Intelligent Control Systems

Gülay Öke Günel
Room:4220
E-mail: gulayoke@itu.edu.tr

Teaching Assistant: Gökçen Devlet Şen
E-mail: sengo@itu.edu.tr

Grading Policy

% 30 Midterm Exam
% 20 Homework
% 20 Project
% 30 Final Exam

Textbook

- Haykin, S., Neural Networks and Learning Machines, Third Edition, Pearson Prentice Hall, New Jersey, 2009
- Jang, J.-S. R., C.-T. Sun, E. Mizutani, Neuro-Fuzzy and Soft Computing, PTR Prentice-Hall, 1997.

Other references

- Alpaydın E., Introduction to Machine Learning, The MIT Ptress, 2010.
- Engelbrecht A.P., Computational Intelligence: An Introduction, John Wiley and Sons, 2007.
- Nguyen H.T., Prasad N.R., Walker C.L., Walker E.A., A First Course in Fuzzy and Neural Control, CRC Press, 2003.
- Efe M.Ö., Kaynak O., Yapay Sinir Ağları ve Uygulamaları, Boğaziçi Üniversitesi, 2000.
- Elmas Ç., Bulanık Mantık Denetleyiciler, Seçkin Yayıncılık, 2003.

Further Reading

- Goodfellow I., Bengio Y., Courvillle A., Deep Learning, MIT Press, 2016.

Contents of the Course

1.Introduction
2.Learning Processes
3.Optimization Techniques
4.Single-Layer Perceptrons
5.Multi-Layer Perceptrons
6.Backpropagation Algorithm
7.Neural Network based Control
8.Fuzzy Sets
9.Fuzzy Rules and Fuzzy Reasoning
10.Fuzzy Inference Systems
11.Fuzzy Logic Based Control
12.Genetic Algorithms
13.Other Derivative-Free Global Optimization Methods
14.Genetic Algorithms in Control

Other Topics

Topics under the scope of computational intelligence that are not in the scope of this course:

- Adaptive Neuro-Fuzzy Inference System (ANFIS)
- Radial Basis Functions
- Hopfield Models
- Support Vector Machines
- Kohonen Self-Organizing Maps
- Learning Vector Quantization
- Principle Component Analysis
- Committee Machines
- Regression Trees
- Data Clustering Algorithms
- Information Theoretic Models
- Artificial Immune Systems
- Recurrent Neural Networks
- Deep Learning

……..

Final Project

In groups of 4:

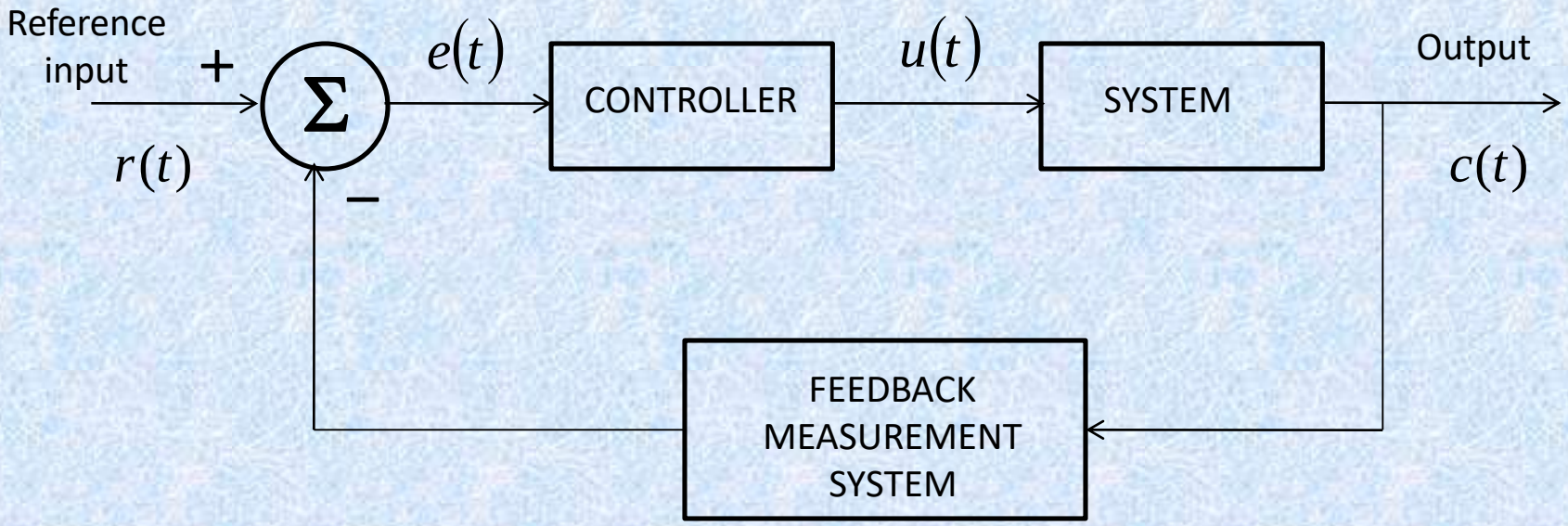
You will be assigned from the following benchmark problems:

- Robotic Manipulator
- Inverted Pendulum
- Ball and Beam
- PH System
- Bioreactor
- Continuously Stirred Tank Reactor
- Magnetic Levitation System

Each group will implement either a **neural** controller or a **fuzzy** controller to the problem they have chosen.

You will write a detailed report about your problem, your implementation and the results you have obtained.
You have to submit your report and you have to be able to show a running program.
You will also make a presention of your study .

A Control System



The block diagram of a control system

Attribute	Desired Value	Purpose	Specifications
Stability	High	Response should not grow without limit; it should decay to the desired value	Percentage overshoot, pole locations, time constants, damping ratios, phase and gain margins
Speed of Response	Fast	System should respond quickly to the inputs	Rise time, peak time, delay time, natural frequencies.
Steady-state error	Low	Error between the output of the system and the desired output after all the transients die, should be low.	Error tolerance for a step input.
Robustness	High	Accurate response under uncertain conditions and under parameter variations.	Input noise tolerance, measurement error tolerance, model error tolerance

Conventional Control Techniques

- Proportional-Integral-Derivative (PID) Control
- Optimal Control
  - Linear Quadratic Gaussian (LQG) control
- Adaptive Control
  - Model Reference Adaptive Control (MRAC)
- Nonlinear Feedback Control
- Sliding Mode Control
- Robust Control
- Stochastic Control

……

……

Need for an intelligent controller is due to:

- Inability to model the system
- Inability to build a controller for the system
- Performance problems
- Incomplete/unreliable information

Significant research has been carried out in understanding and emulating human intelligence while, in parallel, developing inference engines for processing human knowlegde.

The resultant techniques incorporate notions gathered from a wide range of specialization such as neurology, psychology, operations research, conventional control theory, computer science and communications theory.

Many of the results of this effort are used in Control Engineering And their combination led to new techniques for dealing with vagueness and uncertainty.

This is the domain of **Soft Computing** (another related term is **computational intelligence**) which focuses on stochastic, vague, empirical and associative situations, typical of the industrial and manufacturing environment.

Intelligent Controllers (also referred as **soft controllers**) are derivatives of Soft Computing, characterized by their ability to establish the functional relationship between their inputs and outputs from **empirical data**.

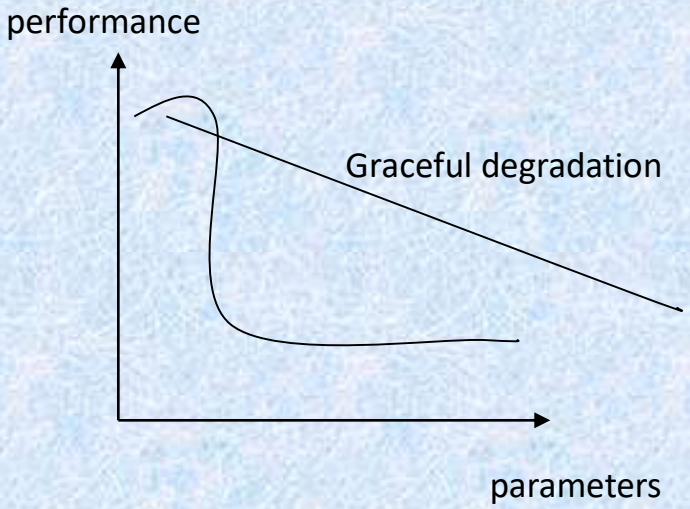
This is an important difference between intelligent controllers and conventional controllers, which are based on **explicit functional relations**.

The functional relationship between the inputs and outputs of an intelligent controller can be established either

- ✓ **Directly** – from a specified training set (ANN)
- ✓ **Indirectly** – by means of a relational algorithm or knowlegde base. (FUZZY)

Soft computing techniques possess the following properties:

- They demonstrate **adaptability**
- They are able to **learn**.
- They are able to **reason and decide**.
- They have an inherent **tolerance** to errors.
- They have the **graceful degradation** property.
- They possess **speeds** comparable to those of humans.



Soft Computing

Soft Computing is an emerging approach to computing which parallels the remarkable ability of the human mind to reason and learn in an environment of uncertainty and imprecision.
(Lotfi A. Zadeh, 1992)

Lotfi Zadeh (1921-2017)
Founder of fuzzy logic

- Intelligent controllers and soft computing techniques have been successfully implemented for nearly four decades now, for the solution of numeruous control problems from vastly different areas.
- Intelligent controllers have been designed to work alone or **in combination with** conventional controllers.
- It is possible to employ a soft computing technique as part of a conventional controller.
- For example, you can design a PID controller whose K_P , K_I and K_D gains are tuned by a neural network, a fuzzy controller or a genetic algorithm.

AN INTRODUCTION TO NEURAL NETWORKS

Neural Networks (neurocomputers, connectionist networks, parallel distributed processors)

Work on artificial neural networks has been motivated right from its inception by the recognition that the human brain computes in an entirely different way from the conventional digital computer.

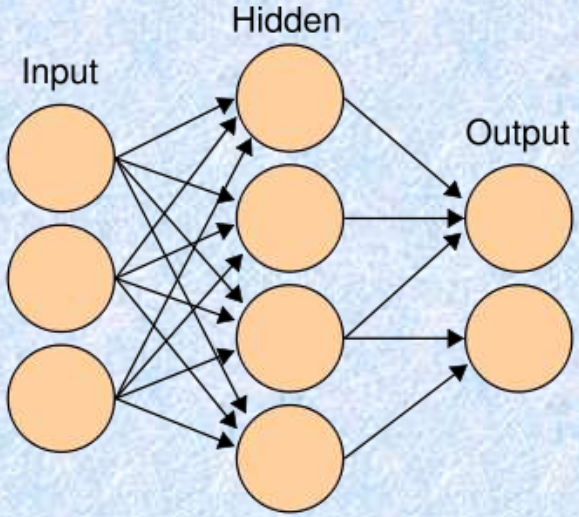
The brain is a highly complex, nonlinear, and parallel computer (information-processing system). It has the capability to organize its structural constituents, known as neurons, so as to perform certain computations many times faster than the fastest digital computer in existence today.
(Haykin, 1999)

A neural network is a massively parallel distributed processor made up of simple processing units, which has a natural propensity for storing experiental knowledge and making it available for use.

It resembles the brain in two respects:

1. Knowledge is acquired by the network from its environment through a learning process.

2. Interneuron connection strengths, known as synaptic weights are used to store the acquired knowlegde.



Main Features of Neural Networks

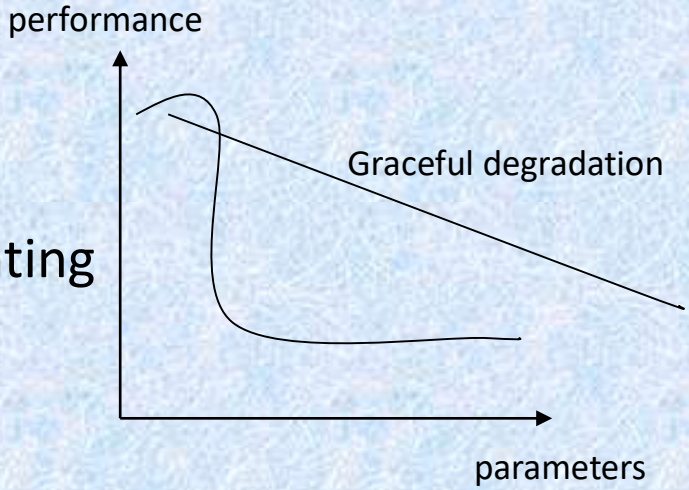
Nonlinearity

An artificial neuron can be linear or nonlinear. A NN made up of an interconnection of nonlinear neurons is itself nonlinear and this nonlinearity is distributed throughout the network.

Adaptivity

ANNs have a built-in capability to adapt their synaptic weights to changes in the surrounding environment. An ANN trained to operate in a specific environment can be easily retrained to deal with minor changes in the operating environmental conditions. When operating in a nonstationary (one where statistics change with time) environment an ANN can be designed to change its synaptic weights in real time.

Fault Tolerance



An ANN is inherently fault tolerant, or capable of robust computation, in the sense that its performance degrades gracefully under adverse operating conditions. For example, if a neuron or its connecting links are damaged, performance of the ANN is not seriously degraded, due to the distributed nature of information storage in the network.

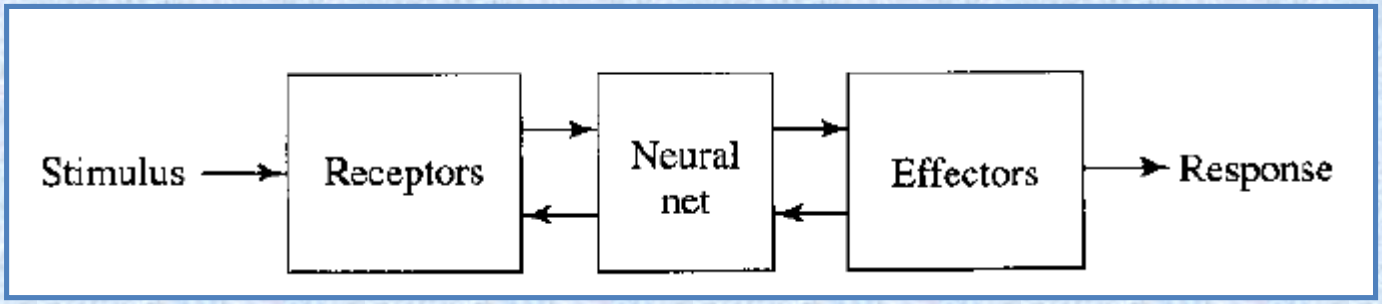
Generalization

An ANN can produce reasonable outputs for inputs not encountered during training (learning).

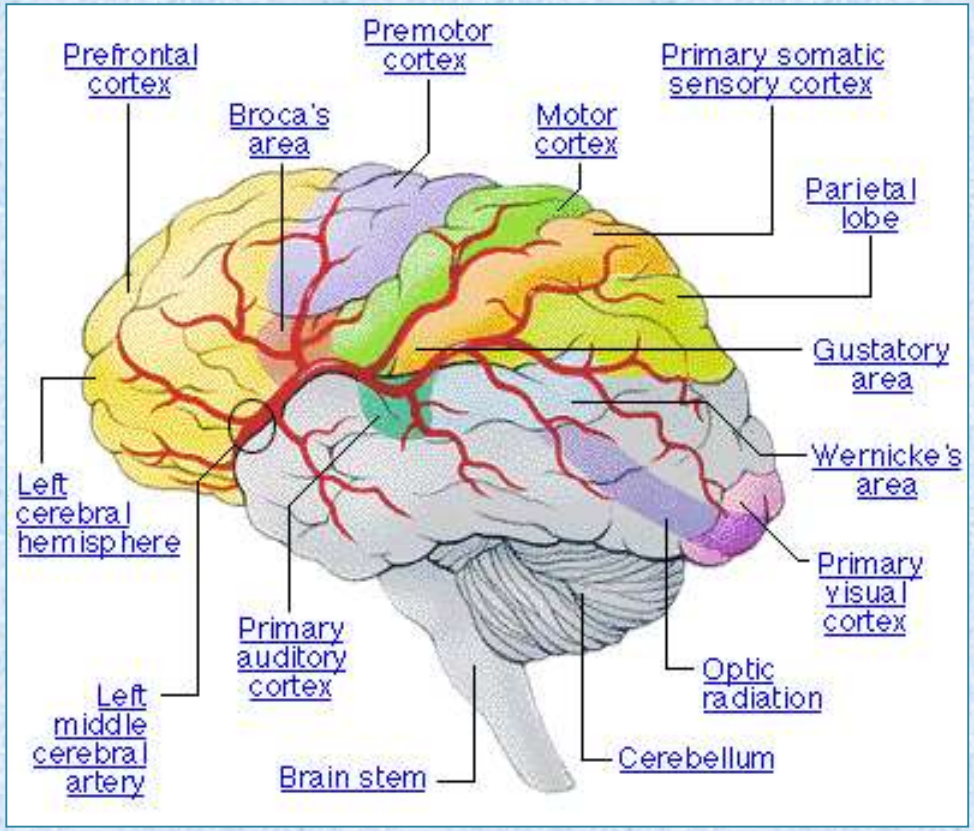
VLSI Implementability

The massively parallel nature of a neural network makes it well suited for implementation using VLSI (very-large-scale-integrated) technology.

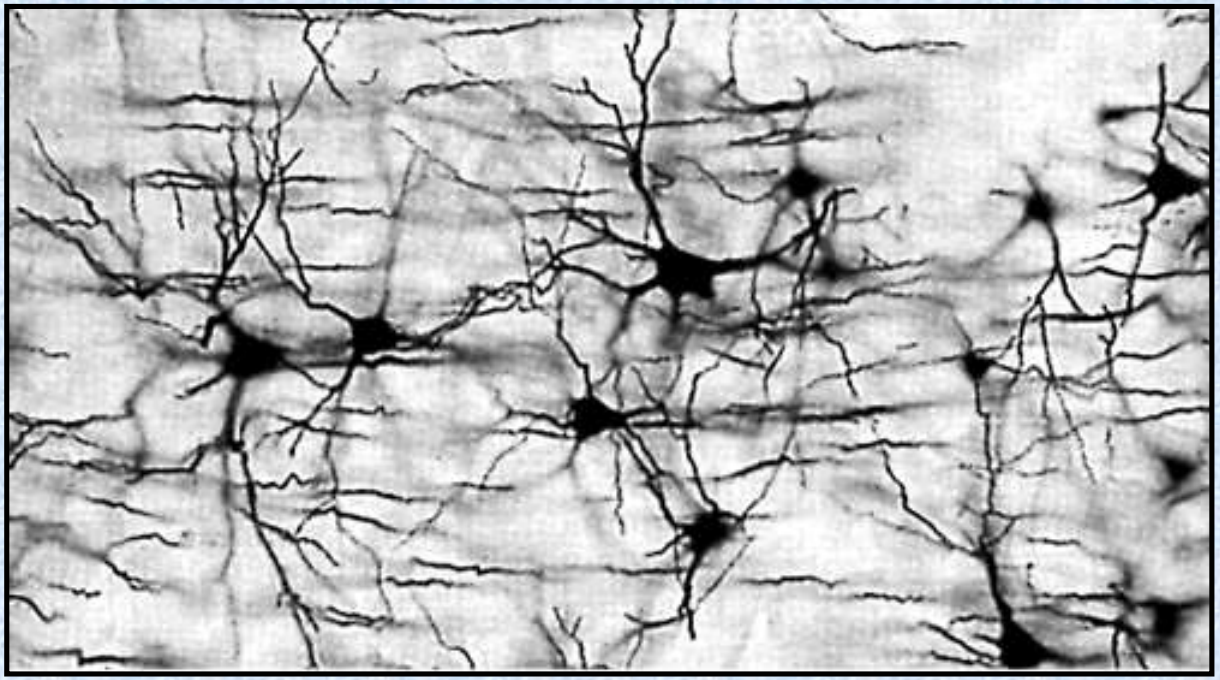
HUMAN BRAIN



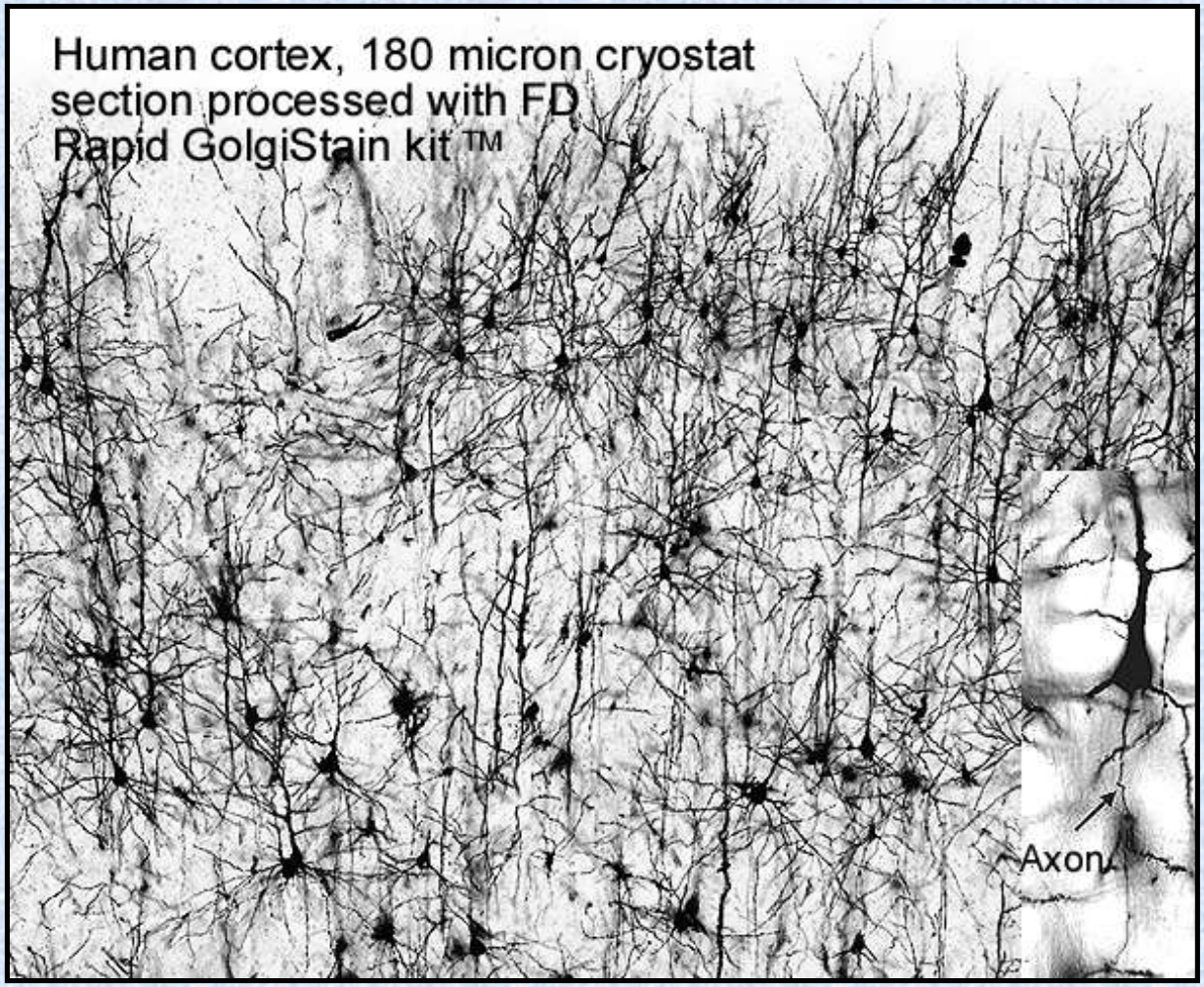
Block diagram representation of nervous system



Main parts of the human brain



A real neural network (Taken by an electron microscope)



http://www.neuralstainkit.com/images/in-p-1401b-NDT104.jpg

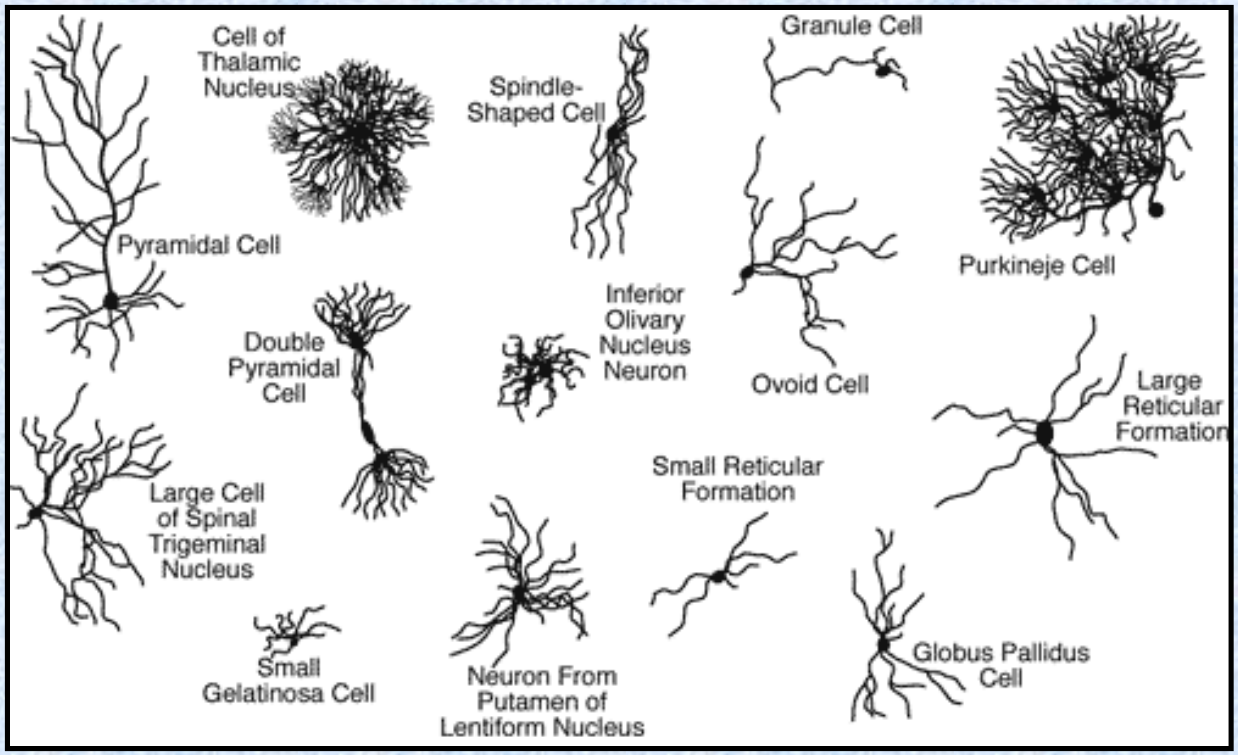
- The structural constituents of the of the brain are the nerve cells, known as **neurons**.

- There are approximately **10 billion neurons** in the human cortex and **60 trillion synapses** or connections

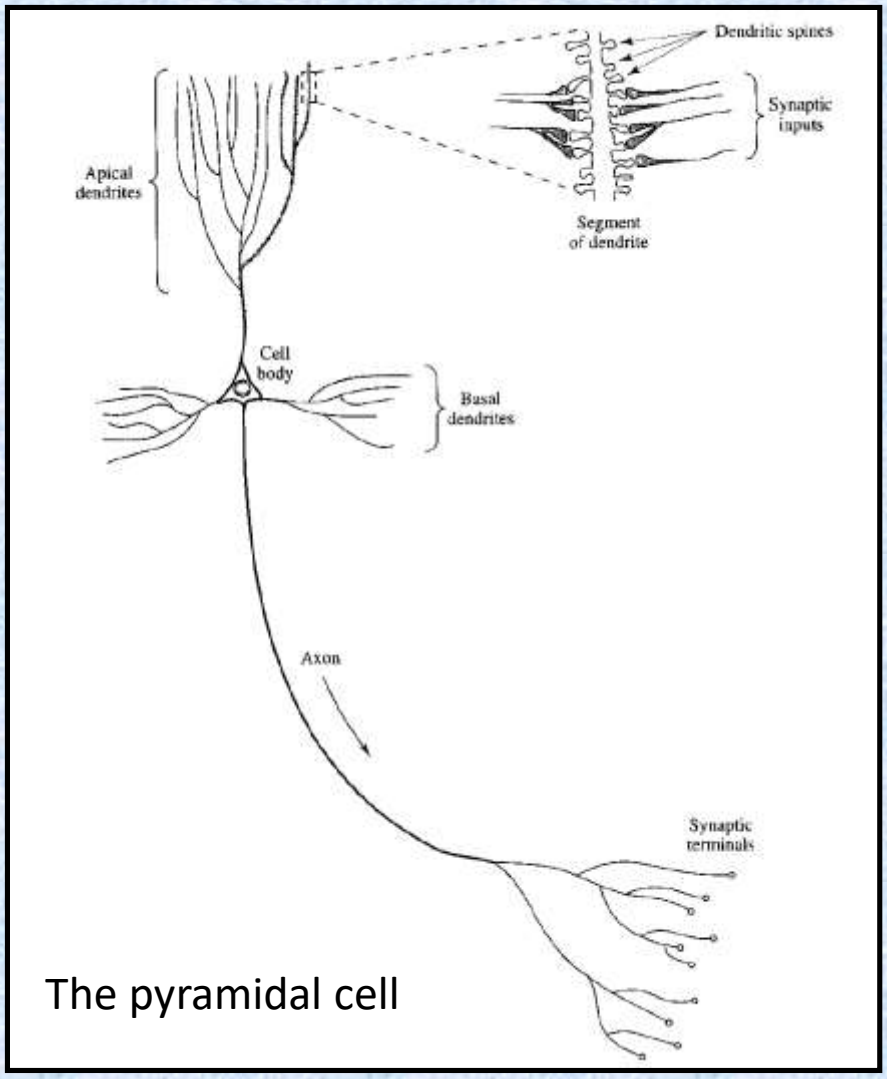
- Neurons are slower than silicon logic gates.

Events in a silicon chip happen in the nanosecond (10^{-9} s) range.
Neural events happen in the millisecond (10^{-3} s) range.

- The brain makes up for the relatively slow rate of operation of a neuron by having an **enormous number** of neurons with **massive interconnections** between them.



Basic Neuron Types



The pyramidal cell

Synapses are elementary structural and functional units that mediate the interaction between neurons.

The most common kind of a synapse is a **chemical synapse**.

A presynaptic process liberates a transmitter substance that diffuses across the synaptic junction between neurons and then acts on a postsynaptic process.

A synapse converts a presynaptic electrical signal into a chemical signal and then back into a postsynaptic electrical signal.

The pyramidal cell can receive 10000 or more synaptic contacts and it can project onto thousands of target cells.

The Neuron With Its Four Structures

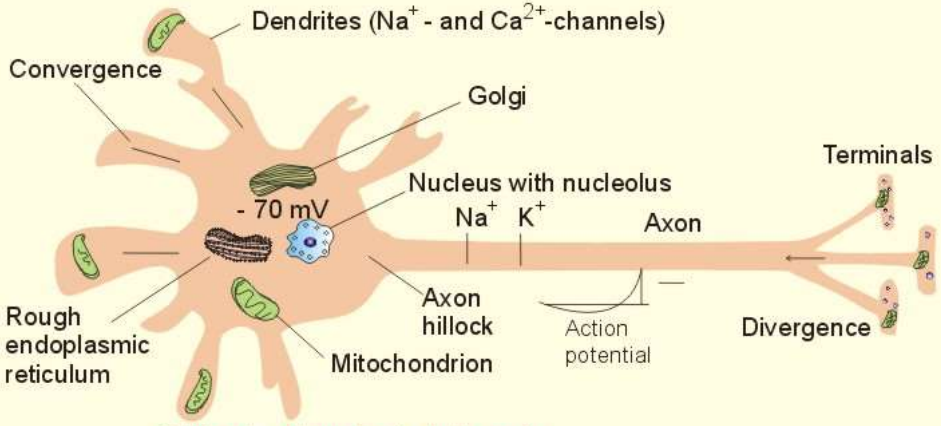
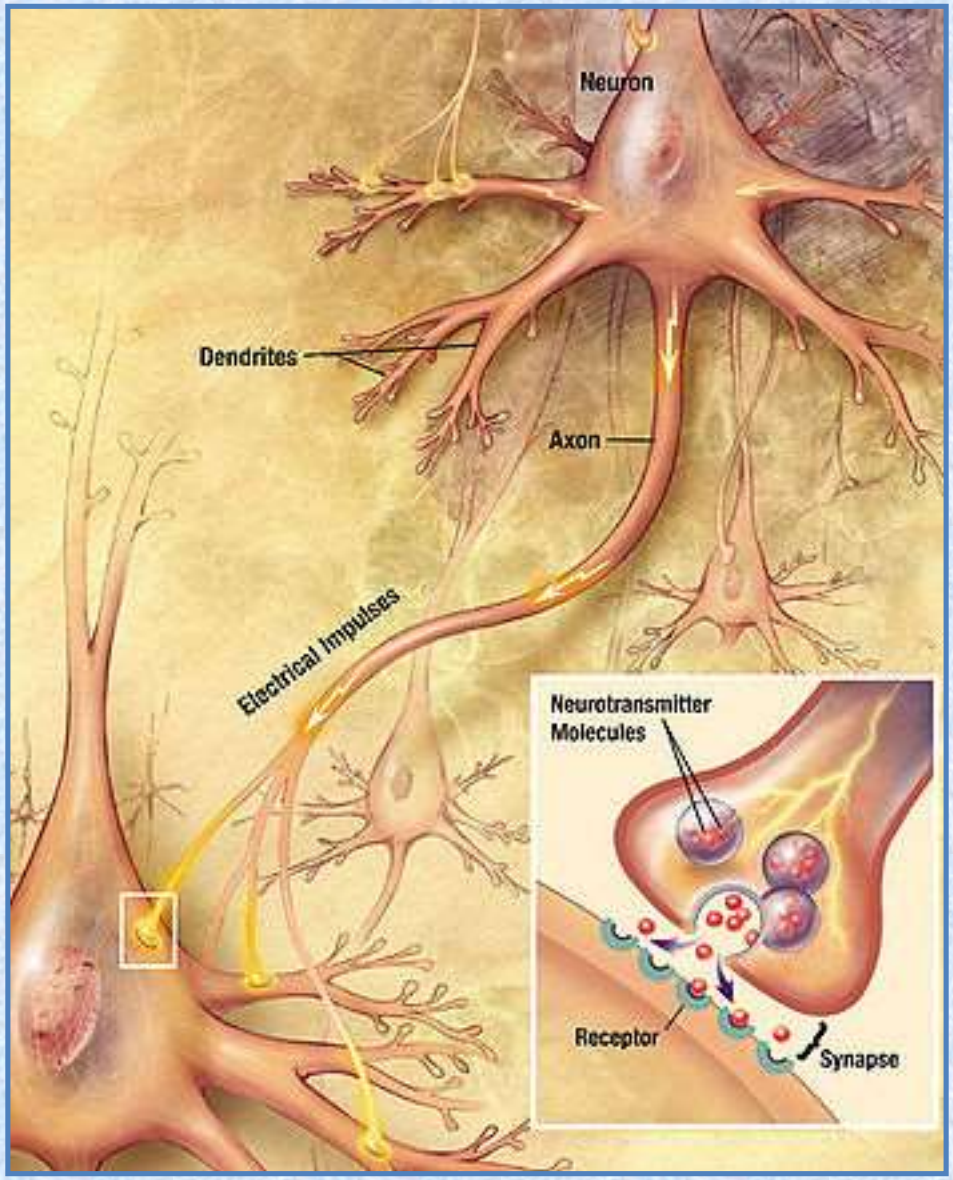
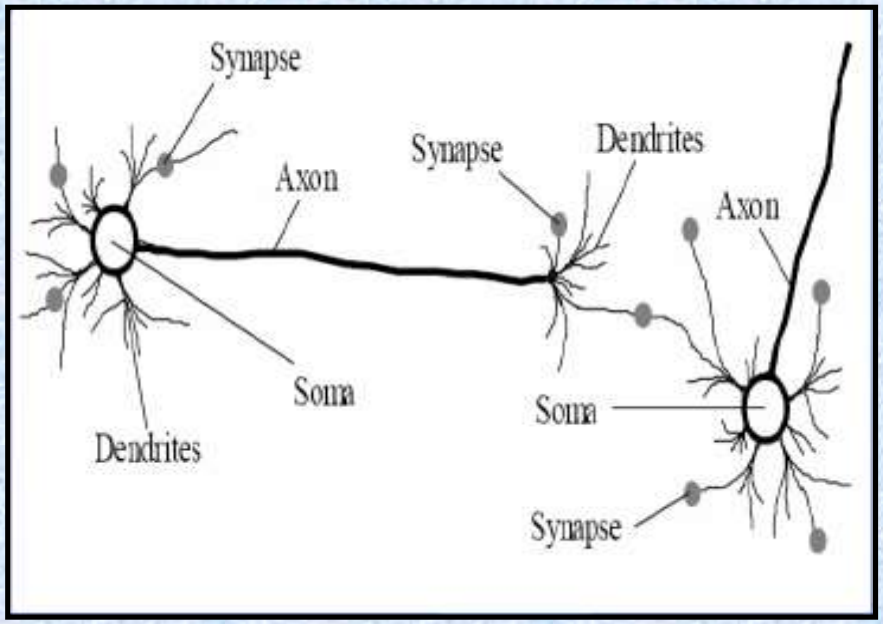


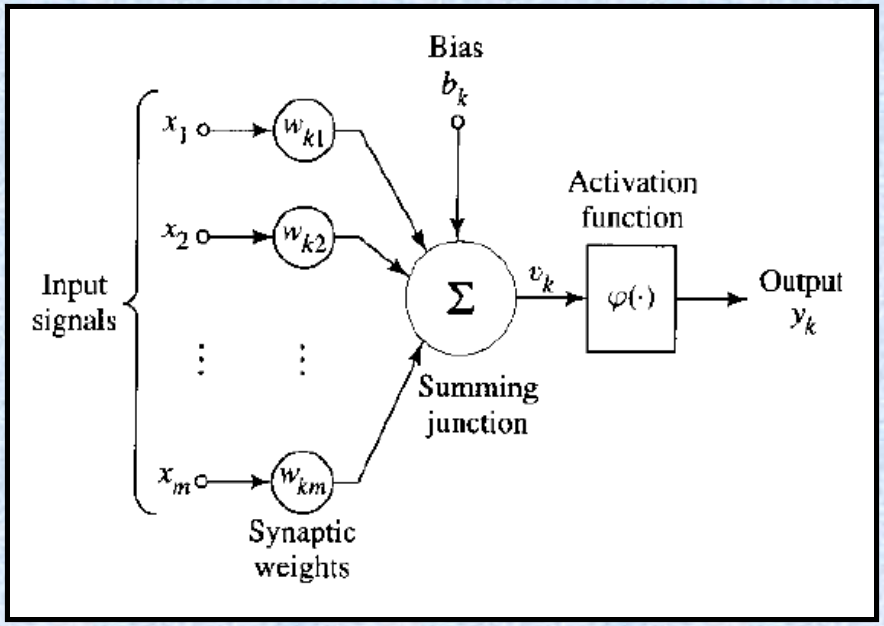
Fig.1-6

KMc


Neuron
Dendrites
Axon
Electrical Impulses
Neurotransmitter Molecules
Receptor
Synapse

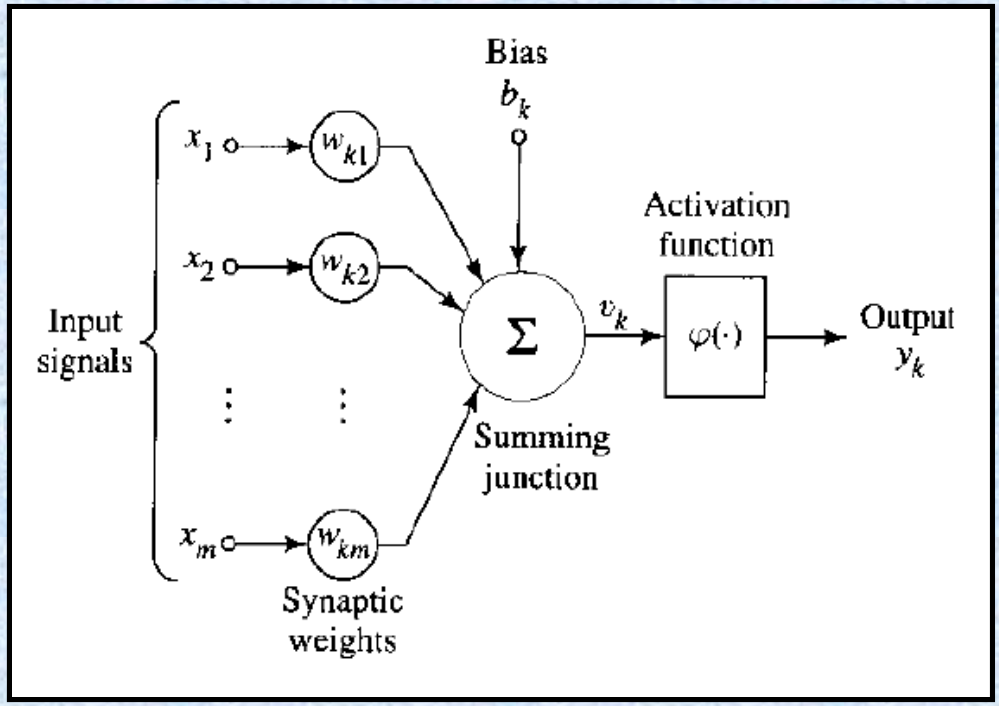


Biological Neuron



Artificial Neuron

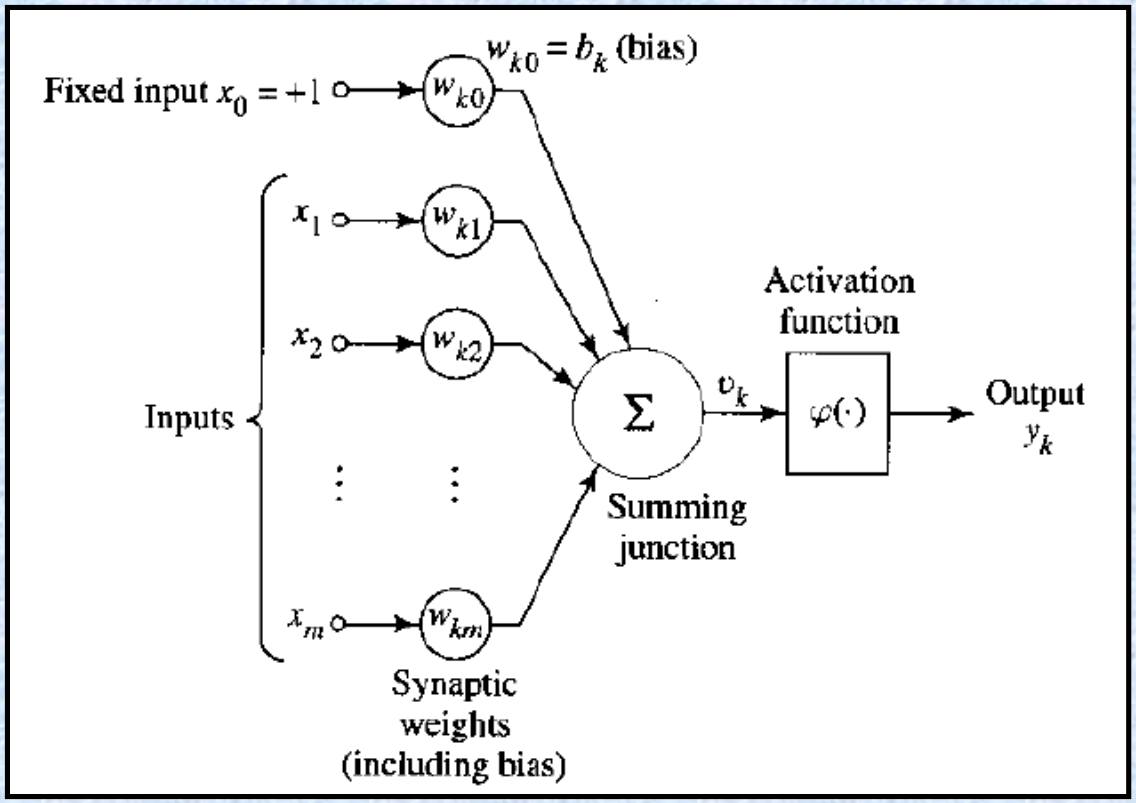
THE PERCEPTRON



$$u_k = \sum_{j=1}^{m} w_{kj} x_j$$

$$v_k = u_k + b_k$$

$$y_k = \varphi(v_k)$$

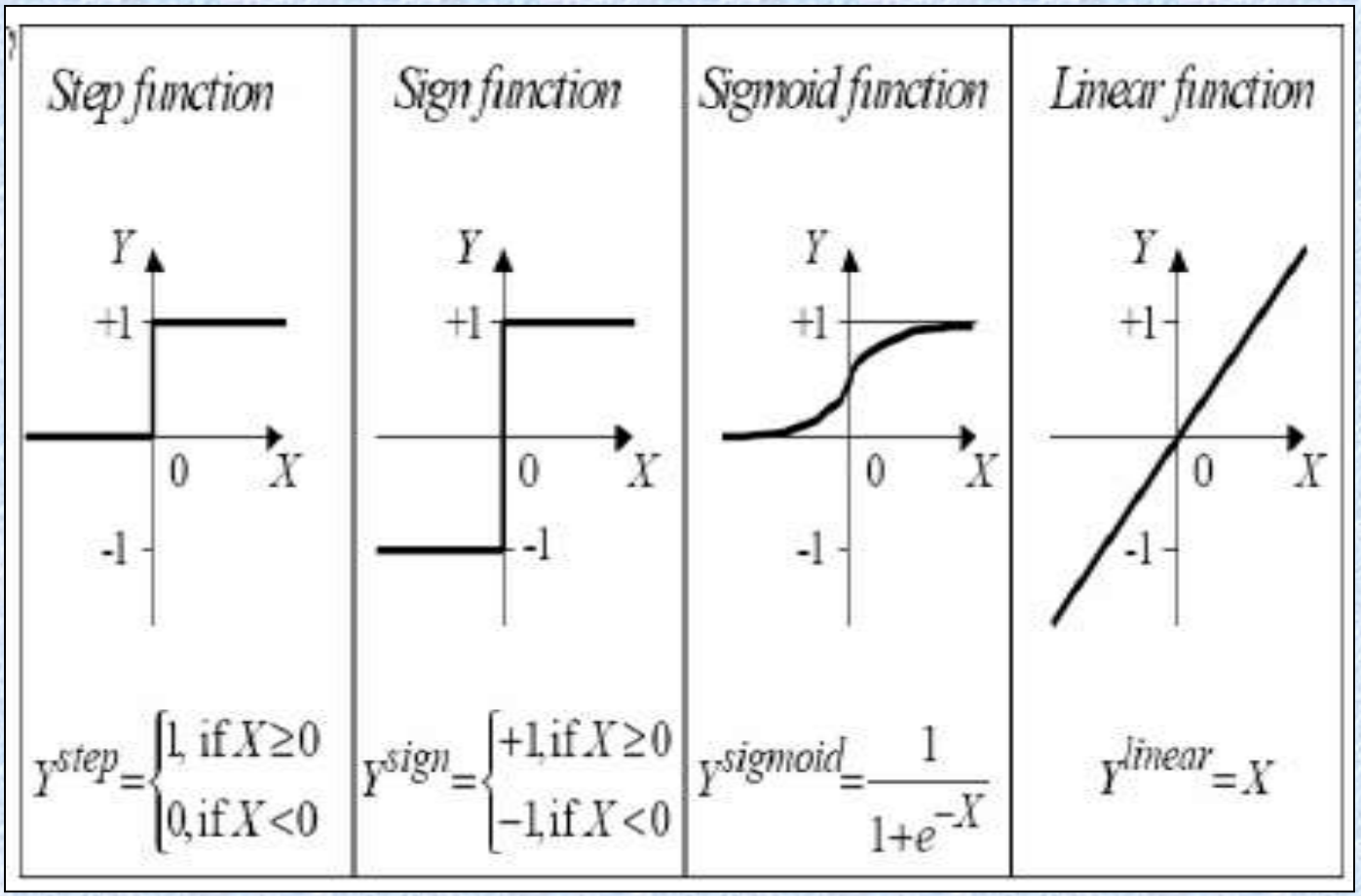


$$\vec{\mathbf{x}} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix}$$

$$\vec{\mathbf{w}} = \begin{bmatrix} w_{k0} & w_{k1} & w_{k2} & \dots & w_{kn} \end{bmatrix}$$

$$v_k = \sum_{i=1}^{n} w_{ki} x_i + w_0 = \vec{\mathbf{w}}.\vec{\mathbf{x}}$$

$$y_k = \varphi(v_k) = \begin{cases} 1 & v_k \geq 0 \\ 0 & v_k < 0 \end{cases}$$



Activation functions

Historical Notes

First studies on neural networks began with the pioneering work of **McCulloch and Pitts** (1943)

McCulloch was a neuroanatomist and Pitts was a mathematician.

They combined the studies of **neurophysiology** and **mathematical logic** to describe the logical calculus of neural networks in their classical paper of 1943.

In 1958 Rosenblatt introduced the **perceptron**.
In 1960, Widrow and Hoff developed the **least means –square** (LMS) method to train the perceptron.

In 1969, Minsky and Papert wrote a book to demonstrate mathematically that there are fundamental limits on what single-layer perceptrons can compute.

1970s were pessimistic. In addition to the negative effect of Minsky and Papert's book, there were technological limitations and financial problems.

In 1980s, there was a resurgence of interest in neural networks.

In 1982, Hopfield used the idea of an energy function and worked on **recurrent neural networks**.

In 1982, Kohonen published a paper on **self-organising maps**.

In 1983, Barto, Sutton and Anderson published a paper on **reinforcement learning.**

In 1986, Rumelhart, Hinton and Williams developed the famous **backpropagation algorithm**.

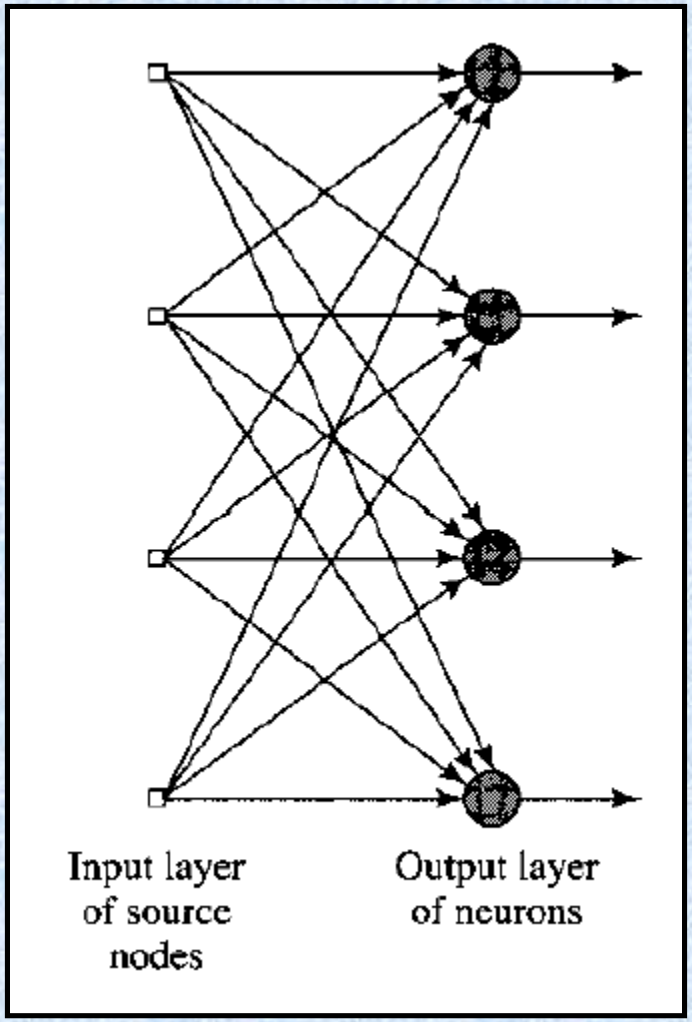
In early1990s, Vapnik and his coworkers invented **support vector machines** for solving pattern recognition, regression and density estimation problems.

NOW, there is a resurgence of interest in neural networks:
Deep neural networks that operate on **big data** without **preprocessing**

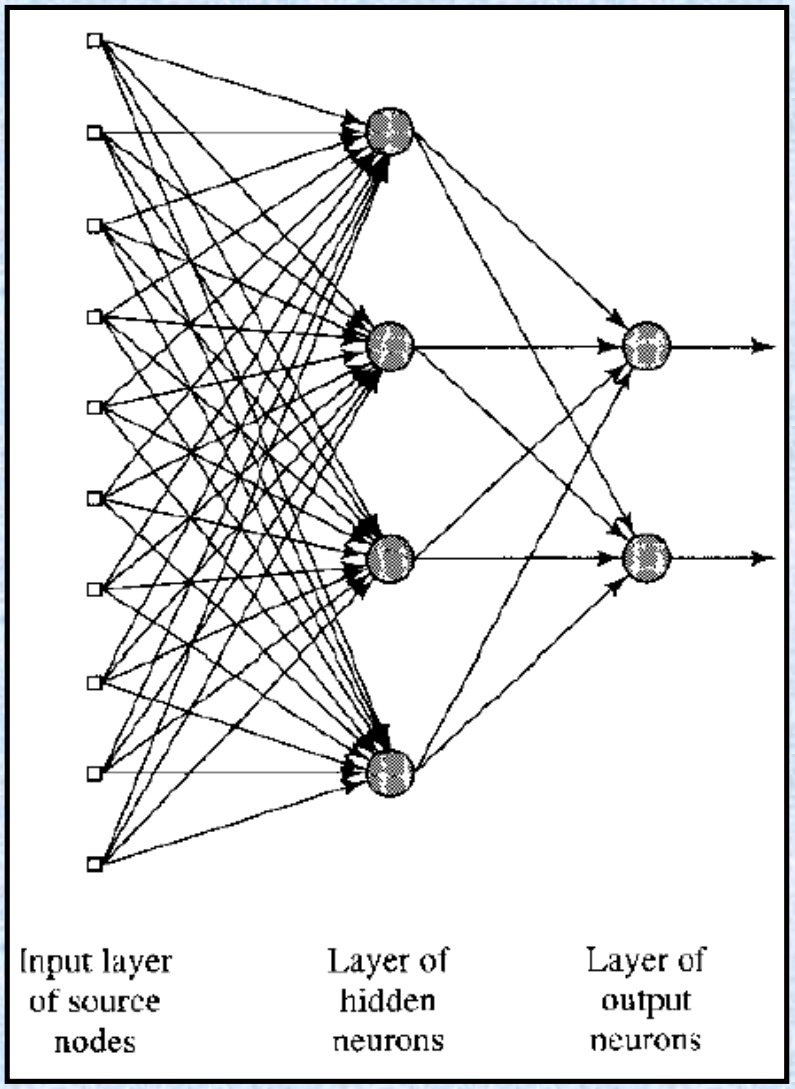
NETWORK ARCHITECTURES

1. Single-Layer Feedforward Networks
2. Multi-Layer Feedforward Networks
3. Recurrent Networks

Single-Layer Feedforward Networks



Multi-Layer Feedforward Networks

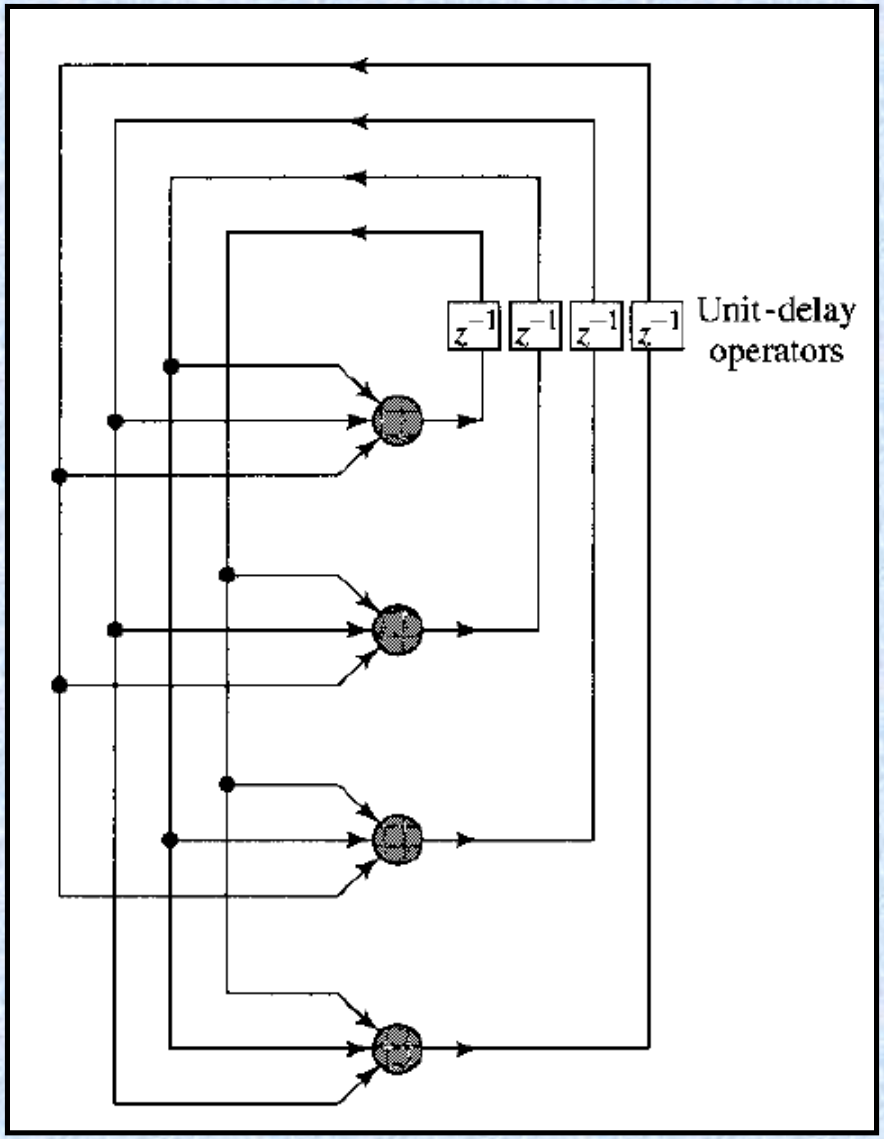


This is a 10-4-2 network.

By adding hidden layers, the network is able to extract higher-order statistics.

The network acquires a global perspective despite its local connections due to the extra set of synaptic connections and neural interactions.

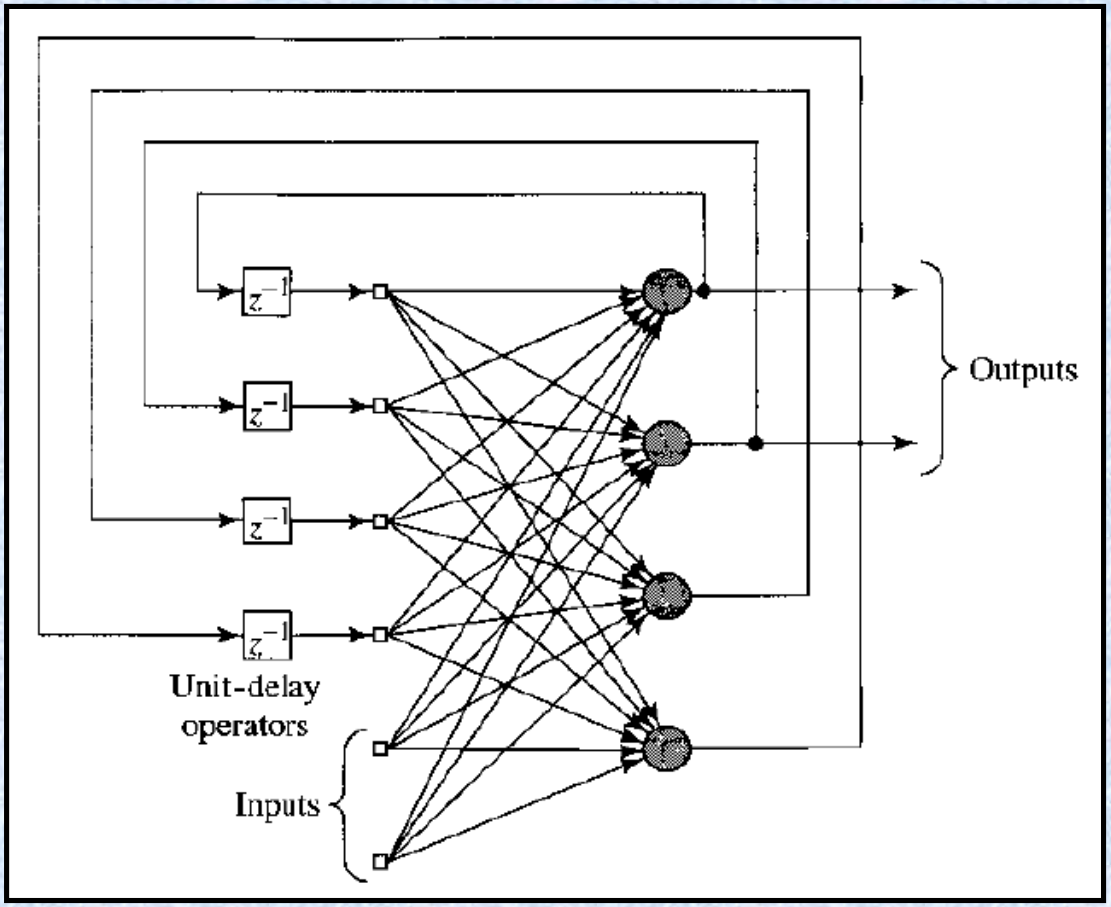
Recurrent Neural Networks



A recurrent neural network is different from a feedforward neural network in that it contains at least one feedback loop.

Hopfield network

Recurrent neural network with no self-feedback loops
and no hidden neurons



Recurrent neural network with hidden neurons

Artificial Neural Networks: Design Options

Architectures

- How the neurons are connected to each other
- The number of layers
- The number of neurons in each layer
- Feedforward or recurrent architectures

Properties of the neurons

- The relation between the input and the output of the neuron

The properties of the nonlinear function

Learning Mechanism

- How the neuron weights are updated during teaching
- Supervised Learning - Unsupervised learning
- Choosing among various learning algorithms
- Determine training data and testing data

SOME APPLICATIONS OF NEURAL NETWORKS

Function Approximation

Consider a nonlinear input-output mapping described by:

$$\mathbf{d} = \mathbf{f}(\mathbf{x})$$

The vector valued function **f(.)** is unknown.
We are given the set of labeled examples:

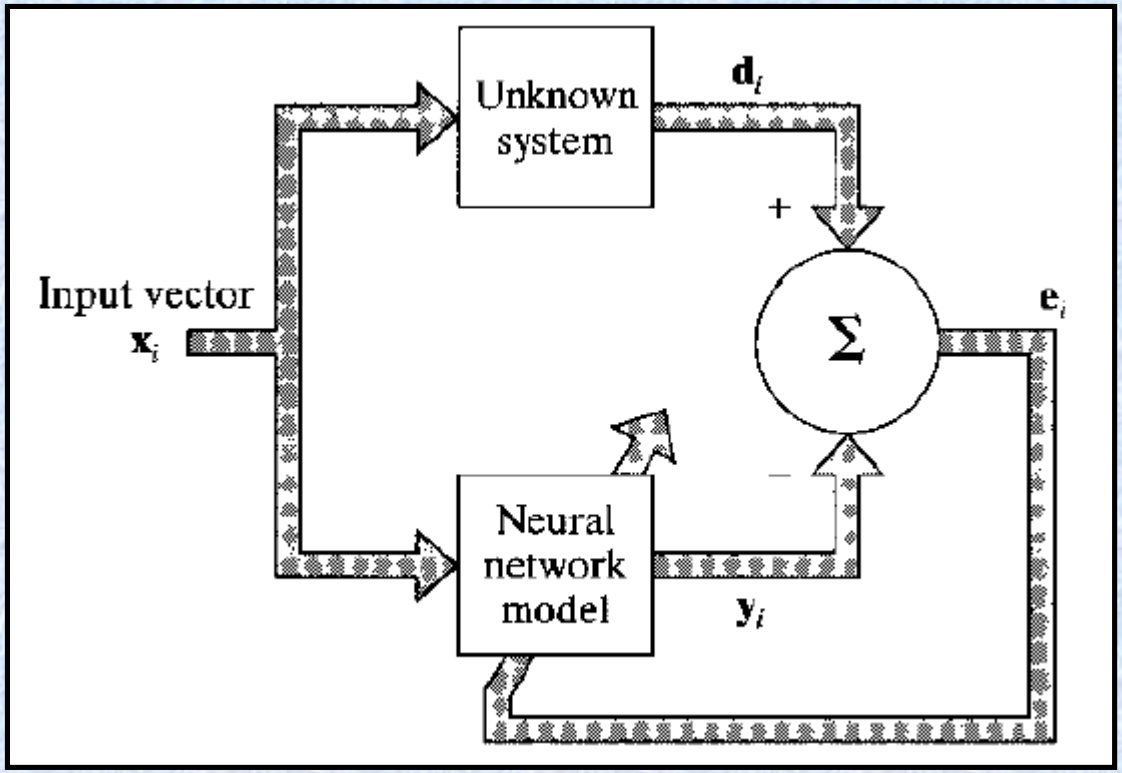
$$\tau = \{(\mathbf{x}_i, \mathbf{d}_i)\}_{i=1}^{N}$$

The requirement is to design a neural network whose input-output Mapping described by **F(.)** is close to **f(.)** in the Euclidean sense.

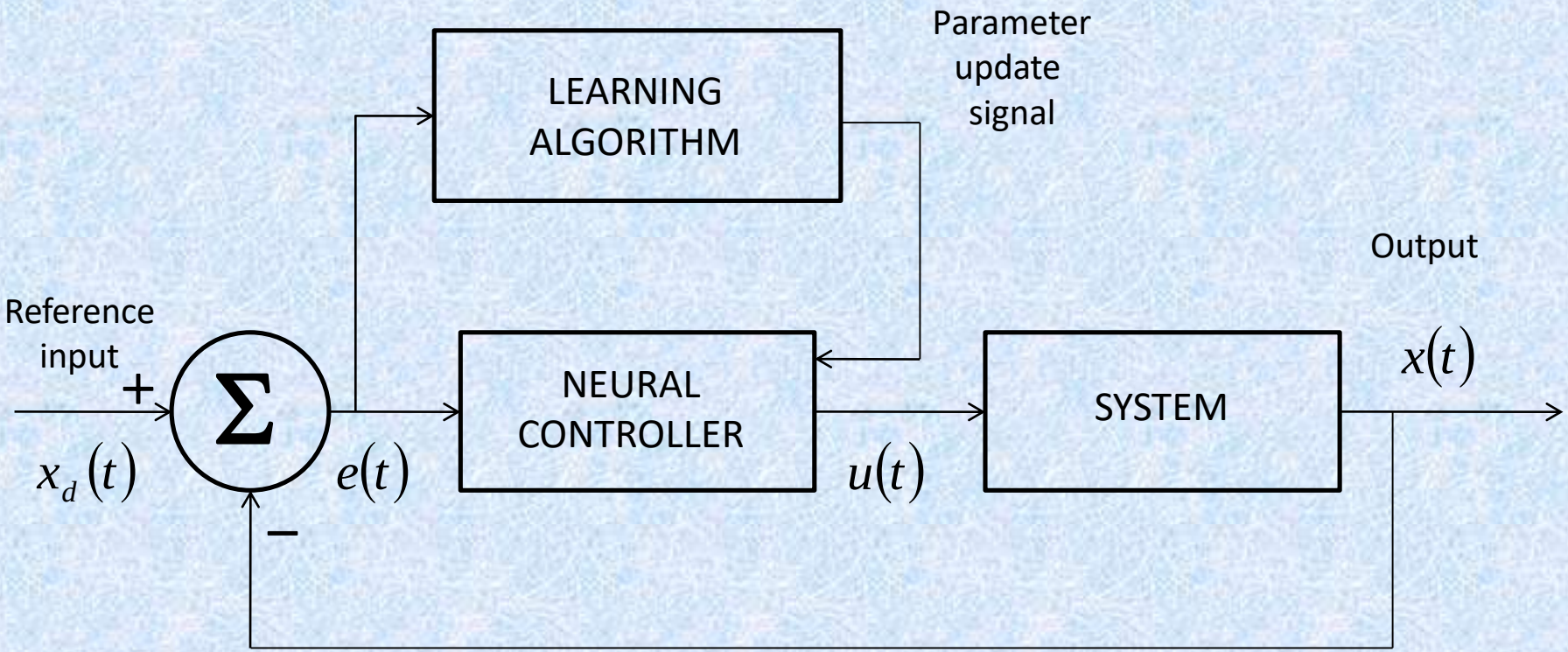
$$\|\mathbf{F}(\mathbf{x}) - \mathbf{f}(\mathbf{x})\| < \varepsilon \qquad \text{for all } \mathbf{x}$$

(NN as a universal approximator – Stone-Weierstrass Theorem)

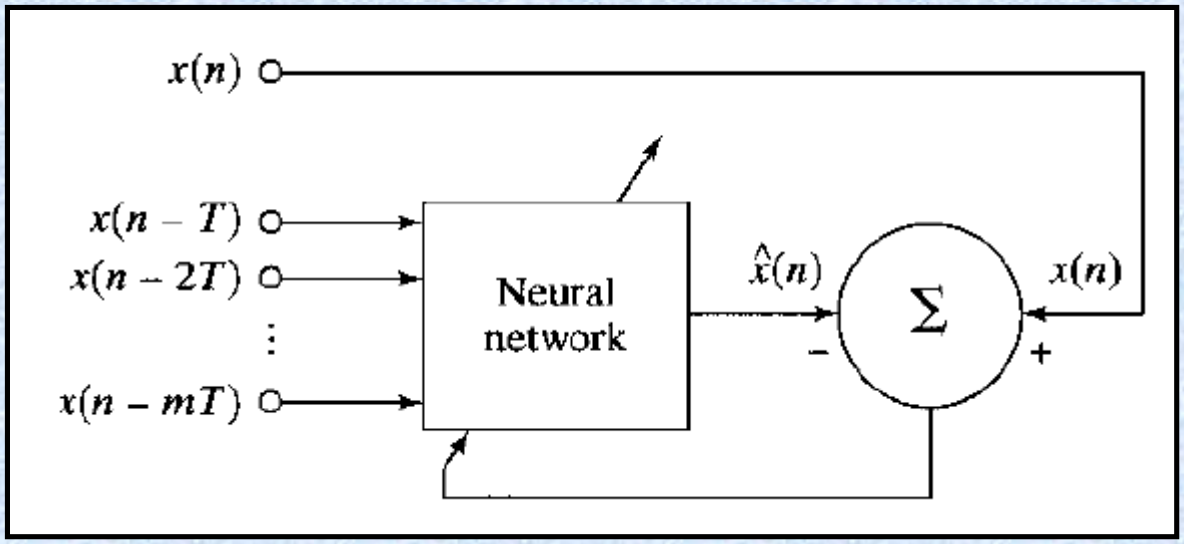
System Identification



Control

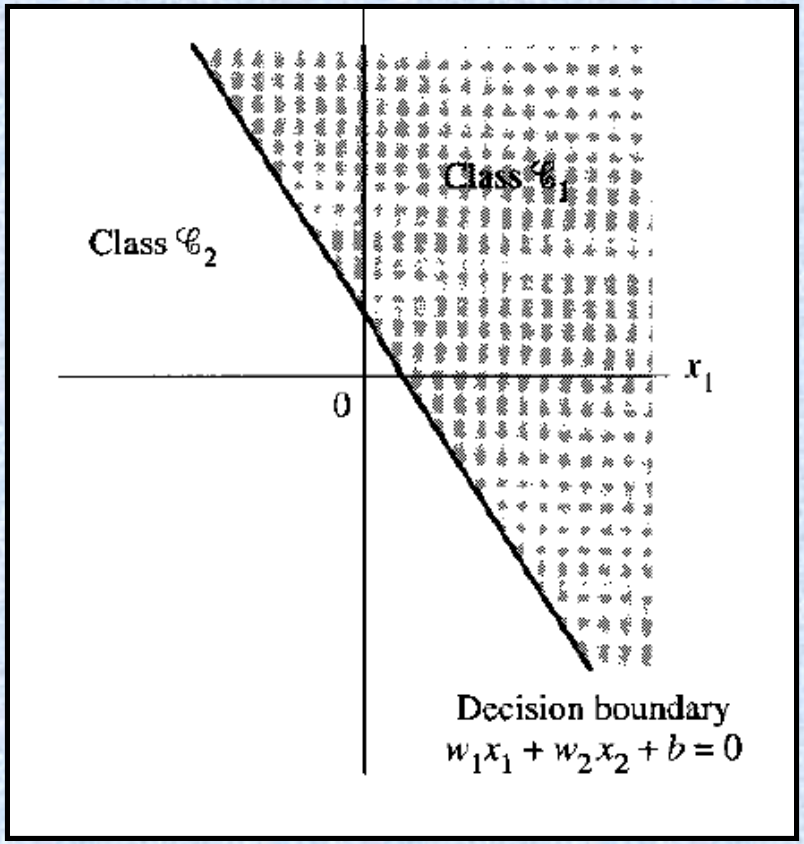

LEARNING
ALGORITHM
Parameter
update
signal
Output
Reference
input
+
Σ
−
$x_d(t)$
$e(t)$
NEURAL
CONTROLLER
$u(t)$
SYSTEM
$x(t)$

Nonlinear Prediction



The requirement in the prediction problem is to predict the present value of $x(n)$ of a process, given past values of the process that are uniformly spaced in time as shown by $x(n\text{-}T)$, $x(n\text{-}2T), \ldots\ldots\ldots x(n\text{-}mT)$, where T is the sampling period and m is the prediction order.

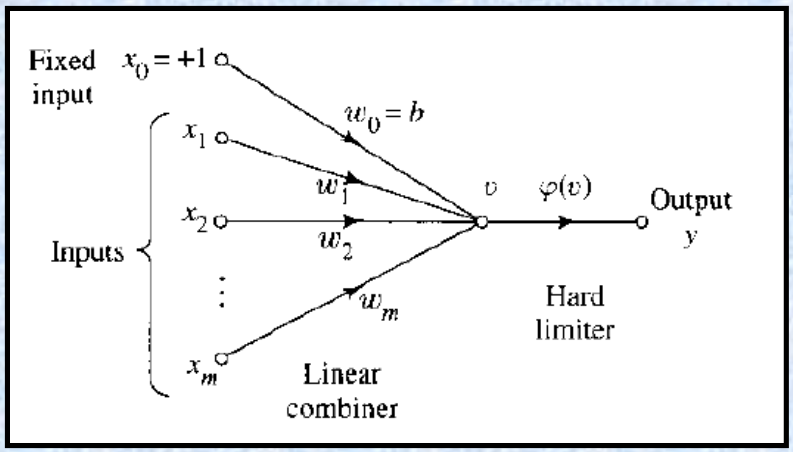
Pattern Classification



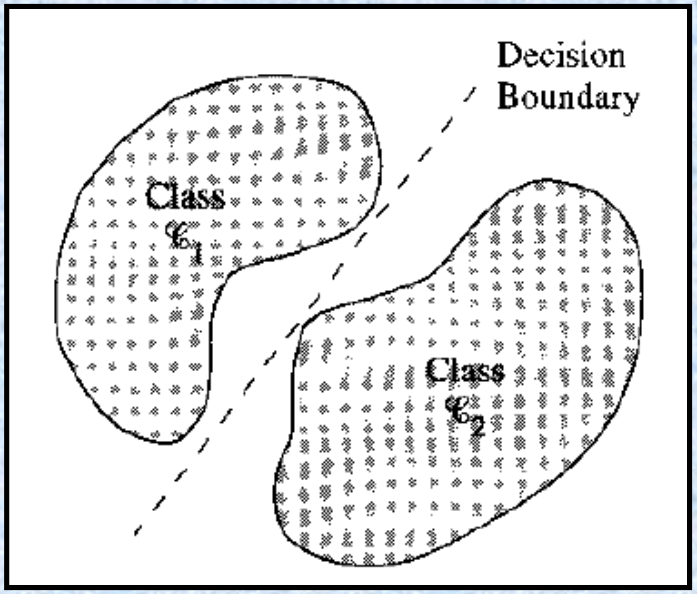
Pattern classification is formally defined as the process whereby a received pattern/signal is assigned to one of a prescribed number of **classes (categories).**

Pattern recognition performed by a neural network is statistical in nature, with the patterns being represented by points in a multidimensional **decision space**.

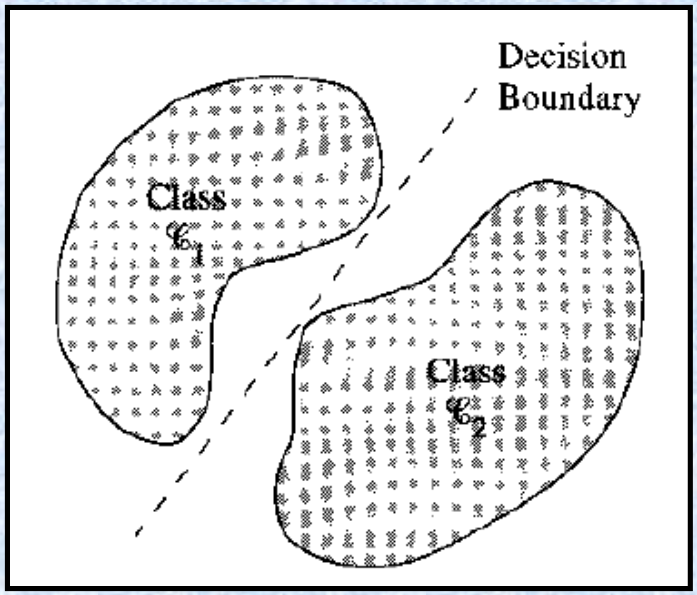
The decision space is divided into regions, each one of which is associated with a class.



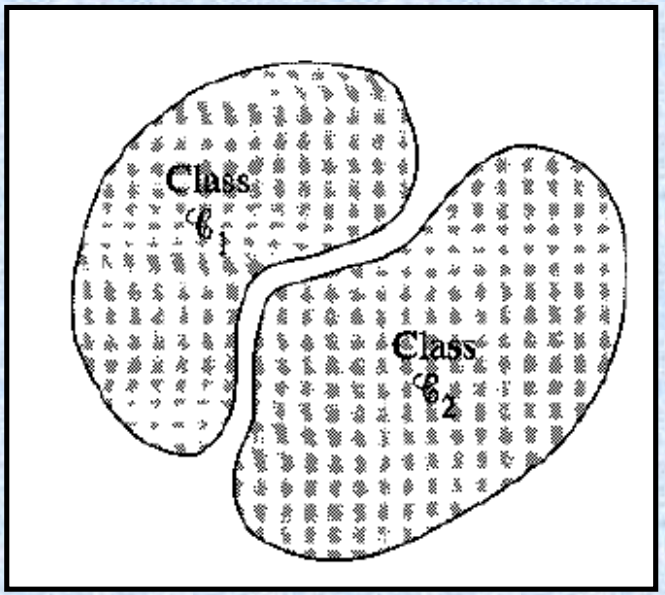
Signal-flow graph of the perceptron



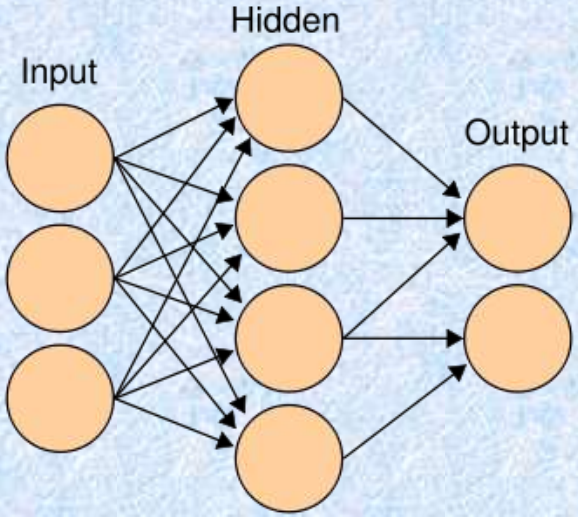
A pair of **linearly separable** patterns

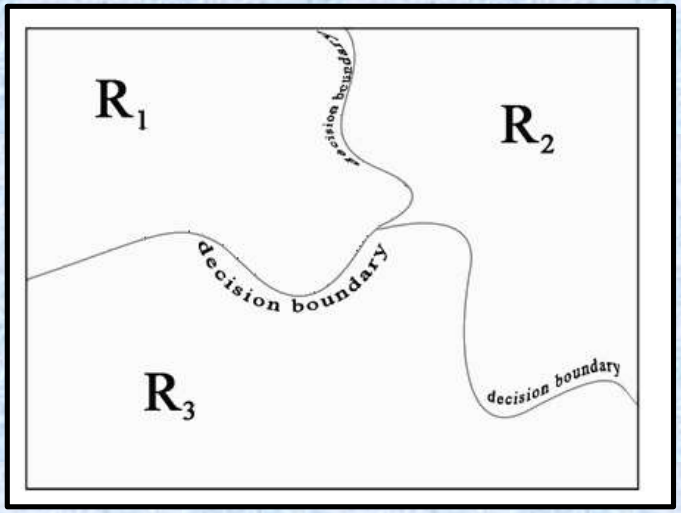


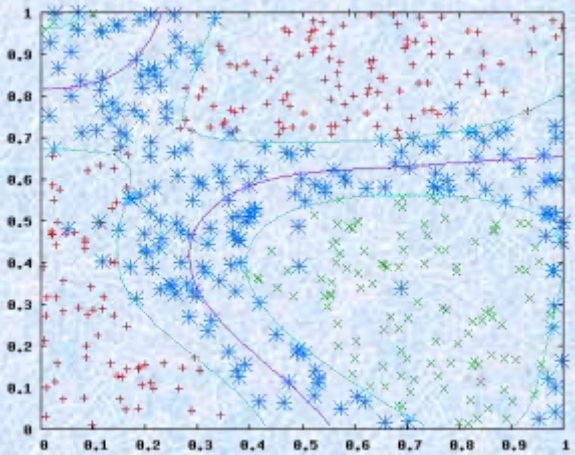
A pair of linearly separable patterns

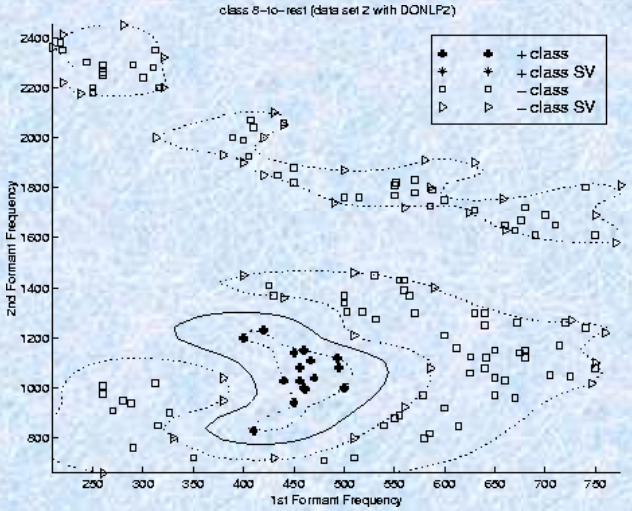


A pair of non-linearly separable patterns


Input
Hidden
Output


R1
R2
R3
decision boundary
decision boundary
decision boundary




class 8-to-rest (data set 2 with DONLP2)
+class
+class SV
−class
−class SV
2nd Formant Frequency
1st Formant Frequency

This was a brief introduction to intelligent controllers and neural networks.

Next week, we'll start studying neural networks in more detail.

And our first topic will be learning processes …

KON 426E
INTELLIGENT CONTROL SYSTEMS

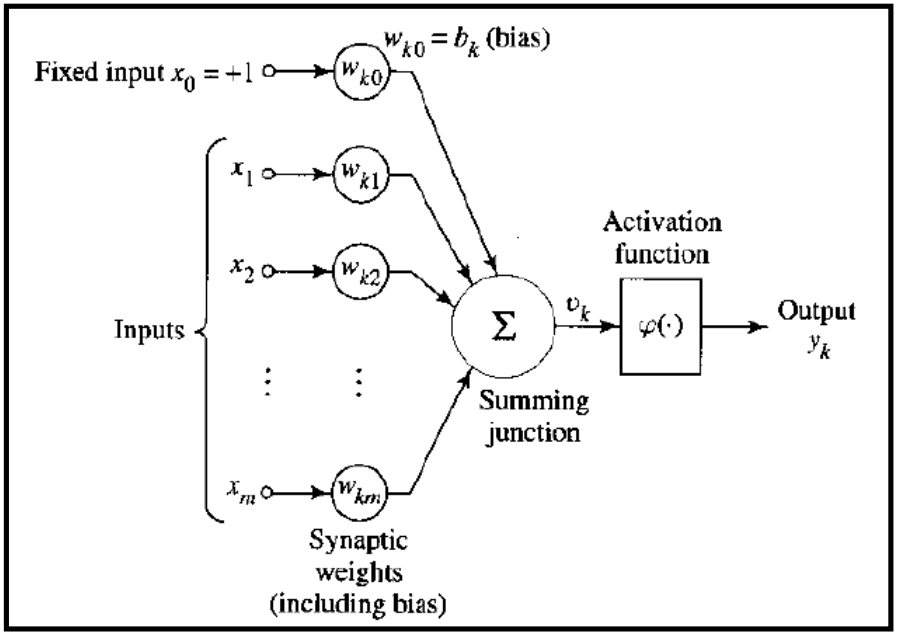
LECTURE 2

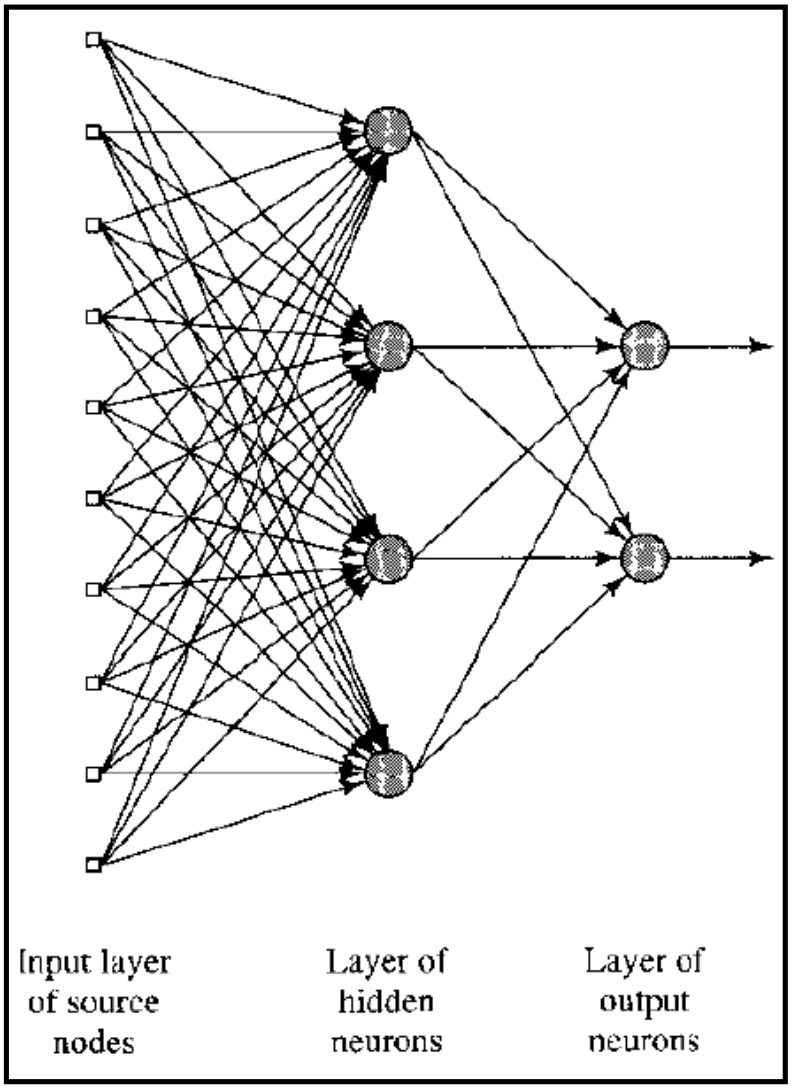
28/02/2022

Learning

- The property that is of primary significance for a neural network is the ability of the network to learn from its environment and to improve its performance through learning. The improvement in performance takes place over time in accordance with some prescribed measure.
- A neural network learns about its environment through an interactive process of adjustments applied to its synaptic weights and bias levels.

* Reference: Haykin, S., Neural Networks and Learning Machines, 2009.





Parameters to be learned:

Weights and bias values

Definition of Learning
(Mendel and McClaren, 1970)

- Learning is a process by which the free parameters of a neural network are adapted through a process of stimulation by the environment in which the network is embedded. The type of learning is determined by the manner in which the parameter changes take place.

*Reference. Haykin, S.,Neural Networks, 1994.

This definition of the learning process implies the following sequence of events:

1. The neural network is **stimulated** by an **environment**.

2. The neural network **undergoes changes** in its free parameters as a result of this stimulation.

3. The neural network **responds in a new way** to the environment because of the changes that have occurred in its internal structure.

These steps are repeated iteratively and eventually the neural network will behave in the desired way.

*Reference. Haykin, S.,Neural Networks, 1994.

- A prescribed set of well-defined rules for the solution of a learning problem is called a **learning algorithm**.

- There are various **learning algorithms**. They differ from each other in the way in which the adjustment to a synaptic weight of a neuron is formulated.

- Another factor to be considered is the manner in which a neural network relates to its environment. This leads us to the concept of a **learning paradigm.** Learning paradigm refers to a model of the **environment** in which the neural network operates.

*Reference. Haykin, S.,Neural Networks, 1994.

Learning Paradigms

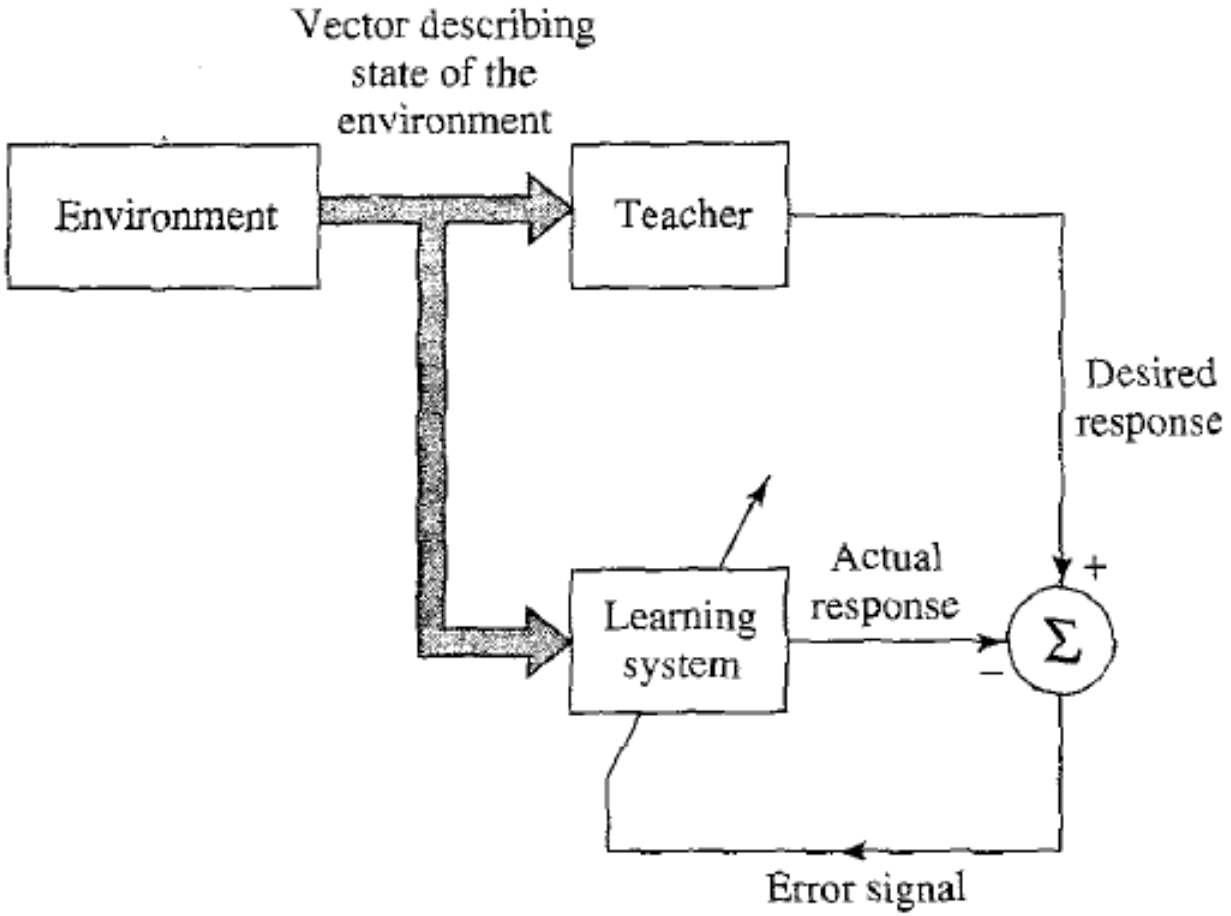
Learning With a Teacher
(Supervised learning)

Learning Without a Teacher

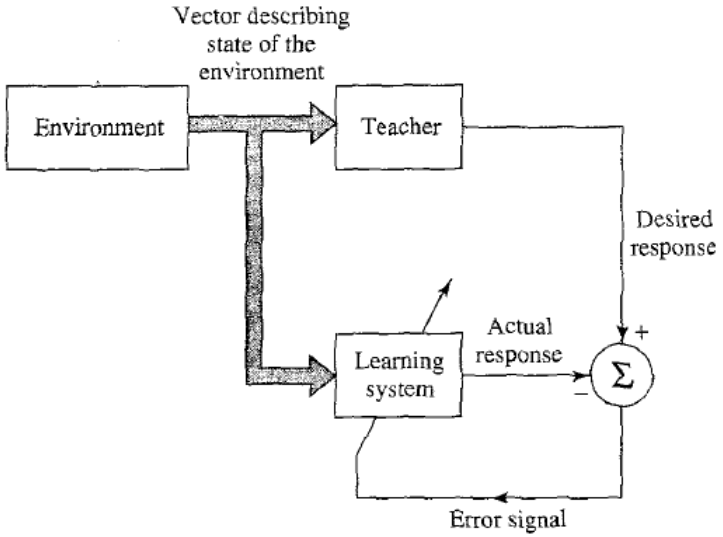
Reinforcement Learning

Unsupervised Learning

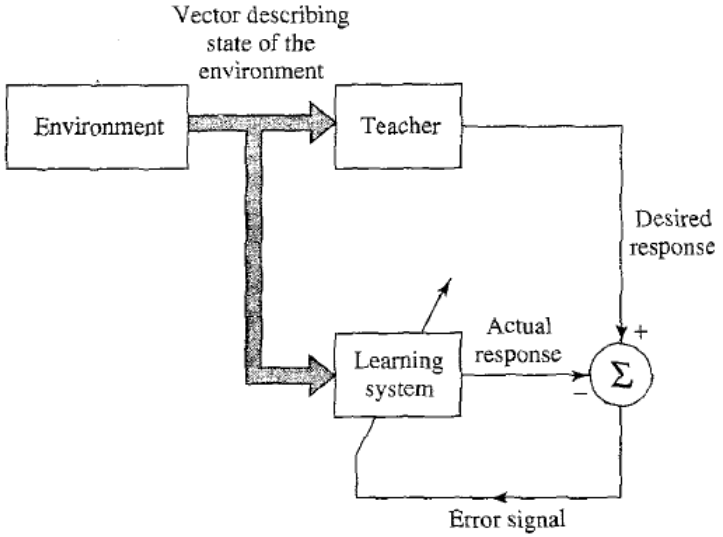
Supervised Learning (Öğreticili Öğrenme)



*Reference. Haykin, S.,Neural Networks, 1994.



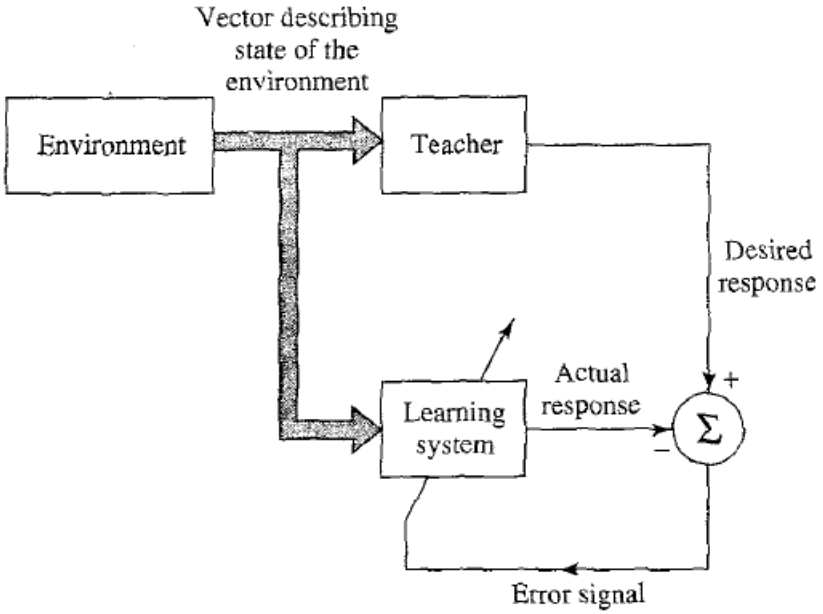
- The teacher has the knowledge of the **environment**.
- This knowledge is represented in the form of **input-output examples**.
- These input-output examples constitutes the **training data set**.
- However, the environment is unknown to the NN.
- Initially, the NN has **randomly** assigned **weights and bias values**.
- Both the teacher and the NN are exposed to a training vector from the environment. (an example from the input-output data set)



- The teacher provides the correct output corresponding to the input vector.
- This output vector is the desired response that represents the optimum action to be performed by the NN.
- Given the input vector the NN computes an ouput with its weights and bias values.
- An error signal is computed between the output of teacher and NN.

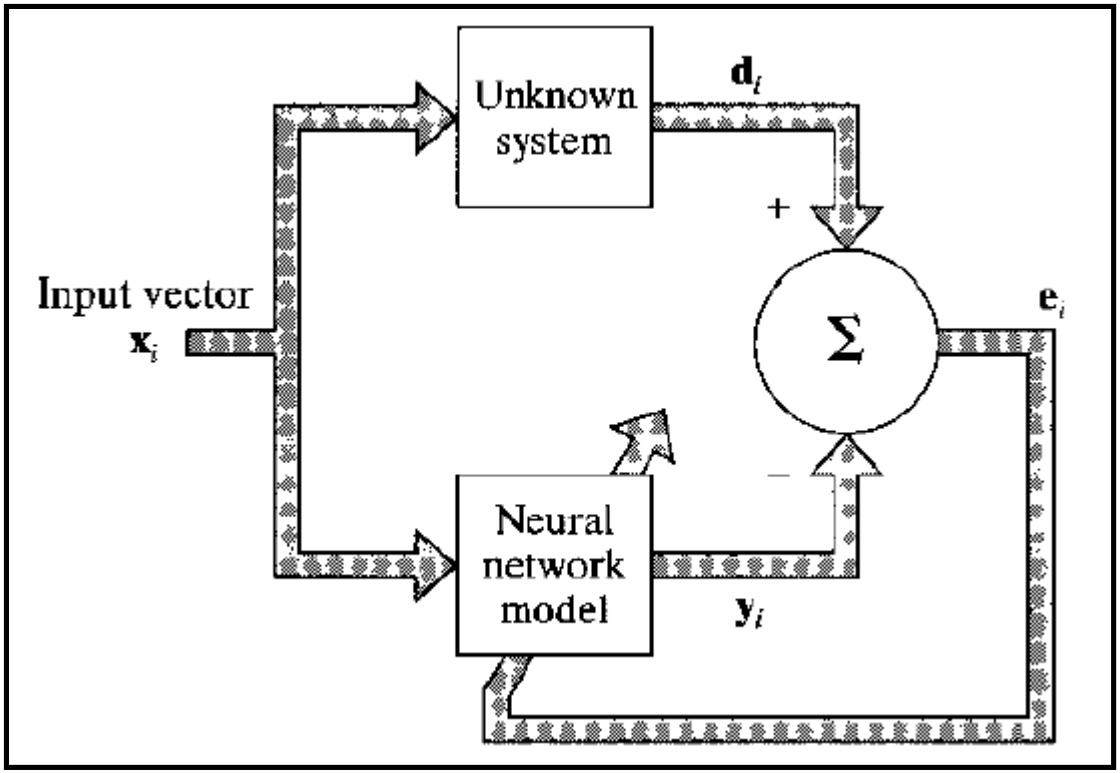
Error= Desired response – Actual response of the NN

- The network parameters are adjusted under the combined influence of the training vector and the error signal.



- This adjustment is carried out **iteratively** in a **step-by-step** fashion.
- Eventually the NN **emulates** (behaves in the same way) the teacher.
- The emulation is assumed to be optimal in some statistical sense.
- So, the knowledge of the environment available to the teacher is **transferred to the NN** through training as fully as possible.
- Now, the teacher can be replaced by the NN.

Example: **System Identification**



Reinforcement Learning
(Pekiştirmeli/Yinelemeli Öğrenme)

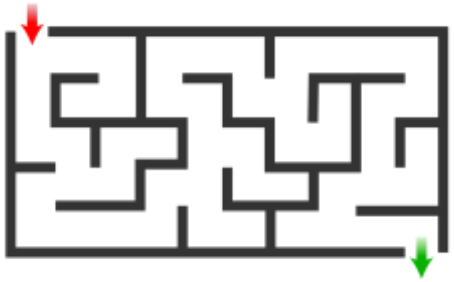
- **Reinforcement learning** (**RL**) is an area of machine learning concerned with how intelligent agents ought to take actions in an environment in order to maximize the notion of cumulative reward.

- The environment is typically stated in the form of a Markov decision process (MDP), and many reinforcement learning algorithms for this context use dynamic programming techniques.

*Reference: Wikipedia

- Consider we want to build a machine that learns to play chess. In this case we cannot use a supervised learner for two reasons.
- First, it is very costly to have a teacher that will take us through many games and indicate us the best move for each position.
- Second, in many cases, there is no such thing as the best move; the goodness of a move depends on the moves that follow.
- A single move does not count; a sequence of moves is good if after playing them we win the game.
- In chess, the only feedback is at the end of the game when we win or lose the game.

*Reference: Alpaydın E, Introduction to Machine Learning, 2010.



- Another example is a robot that is placed in a maze.
- The robot can move in one of the four compass directions and should make a sequence of movements to reach the exit.
- As long as the robot is in the maze, there is no feedback and the robot tries many moves until it reaches the exit and only then does it get a reward.
- In this case there is no opponent, but we can have a preference for shorter trajectories, implying that in this case we play against time.

*Reference: Alpaydın E,Introduction to Machine Learning, 2010.

- These two examples have a number of points in common:

  There is a decision maker, called the **agent**, that is placed in an **environment**

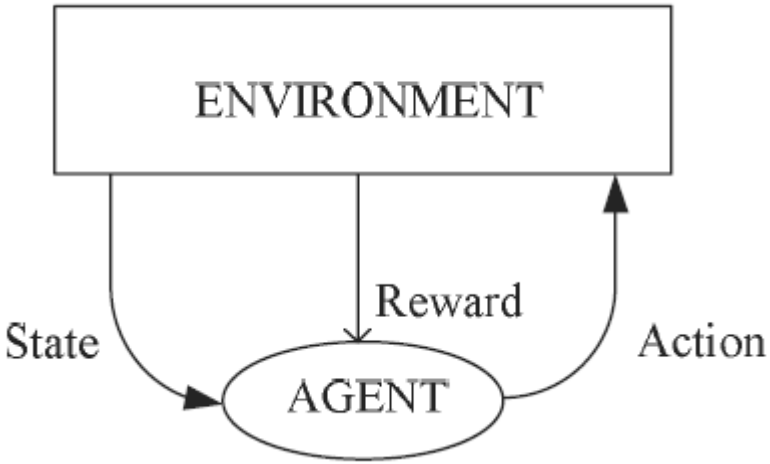
  In chess, the game-player is the decision maker and the environment is the board;

  In the second case, the maze is the environment of the robot.
- At any time, the environment is in a certain **state** that is

  one of a set of possible states—for example, the state of the board, the position of the robot in the maze.
- The **decision maker** has a set of **actions** possible: legal movement of pieces on the chess board, movement of the

  robot in possible directions without hitting the walls, and so forth.
- Once an action is chosen and taken, the state changes.

Elements of Reinforcement Learning



The learning decision maker is called the **agent**.
The agent **interacts** with the **environment** that includes everything outside the agent.
The agent has sensors to decide on its **state** in the environment and takes an action that modifies its state.
When the agent takes an action, the environment provides a **reward.**

*Reference: Alpaydın E,Introduction to Machine Learning, 2010.

- Time is **discrete** as *t = 0, 1, 2, . . .*, and $s_t \in S$ denotes the **state** of the agent at time t where S is the set of all possible states.
- $a_t \in \mathcal{A}(s_t)$ denotes the **action** that the agent takes at time t where $\mathcal{A}(s_t)$ is the set of possible actions in state s_t
- When the agent in state s_t takes the action a_t, the clock ticks, **reward** $r_{t+1} \in \Re$ is received, and the agent moves to the next state s_{t+1}.
- The problem is modeled using a Markov decision process (MDP).
- The reward and next state are sampled from their respective probability distributions, $p(r_{t+1}|s_t, a_t)$ and $P(s_{t+1}|s_t, a_t)$
- This is a Markov system where the state and reward in the next time step depend only on the current state and action.
- In some applications, reward and next state are **deterministic**, and for a certain state and action taken, there is one possible reward value and next state. In other applications, they may be **stochastic**.

- Depending on the application, a certain state may be designated as the **initial state** and in some applications, there is also an absorbing **terminal (goal) state** where the search ends.
- All actions in this terminal state transition to itself with probability 1 and without any reward.
- The sequence of actions from the start to the terminal state is an **episode**, or a **trial**.
- The **policy, π**, defines the agent's behavior and is a mapping from the states of the environment to actions: $\pi : S \rightarrow \mathcal{A}$
- The policy defines the action to be taken in any state s_t: $a_t = \pi(s_t)$
- The **value** of a policy π , $V^{\pi}(s_t)$ is the expected cumulative reward that will be received while the agent follows the policy, starting from state s_t .

Expected Reward

- In the **finite-horizon** or **episodic** model, the agent tries to maximize the expected reward for the next T steps:

$$V^{\pi}(s_t) = E[r_{t+1} + r_{t+2} + \cdots + r_{t+T}] = E\left[\sum_{i=1}^{T} r_{t+i}\right]$$

- In the **infinite-horizon model**, there is no sequence limit, but future rewards are discounted:

$$V^{\pi}(s_t) = E[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots] = E\left[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}\right]$$

where $0 \leq \gamma < 1$ is the **discount rate** to keep the return finite.

- γ is less than 1 because there generally is a time limit to the sequence of actions needed to solve the task. The agent may be a robot that runs on a battery. We prefer rewards sooner rather than later because we are not certain how long we will survive.

Optimal policy

For each policy, π there is an expected reward $V^{\pi}(s_t)$

We want to find the **optimal policy** π^* such that:

$$V^*(s_t) = \max_{\pi} V^{\pi}(s_t), \forall s_t$$

In some applications, for example, in control, instead of working with the values of states, $V(s_t)$ we prefer to work with the values of state action pairs, $Q(s_t, a_t)$

$V(s_t)$ denotes how good it is for the agent to be in state s_t

$Q(s_t, a_t)$ denotes how good it is to perform action a_t when in state s_t. This is called **Q-Learning**.

Bellman's optimality principle suggests that in finding the solution of an optimization problem, regardless of the initial state and initial decision, the consequent decisions must provide an optimal policy with respect to the state resulting from the initial decision.

$$
\begin{aligned}
V^*(s_t) &= \max_{a_t} Q^*(s_t, a_t) \\
&= \max_{a_t} E\left[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}\right] \\
&= \max_{a_t} E\left[r_{t+1} + \gamma \sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i+1}\right] \\
&= \max_{a_t} E\left[r_{t+1} + \gamma V^*(s_{t+1})\right]
\end{aligned}
$$

Bellman's Equation

$$
V^*(s_t) = \max_{a_t}\left(E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) V^*(s_{t+1})\right)
$$

Similarly:

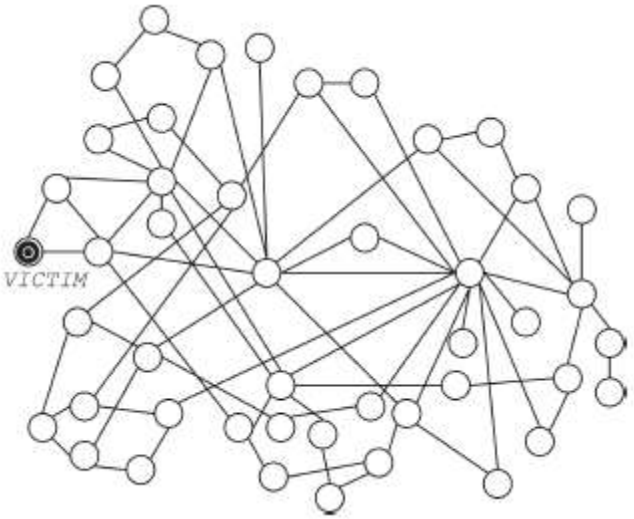
$$
Q^*(s_t, a_t) = E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})
$$

The optimal policy is chosen as:

$$
\pi^*(s_t) : \text{Choose } a_t^* \text{ where } Q^*(s_t, a_t^*) = \max_{a_t} Q^*(s_t, a_t)
$$

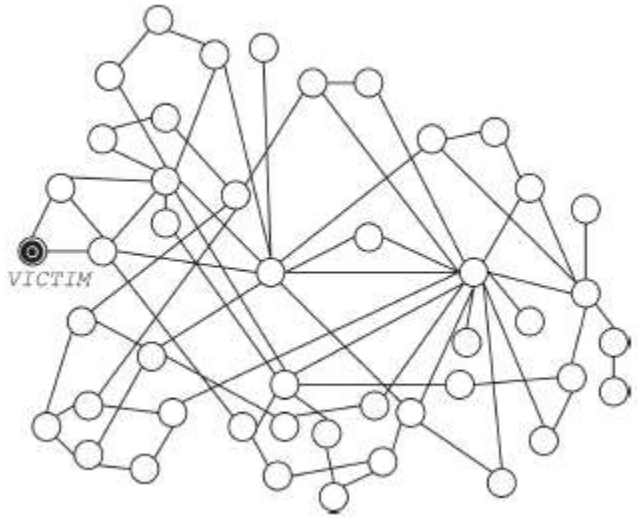
Example: Cognitive Packet Network (CPN)

ÖKE GÜLAY, LOUKAS GEORGIOS (2007). A denial of service detector based on maximum likelihood detection and the random neural network. COMPUTER JOURNAL, 50(6), 717-727., Doi: 10.1093/comjnl/bxm066



A networking which consists of 46 nodes connected with 100 MBits/s links
SwitchLAN backbone network topology:
It is a network that provides service in Switzerland to all universities, two federal institutes of technology and the major research institutes.

Smart Packets (SP) are sent to measure different metrics.



they will read the relevant measurement data from its CM. In the reinforcement learning algorithm, the observed outcome of a decision is used to "reward" or "punish" the routing algorithm with respect to that decision. The "Goal" is the metric that characterizes the success of the outcome, such as packet travel time or transit delay. As an example, the QoS *goal* (G) that smart packets pursue may be formulated by as minimizing *transit delay* (W), *loss probability* (L), *jitter*, or some weighted combination, for instance

$$G = a * W + b * L$$

Reward: $R = G^{-1}$

Decision Treshold: $T_l = a * T_{l-1} + (1 - a) * R_l$

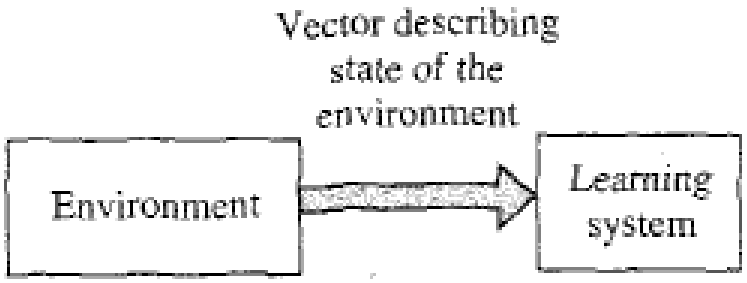
If $T_{l-1} \leq R_l$

$$w^+(i,j) = w^+(i,j) + (R_l - T_{l-1}),$$
$$w^-(i,k) = w^-(i,k) + (R_l - T_{l-1})/(n-1), \quad k \neq j$$

else

$$w^+(i,k) = w^+(i,k) + (T_{l-1} - R_l)/(n-1), \quad k \neq j,$$
$$w^-(i,j) = w^-(i,j) + (T_{l-1} - R_l).$$

Unsupervised Learning (Öğreticisiz Öğrenme)



- There is no external teacher or critic.
- The system learns the **statistical properties** of the input data.
- It develops the ability to form **internal representations of the input**.
- New classes are formed automatically.
- Statistically similar inputs should be classified in the same class.
- A task-independent measure of the quality of the representation of the system is selected and the free parameters of the system are optimized with respect to this measure.

*Reference. Haykin, S.,Neural Networks, 1994.

- Unsupervised Learning is suitable for clustering and classification problems.

- Some unsupervised learning algorithms are:

- K means clustering
- K nearest neighbour
- Kohonen's SOM
- Singular value decomposition
- Voronoi diagrams

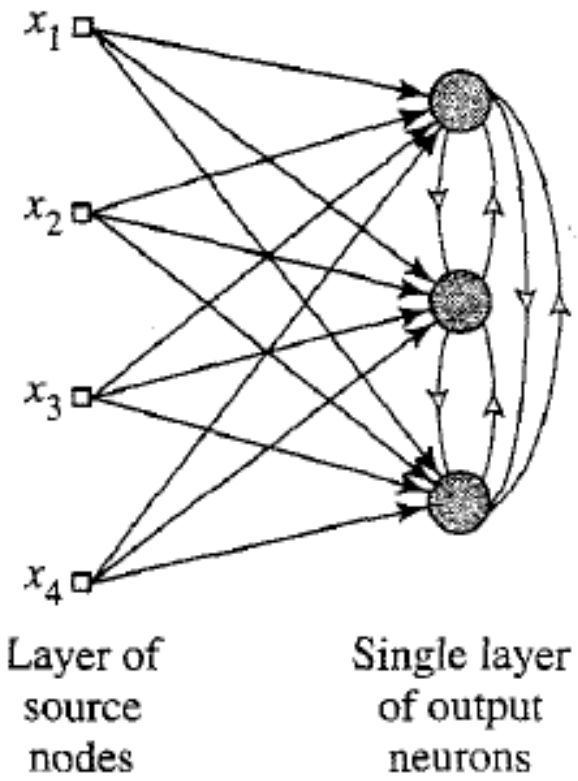
*Reference. Haykin, S.,Neural Networks, 1994.

SELF-ORGANIZING MAPS (SOM) (Kohonen) – (Haykin, Chapter 9)

- They are a special class of artificial neural networks.
- These networks are based **on competitive learning**; the output neurons of the network compete among themselves to be activated or fired, with the result that only one output neuron, or one neuron per group is on at any one time. An output neuron that wins the competition is called a **winner-takes-all neuron**, or simply **a winning-neuron**.

- A self-organizing map is therefore characterized by the formation of a topographic map of the input patterns, in which the spatial locations (i.e., coordinates) of the neurons in the lattice are indicative of intrinsic statistical features contained in the input patterns

*Reference. Haykin, S.,Neural Networks, 1994.



The brain is organized in many places in such a way that different sensory inputs are represented by topologically ordered computational maps.

TABLE 9.2 Animal Names and Their Attributes

Animal		Dove	Hen	Duck	Goose	Owl	Hawk	Eagle	Fox	Dog	Wolf	Cat	Tiger	Lion	Horse	Zebra	Cow
is	small	1	1	1	1	1	1	0	0	0	0	1	0	0	0	0	0
	medium	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0
	big	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
has	2 legs	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
	4 legs	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
	hair	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
	hooves	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
	mane	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0
	feathers	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
likes to	hunt	0	0	0	0	1	1	1	1	0	1	1	1	1	0	0	0
	run	0	0	0	0	0	0	0	0	1	1	0	1	1	1	1	0
	fly	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0
	swim	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0

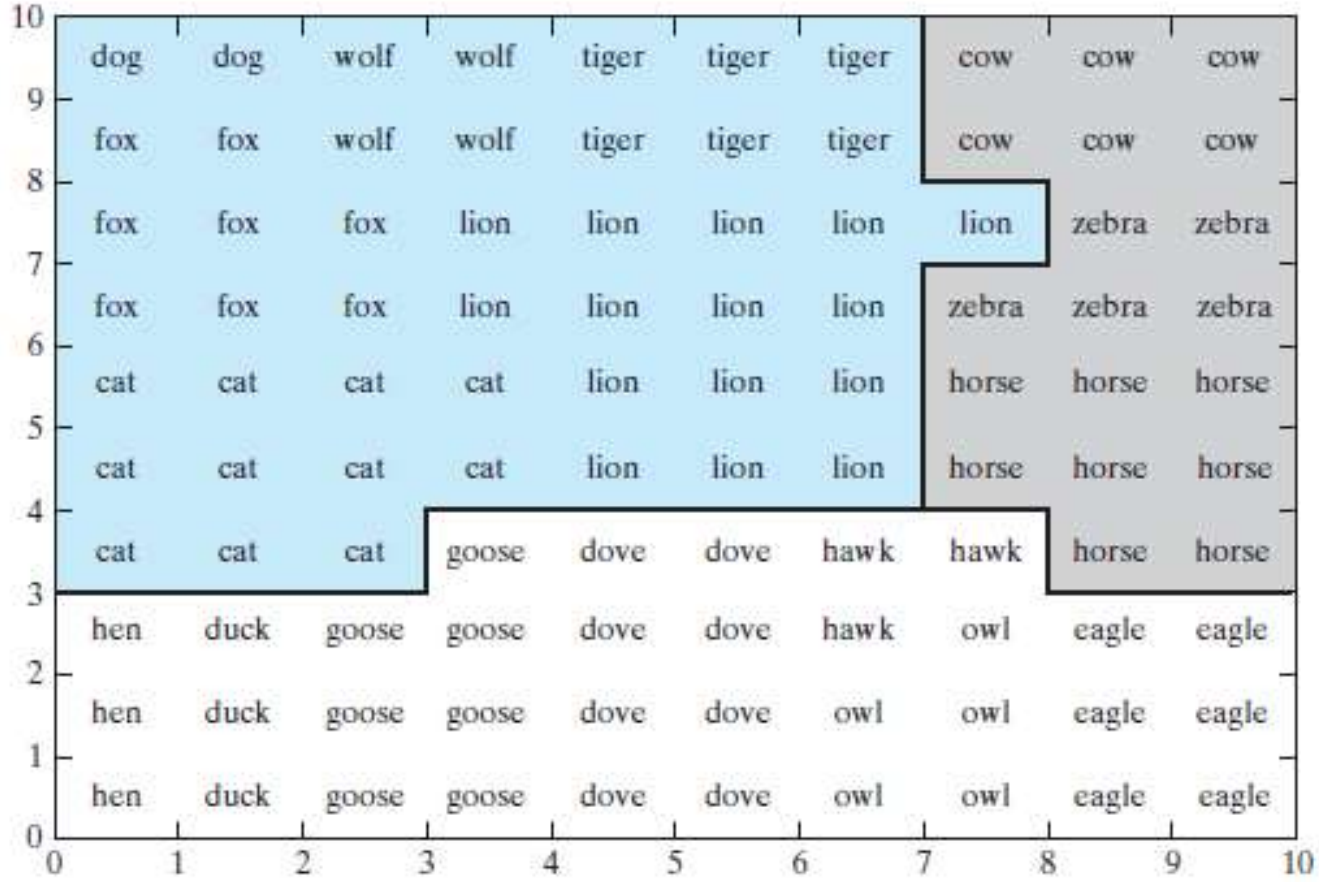


FIGURE 9.11 Semantic map obtained through the use of simulated electrode penetration mapping. The map is divided into three regions, representing birds (white), peaceful species (grey), and hunters (red).

Outline of the SOM Algorithm

1. Initialization.

Choose random values for the initial weight vectors $\mathbf{w}_j(0)$

2. Sampling.

Draw a sample x from the input space with a certain probability; the vector x represents the activation pattern that is applied to the lattice. The dimension of vector x is equal to m.

3. Similarity matching.

Find the best-matching (winning) neuron $i(\mathbf{x})$ at time-step n by using the minimum-distance criterion

$$i(\mathbf{x}) = \arg\min_j \|\mathbf{x}(n) - \mathbf{w}_j\|, \qquad j = 1, 2, ..., l$$

*Reference. Haykin, S.,Neural Networks, 1994.

4. Updating.

Adjust the synaptic-weight vectors of all excited neurons by using the update formula

$$\mathbf{w}_j(n+1) = \mathbf{w}_j(n) + \eta(n) h_{j,i(\mathbf{x})}(n)(\mathbf{x}(n) - \mathbf{w}_j(n))$$

$h_{j,i(\mathbf{x})}(n)$ is the topological neighborhood function.

For example:

$$h_{j,i(\mathbf{x})}(n) = \exp\left(-\frac{d_{j,i}^2}{2\sigma^2(n)}\right), \qquad n = 0, 1, 2, \ldots,$$

5. Continuation.

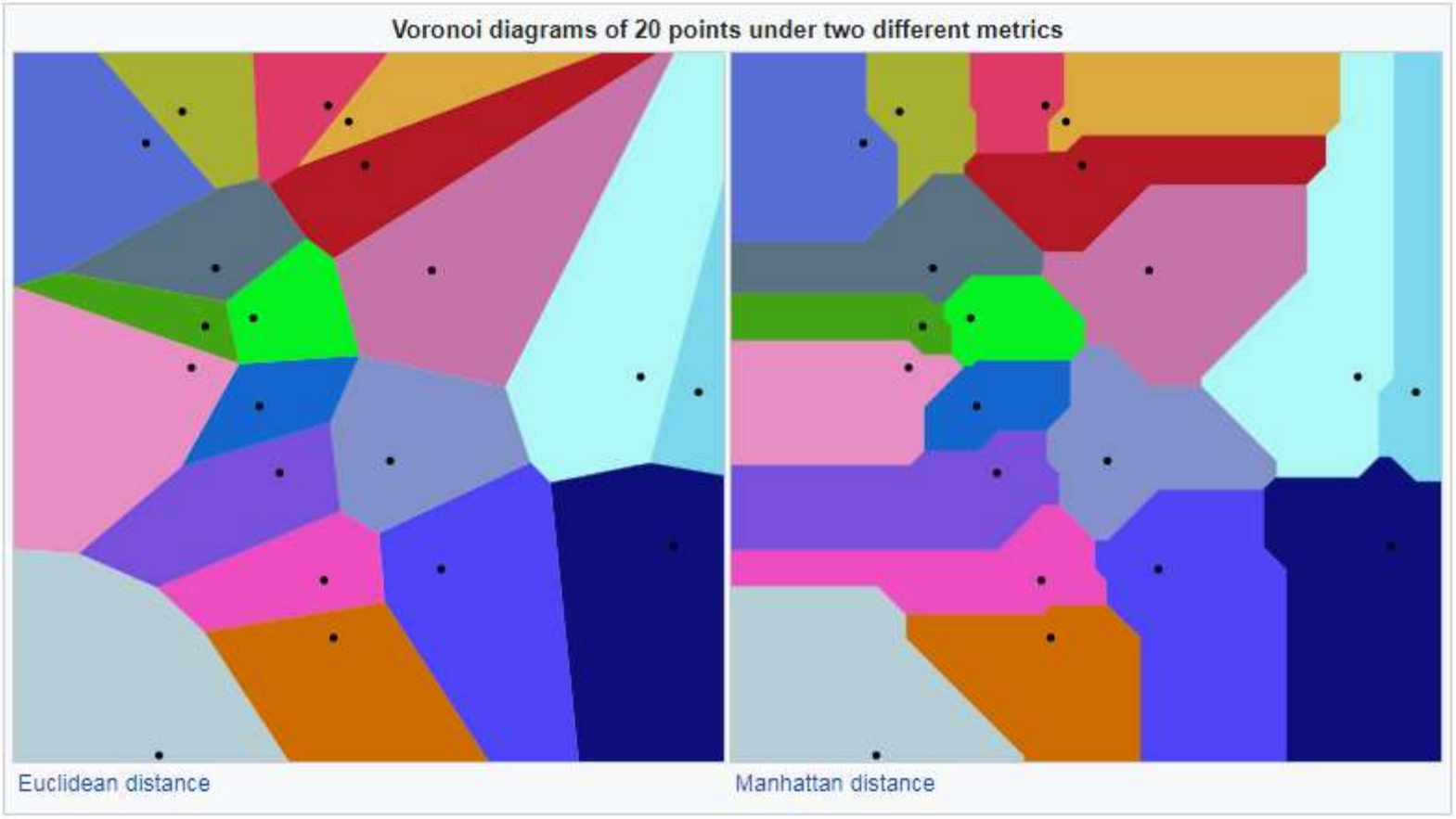
Continue with step 2 until no noticeable changes in the feature map are observed.

VORONOI DIAGRAMS

- A **Voronoi diagram** is a partition of a plane into regions close to each of a given set of objects. In the simplest case, these objects are just finitely many points in the plane (called seeds, sites, or generators). For each seed there is a corresponding region consisting of all points of the plane closer to that seed than to any other. These regions are called Voronoi cells.

- The Voronoi diagram is named after Georgy Voronoy, and is also called a **Voronoi tessellation**, a **Voronoi decomposition**, a **Voronoi partition**, or a **Dirichlet tessellation** (after Peter Gustav Lejeune Dirichlet). Voronoi cells are also known as **Thiessen polygons**. Voronoi diagrams have practical and theoretical applications in many fields, mainly in science and technology, but also in visual art.

(Reference: Wikipedia)



Euclidean distance: $\ell_2 = d\left[(a_1, a_2), (b_1, b_2)\right] = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2}$

Manhattan distance: $d\left[(a_1, a_2), (b_1, b_2)\right] = |a_1 - b_1| + |a_2 - b_2|.$

(Reference: Wikipedia)

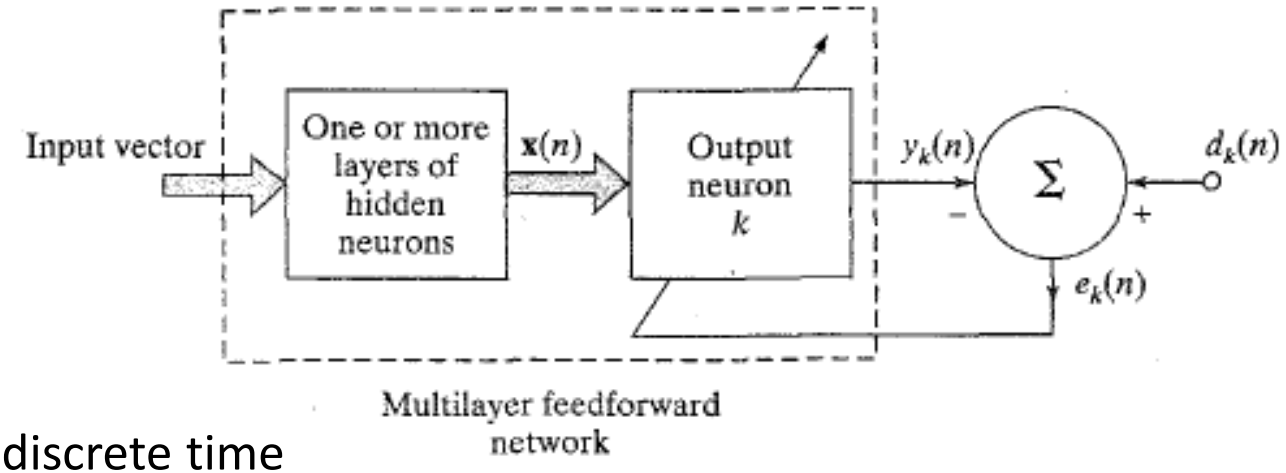
Learning Algorithms

A prescribed set of well-defined rules for the solution of a learning problem is called a learning algorithm.

1. **Error Correction Learning**
2. **Hebbian Learning**
3. **Competitive Learning**
4. **Boltzman Learning**
5. **Memory-Based Learning**

*Reference. Haykin, S.,Neural Networks, 1994.

Error Correction Learning



n: discrete time

$d_k(n)$ Desired response/target output

$y_k(n)$ Actual output signal

$e_k(n)$ Error signal

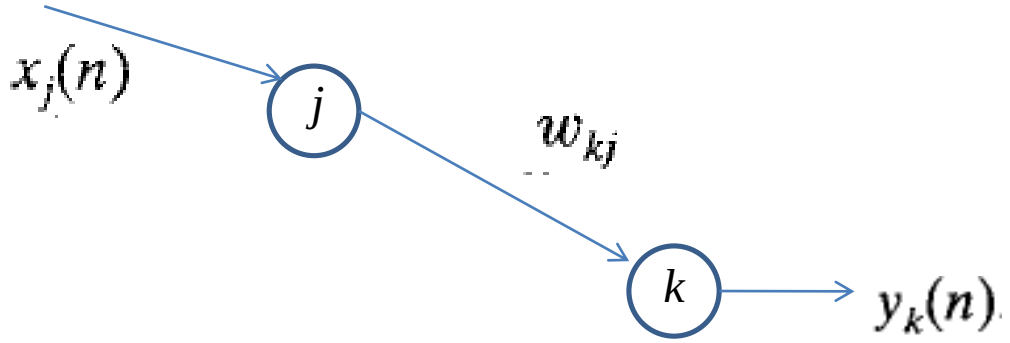
$$e_k(n) = d_k(n) - y_k(n)$$

$\mathcal{E}(n)$ Cost function / Index of performance
(Instantaneous value of the error energy)

$$\mathcal{E}(n) = \frac{1}{2} e_k^2(n)$$

*Reference. Haykin, S.,Neural Networks, 1994.

Widrow-Hoff Rule/ Delta Rule (1960)

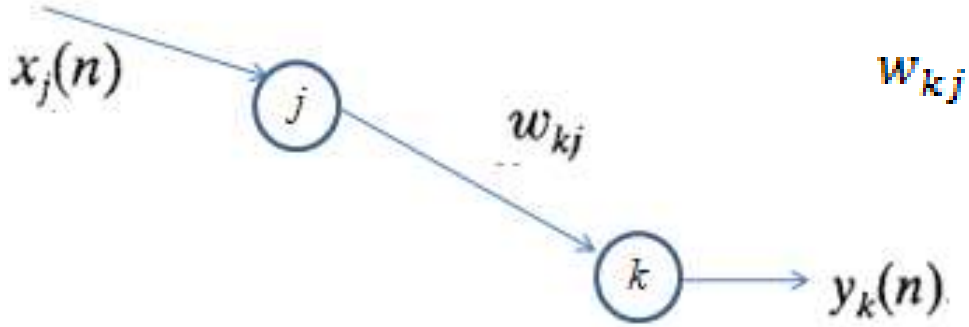


$$w_{kj}(n+1) = w_{kj}(n) + \Delta w_{kj}(n)$$

$$\Delta w_{kj}(n) = \eta e_k(n) x_j(n)$$

η Learning rate

The step-by-step adjustments to the synaptic weights of neuron k are continued until the system reaches a steady-state.
(the synaptic weights are stabilized)



$$w_{kj}(n+1) = w_{kj}(n) + \Delta w_{kj}(n)$$

$$\Delta w_{kj}(n) = \eta e_k(n) x_j(n)$$

$$\Delta w_{kj}(n) = \eta\big(d_k(n) - y_k(n)\big)x_j(n)$$

Case 1 If $y_k(n) > d_k(n)$ and $x_j(n) > 0$

$$\Delta w_{kj}(n) = \eta(-)(+) < 0$$

$$w_{kj}(n+1) < w_{kj}(n)$$

$$x_j(n+1)w_{kj}(n+1) < w_{kj}(n)x_j(n)$$

$$y_k(n+1) < y_k(n)$$

This is what we want ←

Case 2 If $y_k(n) < d_k(n)$ and $x_j(n) > 0$

$$\Delta w_{kj}(n) = \eta(+)(+) > 0$$

$$w_{kj}(n+1) > w_{kj}(n)$$

$$x_j(n+1)w_{kj}(n+1) > w_{kj}(n)x_j(n)$$

$$y_k(n+1) > y_k(n)$$

This is what we want ←

Verify Case 3 and Case 4 yourselves:

Case 3 If $y_k(n) > d_k(n)$ and $x_j(n) < 0$

Case 4 If $y_k(n) < d_k(n)$ and $x_j(n) < 0$

Hebbian Learning

Hebb was a neuroanatomist

Hebb's Postulate of Learning from *The Organization of Behaviour* (1949):

When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic changes take place in one or both cells such that A's efficiency as one of the cells firing B, is increased.

(There is physiological evidence of Hebbian learning in hippocampus part of the brain)

This is a neurobiologıcal statement. It can be rephrased as follows:

1. If two neurons on either side of a synapse (connection) are activated simultaneously (synchronously), the strength of that synapse is selectively increased.
2. If two neurons on either side of a synapse are activated asynchronously, that synapse is selectively weakened or eliminated.

*Reference. Haykin, S.,Neural Networks, 1994.

Such a synapse is called a Hebbian synapse.

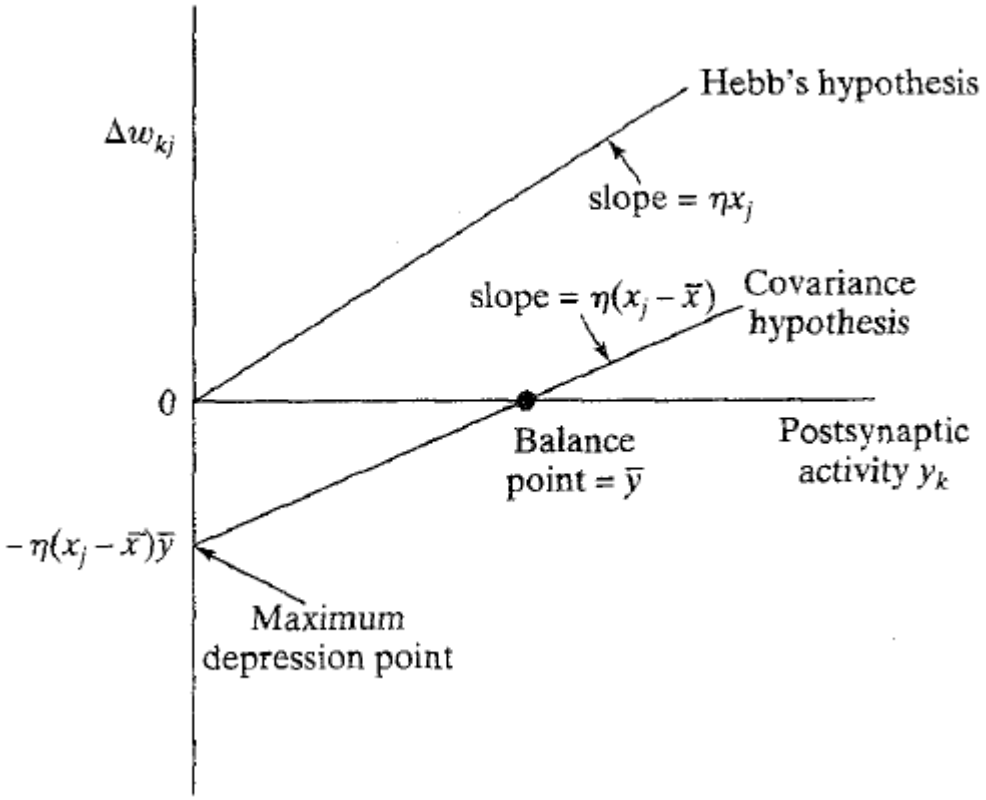
A Hebbian synapse is **time-dependent, highly local**, and **strongly interactive** mechanism to increase synaptic efficiency as a function of the **correlation between the presynaptic and postsynaptic activities**.

Learning Rule: $\Delta w_{kj}(n) = F(y_k(n), x_j(n))$

Hebb's Hypothesis: $\Delta w_{kj}(n) = \eta y_k(n) x_j(n)$

This is also called the **activity-product rule**. The change on the weight is proportional to the product of the input(presynaptic signal) x_j and output (postsynaptic signal) y_k

Covariance Hypothesis (Sejnowski, 1977) :



$$\Delta w_{kj} = \eta(x_j - \bar{x})(y_k - \bar{y})$$

- Convergence to a nontrivial state, which is reached when

  $x_k = \bar{x}$ or $y_j = \bar{y}$

- Synaptic weight w_{kj} is enhenced if there are sufficient levels of presynaptic and postsynaptic activities, that is the conditions $x_j > \bar{x}$ and $y_k > \bar{y}$ are both satisfied.
- Synaptic weight w_{kj} is depressed if there is either

  $x_j > \bar{x}$ and $y_k < \bar{y}$

  or $y_k > \bar{y}$ and $x_j < \bar{x}$

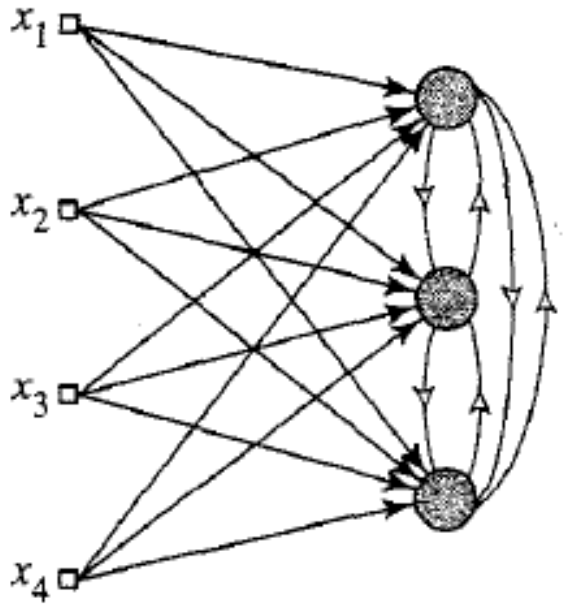
Competitive Learning

- The output neurons of a neural network **compete** among themselves to become **active (fired**).
- Only a single output neuron is active at any one time.
- It is suitable to discover statistical features to classify a set of input patterns.

<u>Basic elements of competitive learning:</u>

- A set of neurons that are **same** except for random synaptic weight, and which therefore respond differently to a given set of inpus.
- A limit imposed on the **strength** of each neuron.
- A mechanism that permits the neurons to compete for the right to respond to a given set of inputs, such that only one neuron is active at a time. That neuron is called **a winner-takes-all neuron.**

*Reference. Haykin, S.,Neural Networks, 1994.



The network may include **feedback** connections to perform **lateral inhibition.**

$$y_k = \begin{cases} 1 & \text{if } v_k > v_j \text{ for all } j, j \neq k \\ 0 & \text{otherwise} \end{cases}$$

$$\sum_j w_{kj} = 1 \qquad \text{for all } k$$

$$\Delta w_{kj} = \begin{cases} \eta(x_j - w_{kj}) & \text{if neuron } k \text{ wins the competition} \\ 0 & \text{if neuron } k \text{ loses the competition} \end{cases}$$

KON 426E
INTELLIGENT CONTROL SYSTEMS

LECTURE 3

07/03/2022

Why optimization?

- Every learning (intelligent agent) has parameters to be adapted.
- For an intelligent agent to learn to do a specific task, we have to adjust its parameters.(e.g weights in an NN)
- For this, we define a cost function which measures the error in performing the task and minimize it.
- So, every learning task involves the solution of an optimization problem.

Optimization

Derivative- based optimization methods

- Steepest descent
- Newton's method
- LMS

……

Derivative- free optimization methods

- Genetic Algorithm
- Particle Swarm Otimization
- Simulated Annealing

…….

Gradient-Based optimization Techniques

- Let us define a real-valued cost function/objective function E on an n-dimensional input space $\boldsymbol{\theta} = [\theta_1, \theta_2, \ldots, \theta_n]^T$

  (Here, $\boldsymbol{\theta}$ denotes the parameter vector to be adjusted e.g. the weights in a NN).
- Find a (possibly local) minimum point $\boldsymbol{\theta} = \boldsymbol{\theta}^*$ that minimizes $E(\boldsymbol{\theta})$

- Mathematically, to find the minimum of a function , we differentiate the function, equate it to zero and solve for the parameter.
- However, here we will be dealing with nonlinear cost functions $E(\boldsymbol{\theta})$. Due to the complexity of $E(\boldsymbol{\theta})$ we often resort to itearative methods to explore the input space.

*Reference.:Jang, Sun, Mizutani, Neuro-Fuzzy and Soft Computing**,** PTR Prentice-Hall, 1997.

- By using an iterative method, we try to find a minimizing parameter vector $\boldsymbol{\theta}^*$ such that:

$$E(\boldsymbol{\theta}^*) < E(\boldsymbol{\theta})$$

- In the iterative method, we start from an initial point. In each iteration we move in the direction of a **direction vector $\mathbf{d}$,** with a step size η

$$\boldsymbol{\theta}_{\text{next}} = \boldsymbol{\theta}_{\text{now}} + \eta \mathbf{d}$$

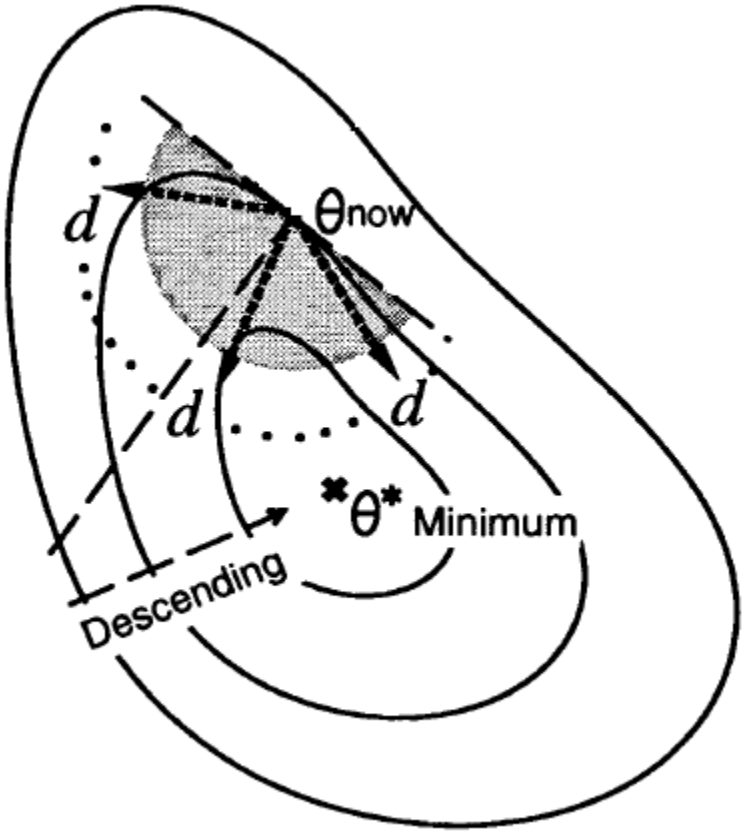
Alternatively:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \eta_k \mathbf{d}_k \quad (k = 1, 2, 3, .$$

$\mathbf{d}$: direction vector

η: learning rate (step size)

k: iteration number



In iterative descent methods, we:

- First determine the direction vector $\mathbf{d}$.
- Then calculate the step size η

In each iteration, the following inequality should be satisfied:

$$E(\boldsymbol{\theta}_{\text{next}}) = E(\boldsymbol{\theta}_{\text{now}} + \eta \mathbf{d}) \; < \; E(\boldsymbol{\theta}_{\text{now}})$$

The main differences in various descent algorithms lie in the calculation of the $\mathbf{d}$, direction vector.

- After calculating $\mathbf{d}$, the step size (learning rate) η is determined:

- Optimal value of the step size can be obtained by **line minimization:**

$$\eta^* = \arg \min_{\eta>0} \phi(\eta)$$

where $$\phi(\eta) = E(\boldsymbol{\theta}_{\text{now}} + \eta \mathbf{d})$$

For the search of the optimal η^* we use, **one-dimensional search methods** such as:

- Newton's method
- Secant method
- Golden search method

Sec. 6.5 of Jang, Sun, Mizutani, Neuro-Fuzzy and Soft Computing, PTR Prentice-Hall, 1997.

Gradient-Based Optimization Methods

- The **gradient** of a differentiable function $E : R^n \rightarrow R$ is the vector of first derivatives of $\boldsymbol{E}$:

$$\mathbf{g}(\boldsymbol{\theta}) \; (= \nabla E(\boldsymbol{\theta})) \stackrel{\text{def}}{=} \left[\frac{\partial E(\boldsymbol{\theta})}{\partial \theta_1}, \frac{\partial E(\boldsymbol{\theta})}{\partial \theta_2}, \cdots, \frac{\partial E(\boldsymbol{\theta})}{\partial \theta_n} \right]^T$$

- When the direction vector $\mathbf{d}$ is determined based on the gradient information, such methods are called **gradient-based descent methods.**

Remember that:

$$E(\boldsymbol{\theta}_{\text{next}}) = E(\boldsymbol{\theta}_{\text{now}} + \eta \mathbf{d}) \; < \; E(\boldsymbol{\theta}_{\text{now}})$$

Using Taylor expansion:

$$E(\boldsymbol{\theta}_{\text{now}} + \eta \mathbf{d}) = E(\boldsymbol{\theta}_{\text{now}}) + \eta \mathbf{g}^T \mathbf{d} + O(\eta^2)$$

This term goes to 0 since η is small.

For the first inequality to hold, it must be true that: $\mathbf{g}^T \mathbf{d} < 0$

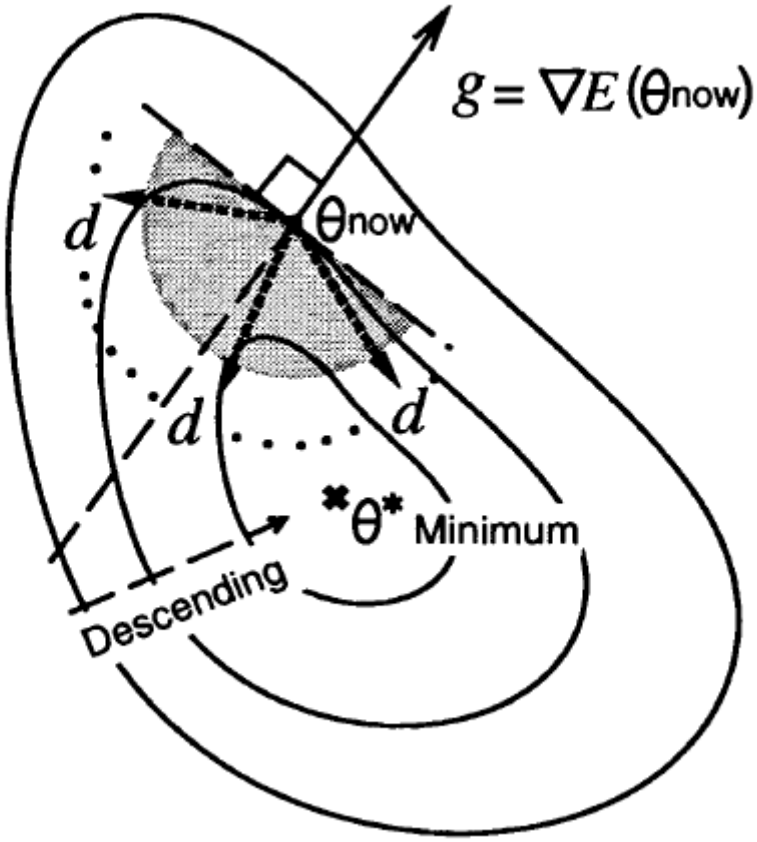
$$\mathbf{g}^T \mathbf{d} = \|\mathbf{g}^T\| \, \|\mathbf{d}\| \cos(\xi(\boldsymbol{\theta}_{\text{now}})) < 0$$

positive

$\xi(\boldsymbol{\theta}_{\text{now}})$ is the angle between $\mathbf{g}$ and $\mathbf{d}$.

For this inequality to hold: $\xi(\boldsymbol{\theta}_{\text{now}})$ must be selected **between 90 and 270 degrees**.

Feasible Descent Directions



- A class of gradient-based descent methods calculates descent directions by deflecting the gradients through multiplication by a positive definite matrix $\mathbf{G}$.

$$\mathbf{d} = -\mathbf{G}\mathbf{g}$$

$$\mathbf{g}^T\mathbf{d} = -\mathbf{g}^T\mathbf{G}\mathbf{g} < 0 \longrightarrow$$ Negative definite

$$\boldsymbol{\theta}_{\text{next}} = \boldsymbol{\theta}_{\text{now}} - \eta\mathbf{G}\mathbf{g}$$

Stopping Criteria

Ideally we stop the iterations when we find a $\boldsymbol{\theta}_{\text{next}}$ such that:

$$\mathbf{g}(\boldsymbol{\theta}_{\text{next}}) = \left.\frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right|_{\boldsymbol{\theta}=\boldsymbol{\theta}_{\text{next}}} = 0$$

In practice, it is difficult to reach this condition.

Therefore the iterations are typically repeated until one of the following stopping criteria is satisfied:

- The objective function E value is sufficiently small.
- The length of the gradient $\mathbf{g}$ is smaller than a specified value.
- The specified computing time is reached.

The Method of Steepest Descent

- Remember from calculus that the **gradient** of a multivariable function computes the direction in which the function **increases the fastest.**
- In solving a minimization problem we must find the direction in which the function decreases the fastest. (E will be decreased locally by the most amount.)
- So, we have to calculate the gradient and move in the **reverse of the gradient direction.**
- Choose: $\mathbf{d} = -\mathbf{g}$

  (Equivalently:) $\mathbf{G} = \eta \mathbf{I}$

  (Equivalently:) $\cos\xi = -1$

  $\xi = 180^o$

$$\boldsymbol{\theta}_{\text{next}} = \boldsymbol{\theta}_{\text{now}} - \eta \mathbf{g}$$

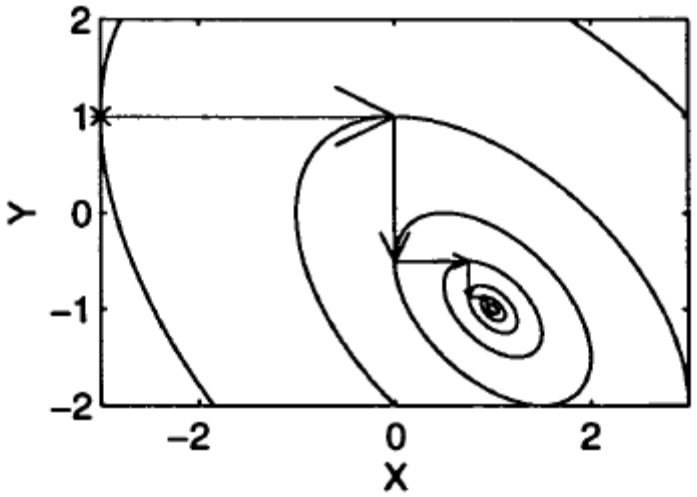
- The main advantage of this method is that it is very simple to apply, computational load is low.
- However it has a disadvantage, it is slow.

$$\frac{dE(\theta_{next})}{d\eta} = \frac{\partial E(\theta_{next})}{\partial \theta_{next}} \frac{\partial \theta_{next}}{d\eta}$$

$$\theta_{next} = \theta_{now} - \eta \mathbf{g_{now}}$$

Line minimization:

$$\frac{dE(\theta_{next})}{d\eta} = -\mathbf{g_{next}^T} \cdot \mathbf{g_{now}} = 0$$

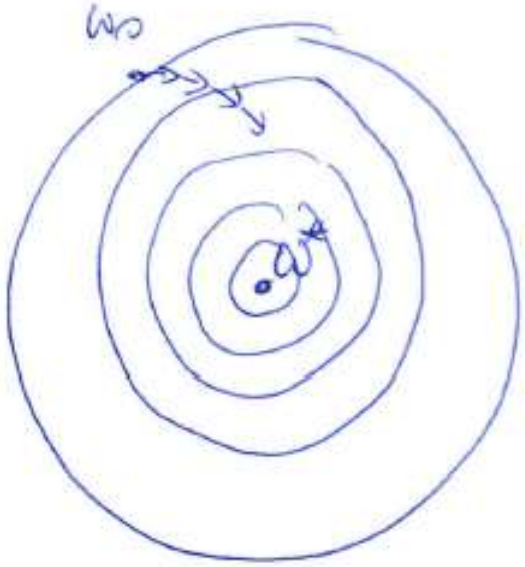


So, consecutive gradients (and search directions) are perpendicular to each other, and this slows down the seaarch.

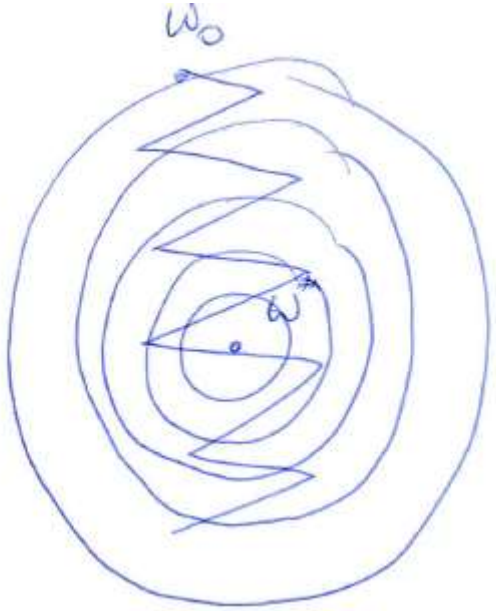
An Important Comment on the Learning Rate

Selection of the learning rate is very important.

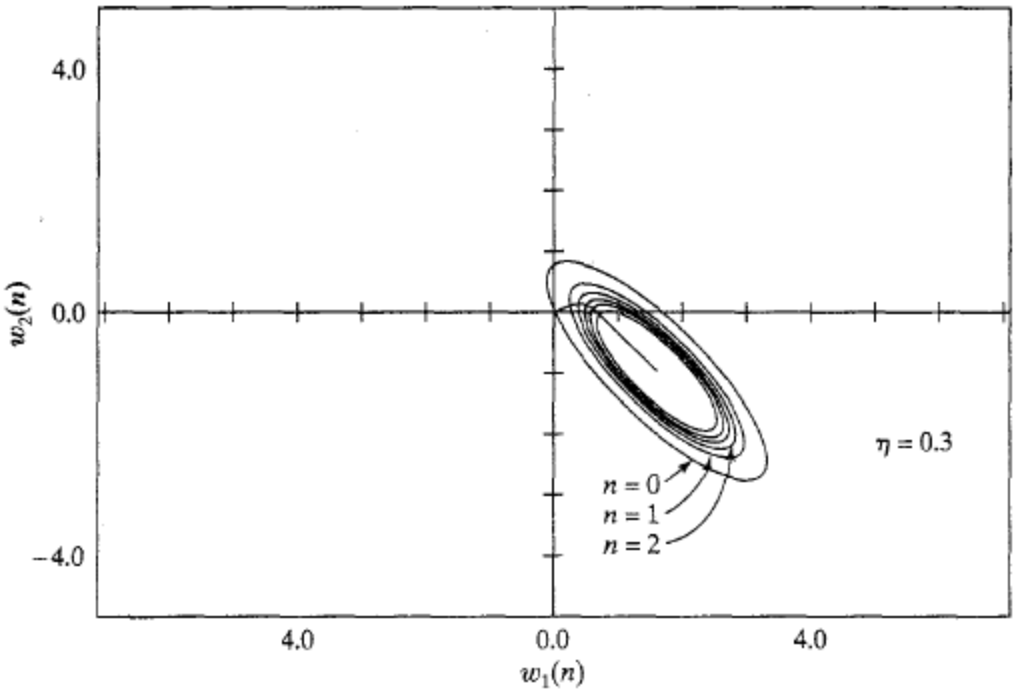
If η is too small:



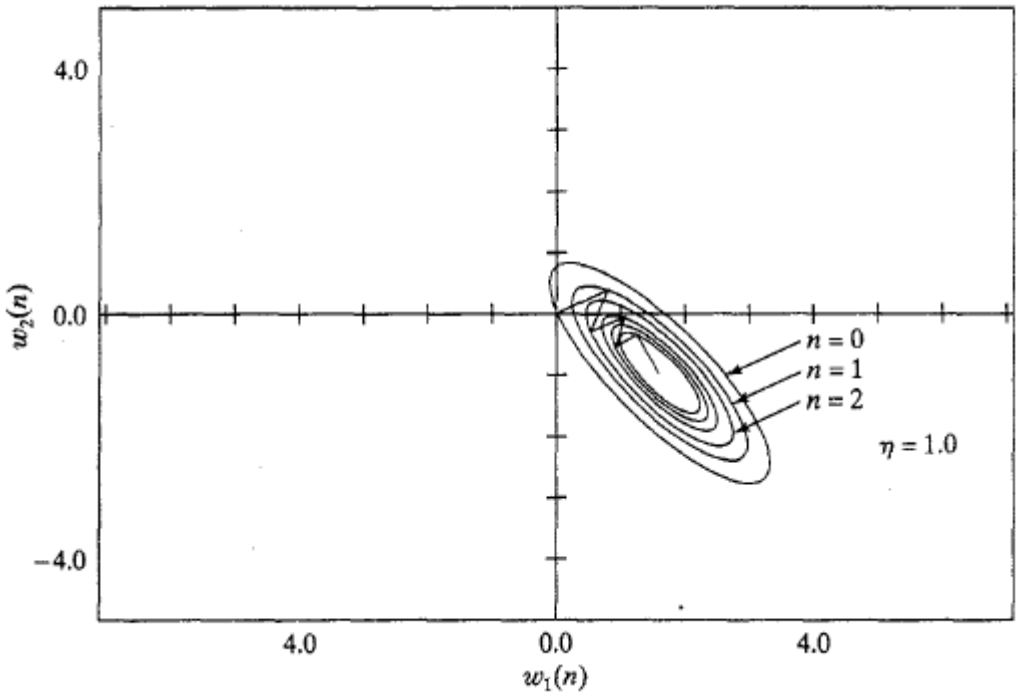
If η is too large:



Trajectory of the steepest descent method
in a two dimensional parameter space for small η



Trajectory of the steepest descent method
in a two dimensional parameter space for large η



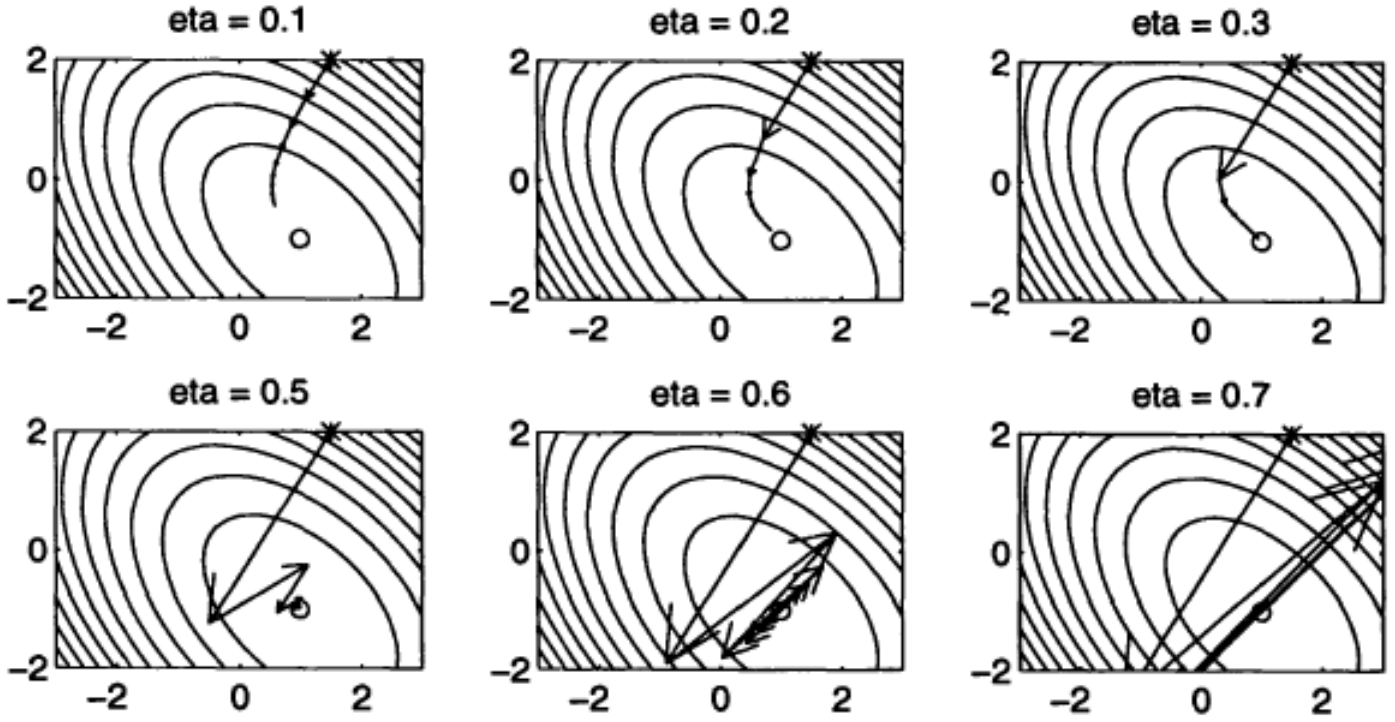


Figure 6.15. *The effects of step sizes η by the steepest descent method defined in Equation(6.60). The search is inefficient when $\eta \leq 0.1$ and unstable when $\eta \geq 0.7$.* (MATLAB file: `gdss1.m`)

- Initially, at the first step the error is large, you will want to go with large steps, so you shoud choose a large step size (learning rate) η

- As the number of iterations increase, you get closer and closer to the minimum point, $\theta = \theta^*$, you want to make a finer search or you may miss the minimum point.

- Therefore it is advisable to have an adaptive step size (learning rate) η where you start with a large step size and then make it smaller and smaller.

- Of course it is possible if you have a constant learning rate, but you must be careful in determining its value.

Newton's Methods

Classical Newton's Method (Newton-Raphson method)

The second derivative of E is used to find to direction vector **d.**

Take the starting point $\boldsymbol{\theta}_{\text{now}}$ sufficiently close to a local minimum point and write down the Taylor expansion:

$$E(\boldsymbol{\theta}) \approx E(\boldsymbol{\theta}_{\text{now}}) + \mathbf{g}^T(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{now}}) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{now}})^T \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{now}})$$

Higher order terms are omitted since $\|\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{now}}\|$ is small.

$\mathbf{H}$ is the Hessian, the matrix of second partial derivatives:

$$\mathbf{H}_f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

To find the optimizing $\boldsymbol{\theta}^*$ that minimizes $E(\boldsymbol{\theta})$, we differentiate $E(\boldsymbol{\theta})$ and equate it to zero.

$$E(\boldsymbol{\theta}) \approx E(\boldsymbol{\theta}_{\text{now}}) + \mathbf{g}^T(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{now}}) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{now}})^T \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{now}})$$

$$0 = \mathbf{g} + \mathbf{H}(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}_{\text{now}})$$

$$\boxed{\hat{\boldsymbol{\theta}} = \boldsymbol{\theta}_{\text{now}} - \mathbf{H}^{-1}\mathbf{g}} \longrightarrow$$

Newton's Method
Newton_Raphson method

$-\mathbf{H}^{-1}\mathbf{g}$: Newton's step

Its direction is called Newton's direction

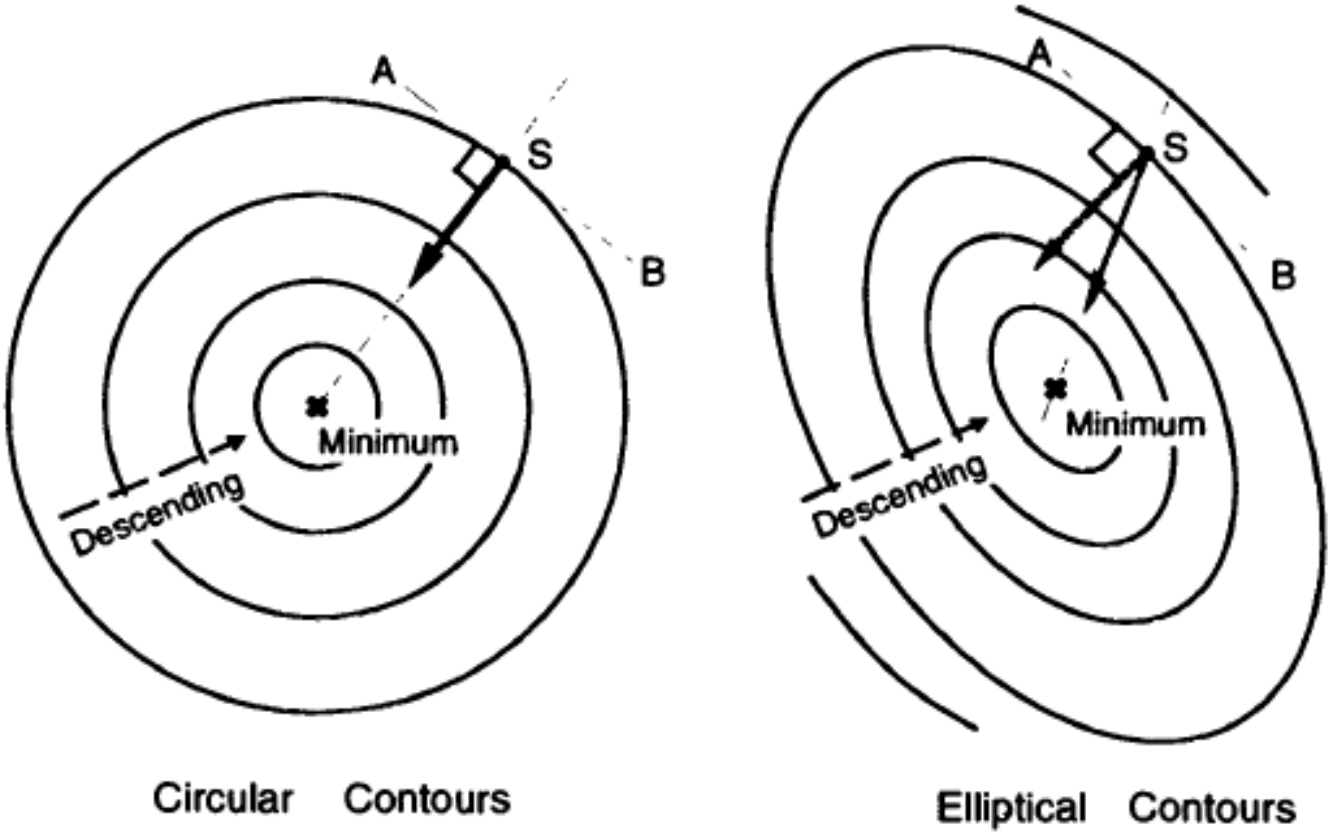
This is equivalent to the general gradient-descent based method with

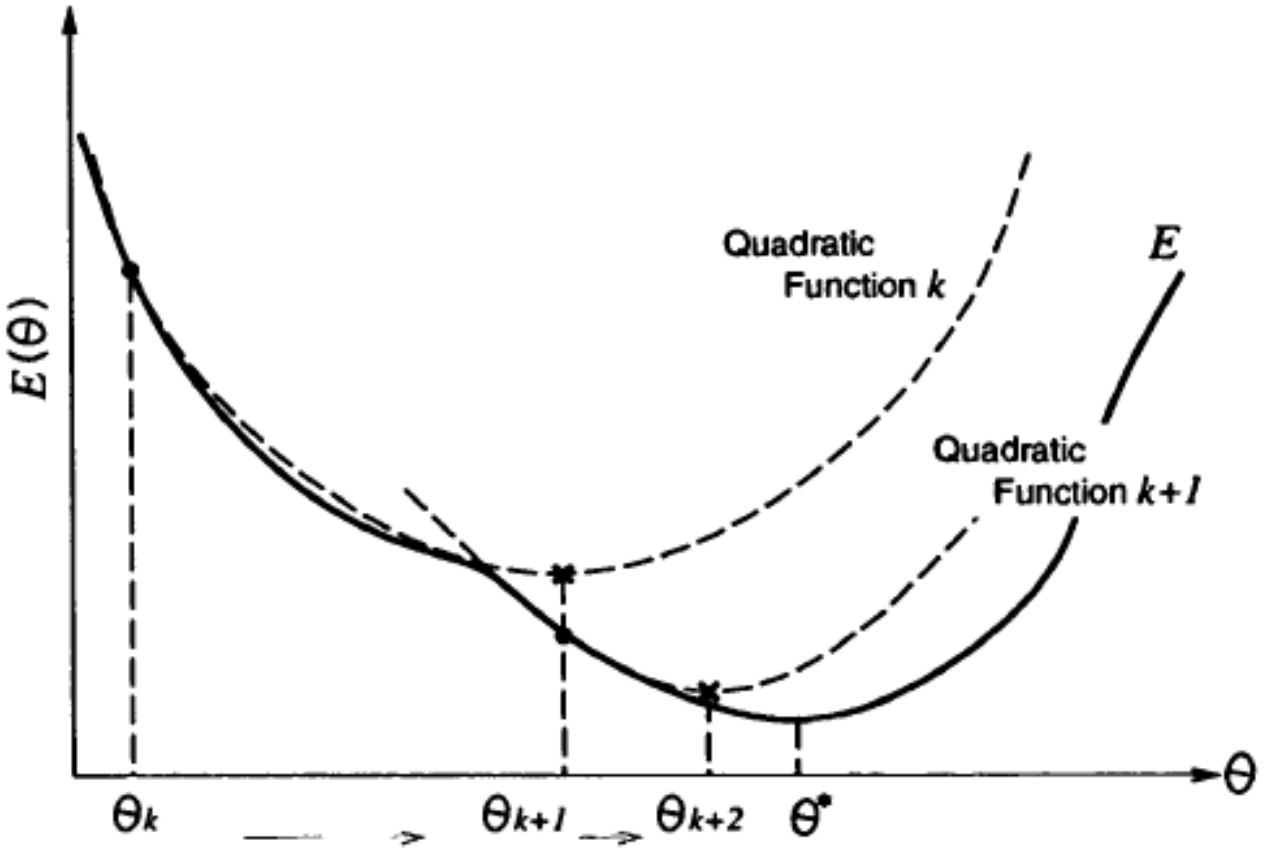
$\mathbf{G} = -\mathbf{H}^{-1}$ and $\eta = 1$

$$\hat{\boldsymbol{\theta}} = \boldsymbol{\theta}_{\text{now}} - \mathbf{H}^{-1}\mathbf{g} \quad \Longrightarrow \quad \boldsymbol{\theta}_{\text{next}} = \boldsymbol{\theta}_{\text{now}} - \eta\mathbf{G}\mathbf{g}$$

If $E(\theta)$ is quadratic, Newton's method finds the optimal value is a single Newton step.

If $E(\theta)$ is not quadratic, then the minimum may not be reached in a single step , Newton's method should be applied iteratively.





$$\boldsymbol{\theta}_{\text{next}} = \boldsymbol{\theta}_{\text{now}} - \mathbf{H}^{-1}\mathbf{g}$$

Major Problems of Newton's Method

- H may not be positive definite
- H may not be invertible.
- Calculating $\mathbf{H}^{-1}$ may be computationally intensive.
- Calculating $\mathbf{H}^{-1}$ may introduce numerical problems due to round-off errors.
- If θ_{now} is not close to θ^* then the Taylor series approximation does not hold.

In order to solve these problems some modifications have been introduced to the Newton's method.

Adaptive Step Length

$$\boldsymbol{\theta}_{\text{next}} = \boldsymbol{\theta}_{\text{now}} - \mathbf{H}^{-1}\mathbf{g}$$

Direct Newton step with $\eta = 1$ may be too large.

Introduce a positive step length:

$$\boldsymbol{\theta}_{\text{next}} = \boldsymbol{\theta}_{\text{now}} - \eta\mathbf{H}^{-1}\mathbf{g}$$

η is selected to minimize E.

Step-halving Procedure:

$$\eta_{k+1} = \frac{1}{2}\eta_k, \quad k = 0, 1, 2, 3, \ldots$$

Other heuristic methods are possible where η is selected to satisfy $E(\boldsymbol{\theta}_{\text{next}}) < E(\boldsymbol{\theta}_{\text{now}})$

Levenberg-Marquardt Modification

If the $\mathbf{H}$ is not positive definite, the Newton direction may point towards a local maximum or saddle point. $\mathbf{H}$ can be made positive definite by adding a positive definite matrix $\mathbf{P}$ to it.

$$\mathbf{P} = \lambda \mathbf{I}$$

$$\boldsymbol{\theta}_{\text{next}} = \boldsymbol{\theta}_{\text{now}} - (\mathbf{H} + \lambda \mathbf{I})^{-1}\mathbf{g}$$

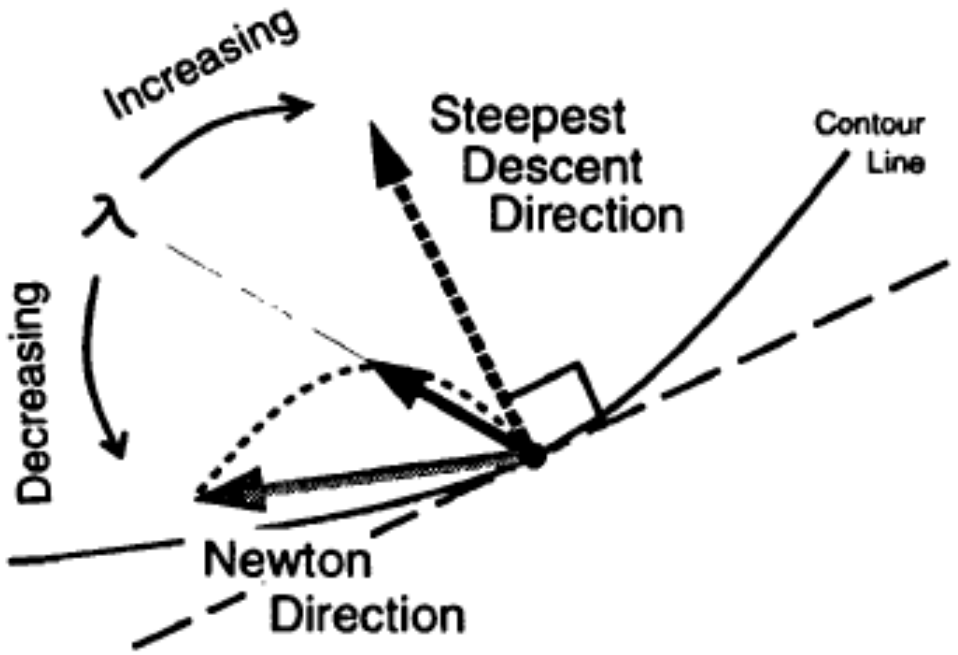
* **MATLAB Neural Network Toolbox** uses Levenberg –Marquardt method as default.

We can combine adaptive step length and LM:

$$\boldsymbol{\theta}_{\text{next}} = \boldsymbol{\theta}_{\text{now}} - \eta(\mathbf{H} + \lambda \mathbf{I})^{-1}\mathbf{g}$$

As $\lambda \to 0$ LM becomes similar to Newton's method.

As $\lambda \to \infty$ LM becomes similar to Steepest Descent Method.



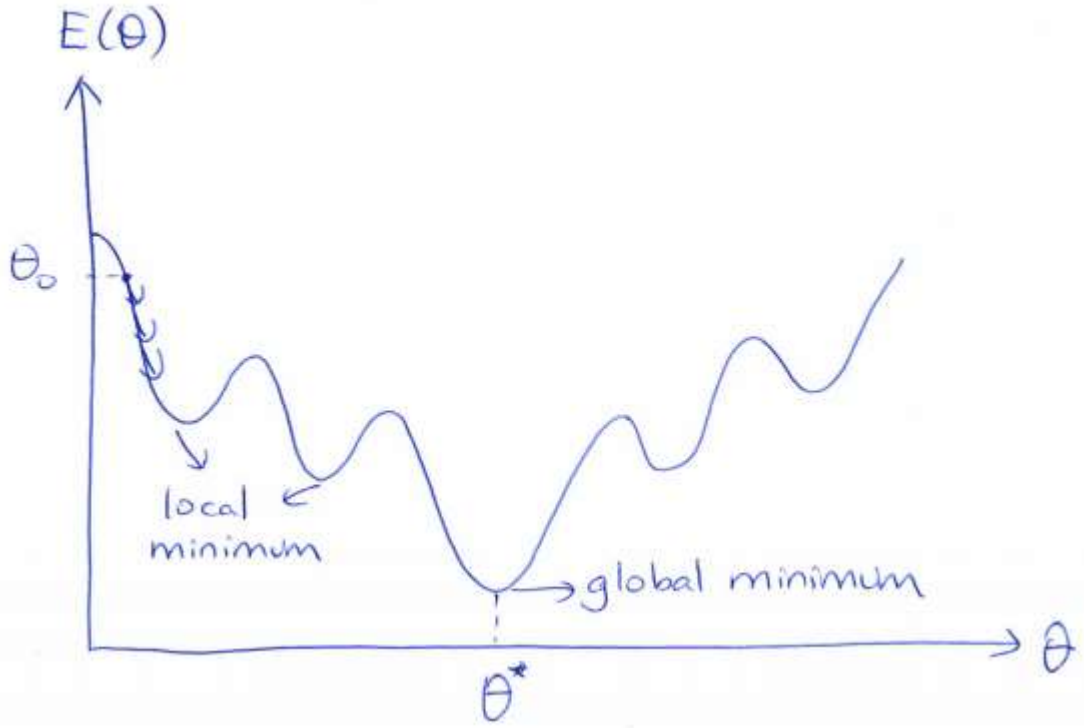
Modified Versions of Newton's Algorithm

- Quasi-Newton Methods
- Conjugate-Gradients Methods

Study **Chapter 6** of:

*Jang, J.-S. R., C.-T. Sun, E. Mizutani, Neuro-Fuzzy and Soft Computing**,** PTR Prentice-Hall, 1997.

An Important Property of Derivative-Based Optimization Methods



Derivative-based optimization methods assure you only the **local minimum**, they do not gurantee the **global minimum**.

The Method of Steepest Descent (Gradient Descent Method)

Step 1 Initialize

Given $\vec{\mathbf{x}}_0$, set $k = 0$

Set ε to a small value.

Step 2 Calculate $\vec{\mathbf{g}}^k = \nabla \vec{\mathbf{E}}^k$. If $\left\|\vec{\mathbf{g}}^k\right\| \leq \varepsilon$, stop.

Step 3 Set $\overrightarrow{\mathbf{d}^k} = -\overrightarrow{\mathbf{g}^k}$

Step 4 Solve $\min_{\eta} \mathbf{E}\left(\mathbf{x}^k + \eta^k \mathbf{d}^k\right)$ for η^k $\Longrightarrow$ Line minimization

Step 5 Set $\vec{\mathbf{x}}^{k+1} = \vec{\mathbf{x}}^k + \eta^k \overrightarrow{\mathbf{d}^k}$

Set $k = k + 1$

Go to Step 2

Example Consider the function $E = 5x_1^2 + x_2^2 + 4x_1x_2 - 14x_1 - 6x_2 + 20$

Start from $[x_1^0 \quad x_2^0]^T = [0 \quad 10]^T$. Apply steepest descent algorithm for one iteration.

Step 1

$[x_1^0 \quad x_2^0] = [0 \quad 10]^T$

Set $\varepsilon = 10^{-6}$

$k = 0$

Step 2

$$\vec{g} = \nabla \vec{E} = \begin{bmatrix} 10x_1 + 4x_2 - 14 \\ 2x_2 + 4x_1 - 6 \end{bmatrix}$$

At $[0 \ 10]^T$

$$\vec{g}^0 = \begin{bmatrix} 26 \\ 14 \end{bmatrix}$$

$$\|\vec{g}^0\| = 29.53 > \varepsilon$$

Step 3

Set $\vec{d}^0 = -\vec{g}^0 = \begin{bmatrix} -26 \\ -14 \end{bmatrix}$

Step 4

$$\vec{x}^{1} = \vec{x}^{0} + \eta^{0}\vec{d}^{0}$$

$$\vec{x}^{1} = \begin{bmatrix} 0 \\ 10 \end{bmatrix} + \eta^{0}\begin{bmatrix} -26 \\ -14 \end{bmatrix} = \begin{bmatrix} -26\eta^{0} \\ 10 - 14\eta^{0} \end{bmatrix}$$

$$E(\eta^{0}) = E(\vec{x}^{0} + \eta^{0}\vec{d}^{0}) =$$

$$= 5(-26\eta^{0})^{2} + (10 - 14\eta^{0})^{2} + 4(-26\eta^{0})(10 - 14\eta^{0})$$

$$-14(-26\eta^{0}) - 6(10 - 14\eta^{0}) + 20$$

$$E(\eta^{0}) = 5032(\eta^{0})^{2} - 872\,\eta^{0} + 60$$

$$\frac{dE(\eta^{0})}{d\eta^{0}} = 10064\,\eta^{0} - 872 = 0 \quad \Longrightarrow \quad \eta^{0} = 0.0866$$

$$\frac{d^2E(\eta^0)}{d\eta^{02}} = 10064 > 0$$

So, it is a minimum point.

Step 5

$$\vec{x}^1 = \begin{bmatrix} 0 \\ 10 \end{bmatrix} + 0.0866 \begin{bmatrix} -26 \\ -14 \end{bmatrix} = \begin{bmatrix} -2.253 \\ 8.788 \end{bmatrix}$$

Newton's Method

Step 1 Initialize

Given $\vec{\mathbf{x}}_0$, set $k = 0$

Set ε to a small value.

Step 2 Calculate $\vec{\mathbf{g}}^k = \nabla \vec{\mathbf{E}}^k$. If $\left\|\vec{\mathbf{g}}^k\right\| \leq \varepsilon$, stop.

Step 3 Calculate $\mathbf{H}, \mathbf{H}^{-1}$

Step 4 Set $\vec{\mathbf{d}}^k = -\mathbf{H}\vec{\mathbf{g}}^k$

Step 5 Solve $\min_{\eta} \mathbf{E}\left(\mathbf{x}^k + \eta^k \mathbf{d}^k\right)$ for η^k ⟹ Line minimization

Step 6 Set $\vec{\mathbf{x}}^{k+1} = \vec{\mathbf{x}}^k + \eta^k \vec{\mathbf{d}}^k$

Set $k = k + 1$

Go to Step 2

Example Consider the function $E = (x_1 + 2)^2 + x_2^2 + x_1 x_2 + 4x_1$

Apply Newton's algorithm for one iteration to find an updated value for $\vec{\mathbf{x}} = [x_1 \quad x_2]^T$ starting from the initial point $\vec{x^0} = [2 \quad 3]^T$.

Step 1

$$\vec{x}^0 = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

Set $\varepsilon = 10^{-6}$

$k = 0$

Step 2

$$\vec{g} = \begin{bmatrix} 2x_1 + x_2 + 4 \\ 2x_2 + x_1 \end{bmatrix}$$

At $\vec{x}^0 = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$ $\quad \vec{g}^0 = \begin{bmatrix} 11 \\ 8 \end{bmatrix}$

$\|\vec{g}^0\| = 13.6 > \varepsilon$ continue

Step 3

$$\vec{H}^{0} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

$$\vec{H}^{-1} = \begin{bmatrix} 0.6667 & -0.333 \\ -0.333 & 0.6667 \end{bmatrix}$$

Step 4

$$\vec{d}^{0} = -\vec{H}^{-1}\vec{g}^{0} = -\begin{bmatrix} 0.6667 & -0.333 \\ -0.333 & 0.6667 \end{bmatrix}\begin{bmatrix} 1 \\ 8 \end{bmatrix}$$

$$= \begin{bmatrix} -4.6667 \\ -1.6667 \end{bmatrix}$$

Step 5

$$\vec{x}^1 = \begin{bmatrix} 2 \\ 3 \end{bmatrix} + \begin{bmatrix} -4.6667 \\ -1.6667 \end{bmatrix} \eta^\circ = \begin{bmatrix} 2 - 4.6667\,\eta^\circ \\ 3 - 1.6667\,\eta^\circ \end{bmatrix}$$

$$E(\eta^\circ) = E(\vec{x}^\circ + \eta^\circ \vec{d}^\circ) =$$

$$= (2 - 4.6667\eta^\circ + 2)^2 + (3 - 1.6667\eta^\circ)^2$$

$$+ (2 - 4.6667\eta^\circ)(3 - 1.6667\eta^\circ) + 4$$

$$E(\eta^\circ) = 32.34\,(\eta^\circ)^2 - 64.67\eta^\circ + 35$$

$$\frac{dE(\eta^\circ)}{d\eta^\circ} = 64.68\,\eta^\circ - 64.67 = 0$$

$$\eta^\circ \cong 1$$

$$\frac{d^2E(\eta^\circ)}{d\eta^{\circ 2}} = 64.68 > 0$$

So, it is a minimum point.

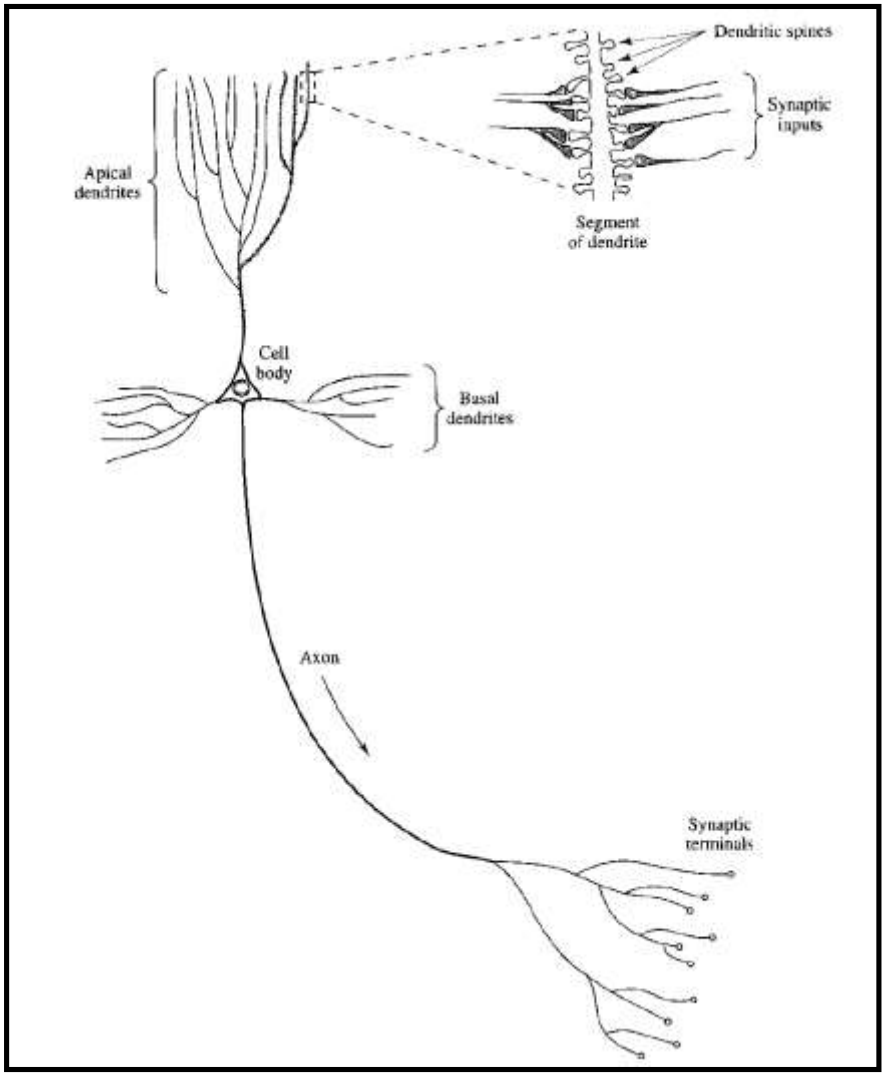
Step 6

$$\vec{x}^1 = \begin{bmatrix} 2 \\ 3 \end{bmatrix} - \begin{bmatrix} 4.6667 \\ 1.6667 \end{bmatrix} = \begin{bmatrix} -2.6667 \\ 1.3337 \end{bmatrix}$$

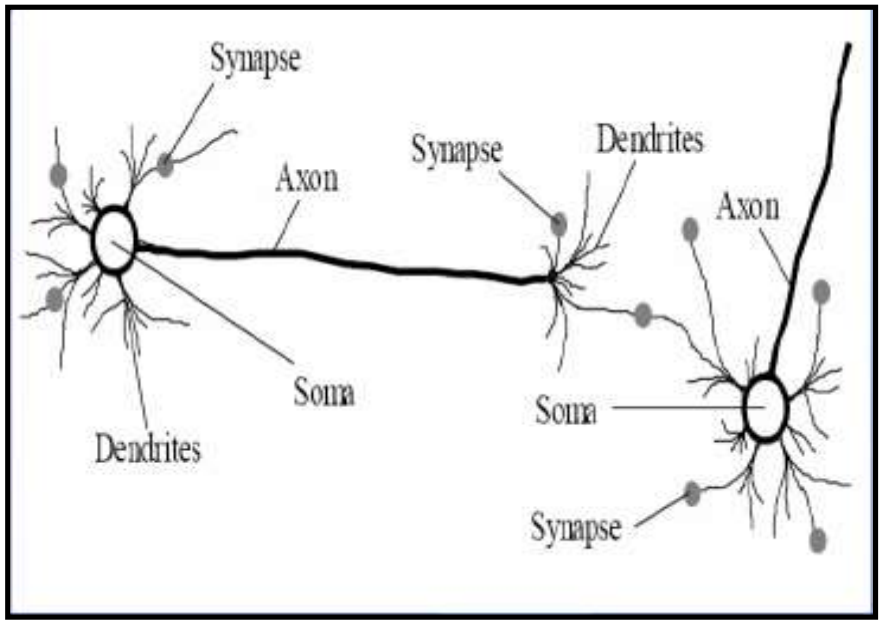
KON 426E
INTELLIGENT CONTROL SYSTEMS

LECTURE 4

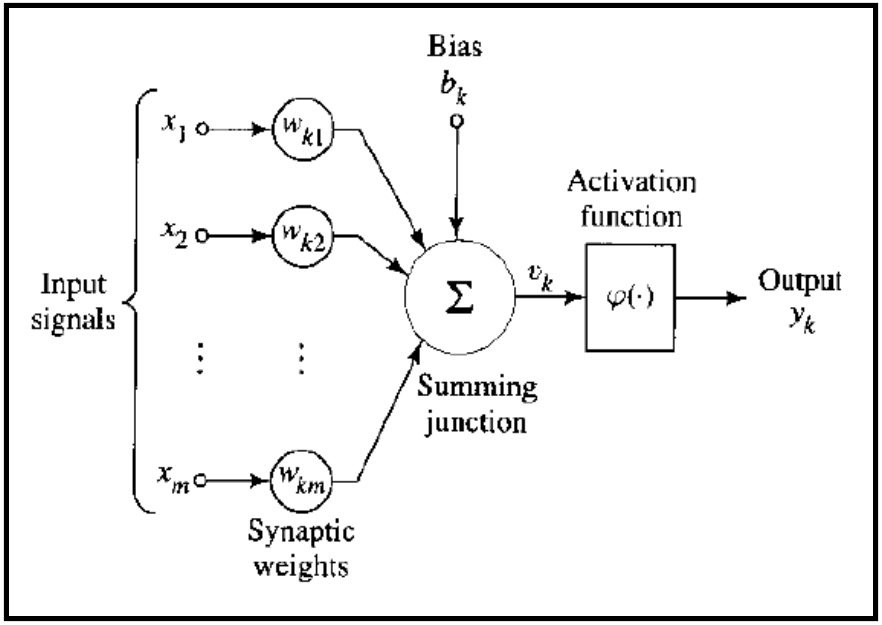
14/03/2022



- **Dendrites** are like **receptive zones**.
- **Cell body (soma)** is the nucleus.
- Signals are received at the dendrites.
- They are summed up at the cell body.
- If the incoming signals are greater than a **threshold**, the cell is **excited**.
- The signals are transmitted through the **axon** electrically.
- Axon is like a transmission line.
- Between two neighboring neurons, there is a gap called as a **synapse**.
- At the synapse, the electrical signal is concerted to a chemical molecule, called as a **neurotransmitter.**

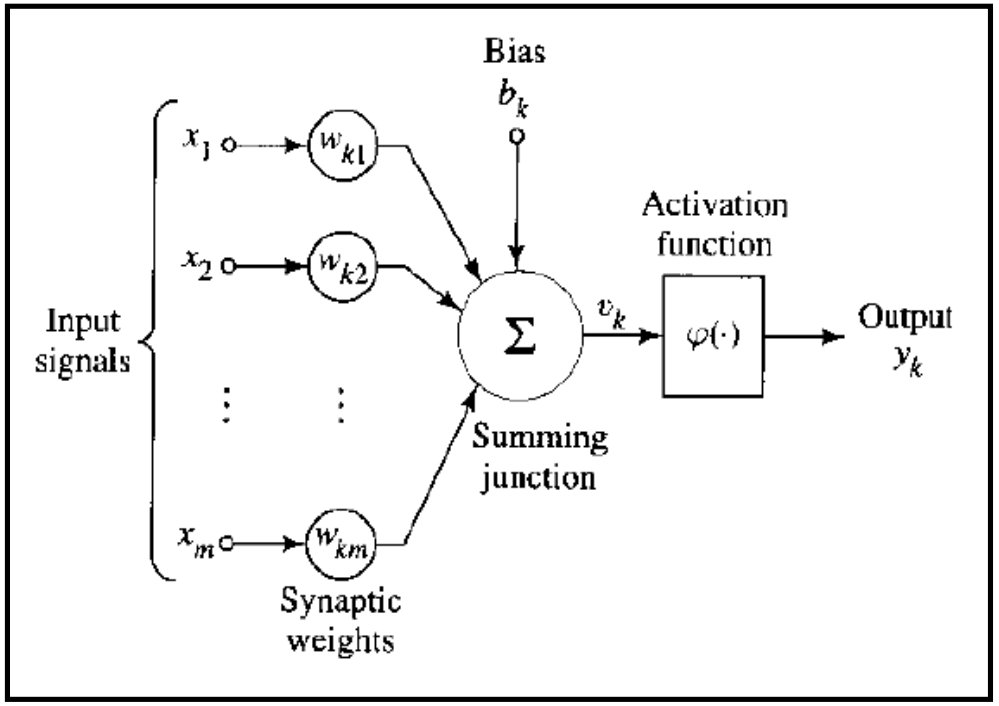


Biological Neuron



Artificial Neuron

ROSENBLATT'S PERCEPTRON (1958)
MC CULLOCH-PITTS PERCEPTRON



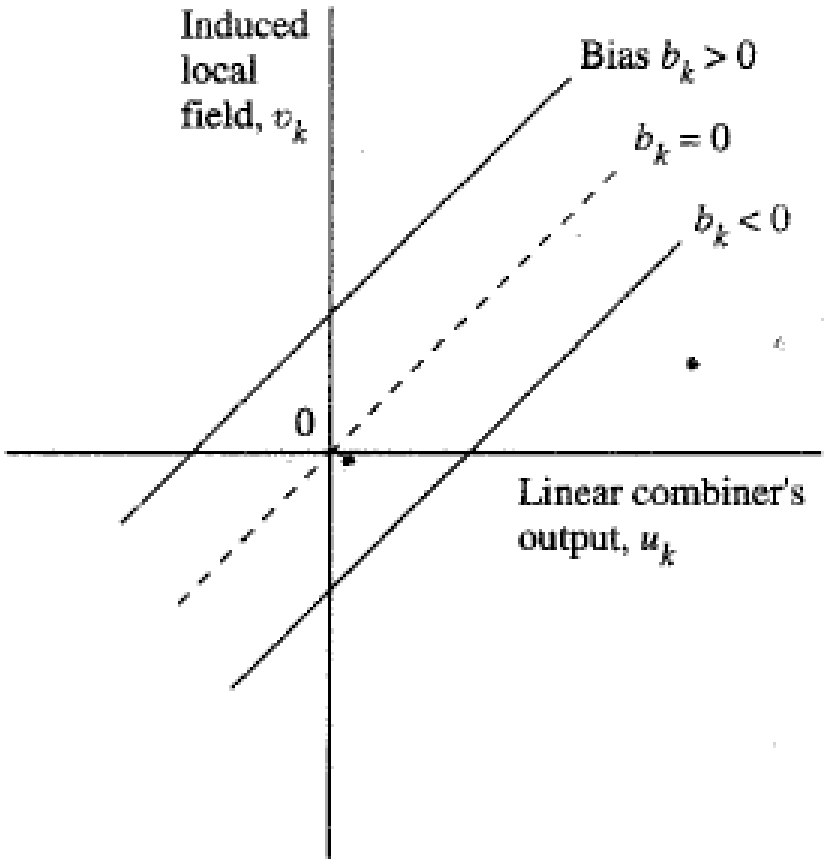
$$u_k = \sum_{j=1}^{m} w_{kj} x_j$$

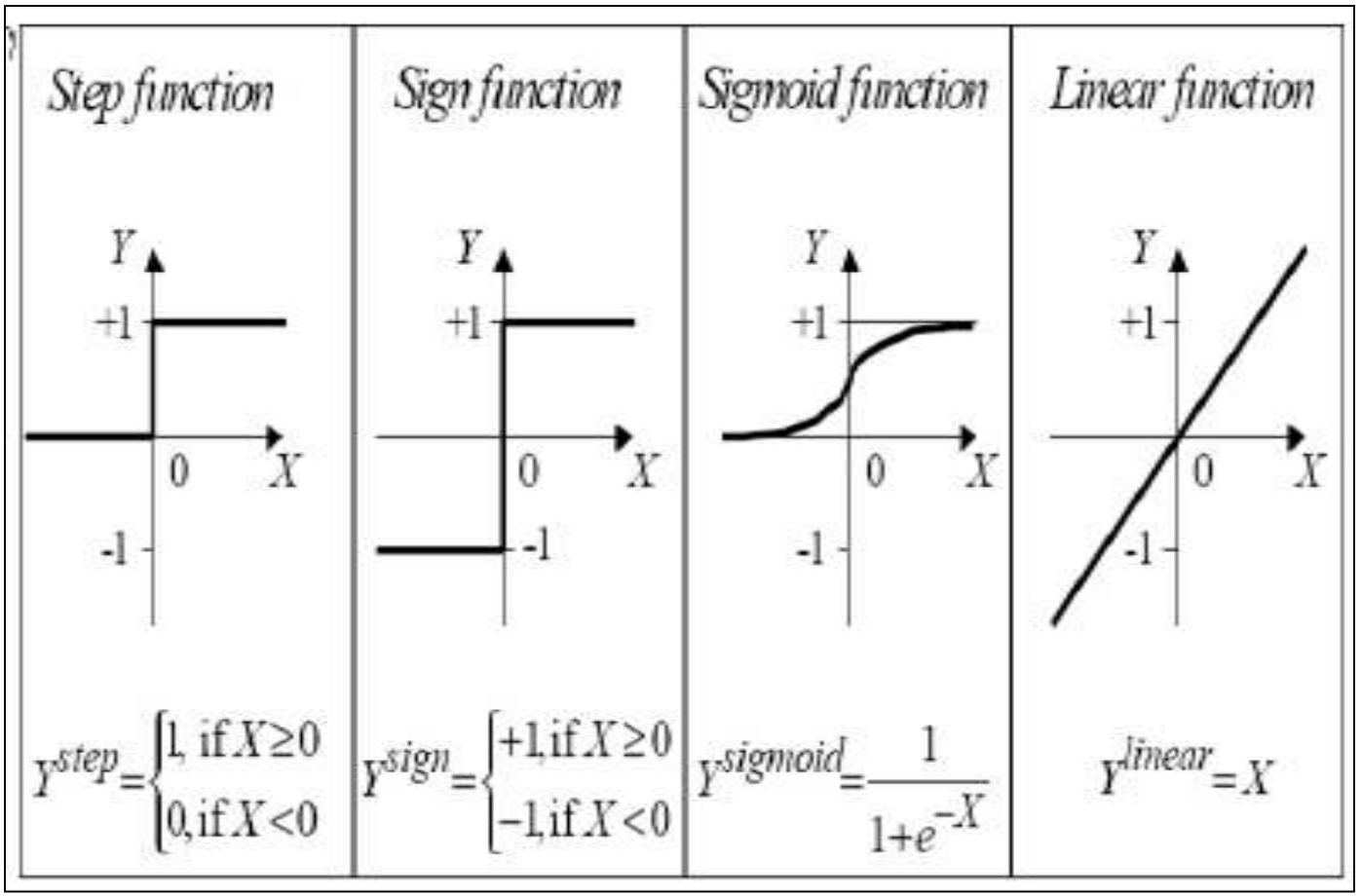
$$v_k = u_k + b_k$$

$$y_k = \varphi(v_k)$$

v_k : induced local field

y_k :output


Induced
local
field, v_k
Bias $b_k > 0$
$b_k = 0$
$b_k < 0$
0
Linear combiner's
output, u_k



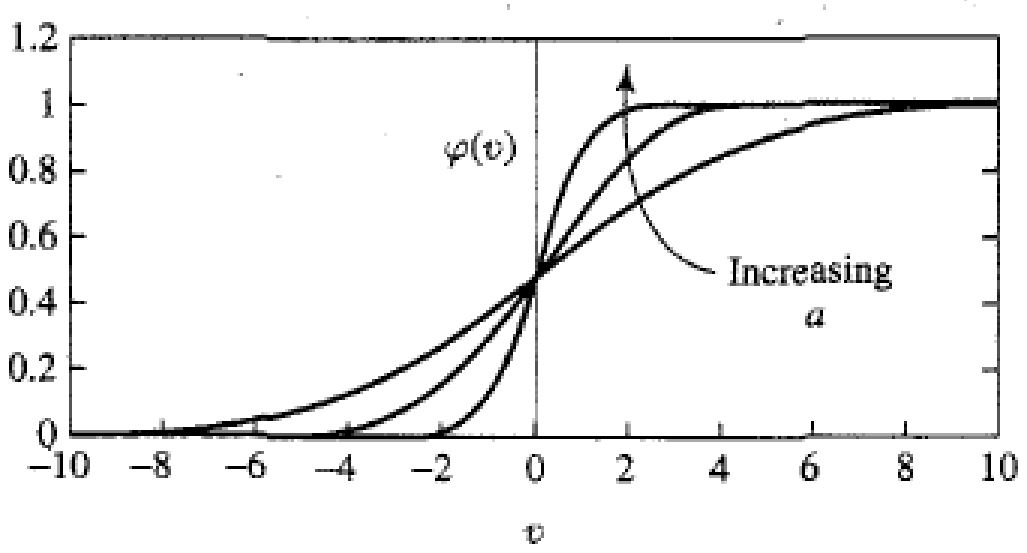
Activation functions

- If you are solving a **classification** problem, at the output layer of an MLP, you should use **a two-valued activation function** (step, sgn, sigmoid, etc.)
- If you are solving a **regression** problem, at the output layer you should use a **linear function**, at the hidden layers you can use two-valued functions.
- Sigmoid and hyperbolic tangent functions are **differentiable**, and this is very importantin implementing learning algorithms (backpropagation algorithm)

Parameterized Activation Functions:

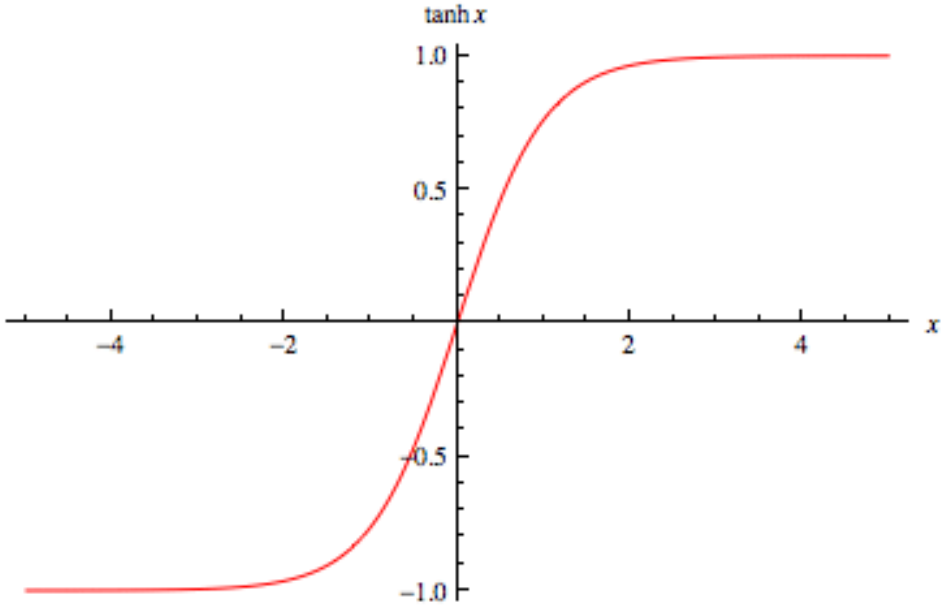
Parameterized Sigmoid Function: $$\varphi(v) = \frac{1}{1 + \exp(-av)}$$

a is the slope parameter.



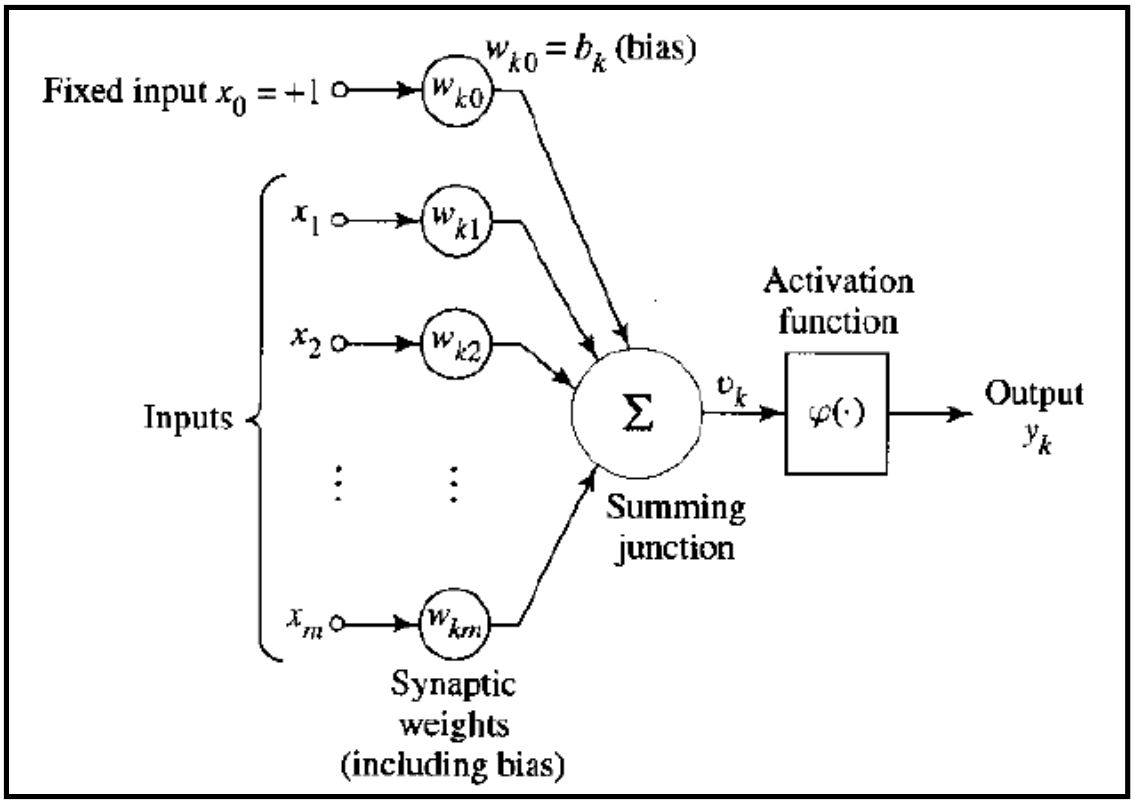
Hyperbolic Tangent Function

$$tanhx = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$



Parameterized Hyperbolic Tangent Function

$$tanhax = \frac{e^{ax} - e^{-ax}}{e^{ax} + e^{-ax}}$$



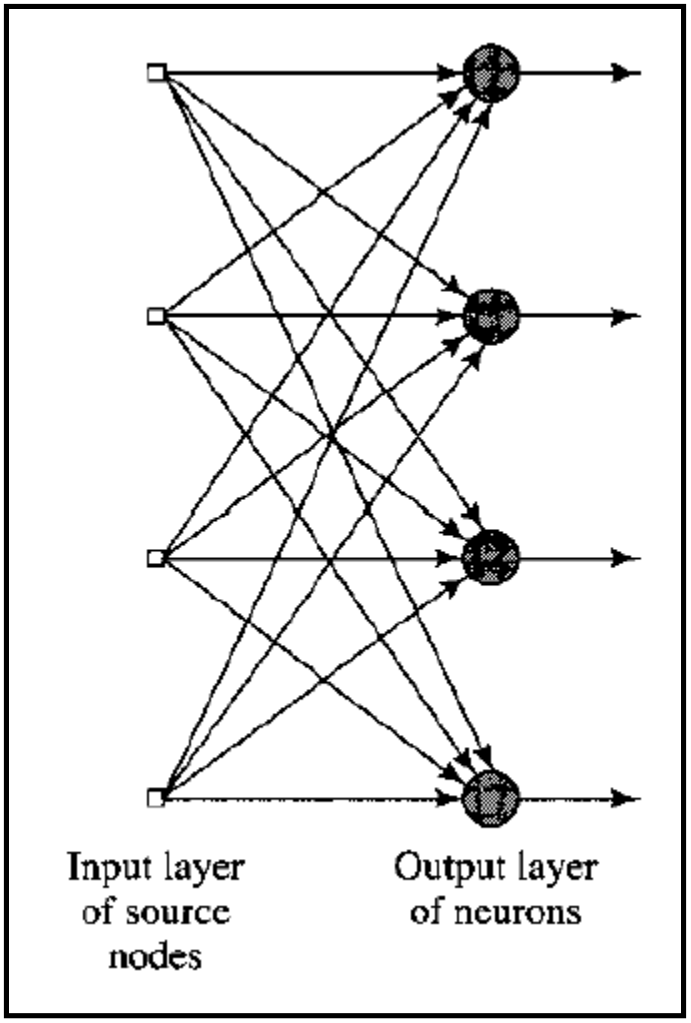
$$\vec{\mathbf{x}} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix}$$

$$\vec{\mathbf{w}} = \begin{bmatrix} w_{k0} & w_{k1} & w_{k2} & \dots\dots & w_{kn} \end{bmatrix}$$

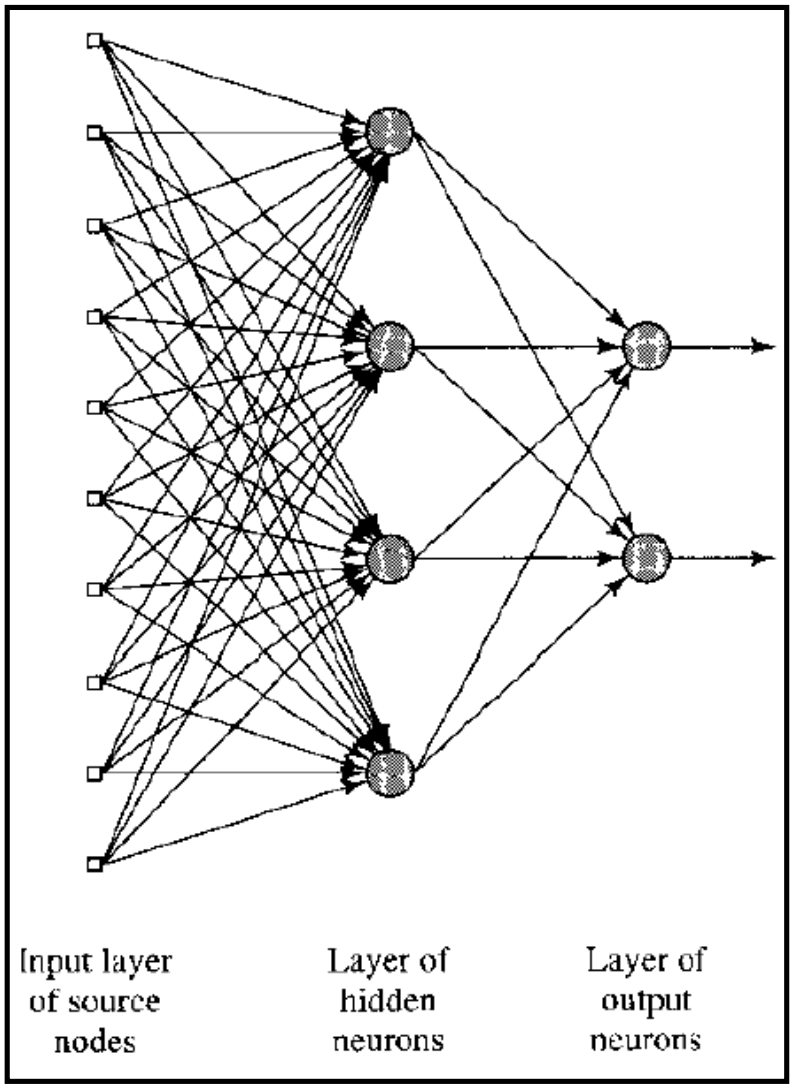
$$v_k = \sum_{i=1}^{n} w_{ki} x_i + w_0 = \vec{\mathbf{w}}.\vec{\mathbf{x}}$$

$$y_k = \varphi(v_k) = \begin{cases} 1 & v_k \geq 0 \\ 0 & v_k < 0 \end{cases}$$

Single-Layer Feedforward Networks



Multi-Layer Feedforward Networks

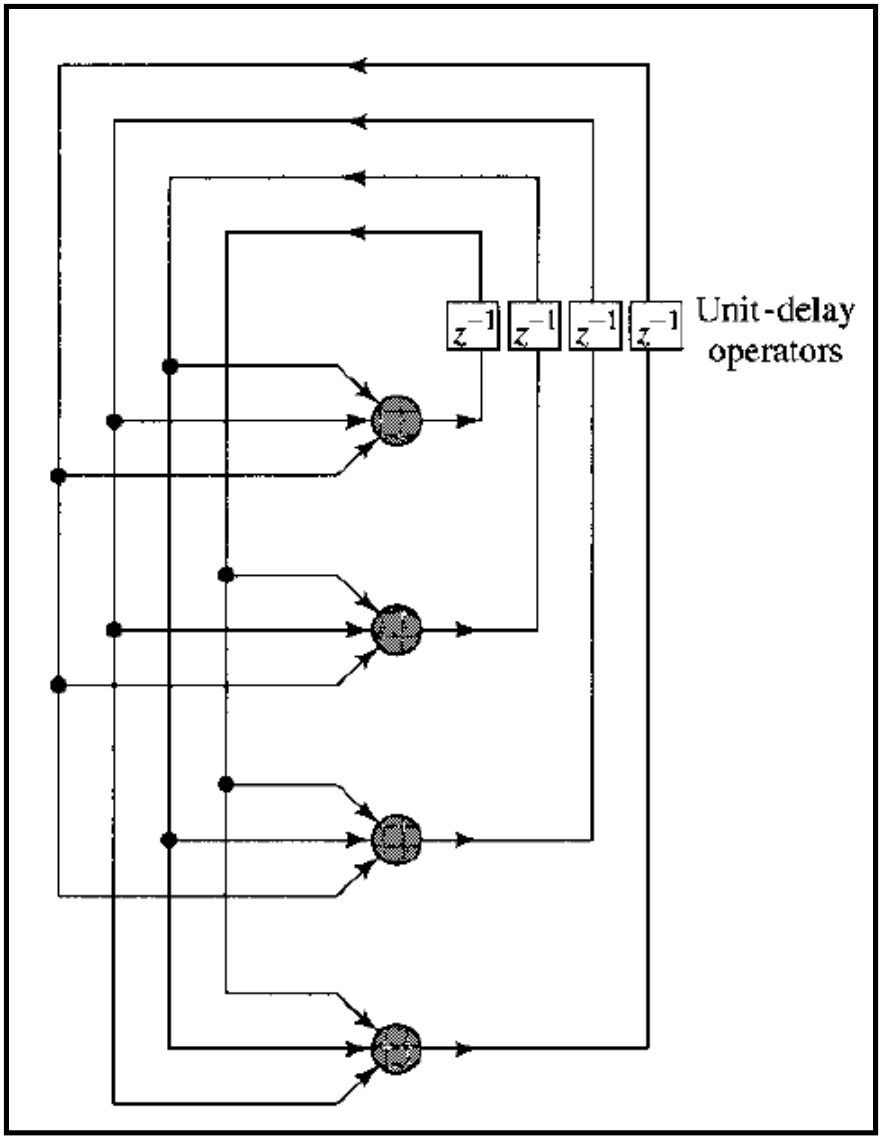


This is a 10-4-2 network.

By adding hidden layers, the network is able to extract higher-order statistics.

The network acquires a global perspective despite its local connections due to the extra set of synaptic connections and neural interactions.

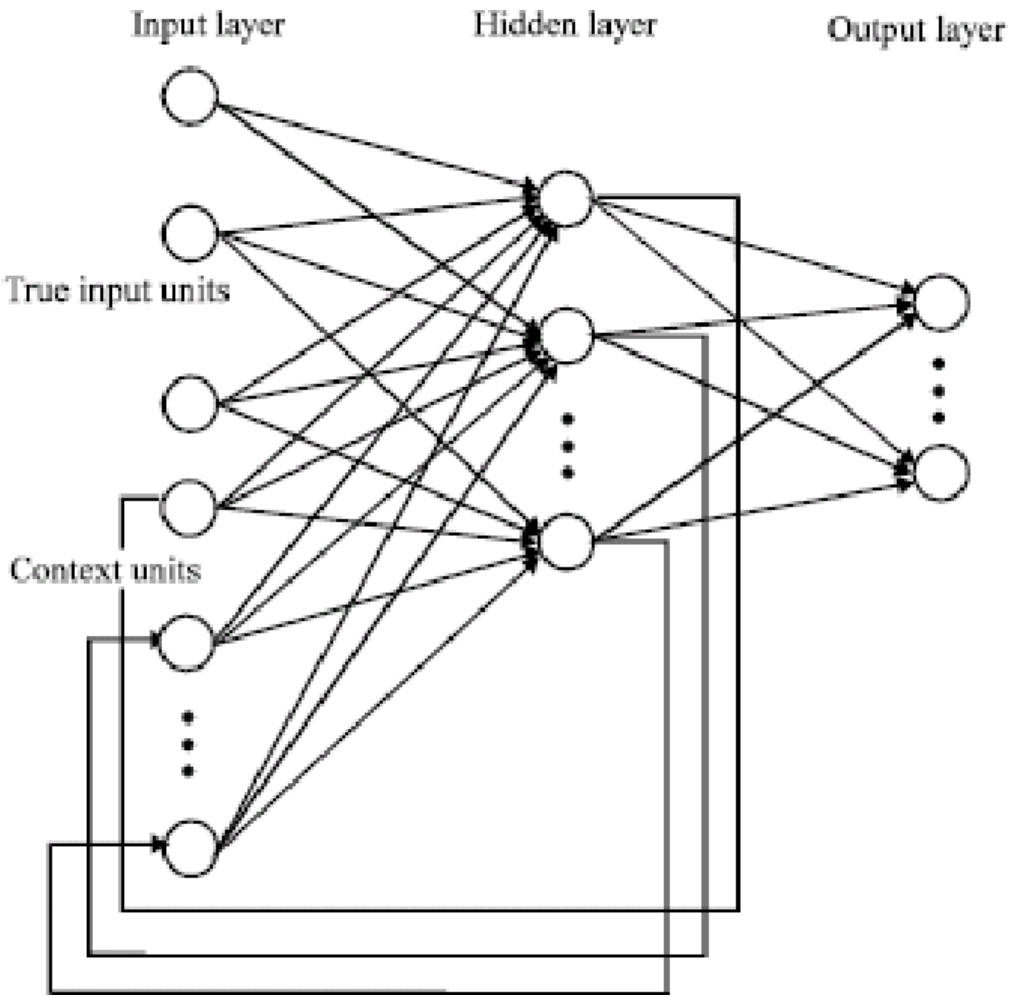
Recurrent Neural Networks



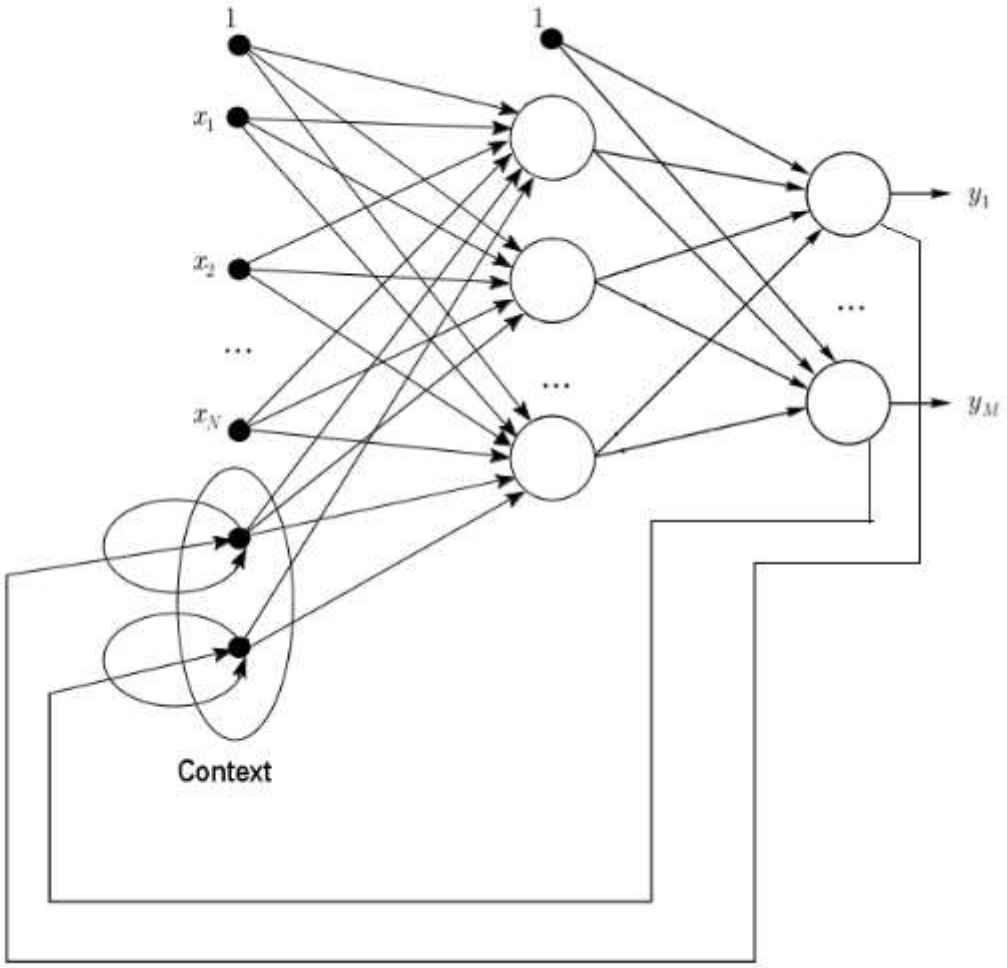
A recurrent neural network is different from a feedforward neural network in that it contains at least one feedback loop.

Hopfield network

Recurrent neural network with no self-feedback loops
and no hidden neurons

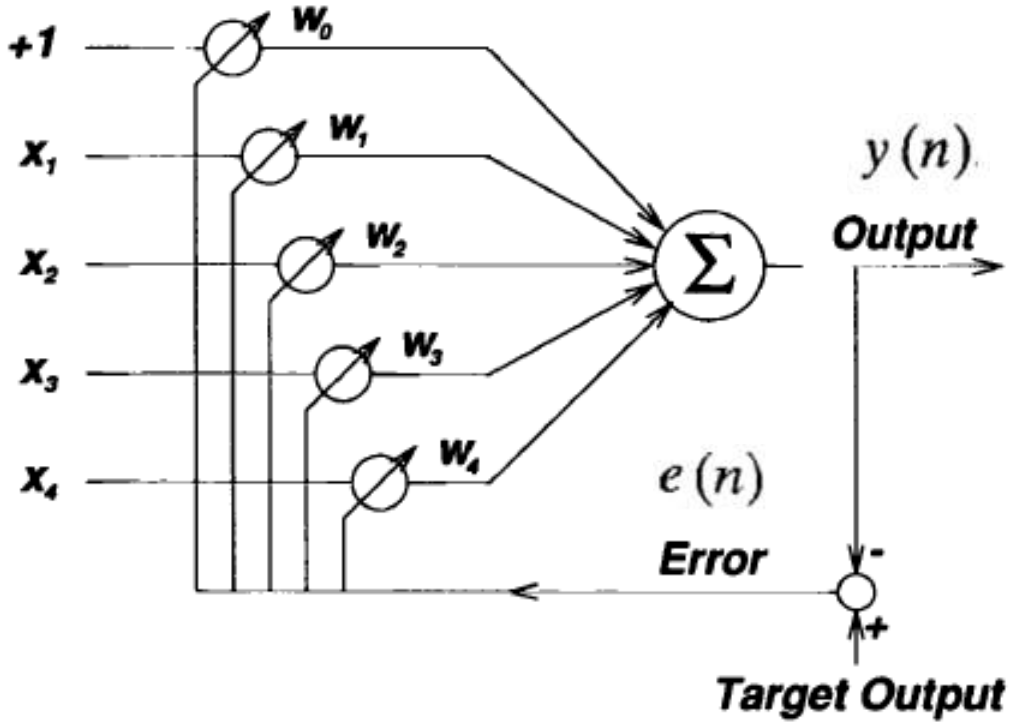


Elman Neural Network



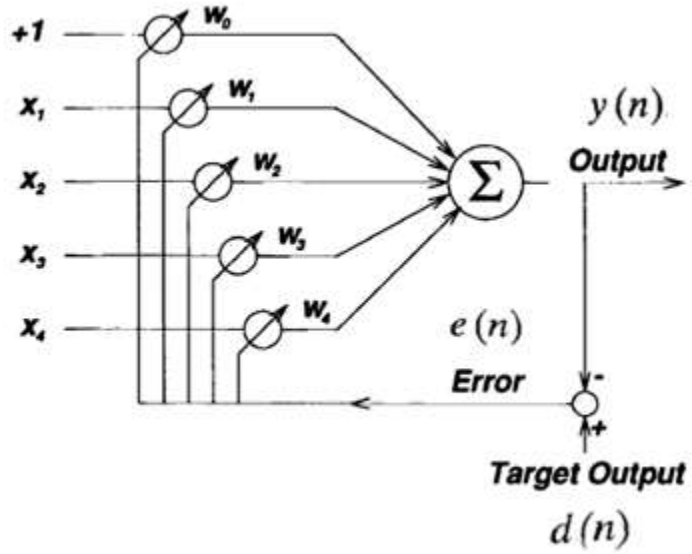
Jordan Neural Network

ADALINE (Adaptive Linear Element)



$$y(n) = \mathbf{x}^T(n)\mathbf{w}(n)$$

Least Means Squares (LMS) Algorithm (En Küçük Kareler Yöntemi)



Cost Function

$$\mathscr{E}(\mathbf{w}) = \frac{1}{2}e^2(n)$$

The least-mean-square (LMS) algorithm minimizes the instantaneous value of the cost function

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \boldsymbol{\eta}\mathbf{g}(n) = \mathbf{w}(n) - \eta\frac{\partial E}{\partial \mathbf{w}}$$

$$\frac{\partial \mathscr{E}(\mathbf{w})}{\partial \mathbf{w}} = e(n)\frac{\partial e(n)}{\partial \mathbf{w}}$$

$$y(n) = \mathbf{x}^T(n)\mathbf{w}(n)$$

$$e(n) = d(n) - \mathbf{x}^T(n)\mathbf{w}(n)$$

$$\frac{\partial e(n)}{\partial \mathbf{w}(n)} = -\mathbf{x}(n)$$

$$\frac{\partial \mathscr{E}(\mathbf{w})}{\partial \mathbf{w}(n)} = -\mathbf{x}(n)e(n)$$

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta\mathbf{x}(n)e(n)$$

⇩

Error correction learning

TABLE 3.1 Summary of the LMS Algorithm

Training Sample: Input signal vector = $\mathbf{x}(n)$
Desired response = $d(n)$

User-selected parameter: η

Initialization. Set $\mathbf{w}(0) = \mathbf{0}$.

Computation. For $n = 1, 2, \ldots$, compute

$$e(n) = d(n) - \mathbf{w}^T(n)\mathbf{x}(n)$$

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta\mathbf{x}(n)e(n)$$

LEARNING RATE SCHEDULES

- Simplest possible form:

$$\eta(n) = \eta_0 \qquad \text{for all } n$$

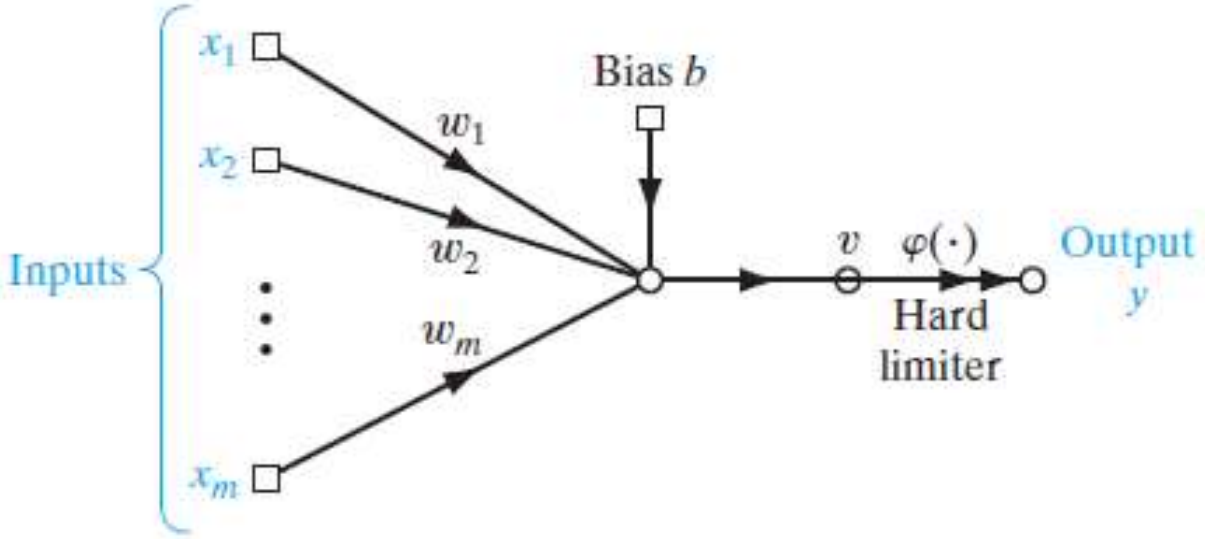
- Stochastic approximation: (Kushner and Clark, 1978)

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

- Search-then-converge: (Darken and Moody, 1992)

$$\eta(n) = \frac{\eta_0}{1 + (n/\tau)}$$

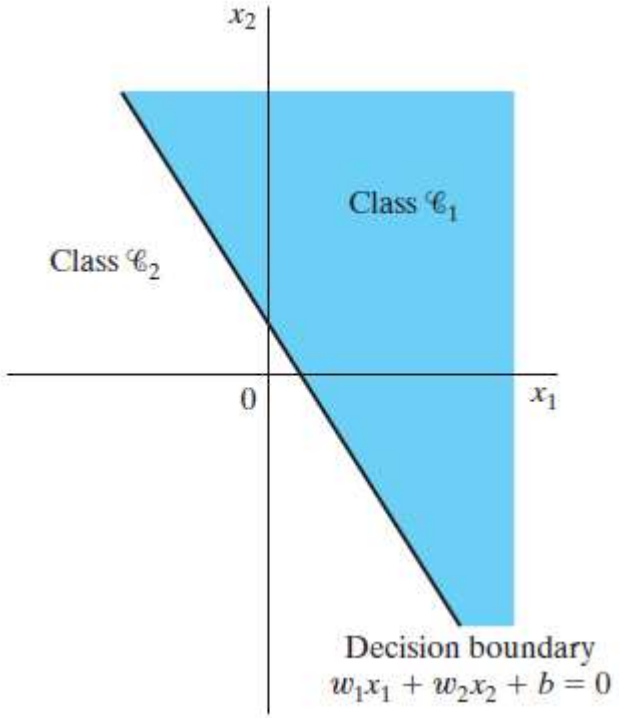
PERCEPTRON CONVERGENCE THEOREM



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

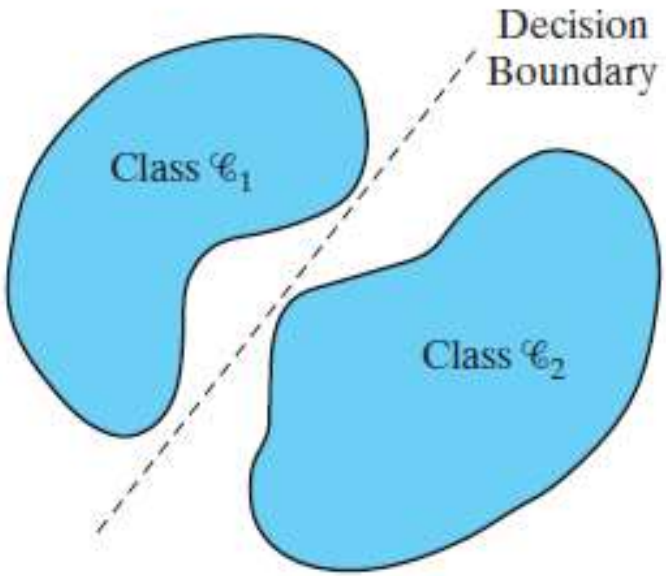
Let us consider the pattern classification problem:

The goal of the perceptron is to correctly classify the set of externally applied stimuli $x_1, x_2, \ldots, x_m$ into one of two classes, $\mathscr{C}_1$ or $\mathscr{C}_2$.
The decision rule for the classification is to assign the point represented by the inputs $x_1, x_2, \ldots, x_m$ to class $\mathscr{C}_1$ if the perceptron output *y* is +1 and to class $\mathscr{C}_2$ if *y* is -1.
For the case of two input variables x_1 and x_2 and two classes:

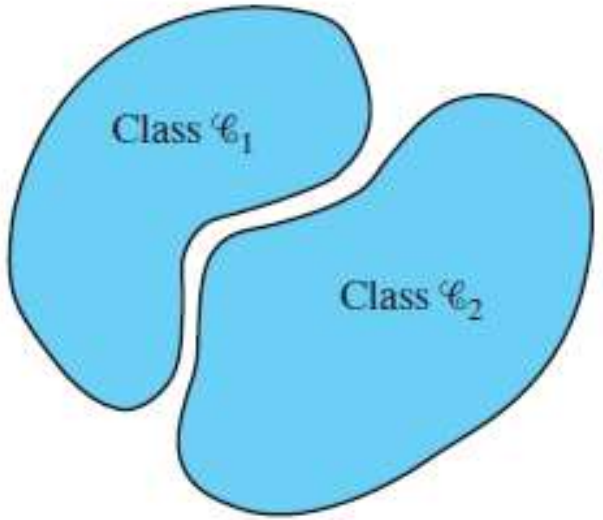


Two decision regions are separated by a **hyperplane** defined by:

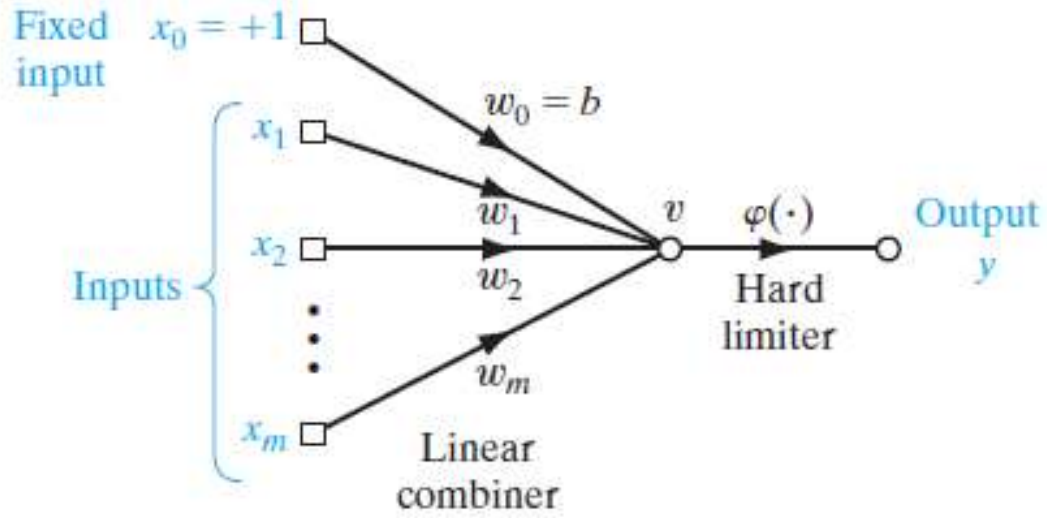
$$\sum_{i=1}^{m} w_i x_i + b = 0$$



A pair of linearly separable patterns



A pair of non-linearly separable patterns



$$\mathbf{x}(n) = [+1, x_1(n), x_2(n), ..., x_m(n)]^T$$ $(m + 1)$-by-1 input vector

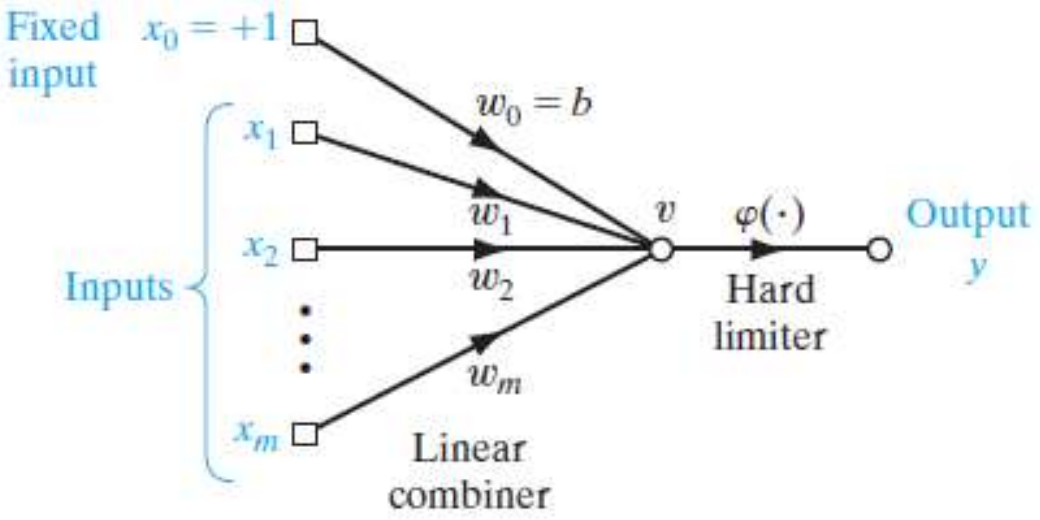
$$\mathbf{w}(n) = [b, w_1(n), w_2(n), ..., w_m(n)]^T$$ $(m + 1)$-by-1 weight vector

$$v(n) = \sum_{i=0}^{m} w_i(n)x_i(n)$$
$$= \mathbf{w}^T(n)\mathbf{x}(n)$$

Training: Assume we have training vectors that belong to both class $\mathscr{C}_1$ and class $\mathscr{C}_2$. We feed the training vectors to the perceptron.

We are trying to find a weight vector **w** such that:

$$\mathbf{w}^T\mathbf{x} > 0 \text{ for every input vector } \mathbf{x} \text{ belonging to class } \mathscr{C}_1$$
$$\mathbf{w}^T\mathbf{x} \leq 0 \text{ for every input vector } \mathbf{x} \text{ belonging to class } \mathscr{C}_2$$



The algorithm for adapting the weight vector $\mathbf{w}$:

1. If the nth member of the training set, $\mathbf{x}(n)$, is correctly classified by the weight vector $\mathbf{w}(n)$ computed at the nth iteration of the algorithm, no correction is made to the weight vector of the perceptron in accordance with the rule:

$$\mathbf{w}(n+1) = \mathbf{w}(n) \quad \text{if} \quad \mathbf{w}^T(n)\mathbf{x}(n) > 0 \quad \text{and} \quad \mathbf{x}(n) \text{ belongs to } \mathcal{C}_1$$
$$\mathbf{w}(n+1) = \mathbf{w}(n) \quad \text{if} \quad \mathbf{w}^T(n)\mathbf{x}(n) \leq 0 \quad \text{and} \quad \mathbf{x}(n) \text{ belongs to } \mathcal{C}_2$$

2. Otherwise, the weight vector of the perceptron is updated in accordance with the rule:

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \eta(n)\mathbf{x}(n) \quad \text{if} \quad \mathbf{w}^T(n)\mathbf{x}(n) > 0 \quad \text{and} \quad \mathbf{x}(n) \text{ belongs to } \mathcal{C}_2$$
$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta(n)\mathbf{x}(n) \quad \text{if} \quad \mathbf{w}^T(n)\mathbf{x}(n) \leq 0 \quad \text{and} \quad \mathbf{x}(n) \text{ belongs to } \mathcal{C}_1$$

How are the update rules obtained?

Least Means Squares (LMS) Algorithm:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta \mathbf{x}(n) e(n)$$

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

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

$$d(n) = \begin{cases} +1 & \text{if } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_1 \\ -1 & \text{if } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_2 \end{cases}$$

Case 1

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \eta(n)\mathbf{x}(n) \text{ if } \mathbf{w}^T(n)\mathbf{x}(n) > 0 \text{ and } \mathbf{x}(n) \text{ belongs to } \mathscr{C}_2$$

$$\mathbf{w}^T(n)\mathbf{x}(n) > 0 \longrightarrow y(n) = +1$$

$$\mathbf{x}(n) \text{ belongs to } \mathscr{C}_2 \longrightarrow d(n) = -1$$

$$e(n) = d(n) - y(n) = -1 - (+1) = -2$$

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta [d(n) - y(n)] \mathbf{x}(n) = \mathbf{w}(n) - \eta\, \mathbf{x}(n)$$

-2 (can be combined with η)

Case 2

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta(n)\mathbf{x}(n) \quad \text{if } \mathbf{w}^T(n)\mathbf{x}(n) \leq 0 \text{ and } \mathbf{x}(n) \text{ belongs to } \mathcal{C}_1$$

$$\mathbf{w}^T(n)\mathbf{x}(n) \leq 0 \longrightarrow y(n) = -1$$

$$\mathbf{x}(n) \text{ belongs to } \mathcal{C}_1 \longrightarrow d(n) = +1$$

$$e(n) = d(n) - y(n) = +1 - (-1) = +2$$

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

+2 (can be combined with η)

TABLE 1.1 Summary of the Perceptron Convergence Algorithm

Variables and Parameters:

$\mathbf{x}(n)$ = $(m+1)$-by-1 input vector
$= [+1, x_1(n), x_2(n), \ldots, x_m(n)]^T$
$\mathbf{w}(n)$ = $(m+1)$-by-1 weight vector
$= [b, w_1(n), w_2(n), \ldots, w_m(n)]^T$
b = bias
$y(n)$ = actual response (quantized)
$d(n)$ = desired response
η = learning-rate parameter, a positive constant less than unity

1. *Initialization.* Set $\mathbf{w}(0) = \mathbf{0}$. Then perform the following computations for time-step $n = 1, 2, \ldots$.
2. *Activation.* At time-step n, activate the perceptron by applying continuous-valued input vector $\mathbf{x}(n)$ and desired response $d(n)$.
3. *Computation of Actual Response.* Compute the actual response of the perceptron as

$$y(n) = \text{sgn}[\mathbf{w}^T(n)\mathbf{x}(n)]$$

where $\text{sgn}(\cdot)$ is the signum function.

4. *Adaptation of Weight Vector.* Update the weight vector of the perceptron to obtain

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

where

$$d(n) = \begin{cases} +1 & \text{if } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_1 \\ -1 & \text{if } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_2 \end{cases}$$

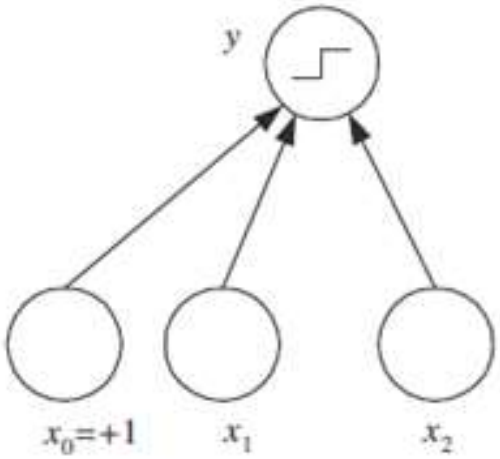
5. *Continuation.* Increment time step n by one and go back to step 2.

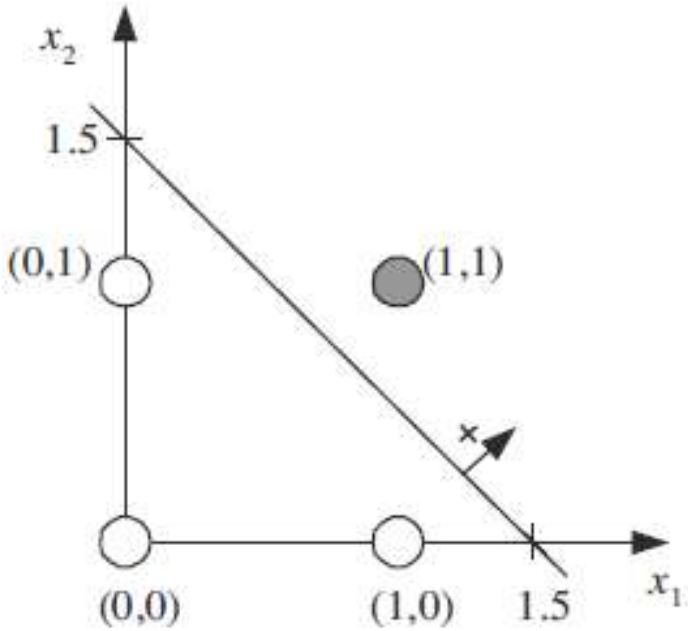
- Study the mathematical proof of Perceptron Convergence Theorem on pages 52-54 of Haykin ,1999.

LIMITATIONS OF THE SINGLE PERCEPTRON

Boolean AND Function

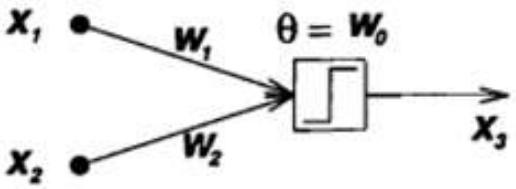
x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1

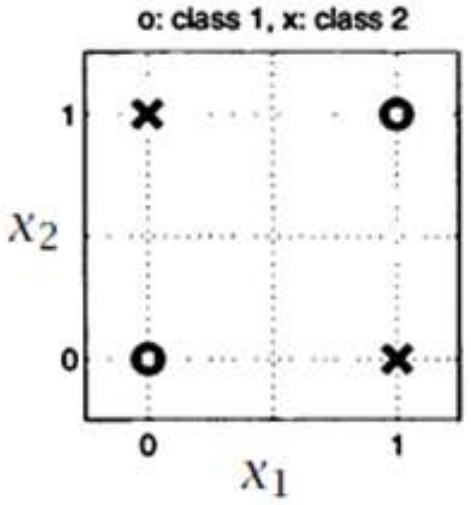




Exclusive XOR Function

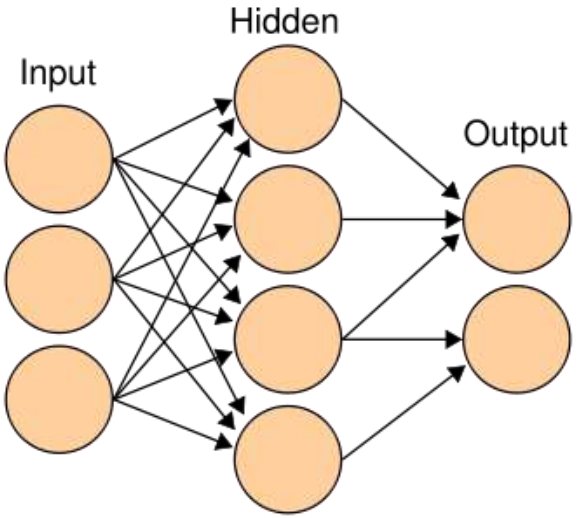
x_1	x_2	r
0	0	0
0	1	1
1	0	1
1	1	0

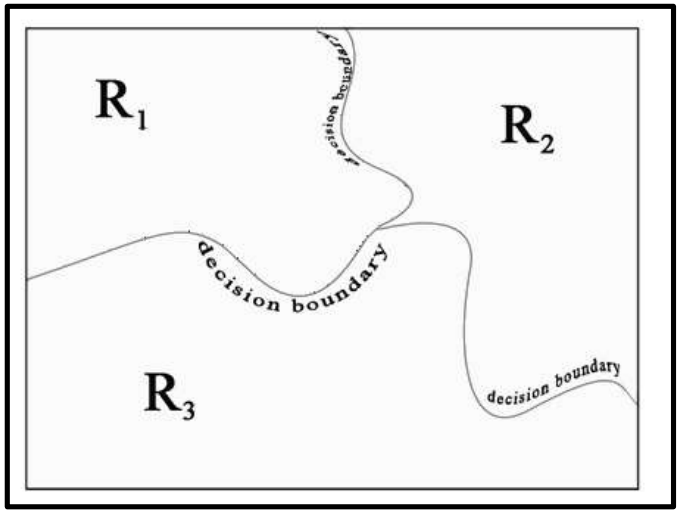


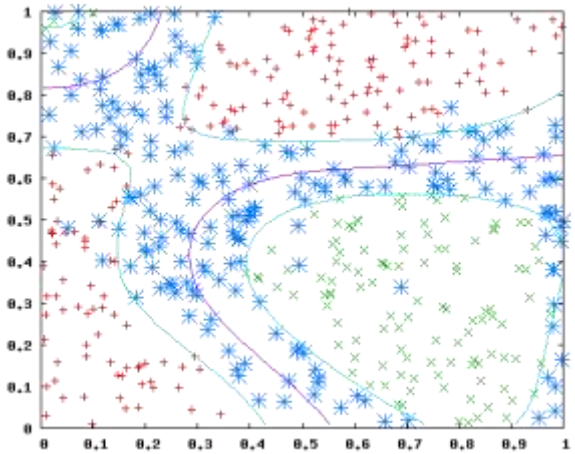


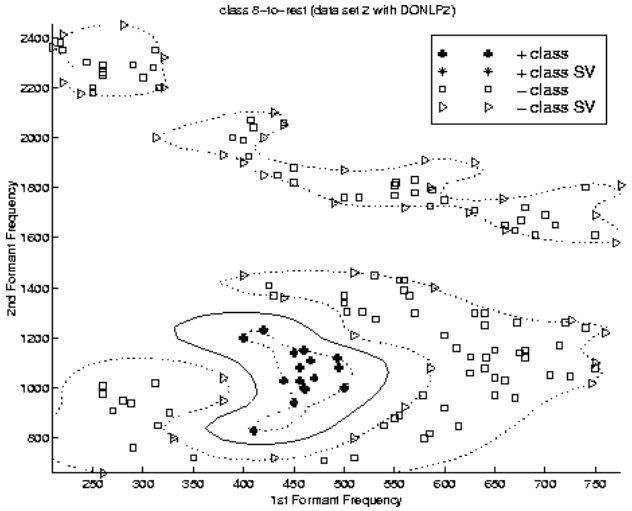
$$
\begin{aligned}
0 \times w_1 + 0 \times w_2 + w_0 \leq 0 &\iff w_0 \leq 0, \\
0 \times w_1 + 1 \times w_2 + w_0 > 0 &\iff w_0 > -w_2 \\
1 \times w_1 + 0 \times w_2 + w_0 > 0 &\iff w_0 > -w_1 \\
1 \times w_1 + 1 \times w_2 + w_0 \leq 0 &\iff w_0 \leq -w_1 - w_2
\end{aligned}
$$

contradictory conditions


Input
Hidden
Output


R1
R2
R3
decision boundary
decision boundary
decision boundary




class 8-to-rest (data set 2 with DONLP2)
+class
+class SV
-class
-class SV
2nd Formant Frequency
1st Formant Frequency

A BRIEF INTRODUTION TO BAYESIAN CLASSIFICATION

A Review of Probability Theory

Conditional Probability

$P(E|F)$ is the probability of the occurrence of event E given that F has occurred.

$$P(E|F) = \frac{P(E \cap F)}{P(F)}$$

$\cap$ is commutative.

$$P(E \cap F) = P(E|F)P(F) = P(F|E)P(E)$$

Bayes' Formula:

$$P(F|E) = \frac{P(E|F)P(F)}{P(E)}$$

When F_i are mutually exclusive and exhaustive, that is $\bigcup_{i=1}^{n} F_i = S$

$$E = \bigcup_{i=1}^{n} E \cap F_i$$

$$P(E) = \sum_{i=1}^{n} P(E \cap F_i) = \sum_{i=1}^{n} P(E|F_i)P(F_i)$$

Using Bayes' formula:

$$P(F_i|E) = \frac{P(E \cap F_i)}{P(E)} = \frac{P(E|F_i)P(F_i)}{\sum_j P(E|F_j)P(F_j)}$$

If E and F are independent, $P(E|F) = P(E)$

$$P(E \cap F) = P(E)P(F)$$

Classification Problem (2-Class Problem)

Let **x** denote the vector of observed variables.

Bernoulli random variable C is a random variable that denotes the Class.

Using Bayes' Theory,

$$P(C|\boldsymbol{x}) = \frac{P(C)p(\boldsymbol{x}|C)}{p(\boldsymbol{x})}$$

For a two-class classification problem:

Let $P(C = 1)$ denote the probability that the class is Class 1.

$P(C = 0)$ denote the probability that the class is Class 2.

$P(C = 1)$ is called the **prior (önsel) probability** that C takes the value 1. (The training sample belongs the Class 1)

$$P(C = 0) + P(C = 1) = 1$$

$p(\boldsymbol{x}|C)$ is called the **class likelihood** and is the conditional probability that an event belonging to C has the associated observation value $\boldsymbol{x}$

$P(C|\boldsymbol{x})$ is called the **posterior (sonsal)** probability.

$p(\boldsymbol{x})$ is called the **evidence**, is the marginal probability that an observation $\boldsymbol{x}$ is seen, regardless of whether it is a positive or negative example.

$$p(\boldsymbol{x}) = \sum_{C} p(\boldsymbol{x}, C) = p(\boldsymbol{x}|C = 1)P(C = 1) + p(\boldsymbol{x}|C = 0)P(C = 0)$$

Note that:

$$P(C|\boldsymbol{x}) = \frac{P(C)p(\boldsymbol{x}|C)}{p(\boldsymbol{x})}$$

$$\text{posterior} = \frac{\text{prior} \times \text{likelihood}}{\text{evidence}}$$

Evidence is normalized, therefore posteriors sum up to 1:

$$P(C = 0|\boldsymbol{x}) + P(C = 1|\boldsymbol{x}) = 1$$

K- Class Classification:

In the general case, we have K mutually exclusive and exhaustive classes; $C_i, i = 1, \ldots, K$

The prior probabilities satisfy:

$$P(C_i) \geq 0 \text{ and } \sum_{i=1}^{K} P(C_i) = 1$$

The posterior probability of class C_i is:

$$P(C_i|\mathbf{x}) = \frac{p(\mathbf{x}|C_i)P(C_i)}{p(\mathbf{x})} = \frac{p(\mathbf{x}|C_i)P(C_i)}{\sum_{k=1}^{K} p(\mathbf{x}|C_k)P(C_k)}$$

Here, $p(\mathbf{x}|C_i)$ is the probability of seeing $\mathbf{x}$ as the input when it is known to belong to class C_i

The **Bayes' classifier** chooses the class with the highest posterior probability:

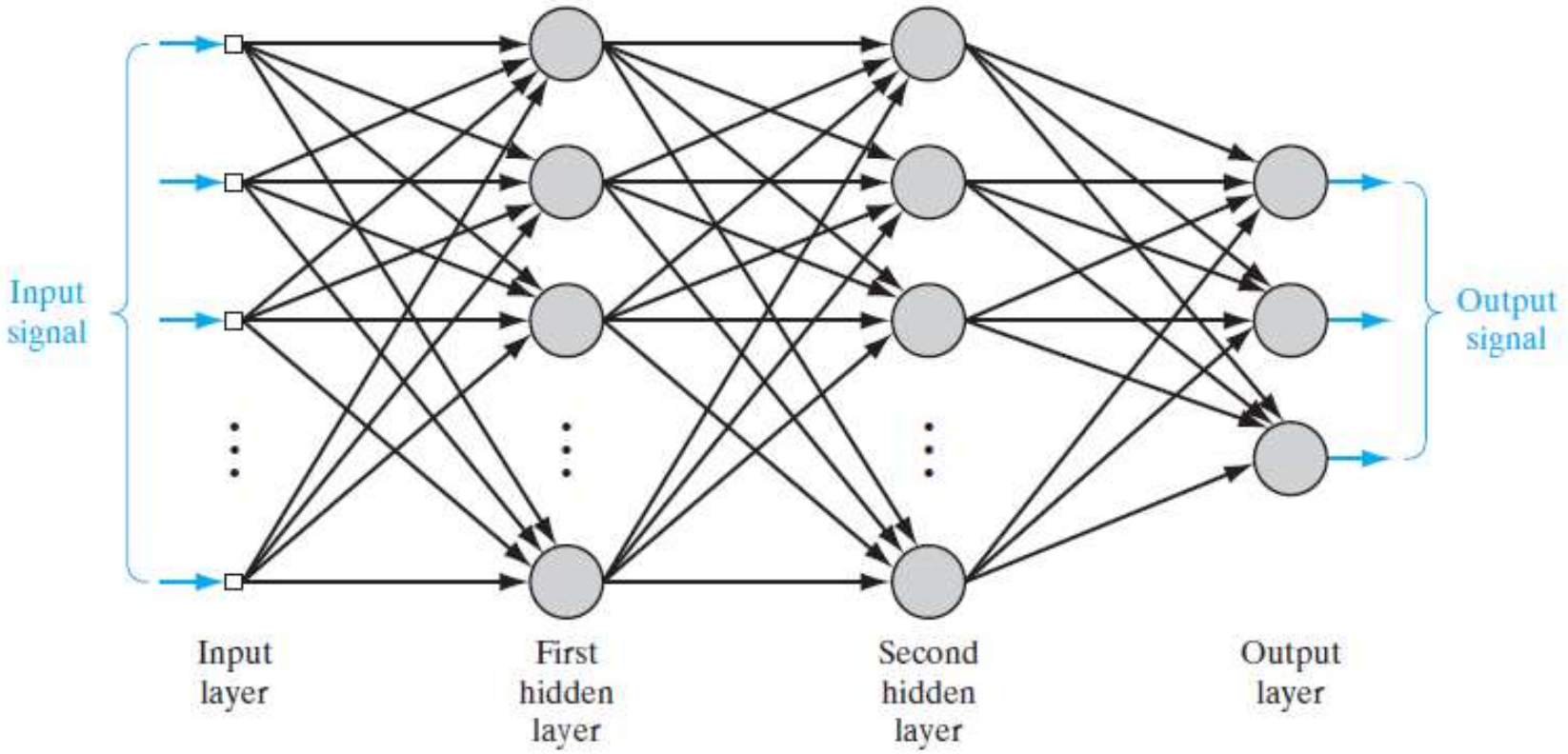
$$\text{choose } C_i \text{ if } P(C_i|\boldsymbol{x}) = \max_k P(C_k|\boldsymbol{x})$$

KON 426E
INTELLIGENT CONTROL SYSTEMS

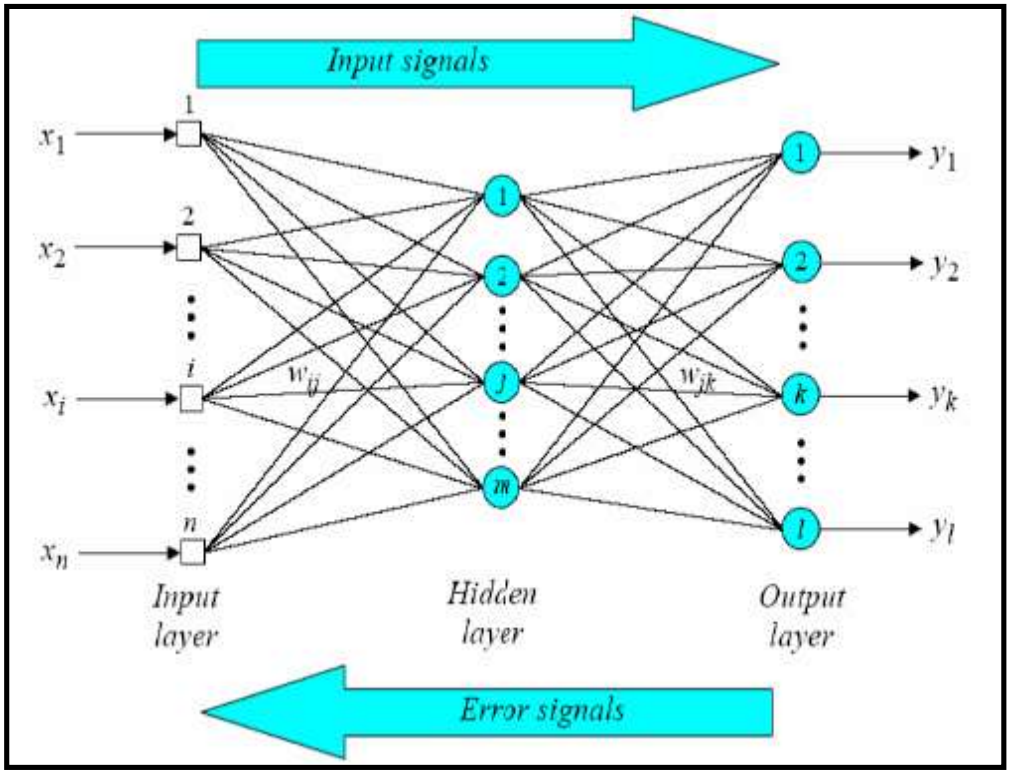
LECTURE 5

21/03/2022

MULTILAYER PERCEPTRONS



Backpropagation Algorithm



There are two phases:

Forward Phase: Inputs signals are propagated from left towards right and the output is computed.

Backward Phase: Error signals calculated at the output layer are propagated from right towards left, gradients are calculated and weights are updated.

Backpropagation Algorithm

1. In the forward phase, the synaptic weights of the network are fixed and the input signal is propagated through the network, layer by layer, until it reaches the output. Thus, in this phase, changes are confined to the activation potentials and outputs of the neurons in the network.

2. In the backward phase, an error signal is produced by comparing the output of the network with a desired response. The resulting error signal is propagated through the network, again layer by layer, but this time the propagation is performed in the backward direction. In this second phase, successive adjustments are made to the synaptic weights of the network. Calculation of the adjustments for the output layer is straightforward, but it is much more challenging for the hidden layers.

Notation

- Indices i,j,k refer to different neurons in the network, neuron j lies in a layer to the right of neuron i, neuron k lies in a layer to the right of neuron j .
- n: iteration time (discrete)
- $\mathcal{E}(n)$: cost function / instantaneous sum of error squares (error energy)
- $\mathcal{E}_{\text{av}}$: average error energy (average of $\mathcal{E}(n)$ over all training set)
- $e_j(n)$: error signal at the output of neuron j for iteration n.
- $d_j(n)$: desired response for neuron j for iteration n
- $y_j(n)$: output for neuron j for iteration n
- $w_{ji}(n)$: synaptic weight connecting the output of neuron i to the input of
- neuron j at iteration n.
- $\Delta w_{ji}(n)$: correction appled to weight $w_{ji}(n)$ at iteration n.
- $v_j(n)$: induced local field of neuron j for iteration n
- $\varphi_j(\cdot)$: activation function for neuron j for iteration n.
- b_j : bias for neuron j, $w_{j0} = b_j$ for a fixed input of +1

- $x_i(n)$: ith element of the input vector (pattern)
- $o_k(n)$: kth element of the output vector (pattern)
- η : learning rate parameter
- m_l : size (number of nodes) in layer l $\quad l = 0, 1, \ldots, L$
- L : depth of the neural network
- m_1 : the size of the first hidden layer
- m_L : the size of the output layer

- Also: $m_L = M$

Training data set:

$$\mathcal{T} = \{\mathbf{x}(n), \mathbf{d}(n)\}_{n=1}^{N}$$

The error signal produced at the output of neuron *j*:

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

The instantaneous error energy of neuron *j*:

$$\mathcal{E}_j(n) = \frac{1}{2} e_j^2(n)$$

The error-energy contributions of all the neurons in the output layer are summed up to express the total instantaneous error energy of the whole network:

$$\begin{aligned} \mathcal{E}(n) &= \sum_{j \in C} \mathcal{E}_j(n) \\ &= \frac{1}{2} \sum_{j \in C} e_j^2(n) \end{aligned}$$

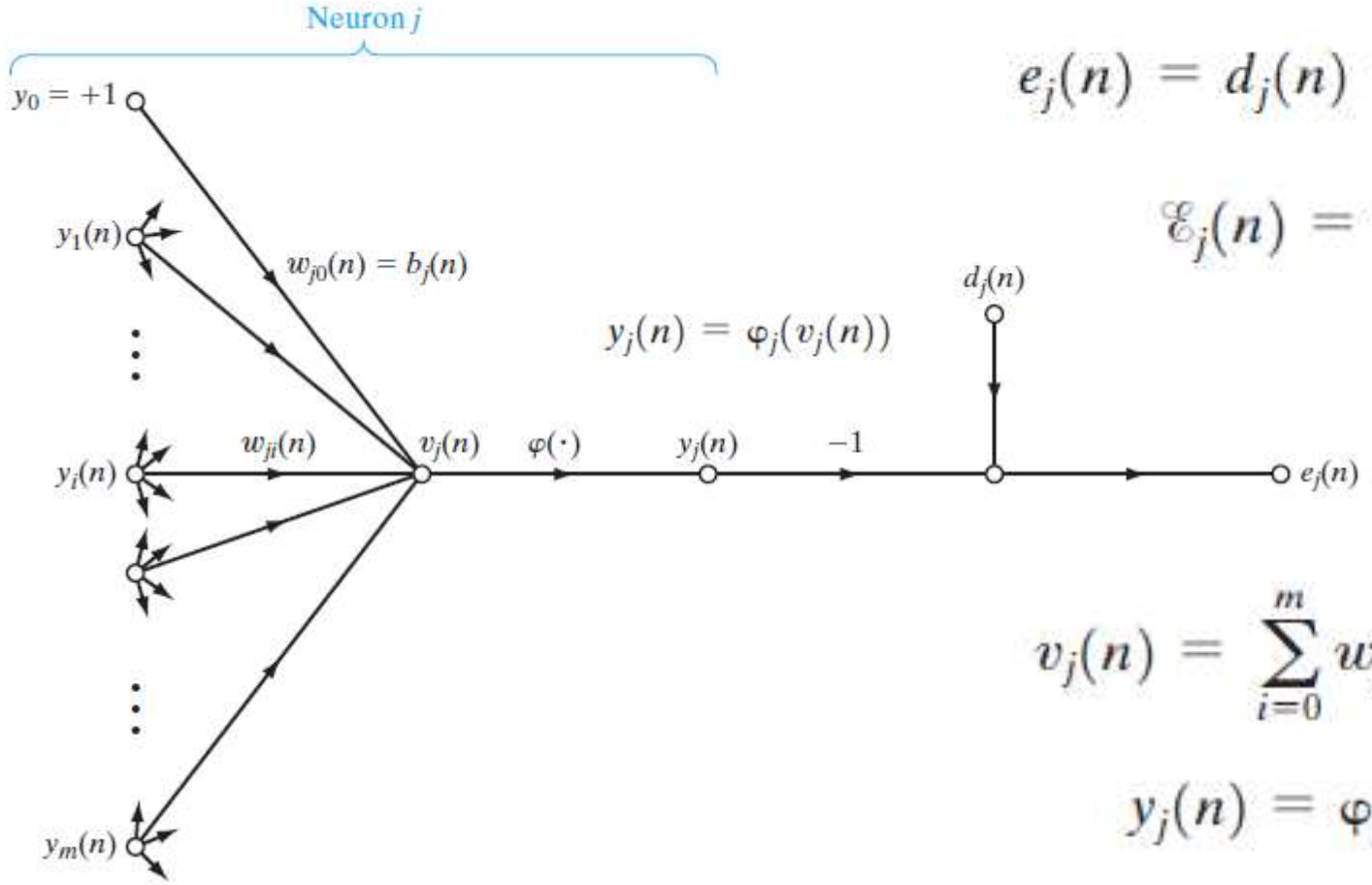
If the training sample consists of N examples, the error energy averaged over the training sample, or the empirical risk is:

$$\mathscr{E}_{\text{av}}(N) = \frac{1}{N}\sum_{n=1}^{N}\mathscr{E}(n)$$

$$= \frac{1}{2N}\sum_{n=1}^{N}\sum_{j\in C}e_j^2(n)$$

Backpropagation algorithm is applied differently at output nodes and at hidden nodes.

Derivation of the backpropagation algorithm for output nodes:



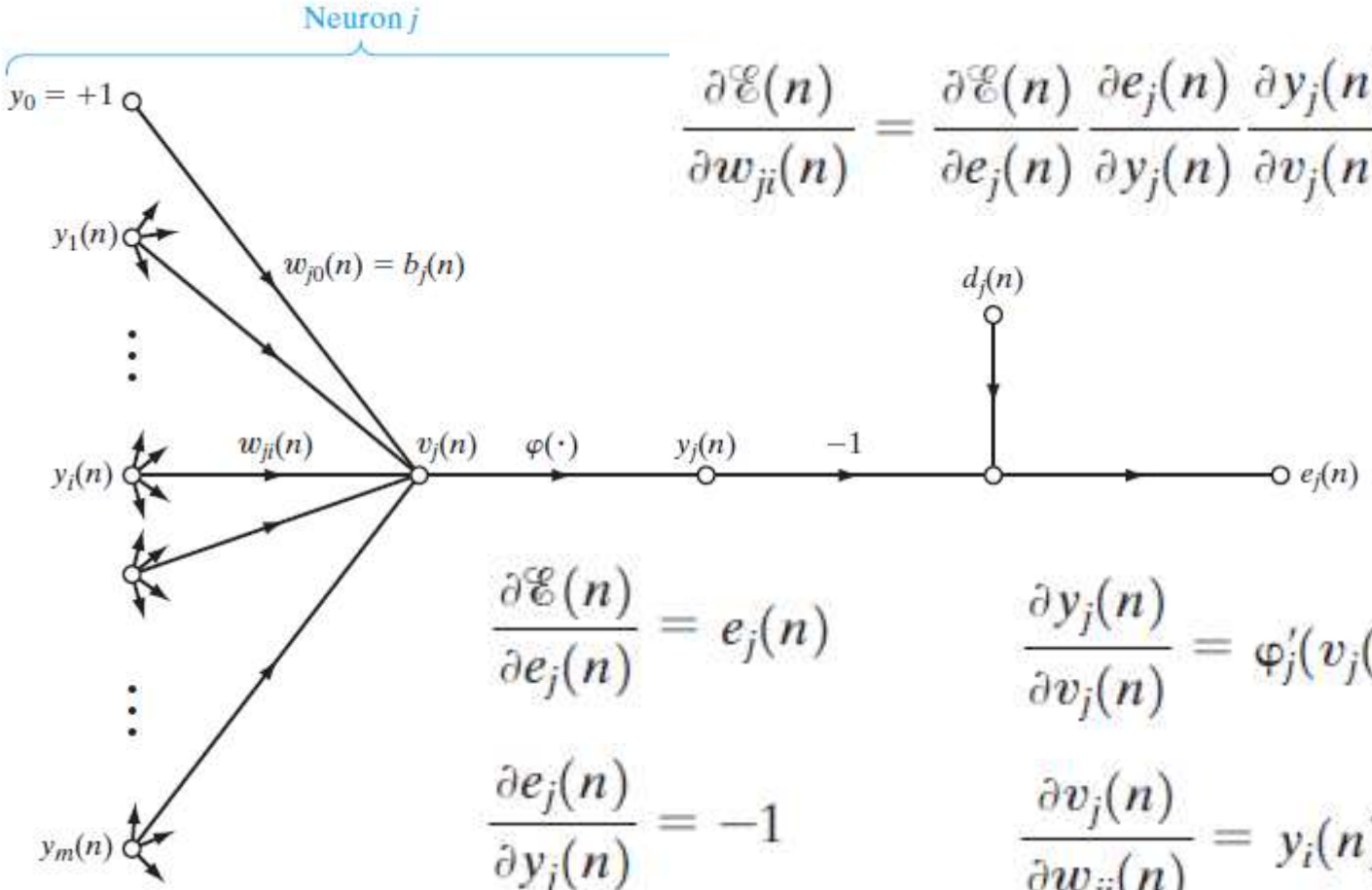
$$e_j(n) = d_j(n) - y_j(n)$$

$$\mathscr{E}_j(n) = \frac{1}{2} e_j^2(n)$$

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

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n)$$

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



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

$$\frac{\partial \mathscr{E}(n)}{\partial e_j(n)} = e_j(n) \qquad \frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n))$$

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \qquad \frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n)$$

$$\frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n)$$

$$v_j = y_1 w_{j1} + y_2 w_{j2} + y_3 w_{j3} + \cdots \ldots \ldots$$

$$\frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n)$$

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

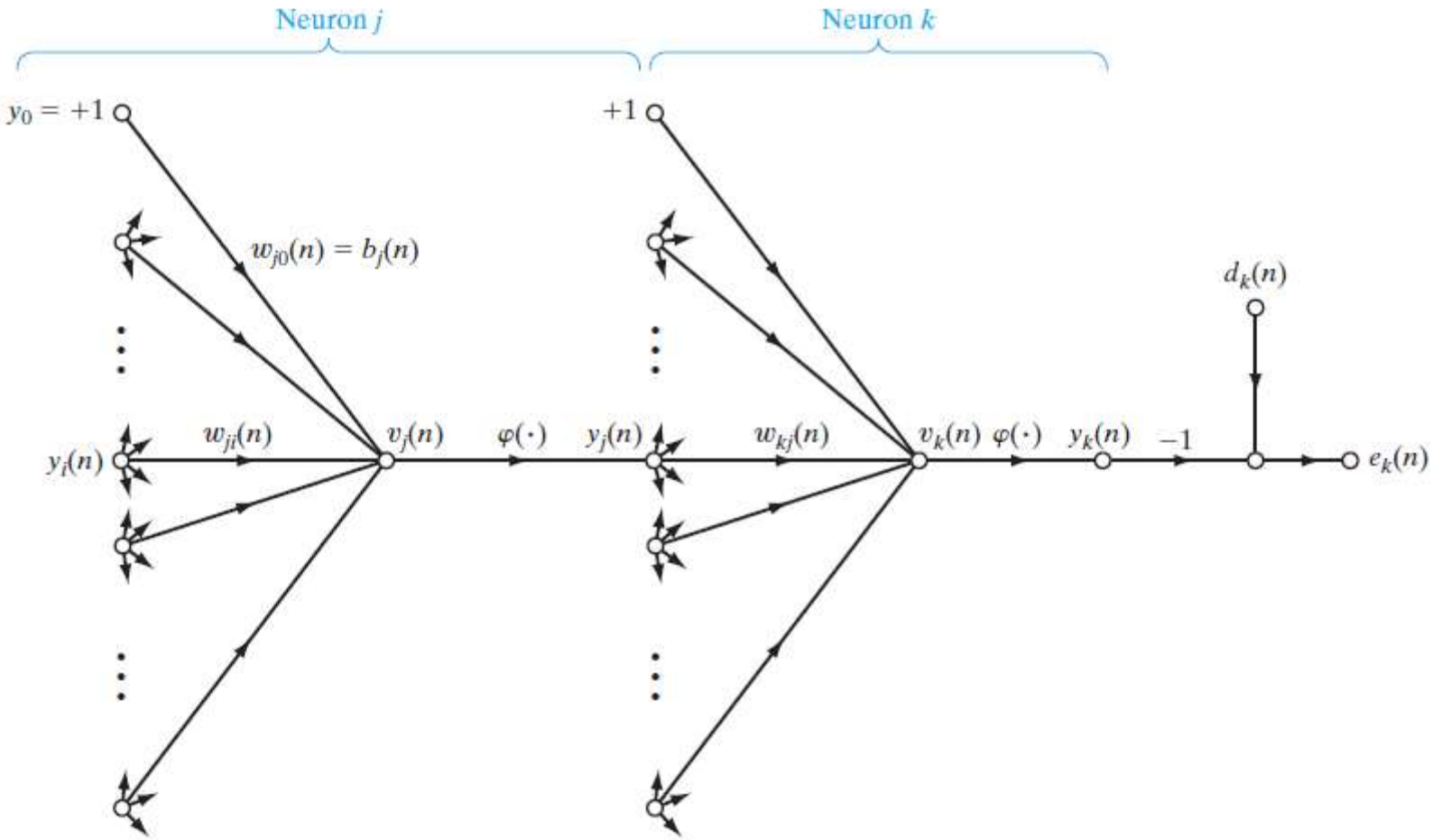
$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n)$$

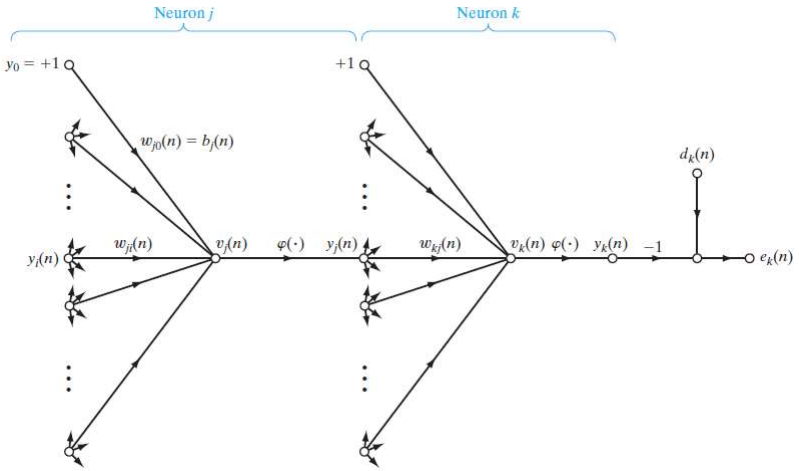
The local gradient $\delta_j(n)$ is defined by:

$$\begin{aligned}\delta_j(n) &= -\frac{\partial \mathscr{E}(n)}{\partial v_j(n)} \\ &= -\frac{\partial \mathscr{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \\ &= e_j(n)\varphi_j'(v_j(n))\end{aligned}$$

$$\frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} = \underbrace{\frac{\partial \mathscr{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)}}_{-\delta_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

Derivation of the backpropagation algorithm for hidden nodes:





neuron *j* is hidden

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n)$$

$$\delta_j(n) = -\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)}$$

$$= -\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} \varphi_j'(v_j(n))$$

$$\mathscr{E}(n) = \frac{1}{2} \sum_{k \in C} e_k^2(n)$$

neuron *k* is an output node

$$\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial y_j(n)}$$

$$\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} = \sum_k e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)}$$

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi_k'(v_k(n))$$

$$e_k(n) = d_k(n) - y_k(n)$$

$$= d_k(n) - \varphi_k(v_k(n))$$

$$v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n)$$

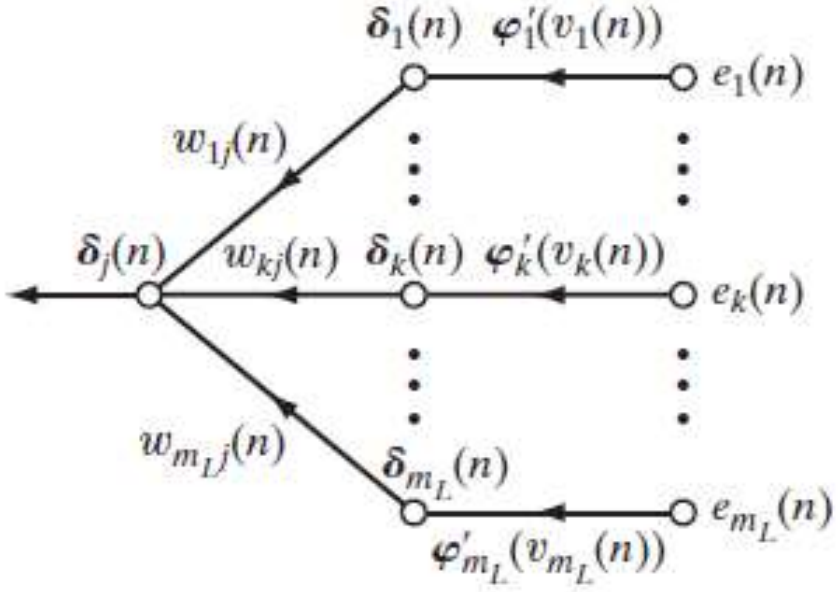
$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n)$$

$$\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} = -\sum_k e_k(n) \varphi_k'(v_k(n)) w_{kj}(n)$$

$$= -\sum_k \delta_k(n) w_{kj}(n)$$

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n)$$ neuron *j* is hidden

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n)$$



$$\begin{pmatrix} \text{Weight} \\ \text{correction} \\ \Delta w_{ji}(n) \end{pmatrix} = \begin{pmatrix} \text{learning-} \\ \text{rate parameter} \\ \eta \end{pmatrix} \times \begin{pmatrix} \text{local} \\ \text{gradient} \\ \delta_j(n) \end{pmatrix} \times \begin{pmatrix} \text{input signal} \\ \text{of neuron } j, \\ y_i(n) \end{pmatrix}$$

Modes of Training

Epoch

One complete presentation of the entire training set during the learning phase.

- **Sequential Mode (Online mode- stochastic mode)**
- **Batch Mode**

Sequential Mode (Online mode- stochastic mode)

Adjustments to the synaptic weights of the multilayer perceptron are performed on an example-by-example basis. The cost function to be minimized is therefore the total instantaneous error energy $\mathscr{E}(n)$

Consider an epoch of N training examples:

$$\{\mathbf{x}(1), \mathbf{d}(1)\}, \{\mathbf{x}(2), \mathbf{d}(2)\}, \ldots, \{\mathbf{x}(N), \mathbf{d}(N)\}$$

- First $\{\mathbf{x}(1), \mathbf{d}(1)\}$ is presented to the network, and the weight adjustments are performed using the method of gradient descent .
- Then $\{\mathbf{x}(2), \mathbf{d}(2)\}$ is presented to the network, and the weight adjustments are performed.

  ……

- This procedure is continued until the last example $\{\mathbf{x}(N), \mathbf{d}(N)\}$ is presented to the network, and the weight adjustments are performed.

Batch Learning

Adjustments to the synaptic weights of the multilayer perceptron are performed after the presentation of all the N examples in the training dataset that constitute one epoch of training.
The cost function for batch learning is defined by the average error energy

$$\mathscr{E}_{\text{av}}(N) = \frac{1}{N}\sum_{n=1}^{N}\mathscr{E}(n)$$

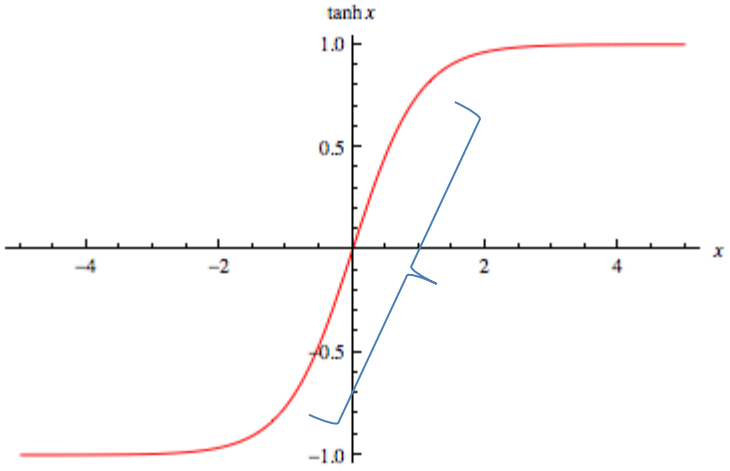
$$= \frac{1}{2N}\sum_{n=1}^{N}\sum_{j\in C}e_j^2(n)$$

Adjustments to the synaptic weights of the multilayer perceptron are made on an epoch-by-epoch basis.

SUMMARY OF THE BACK-PROPAGATION ALGORITHM

1. Initialization.

Assuming that no prior information is available, pick the synaptic weights and thresholds from a uniform distribution whose mean is zero and whose variance is chosen to make the standard deviation of the induced local fields of the neurons lie at the transition between the linear and standard parts of the activation function.



2. Presentations of Training Examples.

Present the network an epoch of training examples.
For each example in the sample, ordered in some fashion, perform the sequence of forward and backward computations described under points 3 and 4, respectively.

3. Forward Computation.

Let a training example in the epoch be denoted by $(\mathbf{x}(n), \mathbf{d}(n))$, with the input vector $\mathbf{x}(n)$ applied to the input layer of sensory nodes and the desired response vector $\mathbf{d}(n)$ presented to the output layer of computation nodes. Compute the induced local fields and function signals of the network by proceeding forward through the network, layer by layer.

The induced local field $v_j^{(l)}(n)$ for neuron j in layer l is:

$$v_j^{(l)}(n) = \sum_i w_{ji}^{(l)}(n) y_i^{(l-1)}(n)$$

$y_i^{(l-1)}(n)$ output signal of neuron i in the previous layer l - 1 at iteration n

$w_{ji}^{(l)}(n)$ synaptic weight of neuron j in layer l that is fed from neuron i in layer l - 1

For $i = 0$, $y_0^{(l-1)}(n) = +1$

$$w_{j0}^{(l)}(n) = b_j^{(l)}(n)$$

Output signal of neuron j in layer l is:

$$y_j^{(l)} = \varphi_j(v_j(n))$$

If neuron j is in the first hidden layer (l =1), set:

$$y_j^{(0)}(n) = x_j(n)$$

$x_j(n)$ is the jth element of the input vector $\mathbf{x}(n)$.

If neuron j is in the output layer ($l = L$, L is called the depth of the NN)

$$y_j^{(L)} = o_j(n)$$

Compute the error signal:

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

$d_j(n)$ is the jth element of the desired response vector $\mathbf{d}(n)$.

4. **Backward Computation**. Compute the δs (i.e., local gradients) of the network defined by

$$\delta_j^{(l)}(n) = \begin{cases} e_j^{(L)}(n)\varphi_j'(v_j^{(L)}(n)) & \text{for neuron } j \text{ in output layer } L \\ \varphi_j'(v_j^{(l)}(n)) \sum_k \delta_k^{(l+1)}(n) w_{kj}^{(l+1)}(n) & \text{for neuron } j \text{ in hidden layer } l \end{cases}$$

Adjust the synaptic weights of the network in layer l according to the generalized delta rule:

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

η Learning rate

α Momentum coefficient

5. Iteration. Iterate the forward and backward computations under points 3 and 4 by presenting new epochs of training examples to the network until the chosen stopping criterion is met.

Stopping Criteria

- Ideally, at a local or global minimum: $\vec{\mathbf{g}}(\mathbf{w}^*) \cong \vec{\mathbf{0}}$

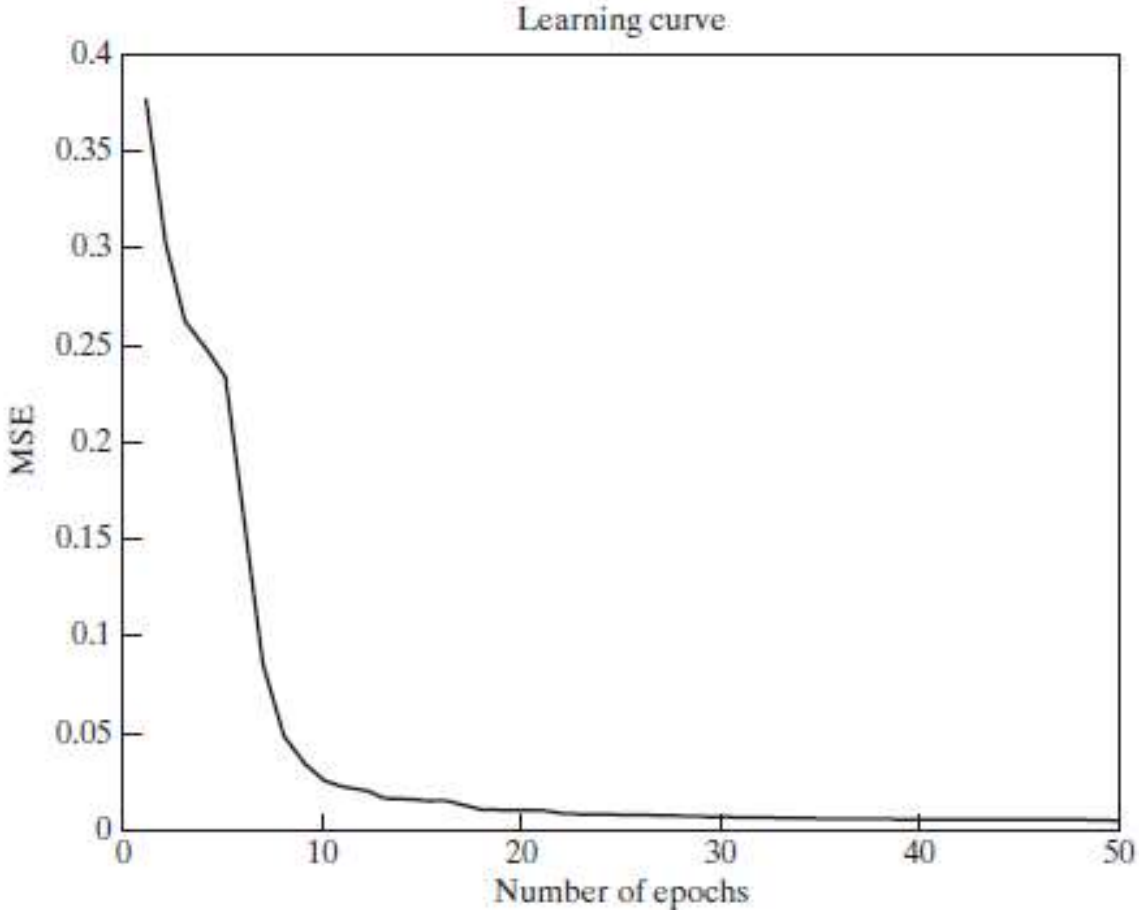
In practice:

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

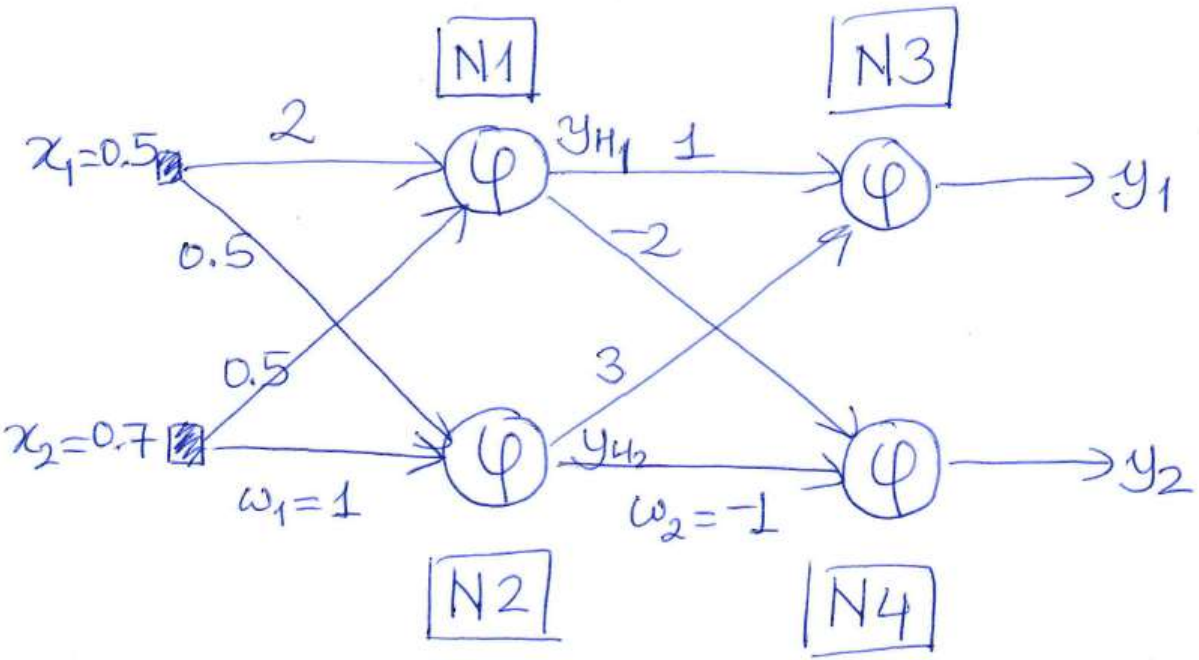
- However this criterion is not practical to use, you have to calculate all the gradients at each iteration.

- The back-propagation algorithm is considered to have converged when the **absolute rate of change in the average squared error per epoch** is sufficiently small.

A Typical Learning Curve



Example



For the MLP depicted in the figure, use backpropagation algorithm to find updated values for weights w_1 and w_2 for one iteration.

Assume $E=\frac{1}{2}\sum_{i=1}^{2} e_i^2$, $e_1=d_1-y_1$,

$e_2=d_2-y_2$, $d_1=1$, $d_2=0$, $\eta=0.3$, $\varphi(x)=\frac{1}{1+e^{-x}}$

Forward Pass

N1

$$y_{H_1} = \varphi(0.5 \times 2 + 0.7 \times 0.5) = \varphi(1.35)$$

$$y_{H_1} = \frac{1}{1 + e^{-1.35}} = 0.7941$$

N2

$$y_{H_2} = \varphi(0.5 \times 0.5 + 0.7 \times 1) = \varphi(0.95)$$

$$y_{H_2} = \frac{1}{1 + e^{-0.95}} = 0.7211$$

N3

$$y_1 = \varphi(y_{H_1} \times 1 + y_{H_2} \times 3) = \varphi(2.9574)$$

$$y_1 = \frac{1}{1 + e^{-2.9574}} = 0.9506$$

N4

$$y_2 = \varphi(y_{H_1} x_{-2} + y_{H_2} x_{-1})$$

$$= \varphi(0.7941\, x_{-2} + 0.7211\, x_{-1})$$

$$= \varphi(-2.3093)$$

$$y_2 = \frac{1}{1 + e^{+2.3093}} = 0.0904$$

$$e_1 = d_1 - y_1 = 1 - 0.9506 = 0.0494$$

$$e_2 = d_2 - y_2 = 0 - 0.0904 = -0.0904$$

Backward Pass

$$w_i(n+1) = w_i(n) + \Delta w_i(n)$$

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

$$E(n) = \frac{1}{2} e_1^2(n) + \frac{1}{2} e_2^2(n)$$

$$e_1(n) = d_1(n) - y_1(n)$$

$$e_2(n) = d_2(n) - y_2(n)$$

w_2 is in output layer.

For w_2, $\Delta w_2(n) = -\eta \frac{\partial E(n)}{\partial w_2(n)}$

$$\Delta w_2(n) = -\eta \frac{\partial E(n)}{\partial e_2(n)} \frac{\partial e_2(n)}{\partial y_2(n)} \frac{\partial y_2(n)}{\partial v_2(n)} \frac{\partial v_2(n)}{\partial w_2(n)}$$

$$\frac{\partial E(n)}{\partial e_2(n)} = \frac{1}{2} \times 2 e_2(n) = e_2(n)$$

$$\frac{\partial e_2(n)}{\partial y_2(n)} = -1$$

$$y_2(n) = \varphi_2(v_2(n))$$

$$\frac{\partial y_2(n)}{\partial v_2(n)} = \varphi_2'(n) = \left(\frac{1}{1+e^{-v_2}}\right)' = \frac{e^{-v_2}}{(1+e^{-v_2})^2}$$

$$v_2(n) = y_{H_1} \times (-2) + y_{H_2} \times \underbrace{(-1)}_{w_2}$$

$$\frac{\partial v_2(n)}{\partial w_2} = y_{H_2}$$

$$\Delta w_2(n) = -\eta \underbrace{\frac{\partial E(n)}{\partial e_2(n)}}_{e_2(n)} \underbrace{\frac{\partial e_2(n)}{\partial y_2(n)}}_{-1} \underbrace{\frac{\partial y_2(n)}{\partial v_2(n)}}_{\varphi_2'(v_2(n))} \underbrace{\frac{\partial v_2(n)}{\partial w_2(n)}}_{y_{H_2}}$$

$$= +\eta\, e_2(n)\, \varphi_2'(v_2(n))\, y_{H_2}$$

$$\Delta w_2(n) = (0.3)(-0.0904)\frac{e^{2.3093}}{(1+e^{2.3093})^2}0.7211$$

$$= 0.001607$$

$$w_2(n+1) = w_2(n) + \Delta w_2(n) = -1 + 0.001607$$

$$= -0.99839$$

$$\omega_1(n+1) = \omega_1(n) + \Delta\omega_1(n)$$

$$= \omega_1(n) + \eta\, \delta_j(n)\, y_1(n)$$

$\delta_j(n) \downarrow$ N2 $\qquad y_1(n) \downarrow x_2$

$$= \omega_1(n) + \eta\, \varphi_j'(v_j(n)) \left(\sum_k e_k \varphi_k'(v_k(n)) \omega_{kj}(n) \right) y_i(n)$$

$$\omega_1(n+1) = \omega_1(n) + \eta \frac{e^{-0.95}}{(1+e^{-0.95})^2} \cdot \left((0.0494) \frac{e^{-2.9574}}{(1+e^{-2.9574})^2} \cdot 3 + (-0.0904) \frac{e^{2.3093}}{(1+e^{2.3093})^2} \cdot (-1) \right) \cdot 0.7$$

$\varphi_2'(v_2)$ $\qquad e_1\ \varphi_3'\ \omega_{23} \qquad e_2\ \varphi_4'\ \omega_{24}$

N3 to N2 $\qquad$ N4 to N2 $\qquad \downarrow x_2$

$$\Delta w_1(n) = 0.0006276$$

$$w_1(n+1) = w_1(n) + \Delta w_1(n) = 1 + 0.0006276$$

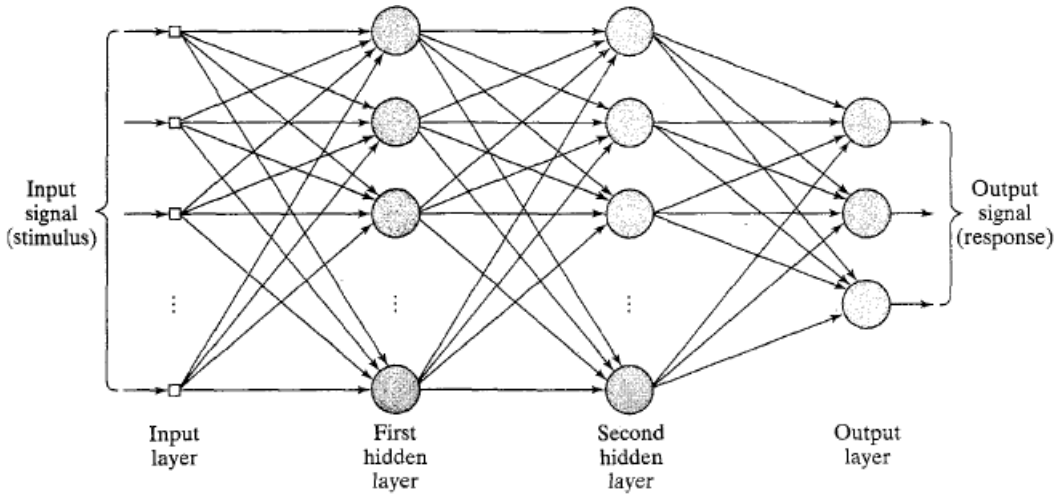
$$= 1.006276$$

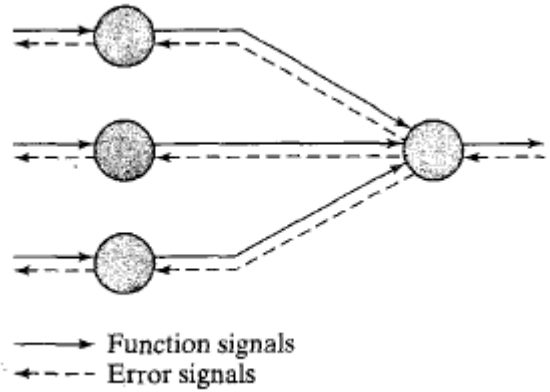
KON 426E
INTELLIGENT CONTROL SYSTEMS

LECTURE 6

28/03/2022

Training Neural Networks





$$\delta_j^{(l)}(n) = \begin{cases} e_j^{(L)}(n)\,\varphi_j'(v_j^{(L)}(n)) & \text{for neuron } j \text{ in output layer } L \\ \varphi_j'(v_j^{(l)}(n)) \sum_k \delta_k^{(l+1)}(n) w_{kj}^{(l+1)}(n) & \text{for neuron } j \text{ in hidden layer } l \end{cases}$$

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

HEURISTIC METHODS TO ENHENCE THE PERFORMANCE OF NEURAL NETWORKS

Sequential versus Batch Modes

✓ Sequential mode of learning is computationally faster then the batch mode.

Normalizing the inputs

✓Inputs should be preprocessed to have a mean value close to zero.

✓Inputs should be uncorrelated and covariences should be approximately equal so that different synaptic weights should learn approximately at the same speed.

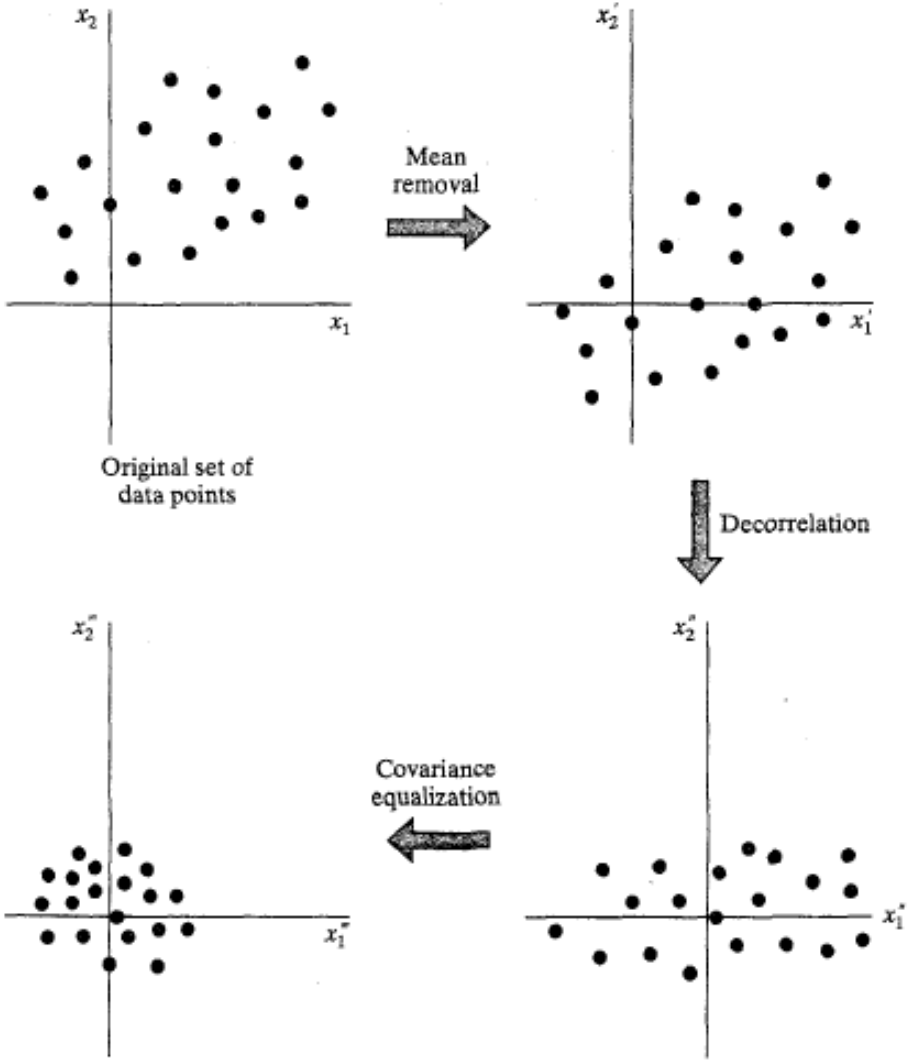


FIGURE 4.11 Illustrating the operation of mean removal, decorrelation, and covariance equalization for a two-dimensional input space.

Maximizing the Information Content

Every training example should be chosen such that the information content is largest possible for the task at hand.

✓Use an example that results in largest training error.

✓Use an example that is radically different from the previous ones, so that you can search more of the parameter space.

✓You can accomplish these if you randomize the order of the training examples at each epoch (so successive training examples will not belong to the same class)

Activation Function: $\varphi(v)$

- ✓Activation function should be differentiable.
- ✓NN will learn faster if the activation function is anti-symmetric than if it is non-symmetric.
- ✓Anti-symmetric (Odd function):

$$\varphi(-v) = -\varphi(v)$$

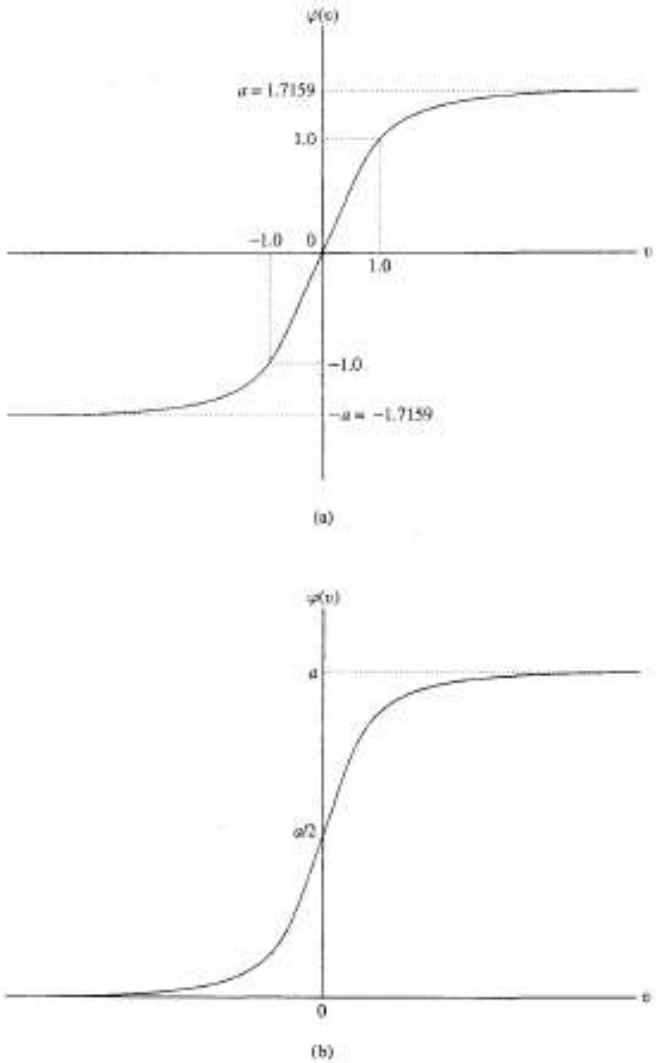


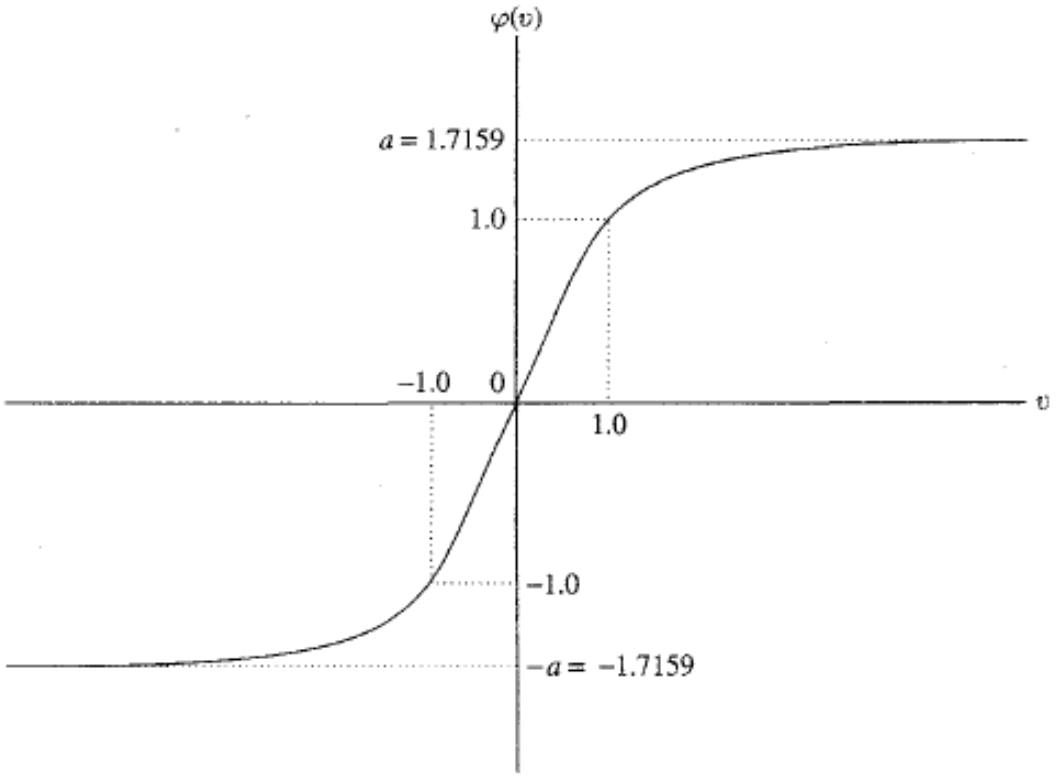
FIGURE 4.10 Antisymmetric activation function. (b) Nonsymmetric activation function.

Target Values:

- The target values (desired response) should be chosen within the range of the sigmoid activation function.
- The desired response d_j for neuron j in the output layer of the multilayer perceptron should be offset by some amount away from the limiting value of the sigmoid activation function, depending on whether the limiting value is positive or negative.
- Otherwise, the backpropagation algorithm tends to drive the free parameters of the network to infinity and thereby slow down the learning process by driving the hidden neurons into saturation.

For the limiting value a, we set: $d_j = a - \varepsilon$

and for the limiting value of -a, we set $d_j = -a + \varepsilon$



Differentiating the activation function and obtaining the local gradients

1. Logistic Function

$$\varphi_j(v_j(n)) = \frac{1}{1 + \exp(-av_j(\mathrm{n}))} \qquad a > 0 \text{ and } -\infty < v_j(n) < \infty \qquad (4.30)$$

where $v_j(n)$ is the induced local field of neuron j. According to this nonlinearity, the amplitude of the output lies inside the range $0 \leq y_j \leq 1$. Differentiating Eq. (4.30) with respect to $v_j(n)$, we get

$$\varphi_j'(v_j(n)) = \frac{a\exp(-av_j(n))}{[1 + \exp(-av_j(n))]^2} \qquad (4.31)$$

With $y_j(n) = \varphi_j(v_j(n))$, we may eliminate the exponential term $\exp(-av_j(n))$ from Eq. (4.31), and so express the derivative $\varphi_j'(v_j(n))$ as

$$\varphi_j'(v_j(n)) = ay_j(n)[1 - y_j(n)] \qquad (4.32)$$

$$\varphi_j\left(v_j(n)\right)' = -(1+e^{-av})^{-2}(-a)e^{-av} = \frac{ae^{-av}}{(1+e^{-av})^2}$$

$$y = \frac{1}{1+e^{-av}}$$

$$y + ye^{-av} = 1$$

$$ye^{-av} = 1 - y$$

$$e^{-av} = \frac{1-y}{y}$$

$$\varphi_j\left(v_j(n)\right)' = \frac{a(1-y)}{y}y^2$$

$$\varphi_j\left(v_j(n)\right)' = ay_j(n)\left[1-y_j(n)\right]$$

$$y_j(n) = o_j(n)$$

$$\delta_j(n) = e_j(n)\varphi'\left(v_j(n)\right)$$

$$\delta_j(n) = a\left[d_j(n) - o_j(n)\right]o_j(n)\left[1-o_j(n)\right]$$

For a neuron j located in the output layer, $y_j(n) = o_j(n)$. Hence, we may express the local gradient for neuron j as

$$\begin{aligned}\delta_j(n) &= e_j(n)\varphi_j'(v_j(n)) \\ &= a[d_j(n) - o_j(n)]o_j(n)[1 - o_j(n)], \qquad \text{neuron } j \text{ is an output node}\end{aligned} \tag{4.33}$$

where $o_j(n)$ is the function signal at the output of neuron j, and $d_j(n)$ is the desired response for it. On the other hand, for an arbitrary hidden neuron j, we may express the local gradient as

$$\begin{aligned}\delta_j(n) &= \varphi_j'(v_j(n)) \sum_k \delta_k(n)w_{kj}(n) \\ &= ay_j(n)[1 - y_j(n)] \sum_k \delta_k(n)w_{kj}(n), \qquad \text{neuron } j \text{ is hidden}\end{aligned} \tag{4.34}$$

✓The aim of this procedure is to avoid explicit computation of the activation function. Just compute the output of the NN.

2. Hyperbolic Tangent Function

$$\varphi_j(v_j(n)) = a \tanh(bv_j(n))$$

a and b are positive constants.

$$\begin{aligned}\varphi_j'(v_j(n)) &= ab \operatorname{sech}^2(bv_j(n)) \\ &= ab(1 - \tanh^2(bv_j(n))) \\ &= \frac{b}{a}[a - y_j(n)][a + y_j(n)]\end{aligned}$$

For a neuron j located in the output layer, the local gradient is

$$\begin{aligned}\delta_j(n) &= e_j(n)\varphi_j'(v_j(n)) \\ &= \frac{b}{a}[d_j(n) - o_j(n)][a - o_j(n)][a + o_j(n)]\end{aligned}$$

For a neuron *j* in a hidden layer:

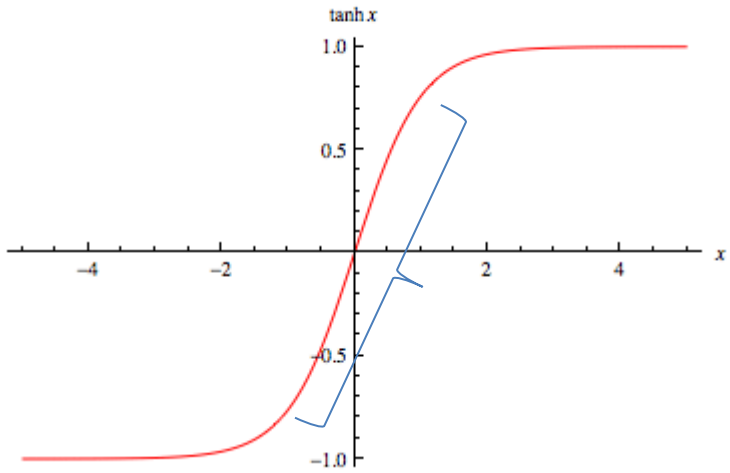
$$\begin{aligned}\delta_j(n) &= \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n) \\ &= \frac{b}{a}[a - y_j(n)][a + y_j(n)] \sum_k \delta_k(n) w_{kj}(n)\end{aligned}$$

Initialization

✓ When weights are initialized with large values, they are driven to saturation.

✓ When weights are initialized with small values, NN operates on a flat area around the origin of the error surface.

✓ Weights should be selected from a uniform distribution with a mean of zero and standard deviation of the induced local fields ($v = \sum_{i=0}^{m} x_i w_i$) should lie in the transition area between the linear and saturated parts of its activation function.



Rate of Learning

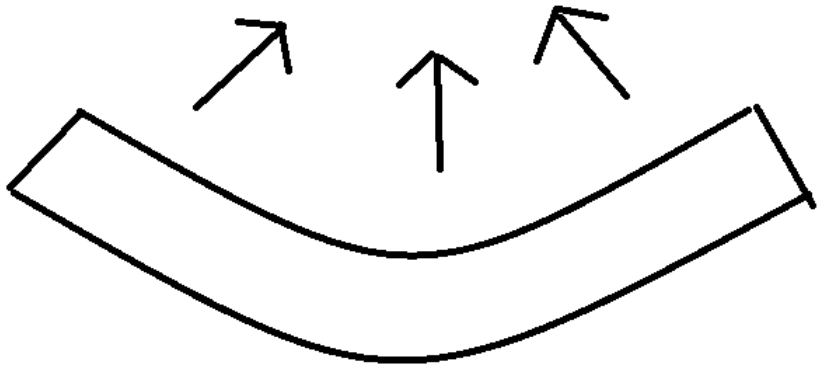
Momentum and learning rate parameters are typically adjusted (and usually decreased) as the number of training iterations increases.

- Every adjustable network parameter (weight) should have its own individual learning rate parameter.

- Every learning rate parameter should be allowed to vary from one iteration to the next.

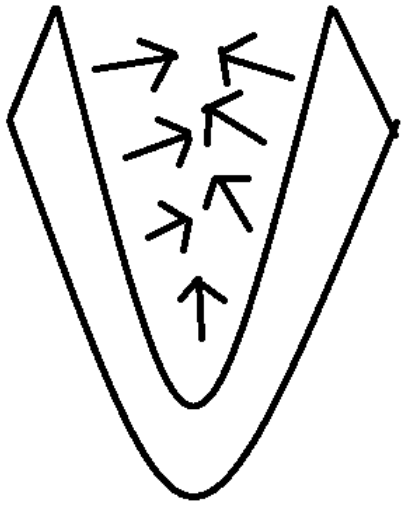
- When the derivative of the cost function with respect to a synaptic weight $\frac{\partial E}{\partial w}$ has the same algebraic sign for several consecutive iterations, learning rate for that weight should be increased.

In this case the graph of E(n) will be flat, this means that NN is operation in a flat part of the error surface and can speed up.



- When the algebraic sign of the derivative of the cost function with respect to a particular synaptic weight alternates for several consecutive iterations, the learning rate for that weight should be decreased.

This means that the graph of the cost function has peaks and valleys (not flat) and in order not to miss the minimum point we should slow down.



- All neurons should learn at the same rate. Neurons at the last layers generally have larger local gradients, so they should be assigned smaller learning rates.

Neurons with larger number of inputs should also be assigned smaller learning rates.

A Simple Heuristic Rule to Adapt Learning Rate

$$\Delta\eta = \begin{cases} +\gamma, & \text{if } \Delta E < 0 \\ -\beta\eta, & \text{if } \Delta E > 0 \\ 0, & \text{otherwise} \end{cases}$$

If $\Delta E < 0$ for several steps, update direction is fine, learning rate may be increased to speed convergence.

If $\Delta E > 0$ for several steps, then decrease the learning rate geometrically to allow rapid decay.

The effect of the momentum term

Generalized delta rule:

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

α : momentum constant (small positive constant)
(generally between 0.5 and 1)

η : learning rate

Now, let us do the following derivation:

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

$$\Delta w_{ji}(0) = \eta \delta_j(0) y_i(0)$$

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

$$\Delta w_{ji}(2) = \alpha \Delta w_{ji}(1) + \eta \delta_j(2) y_i(2)$$

$$\Delta w_{ji}(2) = \alpha\big(\alpha\, \eta \delta_j(0) y_i(0) + \eta \delta_j(1) y_i(1)\big) + \eta \delta_j(2) y_i(2)$$

$$\Delta w_{ji}(2) = \alpha^2\, \eta \delta_j(0) y_i(0) + \alpha \eta \delta_j(1) y_i(1) + \eta \delta_j(2) y_i(2)$$

If you continue with the derivation of $\Delta w_{ji}(3)$, $\Delta w_{ji}(4)$, etc. you arrive at:

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

Since, $\delta_i(n) y_i(n) = -\dfrac{\partial E(n)}{\partial w_{ji}(n)}$, we finally have:

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

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

- This is an exponentially weighted time series, to be convergent, $0 \leq |\alpha| < 1$
- When $\frac{\partial E(n)}{\partial w_{ji}(n)}$ has the same algebraic sign in consecutive iterations, $\Delta w_{ji}(n)$ grows in magnitude and $w_{ji}(n)$ is adjusted by a large amount. **Momentum accelerates descent.**
- When $\frac{\partial E(n)}{\partial w_{ji}(n)}$ has opposite signs in consecutive iterations , $\Delta w_{ji}(n)$ shrinks in magnitude, so $w_{ji}(n)$ is adjusted by a small amount. **Momentum term has a stabilizing effect.**

Compare these result s with rate of learning rules in slides 18,19.

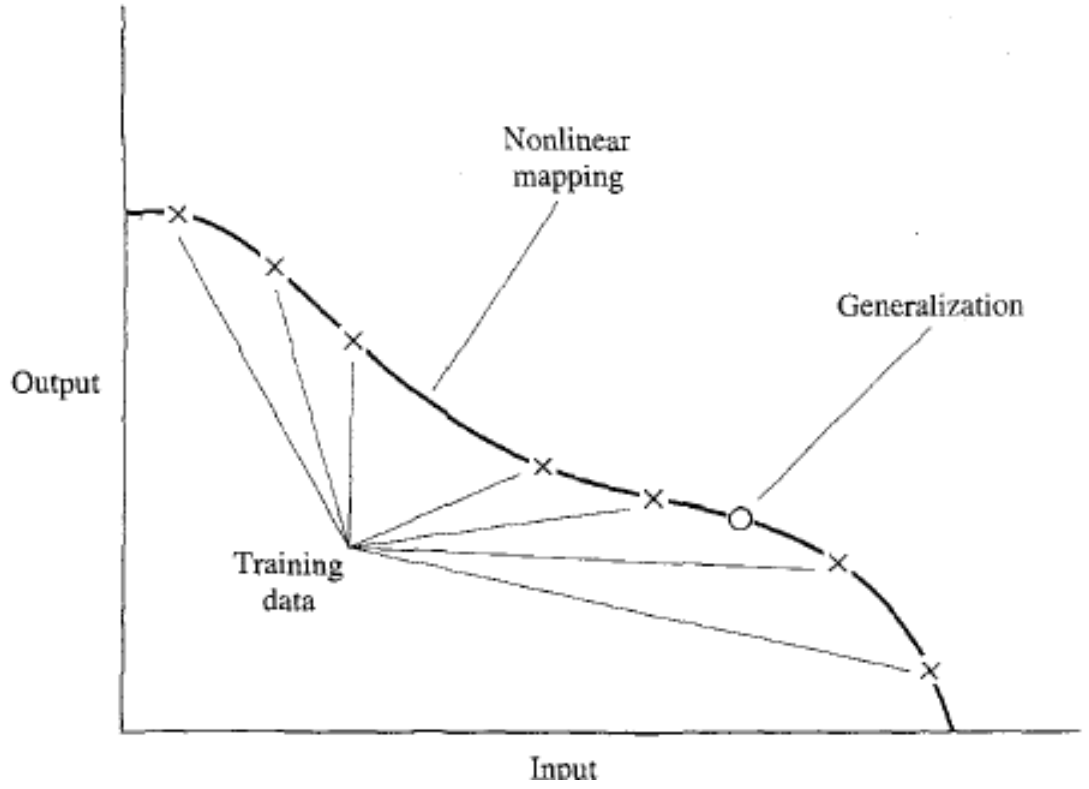
Training Data

- The training set should contain examples from all of the input space which you want the NN to learn.
- The NN will not learn the parts of the input space from which it has not seen any examples.
- Generally two thirds (2/3) of the data is selected as training data and one thirds (1/3) is selected as the test data.
- The training and test data should be selected from the same data set. (E.g. data collected from the same experiment)

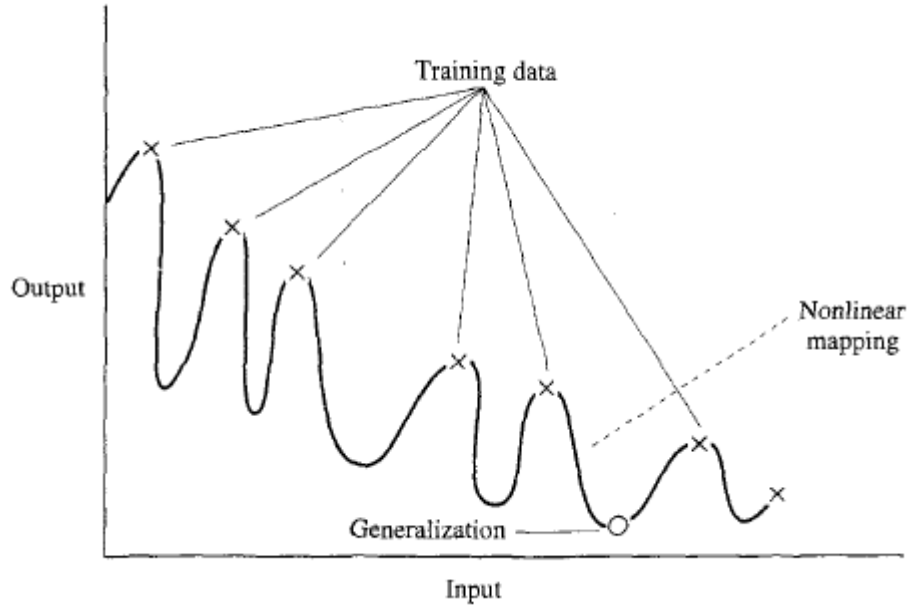
Generalization

A network is said to **generalize well** when the input-output mapping computed by the network is **correct for test data** never used in creating or training the network. (Assuming that the test data is drawn from the same population used to generate the training data.)

- Learning process can be viewed as a “curve fitting” problem.
- It is actually a nonlinear input-output mapping.
- Or good nonlinear interpolation of data.



Properly fitted data (good generalization) :
Correct input-output mapping even if the input is slightly different from the examples used to train the network.



Overfitted data: When NN learns too many input-output examples or undergoes too many training epochs, NN may end up memorizing the training data. It may find a feature (for example due to noise) that is present in the training data but not true of the underlying function that is to be learned.

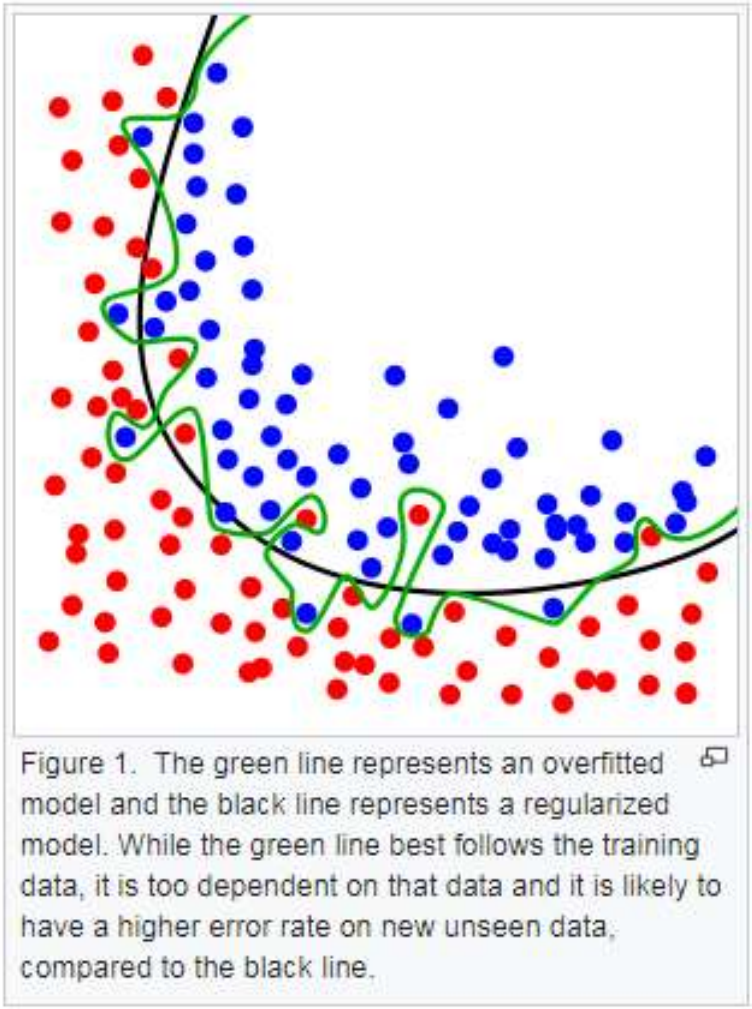


Figure 1. The green line represents an overfitted model and the black line represents a regularized model. While the green line best follows the training data, it is too dependent on that data and it is likely to have a higher error rate on new unseen data, compared to the black line.

Overtraining occurs when:

- The number of input-output examples in the training set is too much.
- When the training is continued too long.
- When the number of adaptable parameters is too much. (number of weights)

The opposite of overtraining is **undertraining** which occurs when a statistical model or machine learning algorithm cannot adequately capture the underlying structure of the data

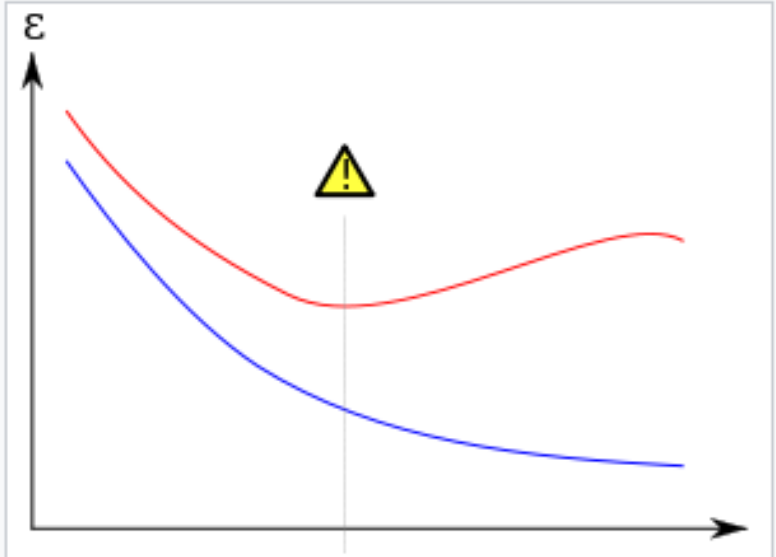


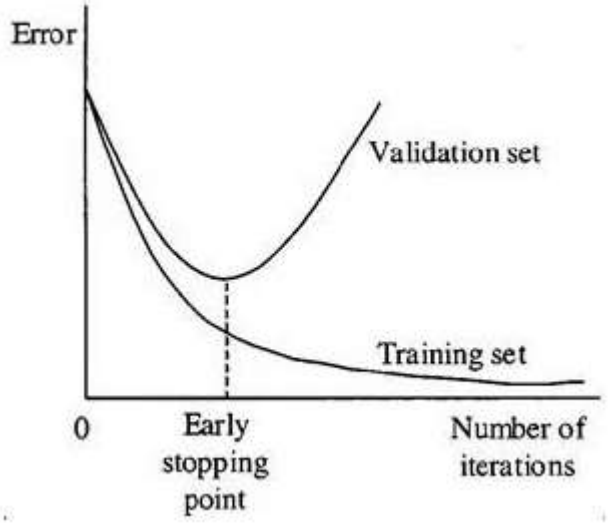
Figure 4. Overfitting/overtraining in supervised learning (e.g., neural network). Training error is shown in blue, validation error in red, both as a function of the number of training cycles. If the validation error increases(positive slope) while the training error steadily decreases(negative slope) then a situation of overfitting may have occurred. The best predictive and fitted model would be where the validation error has its global minimum.

To avoid **overfitting (memorization)**:
Divide the training data into two sets:

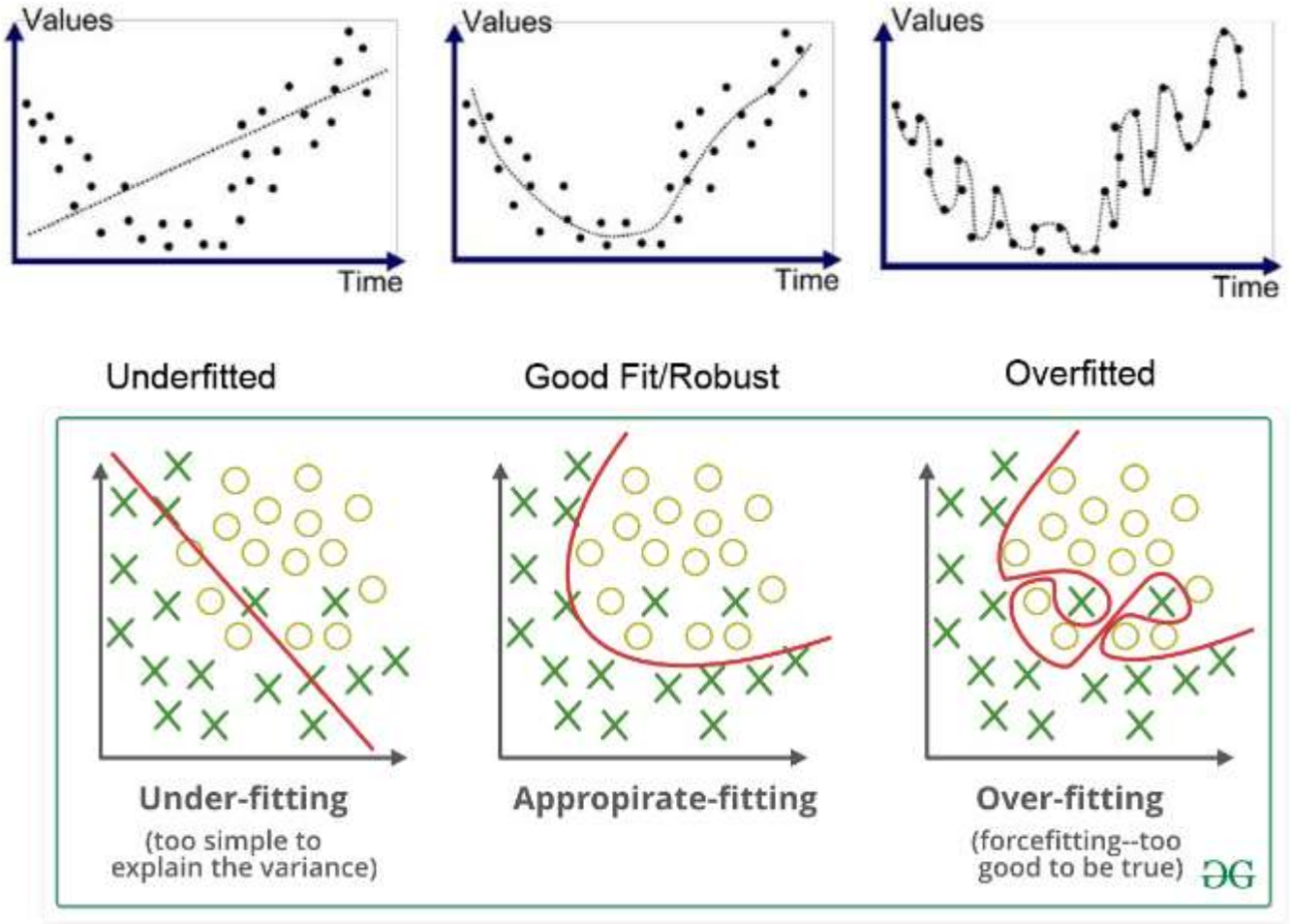
1) One for training
2) Validation set

Keep in mind that the training error will always **decrease** whatsoever.

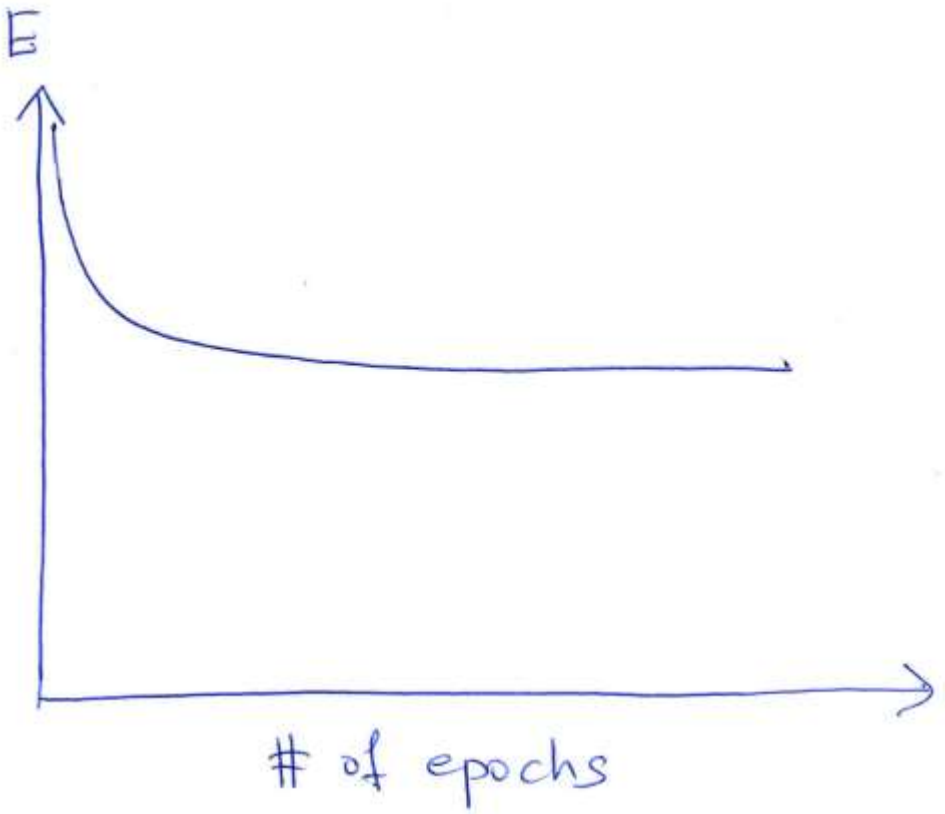
You have to stop training WHEN the validation error starts to **increase**.



Examples of overfitting and underfitting



To avoid underfitting you must increase the number of adaptable parameters (the number of weights and therefore the number of nodes/layers)



If the cost function does not decrease, this suggests **underfitting.**
Increase the number of parameters (hidden neurons-hidden layers)

Applications of Neural Networks

Function Approximation

Consider a nonlinear input-output mapping described by $\vec{\mathbf{d}}=\vec{\mathbf{f}}(\vec{\mathbf{x}})$

$\vec{\mathbf{x}}$: input vector

$\vec{\mathbf{d}}$: output vector

$\vec{\mathbf{f}}(.)$: vector-valued function which is not known

We are given a set of labeled examples:

$$\{(\vec{x}_i, \vec{d}_i)\}_{i=1}^{N}$$

Examples

$$f(x_1,x_2) = x_1^2 + x_2^3$$

$$f(x_1,x_2) = x_1 \sin x_2$$

$$f(x_1,x_2,x_3) = x_1(\sin x_2)^2 + x_3^4$$

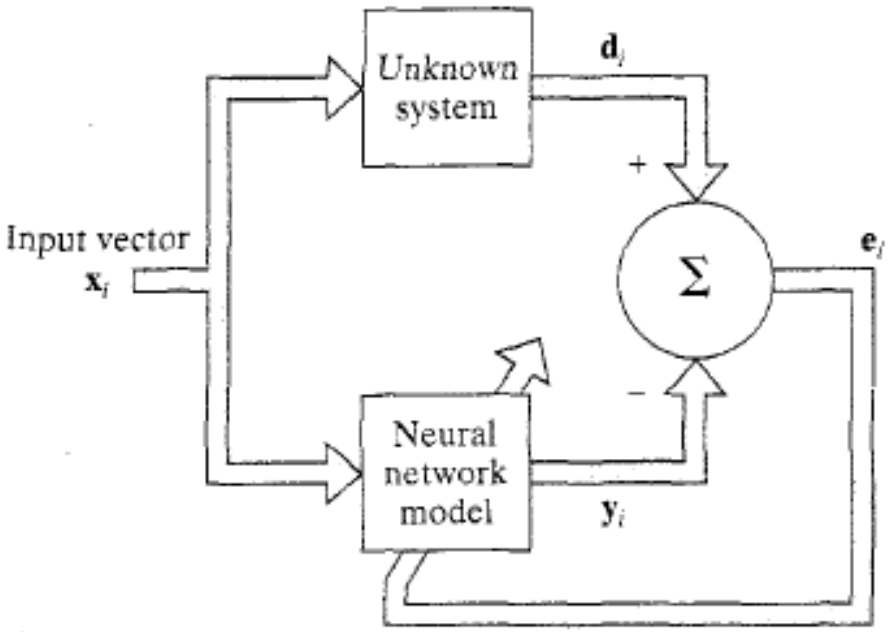
The requirement is to design a NN that approximates $\vec{\mathbf{f}}(.)$ such that function $\vec{\mathbf{F}}(.)$ describing the input-output mapping actually realized by the NN is close enough to $\vec{\mathbf{f}}(.)$ in a Euclidean sence over all inputs (ε is a small positive number):

$$\left\|\vec{\mathbf{F}}(\vec{x}) - \vec{\mathbf{f}}(\vec{x})\right\| < \varepsilon \quad \text{for all} \quad \vec{x}$$

Provided that ***N* is large enough** and the network has an **adequate number of free parameters (weights)**, ε can be made small enough for the task.

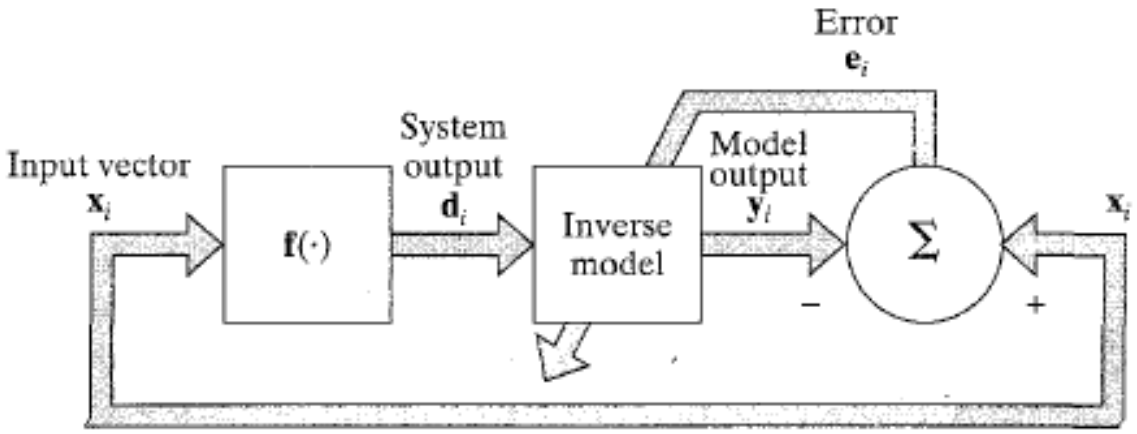
This may sound too theoretical but a common engineering application of it is **system identification**.

System Identification



Initially since the NN model starts with random weights the output will be very different from the desired response, so error will be high. As iterations continue NN weights will be updated with the error information and finally NN will behave similar to the unknown system. (Supervised learning setting)

Inverse System Modelling



The inverse system is described by: $\mathbf{x} = \mathbf{f}^{-1}(\mathbf{d})$

The vector valued functior $\mathbf{f}^{-1}(\cdot)$ is the inverse of $\mathbf{f}(\cdot)$

(Be careful : It is not the reciprocal)

Nonlinear Prediction

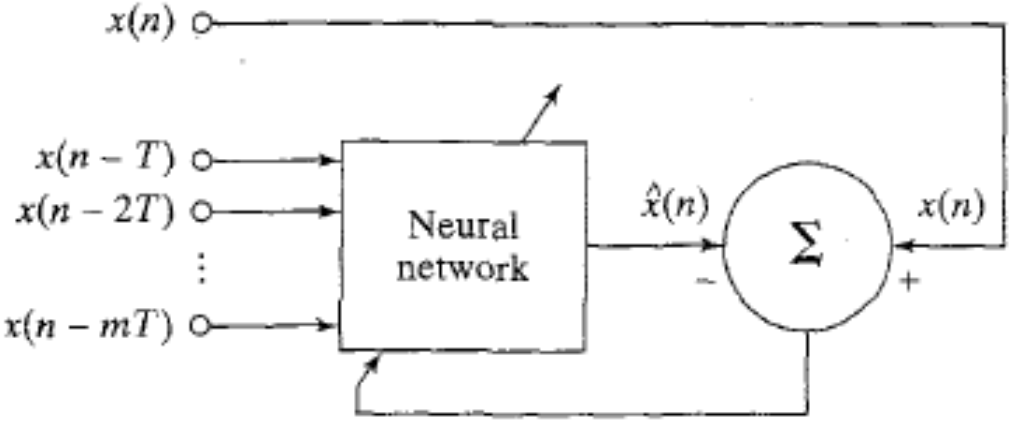


FIGURE 2.15 Block diagram of nonlinear prediction.

Nonlinear Systems

In this course we are mainly involved with the control of nonlinear systems. Linear systems can be adequately controlled with classical approaches.

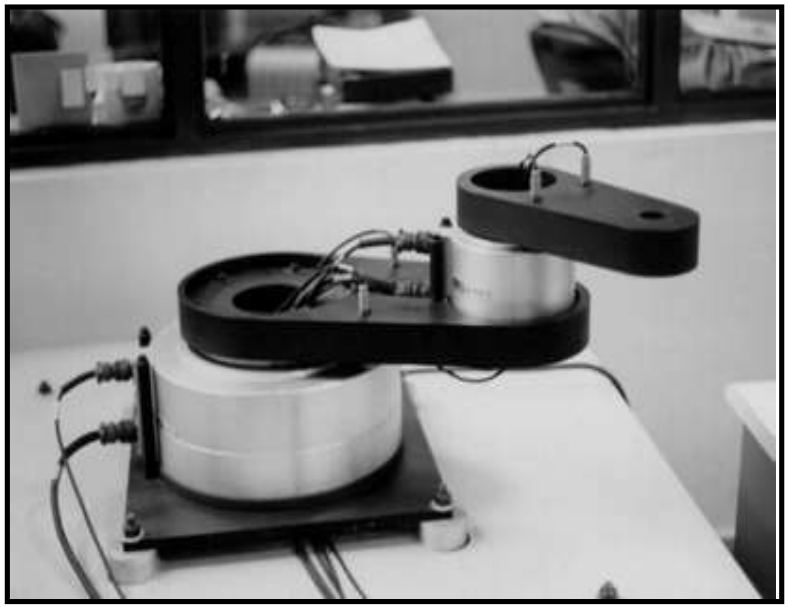
State equations for a linear system

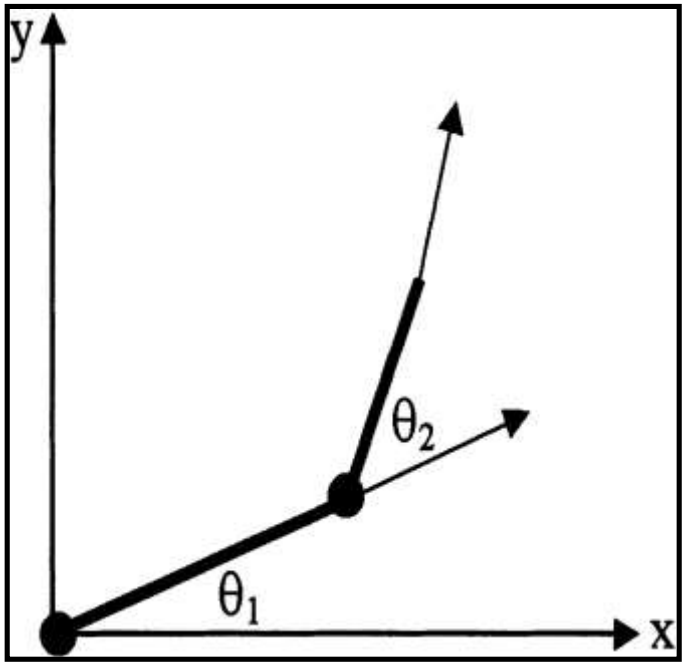
$$\dot{\vec{x}}(t) = A\vec{x}(t) + B\vec{u}(t)$$
$$\vec{y}(t) = C\vec{x}(t) + D\vec{u}(t)$$
$$\vec{u}(t) = K\vec{x}(t)$$

State equations for a nonlinear system

$$\dot{\vec{x}}(t) = \vec{f}\left(\vec{x}(t), \vec{u}(t)\right)$$
$$\vec{y}(t) = \vec{h}\left(\vec{x}(t)\right)$$
$$\vec{u}(t) = \vec{\Phi}(t)$$

Two-link robotic manipulator



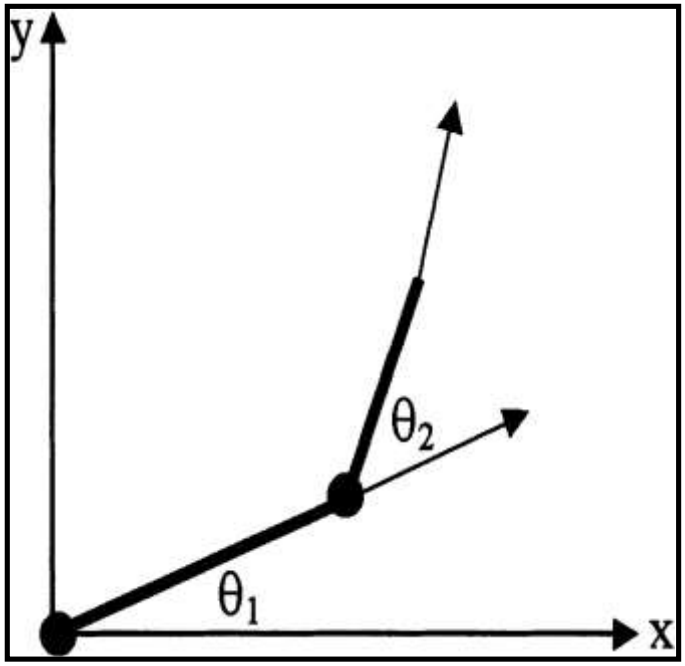


Rotational robot manipulator with 2- degrees of feedom

Parameters of the manipulator:

Motor 1 rotor inertia	0:267	I1
Arm 1 inertia	0:334	I2
Motor 2 rotor inertia	0:0075	I3
Motor 2 stator inertia	0:040	I3C
Arm 2 inertia	0:063	I4
Payload inertia	0:000	IP
Motor 1 mass	73:0	M1
Arm 1 mass 9:78	M2	
Motor 2 mass	14:0	M3
Arm 2 mass 4:45	M4	
Payload mass	0:00	Mp
Arm 1 length	0:359	L1
Arm 2 length	0:24	L2
Arm 1 CG	0:136	L3
Arm 2 CG	0:102	L4
Axis 1 friction	5:3	F1
Axis 2 friction	1:1	F2
Torque limit 1	245:0	
Torque limit 2	39:2	

Two-link robotic manipulator



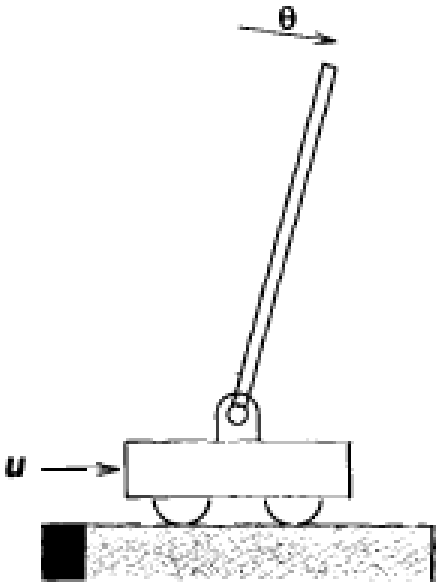
$$M(\theta)\theta'' + V(\theta,\theta') = \tau(t) - f$$

$$M(\theta) = \begin{bmatrix} p_1 + 2p_3\cos(\theta_2) & p_2 + p_3\cos(\theta_2) \\ p_2 + p_3\cos(\theta_2) & p_2 \end{bmatrix}$$

$$V(\theta,\theta') = \begin{bmatrix} -\theta_2'(2\theta_1' + \theta_2')p_3\sin(\theta_2) \\ \theta_1'^2 p_3\sin(\theta_2) \end{bmatrix}$$

$$p_1 = 2.0857, \quad p_2 = 0.1168, \quad p_3 = 0.1630$$

An Inverted Pendulum System



$$\ddot{\theta} = \frac{g \sin\theta + \cos\theta \left(\dfrac{-u - ml\dot{\theta}^2 \sin\theta}{m_c + m} \right)}{l \left(\dfrac{4}{3} - \dfrac{m \cos^2\theta}{m_c + m} \right)},$$

$$\ddot{z} = \frac{u + ml(\dot{\theta}^2 \sin\theta - \ddot{\theta} \cos\theta)}{m_c + m},$$

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \mathbf{f}(\mathbf{x},u) = \begin{bmatrix} x_2 \\ \dfrac{g \sin x_1 + \cos x_1 \left(\dfrac{-u - mlx_2^2 \sin x_1}{m_c + m} \right)}{l \left(\dfrac{4}{3} - \dfrac{m \cos^2 x_1}{m_c + m} \right)} \\ x_4 \\ \dfrac{u + ml(x_2^2 \sin x_1 - \dot{x}_2 \cos x_1)}{m_c + m} \end{bmatrix}.$$