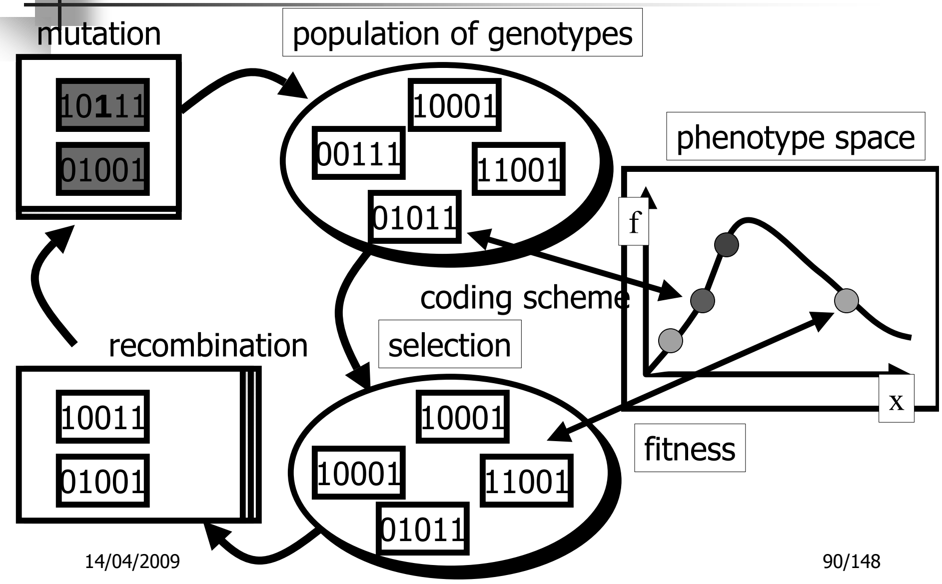
- All living organisms consist of cells. In each cell there is the same set of **chromosomes**. Chromosomes are strings of DNA and serves as a model for the whole organism. A chromosome consist of **genes**, blocks of DNA. Each gene encodes a particular protein. Basically can be said, that each gene encodes a **trait**, for example color of eyes. Possible settings for a trait (e.g. blue, brown) are called **alleles**. Each gene has its own position in the chromosome. This position is called **locus**.
- Complete set of genetic material (all chromosomes) is called **genome**. Particular set of genes in genome is called **genotype**. The genotype is with later development after birth base for the organism's **phenotype**, its physical and mental characteristics, such as eye color, intelligence etc.

Biological Background

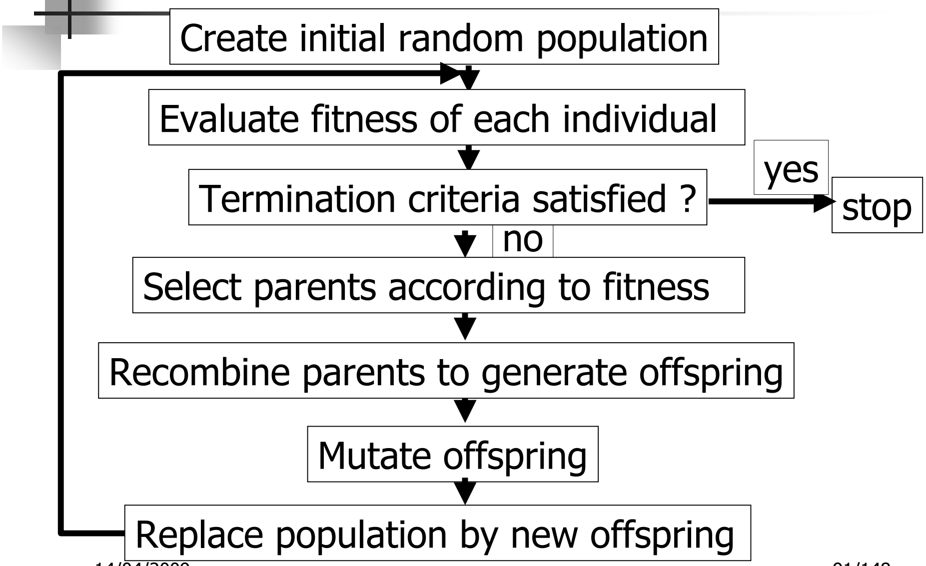
Reproduction.

- During reproduction, first occurs **recombination** (or **crossover**). Genes from parents form in some way the whole new chromosome. The new created offspring can then be mutated. **Mutation** means, that the elements of DNA are a bit changed. This changes are mainly caused by errors in copying genes from parents.
- The **fitness** of an organism is measured by success of the organism in its life.

Evolutionary Algorithms



Pseudo Code of an Evolutionary Algorithm



A Simple Genetic Algorithm

- Optimization task : find the maximum of f(x)
  for example f(x)=x•sin(x) $x \in [0,\pi]$
  - genotype: binary string $s \in [0,1]^5$ e.g. 11010, 01011, 10001
  - mapping : genotype ⇨ phenotype
    binary integer encoding: $x = \pi \bullet \sum_{i=1}^{n=5} s_i \bullet 2^{n-i-1} / (2^n-1)$

Initial population

genotype	integ.	phenotype	fitness	prop. fitness
11010	26	2.6349	1.2787	30%
01011	11	1.1148	1.0008	24%
10001	17	1.7228	1.7029	40%
00101	5	0.5067	0.2459	6%

Radial Basis Functions

Radial Basis Functions
Overview

Radial-basis function (RBF) networks

- RBF = radial-basis function

- a function which depends only on the radial distance from a point

Radial-basis function (RBF) networks

So RBFs are functions taking the form

$$\phi\,(\|\ \underline{x} - \underline{x}_i\ \|)$$

where ϕ is a non-linear activation function, $\underline{x}$ is the input and $\underline{x}_i$ is the *i'th* position, prototype, *basis* or *centre* vector.

The idea is that points near the centres will have similar outputs (i.e. if $\underline{x} \sim \underline{x}i$ then $f(\underline{x}) \sim f(\underline{x}i)$) since they should have similar properties.

The simplest is the linear RBF : $\phi(x) = ||\underline{x} - \underline{x}_i||$

Typical RBFs include

(a) Multi-quadrics

$$\phi(r) = (r^2 + c^2)^{1/2}$$

for some c>0

(b) Inverse multi-quadrics

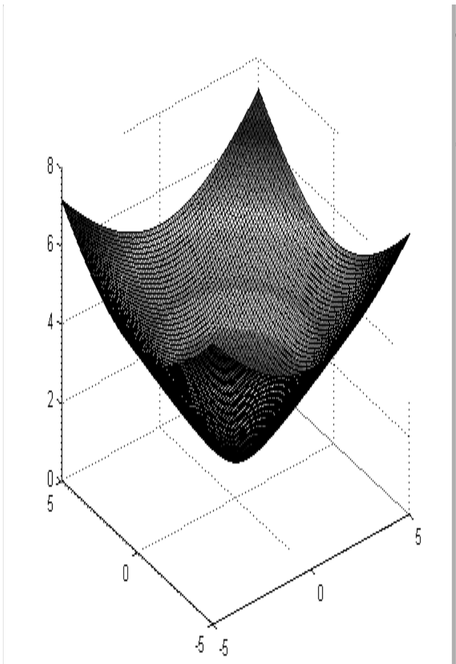
$$\phi(r) = (r^2 + c^2)^{-1/2}$$

for some c>0

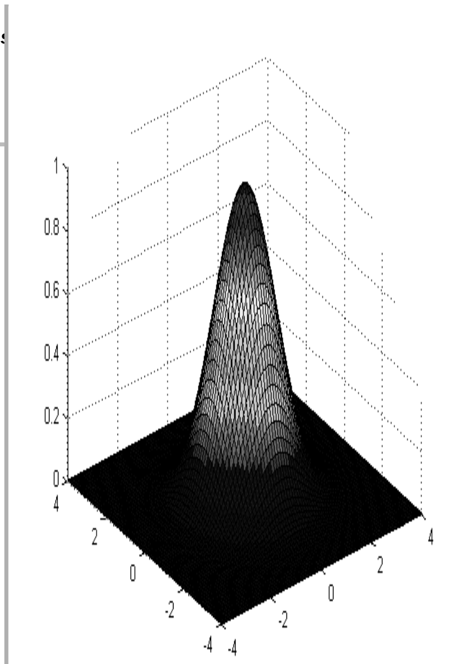
(c) Gaussian

$$\phi(r) = \exp(-\frac{r^2}{2\sigma^2})$$

for some σ >0

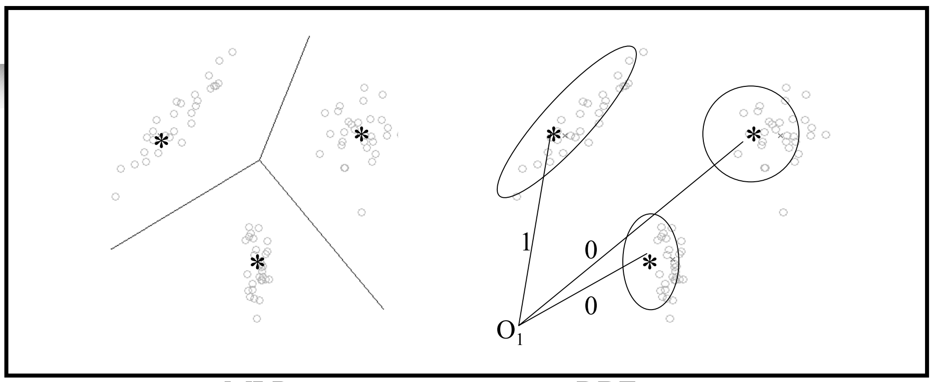


‘nonlocalized’ functions



‘localized’ functions

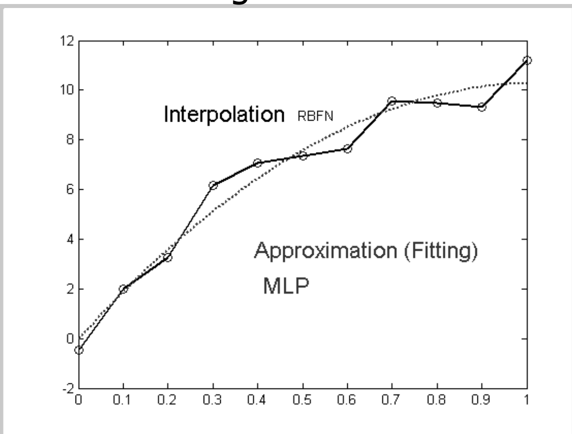
➢ Idea is to use a weighted sum of the outputs from the basis functions to represent the data.
➢ Thus centers can be thought of as prototypes of input data.



MLP vs RBF
distributed local

Starting point: exact interpolation

Each input pattern x must be mapped onto a target value d



That is, given a set of N vectors $\underline{x}_i$ and a corresponding set of N real numbers, d_i (the targets), find a function F that satisfies the interpolation condition:

$F(\underline{x}_i) = d_i$ for $i = 1,\ldots,N$

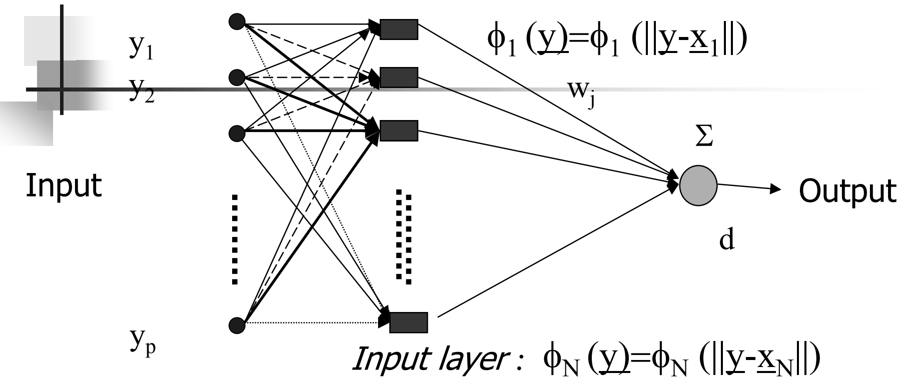
or more exactly find:

$$F(\underline{x}) = \sum_{j=1}^{N} w_j \phi(\|\underline{x} - \underline{x}_j\|)$$

satisfying:

$$F(\underline{x}_i) = \sum_{j=1}^{N} w_j \phi(\|\underline{x}_i - \underline{x}_j\|) = d_i$$

Single-layer networks



- output = $\Sigma\, w_i\, \phi_i(\underline{y} - \underline{x}_i)$
- adjustable parameters are weights w_j
- number of input units $\leq$ number of data points
- Form of the basis functions decided in advance

To summarize:

- ❖ For a given data set containing N points $(\underline{x}_i, d_i)$, i=1,…,N
- ❖ Choose a RBF function ϕ
- ❖ Calculate $\phi(\underline{x}_j - \underline{x}_i)$
- ❖ Solve the linear equation $\Phi\, \underline{W} = \underline{D}$
- ❖ Get the unique solution
- ❖ Done

➢ Like MLP's, RBFNs can be shown to be able to approximate any function to arbitrary accuracy (using an arbitrarily large numbers of basis functions).

➢ Unlike MLP's, however, they have the property of 'best approximation' i.e. there exists an RBFN with minimum approximation error.

Problems with exact interpolation

can produce poor generalisation performance as only data points constrain mapping

Overfitting problem

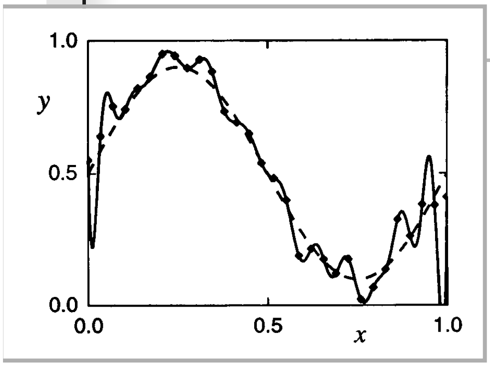
Bishop(1995) example

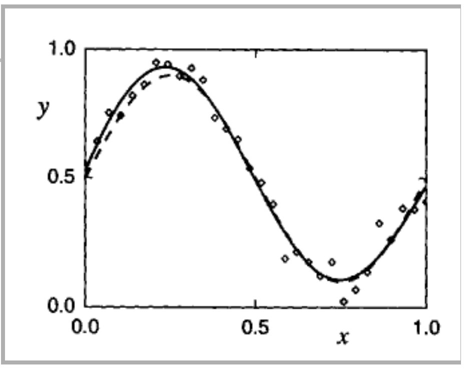
Underlying function f(x)=0.5+0.4sine(2π x)
sampled randomly for 30 points

added Gaussian noise to each data point

30 data points 30 hidden RBF units

fits all data points but creates oscillations due added noise and unconstrained between data points





All Data Points 5 Basis functions

To fit an RBF to every data point is very inefficient due to the computational cost of matrix inversion and is very bad for generalization so:

✓ Use less RBF's than data points, *i.e.* M<N

✓ Therefore don't necessarily have RBFs centred at data points

✓ Can include bias terms

✓ Can have Gaussian with general covariance matrices but there is a trade-off between complexity and the number of parameters to be found eg for *d* rbfs we have:

Fuzzy Modelling and Identification

Fuzzy Clustering with Application to Data-Driven Modelling

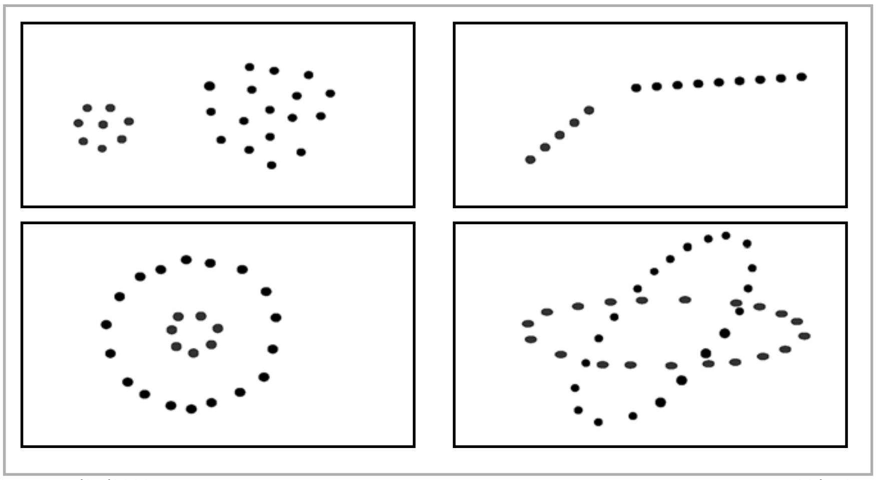
Introduction

- The ability to cluster data (concepts, perceptions, etc.)
  - essential feature of human intelligence.
- A cluster is a set of objects that are more similar to each other than to objects from other clusters.
- Applications of clustering techniques in pattern recognition and image processing.
- Some machine-learning techniques are based on the notion of similarity (decision trees, case-based reasoning)
- Non-linear regression and black-box modelling can be based on the partitioning data into clusters.

Section Outline

- **Basic concepts in clustering**
  - data set
  - partition matrix
  - distance measures
- **Clustering algorithms**
  - fuzzy c-means
  - Gustafson–Kessel
- **Application examples**
  - system identification and modelling
  - diagnosis

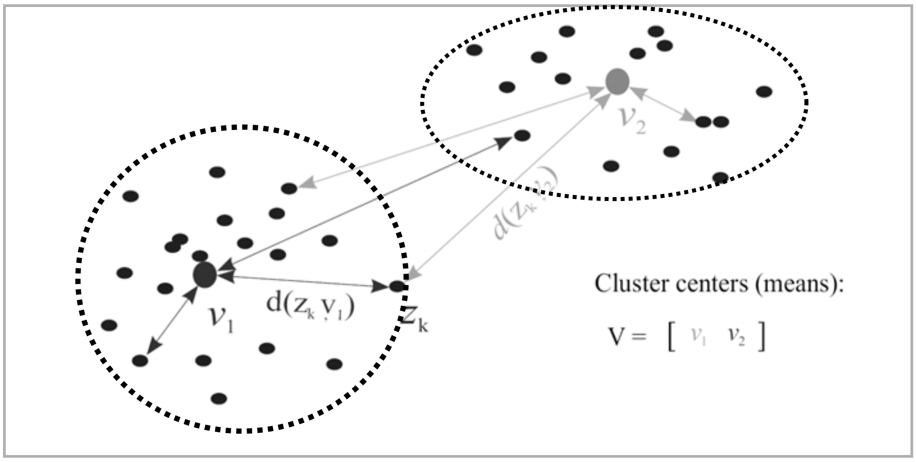
Examples of Clusters



Problem Formulation

- Given is a set of data in R^n and the (estimated) number of clusters to look for (a difficult problem, more on this later).
- Find the partitioning of the data into subsets (clusters), such that samples within a subset are more similar to each other than to samples from other subsets.
- Similarity is mathematically formulated by using a distance measure (i.e., a dissimilarity function).
- Usually, each cluster will have a prototype and the distance is measured from this prototype.

Distance Measure



Distance Measures

- **Euclidean norm:**
  - $d^2(\mathbf{z}_j, \mathbf{v}_i) = (\mathbf{z}_j - \mathbf{v}_i)^T (\mathbf{z}_j - \mathbf{v}_i)$
- **Inner-product norm:**
  - $d^2_{\mathbf{A}_i}(\mathbf{z}_j, \mathbf{v}_i) = (\mathbf{z}_j - \mathbf{v}_i)^T \mathbf{A}_i (\mathbf{z}_j - \mathbf{v}_i)$
- **Many other possibilities . . .**

Fuzzy Clustering: an Optimisation Approach

- **Objective function (least-squares criterion):**

$$J(\mathbf{Z}; \mathbf{V}, \mathbf{U}, \mathbf{A}) = \sum_{i=1}^{c} \sum_{j=1}^{N} \mu_{i,j}^{m} d_{\mathbf{A}_i}^{2}(\mathbf{z}_j, \mathbf{v}_i)$$

- **subject to constraints:**

$0 \leq \mu_{i,j} \leq 1,$	$i = 1, \ldots, c,\ j = 1, \ldots, N$	membership degree
$0 < \sum_{j=1}^{N} \mu_{i,j} < 1,$	$i = 1, \ldots, c$	no cluster empty
$\sum_{i=1}^{c} \mu_{i,j} = 1,$	$j = 1, \ldots, N$	total membership

Fuzzy Algorithm

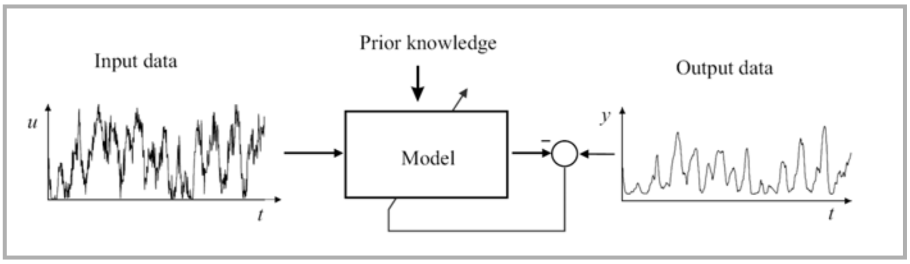
Repeat:

1. **Compute cluster prototypes (means):** $$v_i = \frac{\sum_{k=1}^{N} \mu_{i,k}^{m} \mathbf{z}_k}{\sum_{k=1}^{N} \mu_{i,k}^{m}}$$
2. **Calculate distances:** $$d_{ik} = (\mathbf{z}_k - \mathbf{v}_i)^T (\mathbf{z}_k - \mathbf{v}_i)$$
3. **Update partition matrix:** $$\mu_{ik} = \frac{1}{\sum_{j=1}^{c} (d_{ik}/d_{jk})^{1/(m-1)}}$$

until $\|\Delta \mathbf{U}\| < \epsilon$

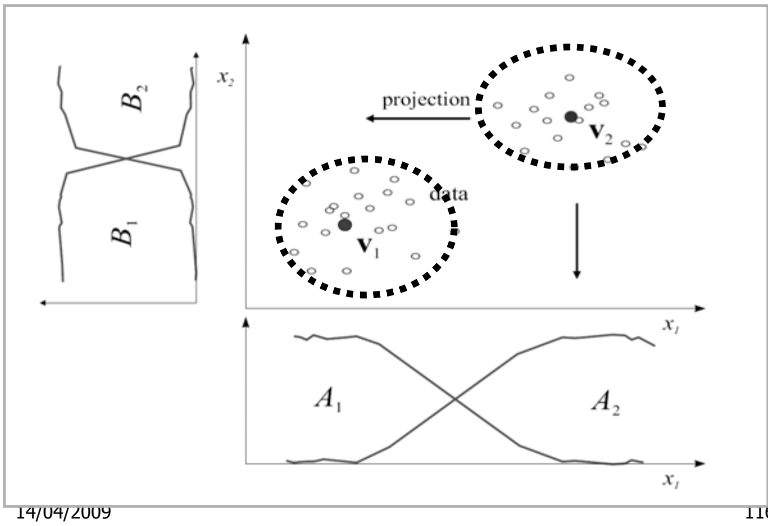
$$(i = 1,\ \cdots,\ c.\ k = 1, \cdots,\ N)$$

Data-Driven (Black-Box) Modelling

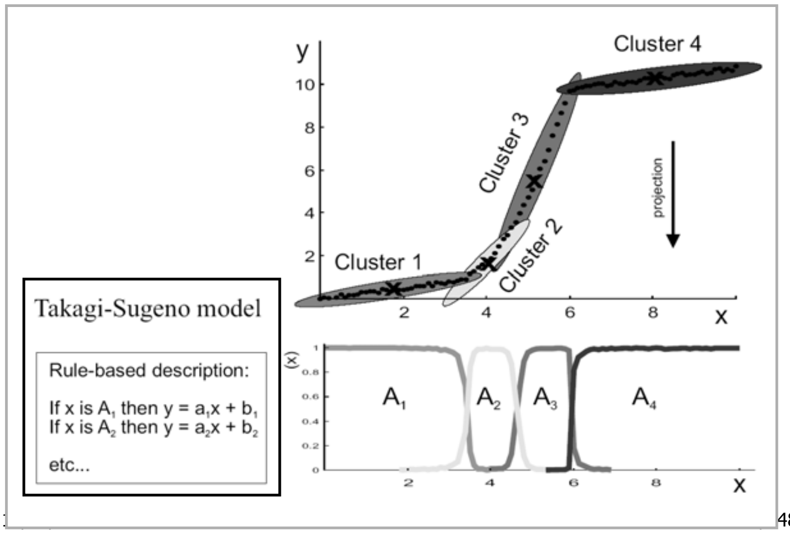


- ➢ Linear model (for linear systems only, limited in use)
- ➢ Neural network (black box, unreliable extrapolation)
- ➢ Rule-based model (more transparent, 'grey-box')

Extraction of Rules by Fuzzy Clustering



Extraction of Rules by Fuzzy Clustering



Example: Non-linear Autoregressive System (NARX)

$$\boxed{x(k+1) = f(x(k)) + \epsilon(k)}$$

$$f(x) = \begin{cases} 2x - 2, & 0.5 < x \\ -2x, & -0.5 \le x < 0.5 \\ 2x + 2, & x \le -0.5 \end{cases}$$

Structure Selection and Data Preparation

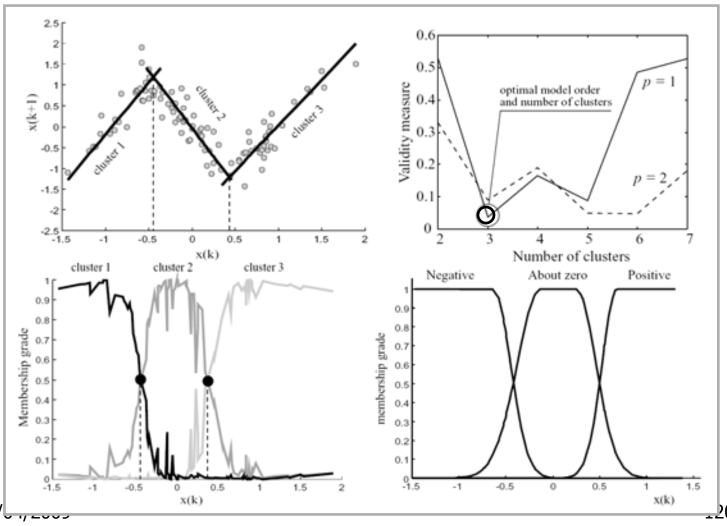
1. Choose model order p

$$x(k+1) = f(\underbrace{x(k), x(k-1), \ldots, x(k-p+1)}_{\mathbf{x}(k)})$$

2. Form pattern matrix **Z** to be clustered

$$\mathbf{Z}^T = \begin{bmatrix} x(1) & x(2) & \ldots & x(p) & x(p+1) \\ x(2) & x(3) & \ldots & x(p+1) & x(p+2) \\ \vdots & \vdots & & \vdots & \vdots \\ x(N-p) & x(N-p+1) & \ldots & x(N-1) & x(N) \end{bmatrix}$$

Clustering Results



Rules Obtained

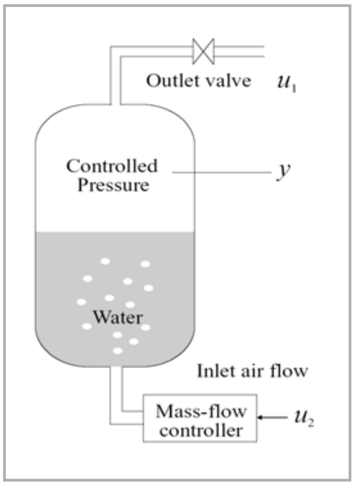
1) If $x(k)$ is $Positive$ then $x(k+1) = 2.0244x(k) - 2.0289$

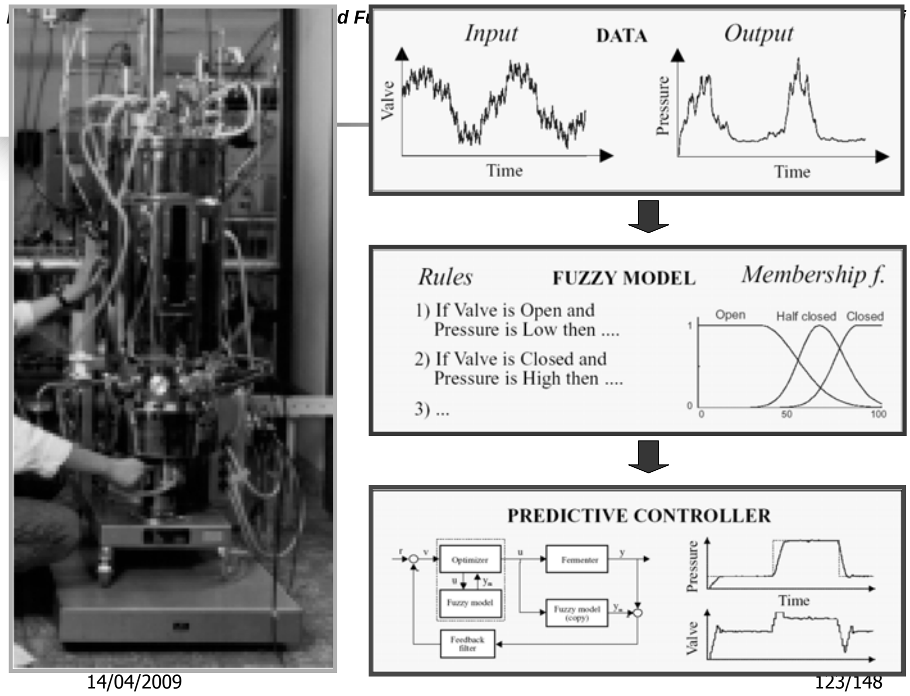
2) If $x(k)$ is $About\ zero$ then $x(k+1) = -1.8852x(k) + 0.0005$

3) If $x(k)$ is $Negative$ then $x(k+1) = 1.9050x(k) + 1.9399$

original function: $$f(x) = \begin{cases} 2x-2, & 0.5 < x \\ -2x, & -0.5 \leq x < 0.5 \\ 2x+2, & x \leq -0.5 \end{cases}$$

Identification of Pressure Dynamics





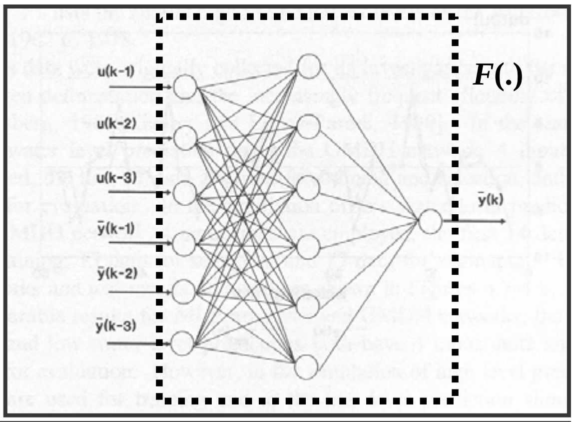
Application Examples

Neural Networks for

Non-linear Identification, Prediction and Control

Nonlinear *Dynamic* System

- Take a static NN
- From static to dynamic NN
- "Quasi-static" NN
- Add inputs, outputs and delayed signals



$$\tilde{y}(k) = F\left(u(k-1), u(k-2), u(k-3), \tilde{y}(k-1), \tilde{y}(k-2), \tilde{y}(k-3)\right)$$

Example of Quasi-static NN with 3 delayed inputs and outputs

Nonlinear System Identification

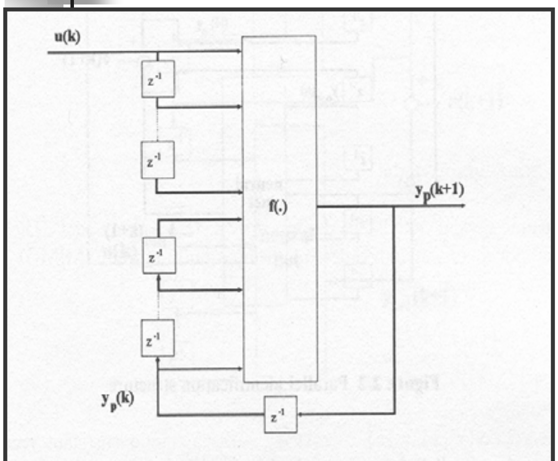


Figure 2.1 Input-output model

- f(.), unknown target function
- Nonlinear dynamic model
- Approximated via a quasi-static NN
- Nonlinear dynamic system identification
- Recall "*linear system identification*"

Nonlinear System Identification

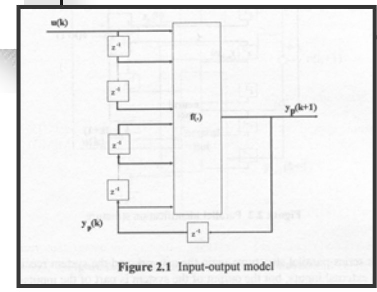
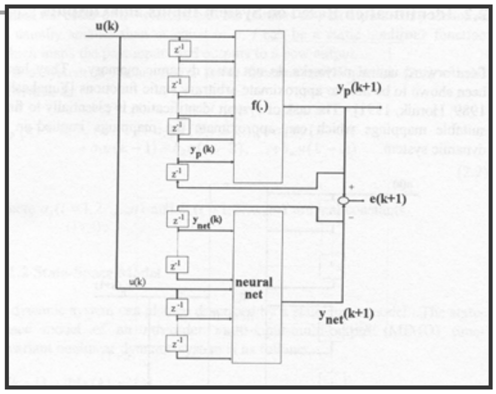
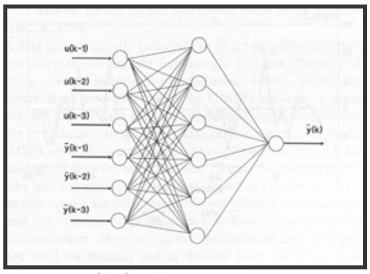


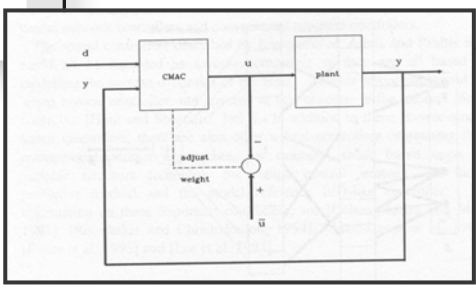
Figure 2.1 Input-output model

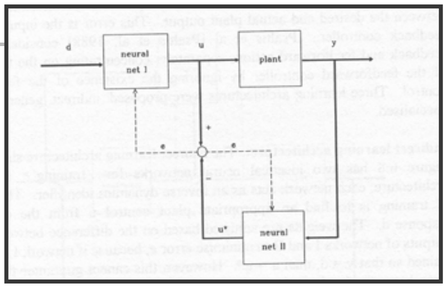




Target function: $y_p(k+1) = f(.)$
Identified function: $y_{NET}(k+1) = F(.)$
Estimation error: $e(k+1)$

Nonlinear System Neural Control





d: *reference/desired response*
y: *system output/desired output*
u: *system input/controller output*
ū: *desired controller input*
u^*: *NN output*
e: *controller/network error*

The goal of training is to find an appropriate plant control u from the desired response d. The weights are adjusted based on the difference between the outputs of the networks I & II to minimise e. If network I is trained so that y = d, then $u = u^*$. Networks act as inverse dynamics identifiers.

Neural Networks for Control

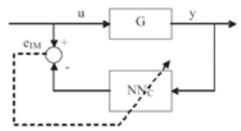


Figure 3: Training the neural network NN_C

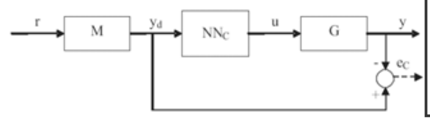


Figure 1: Direct Inverse Control using neural networks

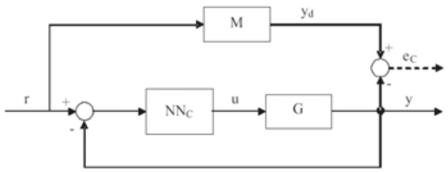


Figure 2: Model Reference Control using neural networks

Figures 1 and 3 Problems.

- Open-loop unstable models
- Disturbances

Neural Model Reference Adaptive Control

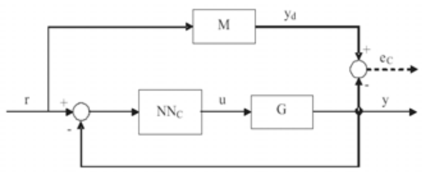


Figure 2: Model Reference Control using neural networks

The signal e_C is used to train or adapt online the weights of the controller NN_C. Two are the approaches used to design a MRAC control for an unknown plant: **Direct and Indirect Control**.

Direct Control: This procedure aims at designing a controller without having a plant model. As the knowledge of the plant is needed in order to train the neural network which corresponds to the controller (*i.e.* NN_C), until present, no method has been proposed to deal with this problem.

Neural Model Reference Adaptive Control

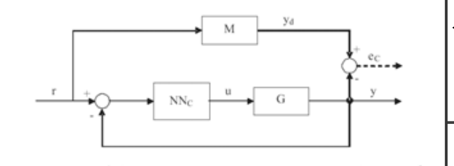


Figure 2: Model Reference Control using neural networks

Indirect Control

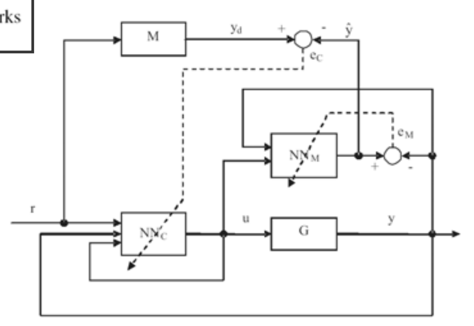


Figure 4: Indirect MRAC

- The signal e_C is used to train or adapt online the weights of the neural controller NN_C.

Indirect Control: NN_M & NN_C

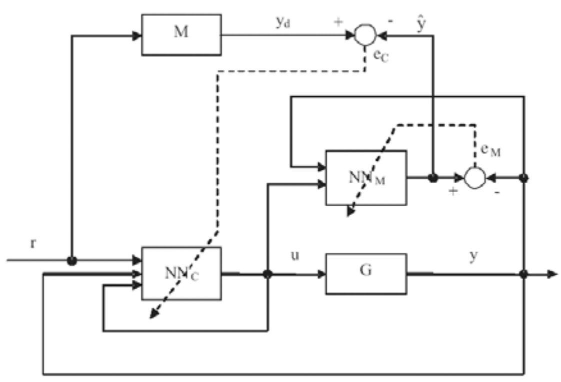


Figure 4: Indirect MRAC

- This approach uses two neural networks: one for modelling the plant dynamics (NN_M), and another one trained to control the real plant (G) so as its behaviour is as close as possible to the reference model (M) via the neural controller (NN_C).

Indirect Control (1)

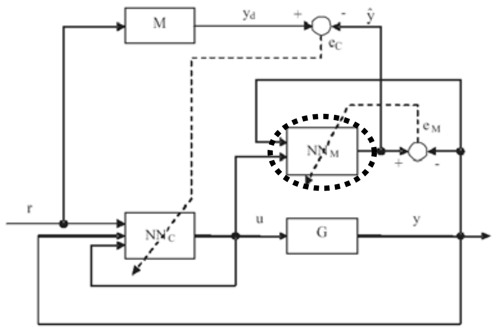


Figure 4: Indirect MRAC

The **neural network NN$_M$** is trained to approximate the plant **G** input/output relation using the signal e_M. This is usually done offline, using a batch of data gathered from the plant in open loop.

Indirect Control (2)

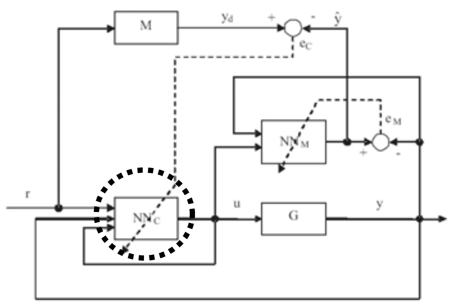


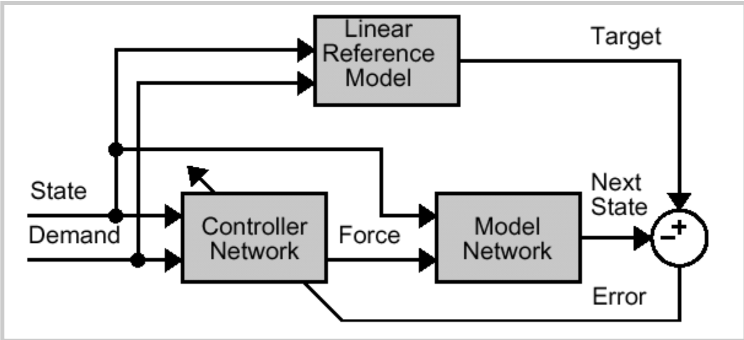
Figure 4: Indirect MRAC

Then, NN_M is fixed, its output and behaviou are known and easy to compute.

Once the model NN_M is trained, it is used to train the network NN_C which will act as the controller. The model NN_M is used instead of the real plant's output because the real plant is unknown, so back-propagation algorithms can not be used. In this way, the control error e_C is calculated as the difference between the desired reference model output y_d and $\hat{y}$, which is the closed loop predicted output.

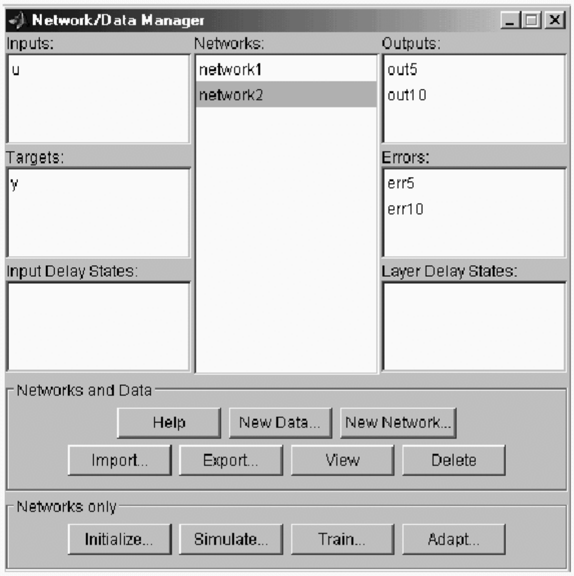
Model Reference Control

Matlab and Simulink solution

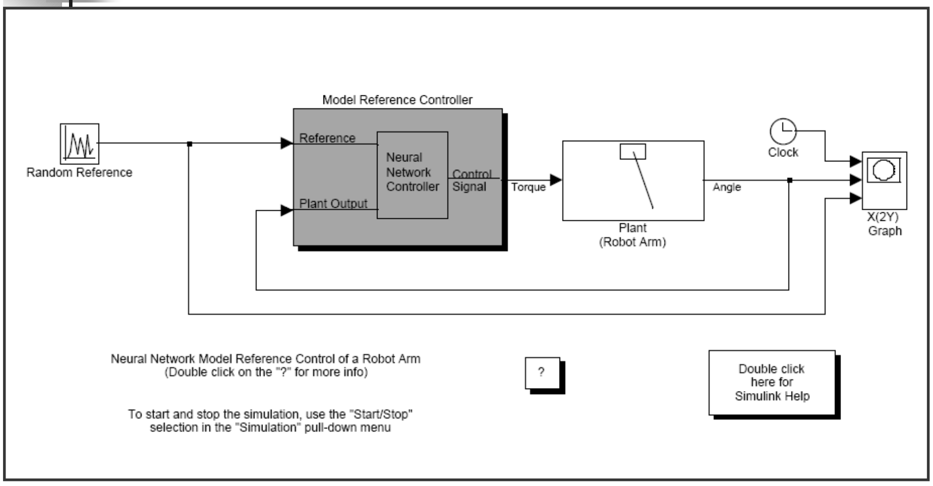


Neural controller, reference model, neural model

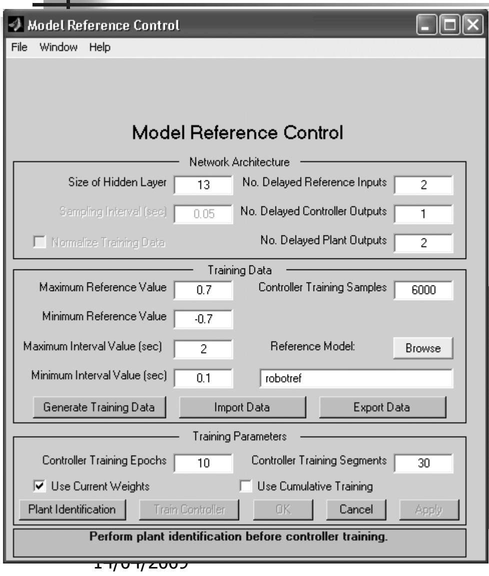
Matlab NNtool GUI (Graphical User Interface)

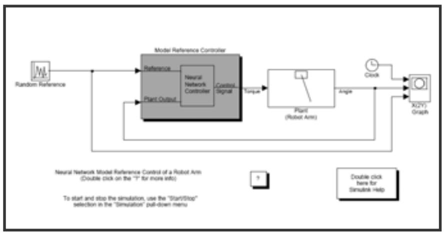


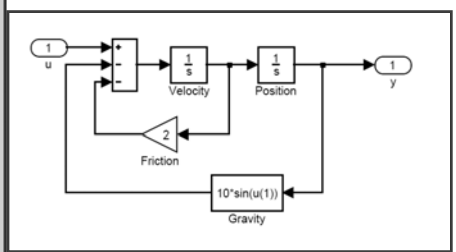
Control of a Robot Arm Example



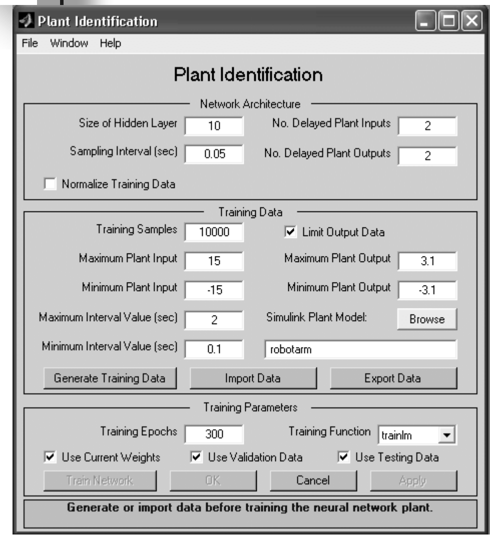
Control of a Robot Arm Example

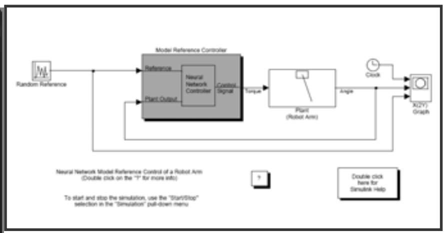






Control of a Robot Arm Example

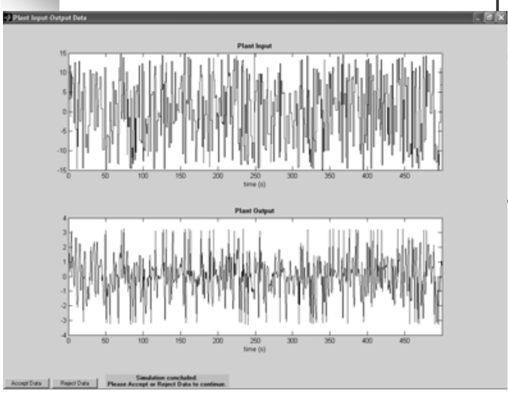


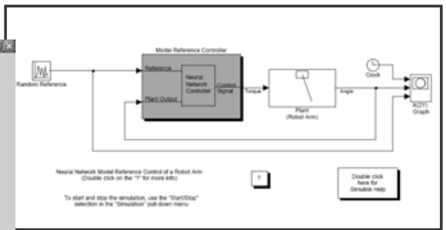


Plant Identification:

Data generation from the Reference Model for Neural Network training

Control of a Robot Arm Example

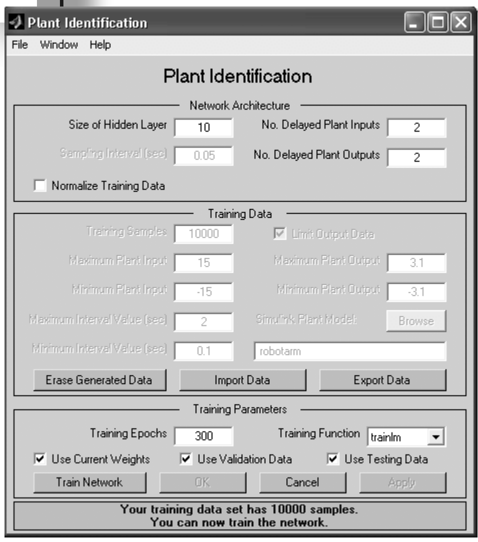


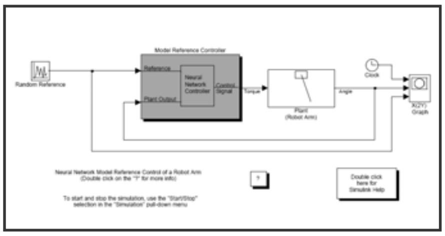


After Plant Identification:

Neural Network training

Control of a Robot Arm Example

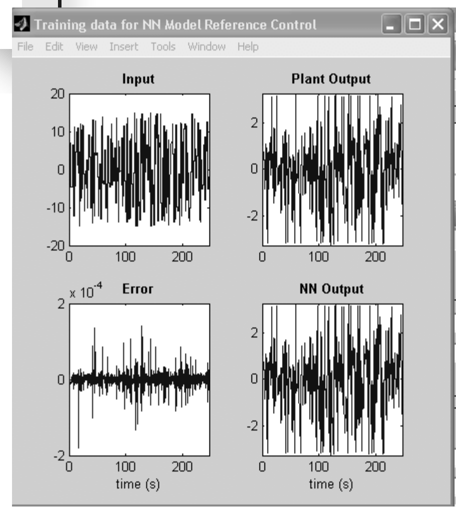


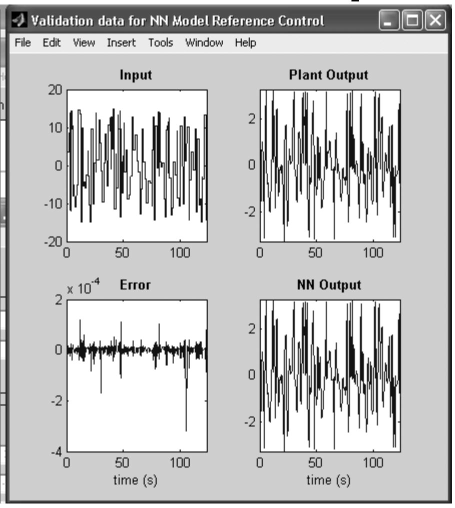


After Plant Identification:

Neural Network training

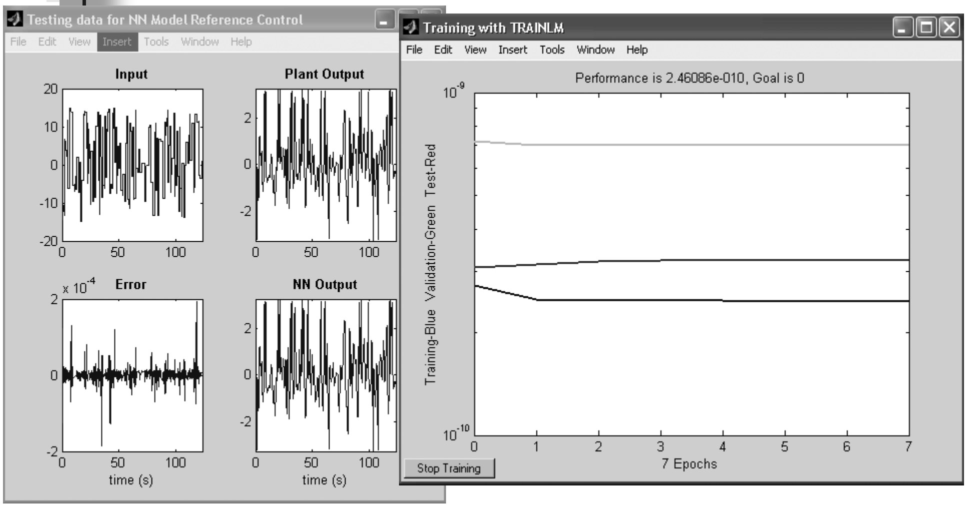
Control of a Robot Arm Example





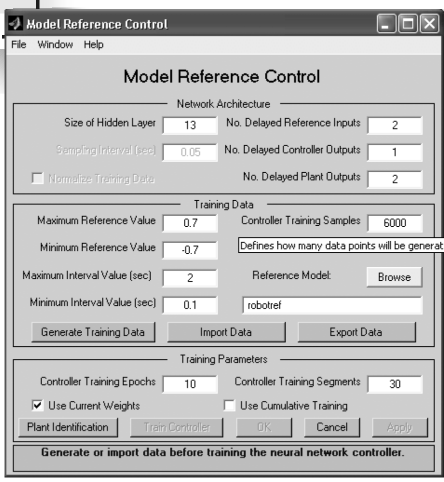
Training and Validation Data

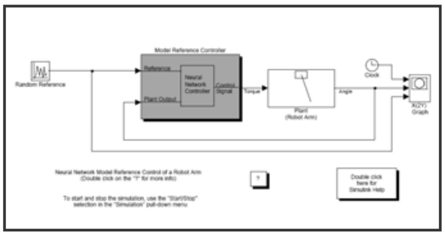
Control of a Robot Arm Example



Testing Data and Training Results

Control of a Robot Arm Example

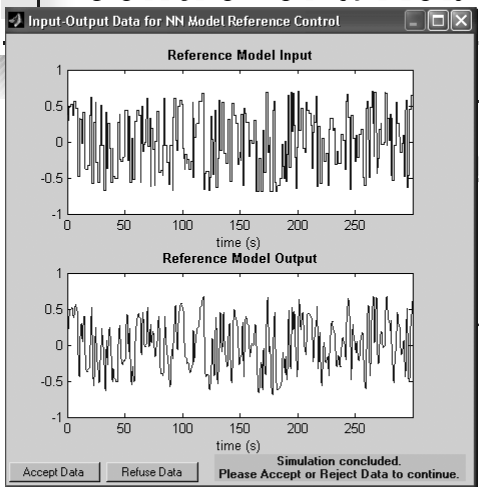


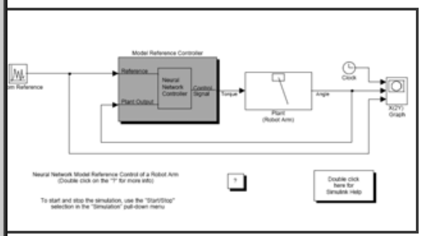


Plant identification with a NN

Data Generation for NN Controller Identification

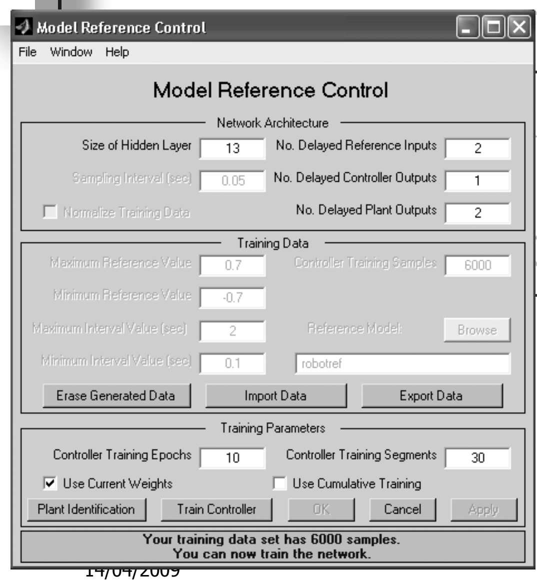
Control of a Robot Arm Example

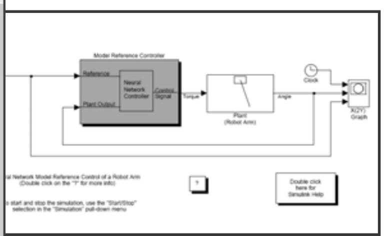




Accept the Data Generated for NN Controller Identification

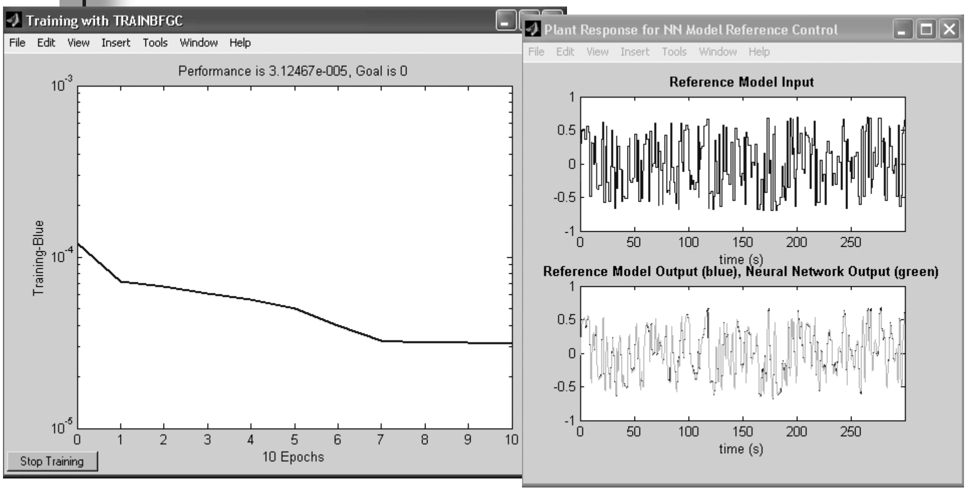
Control of a Robot Arm Example





NN Controller Training

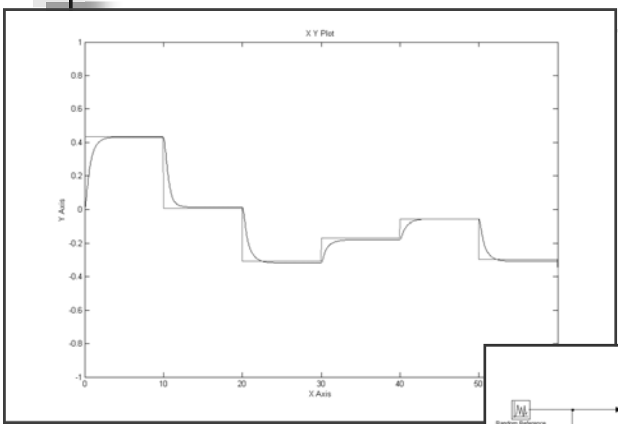
Control of a Robot Arm Example



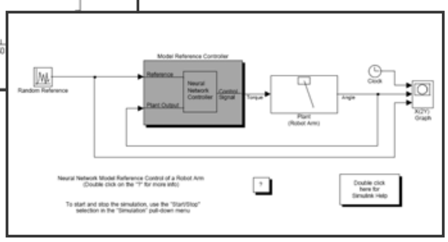
NN Controller Training and Results

Lecture Notes on Neural Networks and Fuzzy Systems *Silvio Simani*

Control of a Robot Arm Example



Reference and Tracked Output Signals



Simulation Final Results