Artificial Neural Networks

By: ***Bijan Moaveni***

Email: *b_moaveni@iust.ac.ir*
http://webpages.iust.ac.ir/b_moaveni/

Programs of the Course

- Aims of the Course
- Reference Books
- Preliminaries
- Evaluation

Aims of the Course

1. Discuss the fundamental techniques in Neural Networks.
2. Discuss the fundamental structures and its learning algorithms.
3. Introduce the new models of NNs and its applications.

Neural Network is an intelligent numerical computation method.

Learning Outcomes

1. Understand the relation between real brains and simple artificial neural network models.
2. Describe and explain the most common architectures and learning algorithms for Multi-Layer Perceptrons, Radial-Basis Function Networks and Kohonen Self-Organising Maps.
3. Explain the learning and generalization aspects of neural network systems.
4. Demonstrate an understanding of the implementation issues for common neural network systems.
5. Demonstrate an understanding of the practical considerations in applying neural networks to real classification, recognition, identification, approximation problems and control.

Course Evaluation

1. **Course Projects 40%**
2. **Final Exam 50%**
3. **Conference Paper 10%**

Reference Books

- Haykin S., **Neural Networks: A Comprehensive Foundation**., Prentice Hall, 1999.
- Hagan M.T., Dcmuth H.B. and Beale M., **Neural Network Design**, PWS Publishing Co., 1996.

Preliminaries

1. Matrices Algebra to Neural Network design and implementation.
2. MATLAB software for simulation. (NN toolbox is arbitrary).

Artificial Neural Networks

Lecture 2

1

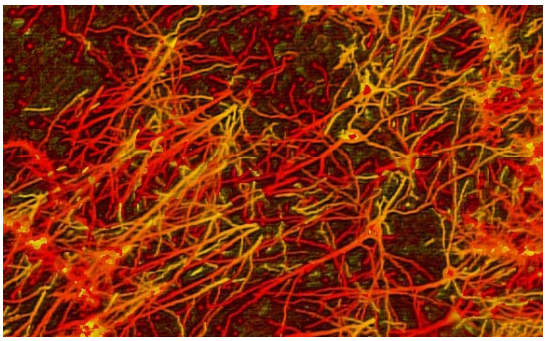
Introduction

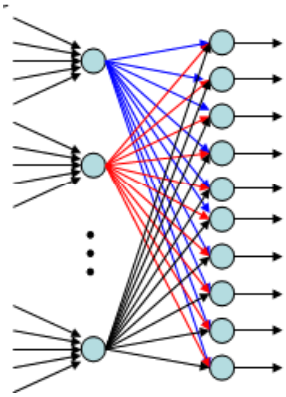
1. What are Neural Networks?
2. Why are Artificial Neural Networks Worth Noting and Studying?
3. What are Artificial Neural Networks used for?
4. Learning in Neural Networks
5. A Brief History of the Field
6. Artificial Neural Networks compared with Classical Symbolic A.I.
7. Some Current Artificial Neural Network Applications

2

What are Neural Networks?

1. Neural Networks (NNs) are networks of neurons such as found in real (i.e. biological) brains.

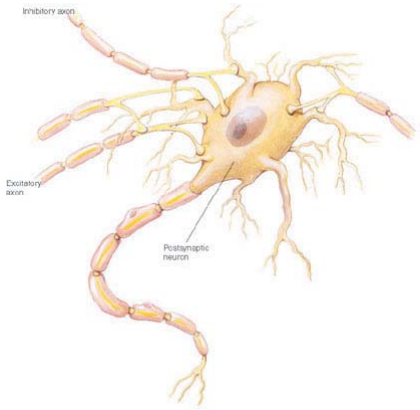


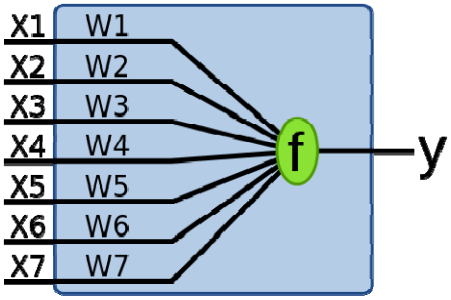


3

What are Neural Networks?

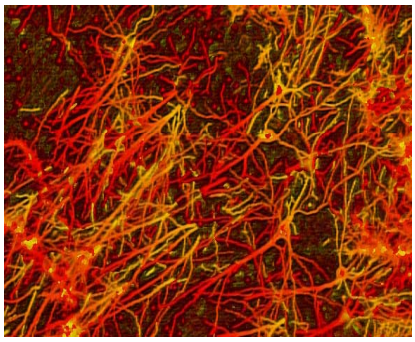
2. Artificial Neurons are crude approximations of the neurons found in real brains.

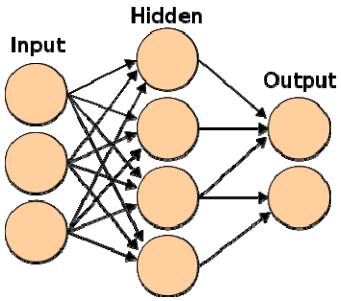




What are Neural Networks?

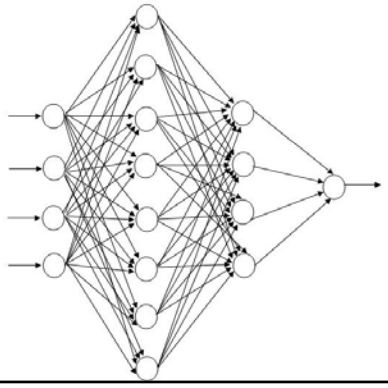
3. Artificial Neural Networks (ANNs) are networks of Artificial Neurons, and hence constitute crude approximations to parts of real brains.





What are Neural Networks?

4. From a practical point of view, an ANN is just a parallel computational system consisting of many simple processing elements connected together in a specific way in order to perform a particular task.



6

Why are Artificial Neural Networks Worth Noting and Studying?

1. They are extremely powerful computational devices.
2. Parallel Processing makes them very efficient.
3. They can learn and generalize from training data – so there is no need for enormous feats of programming.
4. They are particularly fault tolerant – this is equivalent to the "graceful degradation" found in biological systems.
5. They are very noise tolerant – so they can cope or deal with situations where normal symbolic (classic) systems would have difficulty.
6. In principle, they can do anything a symbolic or classic system can do, and more.

7

What are Artificial Neural Networks used for?

- **Brain modeling** : The scientific goal of building models of how real brains work. This can potentially help us understand the nature of human intelligence, formulate better teaching strategies, or better remedial actions for brain damaged patients.
- **Artificial System Building :** The engineering goal of building efficient systems for real world applications. This may make machines more powerful, relieve humans of tedious tasks, and may even improve upon human performance.

8

Learning in Neural Networks

There are many forms of neural networks. Most operate by passing neural 'activations' through a network of connected neurons.

One of the most powerful features of neural networks is their ability to ***learn*** and ***generalize*** from a set of training data. They adapt the strengths/weights of the connections between neurons so that the final output activations are correct.

There are three broad types of learning:

1. Supervised Learning (i.e. learning with a teacher)
2. Reinforcement learning (i.e. learning with limited feedback)
3. Unsupervised learning (i.e. learning with no help)

There are most common learning algorithms for the most common types of neural networks.

9

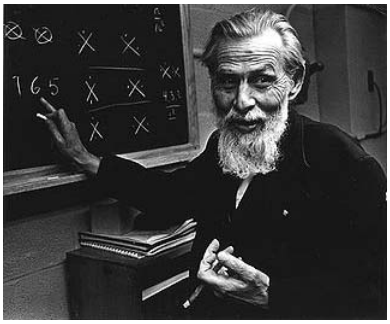
A Brief History

- **1943** McCulloch and Pitts proposed the McCulloch-Pitts neuron model
- **1949** Hebb published his book *The Organization of Behavior*, in which the Hebbian learning rule was proposed.
- **1958** Rosenblatt introduced the simple single layer networks now called Perceptrons.
- **1969** Minsky and Papert's book *Perceptrons* demonstrated the limitation of single layer perceptrons and almost the whole field went into hibernation.
- **1982** Hopfield published a series of papers on Hopfield networks.
- **1982** Kohonen developed the Self-Organising Maps that now bear his name.
- **1986** The Back-Propagation learning algorithm for Multi-Layer Perceptrons was rediscovered and the whole field took off again.
- **1990s** The sub-field of Radial Basis Function Networks is developed.
- **2000s** The power of Ensembles of Neural Networks and Support Vector Machines becomes apparent.

10

A Brief History

- **1943** McCulloch and Pitts proposed the McCulloch-Pitts neuron model



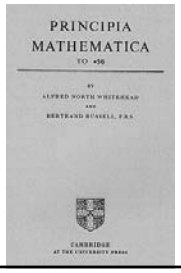
Warren S. McCulloch
(Nov., 16, 1898 – Sep., 24, 1969)
American neurophysiologist and cybernetician

W. McCulloch and W. Pitts, 1943 "***A Logical Calculus of the Ideas Immanent in Nervous Activity***". In :*Bulletin of Mathematical Biophysics* Vol 5, pp 115-133 .

A Brief History

- **1943** McCulloch and Pitts proposed the McCulloch-Pitts neuron model

Walter Pitts
(23 April 1923 – 14 May 1969)



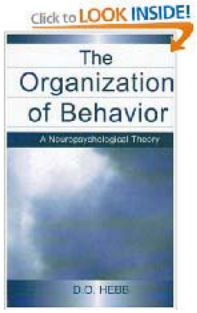
At the age of 12 he spent three days in a library reading *Principia Mathematica* and sent a letter to Bertrand Russell pointing out what he considered serious problems with the first half of the first volume.

12

A Brief History

- **1949** Hebb published his book *The Organization of Behavior The Organization of Behavior*, in which the Hebbian learning rule was proposed.

Donald Olding Hebb
(July 22, 1904 – August 20, 1985)



The Organization of Behavior

13

A Brief History

- **1958** Rosenblatt introduced the simple single layer networks now called Perceptrons.

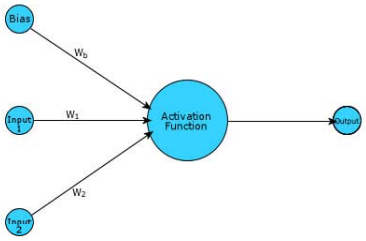
Frank Rosenblatt
(11 July 1928 – 1971)

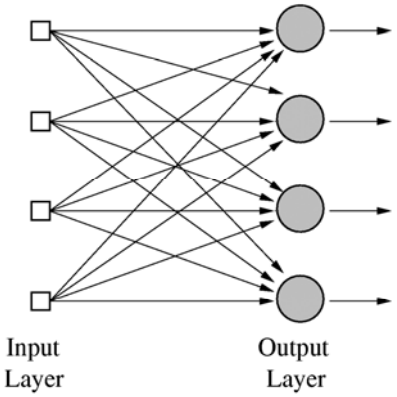
2006 - LAWRENCE J. FOGEL
2007 - JAMES C. BEZDEK
2008 - TEUVO KOHONEN
2009 - JOHN J. HOPFIELD
2010 - MICHIO SUGENO

14

A Brief History

- **1958** Rosenblatt introduced the simple ***single layer networks*** now called Perceptrons.





15

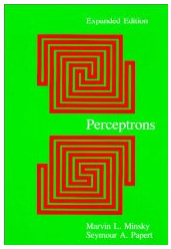
A Brief History

- **1969** Minsky and Papert's book *Perceptrons* demonstrated the limitation of single layer perceptrons and almost the whole field went into hibernation.

Marvin Minsky
(born August 9, 1927)

Seymour Papert
(born February 29, 1928)

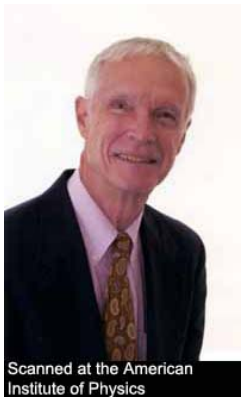
Perceptrons



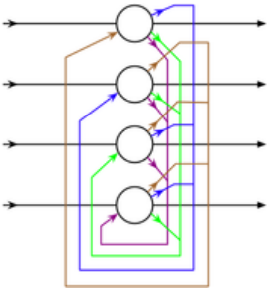
16

A Brief History

- **1982** Hopfield published a series of papers on Hopfield networks.



John Joseph Hopfield
(born July 15, 1933)



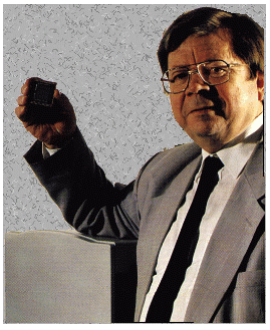
A Hopfield Net

He was awarded the Dirac Medal of the ICTP in 2001.

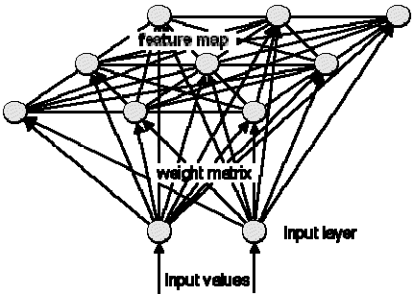
17

A Brief History

- **1982** Kohonen developed the Self-Organizing Maps that now bear his name.



Teuvo Kohonen
(born July 11, 1934)



S.O.M

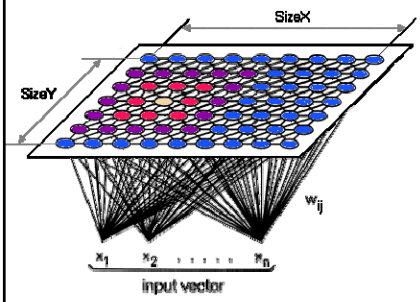
Self-Organizing Maps

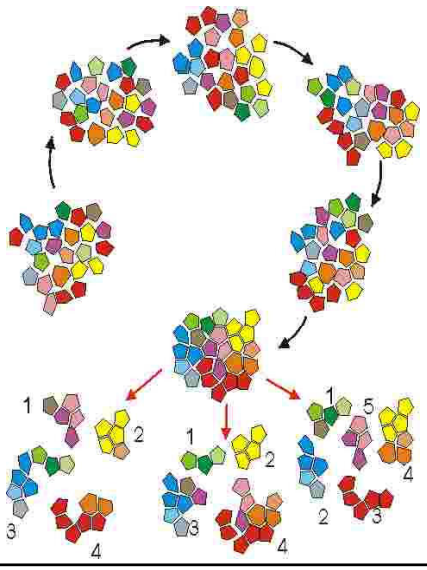
New ed.: 2001

18

A Brief History

- **1982** Kohonen developed the ***Self-Organizing Maps*** that now bear his name.





19

A Brief History

- **1986** The Back-Propagation learning algorithm for Multi-Layer Perceptrons was rediscovered and the whole field took off again.
- **1990s** The sub-field of Radial Basis Function Networks is developed.
- **2000s** The power of Ensembles of Neural Networks and Support Vector Machines becomes apparent.

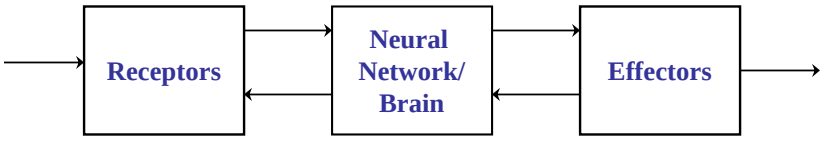
20

Artificial Neural Networks

Lecture 3

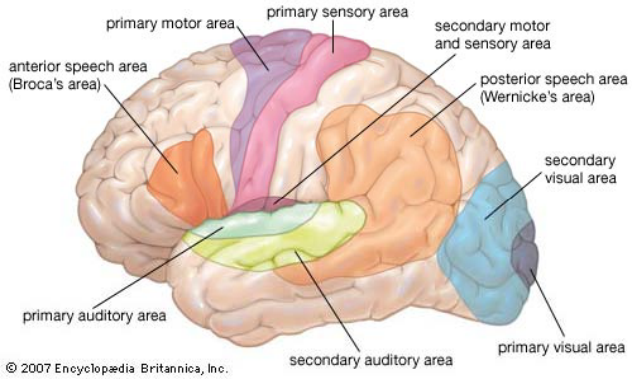
Human Nervous System

- **The human nervous system can be represented to three stages as the following block diagram:**



The Human Brain

- **The middle block of last block-diagram (Brain)**

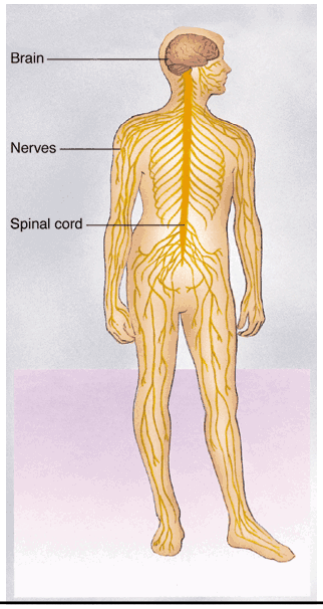


Brains versus Computers

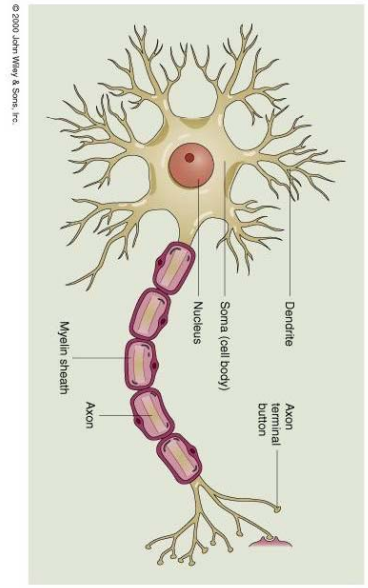
1. There are approximately 10 billion neurons in the human cortex, compared with 10 of thousands of processors in the most powerful parallel computers.
2. Each biological neuron is connected to several thousands of other neurons, similar to the connectivity in powerful parallel computers.
3. Lack of processing units can be compensated by speed. The typical operating speeds of biological neurons is measured in milliseconds (10^{-3} s), while a silicon chip can operate in nanoseconds (10^{-9} s).
4. The human brain is extremely energy efficient, using approximately 10^{-16} joules per operation per second, whereas the best computers today use around 10^{-6} joules per operation per second.
5. Brains have been evolving for tens of millions of years, computers have been evolving for tens of decades.

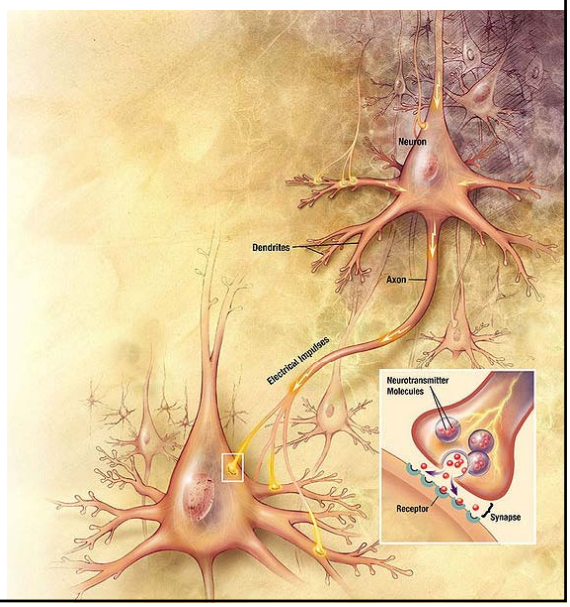
Human Nervous System

- **The real structure of the human nervous corresponding to last block-diagram.**
- **It contains the neurons to transfer the signal form the receptors to brain and vice-versa to the effectors.**

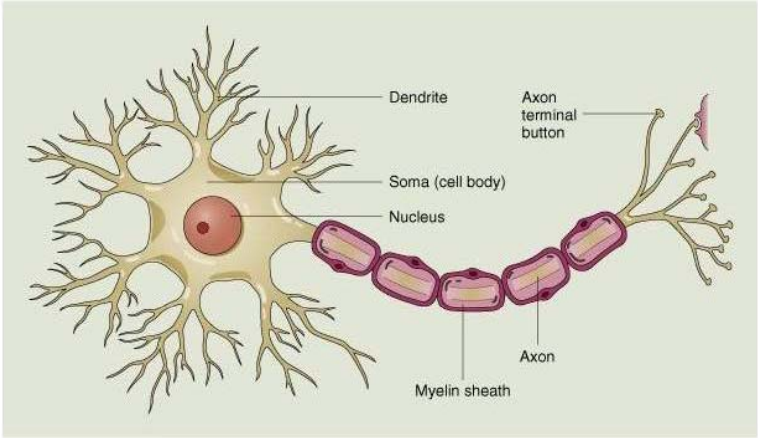


The Biological Neuron

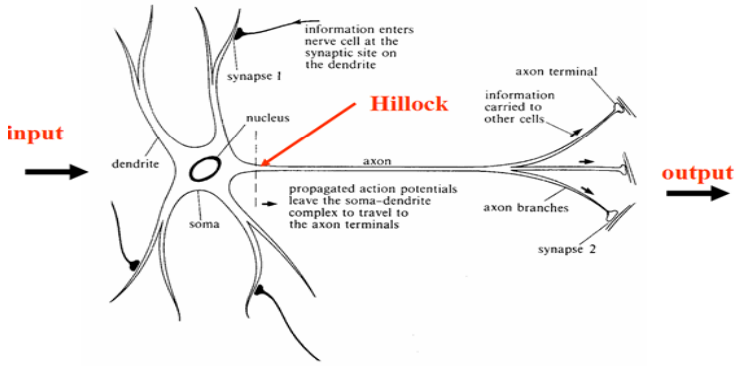




The Biological Neuron



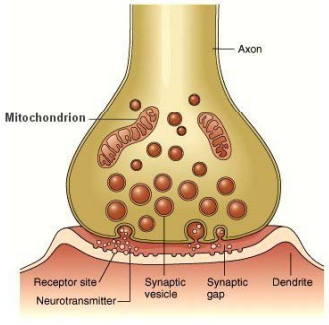
Components of Biological Neuron



1. The majority of ***neurons*** encode their activations or outputs as a series of brief electrical pulses (i.e. spikes or action potentials).
2. The neuron's ***cell body*** ***(soma)*** processes the incoming activations and converts them into output activations.
3. The neuron's ***nucleus*** contains the genetic material in the form of DNA. This exists in most types of cells, not just neurons.

Components of Biological Neuron

4. ***Dendrites*** are fibres which come from the cell body and provide the receptive zones that receive activation from other neurons.
5. ***Axons*** are fibres acting as transmission lines that send activation to other neurons.
6. The junctions that allow signal transmission between the axons and dendrites are called ***synapses***. The process of transmission is by diffusion of chemicals called ***neurotransmitters*** across the synaptic cleft.



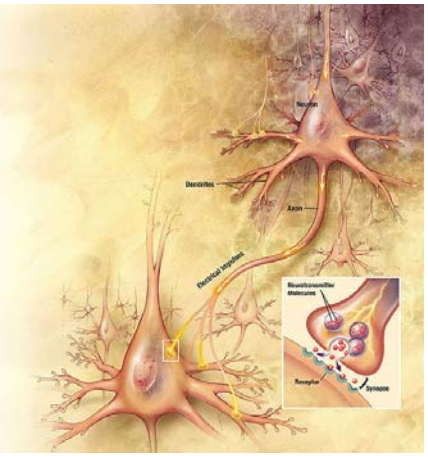
Level of Brain Organization

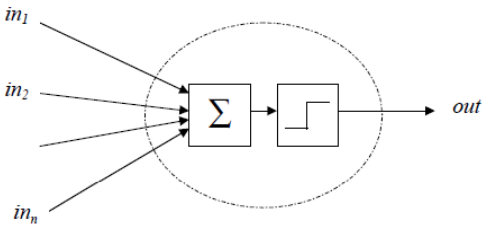
There is a hierarchy of interwoven levels of organization:

1. Molecules and Ions
2. Synapses
3. Neuronal microcircuits
4. Dendrite trees
5. Neurons
6. Local circuits
7. Inter-regional circuits
8. Central nervous system

The ANNs we study in this module are crude approximations to levels 5 and 6.

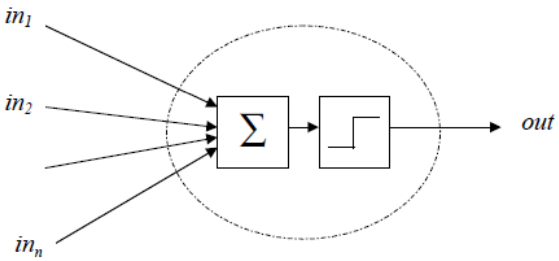
The McCulloch and Pitts Neuron





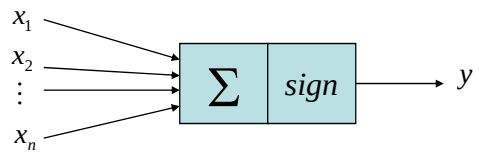
This is a simplified model of real neurons known as a ***Threshold Logic Unit.***

The McCulloch and Pitts Neuron

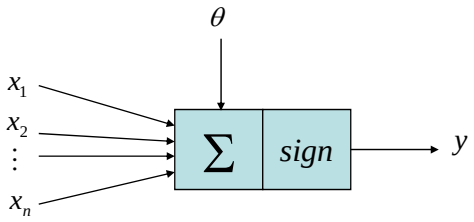


1. A set of synapses (i.e. connections) brings in activations from other neurons.
2. A processing unit sums the inputs, and then applies a non-linear activation function (i.e. squashing/transfer/threshold function).
3. An output line transmits the result to other neurons.

The McCulloch and Pitts Neuron Equation



$$y = sign\left(\sum_{i=1}^{n} x_i\right)$$



$$y = sign\left(\sum_{i=1}^{n} x_i - \theta\right)$$

The McCulloch and Pitts Neuron Analysis

• Note that the McCulloch-Pitts neuron is an extremely simplified model of real biological neurons. Some of its missing features include: non-binary inputs and outputs, non-linear summation, smooth thresholding, stochasticity, and temporal information processing.

• Nevertheless, McCulloch-Pitts neurons are computationally very powerful. One can show that assemblies of such neurons are capable of universal computation.

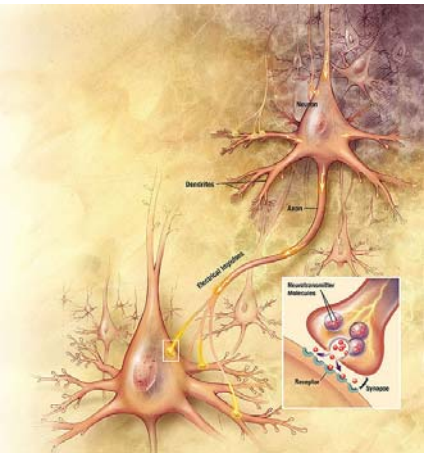
Artificial Neural Networks

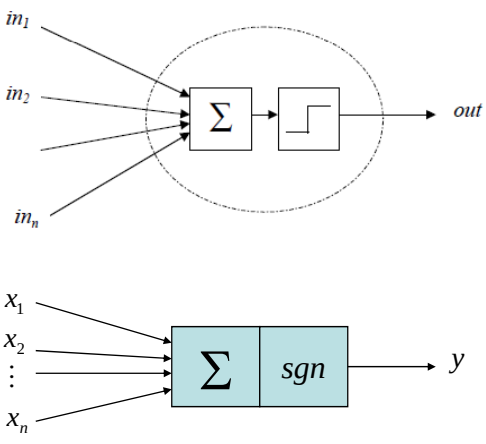
Lecture 4

Networks of McCulloch-Pitts Neurons

1

The McCulloch and Pitts (M_P) Neuron

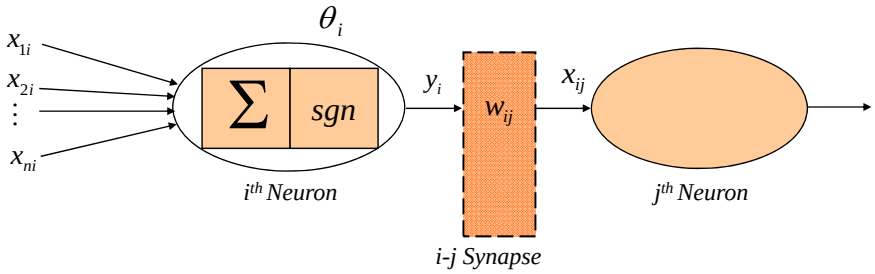




2

Networks of M-P Neurons

One neuron can't do much on its own, but a net of these neurons …



$$x_{ki} = y_k w_{ki}$$

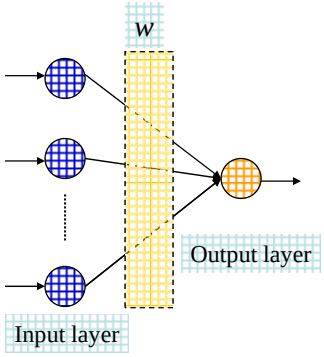
$$x_{ij} = y_i w_{ij}$$

$$y_i = \text{sgn}(\sum_{k=1}^{n} x_{ki} - \theta_i)$$

3

Networks of M-P Neurons

We can connect several number of McCulloch-Pitts neurons together, as follow:



An arrangement of one input layer of McCulloch-Pitts neurons feeding forward to one output layer of McCulloch-Pitts neurons as above is known as a ***Perceptron***.

4

Implementing Logic Gates with M-P Neurons

According to the McCulloch-Pitts Neuron properties we can use it to implement the basic logic gates.

Not	
in	*out*
1	*0*
0	*1*

And		
In_1	in_2	*out*
1	*1*	*1*
1	*0*	*0*
0	*1*	*0*
0	*0*	*0*

OR		
In_1	in_2	*out*
1	*1*	*1*
1	*0*	*1*
0	*1*	*1*
0	*0*	*0*

What should we do to implement or realize a logic gate, Not/AND/OR, by N.N.?

5

Implementing Logic Gates with M-P Neurons

What should we do to implement or realize a logic gate, Not/AND/OR, by N.N.?

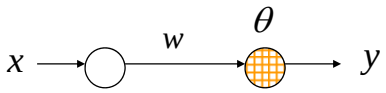
All we need to do is find the appropriate synapses (connection) weights and neuron thresholds to produce the right outputs corresponding to each set of inputs.

Two solutions can be introduced for this problem:

1. ***Analytically Approach***
2. ***Learning Algorithms***

6

Find Weights Analytically for NOT



$$y = \text{sgn}(x \times w - \theta)$$

Not	
in	*out*
1	***0***
0	*1*

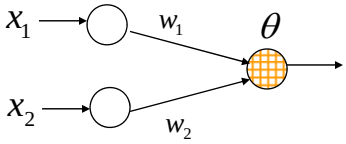
$$y = \text{sgn}(w - \theta) = 0 \Rightarrow w - \theta < 0 \Rightarrow w < \theta$$

$$y = \text{sgn}(-\theta) = 1 \Rightarrow \theta < 0$$

So: $\theta = -0.5$ $\Rightarrow w = -1$

7

Find Weights Analytically for AND gate



$$y = \text{sgn}(x_1 \times w_1 + x_2 \times w_2 - \theta)$$

And		
In1	*in2*	*out*
1	*1*	*1*
1	*0*	*0*
0	*1*	*0*
0	*0*	*0*

$$y = \text{sgn}(w_1 + w_2 - \theta) = 1 \Rightarrow w_1 + w_2 > \theta$$

$$y = \text{sgn}(w_1 - \theta) = 0 \Rightarrow w_1 < \theta$$

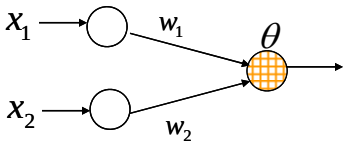
$$y = \text{sgn}(w_2 - \theta) = 0 \Rightarrow w_2 < \theta$$

$$y = \text{sgn}(-\theta) = 0 \Rightarrow \theta > 0$$

So: $\theta = 1.5$ $w_1 = w_2 = 1$

8

Find Weights Analytically for XOR gate



$$y = \text{sgn}(x_1 \times w_1 + x_2 \times w_2 - \theta)$$

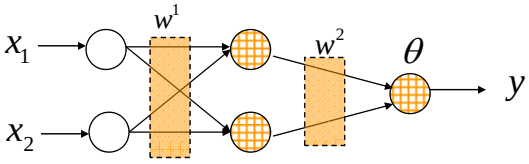
XOR		
In_1	in_2	*out*
1	1	0
1	0	1
0	1	1
0	0	0

$$y = \text{sgn}(w_1 + w_2 - \theta) = 0 \Rightarrow \quad w_1 + w_2 < \theta$$
$$y = \text{sgn}(w_1 - \theta) = 1 \quad \Rightarrow \quad w_1 > \theta$$
$$y = \text{sgn}(w_2 - \theta) = 1 \quad \Rightarrow \quad w_2 > \theta$$
$$y = \text{sgn}(-\theta) = 0 \quad \Rightarrow \quad \theta > 0$$

But, the 1st equation is not compatible with others.

Find Weights Analytically for XOR gate

What is the solution?



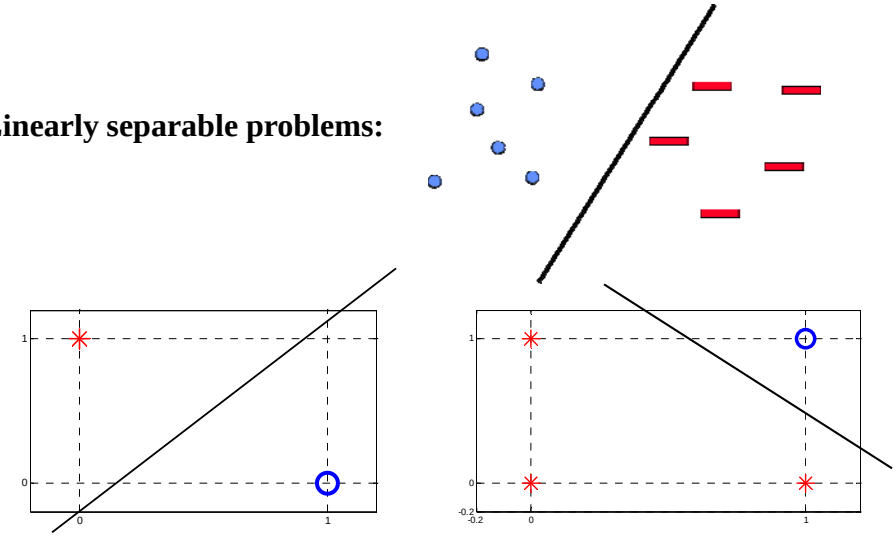
New questions:

- How can compute the weights and thresholds?
- Is analytically solution reasonable and practical or not?

10

A New Idea: *Learning Algorithm*

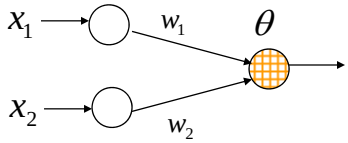
Linearly separable problems:



Not ***AND***

A New Idea: *Learning Algorithm*

Why is single layer neural networks capable to solve the linearly separable problems ?



$$y = sign(x_1 \times w_1 + x_2 \times w_2 - \theta)$$

$$\sum_{i=1}^{2} w_i x_i - \theta = 0$$

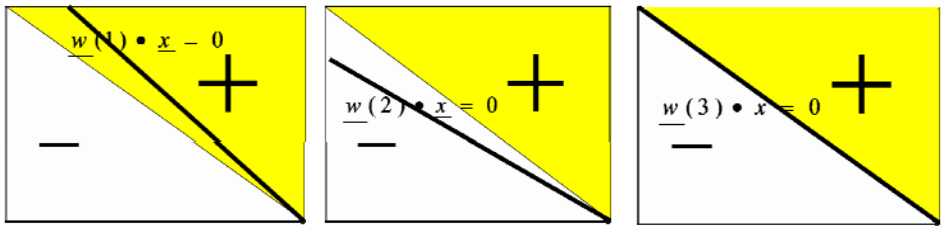
$$\sum_{i=1}^{2} w_i x_i - \theta_i > 0$$

$$\sum_{i=1}^{2} w_i x_i - \theta_i < 0$$

Learning Algorithm

What is the goal of learning algorithm?

We need a learning algorithm which it updates the weights w_i ($\underline{w}$) so that finally (at end of learning process) the input patterns lie on both sides of the line decided by the Perceptron.



Step: 1 *Step: 2* *Step: 3*

13

Learning Algorithm

Perceptron Learning Rule:

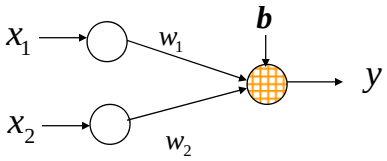
$$\underline{w}(t+1) = \underline{w}(t) + \eta(t)[d(t) - sign(\underline{w}(t) \bullet \underline{x}(t)^T)]\underline{x}(t)$$

***Desired Output*:** $d(t) = \begin{cases} +1 & \text{if } x(t) \text{ in class } + \\ -1 & \text{if } x(t) \text{ in class } - \end{cases}$

$\eta(t) > 0$: Learning rate

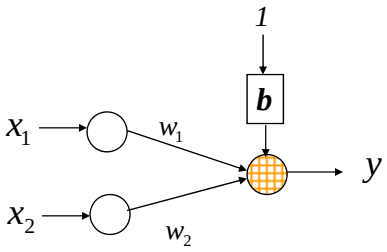
14

Preparing the Perceptron for Learning



$$\underline{x}(t) = \begin{pmatrix} 1 & x_1(t) & x_2(t) \end{pmatrix}$$

$$\underline{w}(t) = \begin{pmatrix} b(t) & w_1(t) & w_2(t) \end{pmatrix}$$



$b(t)$: **bias**

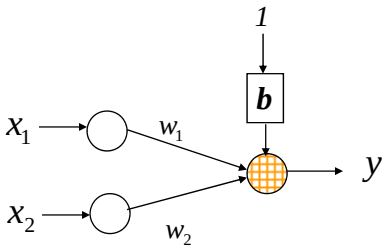
$y(t)$: **Actual Response of N.N.**

15

Preparing the Perceptron for Learning

Training Data:

$$\begin{pmatrix} \underline{x}(1), d(1) \end{pmatrix}$$
$$\begin{pmatrix} \underline{x}(2), d(2) \end{pmatrix}$$
$$\vdots$$
$$\begin{pmatrix} \underline{x}(p), d(p) \end{pmatrix}$$



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

16

Learning Algorithm

1. Initialization Set $\underline{w}(0)=rand$. Then perform the following computatio
for time step t=1,2,...

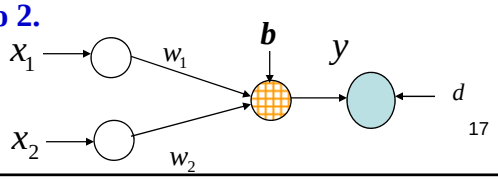
2. Activation At time step t, activate the Perceptron by applying
input vector $\underline{x}(t)$ and desired response $d(t)$

3. Computation the actual response of N.N.
Compute the actual response of the Perceptron
$y(t) = sign\ (\ \underline{w}(t) \cdot \underline{x}(t)^T\ )$

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

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

5. Continuation and return to 2.



17

Learning Algorithm

Where or When to stop?

There are two approaches to stop the learning process:

1. Converging the generalized error to a constant value.
2. Repeat the learning process for predefined number.

$$\begin{matrix}(\underline{x}(1), d(1)) \\ (\underline{x}(2), d(2)) \\ \vdots \\ (\underline{x}(p), d(p))\end{matrix} \longrightarrow G.E. = \sum_{t=1}^{p} \left[d(t) - sign\left(\underline{w}(t) \bullet \underline{x}(t)^T\right) \right]^2$$

18

Training Types

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 pattern in a period is presented

19

1st Mini Project

1. By using the perceptron learning rule generate a N.N. to represent a NOT gate.
2. By using the perceptron learning rule generate a N.N. to represent a AND gate.
3. By using the perceptron learning rule generate a N.N. to represent a OR gate.
4. Please show that the generalized error converge to constant value after a learning process.
5. Please test the above N.N.s by testing data?
6. Please check the above N. N.s with data which added to noise.
7. Repeat the learning process for above N.N.s in both with and without bias.
8. Please plot the updated weights.

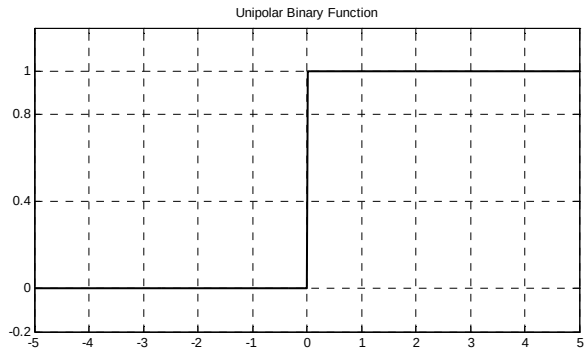
20

Artificial Neural Networks

Lecture 5

Activation Functions

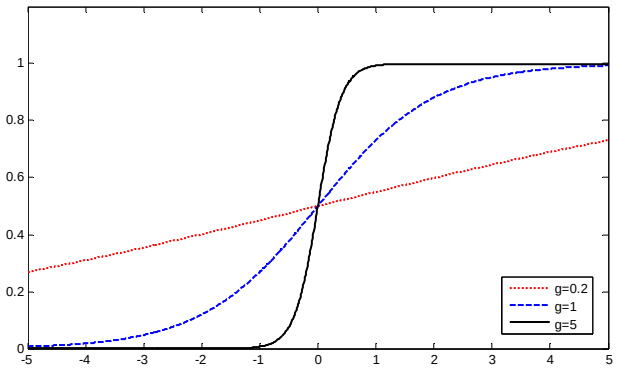
Unipolar Binary Function



$$F(u)=\begin{cases}1 & u\geq 0\\ 0 & u<0\end{cases}$$

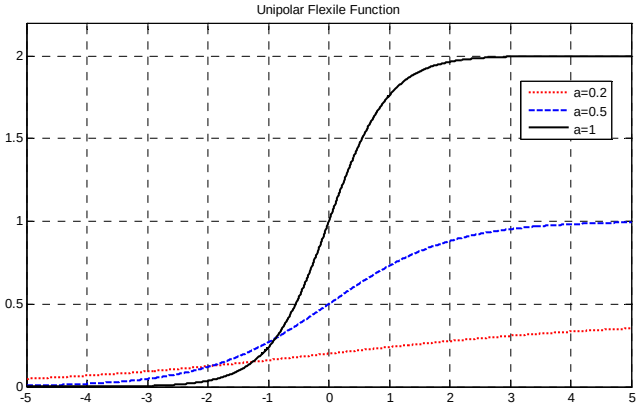
It is obvious that this activation function is not differentiable.

Unipolar Sigmoid Function



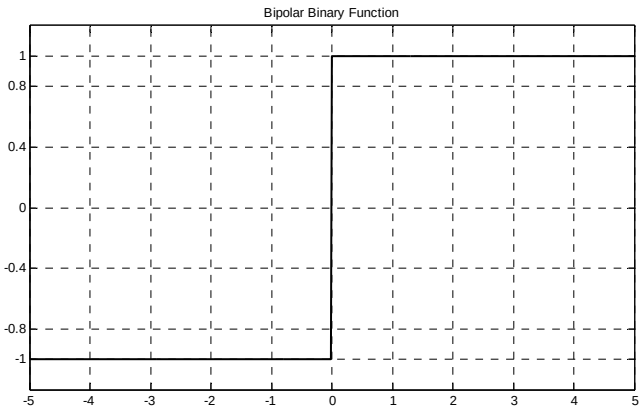
$$F(u)=\frac{1}{1+e^{-gu}},\quad g>0 \quad\Rightarrow\quad F'=\frac{\partial F}{\partial u}=gF(1-F)$$

Unipolar Flexible Function



$$F(u)=\frac{2|a|}{1+e^{-2|a|u}},\quad a>0 \quad\Rightarrow\quad \begin{cases} F'=\dfrac{\partial F}{\partial u}=F(2|a|-F) \\ F^{*}=\dfrac{\partial F}{\partial a}=\dfrac{1}{|a|}(F'u+F) \end{cases}$$

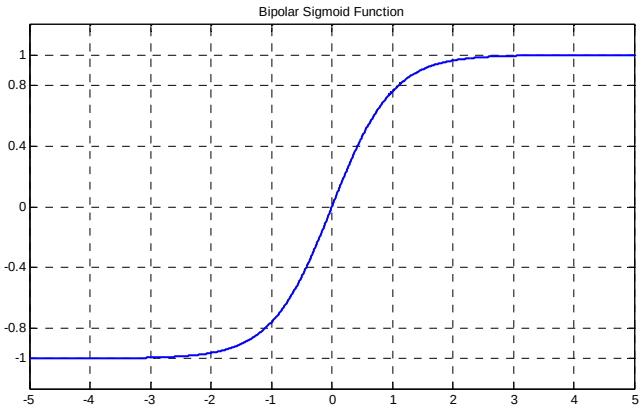
Bipolar Binary Function



$$F(u)=\begin{cases}+1 & u\geq 0\\ -1 & u<0\end{cases}$$

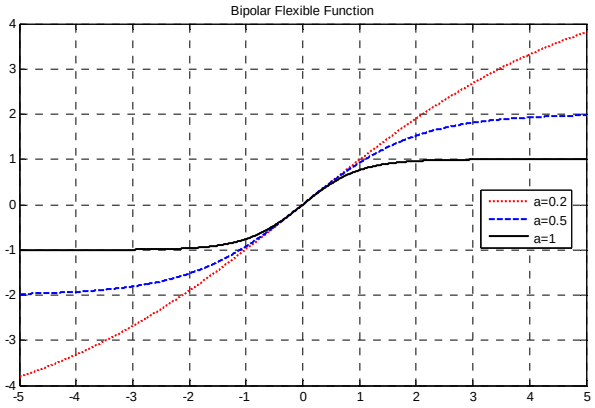
It is obvious that this activation function is not differentiable.

Bipolar Sigmoid Function



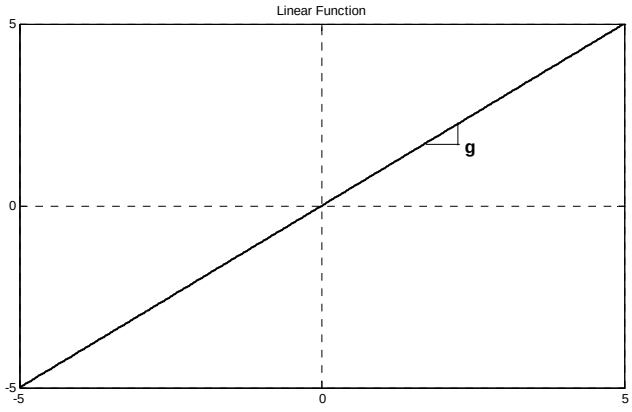
$$F(u)=\frac{1-e^{-2u}}{1+e^{-2u}} \quad \Rightarrow \quad F'=1-F^2$$

Bipolar Flexible Function



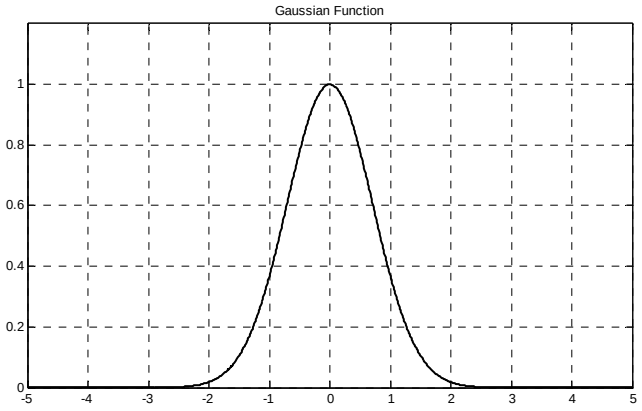
$$F(u)=\frac{1}{a}\frac{1-e^{-2au}}{1+e^{-2au}} \quad \Rightarrow \quad \begin{cases} F'=1-aF^2 \\ F^*=\dfrac{1}{a}(F'u-F) \end{cases}$$

Linear Function



$$F(u)=gu \quad \Rightarrow \quad F'=g$$

Gaussian Function



$$F = e^{-u^2} \quad \Rightarrow \quad F' = -2ue^{-u^2} < 0$$

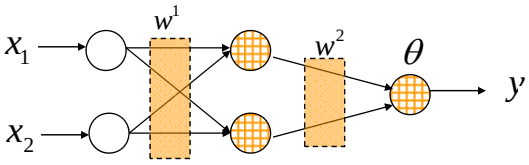
Artificial Neural Networks

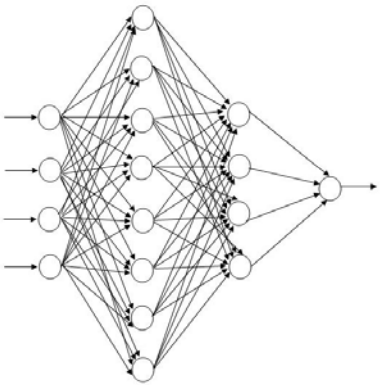
Lecture 6

Multi-layer Perceptrons

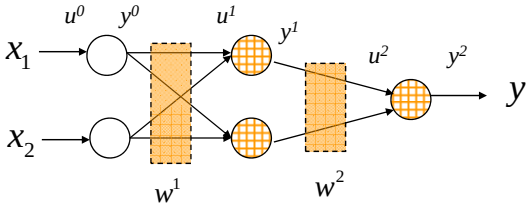
1

Multi-Layer Perceptron





Feed Forward Equations



$$u^0 = [x_1 \quad x_2]^T$$

$$y^0 = u^0$$

$$u^1 = w^1 y^0$$

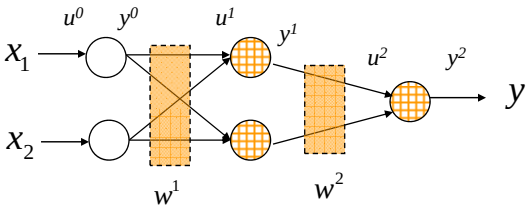
$$y^1 = f_1(u^1)$$

$$u^2 = w^2 y^1$$

$$y = y^2 = f_2(u^2)$$

3

Learning Rule?



What are the learning rules or algorithms to tune the N.N. weights?

- ***Unsupervised Learning Algorithms***
- ***Supervised Learning Algorithms***

4

Learning Rules

What is the Goal of N.N. learning?

The learning algorithm introduces an approach to achieve the zero error signal. Where, Error Signal is:

$$e(k) = d(k) - y(k)$$

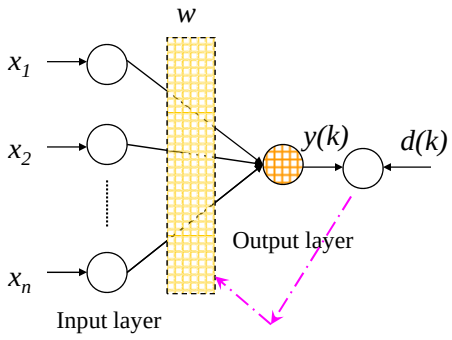
Also, the above goal can be obtained by minimizing the following cost function.

$$E = \sum_{k}^{p} \frac{1}{2} e(k)^2$$

5

Learning Rules

Which parameters have effect in optimizing the above cost function?



$$E = \sum_{k}^{p} \frac{1}{2} (d(k) - y(k))^2 = \sum_{k}^{p} \frac{1}{2} \left(d(k) - f(w(k).x^T(k))\right)^2$$

6

Learning Rules

So, the N.N. can be optimized by minimizing the corresponding cost function with respect to the synaptic weights of network.

According to above explanation, Widrow and Hoff in 1960 proposed a new method to update the weights based on *delta rule*.

$$\Delta w_j(k) = \eta e(k) x_j(k)$$

$$w_j(k+1) = w_j(k) + \eta e(k) x_j(k)$$

7

Learning Rules

Hebbian Learning rule:

Hebb's postulate of learning is the oldest and most famous of all learning rules.
His theory can be rephrased as a two-part as follows:

1. If two neurons on either side of a synapse (connection) are activated simultaneously (i.e. synchronously), then the strength of that synapse is selectively increased.

2. If two neurons on either side of a synapse are activated asynchronously, then that synapse is selectively weakened or eliminated.

8

Learning Rules

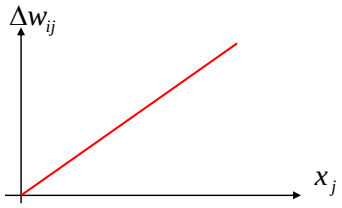
Mathematical Model of Hebbian Learning rule:

According to the Hebb's postulate the synaptic weight has relation with pre-synaptic and post-synaptic activities.

$$\Delta w_{ij}(k) = F\left(y_i(k), x_j(k)\right)$$

As a special case, we can rewrite it as follow:

$$\Delta w_{ij}(k) = \eta y_i(k) x_j(k)$$



9

Learning Rules

From this figure we can see that the repeated application of the input signal (pre-synaptic activity) x_j leads to an ***exponential growth*** that finally drives the synaptic weight into saturation.

To avoid such a situation from arising, we need to impose a limit on the growth of synaptic weights. One method for doing this is to introduce a nonlinear ***forgetting factor*** into the formula for the synaptic adjustment (Kohonen, **1988):**

$$\Delta w_{ij}(k) = \eta y_i(k) x_j(k) - \alpha y_i(k) w_{ij}(k) = \alpha y_i(k)[\tfrac{\eta}{\alpha} x_j(k) - w_{ij}(k)]$$

$$w_{ij}(k+1) = w_{ij}(k) + \Delta w_{ij}(k) = \eta y_i(k) x_j(k) + \left(1 - \alpha y_i(k)\right) w_{ij}(k)$$

$$0 < \alpha < 1$$

10

Back-Propagation Algorithm

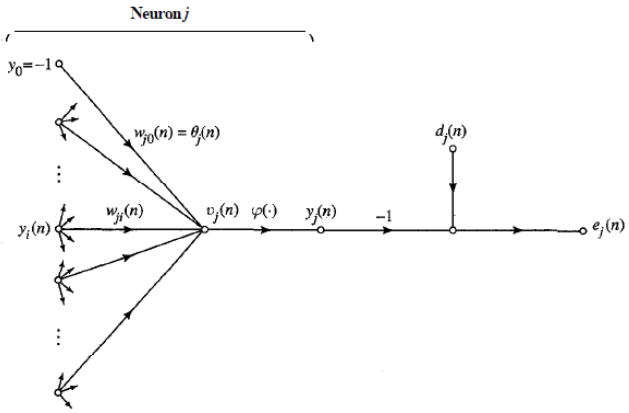
We look for a simple method of training in which the weights are updated on a ***pattern-by-pattern*** basis (online method).
The adjustments to the weights are made in accordance with the respective errors computed for ***each*** pattern presented to the network.
The arithmetic average of these individual weight changes over the training set is therefore an ***estimate*** of the true change that would result from modifying the weights based on minimizing the cost function E over the entire training set.

$$E = \frac{1}{2}\sum_{j \in O.L.} e_j^2(n)$$

Back-Propagation Algorithm

So, by using the Gradient Descent method we can introduce the following rule defined as *Delta rule* (MIT rule):

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



$$v_j(n) = \sum_i w_{ji}(n) y_i(n)$$

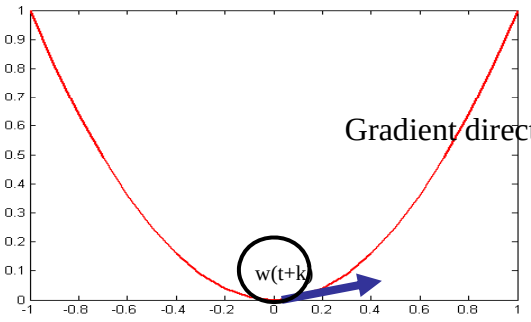
$$y_j(n) = \varphi_j(v_j(n))$$

$$e_j(n) = d_j(n) - y_j(n)$$

Back-Propagation Algorithm

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

•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.



13

Back-Propagation Algorithm

$$v_j(n) = \sum_i w_{ji}(n) y_i(n)$$

$$y_j(n) = \varphi_j(v_j(n))$$

$$e_j(n) = d_j(n) - y_j(n)$$

$$E = \frac{1}{2} \sum_{j \in O.L.} e_j^2(n)$$

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

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \frac{\partial E(n)}{\partial e_j(n)} \cdot \frac{\partial e_j(n)}{\partial y_j(n)} \cdot \frac{\partial y_j(n)}{\partial v_j(n)} \cdot \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

$$e_j(n) \quad -1 \quad \varphi'(v_j(n)) \quad y_i(n)$$

14

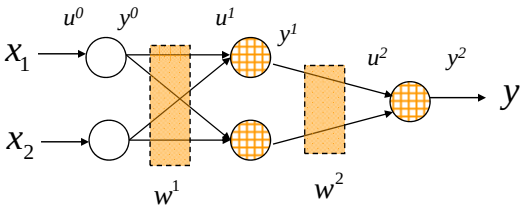
Back-Propagation Algorithm

$$\Delta w_{ji}(n) = \underbrace{\eta e_j(n)\varphi'(v_j(n))}_{\delta_j(n)} y_i(n) = \eta \delta_j(n) y_i(n)$$

$\delta_j(n)$: Local Gradient

15

Back-Propagation Algorithm

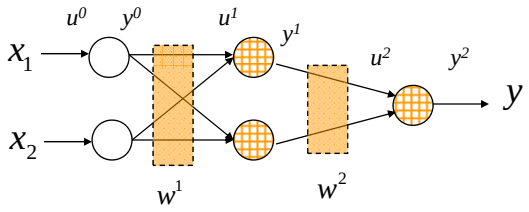


$$\Delta w^2(n) = -\eta \frac{\partial E(n)}{\partial w^2(n)} = -\eta \frac{\partial E(n)}{\partial e(n)} \cdot \frac{\partial e(n)}{\partial y^2(n)} \cdot \frac{\partial y^2(n)}{\partial u^2(n)} \cdot \frac{\partial u^2(n)}{\partial w^2(n)}$$

$$\Delta w^2(n) = \eta.e(n).f_2'(u^2(n)).y^1 \quad \longleftrightarrow \quad \begin{cases} \Delta w_1^2(n) = \eta.e(n).f_2'(u^2(n)).y_1^1 \\ \Delta w_2^2(n) = \eta.e(n).f_2'(u^2(n)).y_2^1 \end{cases}$$

16

Back-Propagation Algorithm

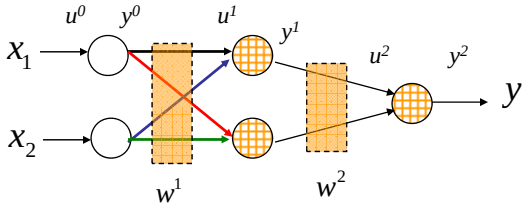


$$\Delta w^1(n) = -\eta \frac{\partial E(n)}{\partial w^1(n)} = -\eta \frac{\partial E(n)}{\partial e(n)} \cdot \frac{\partial e(n)}{\partial y^2(n)} \cdot \frac{\partial y^2(n)}{\partial u^2(n)} \cdot \frac{\partial u^2(n)}{\partial y^1(n)} \cdot \frac{\partial y^1(n)}{\partial u^1(n)} \cdot \frac{\partial u^1(n)}{\partial w^1(n)}$$

$$\Delta w^1(n) = \eta . e(n) . f_2'(u^2(n)) . w^2 . f_1'(u^1(n)) . y^0$$

17

Back-Propagation Algorithm



$$\Delta w_{11}^1(n) = \eta . e(n) . f_2'(u^2(n)) . w_1^2 . f_1'(u_1^1(n)) . y_1^0$$

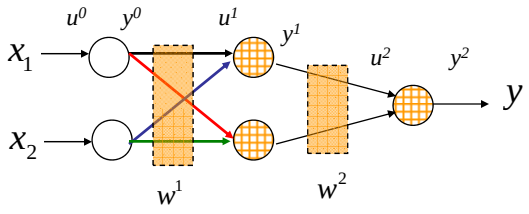
$$\Delta w_{12}^1(n) = \eta . e(n) . f_2'(u^2(n)) . w_1^2 . f_1'(u_1^1(n)) . y_2^0$$

$$\Delta w_{21}^1(n) = \eta . e(n) . f_2'(u^2(n)) . w_2^2 . f_1'(u_2^1(n)) . y_1^0$$

$$\Delta w_{22}^1(n) = \eta . e(n) . f_2'(u^2(n)) . w_2^2 . f_1'(u_2^1(n)) . y_2^0$$

18

Back-Propagation Algorithm



$$\left[\Delta w^1(n)\right]_{2\times 2} = \left[\eta\right]_{1\times 1}.\left[e(n)\right]_{1\times 1}.\left[f_2'(u^2(n))\right]_{1\times 1}.\left[w^2\right]_{2\times 1}.\left[f_1'(u^1(n))\right]_{2\times 1}.\left[y^0\right]_{1\times 2}$$

$$\left[\Delta w^1(n)\right]_{2\times 2} = \left[\eta\right]_{1\times 1}.\left[e(n)\right]_{1\times 1}.\left[f_2'(u^2(n))\right]_{1\times 1}.\left(\left[w^2\right]_{2\times 1}.*\left[f_1'(u^1(n))\right]_{2\times 1}\right).\left[y^0\right]_{1\times 2}$$

19

2nd Mini Project

1. By using the MLP and Hebbian learning rule generate a N.N. to represent the AND and XOR gates.
2. By using the MLP and Kohonen learning rule generate a N.N. to represent the AND and XOR gates.
3. By using the MLP and Back-Propagation learning rule generate a N.N. to represent the AND and XOR gates.
4. Please show that the generalized error converge to constant value after a learning process.
5. Please test the above N.N.s by testing data?
6. Please check the above N.N.s with data which are added to noise.
7. Please plot the updated weights.

20

Artificial Neural Networks

Lecture 7

Some Notes on Back-Propagation

1

Learning Rate

The smaller we make the learning-rate parameter η, the smaller will the changes to the synaptic weights in the network be from one iteration to the next and the smoother will be the trajectory in weight space.

If, on the other hand, we make the learning-rate parameter η too large so as to speed up the rate of learning, the resulting large changes in the synaptic weights assume such a form that the network may become unstable (i.e., oscillatory).

Solution: A simple method of increasing the rate of learning and yet avoiding the danger of instability is to modify the delta rule by including a *momentum* term, as shown by' (Rumelhart et al., 1986a)

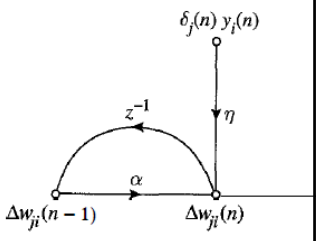
$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n) + \alpha \Delta w_{ji}(n-1)$$

2

Generalized delta-rule

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n) + \alpha \Delta w_{ji}(n-1)$$

α : momentum constant



$$\Delta w_{ji}(1) = \eta \delta_j(1) y_i(1) + \alpha \underbrace{\Delta w_{ji}(0)}_{0}$$

$$\Delta w_{ji}(2) = \eta \delta_j(2) y_i(2) + \alpha \Delta w_{ji}(1) =$$
$$= \eta \delta_j(2) y_i(2) + \alpha \eta \delta_j(1) y_i(1) \Longrightarrow \Delta w_{ji}(n) = \eta \sum_{t=0}^{n} \alpha^{n-t} \delta_j(t) y_i(t)$$

$$\Delta w_{ji}(3) = \eta \delta_j(3) y_i(3) + \alpha \Delta w_{ji}(2)$$

$$\vdots$$

$$\Delta w_{ji}(n) = -\eta \sum_{t=0}^{n} \alpha^{n-t} \frac{\partial E(t)}{\partial w_{ji}(t)}$$

3

Generalized delta-rule

$$\Delta w_{ji}(n) = -\eta \sum_{t=0}^{n} \alpha^{n-t} \frac{\partial E(t)}{\partial w_{ji}(t)}$$

Based on this relation, we may make the following insightful observations (Watrous, 1987; Jacobs, 1988; Goggin et al., 1989):

1. The current adjustment w_{ij} represents the sum of an exponentially weighted time series. For the time series to be ***convergent***, the momentum constant must be restricted to the range **0=<|α| < 1**.
   - When α is zero, the back-propagation algorithm operates without momentum.
   - Note **also** that the momentum constant α can be positive or negative, although it is unlikely that a negative α would be used in practice.

4

Generalized delta-rule

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n) + \alpha \Delta w_{ji}(n-1)$$

$$\Delta w_{ji}(n) = -\eta \sum_{t=0}^{n} \alpha^{n-t} \frac{\partial E(t)}{\partial w_{ji}(t)}$$

2. When the partial derivative has the same algebraic sign on consecutive iterations, the exponentially weighted sum Δw_{ji} grows in magnitude, and so the weight w_{ji} is adjusted by a large amount. Hence the inclusion of momentum in the back-propagation algorithm tends to ***accelerate descent*** in steady downhill directions.

3. When the partial derivative has opposite signs on consecutive iterations, the exponentially weighted sum Δw_{ji} shrinks in magnitude, and so the weight w_{ji} is adjusted by a small amount. Hence the inclusion of momentum in the back-propagation algorithm has a ***stabilizing effect*** in directions that oscillate in sign.

5

Sequential Mode and Batch Mode

Sequential Mode or Pattern Mode:

In the ***pattern mode*** of back-propagation learning, weight updating is performed after the presentation of each training data.

$$\text{an epoch}: \begin{cases} [x(1), d(1)] \\ [x(2), d(2)] \\ \vdots \\ [x(k), d(k)] \\ \vdots \\ [x(N), d(N)] \end{cases} \longrightarrow \Delta w(k) \longrightarrow w(k+1) = w(k) + \Delta w(k)$$

An Estimation

$$\Delta \hat{w}_{ji}(n) = \frac{1}{N} \sum_{k=1}^{N} \Delta w_{ji}(k) \Longrightarrow \Delta \hat{w}_{ji}(n) = -\frac{\eta}{N} \sum_{k=1}^{N} \frac{\partial E}{\partial w_{ji}}$$

$$\Longrightarrow \Delta \hat{w}_{ji}(n) = -\frac{\eta}{N} \sum_{k=1}^{N} e_j(k) \frac{\partial e_j(k)}{\partial w_{ji}(k)}$$

6

Sequential Mode and Batch Mode

Batch Mode:

In the ***batch mode*** of back-propagation learning, weight updating is performed ***after*** the presentation of *all* the training examples that constitute an epoch.

$$\text{an epoch}: \left\{ \begin{matrix} [x(1), d(1)] \\ [x(2), d(2)] \\ \vdots \\ [x(k), d(k)] \\ \vdots \\ [x(N), d(N)] \end{matrix} \right\} \longrightarrow \Delta w = \frac{1}{N}\sum_{k=1}^{N} w(k) \longrightarrow w(1) = w(0) + \Delta w$$

$$\updownarrow$$

$$E_{av} = \frac{1}{N}\sum_{k=1}^{N}\left(\frac{1}{2}\sum_{j \in O.L.} e_j^2(k)\right) = \frac{1}{2N}\sum_{k=1}^{N}\sum_{j \in O.L.} e_j^2(k)$$

$$\Delta w_{ji}(n) = -\eta \frac{\partial E_{av}}{\partial w_{ji}} \quad \Longrightarrow \quad \boxed{\Delta w_{ji}(n) = -\frac{\eta}{N}\sum_{k=1}^{N} e_j(k)\frac{\partial e_j(k)}{\partial w_{ji}(k)}}$$

7

Sequential Mode and Batch Mode

From an "on-line" operational point of view, the pattern mode of training is preferred over the batch mode, because it requires ***less local storage*** for each synaptic connection.

Moreover, given that the patterns are presented to the network in a random manner, the use of pattern-by-pattern updating of weights makes the search in weight space ***stochastic*** in nature, which, in turn, makes it less likely for the back-propagation algorithm to be trapped in a local minimum.

On the other hand, the use of batch mode of training provides a more accurate estimate of the gradient vector.

* So, the training process can be started with batch mode and then it can be changed to sequential mode.

8

Stopping Criteria

- The back-propagation algorithm is considered to have converged when the Euclidean norm of the gradient vector reaches a sufficiently small gradient threshold.

The drawback of this convergence criterion is that, for successful trials, learning times may be long.

- The back-propagation algorithm is considered to have converged when the absolute rate of change in the average squared error (ΔE_{av}) per epoch is sufficiently small.

Typically, the rate of change in the average squared error is considered to be small enough if it lies in the range of 0.1 to 1 percent per epoch; sometimes, a value as small as 0.01 percent per epoch is used.

9

Stopping Criteria

• Another useful criterion for convergence is as follows. After each learning iteration, the network is tested for its generalization performance. The learning process is stopped when the generalization performance is adequate.

10

Initializing in Back-Propagation

In Lee et al. (1991), a formula for the ***probability*** *of* ***premature saturation*** in back-propagation learning has been derived for the batch mode of updating, and it has been verified using computer simulation. The essence (core) of this formula may be summarized as follows:

1. Incorrect saturation is avoided by choosing the initial values of the synaptic weights and threshold levels **of** the network to be uniformly distributed inside a ***small*** range of values.
2. Incorrect saturation is less likely to occur ***when the number*** *of* ***hidden neurons is maintained low,*** consistent with a satisfactory operation of the network.
3. Incorrect saturation rarely occurs when the neurons of the network operate in their ***linear regions.***

Note: For pattern-by-pattern updating, computer simulation results show similar trends to the batch mode of operation referred to herein

11

Heuristics for making the Back-Propagation Algorithm Perform Better

1. A M.L.P trained with the back-propagation algorithm may, in general, learn faster (in terms of the number of training iterations required) when the **a**symmetric sigmoidal activation function are used in neuron model. than when it is non-symmetric.

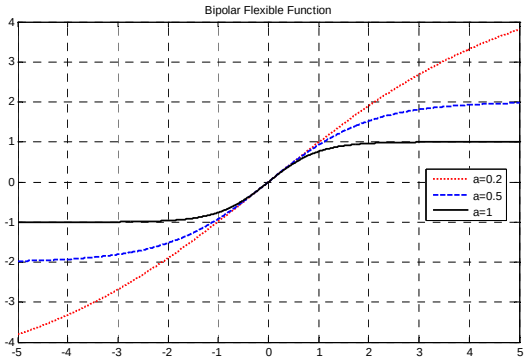
Asymmetric function: $\varphi(-v) = -\varphi(v)$

12

Heuristics for making the Back-Propagation Algorithm Perform Better

2. It is important that the **desired values** are chosen within the range of the sigmoid activation functions.
Otherwise, the back-propagation algorithm tends to drive the free parameters of the network to infinity, and thereby slow down the learning process by orders of magnitude.

$$\frac{1}{a} = d + \underbrace{e}_{offset}$$



Heuristics for making the Back-Propagation Algorithm Perform Better

3. The initialization of the synaptic weights and threshold levels of the network should be ***uniformly distributed*** inside a ***small range***. The reason for making the range small is to reduce the likelihood of the neurons in the network saturating and producing small error gradients.

However, the range should not be made too small, as it can cause the error gradients to be very small and the learning therefore to be initially very slow.

Heuristics for making the Back-Propagation Algorithm Perform Better

4. All neurons in the multilayer Perceptron should desirably learn at the same rate.
Typically, the last layers tend to have larger local gradients than the layers at the front end of the network. Hence, the learning-rate parameter η should be assigned a ***smaller value*** in the ***last layers*** than the front layers.

* Neurons with **many inputs** should have a **smaller learning-rate** parameter than neurons with few inputs.

15

Heuristics for making the Back-Propagation Algorithm Perform Better

5. For on-line operation, pattern-by-pattern updating rather than batch updating should be used for weight adjustments.

For pattern-classification problems involving a large and redundant database, pattern-by-pattern updating tends to be orders of magnitude faster than batch updating.

6. The ***order*** in which the training examples are *presented to the network* should be ***randomized*** (shuffled) from one epoch to the next. This form of randomization is critical for improving the speed of convergence.

16

Heuristics for making the Back-Propagation Algorithm Perform Better

7. Learning-rate:

In previous lectures and projects we studied the important effect of learning-rate in back-propagation learning algorithm. Here, some new methods to improve the learning-rate value is introduced.

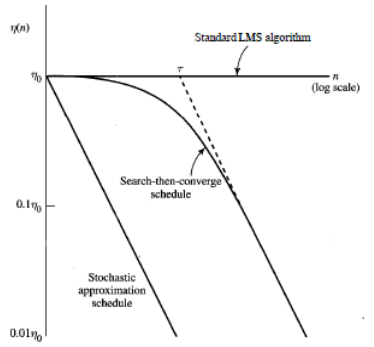
Conventional learning rate: $\eta = \eta_0$

In each iteration the learning rate value decreases (***stochastic approximation***):

$$\eta(n) = \frac{\eta_0}{n}$$

Search then converge: $\eta(n) = \frac{\eta_0}{1 + \frac{n}{\tau}}$

(Where, τ is search time constant)



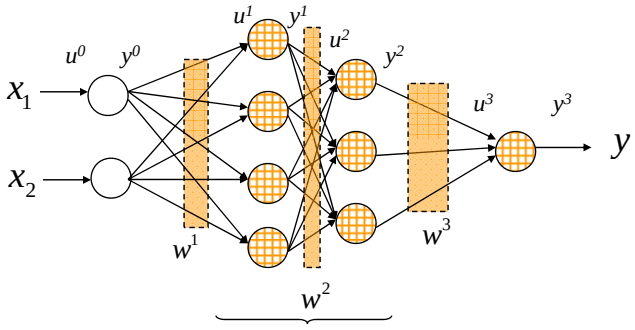
Artificial Neural Networks

Lecture 8

Flexible Neural Networks

1

Typical Multi Layer Perceptron



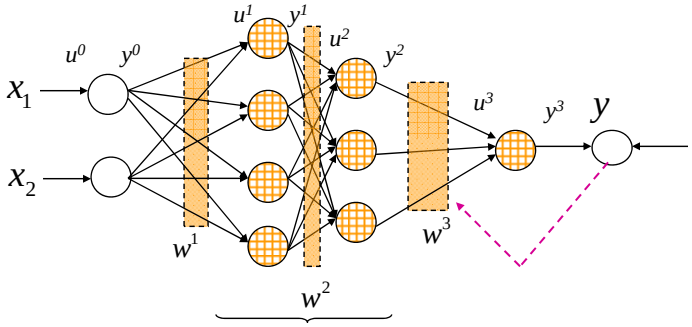
Feed-forward Equations:

$$u^0 = [x_1 \quad x_2]^T \qquad u^1 = w^1 y^0 \qquad u^2 = w^2 y^1 \qquad u^3 = w^3 y^2$$

$$y^0 = u^0 \qquad y^1 = f_1(u^1) \qquad y^2 = f_2(u^2) \qquad y^3 = f_3(u^3)$$

2

Typical Multi Layer Perceptron



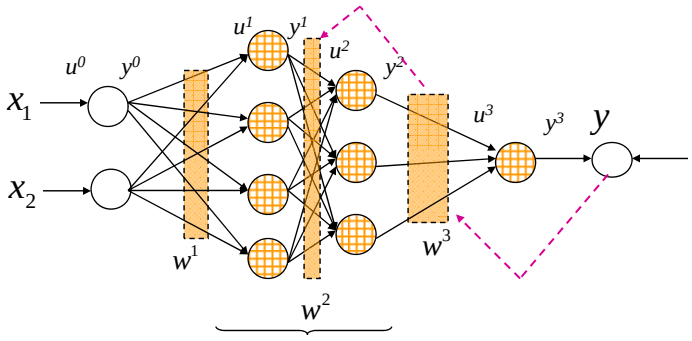
Back Propagation Equations:

$$\Delta w^3(n) = -\eta_3 \frac{\partial E(n)}{\partial w^3(n)} = -\eta_3 \frac{\partial E(n)}{\partial e(n)} \cdot \frac{\partial e(n)}{\partial y^3(n)} \cdot \frac{\partial y^3(n)}{\partial u^3(n)} \cdot \frac{\partial u^3(n)}{\partial w^3(n)}$$

$$\Delta w^3(n) = \eta_3 . e(n) . f_3'(u^3(n)) . y^2(n)$$

3

Typical Multi Layer Perceptron



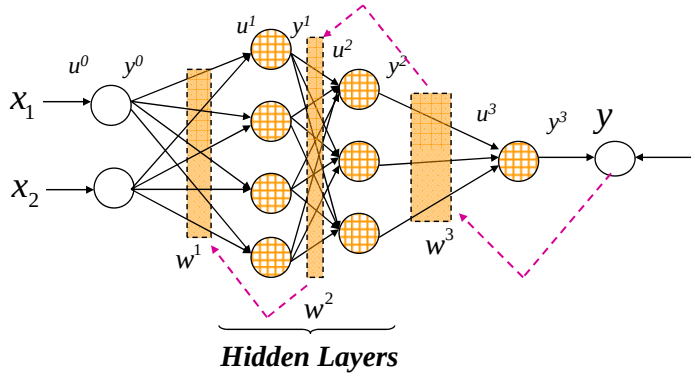
Back Propagation Equations:

$$\Delta w^2(n) = -\eta_2 \frac{\partial E(n)}{\partial w^2(n)} = -\eta_2 \frac{\partial E(n)}{\partial e(n)} \cdot \frac{\partial e(n)}{\partial y^3(n)} \cdot \frac{\partial y^3(n)}{\partial u^3(n)} \cdot \frac{\partial u^3(n)}{\partial y^2(n)} \frac{\partial y^2(n)}{\partial u^2(n)} \cdot \frac{\partial u^2(n)}{\partial w^2(n)}$$

$$\Delta w^2(n) = \eta_2 . e(n) . f_3'(u^3(n)) . w^3(n) . f_2'(u^2(n)) . y^1(n)$$

4

Typical Multi Layer Perceptron



Back Propagation Equations:

$$\Delta w^1(n) = -\eta_1 \frac{\partial E(n)}{\partial w^1(n)} = -\eta_1 \frac{\partial E(n)}{\partial e(n)} . \frac{\partial e(n)}{\partial y^3(n)} . \frac{\partial y^3(n)}{\partial u^3(n)} . \frac{\partial u^3(n)}{\partial y^2(n)} . \frac{\partial y^2(n)}{\partial u^2(n)} . \frac{\partial u^2(n)}{\partial y^1(n)} . \frac{\partial y^1(n)}{\partial u^1(n)} . \frac{\partial u^1(n)}{\partial w^1(n)}$$

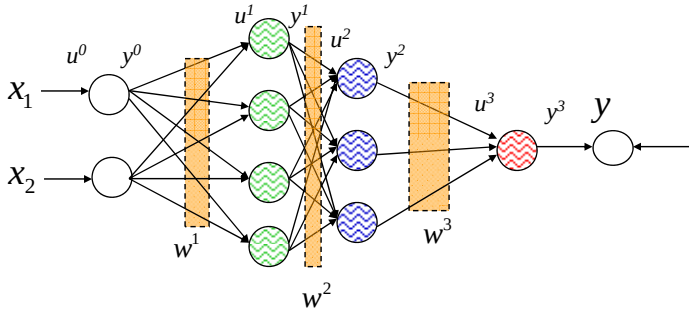
$$\Delta w^1(n) = \eta_1 . e(n) . f_3'(u^3(n)) . w^3(n) . f_2'(u^2(n)) . w^2(n) . f_1'(u^1(n)) . y^0$$

5

Flexible Neural Network

A Perceptron neural network which contains the flexible sigmoid functions in neurons is known as Flexible Neural Network.

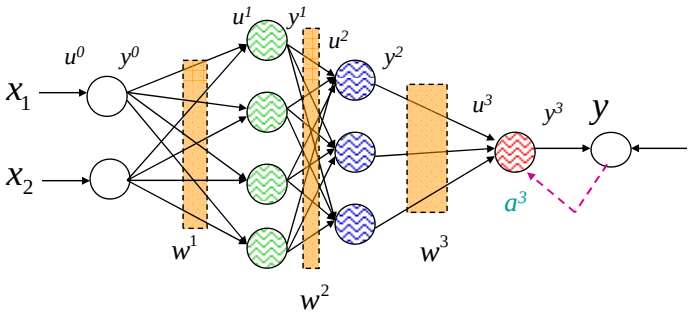
Increasing the flexibility of neural network structure induces a more efficient learning ability.



f_1, f_2 and f_3 are flexible functions.

6

Flexible Neural Network

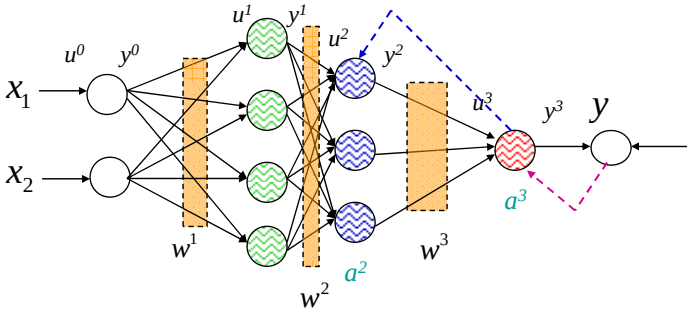


$$\Delta a^3(n) = -\eta_3' \frac{\partial E(n)}{\partial a^3(n)} = -\eta_3' \frac{\partial E(n)}{\partial e(n)} \frac{\partial e(n)}{\partial y^3(n)} \frac{\partial y^3(n)}{\partial a^3(n)}$$

$$\Delta a^3(n) = \eta_3' e(n)(f^3(u^3(n), a^3(n)))^*$$

7

Flexible Neural Network

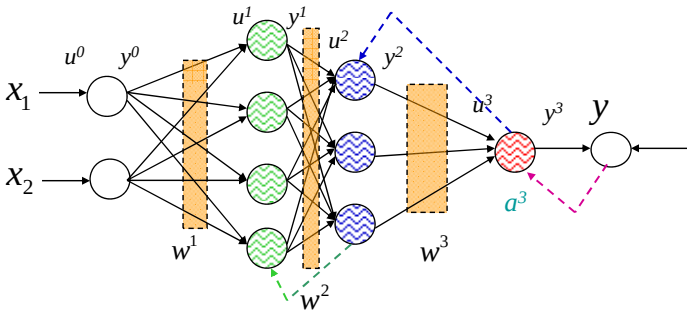


$$\Delta a^2(n) = -\eta_2' \frac{\partial E(n)}{\partial a^2(n)} = -\eta_2' \frac{\partial E(n)}{\partial e(n)} \frac{\partial e(n)}{\partial y^3(n)} \frac{\partial y^3(n)}{\partial u^3(n)} \frac{\partial u^3(n)}{\partial y^2(n)} \frac{\partial y^2(n)}{\partial a^2(n)}$$

$$\Delta a^2(n) = \eta_2' e(n) f'(u^3(n), a^3(n)).w^3(n).(f^2(u^2(n), a^2(n)))^*$$

8

Flexible Neural Network



$$\Delta a^1(n) = -\eta_1' \frac{\partial E(n)}{\partial a^1(n)} = -\eta_1' \frac{\partial E(n)}{\partial e(n)} \frac{\partial e(n)}{\partial y^3(n)} \frac{\partial y^3(n)}{\partial u^3(n)} \frac{\partial u^3(n)}{\partial y^2(n)} \frac{\partial y^2(n)}{\partial u^2(n)} \frac{\partial u^2(n)}{\partial y^1(n)} \frac{\partial y^1(n)}{\partial a^1(n)}$$

$$\Delta a^1(n) = \eta_1' e(n) f'(u^3(n), a^3(n)).w^3(n).f'(u^2(n), a^2(n)).w^2(n).(f^1(u^1(n), a^1(n)))^*$$

9

A new method to tune the learning-rate

- *Delta-bar-Delta*
  - This method is applicable to learning rates in MLP and F.MLP.

$$\eta(k) = \begin{cases} \eta(k-1) + \alpha & \delta(k-1)\delta(k) > 0 \\ b\eta(k-1) & \delta(k-1)\delta(k) < 0 \\ 0 & Otherwise \end{cases}$$

$$10^{-4} \le \alpha \le 10^{-1}$$

$$0.5 \le b \le 0.9$$

10

Artificial Neural Networks

Lecture 9

Some Applications of Neural Networks (1)

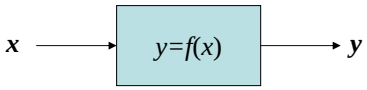
(Function Approximation)

1

Function Approximation

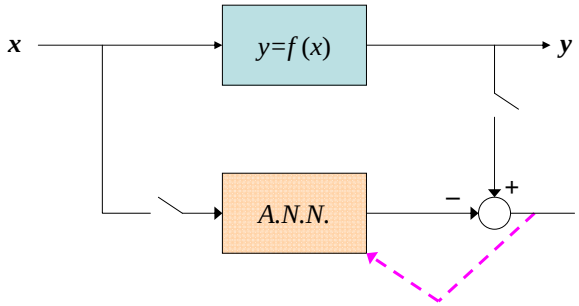
Many computational models can be described as functions mapping some numerical input vectors to numerical outputs. The outputs corresponding to some input vectors may be known from training data, but we may not know the mathematical function describing the actual process that generates the outputs from the input vectors.

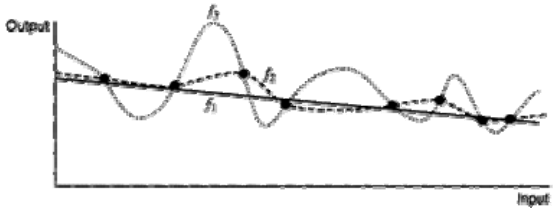
Function approximation is the task of learning or constructing a function that generates approximately the same outputs from input vectors as the process being modeled, based on available training data.



2

Function Approximation

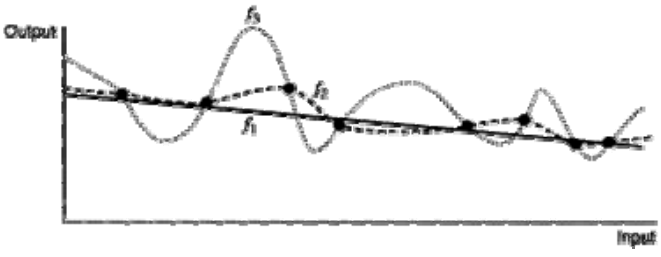




3

Function Approximation

Training Data is created of a finite set of input-output samples.



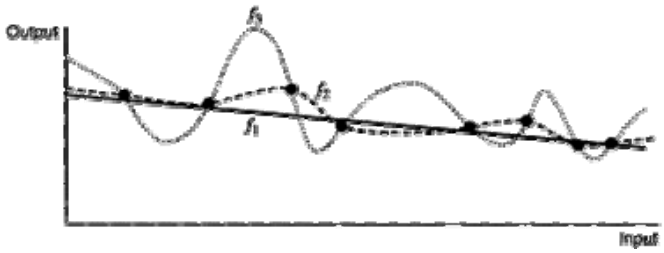
The above figure shows that the same finite set of samples can be used to obtain **many different functions**, all of which perform reasonably well on the given set of points.

4

Function Approximation

Since, infinitely many functions exist that match for a finite set of points, additional criteria are necessary to decide ***which*** of these functions are ***desirable***.

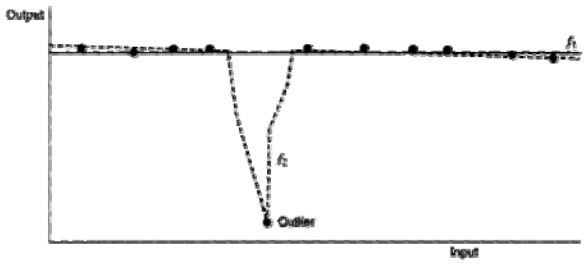
f_1, f_2 or f_3: ?



5

Function Approximation

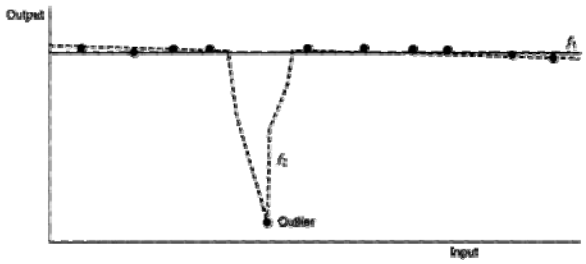
- **Continuity and smoothness** of the function are almost always required.
- Following established scientific practice, an important criterion is that of **simplicity of the model**, i.e., the neural network should have as few parameters as possible.



6

Function Approximation

• Function f_2 passes through all the points in the graph and thus performs best; but f_1,which misses the outlier, is a much simpler function and is **preferable**.

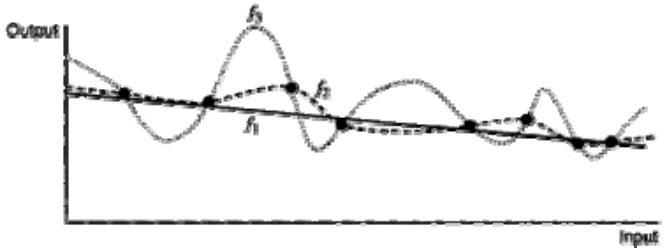


7

Function Approximation

In following figure, where the **straight line** (f_1)performs reasonably well, although f_2 and f_3 perform best in that they have zero error. Among the latter, f_2 is certainly desirable because it is **smoother** and can be represented by a network with **fewer parameters**.

• Implicit in such comparisons is the assumption that the given samples themselves might contain some errors due to the method used in obtaining them, or due to environmental factors.



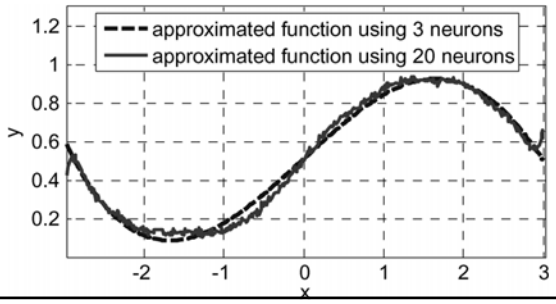
8

Function Approximation

Example 1: The desired function to be approximated is $y(x)=0.4\sin(x)+0.5$. A three-layered MLP is used as the learning prototype.

- The number of hidden neurons in these two hidden layers is set equal in the simulation.
- The training and validation data sets, containing **200** samples each, are randomly sampled from the input space, and the outputs are subjected to WGN with a standard deviation of **0.2**.

$y(x)=0.4\sin(x)+0.5$



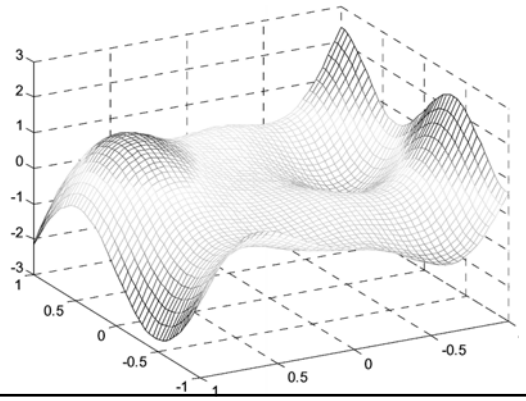
9

Function Approximation

Example 2: The desired function to be approximated is

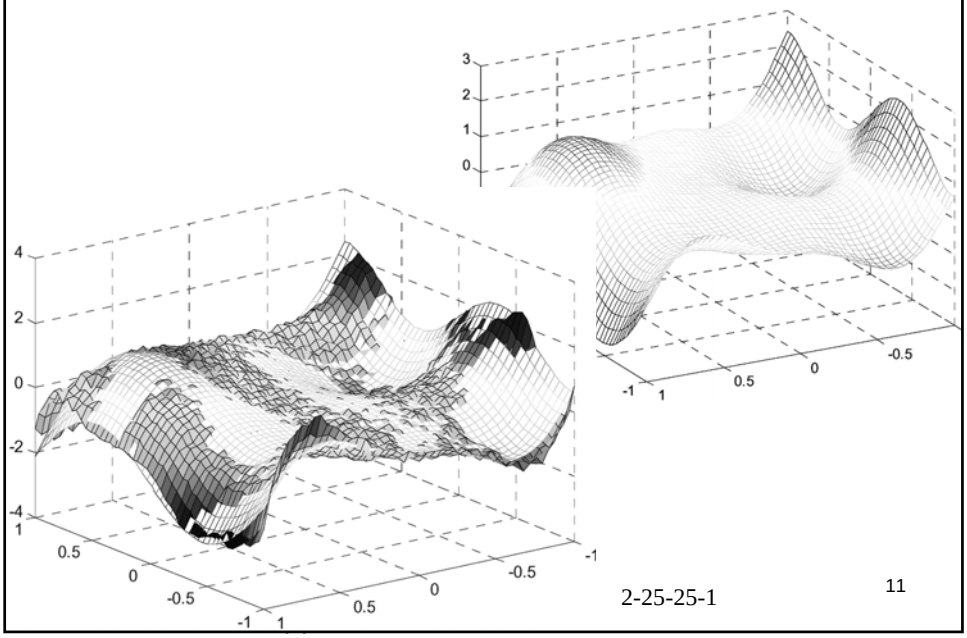
$$y(x)=x_2^2+\sin(3x_2)+2x_1^2\sin(4x_1)+x_1\sin(4x_2).$$

- Data points are randomly sampled adding WGN with a standard deviation of 0.1 to produce training and validation data sets, each containing 100 samples.



10

Function Approximation



2-25-25-1

11

Overfitting Problem

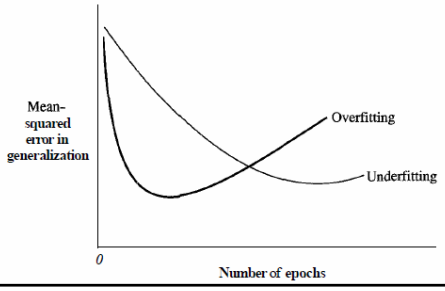
What is the **minimum number of hidden layers** in a multilayer Perceptron with an input-output mapping that provides an approximate realization of any continuous mapping ?

One curve relates to the use of few adjustable parameters (i.e., **underfitting**), and the other relates to the use of many parameters (i.e., **overfitting**).

In both cases, we usually find that

(1) the error performance on generalization exhibits a minimum point, and

(2) the minimum mean squared error for overfitting is smaller **and** better defined than that for underfitting.



Overfitting Problem

A network that is not sufficiently complex can fail to detect fully the signal in a complicated data set, leading to **underfitting**.

But, a network that is too complex may fit the noise, not just the signal, leading to **overfitting**.

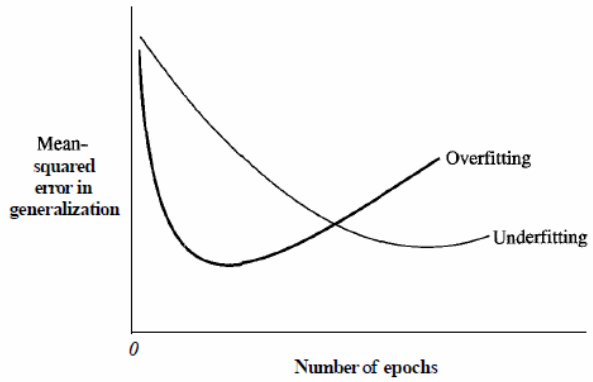
- Overfitting is especially dangerous because it can easily lead to predictions that are far beyond the range of the training data with many of the common types of NNs.

- Overfitting can also produce wild predictions in multilayer perceptrons even with noise-free data.

13

Overfitting Problem

Accordingly, we may achieve good generalization even if the neural network is designed to have too many parameters, provided that training of the network on the training set is stopped at a number of epochs corresponding to the minimum point of the error-performance curve on cross-validation.



14

Overfitting Problem

The **best way to avoid overfitting** is to **use lots of training data**.

* If you have at least 30 times as many training data as there are weights in the network, you are unlikely to suffer from much overfitting.

* For noise-free data, 5 times as many training data as weights may be sufficient.

* You can't arbitrarily reduce the number of weights due to fear of underfitting.

* Underfitting produces excessive bias in the outputs, whereas overfitting produces excessive variance.

15

3rd Mini Project

By using of an arbitrary neural network (MLP) approximate the function which is presented in example 1.

1st Part of Final Project

By using of an arbitrary neural network (MLP) approximate the function which is presented in example 2.
In this project You can use all hints which are introduced in previous lectures but, you should explain their effects (score: 2 points)

16

Artificial Neural Networks

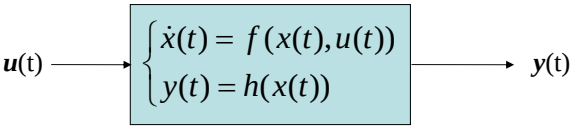
Lecture 10

Some Applications of Neural Networks (2)
(System Identification)

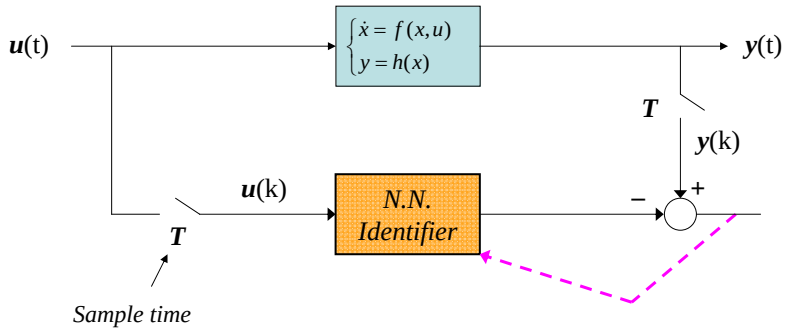
System Identification

The main objective of identification process is to propose **specific neural network architectures** that can be used for effective identification of a linear/nonlinear system using only **input-output data**.

Here, the main result is the establishment of input-output models using feedforward neural networks.



System Identification



The **supervised training** of a MLP may be viewed as a ***global nonlinear identification problem,*** the solution of which requires the ***minimization*** of ***a certain cost function.*** The cost function *E* is defined in terms of deviations (error) of the network outputs from desired outputs, and expressed as a function of the weight vector **w** representing the free parameters (i.e., synaptic weights and thresholds and) of the network.
The **goal of the training** is to adjust these free parameters so as to make the **actual outputs** of the network **match** the **desired outputs** as closely as possible

3

System Identification

Two facts make the MLP a powerful tool for approximating the functions or identifying the systems:

Multilayer feedforward neural networks are universal approximators:

It was proved by Cybenko (1989) and Hornik et al. (1989) that any continuous mapping over a compact domain can be approximated as accurately as necessary by a feedforward neural network with one hidden layer.

The back propagation algorithm:

This algorithm which performs stochastic gradient descent, provides an effective method to train a feedforward neural network to approximate a given continuous function over a compact domain.

4

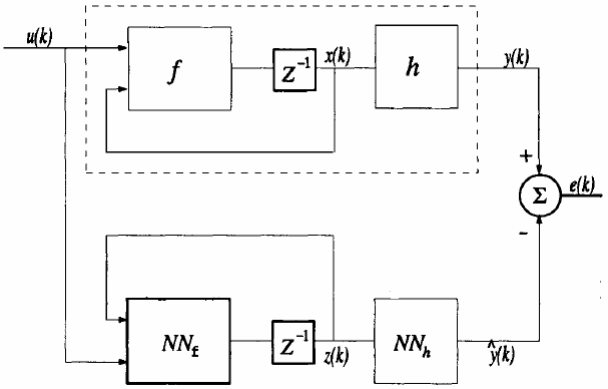
State Space model for Identification

Consider a discrete plant as:

$$x(k+1) = f(x(k), u(k))$$
$$y(k) = h(x(k))$$

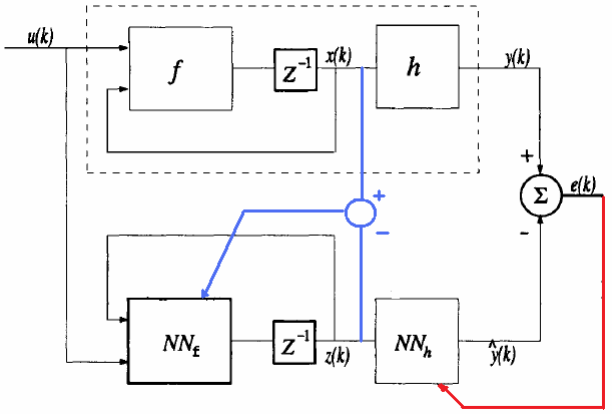
If the **state** of the system is assumed to be directly measurable, the identification model can be chosen as:

$$z(k+1) = NN_f[z(k), u(k)]$$
$$\hat{y}(k) = NN_h[(x(k)]$$



State Space model for Identification

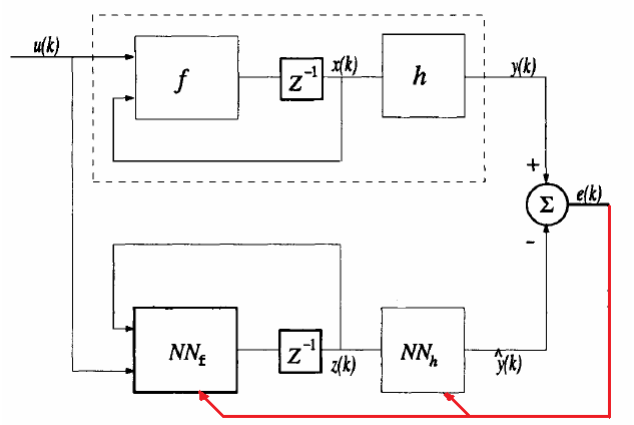
In this case, the **states** of the plant to be **identified are assumed** to be **directly accessible**, and each of the networks NN_f and NN_h can be independently trained using static learning.



6

State Space model for Identification

Since $x(k)$ is not accessible and the error can be measured only at the output, the networks cannot be trained separately. Since the model contains a feedback loop, the gradient of the performance criterion with respect to the weights of NN_f varies with time, and thus dynamic back propagation needs to be used.



7

State Space model for Identification

In this structure, the states of the N.N. model provide an approximation or estimation to the states of the system.

A natural ***performance criterion*** for the model would be the sum of the squares of the errors between the system and the model outputs:

$$E(k) = \frac{1}{2}\sum_k \left\| y(k) - \hat{y}(k) \right\|^2 = \sum_k \left\| e(k) \right\|^2$$

Dynamic Back Propagation:

$w_h \in NN_h \longrightarrow \Delta w_h = -\eta_h \frac{dE(k)}{dw_h(k)}$

$w_f \in NN_f \longrightarrow \Delta w_f = -\eta_f \frac{dE(k)}{dw_f(k)} = -\eta_f \sum_{j=1}^{n} \frac{\partial E(k)}{\partial z_j(k)} \cdot \frac{dz_j(k)}{dw_f(k)}$

$$\frac{dz_j(k)}{dw_f} = \sum_{l=1}^{n} \frac{\partial z_j(k)}{\partial z_l(k-1)} \cdot \frac{\partial z_l(k-1)}{\partial w_f} + \frac{\partial z_j(k)}{\partial w_f}$$

8

State Space model for Identification

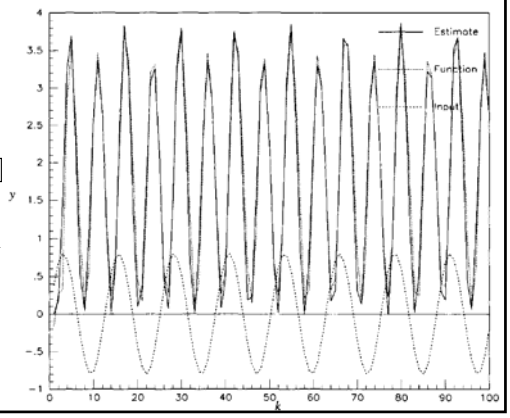
Example 1:

$$x_1(k+1) = x_2(k)\left(1+0.2u(k)\right)$$
$$x_2(k+1) = -0.2x_1(k) + 0.5x_2(k) + u(k)$$
$$y(k) = 0.3\left(x_1(k) + 2x_2(k)\right)^2$$

$$\hat{x}_1(k+1) = NN_{f1}\left[\hat{x}_1(k), \hat{x}_2(k), u(k)\right]$$
$$\hat{x}_1(k+1) = NN_{f2}\left[\hat{x}_1(k), \hat{x}_2(k), u(k)\right]$$
$$\hat{y}(k) = NN_h\left[\hat{x}_1(k), \hat{x}_2(k)\right]$$

• Training was done with a random input uniformly distributed in [—1,1]

• The identification model was tested with sinusoidal inputs



Input-Output model for Identification

Clearly, choosing the state space models for identification requires the use of dynamic back propagation, which is computationally a very **intensive procedure**. At the same time, to avoid instabilities while training, one needs to use small learning rate to adjust the parameters, and this in turn results in **long convergence times**.

Input-Output Model of plant:

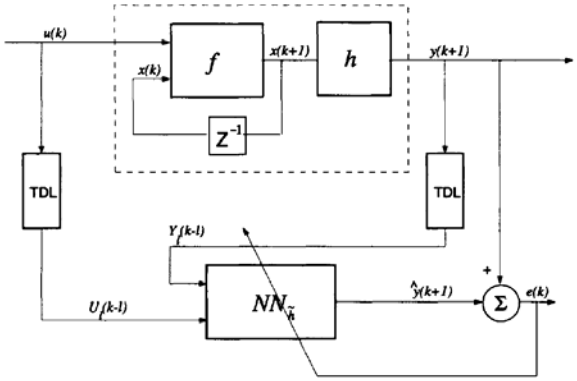
Consider the difference Equation corresponding to a typical linear plant:

$$y(k) = \sum_{i=1}^{n} a_i y(k-i) + \sum_{j=1}^{n-1} b_j u(k-j)$$

10

Input-Output model for Identification

Linear Model: $y(k) = \sum_{i=1}^{n} a_i y(k-i) + \sum_{j=1}^{n-1} b_j u(k-j)$



$$\tilde{y}(k+1) = \tilde{h}\left[Y_l(k-l+1), U_l(k-l+1)\right]$$

11

Input-Output model for Identification

$$S_{nl}\begin{cases} x(k+1) = f(x(k), u(k)) \\ y(k) = h(x(k)) \end{cases} \longrightarrow S_{linearized}\begin{cases} \delta x(k+1) = \underbrace{\left.\frac{\partial f}{\partial x}\right|_{x_0,u_0}}_{A} \delta x(k) + \underbrace{\left.\frac{\partial f}{\partial u}\right|_{x_0,u_0}}_{b} \delta u(k) \\ \delta y(k) = \underbrace{\left.\frac{\partial h}{\partial x}\right|_{x_0}}_{c} \delta x(k) \end{cases}$$

Theorem Let S_{nl} be the nonlinear system, and $S_{linearized}$ its linearization around the equilibrium point. If $S_{linearized}$ is observable, then S_{nl} is locally strongly observable. ***Furthermore, locally, S_{nl} can be realized by an input-output model.***

Observability Matrix $\varphi_o = \begin{bmatrix} c \\ cA \\ \\ cA^{n-1} \end{bmatrix}$

12

Input-Output model for Identification

***Neural Network Implementation*:**

If strong observability conditions are known (or assumed) to be satisfied in the system's region of operation with ***n* state variables**, then the identification procedure using a feedforward neural network is quite straightforward.

At each instant of time, the **inputs** to the network consisting of the **system's past *n* input** values and **past *n* output values** (all together 2n), are fed into the neural network.

The network's output is compared with the next observation of the system's output to yield the error

$$e(k+1) = y(k+1) - \tilde{y}(k+1) = y(k+1) - \tilde{h}\left[Y_l(k-n+1), U_l(k-n+1)\right]$$

The weights of the network are then adjusted using static back propagation to minimize the sum of the squared error.

13

State Space model for Identification

Example 2:

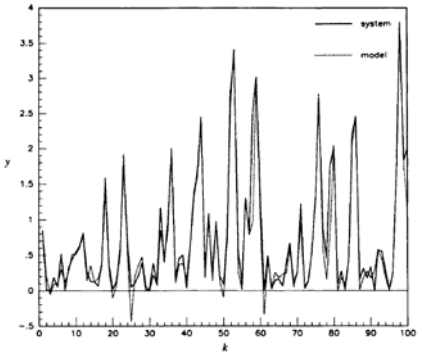
$$x_1(k+1) = 0.5x_2(k) + 0.2x_1(k)x_2(k)$$
$$x_2(k+1) = -0.3x_1(k) + 0.8x_2(k) + u(k)$$
$$y(k) = x_1(k) + (x_2(k))^2$$

The linearized system around the equilibrium point:

$$\delta x_1(k+1) = 0.5\delta x_2(k)$$
$$\delta x_2(k+1) = -0.3\delta x_1(k) + 0.8\delta x_2(k) + \delta u(k)$$
$$\delta y(k) = \delta x_1(k)$$

And its observability:

$$\varphi_o = \begin{bmatrix} 1 & 0 \\ 0 & 0.5 \end{bmatrix}: \text{ful rank}$$

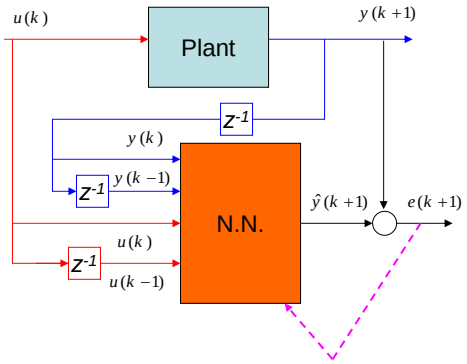


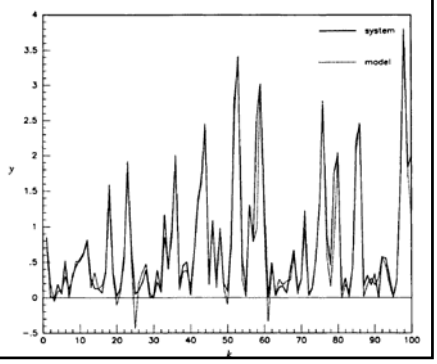
14

State Space model for Identification

* A neural network was trained to implement the model (4-12-6-1).

* The system was driven with **random input** $u(k) \in [-1,1]$

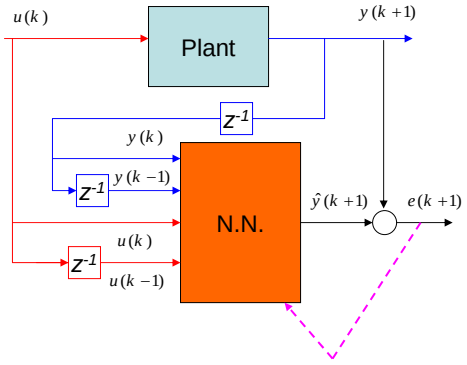


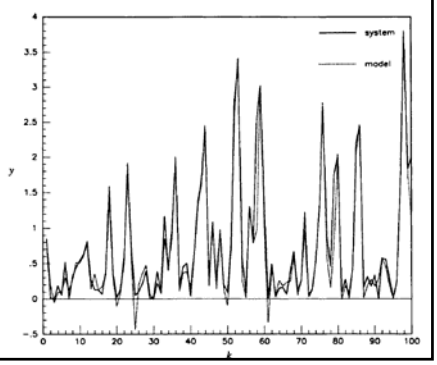


State Space model for Identification

* A neural network was trained to implement the model (4-12-6-1).

* The system was driven with **random input** $u(k) \in [-1,1]$





2nd Part of Final Project

By using of an arbitrary neural network (MLP) identify the discrete nonlinear plant which is presented in example 2 (Score: 1 points).

- By using a test signal, show that the N.N. identifier perform a appropriate input-output model of plant.
- By using of the PRBS signal, repeat the identifying procedure and compare the results.

The material of this lecture is based on:

Omid Omidvar and David L. Elliott, **Neural Systems for Control**, Academic Press; 1st edition (1997).

17

Artificial Neural Networks

Lecture 11

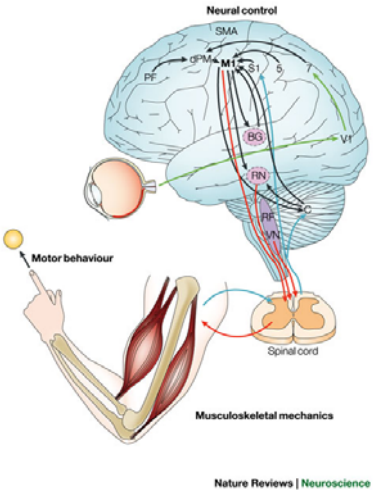
Some Applications of Neural Networks (3) (Control)

NN-based Control

One of the most important applications of N.N. is its employment in control theory.

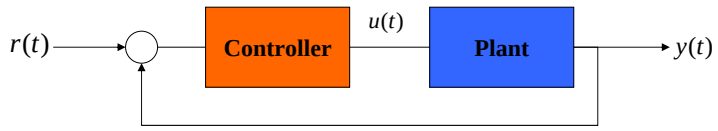
In most cases, the ordinary control theory cannot be easily applied, due to the presence of uncertainty, nonlinearity or time varying parameters in real plants.
N.N. can overcome these problems with interesting properties such as parallel processing, flexibility in structure and real time learning.

Generally, the NN-based control is called **neuromorphic** control

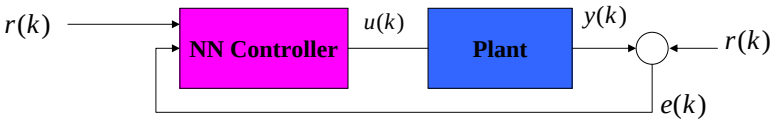


NN-based Control

Classical Control:

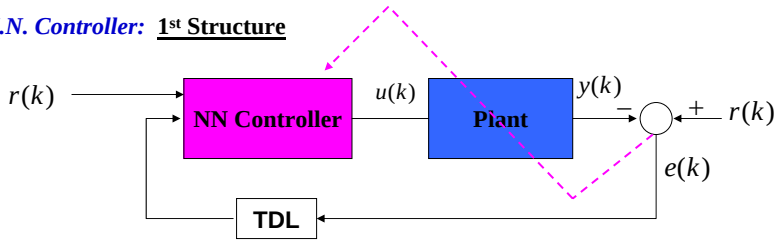


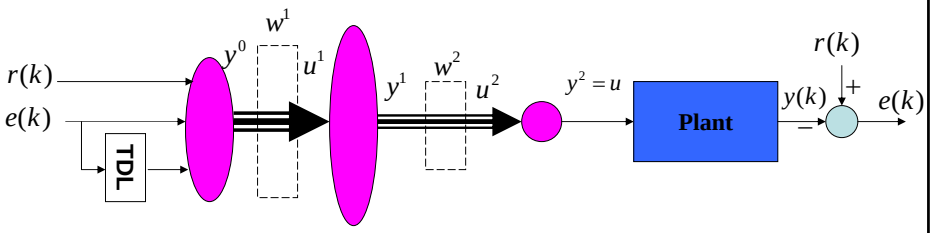
N.N. Controller: **1st Structure**



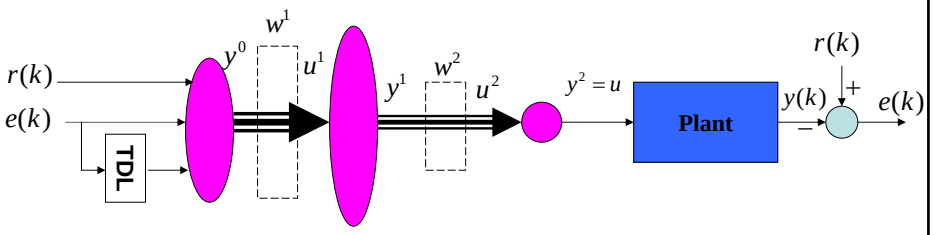
NN-based Control and Specialized learning

N.N. Controller: **1st Structure**





NN-based Control and Specialized learning



Forward Equations:

$$u^1 = w^1 y^0$$

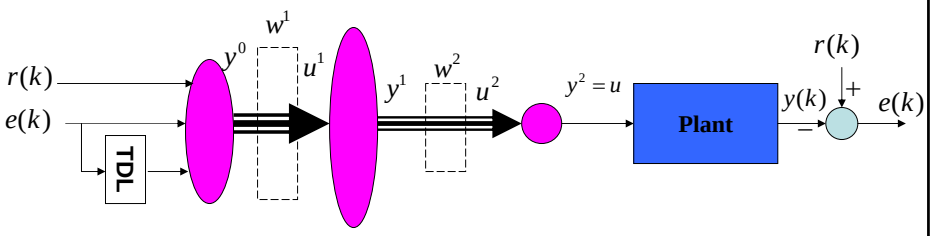
$$y^1 = f_1(u^1)$$

$$u^2 = w^2 y^1$$

$$u(k) = y^2 = f(u^2)$$

$$\begin{cases} x(k+1) = f(x(k), u(k)) \\ y(k) = h(x(k)) \end{cases}$$

NN-based Control and Specialized learning



Backward Equations: $e(k) = r(k) - y(k) \quad \longleftrightarrow \quad E = \frac{1}{2}\sum_k \|e(k)\|^2$

$$\Delta w^2(k) = -\eta \frac{\partial E(k)}{\partial w^2(k)} = \eta e \underbrace{\frac{\partial y(k)}{\partial u(k)}}_{\text{Plant Jacobi}} \frac{\partial u(k)}{\partial y^2(k)} \frac{\partial y^2(k)}{\partial u^2(k)} \frac{\partial u^2(k)}{\partial w^2(k)}$$

$$\Delta w^1(k) = -\eta \frac{\partial E(k)}{\partial w^1(k)} = \eta e \underbrace{\frac{\partial y(k)}{\partial u(k)}}_{\text{Plant Jacobi}} \frac{\partial u(k)}{\partial y^2(k)} \frac{\partial y^2(k)}{\partial u^2(k)} \frac{\partial y^1(k)}{\partial u^1(k)} \frac{\partial u^1(k)}{\partial w^1(k)}$$

Plant Jacobian Computation

Plant Jacobian Computation:

1st method:

$$J_p = \frac{\partial y(k)}{\partial u(k)} = \frac{\Delta y(k)}{\Delta u(k)} = \frac{y(k) - y(k-1)}{u(k) - u(k-1)}$$

Drawback: $u(k) \to u(k-1) \;\Rightarrow\; J_p \to \infty$

2nd method:

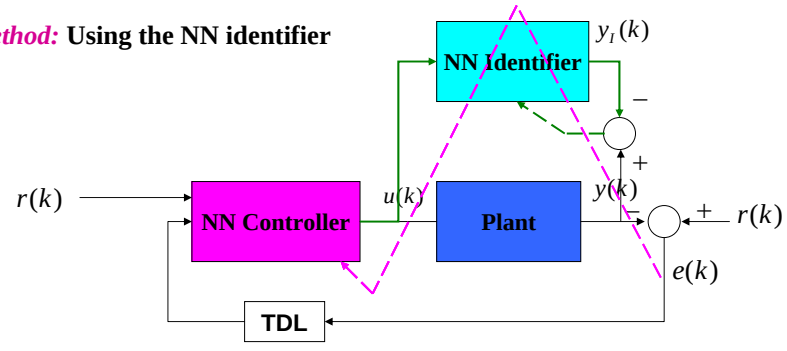
$$J_p = \frac{\partial y(k)}{\partial u(k)} = \frac{sign\left[y(k) - y(k-1)\right]}{sign\left[u(k) - u(k-1)\right]}$$

Drawback: In additional to the above drawback, this method can perform an oscillating behavior in learning process.

Plant Jacobian Computation

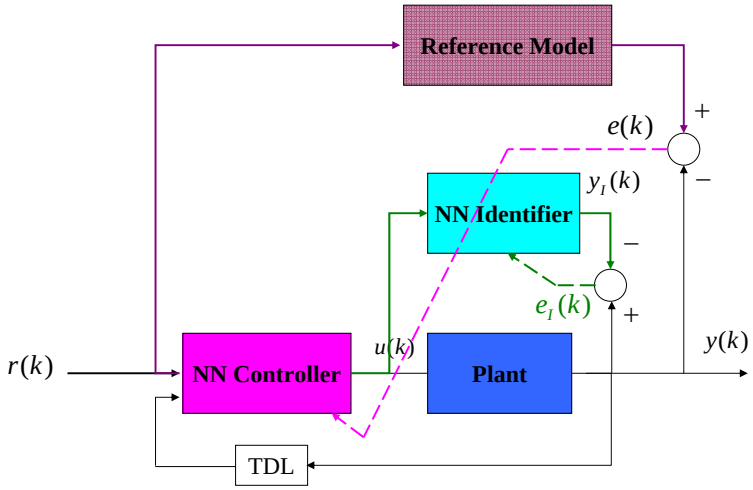
Plant Jacobian Computation:

3rd method: **Using the NN identifier**



$$J_p = \frac{\partial y(k)}{\partial u(k)} = \frac{\partial y_I(k)}{\partial u(k)}$$

Model Reference N.N. Adaptive Control



Note: This method is useful when you can realize the desire performance as a Reference model.

Self Tuning PID Control

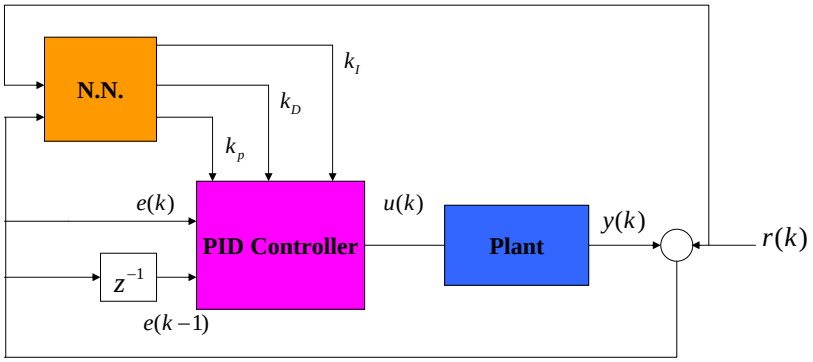
PID controller has been widely used in industry. The discrete time PID controller usually has a structure described by the following equation:

$$u(k) = k_p e(k) + k_D \left[e(k) - e(k-1) \right] + k_I z(k)$$

$$z(k) = z(k-1) + e(k)$$

The conventional PID controller cannot be useful in deal with uncertain, nonlinear and/or time varying plants. So, the self tuning PID controller can be proposed to tackle this crucial problem due to the real time parameters adjustment.

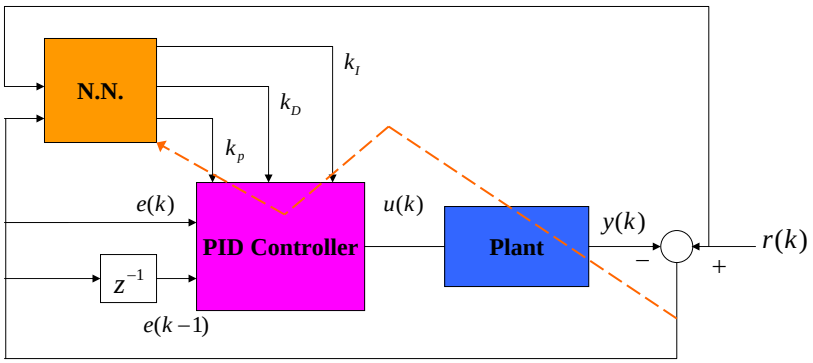
Self Tuning PID Control



$$u(k) = k_p e(k) + k_D \left[e(k) - e(k-1) \right] + k_I z(k)$$

$$z(k) = z(k-1) + e(k)$$

Learning in Self Tuning PID Control



$$\Delta k_p = -\eta \frac{\partial E}{\partial k_p} = \eta e \frac{\partial y(k)}{\partial u(k)} . \frac{\partial u(k)}{\partial k_p(k)} = \eta . e(k) . J_p . e(k)$$

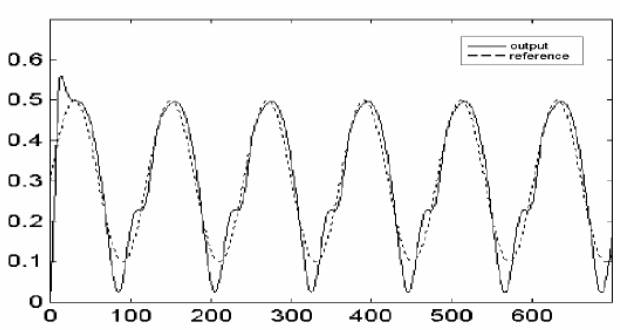
$$\Delta k_D = -\eta \frac{\partial E}{\partial k_D} = \eta e \frac{\partial y(k)}{\partial u(k)} . \frac{\partial u(k)}{\partial k_D(k)} = \eta . e(k) . J_p . \left[e(k) - e(k-1) \right]$$

$$\Delta k_I = -\eta \frac{\partial E}{\partial k_I} = \eta e \frac{\partial y(k)}{\partial u(k)} . \frac{\partial u(k)}{\partial k_I(k)} = \eta . e(k) . J_p . z(k)$$

Self Tuning PID Control

Example: Consider a servo model of the robot manipulator with following dynamic equation:

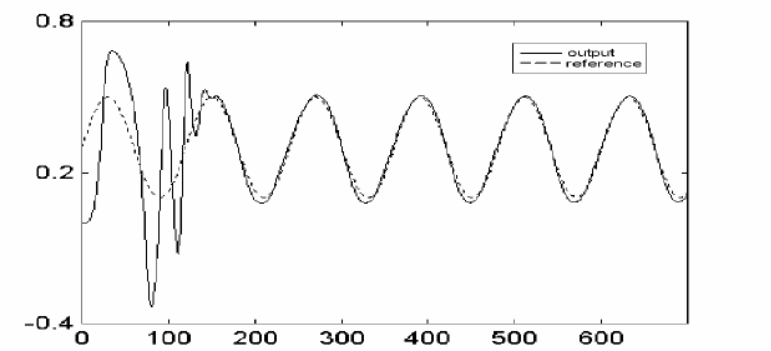
$$y(k) = 0.2\left[y^2(k-2) + y(k-1) \right] + 0.25\left[y(k-2) + u(k-1) \right] + 0.225\sin\left(y(k-1) + y(k-2) \right)$$



Output responses using conventional PID (Reference input is a sine wave)

Self Tuning PID Control

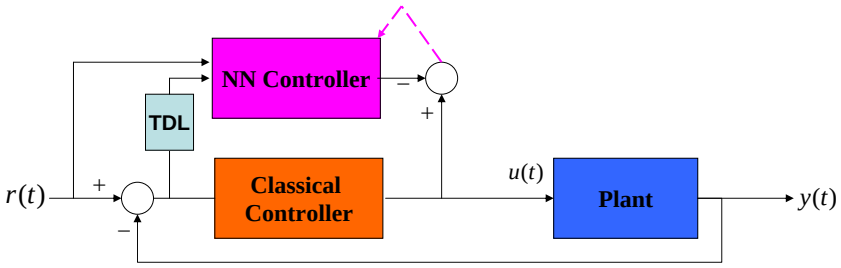
$$y(k) = 0.2\left[y^2(k-2) + y(k-1) \right] + 0.25\left[y(k-2) + u(k-1) \right] + 0.225\sin\left(y(k-1) + y(k-2) \right)$$



Output responses using self tuning PID (Reference input is a sine wave)

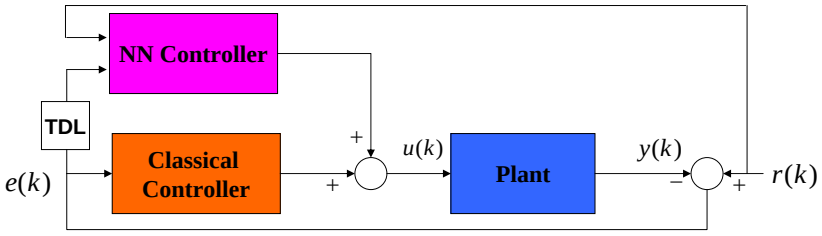
A Reliable Structure for Control

1st Step: Free parameters of the NN controller can be adjusted using the identification architecture:



A Reliable Structure for Control

2nd Step: In this step, both the classical controller and the NN controller produce the control effort signal.
Free parameters of the NN controller, which are adjusted in step 1, should be adjusted again by employing the Specialized learning.



3rd Step: You can smoothly remove the classical controller, when the closed loop control system performance is sufficiently suitable.

3rd Part of Final Project

In this project, you should find a practical plant in papers and by using of NN controllers provide a suitable closed loop control performance. (Score: 2 points)

- In this project you can use of any NN controllers structure which are presented in this lecture.
- In this project you can use of any NN controllers which are introduced in papers and text books (score: +1 point).

Literature Cited

The material of this lecture is based on:

[1] M. Teshnehlab, K. Watanabe, **Intelligent Control based on Flexible Neural Network**, Springer; 1 edition, **1999**.

[2] Woo-yong Han, Jin-wook Han, Chang-goo Lee, *Development of a Self-tuning PID Controller based on Neural Network for Nonlinear Systems*, In: Proc. of the 7th Mediterranean Conference on Control and Automation (MED99) June 28-30, **1999.**

Artificial Neural Networks

Lecture 12

Recurrent Neural Networks

Recurrent Neural Networks

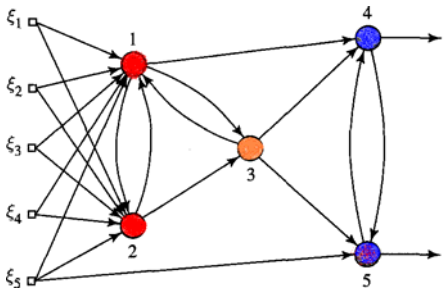
The **conventional feedforward neural networks** can be used to approximate ***any* spatiality finite function**. That is, for functions which have a *fixed* input space there is always a way of encoding these functions as neural networks.
For example in function approximation, we can use the automatic learning techniques such as backpropagation to find the weights of the network if sufficient samples from the function is available.

Recurrent neural networks are fundamentally different from feedforward architectures in the sense that they not only operate on an input space but also on an **internal *state* space**.

These are proposed to learn sequential or time varying patterns.

Recurrent Neural Networks

Recurrent Neural Networks, unlike the feed-forward neural networks, contain the feedback connections among the neurons.



Three subsets of neurons are presented in the recurrent networks:

1. ***Input neurons***
2. ***Output neurons***
3. ***Hidden neurons,*** which are neither input nor output neurons.

Note that a neuron can be simultaneously an input and output neuron; such neurons are said to be ***autoassociative.***

Recurrent Neural Networks

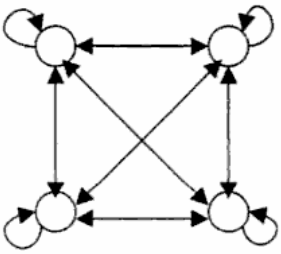


Figure 1. An example of a fully connected recurrent neural network.

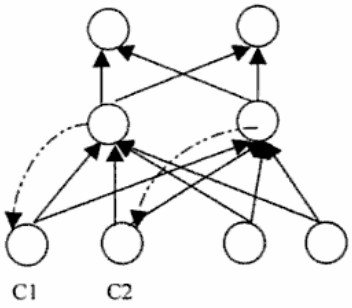
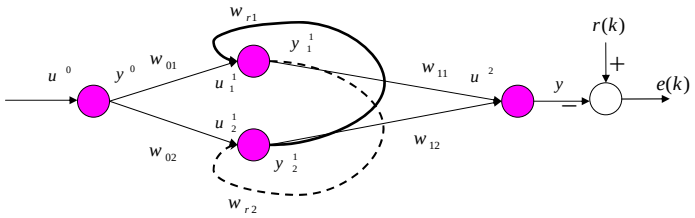


Figure 2. An example of a simple recurrent network.

Recurrent Neural Networks



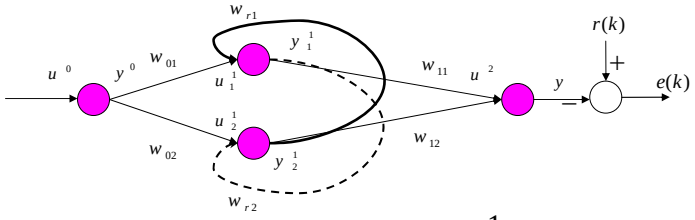
Forward Equations:

$$y^0(k) = u^0(k)$$

$$u^1(k) = \begin{bmatrix} w_{01}(k)y_0(k) + w_{r1}(k)y_2^1(k-1) \\ w_{02}(k)y_0(k) + w_{r2}(k)y_1^1(k-1) \end{bmatrix} \qquad y^1(k) = f_1(u^1(k))$$

$$u^2(k) = w_{11}(k)y_1^1(k) + w_{12}(k)y_2^1(k) \qquad y(k) = f(u^2(k))$$

Recurrent Neural Networks



$$e(k) = r(k) - y(k) \quad \longleftrightarrow \quad E = \frac{1}{2}\sum_k \|e(k)\|^2$$

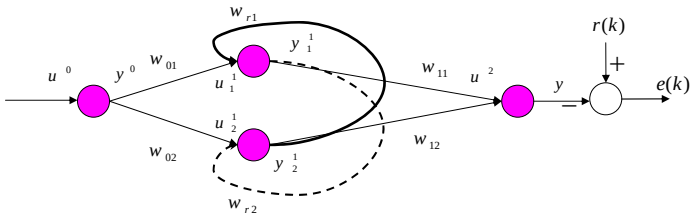
Back Propagation Equations:

$$\Delta w_{1*} = -\eta \frac{\partial E}{\partial w_{1*}}$$

$$\Delta w_{0*} = -\eta \frac{\partial E}{\partial w_{0*}} = \eta e(k) \frac{\partial y}{\partial u^2} \frac{\partial u^2}{\partial y^1} \frac{\partial y^1}{\partial u^1} \frac{\partial u^1}{\partial w_{0*}}$$

$$\frac{\partial u^1}{\partial w_{0*}} = y^0 + w_{r*} \frac{\partial y_*^1}{\partial w_{0*}} \longrightarrow \textit{Dynamic Back-propagation}$$

Recurrent Neural Networks



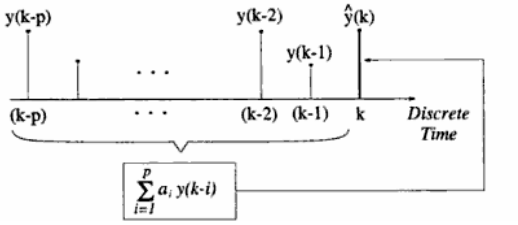
Back Propagation Equations:

$$\Delta w_{r*} = -\eta \frac{\partial E}{\partial w_{r*}} = \eta e(k) \frac{\partial y}{\partial u^2} \frac{\partial u^2}{\partial y^1} \frac{\partial y^1}{\partial u^1} \frac{\partial u^1}{\partial w_{r*}}$$

$$\frac{\partial u^1}{\partial w_{r1}} = y_2^1 + w_{r1} \frac{\partial y_2^1}{\partial w_{r1}} \qquad \frac{\partial u^1}{\partial w_{r2}} = y_1^1 + w_{r1} \frac{\partial y_1^1}{\partial w_{r1}}$$

Linear Prediction

Linear Prediction:



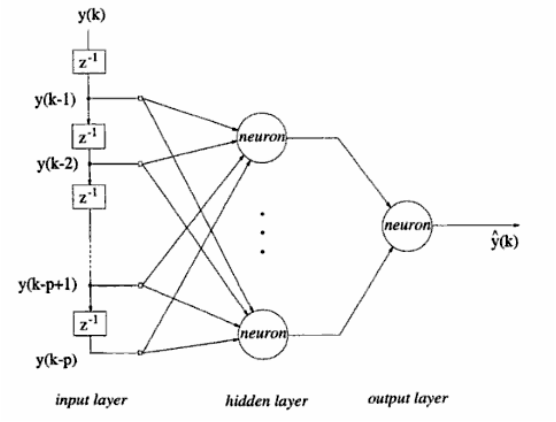
$$\hat{y}(k) = \sum_{i=1}^{p} a_i\, y(k-i)$$

$$e(k) = y(k) - \hat{y}(k) = y(k) - \sum_{i=1}^{p} a_i\, y(k-i)$$

The estimation of the parameters a_i is based on minimizing a function of error.

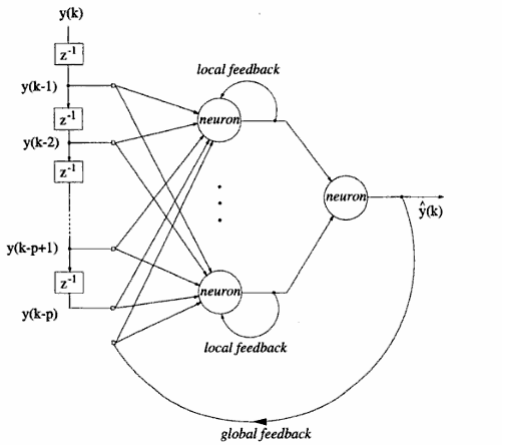
Prediction using FF Neural Network

F.F. Neural Network structure for Prediction:



Prediction using Recurrent N. N.

Recurrent Neural Network architecture for Prediction:



Example for one step ahead Prediction

$$\underbrace{y(k-n),\quad y(k-n+1),\quad \cdots,\quad y(k-n+p-1)}_{\downarrow\ \hat{y}(k-n+p)},\quad y(k-n+p)$$

Window size: **P**

$$e(k-n+p) = y(k-n+p) - \hat{y}(k-n+p)$$

It is used for back-propagatoin.

$$y(k-n),\quad \underbrace{y(k-n+1)\quad \cdots\quad y(k-n+p-1)\quad y(k-n+p)}_{\downarrow\ \hat{y}(k-n+p+1)},\quad y(k-n+p+1)$$

$$e(k-n+p+1) = y(k-n+p+1) - \hat{y}(k-n+p+1)$$

Example for one step ahead Prediction

$$y(k-n),\ \cdots,\ y(k-p-1),\ \underbrace{y(k-p),\quad \cdots,\quad y(k-2),\quad y(k-1)}_{\downarrow\ \hat{y}(k)}\quad x = ?$$

$$\Longrightarrow \quad x = \hat{y}(k)$$

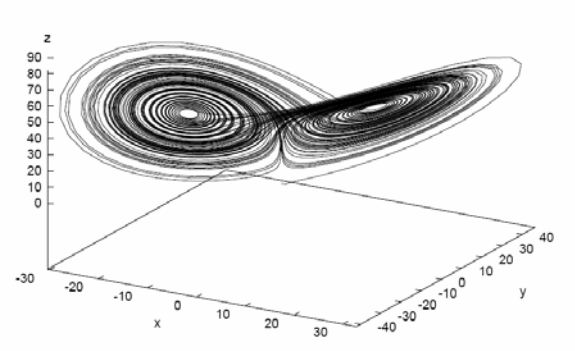
4th Mini Project

In this project, a typical time series like the Lorenz data should be employed to one step ahead prediction by using of any neural network.

Time step = 0.01
Window size = 5

$$\begin{cases} \dot{x} = \sigma(y - x) \\ \dot{y} = -xz + r(x - y) \\ \dot{z} = xy - bz \end{cases}$$

$r = 45.92,\ b = 4,\ \sigma = 16.5$



Literature Cited

The material of this lecture is based on:

[1] Mikael Boden. *A guide to recurrent neural networks and backpropagation*, Halmstad University, 2001.

[2] Danilo P. Mandic, Jonathon A. Chambers, **Recurrent neural networks for prediction: learning algorithms, architectures**, 2001.

[3] R.J.Frank, N.Davey, S.P.Hunt, *Time Series Prediction and Neural Networks*, Department of Computer Science, University of Hertfordshire, Hatfield, UK.

Artificial Neural Networks

Lecture 13

RBF Networks

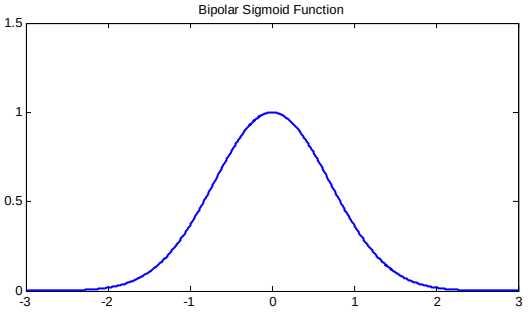
Radial Basis Function

Radial functions are a special class of function. Their characteristic feature is that their **response** decreases (or increases) monotonically with distance from a central point.

$$y_i = \varphi(\|x - c_i\|)$$

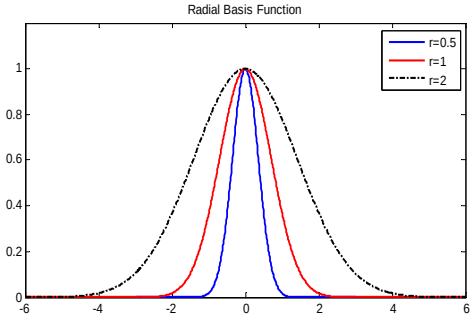
A typical radial function is the **Gaussian** which in the case of a scalar input is

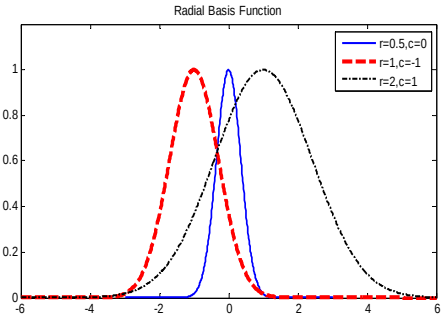
$$y_i = e^{-(\frac{x-c_i}{r})^2}$$



Gaussian RBF

$$y_i = e^{-(\frac{x-c_i}{r})^2}$$





A Gaussian RBF monotonically decreases with distance from the center.

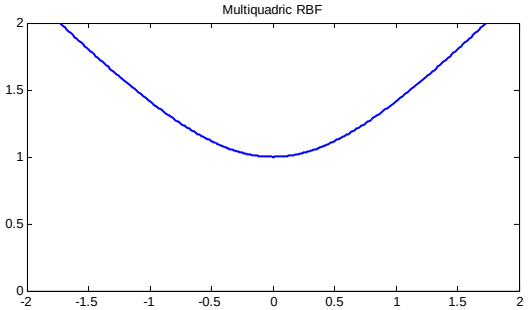
Multiquadric RBF

$$y_i = \frac{\sqrt{r^2 + (x - c_i)^2}}{r}$$

A Multiquadric RBF monotonically increases with distance from the center.

$c_i = 0$

$r = 1$



General RBFs

The most general formula for any radial basis function RBF is:

$$h(\mathbf{x}_{m*1}) = \varphi\left[(\mathbf{x}-\mathbf{c})^T \mathbf{R}^{-1}(\mathbf{x}-\mathbf{c})\right]$$ Often, $\mathbf{R}=r^2 I$.

Obviously, $(\mathbf{x}-\mathbf{c})^T\mathbf{R}^{-1}(\mathbf{x}-\mathbf{c})$ is the distance between the input $\mathbf{x}$ and the center $\mathbf{c}$ in the metric defined by R.

There are several common types of functions used:

The Gaussian: $\varphi(z) = e^{-z}$

The Multiquadric: $\varphi(z) = (1+z)^{0.5}$

The invers Multiquadric: $\varphi(z) = (1+z)^{-0.5}$

The Cauchy: $\varphi(z) = (1+z)^{-1}$

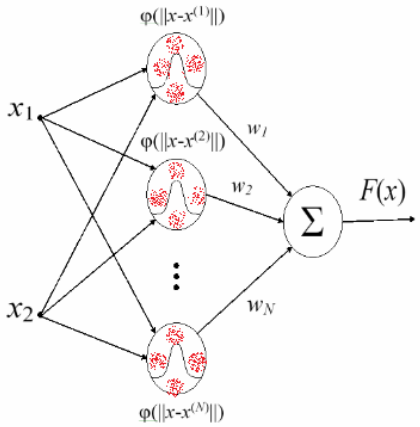
RBF Networks

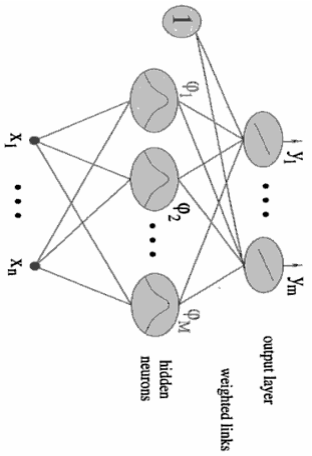
After the FF networks, the radial basis function (RBF) network comprises one of the most used network models.

The construction of a *radial-basis function* ***(RBF)*** *network* in its most basic form **involves three entirely different layers**.
The **input** layer is made up of source nodes (sensory units).
The ***second layer*** is a ***hidden layer*** of high enough dimension, which serve a different purpose from that in a MLP.
The **output layer** supplies the response of the network to the activation patterns applied to the input layer.

RBF Networks

The transformation from the input space to the hidden-unit space is ***nonlinear***, whereas the transformation from the hidden-unit space to the output space is ***linear***.

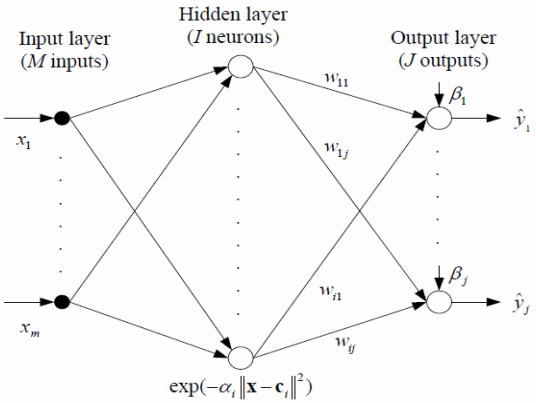




RBF Networks

In a RBF network there are three types of parameters that need to be chosen to ***adapt*** the network for a particular task:

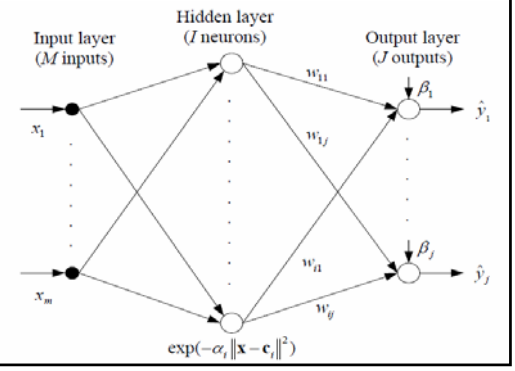
1. the center vectors c_i
2. the output weights w_{ij}, β_j
3. the RBF width parameters r_i.



RBF Networks

Characteristics of a typical RBF neural network:

I Number of neurons in the hidden layer $i \in \{1,2,..., I\}$
J Number of neurons in the output layer $j \in \{1,2,..., J\}$
w_{ij} Weight of the ith neuron and jth output
φ Radial basis function
α_i Spread parameter of the ith neuron
$\mathbf{x}$ Input data vector
$\mathbf{c}_i$ Center vector of the ith neuron
β_j Bias value of the output jth neuron
$\hat{y}_j$ Network output of the jth neuron

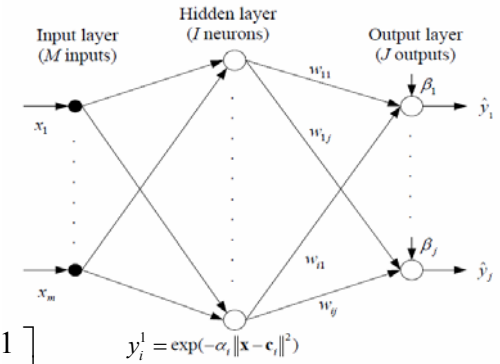


Training the RBF Networks

Feedforward equations of a typical RBF neural network:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

$$y_i^1 = e^{-\left(\frac{\mathbf{x}-\mathbf{c}_i}{r_i}\right)^2} = e^{-\alpha_i(\mathbf{x}-\mathbf{c}_i)^2}$$

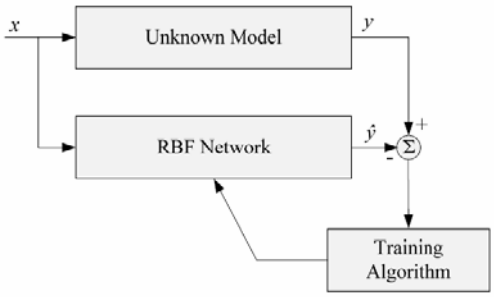


$$\hat{y} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_J \end{bmatrix} = \begin{bmatrix} \beta_1 & w_{11} & \cdots & w_{1J} \\ \beta_2 & w_{21} & \cdots & w_{2J} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_J & w_{I1} & \cdots & w_{IJ} \end{bmatrix} \begin{bmatrix} 1 \\ y_1^1 \\ \vdots \\ y_I^1 \end{bmatrix}$$

Training the RBF Networks

Back-propagation:

$$e(k) = y(k) - \hat{y}(k) \quad \rightarrow \quad E = \frac{1}{2}\sum_k e^2(k)$$



$$\Delta w_{ij}(k) = -\eta_w \frac{\partial E}{\partial w_{ij}} = \eta_w e(k) \frac{\partial \hat{y}(k)}{\partial w_{ij}}$$

$$\Delta c_i(k) = -\eta_c \frac{\partial E}{\partial c_i} = \eta_w e(k) \frac{\partial \hat{y}(k)}{\partial y_i^1} \frac{\partial y_i^1(k)}{\partial c_i}$$

$$\Delta \alpha_i(k) = -\eta_\alpha \frac{\partial E}{\partial \alpha_i} = \eta_w e(k) \frac{\partial \hat{y}(k)}{\partial y_i^1} \frac{\partial y_i^1(k)}{\partial \alpha_i}$$

Training the RBF Networks

Back-propagation:

$$y^1 = e^{-\underbrace{(\mathbf{x}-\mathbf{c})^T \overbrace{\boldsymbol{\alpha}(\mathbf{x}-\mathbf{c})}^{\psi}}_{\varphi}}$$

$$\frac{\partial y^1(k)}{\partial c} = \frac{\partial y^1(k)}{\partial \varphi}\frac{\partial \varphi}{\partial \psi}\frac{\partial \psi}{\partial c} = -e^{-\varphi}\left\{\alpha\psi + \alpha^T\psi\right\}(-1)$$

$$\frac{\partial y_i^1(k)}{\partial \alpha_i} = \frac{\partial y_i^1(k)}{\partial \varphi}\frac{\partial \varphi}{\partial \alpha_i} = -e^{-\varphi}(\mathbf{x}-\mathbf{c}_i)^T(\mathbf{x}-\mathbf{c}_i)$$

Comparison of RBF Networks and MLP [1]

Radial-basis function (RBF) networks and multilayer perceptrons are examples of nonlinear layered feedforward networks. They are both universal approximators.

However, these two networks differ from each other in several important respects, as:

1. An RBF network (in its most basic form) has a single hidden layer, whereas an MLP may have one or more hidden layers.

2. Typically, the computation nodes of an MLP, be they located in a hidden or output layer, share a common neuron model. On the other hand, the computation nodes in the hidden layer *of* an RBF network are quite different and serve a different purpose from those in the output layer of the network.

Comparison of RBF Networks and MLP [1]

3. The hidden layer of an RBF network is nonlinear, whereas the output layer is linear. On the other hand, the hidden and output layers of an MLP used as a classifier are usually all nonlinear; however, when the MLP is used to solve nonlinear regression problems, a linear layer for the output is usually the preferred choice.

4. The argument of the activation function of each hidden unit in an RBF network computes the ***Euclidean norm (distance)*** between the input vector and the center of that unit. On the other hand, the activation function of each hidden unit in an MLP computes the ***inner product*** of the input vector and the synaptic weight vector of that unit.

Comparison of RBF Networks and MLP [1]

5. MLPs construct ***global*** approximations to nonlinear input-output mapping. Consequently, they are capable of generalization in regions of the input space where little or no training data are available.

On the other hand, RBF networks using exponentially decaying localized nonlinearities (e.g., Gaussian functions) construct ***local*** approximations to nonlinear input-output mapping, with the result that these networks are capable of fast learning and reduced sensitivity to the order of presentation of training data.

5th Mini Project

In this project, a chaotic time series is considered therein is the *logistic* ***map*** whose dynamics is governed by the following difference equation

Window size = 5

$$x(n) = 4x(n-1)\big(1-x(n-1)\big)$$

* Compare the results with MLP neural networks.

Literature Cited

The material of this lecture is based on:

[1] Simon Haykin, **Neural Networks: A Comprehensive Foundation**., Prentice Hall, 1998.

[2] Tuba Kurban and Erkan Beşdok, A *Comparison of RBF Neural Network Training Algorithms for Inertial Sensor Based Terrain Classification.*, Sensors, 9, 6312-6329; (doi:10.3390/s90806312), 2009.

[3] Mark J. L. Orr, *Introduction to Radial Basis Function Networks*, Centre for Cognitive Science, University of Edinburgh, 1996.

Artificial Neural Networks

Lecture 14

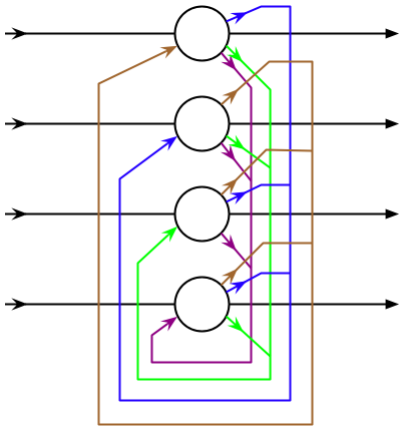
Hopfield Neural Network

Hopfield Neural Network

A **Hopfield net** is a form of **recurrent artificial neural network** invented by John Hopfield.

Hopfield nets serve as content-addressable memory systems with binary threshold units.

It can be used to solve the optimization problems.



Hopfield Neural Network

A **Hopfield network:**

$$u_j(n) = \sum_{i \neq j} w_{ij} y_i(n-1) + x_j(n)$$

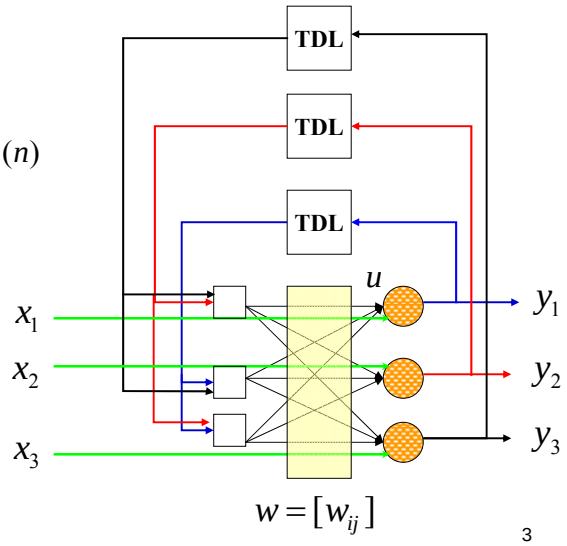
$$y_j(n) = f_j(u_j(n))$$

In Hopfield network the synaptic weights are symmetric:

$$w_{ij} = w_{ji}$$

Also, in HN there are no self feedback:

$$w_{ii} = 0$$



Hopfield Neural Network

A **Binary Hopfield network:**

$$u_j(n) = \sum_{i \neq j} w_{ij} y_i(n-1) + x_j(n)$$

$$y_j(n) = \begin{cases} 1 & u_j(n) \geq \theta_j \\ 0 & u_j(n) < \theta_j \end{cases} \quad \text{or} \quad y_j(n) = \begin{cases} 1 & u_j(n) > \theta_j \\ y_j(n-1) & u_j(n) = \theta_j \\ 0 & u_j(n) < \theta_j \end{cases}$$

Hopfield Neural Network

The energy E for the whole network can be determined from energy function as the following equation:

$$E = -\frac{1}{2}\sum_i \sum_j w_{ij} y_i y_j - \sum_i x_i y_i + \sum_i y_i \theta_i$$

So: $$\Delta E_i = -\left(\sum_j w_{ij} y_j + x_i - \theta_i \right) \Delta y_i$$

Δy_i is ***positive*** when the terms in brackets is ***positive***; and Δy_i becomes ***negative*** in the other case.
Therefore the energy increment for the whole network ΔE will always decrease however the input changes.

Hopfield Neural Network

So, the following two statements can be introduced:
1. The energy function ***E*** is a Lyapunov function.
2. The HNN is a stable in accordance with Lyapunov's Theorem.

The ability to minimize the energy function in a very short convergence time makes the HN described above be very useful in solving the problems with solutions obtained through minimizing a cost function.
Therefore, this cost function can be rewritten into the form of the energy function as E if the synaptic weights w_{ij} and the external input x_i can be determined in advance.

Hopfield Neural Network

Hopfield networks can be implemented to operate in two modes:

- ***Synchronous mode*** of training Hopfield networks means that all neurons fire at the same time.
- ***Asynchronous mode*** of training Hopfield networks means that the neurons fire at random.

Example: Consider a Hopfield network with three neurons

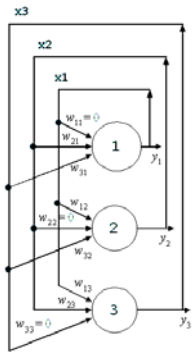
$$W = \begin{bmatrix} 0 & -0.4 & 0.2 \\ -0.4 & 0 & 0.5 \\ 0.2 & 0.5 & 0 \end{bmatrix}$$

Let the state of the network be: $y(0)=[\ 1,\ 1,\ 0\ ]^T$.

Hopfield Neural Network

Example:

$$W = \begin{bmatrix} 0 & -0.4 & 0.2 \\ -0.4 & 0 & 0.5 \\ 0.2 & 0.5 & 0 \end{bmatrix} \quad y(0) = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$$



	Synchronous mode	***Asynchronous mode***		
$y(0)=[1,1,0]^T$	$y_1(1)=[\ 0,1,0\ ]^T$ $y_2(1)=[\ 0,0,0\ ]^T$ $y=y_3(1)=[\ 0,0,0]^T$	$y(1)=[\ 0,1,0\ ]^T$	$y(1)=[\ 1,0,0\ ]^T$	$y(1)=[1,1,1]$

State table of Hopfield N.N.

A Hopfield net with n neurons has 2^n possible states, assuming that each neuron output produces two values 0 and 1.

The state table for the above example Hopfield network with 3 neurons is given below.

Init. state	*state if N1 fires*	*state if N2 fires*	*state if N3 fires*
000	100	000	*000*
001	101	011	000
010	010	*000*	011
011	011	011	011
100	100	100	101
101	101	111	101
110	***010***	**100**	**111**
111	011	111	111

Hopfield N.N. as BAM

Hopfield networks are used as **content-addressable memory** or **Bidirectional Associative Memory (BAM).** The content-addressable memory is such a device that returns a pattern when given a noisy or incomplete version of it.

In this sense a content-addressable memory is ***error-correcting*** as it can override provided inconsistent information.

The discrete Hopfield network as a memory device operates in two phases: ***storage phase*** and ***retrieval phase***.
During the ***storage phase*** the network learns the weights after presenting the training examples. The training examples for this case of automated learning are binary vectors, called also fundamental memories. The weights matrix is learned using the Widrow-Hoff rule. According to this rule when an input pattern is passed to the network and the estimated network output does not match the given target, the corresponding weights are modified by a small amount.
The difference from the single-layer perceptron is that no error is computed, rather the target is taken directly for weight updating.

Widrow-Hoff Learning

Learning: The Widrow-Hoff learning rule suggests to compute the summation block of the *i*-th neuron:

$$u_j(n) = \sum_{i \neq j} w_{ij} y_i\,(n-1) + x_j(n)$$

There are two cases to consider:

$$\left.\begin{array}{l} u_j(n) \geq 0 \\ \& \\ y_j(n-1) = 0 \end{array}\right\} \Rightarrow w_{ji} = w_{ji} - \frac{0.1 + u_j(n)}{n}$$

$$\left.\begin{array}{l} u_j(n) < 0 \\ \& \\ y_j(n-1) = 1 \end{array}\right\} \Rightarrow w_{ji} = w_{ji} + \frac{0.1 - u_j(n)}{n}$$

Where, *n* denotes the number of neurons.

Outer product Learning

Learning: Suppose that we wish to ***store*** a set of ***N*-dimensional** vectors (binary words), denoted by $\{\xi_\mu,\ \mu = 1, 2, \ldots, M\}$. We call these *M* vectors fundamental memories, representing the patterns to be memorized by the network.

The **outer product** learning rule, that is, the generalization of *Hebb*'s learning rule:

$$\mathbf{W} = \frac{1}{N}\left(\sum_{\mu=1}^{M} \boldsymbol{\xi}_\mu \boldsymbol{\xi}_\mu^T - M\mathbf{I}\right)$$

From these defining equations of the synaptic weights matrix, we note the following:

- The output of each neuron in the network is fed back to all other neurons.
- There is no self-feedback in the network (i.e., $w_{ii} = 0$).
- The weight matrix of the network is symmetric. (i.e., $W^T = W$)

Learning Algorithm

Initialization: Let the testing vector become initial state $\mathbf{x}(0)$

Repeat

-*update* asynchronously the components of the state $\mathbf{x}(t)$

$$u_j(n) = \sum_{i \neq j} w_{ij} y_i(n-1) + x_j(n) > 0 \Rightarrow y_i(n) = 1$$

$$u_j(n) = \sum_{i \neq j} w_{ij} y_i(n-1) + x_j(n) < 0 \Rightarrow y_i(n) = 0$$

-*continue* this updating until the state remains unchanged

until convergence

Generate output: return the **stable state** (fixed point) as a result. The network finally produces a time invariant state vector $\mathbf{y}$ which satisfies the ***stability condition:***

$$\mathbf{y} = \text{sgn}(\mathbf{W}\mathbf{y} + \mathbf{b})$$

Learning Algorithm

During the ***retrieval phase*** a testing vector called probe is presented to the network, which initiates computing the neuron outputs and developing the state.
After sending the training input to the recurrent network its output changes for a number of steps until reaching a stable state.
The selection of the next neuron to fire is asynchronous, while the modifications of the state are deterministic.
After the state evolves to a stable configuration, that is the state is not more updated, the network produces a solution.

This state solution can be envision as a fixed point of the dynamical network system. The solution is obtained after adaptive training.

Summary of Hopfield Model

The operational procedure for the Hopfield network may now be summarized as follows:

1. ***Storage (Learning).*** Let $\xi_1, \xi_2, \ldots, \xi_M$ denote a known set of N-dimensional memories. Construct the network by using the **Widrow-Hoff** or **outer product** rule (Le., Hebb's postulate of learning) to compute the synaptic weights of the network**.**
The elements of the vector ξ_M equal +1/-1. Once they are computed, the synaptic weights are kept fixed.

2. Initialization. Let ξ_{probe} denote an unknown N-dimensional input vector (probe) presented to the network. The algorithm is initialized by setting

$$y_j(0) = \xi_{j,probe} \quad j = 1, \ldots, N$$

where $y_j(0)$ is the state of neuron j at time $n = 0$.

3. Iteration until Convergence. Update the elements of state vector ***y(n)*** asynchronously (i.e., randomly and one at a time) according to the rule

$$y(n+1)=\text{sgn}[w.y(n)]$$

Repeat the iteration until the state vector **s** remains unchanged.

4. ***Outputting.*** Let y_{fixed} denote the fixed point (stable state) computed at the end of step 3. The resulting output vector y of the network is

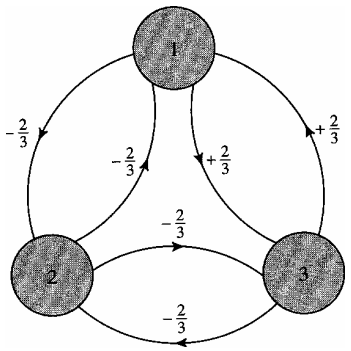
$$\mathbf{Y} = y_{fixed}$$

Summary of Hopfield Model

Example: Consider a Hopfield N.N. with 3 neurons, which we want store two vectors (1,-1,1) and (-1,1,-1):

$$\mathbf{W} = \frac{1}{3}\left(\begin{bmatrix}1\\-1\\1\end{bmatrix}\begin{bmatrix}1 & -1 & 1\end{bmatrix} + \begin{bmatrix}-1\\1\\-1\end{bmatrix}\begin{bmatrix}-1 & 1 & -1\end{bmatrix} - 2\begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}\right) = \frac{1}{3}\begin{bmatrix}0 & -2 & 2\\-2 & 0 & -2\\2 & -2 & 0\end{bmatrix}$$

The threshold applied to each neuron is assumed to be zero and the corresponding HNN has no external input.



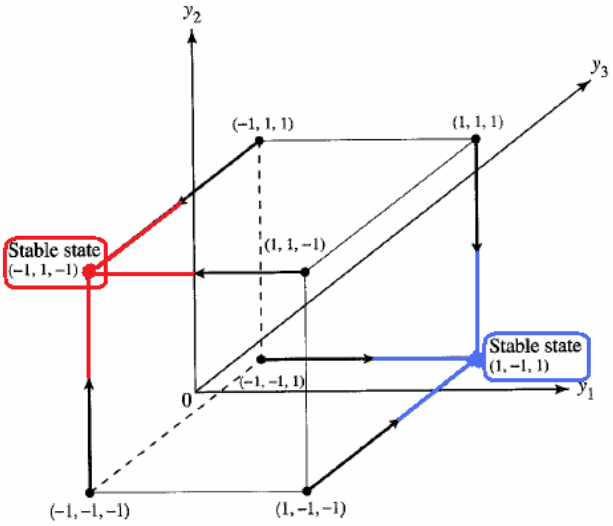
Summary of Hopfield Model

$$\mathbf{W} = \frac{1}{3}\begin{bmatrix} 0 & -2 & 2 \\ -2 & 0 & -2 \\ 2 & -2 & 0 \end{bmatrix}$$

$$y(0) = (1 \quad 1 \quad 1)^T \rightarrow (0 \quad -1 \quad 0)^T \rightarrow (1 \quad -1 \quad 1)^T$$
$$y(0) = (1 \quad 1 \quad -1)^T \rightarrow (-1 \quad 0 \quad 0)^T \rightarrow (-1 \quad 1 \quad -1)^T$$
$$y(0) = (-1 \quad 1 \quad 1)^T \rightarrow (0 \quad 0 \quad -1)^T \rightarrow (-1 \quad 1 \quad -1)^T$$
$$y(0) = (-1 \quad -1 \quad 1)^T \rightarrow (1 \quad 0 \quad 0)^T \rightarrow (1 \quad -1 \quad 1)^T$$
$$y(0) = (1 \quad -1 \quad -1)^T \rightarrow (0 \quad 0 \quad 1)^T \rightarrow (1 \quad -1 \quad 1)^T$$
$$y(0) = (-1 \quad -1 \quad -1)^T \rightarrow (0 \quad 1 \quad 0)^T \rightarrow (-1 \quad 1 \quad -1)^T$$

Therefore, the network has **two** fundamental memories.

Summary of Hopfield Model



$$y(0)=(1 \quad 1 \quad 1)^T \rightarrow (1 \quad -1 \quad 1)^T$$
$$y(0)=(1 \quad 1 \quad -1)^T \rightarrow (-1 \quad 1 \quad -1)^T$$
$$y(0)=(-1 \quad 1 \quad 1)^T \rightarrow (-1 \quad 1 \quad -1)^T$$
$$y(0)=(-1 \quad -1 \quad 1)^T \rightarrow (1 \quad -1 \quad 1)^T$$
$$y(0)=(1 \quad -1 \quad -1)^T \rightarrow (1 \quad -1 \quad 1)^T$$
$$y(0)=(-1 \quad -1 \quad -1)^T \rightarrow (-1 \quad 1 \quad -1)^T$$

Literature Cited

The material of this lecture is based on:

[1] Simon Haykin, **Neural Networks: A Comprehensive Foundation**., Prentice Hall, 1998.

[2] *http://homepages.gold.ac.uk/nikolaev/cis311.htm*

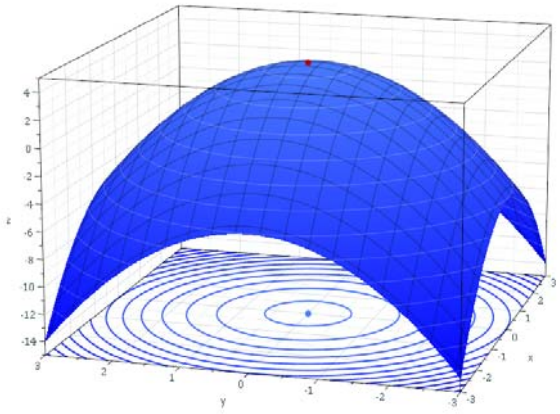
Genetic Algorithm

Lecture 15

Introduction to G.A.

Optimization

In mathematics and engineering science, **optimization**, refers to choosing the best element from some set of available alternatives.



Methods of Optimization

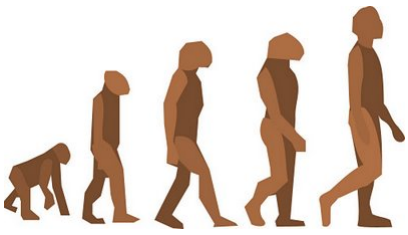
Main approaches to solve an optimization problem are:

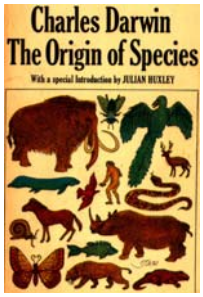
- Newton's Method
- Quasi-Newton method
- Gradient descent
- Gauss–Newton algorithm
- Levenberg–Marquardt algorithm
- Steepest descent
- Simulated Annealing (Monte Carlo)
- **Genetic Algorithms**

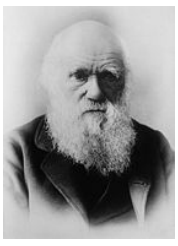
Genetic Algorithm (GA) is the most popular type of Evolutionary Algorithm (EA).

Evolutionary Algorithm

Darwin's Theory of Evolutionary:







History of Genetic Algorithm

Genetic Algorithms (GAs) are adaptive random search algorithm premised on the evolutionary ideas of natural selection and genetic. The basic concept of GAs is designed to ***simulate processes in natural system*** necessary for evolution, specifically those that follow the principles first laid down by Charles Darwin of survival of the fittest.
As such they represent an intelligent exploitation of a random search within a defined search space to solve a problem.

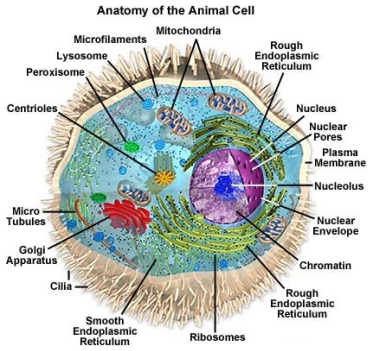
Genetic algorithms originated from the studies of *cellular automata*, conducted by ***John Holland*** and his colleagues in 60s at the University of Michigan. Research in GAs remained largely theoretical until the mid-1980s, when The *First International Conference* on Genetic Algorithms was held at The University of Illinois.

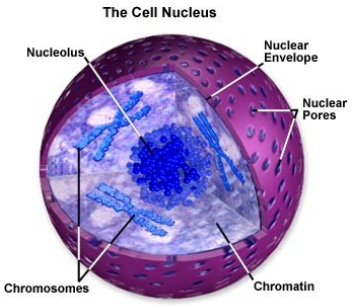
Genetic Algorithm

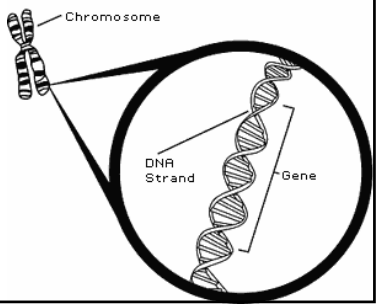
GAs were introduced as a computational analogy of adaptive systems. They are *modeled loosely* on the principles of the evolution via natural selection, employing a population of individuals that undergo selection in the presence of variation-inducing operators such as **mutation** and **recombination** (**crossover**). A fitness function is used to evaluate individuals, and reproductive success varies with **fitness**.
The Algorithms can be summarized as:

1. Randomly generate an initial population M(0)
2. Compute and save the fitness u(*m*) for each individual *m* in the current population M(t).
3. Define selection probabilities p(*m*) for each individual *m* in M(t) so that p(*m*) is proportional to u(*m*)
4. Generate M(t+1) by probabilistically selecting individuals from M(t) to produce offspring via genetic operators (Crossover and Mutation)
5. Repeat step 2 until satisfying solution is obtained.

GA and Biological background







1. Randomly generate an initial population

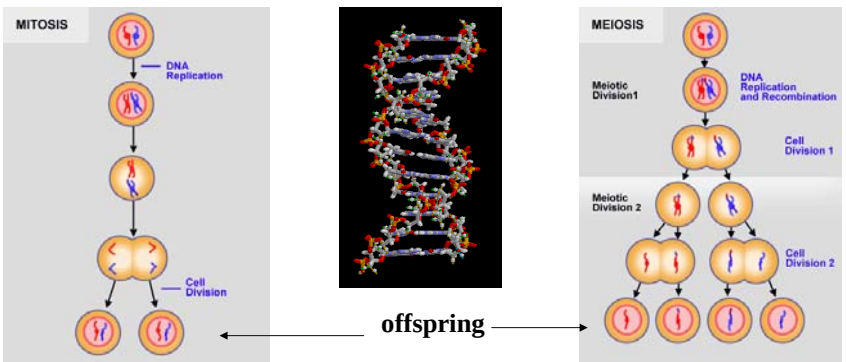
- Genetic information is stored in the **chromosomes.**
- Each chromosome is build of **DNA**
- Chromosomes in humans form pairs
- The chromosome is divided in parts: **genes**
- Genes code for properties

GA and Biological background

2. Compute and save the fitness u(*m*) for each individual *m* in the current population M(t).

3. Define selection probabilities p(*m*) for each individual *m* in M(t) so that p(*m*) is proportional to u(*m*)

4. Generate M(t+1) by probabilistically selecting individuals from M(t) to produce offspring via genetic operators (Crossover and Mutation)



Genetic Algorithm

GAs are useful and efficient when

- The search space is large, complex or poorly understood.
- Domain knowledge is scarce or expert knowledge is difficult to encode to narrow the search space.
- No mathematical analysis is available.
- Traditional search methods fail.

Genetic Algorithm

Basic algorithm of Genetic algorithm

0 START : Create random population of **n** chromosomes

1 FITNESS : Evaluate fitness **f(x)** of each chromosome in the population

2 NEW POPULATION (using Genetic operations)
0 SELECTION **:** Based on **f(x)**
1 RECOMBINATION **:** Cross-over chromosomes
2 MUTATION **:** Mutate chromosomes
3 ACCEPTATION **:** Reject or accept new one

3 REPLACE : Replace old with new population: the new generation

4 TEST : Test problem criterium

5 LOOP : Continue step 1 – 4 until criterium is satisfied

Encoding Issue

Each choromosme contains a possible solution of optimization problem. To define a possible solution in choromosms the encoding procedure should be employed. The encoding methods can be classified as follows:

- The binary encoding

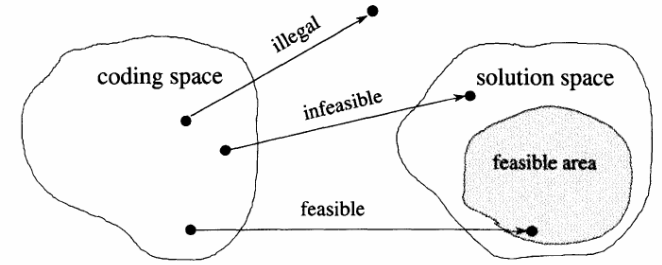
0	0	0	1	1	0	1	1	1	0	1	1

- Real-number encoding

12.25	8.64	-7.26	4.40

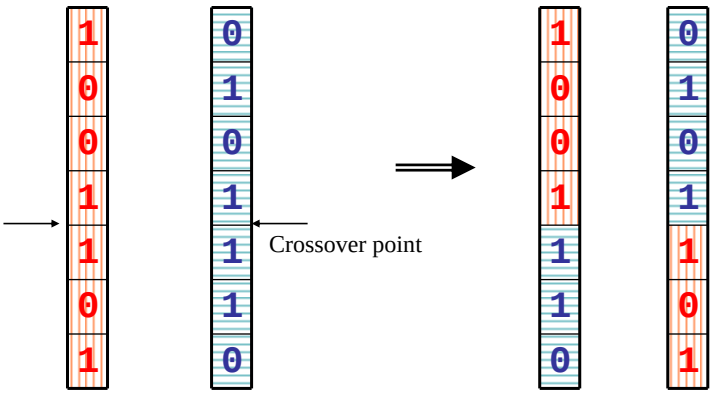
- Integer encoding

1	3	2	8



Crossover

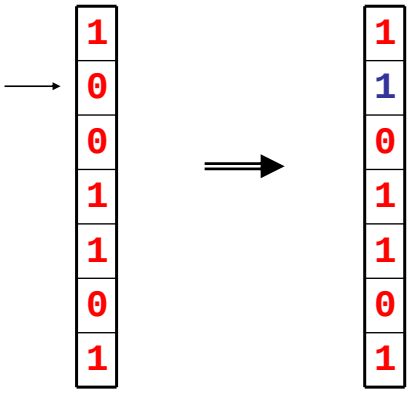
Recombination (cross-over) can when using bitstrings schematically be represented as:



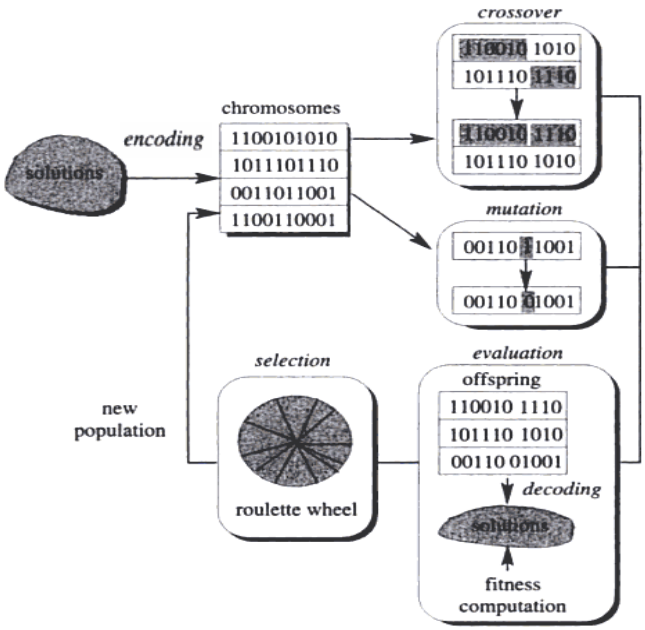
offspring

Mutation

- Mutation is anotehr operator which prevents the algorithm to be trapped in a local minimum.
- In the bitstring approach mutation is simply the flipping of one of the bits.



Evaluation and Selection



Encoding Procedure

In G.A. preparing, first step is the encoding procedure. In this step we encode the decision variables into **binary string**.

The length of string depends on the required precision. Consider variable x_j as

$$x_j \in \left[b_j \quad a_j \right]$$

And the *Required precision* is ***n*** place after the decimal point.

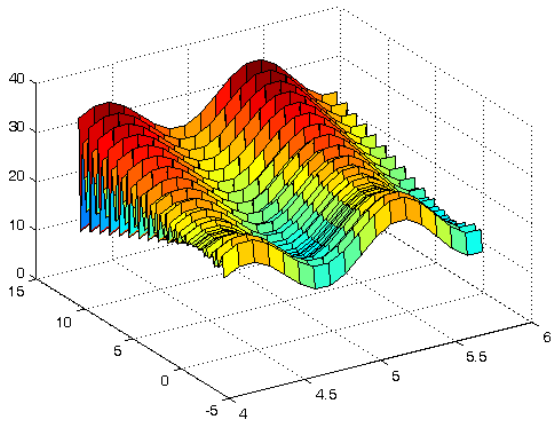
The number of bits (m_j) can be calculated using the inequality:

$$2^{m_j - 1} < (b - a) \times 10^n < 2^{m_j} - 1$$

Example for Encoding Procedure

Example: $\max f(x, x) = 21.5 + x_1 \sin(4\pi x_1) + x_2 \sin(20\pi x_2)$

$$-3 \le x_1 \le 12.1 \qquad 4.1 \le x_2 \le 5.8$$



Example for Encoding Procedure

Example: $\max f(x, x) = 21.5 + x_1 \sin(4\pi x_1) + x_2 \sin(20\pi x_2)$

$-3 \le x_1 \le 12.1 \qquad 4.1 \le x_2 \le 5.8$

$(12.1 - (-3)) * 10^4 = 151000$

$2^{18-1} = 131072 < 151000 < 2^{18} - 1 = 262143$

$m_1 = 18$

$(5.8 - 4.1) * 10^4 = 17000$

$2^{15-1} = 16384 < 17000 < 2^{15} - 1 = 32767$

$m_2 = 15$

A chromosome: v_j $\underbrace{000001010100101001}_{\text{18 bit}}$ $\underbrace{101111011111110}_{\text{15 bit}}$

Mapping from binary to decimal

We can use the following equation to map a binary string to a real value:

$$x_j = a_j + decimal(substring) \times \frac{b_j - a_j}{2^{m_j} - 1}$$

For example: v_j $\underbrace{000001010100101001}_{\text{18 bit}}$ $\underbrace{101111011111110}_{\text{15 bit}}$

Binary	*Decimal* (*substring*)	*Real value*
000001010100101001	5417	-2.6880
101111011111110	24318	5.3617

Initial Population

In GA process to solve an optimization problem in first step an **Initial Population** should randomly be generated.

To generate the initial population firstly define a **Population Size**.

$v_1 = [00000101010010100110111011111110]$
$v_2 = [00111010111001100000010101001000]$
$v_3 = [11100011100000100001010100100110]$
$v_4 = [10011011010010110100000010111001]$
$v_5 = [00001011110110001000111000101000]$
$v_6 = [11111010101101100000001010011001]$
$v_7 = [11010001001111100010011001110110 1]$
$v_8 = [00101101010000110001011001100 1100]$
$v_9 = [11111000101110110001110100011110 1]$
$v_{10} = [11110100111010101000001010110 1010]$

$v_1 = [x_1, x_2] = [-2.687969, 5.361653]$
$v_2 = [x_1, x_2] = [\ 0.474101, 4.170144]$
$v_3 = [x_1, x_2] = [10.419457, 4.661461]$
$v_4 = [x_1, x_2] = [\ 6.159951, 4.109598]$
$v_5 = [x_1, x_2] = [-2.301286, 4.477282]$
$v_6 = [x_1, x_2] = [11.788084, 4.174346]$
$v_7 = [x_1, x_2] = [\ 9.342067, 5.121702]$
$v_8 = [x_1, x_2] = [-0.330256, 4.694977]$
$v_9 = [x_1, x_2] = [11.671267, 4.873501]$
$v_{10} = [x_1, x_2] = [11.446273, 4.171908]$

Selection

Evaluation:

$eval(v_1) = f(-2.687969, 5.361653) = 19.805119$
$eval(v_2) = f(\ 0.474101, 4.170144) = 17.370896$
$eval(v_3) = f(10.419457, 4.661461) = 9.590546$
$eval(v_4) = f(\ 6.159951, 4.109598) = 29.406122$
$eval(v_5) = f(-2.301286, 4.477282) = 15.686091$
$eval(v_6) = f(11.788084, 4.174346) = 11.900541$
$eval(v_7) = f(\ 9.342067, 5.121702) = 17.958717$
$eval(v_8) = f(-0.330256, 4.694977) = 19.763190$
$eval(v_9) = f(11.671267, 4.873501) = 26.401669$
$eval(v_{10}) = f(11.446273, 4.171908) = 10.252480$

Selection

Selection: (based on Roulette Wheel)

1. Calculate the fitness value for each chromosome in the population
$$eval(v_j)=f(v_j)$$
2. Calculate the total fitness for the population
$$F=\Sigma f(v_j)$$
3. Calculate the selection probability for each chromosome of the population
$$P_j=f(v_j)/F$$
4. Calculate the cumulative probability for each chromosome
$$q_k=\Sigma^{k}_{j=1}p_j$$

5. Generate a random number r in [0,1]
6. If $r<q_1$ select the first chromosome
and if $q_{k-1}< r <q_k$ then select the chromosome v_k.

Selection

In the last example:

The total fitness is $$F = \sum_{k=1}^{10} eval(v_k) = 178.135372$$

And the probability of a selection for each chromosome is:

$p_1 = 0.111180$, $p_2 = 0.097515$, $p_3 = 0.053839$
$p_4 = 0.165077$, $p_5 = 0.088057$, $p_6 = 0.066806$
$p_7 = 0.100815$, $p_8 = 0.110945$, $p_9 = 0.148211$
$p_{10} = 0.057554$

Selection

In the last example:
The cumulative probability q_k for each chromosome is:

$q_1 = 0.111180$, $q_2 = 0.208695$, $q_3 = 0.262534$
$q_4 = 0.427611$, $q_5 = 0.515668$, $q_6 = 0.582475$
$q_7 = 0.683290$, $q_8 = 0.794234$, $q_9 = 0.942446$
$q_{10} = 1.000000$

Now we are ready to spin the roulette wheel 10 (population size) times, and each time we select a chromosome. So, *r* sequence can be generated randomly:

0.301431 0.322062 0.766503 0.881893
0.350871 0.583392 0.177618 0.343242
0.032685 0.197577

Selection

So, the new population is:

$v'_1 = [1001101101001011010000000101110 01]$ (v_4)
$v'_2 = [1001101101001011010000000101110 01]$ (v_4)
$v'_3 = [0010110101000011000101100110011 00]$ (v_8)
$v'_4 = [1111100010111011000111010001111 01]$ (v_9)
$v'_5 = [1001101101001011010000000101110 01]$ (v_4)
$v'_6 = [1101000100111110001001100111011 01]$ (v_7)
$v'_7 = [0011101011100110000000101010010 00]$ (v_2)
$v'_8 = [1001101101001011010000000101110 01]$ (v_4)
$v'_9 = [0000010101001010011011110111111 10]$ (v_1)
$v'_{10} = [0011101011100110000000101010010 00]$ (v_2)

Crossover

One of the important GA operators which can help us to search the corresponding space is **Crossover**:
In crossover procedure there are two steps:

1. Define the crossover rate (p_c) to select the chromosomes for crossover.
2. Choose the crossover method (e.g. one-cut-point) and generate the new chromosomes.

In the last example: $p_c = 0.25$

0.6257	0.2668	0.2886	0.2951
$0.1632 < p_c$	0.5674	$0.0859 < p_c$	0.3928
0.7707	0.5486		

v'_5 = [1001101101001011010000000101111001]

v'_7 = [0011101011100110000000101010010000]

Crossover

Crossover methods: **one-cut-point**

Cutting point: a random number in [1-33] (e.g.: **17**)

v'_5 = [10011011010010110 0000010101001000]

v'_7 = [00111010111001100 1000000010111001]

Mutation

To prevent the GA of trapped in local minimum, Mutation operator is employed. In mutation procedure the following 2 steps are important.

1. Define the mutation rate (p_m) to select genes.
2. Generate "number of genes*population size" random numbers (r_m) and by comparing those with mutation rate choose the corresponding genes which satisfy the following equation to mutate (0→1 and 1→0).

$$r_m < p_m$$

Mutation

In the last example: $p_m = 0.01$

Random_num.	*Bit_ position*	*Chrom._No.*	*Bit_No.*
0.009857	105	4	6
0.003113	164	5	32
0.000946	199	7	1
0.001282	329	10	32

$v_4' = [1111100010111011000111010001111 01]$

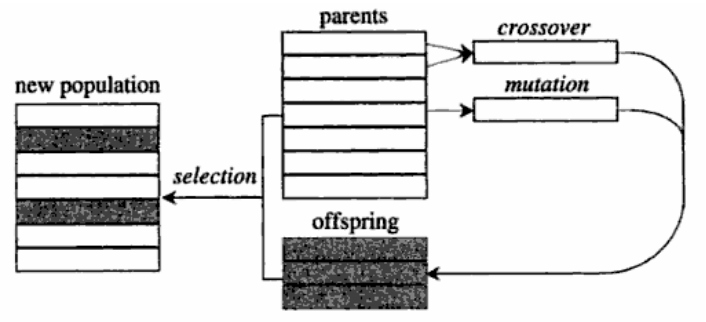
1

The solution

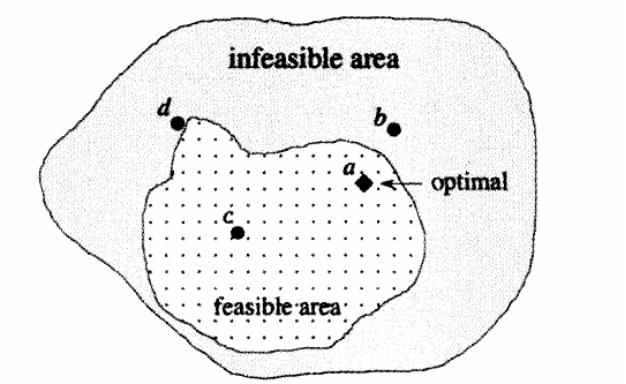
In the last example: After 1000 generation, the best chromosome is as follow and it is obtained in 419th generation.

$$\boldsymbol{v}^* = (11111000000011100011110100101010110)$$
$$eval(\boldsymbol{v}^*) = f(11.631407, 5.724824) = 38.818208$$
$$x_1^* = 11.631407$$
$$x_2^* = 5.724824$$
$$f(x_1^*, x_2^*) = 38.818208$$

General Structure of G.A.



Solution Space, feasible and infeasible space



Solution space: feasible area and infeasible area.

G.A. for minimizing the Ackley's function

$$f(x_1, x_2) = -c_1.\exp\left(-c_2\sqrt{\frac{1}{2}\sum_{j=1}^{2} x_j^2}\right) - \exp\left(\frac{1}{2}\sum_{j=1}^{2}\cos(c_3.x_j)\right) + c_1.e$$

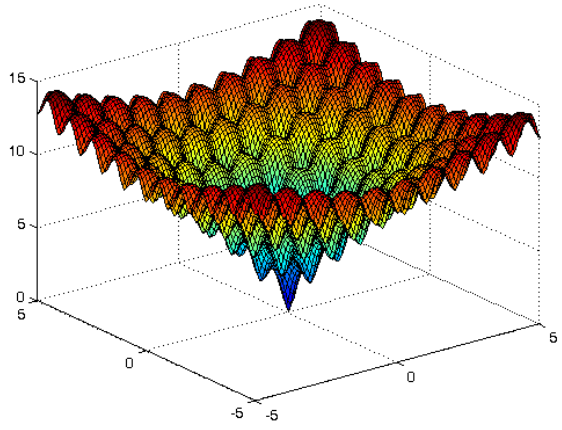
$-5 \le x_1, x_2 \le 5$

$c_1 = 20$

$e = 2.71282$

$c_2 = 0.2$

$c_3 = 2\pi$



Find a solution for Ackley's Function Optimization Problem

The G.A. parameters are set as:

- population size: 10
- max. generation: 1000
- *Pm*: 0.1
- *Pc*: 0.3

Initial conditions
(**Real number encoding):**

$-5 \le x_1, x_2 \le 5$

	x_1	x_2
v_1 = [	4.954222,	0.169225]
v_2 = [	−4.806207,	−1.630757]
v_3 = [	4.672536,	−1.867275]
v_4 = [	1.897794,	−0.196387]
v_5 = [	−2.127598,	0.750603]
v_6 = [	−3.832667,	−0.959655]
v_7 = [	−3.792383,	4.064608]
v_8 = [	1.182745,	−4.712821]
v_9 = [	3.812220,	−3.441115]
v_{10} = [	−4.515976,	4.539171]

Arithmetic Crossover

$$\begin{cases} v_1 \\ v_2 \end{cases} \longrightarrow \begin{cases} v_1' = v_1 + (1-\lambda)v_2 \\ v_2' = v_2 + (1-\lambda)v_1 \end{cases}$$

where $\lambda \in [0 \quad 1]$

Non-uniform Mutation

$$v = [x_1, \cdots x_k, \cdots x_n] \xrightarrow{\textit{Mutation}} v = [x_1, \cdots x'_k, \cdots x_n]$$

$$x'_k = x_k + \Delta(t, x_k^U - x_k), \quad \text{or} \quad x'_k = x_k + \Delta(t, x_k - x_k^L)$$

$$\Delta(t, y) = y.r.(1 - \frac{t}{T})^b$$

t : generation number

T: maximal generation number

r: random number $\in [0 \quad 1]$

b : degree of nonuniformity

G.A. solution

Evaluation

Here you can see the corresponding fitness function for parent chromosomes:

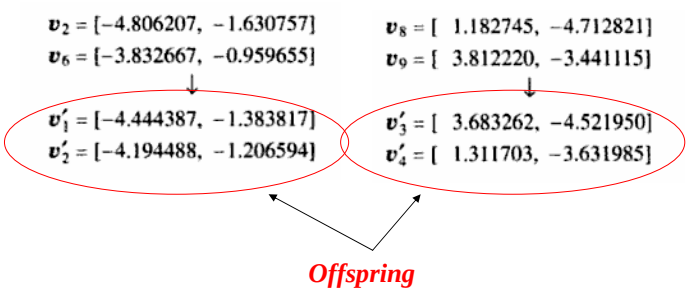
$eval(v_1) = f(\ 4.954222,\ \ 0.169225) = 10.731945$
$eval(v_2) = f(-4.806207,\ -1.630757) = 12.110259$
$eval(v_3) = f(\ 4.672536,\ -1.867275) = 11.788221$
$eval(v_4) = f(\ 1.897794,\ -0.196387) = \ 5.681900$
$eval(v_5) = f(-2.127598,\ \ 0.750603) = \ 6.757691$
$eval(v_6) = f(-3.832667,\ -0.959655) = \ 9.194728$
$eval(v_7) = f(-3.792383,\ \ 4.064608) = 11.795402$
$eval(v_8) = f(\ 1.182745,\ -4.712821) = 11.559363$
$eval(v_9) = f(\ 3.812220,\ -3.441115) = 12.279653$
$eval(v_{10}) = f(-4.515976,\ \ 4.539171) = 14.251764$

Now, we generate a sequence of random numbers:

0.828211 0.199683 0.639149 0.629170 0.957427
0.149358 0.304788 0.058504 0.149693 0.326670

G.A. solution

So, the chromosomes v_2, v_6, v_8, v_9 are selected for **crossover**



G.A. solution

Mutation:

bit_pos	*chrom_num*	*variable*	*random_num*
11	6	x_1	0.081393

offspring ⟶ $v'_5 = [-4.068506, -0.959655]$

The fitness value for each offspring:

$$eval(v'_1) = f(-4.444387, -1.383817) = 11.927451$$
$$eval(v'_2) = f(-4.194488, -1.206594) = 10.566867$$
$$eval(v'_3) = f(3.683262, -4.521950) = 13.449167$$
$$eval(v'_4) = f(1.311703, -3.631985) = 10.538330$$
$$eval(v'_5) = f(-4.068506, -0.959655) = 9.083240$$

7^{th} Mini Project

In this project, by using of G.A. you should find the minimum point of Ackley's function.

Literature Cited

The material of this lecture is based on:

[1] Mitsuo Gen, Runwei Cheng, **Genetic Algorithms and Engineering Design**., Wiley-Interscience, 1997.