

بسم الله الرحمن الرحيم.



# Neural networks

Isfahan University of Technology

Marzieh Kamali

# Aims and Learning Outcomes

- Describe the relation between real brains and simple artificial neural network models.
  - Introduce the main fundamental principles and techniques of neural network systems.
  - Investigate the principal neural network models and applications.
  - Explain the most common architectures and learning algorithms for Multi-Layer Perceptrons, Radial-Basis Function Networks, and Self-Organising Maps.
  - Discuss the main factors involved in achieving good learning and generalization.

# Outline

- Introduction
- Neuron Model and Network Architectures
- Linear algebra review
- Perceptron Learning rule
- Supervised Hebbian Learning
- Performance Surfaces and Optimum Points
- Performance Optimization
- Widrow-Hoff Learning
- Backpropagation
- Variations on Backpropagation

- Generalization
- Associative Learning
- Competitive Networks
- Radial Basis Networks

# Grading Criteria

- Midterm Exam and Quizzes  $\approx 40\%$
- Home Works, Comp. Assignments  $\approx 30\%$
- Final exam  $\approx 30\%$
- **Course Website:**
- <http://yekta.iut.ac.ir>
  
- Email: [m.kamali@iut.ac.ir](mailto:m.kamali@iut.ac.ir)
- T.A: Miss Niyazmand, [t.niyazmand@ec.iut.ac.ir](mailto:t.niyazmand@ec.iut.ac.ir)
- Telegram group Link: <https://t.me/+2oQ5K9TxGqNjYWZk>
  - جلسات آنلاین درس روزهای یکشنبه ساعت ۱۵:۳۰-۱۳:۳۰ و جلسات حل تمرین روزهای سه شنبه ساعت ۱۵:۳۰-۱۳:۳۰ برگزار می‌گردد.

# Textbooks

- Martin T. Hagan, Howard B. Demuth, Mark Beale, Orlando De Jesús, *Neural Network Design*. 2014.  
(<http://hagan.okstate.edu/nnd.html>)
- Simon Haykin, *Neural Networks and Learning Machines*, 3<sup>rd</sup> Edition, 2009.
- Christopher M. Bishop, *Neural Networks for Pattern Recognition*, 1995.
- Michael Nielsen, *Neural Networks and Deep Learning*, 2017.  
(Online).
- Laurene Fausett, **Fundamentals of Neural networks**, 1994.

# Introduction

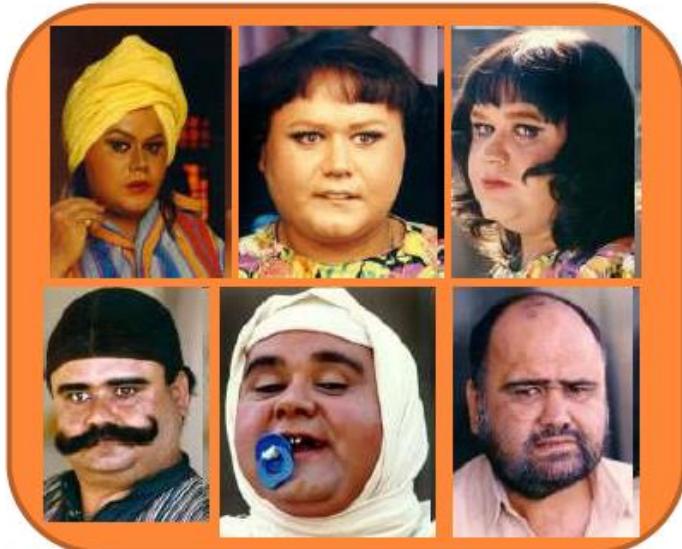
# Introduction to Neural Networks and Their History

1. What are Neural Networks?
2. Biological Neurons and Neural Networks
2. Why Artificial Neural Networks?
3. A Brief History of the Field
5. Some Current Artificial Neural Network Applications

# What are Neural Networks?

1. **Neural Networks** (NNs) are networks of neurons, for example, as found in real (i.e. biological) brains.
2. **Artificial Neurons** are crude approximations of the neurons found in brains. They may be physical devices, or purely mathematical constructs.
3. **Artificial Neural Networks** (ANNs) are networks of Artificial Neurons, and hence constitute crude approximations to parts of real brains. They may be physical devices, or simulated on conventional computers.
4. From a practical point of view, an ANN is just a parallel computational system consisting of many simple processing elements connected together in a specific way in order to perform a particular task.
5. One should never lose sight of how crude the approximations are, and how over-simplified our ANNs are compared to real brains.

# شبکه عصبی؟



○ کارهایی که مغز انسان انجام می‌دهد

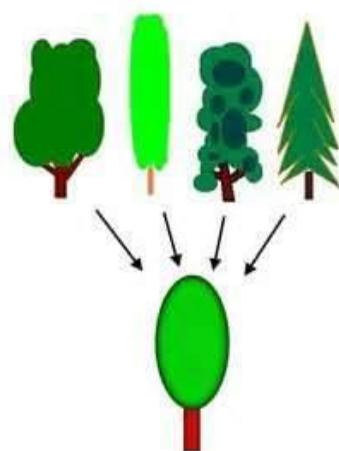
- یادگیری (تشخیص چهره)

- ذخیره‌سازی اطلاعات

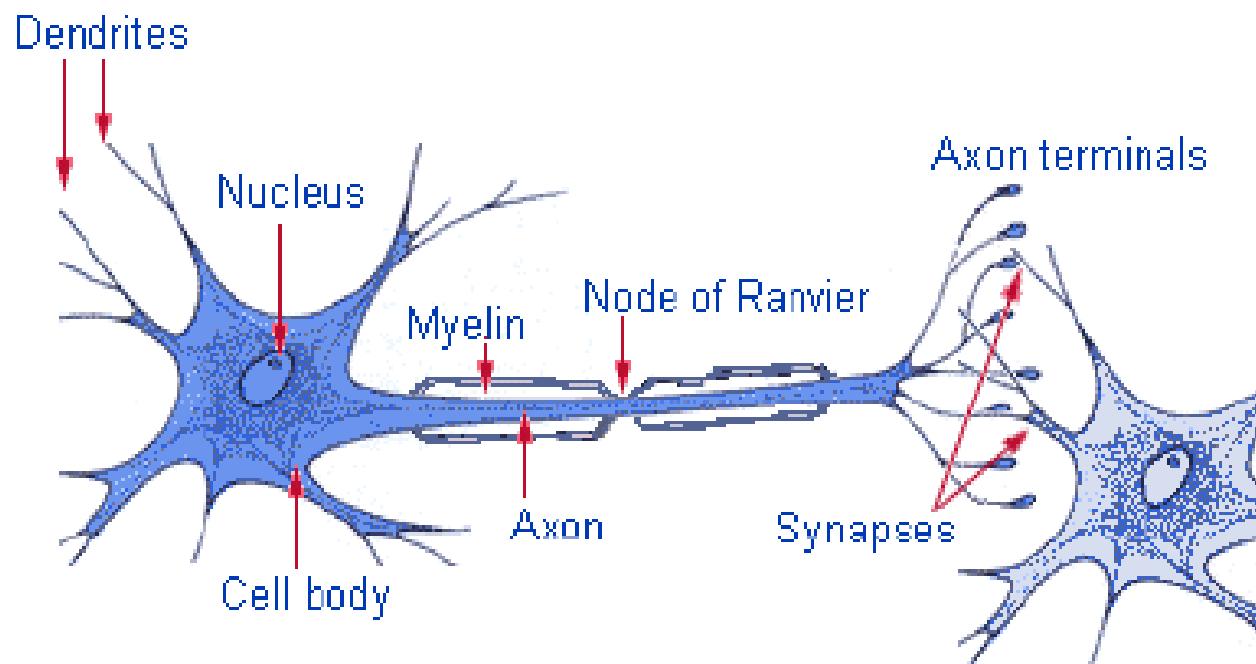
- تصمیم‌گیری

- پیش‌بینی

- محاسبه



# Schematic Drawing of Biological Neurons



# Basic components of biological neurones

1. The majority of *neurons* encode their activations or outputs as a series of brief electrical pulses (i.e. spikes or action potentials).
2. The neuron's *cell body (soma)* processes the incoming activations and converts them into output activations.
3. *Dendrites* are fibres which emanate from the cell body and provide the receptive zones that receive activation from other neurons.
5. *Axons* are fibres acting as transmission lines that send activation to other neurons.
6. The junctions that allow signal transmission between the axons and dendrites are called *synapses*.

# عملکرد نرون طبیعی

- دریافت سیگنال از سایر نرون‌ها توسط دندانه‌های انتقالی (Receptor)
- عبور سیگنال‌ها با یک فرآیند شیمیایی از فاصلهٔ سیناپسی (Synaptic Gap)
- عمل شیمیایی انتقال دهنده، سیگنال ورودی را تغییر می‌دهند (تضییف/تقویت سیگنال)
- سوما سیگنال‌های ورودی به سلول را جمع می‌بندد
- زمانی که یک سلول به اندازهٔ کافی ورودی دریافت نماید، برانگیخته می‌شود و سیگنالی را از آکسون خود به سلول‌های دیگر می‌فرستد.
- انتقال سیگنال از یک نرون خاص نتیجهٔ غلظت‌های مختلف یون‌ها در اطراف پوشش آکسون نرون («مادهٔ سفید» مغز) می‌باشد.
  - یون‌ها = پتاسیم، سدیم و کلرید
- سیگنال‌ها به صورت ضربه‌های الکتریکی هستند

# Brains versus Computers

- Neurons respond slowly
  - $10^{-3}$  s compared to  $10^{-9}$  s for electrical circuits
- The brain uses massively parallel computation
  - $\approx 10^{11}$  neurons in the brain compared with 10 of thousands of processors in the most powerful parallel computers.
  - $\approx 10^4$  connections per neuron
- the brain is able to perform many tasks much faster than any conventional computer. This is because of the massively parallel structure of biological neural networks.

شبکه عصبی مصنوعی: یک سیستم پردازش اطلاعات با ویژگی های مشترک  
با شبکه های عصبی طبیعی

# Why Artificial Neural Networks?

There are two basic reasons why we are interested in building artificial neural networks (ANNs):

- **Technical viewpoint:** Some problems such as character recognition or the prediction of future states of a system require massively parallel and adaptive processing.
- **Biological viewpoint:** ANNs can be used to replicate and simulate components of the human (or animal) brain, thereby giving us insight into natural information processing.

# A Brief History of Neural Network

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

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

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

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

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

**1982** Kohonen developed the Self-Organising Maps  
**1986** Back propagation Overcame many of the Minsky & Papert objections

**1990s** Radial Basis Function Networks, Support Vector Machines(SVM), Bayesian approaches

**2000-present** Many developments such as:

- Attempts to recurrent neural nets with unsupervised pretraining, probabilistic neural nets, alternative learning rules
- Deep Neural Networks
- Convolutional Neural Networks

# Some Applications

- Brain modelling
- Models of human development – help children with developmental problems اختلالات تکاملی
- Simulations of adult performance – aid our understanding of how the brain works
- Neuropsychological models – suggest remedial actions for brain damaged patients مدل های عصب روانشناختی – اقدامات درمانی برای بیماران آسیب دیده مغزی

- **Real world applications**
- Financial- currency price prediction, predicting stocks
- Medical- Breast cancer cell analysis, EEG and ECG analysis, prosthesis design, hospital expense reduction, hospital quality improvement
- Robotics-Trajectory control, manipulator controllers, vision systems
- Pattern recognition-speech recognition, hand-writing recognition
- Data analysis - data compression, data mining, PCA
- Noise reduction - function approximation, ECG noise reduction
- Climate change, Computer games, flight control, chemical engineering,...

## بازشناسی چهره

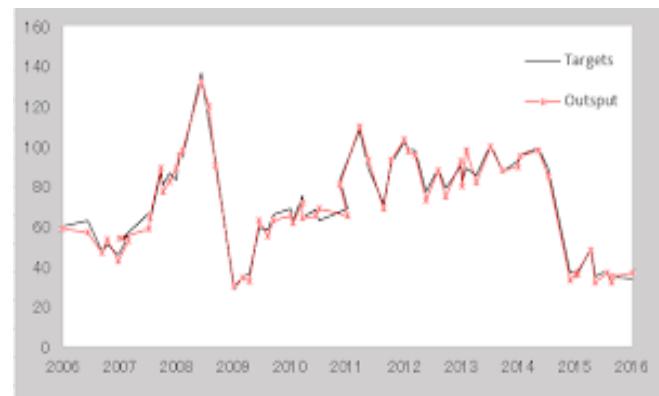


## تشخیص دستخط

Handwritten note:

$$f(z) = \frac{1}{2\pi} \int_0^{2\pi} u(e^{i\psi}) \frac{e^{i\psi} + z}{e^{i\psi} - z} d\psi, |z| < 1$$

## پیش بینی قیمت نفت

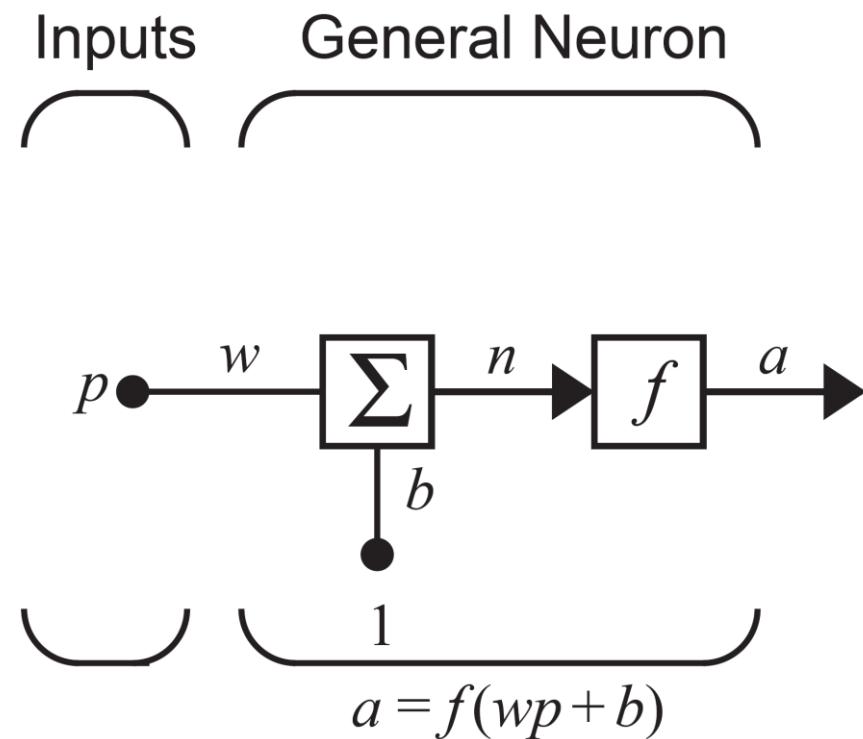


## بازشناسی گفتار

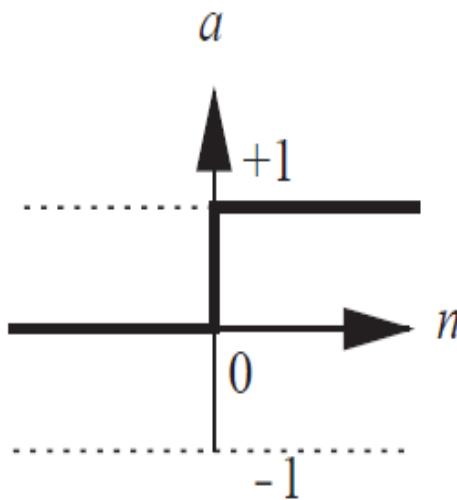


# Neuron Model and Network Architectures

# Single-Input Neuron

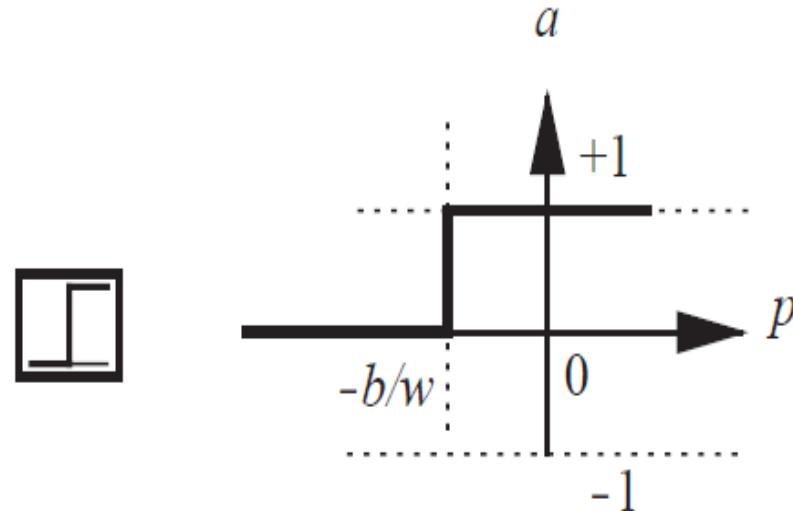


# Transfer Functions



$$a = \text{hardlim}(n)$$

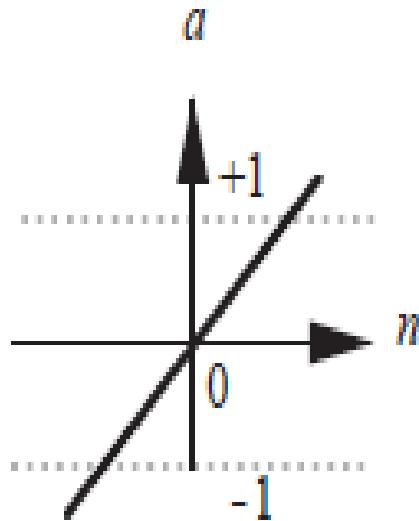
Hard Limit Transfer Function



$$a = \text{hardlim}(wp + b)$$

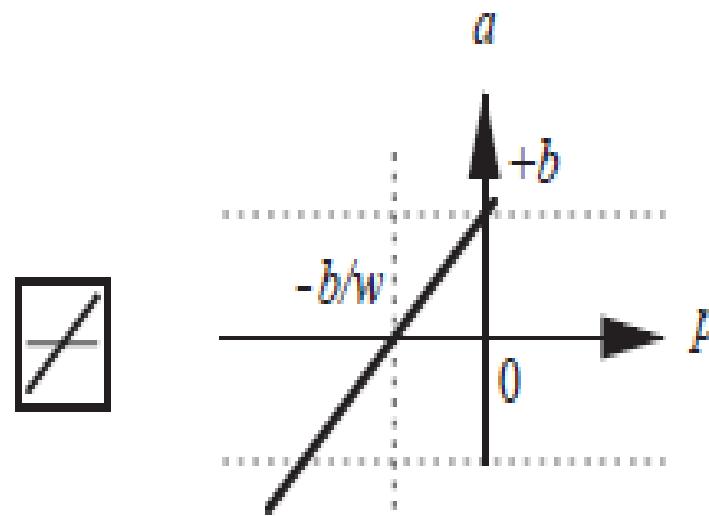
Single-Input *hardlim* Neuron

# Transfer Functions



$$a = \text{purelin}(n)$$

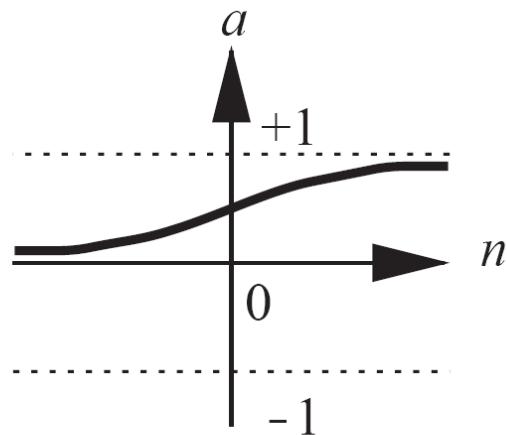
Linear Transfer Function



$$a = \text{purelin}(wp + b)$$

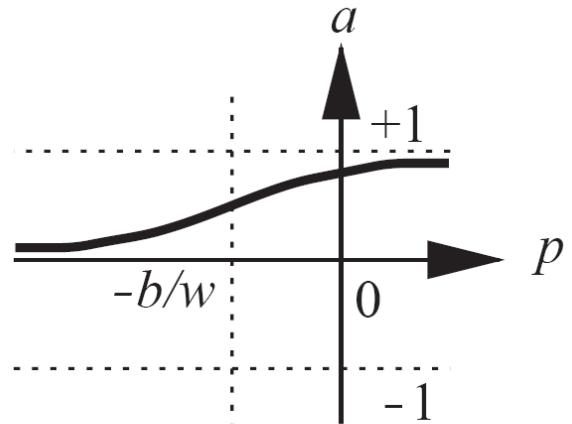
Single-Input *purelin* Neuron

# Transfer Functions



$$a = \text{logsig}(n)$$

Log-Sigmoid Transfer Function



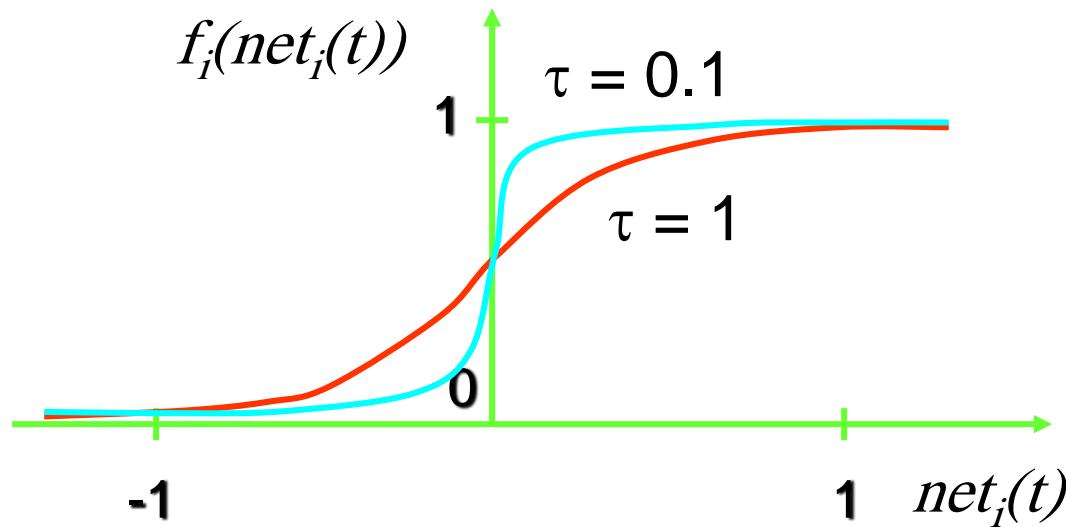
$$a = \text{logsig}(wp + b)$$

Single-Input  $\text{logsig}$  Neuron

| Name                        | Input/Output Relation   | Icon  | MATLAB Function |
|-----------------------------|---|---|-----------------|
| Hard Limit                  | $a = 0 \quad n < 0$<br>$a = 1 \quad n \geq 0$                                     |    | hardlim         |
| Symmetrical Hard Limit      | $a = -1 \quad n < 0$<br>$a = +1 \quad n \geq 0$                                   |    | hardlims        |
| Linear                      | $a = n$   |    | purelin         |
| Saturating Linear           | $a = 0 \quad n < 0$<br>$a = n \quad 0 \leq n \leq 1$<br>$a = 1 \quad n > 1$       |    | satlin          |
| Symmetric Saturating Linear | $a = -1 \quad n < -1$<br>$a = n \quad -1 \leq n \leq 1$<br>$a = 1 \quad n > 1$    |    | satlins         |
| Log-Sigmoid                 | $a = \frac{1}{1 + e^{-n}}$  |    | logsig          |
| Hyperbolic Tangent Sigmoid  | $a = \frac{e^n - e^{-n}}{e^n + e^{-n}}$   |  | tansig          |
| Positive Linear             | $a = 0 \quad n < 0$<br>$a = n \quad 0 \leq n$                                     |  | poslin          |
| Competitive                 | $a = 1 \quad \text{neuron with max } n$<br>$a = 0 \quad \text{all other neurons}$ |  | compet          |

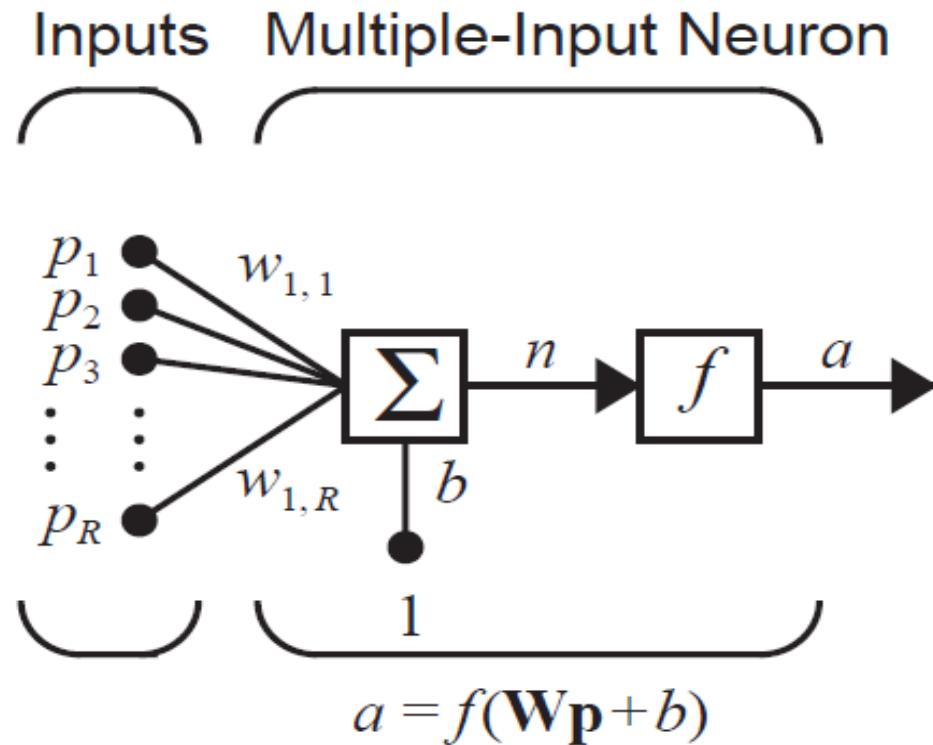
# Sigmoidal Neurons

$$f_i(\text{net}_i(t)) = \frac{1}{1 + e^{-(\text{net}_i(t) - \theta)/\tau}}$$



- The parameter  $\tau$  controls the slope of the sigmoid function, while the parameter  $\theta$  controls the horizontal offset of the function in a way similar to the threshold neurons.

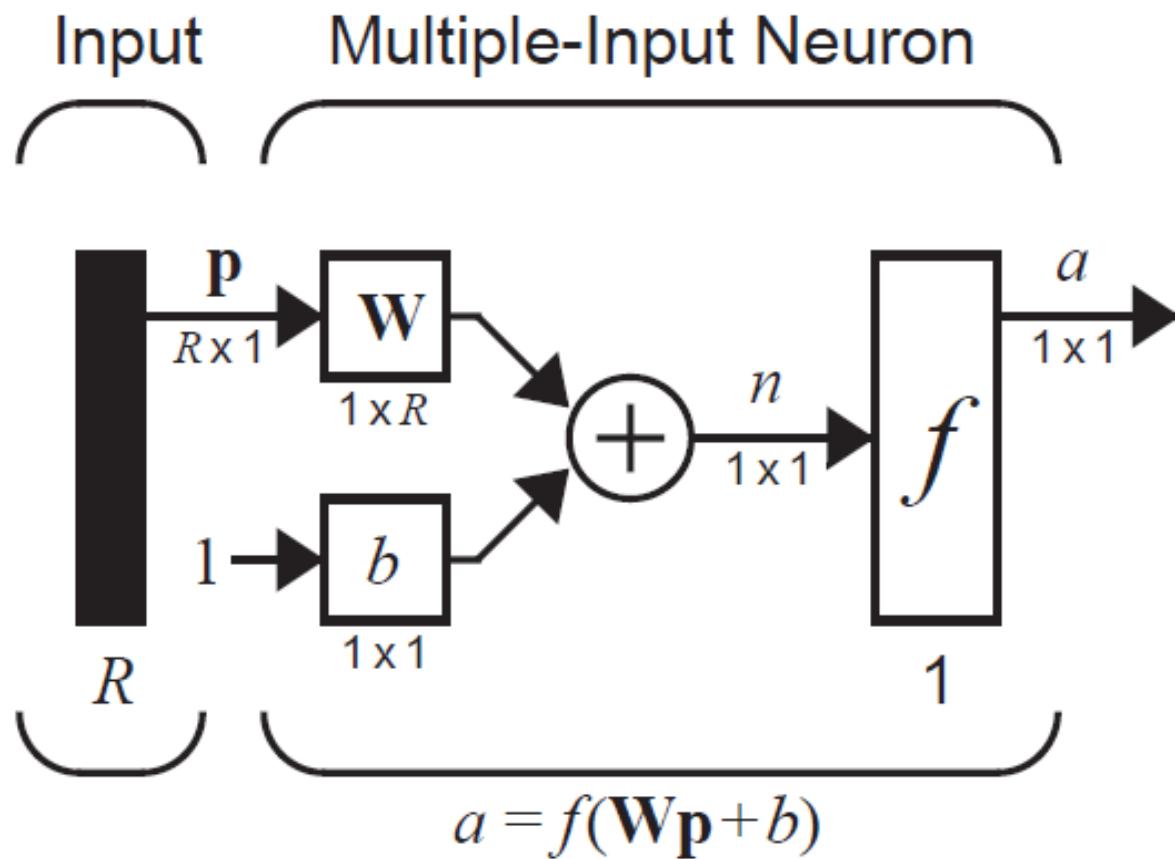
# Multiple-Input Neuron



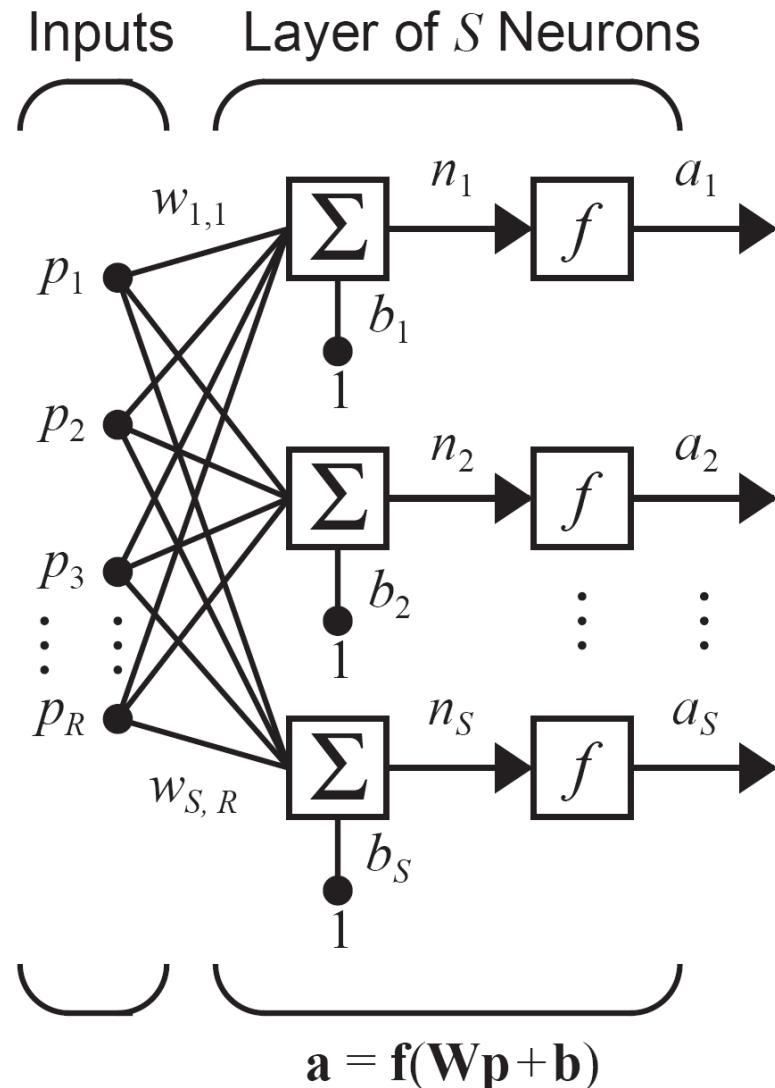
$$n = w_{1,1}p_1 + w_{1,2}p_2 + \dots + w_{1,R}p_R + b$$

$$n = \mathbf{W}\mathbf{p} + b$$

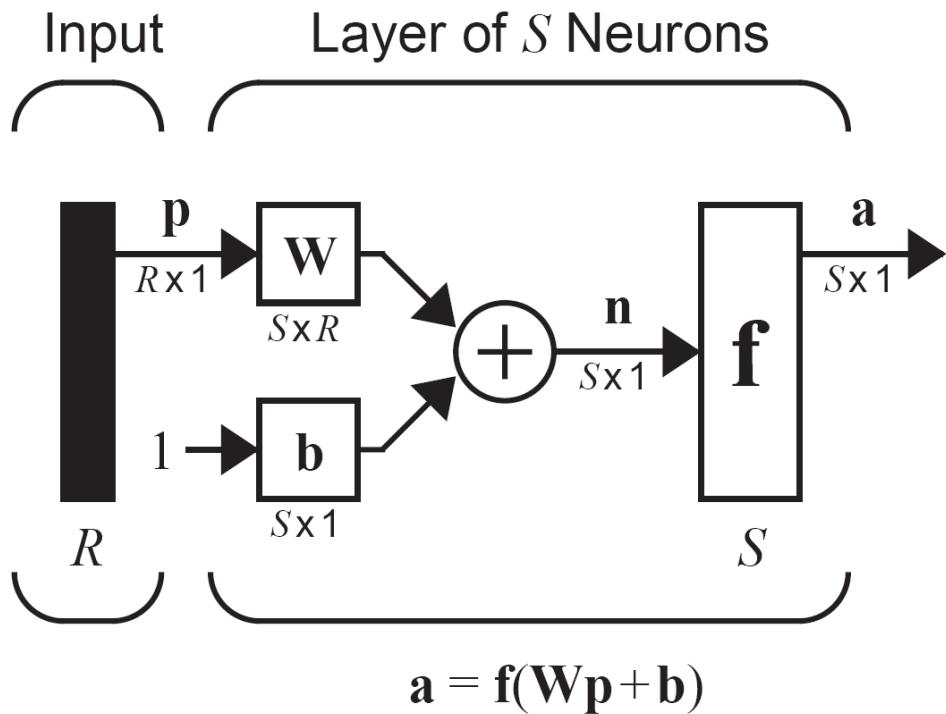
# Abbreviated Notation



# Layer of Neurons



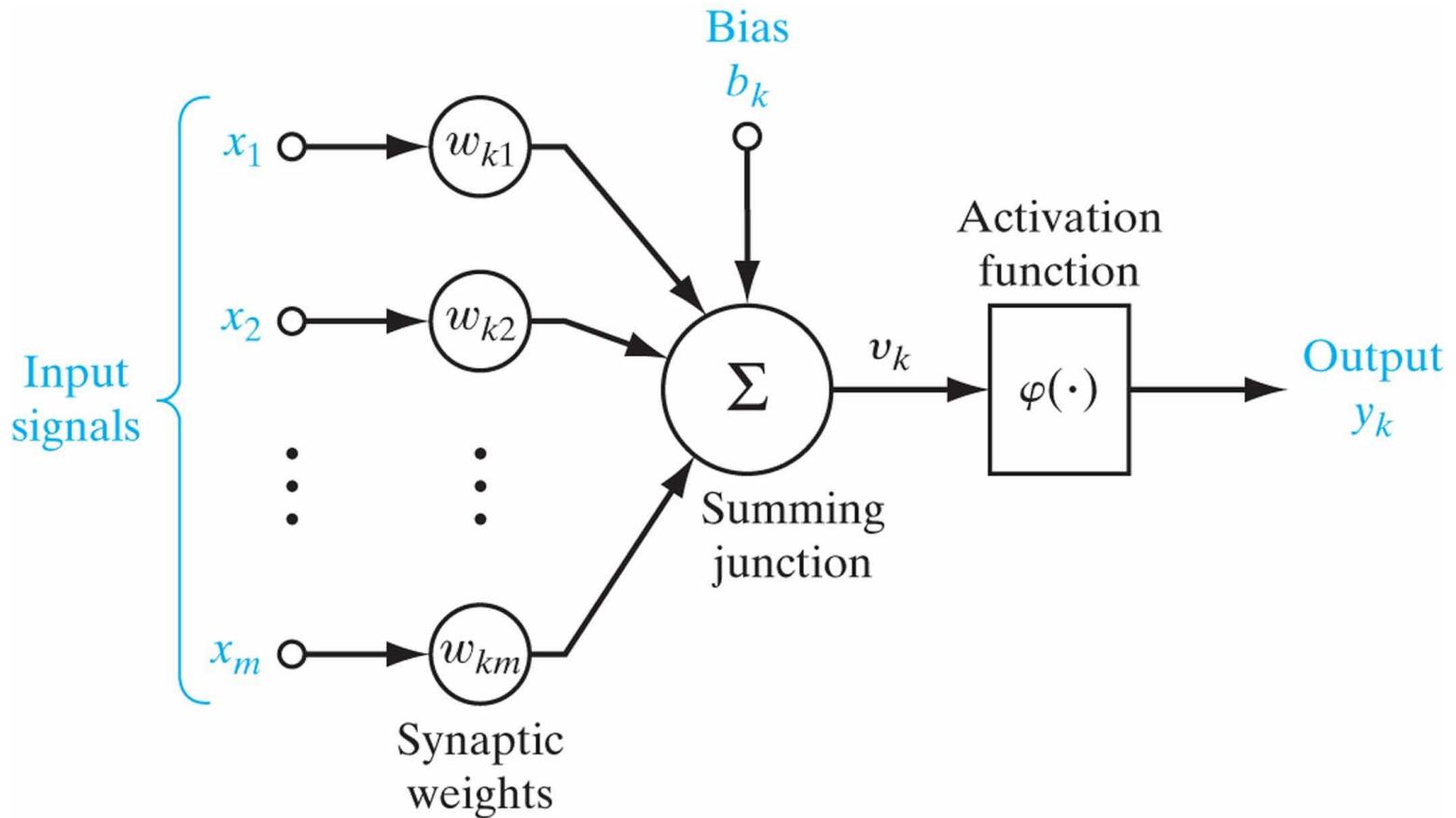
# Abbreviated Notation



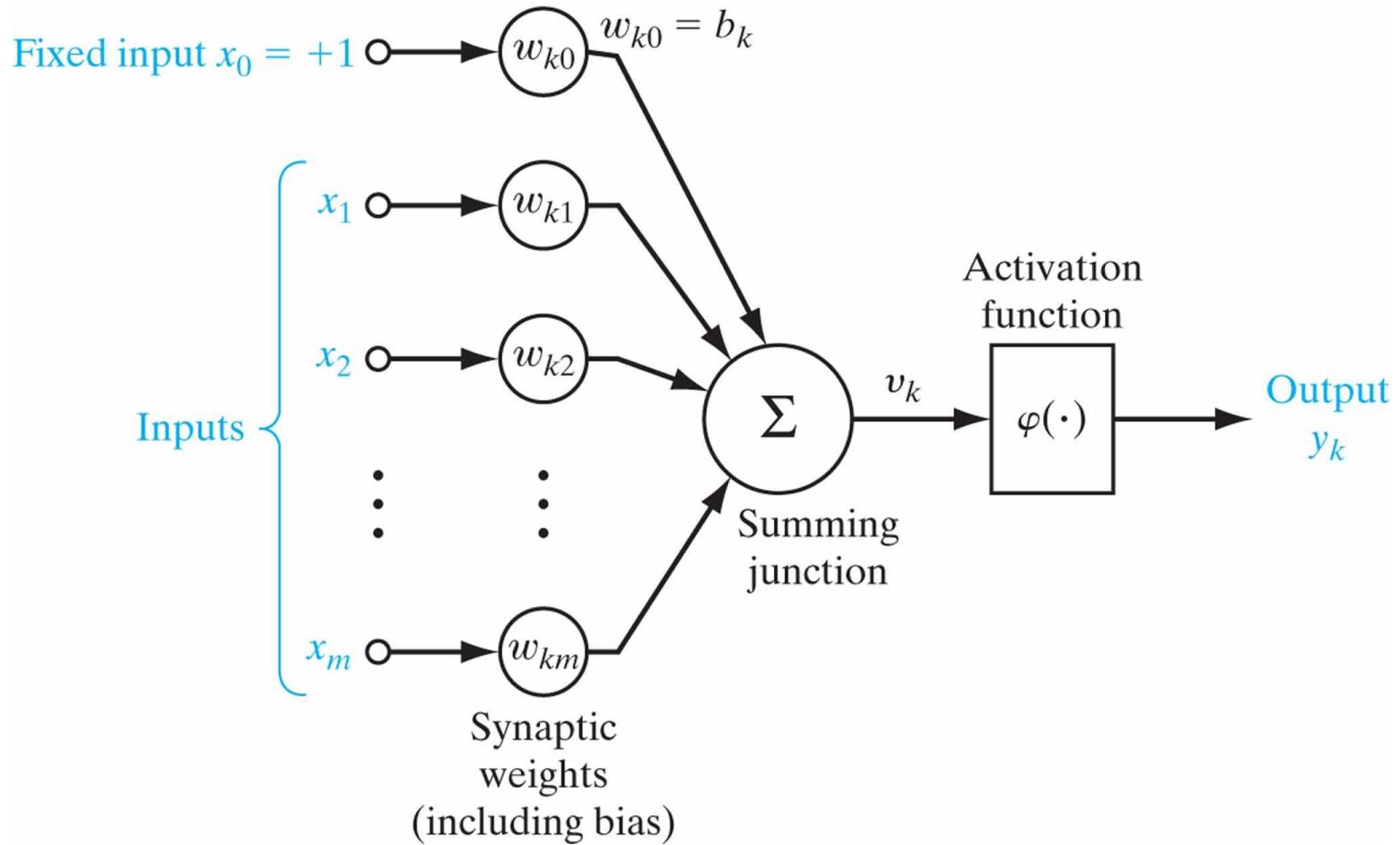
$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,R} \\ w_{2,1} & w_{2,2} & \dots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \dots & w_{S,R} \end{bmatrix}$$

$$\mathbf{p} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_R \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_S \end{bmatrix} \quad \mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_S \end{bmatrix}$$

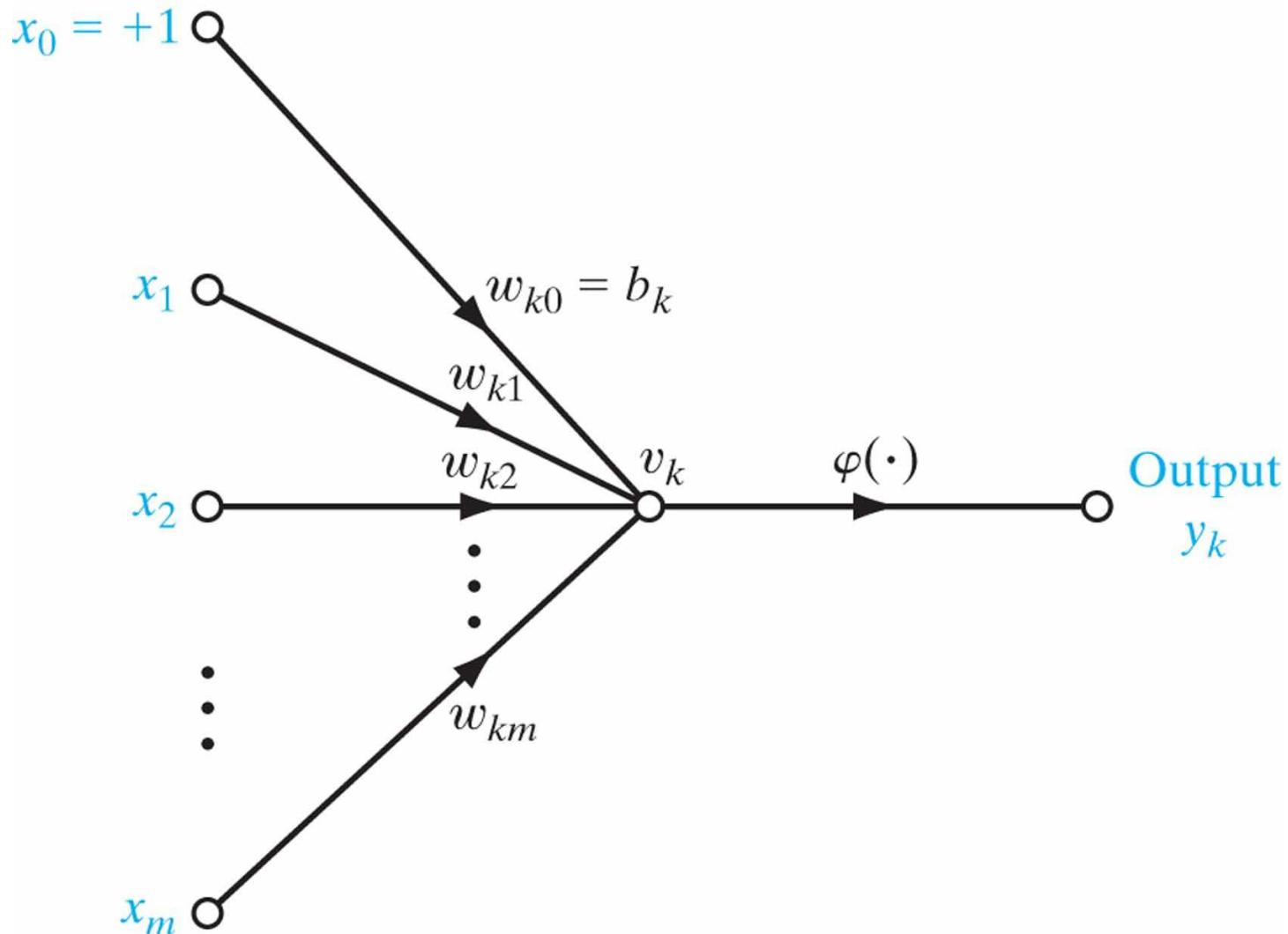
## Another model of a neuron, labeled $k$



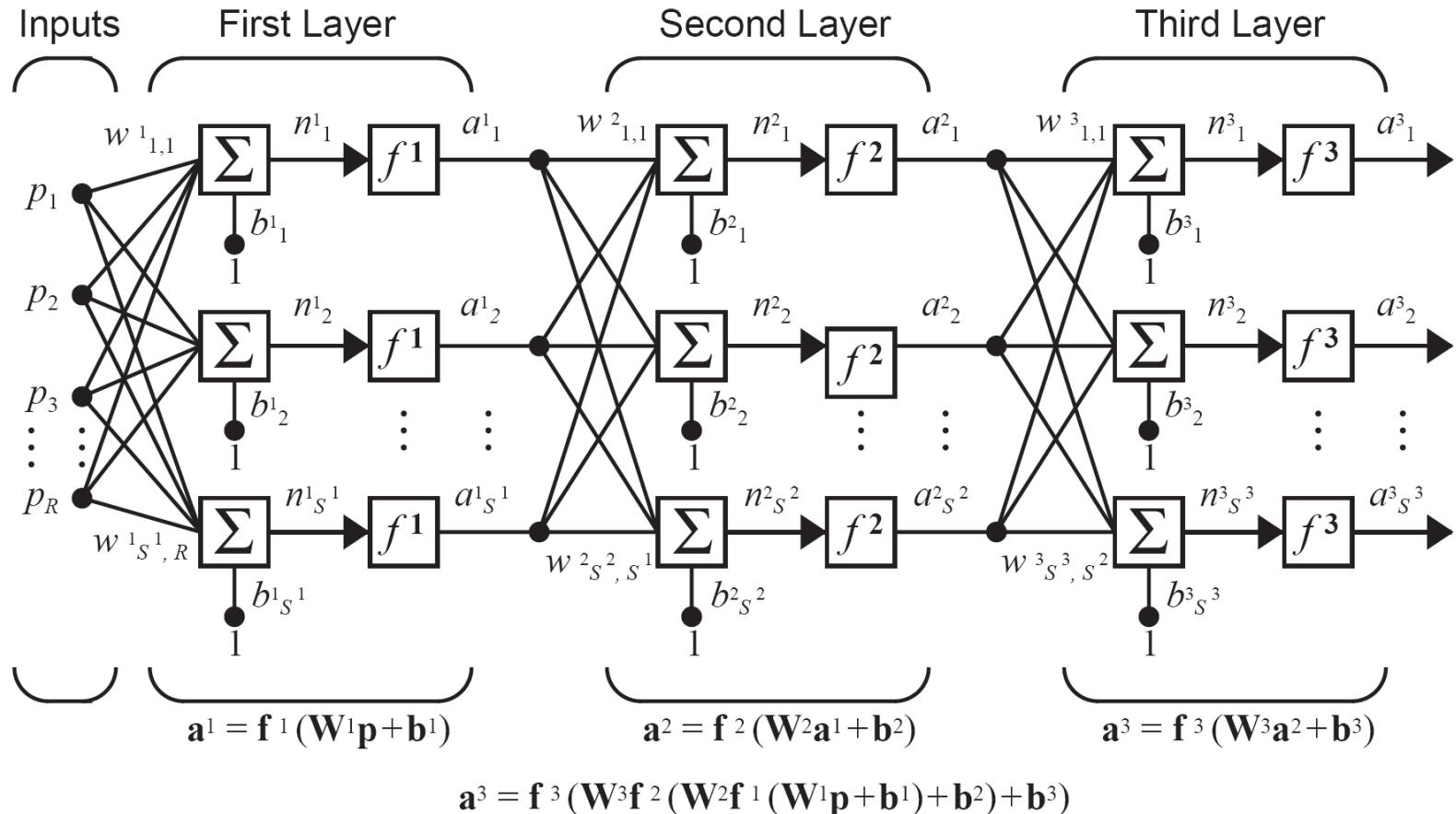
# Nonlinear model of a neuron; $w_{k0}$ accounts for the bias $b_k$



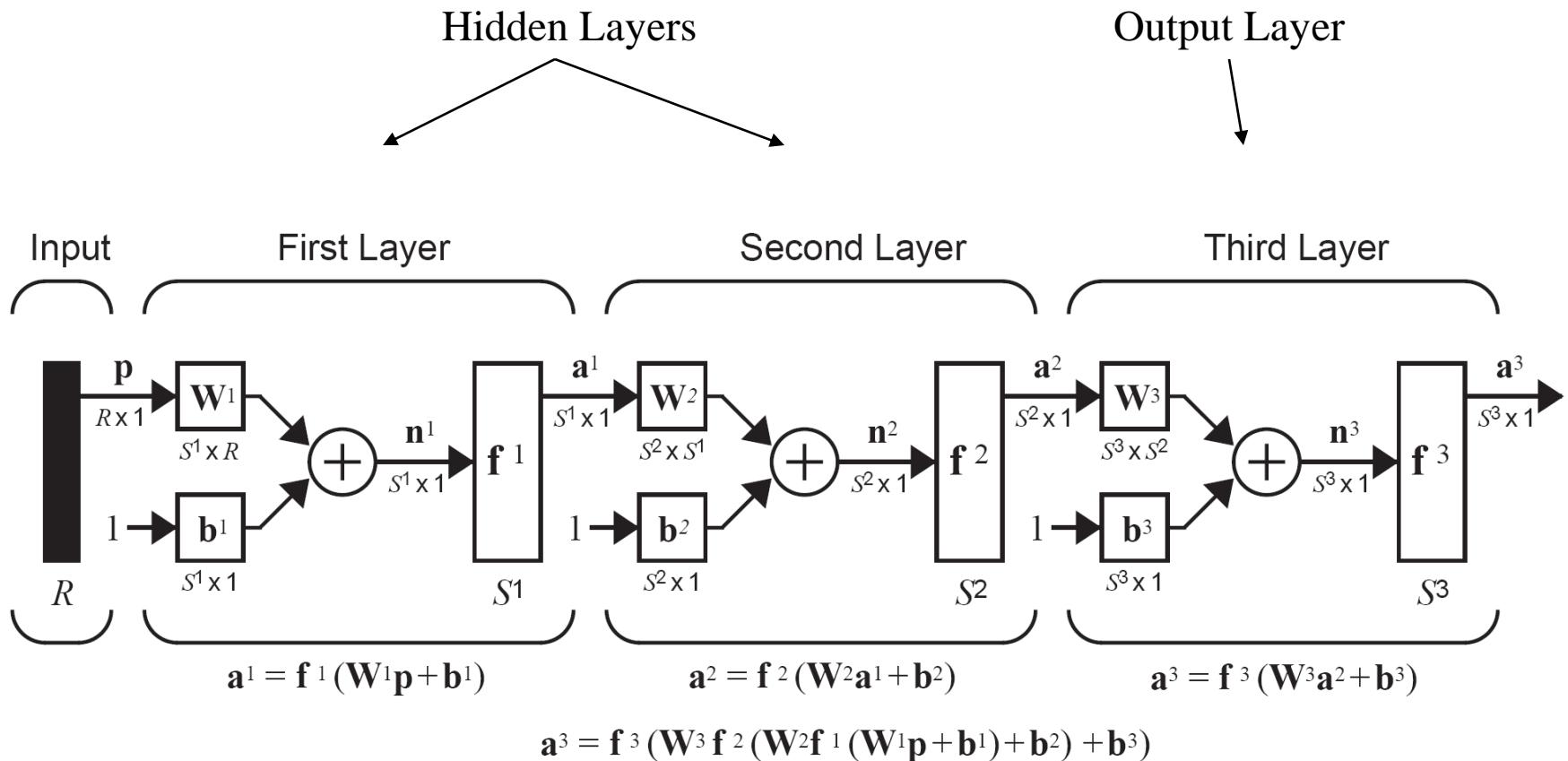
# Signal-flow graph of a neuron



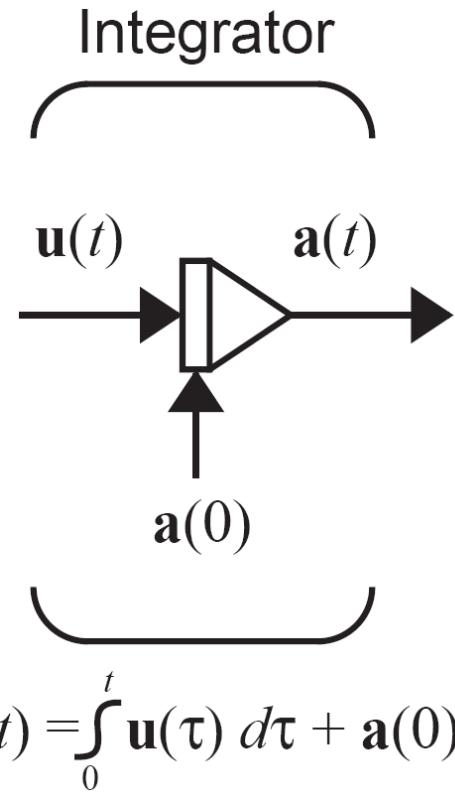
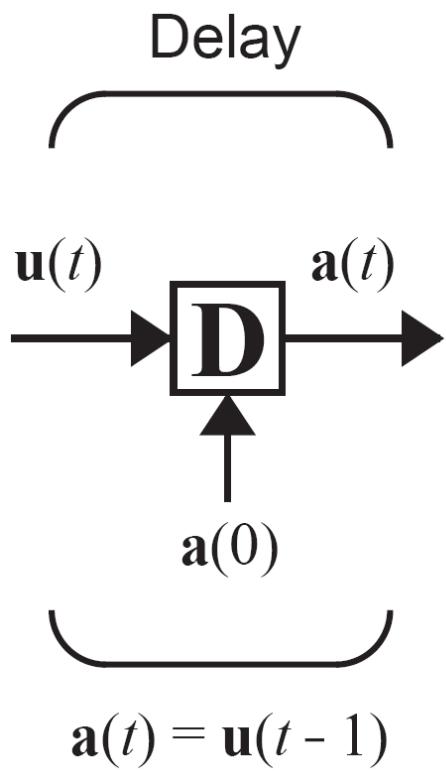
# Multilayer Network



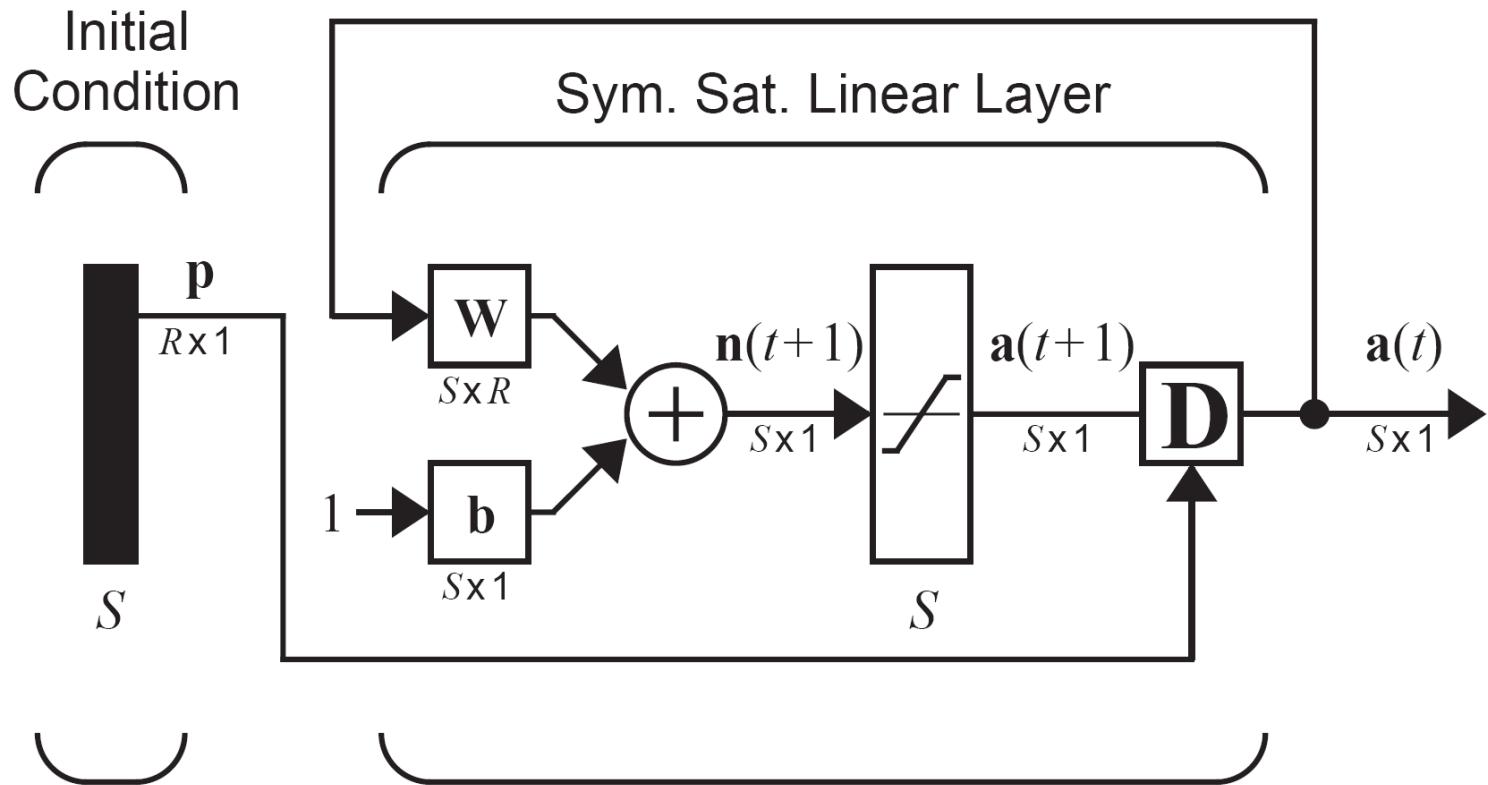
# Abbreviated Notation



# Delays and Integrators



# Recurrent Network



Discrete time recurrent network

$$\mathbf{a}(1) = \text{satlins}(\mathbf{W}\mathbf{a}(0) + \mathbf{b}) = \text{satlins}(\mathbf{W}\mathbf{p} + \mathbf{b})$$

$$\mathbf{a}(2) = \text{satlins}(\mathbf{W}\mathbf{a}(1) + \mathbf{b})$$

# How to pick an architecture

- Number of network inputs = number of problem inputs
- Number of neurons in output layer = number of problem outputs
- Output layer transfer function choice at least partly determined by problem specification of the outputs

# Linear Algebra review

Let

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

be a column vector with  $n$  elements. A column vector is distinct from a row vector with  $n$  elements, which we denote:  $\mathbf{z} = [z_1 \ z_2 \ \dots \ z_n]$ .

- The transpose of a column vector is a row vector, and vice versa. Concretely,  $\mathbf{x} = [x_1 \ x_2 \ \dots \ x_n]^T$ .
- The dot product of two column vectors,  $\mathbf{x}$  and  $\mathbf{y}$ , is given by

$$(\mathbf{x}, \mathbf{y}) = \mathbf{x}^T \mathbf{y} = \sum_{i=1}^n x_i y_i$$

- $\mathbf{x}$  and  $\mathbf{y}$  are *orthogonal* if  $\mathbf{x}^T \mathbf{y} = 0$ .

# Norm

- A scalar function of a vector  $\mathbf{x}$  is called a **norm**,  $\|\mathbf{x}\|$ ,
- provided the following are satisfied:
- $\|\mathbf{x}\| \geq 0$  .
- $\|\mathbf{x}\| = 0$  iff  $\mathbf{x} = 0$  .
- $\|a\mathbf{x}\| = |a| \|\mathbf{x}\|$  for scalar  $a$  .
- $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$  .

# Example:

The  $p$ -norm of a vector  $\Rightarrow$

$$\|\mathbf{x}\|_p = \left( \sum_{i=1}^n |x_i|^p \right)^{1/p}$$

Standard Euclidean Norm

$$\|\mathbf{x}\| = (\mathbf{x}, \mathbf{x})^{1/2}$$

$$\|\mathbf{x}\| = (\mathbf{x}^T \mathbf{x})^{1/2} = (x_1^2 + x_2^2 + \dots + x_n^2)^{1/2}$$

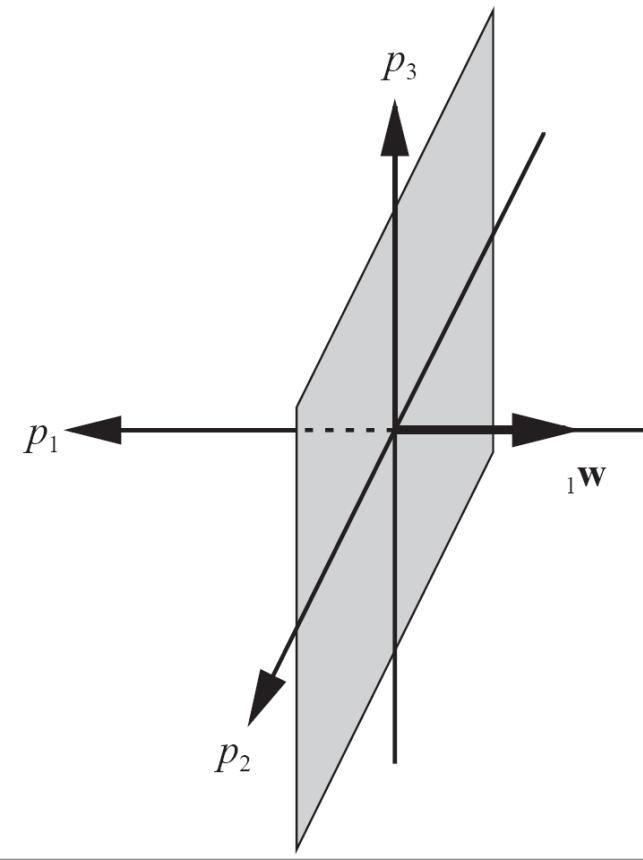
Generalizing the concept of angle for vectors  
of dimension greater than two:

$$\cos(\theta) = (\mathbf{x}, \mathbf{y}) / (\|\mathbf{x}\| \|\mathbf{y}\|)$$

# Orthogonality

Two vectors  $x, y \in X$  are orthogonal if  $(x, y) = 0$ .

## Example



Any vector in the  $p_2, p_3$  plane is orthogonal to the weight vector.

We can also have orthogonal spaces. A vector  $x \in X$  is orthogonal to a subspace  $X_1$  if  $x$  is orthogonal to every vector in  $X_1$  Or  $x \perp X_1$ .

# Matrices

Let

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

be an  $m \times n$  matrix. Note that we will sometimes denote the  $i, j$  element of  $\mathbf{A}$  as  $\mathbf{A}_{ij}$ . For the above matrix,  $\mathbf{A}_{ij} = a_{ij}$ .

- The transpose of  $\mathbf{A}$ , denoted  $\mathbf{A}^T$ , satisfies:

$$\mathbf{A}_{ij} = (\mathbf{A}^T)_{ji}$$

- If the matrix is square, i.e.,  $m = n$ , and has rank  $n$ , then the inverse of  $\mathbf{A}$ , denoted  $\mathbf{A}^{-1}$ , satisfies

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$$

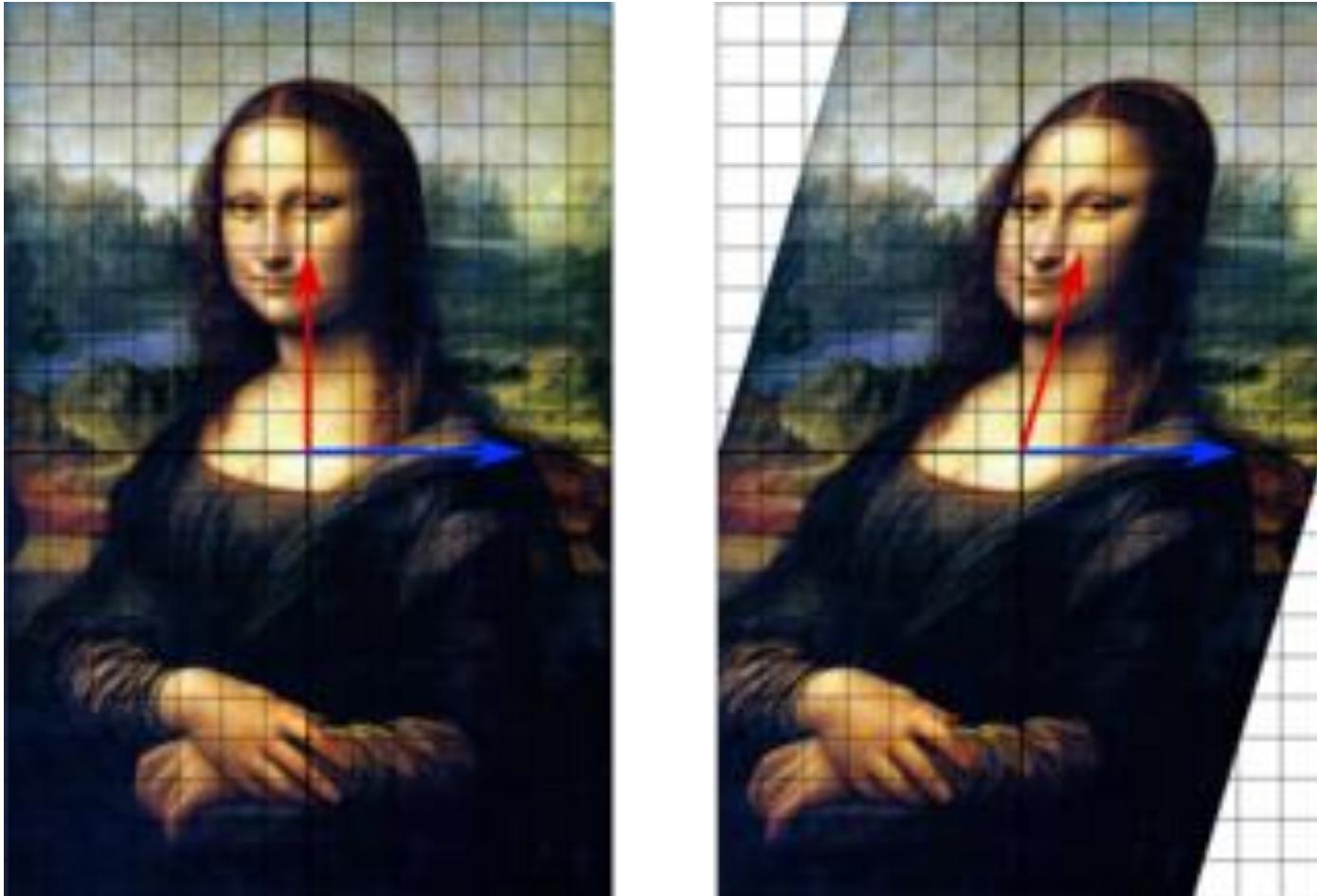
where  $\mathbf{I}$  is an  $n \times n$  identity matrix.

- A matrix is *symmetric* if  $\mathbf{A} = \mathbf{A}^T$ .

# Computing the Eigenvalues

- An eigenvector,  $\mathbf{u}_i$ , and its corresponding eigenvalue,  $\lambda_i$ , of a matrix  $\mathbf{A} \in \mathbb{R}^{n \times n}$  satisfy:

$$\mathbf{A}\mathbf{u}_i = \lambda_i\mathbf{u}_i$$



In this [mapping](#) the red arrow changes direction, but the blue arrow does not. The blue arrow is an eigenvector of this shear mapping because it does not change direction, and since its length is unchanged, its eigenvalue is 1.

# Computing the Eigenvalues

$$\mathbf{A}\mathbf{z} = \lambda\mathbf{z}$$

$$[\mathbf{A} - \lambda \mathbf{I}] \mathbf{z} = \mathbf{0} \quad \rightarrow \quad |[\mathbf{A} - \lambda \mathbf{I}]| = 0$$

example:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad \left| \begin{bmatrix} 1-\lambda & 1 \\ 0 & 1-\lambda \end{bmatrix} \right| = 0 \quad (1-\lambda)^2 = 0 \quad \begin{aligned} \lambda_1 &= 1 \\ \lambda_2 &= 1 \end{aligned}$$

$$\begin{bmatrix} 1-\lambda & 1 \\ 0 & 1-\lambda \end{bmatrix} \mathbf{z} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \mathbf{z}_1 = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} z_{11} \\ z_{21} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad z_{21} = 0 \quad \mathbf{z}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

For this transformation there is only one eigenvector.

# Diagonalization

If the eigenvalues are distinct, and we collect all eigenvectors  
In the following matrix:

$$\mathbf{B} = \begin{bmatrix} \mathbf{z}_1 & \mathbf{z}_2 & \dots & \mathbf{z}_n \end{bmatrix} \quad \{\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_n\} \quad \text{Eigenvectors}$$
$$\{\lambda_1, \lambda_2, \dots, \lambda_n\} \quad \text{Eigenvalues}$$

With the  
transform:

$$[\mathbf{B}^{-1} \mathbf{A} \mathbf{B}] = \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & \lambda_n \end{bmatrix}$$

the new matrix will be diagonal.<sup>n</sup>

# Example

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1-\lambda & 1 \\ 1 & 1-\lambda \end{bmatrix} = 0 \quad \lambda^2 - 2\lambda = (\lambda)(\lambda - 2) = 0 \quad \begin{aligned} \lambda_1 &= 0 \\ \lambda_2 &= 2 \end{aligned} \quad \begin{bmatrix} 1-\lambda & 1 \\ 1 & 1-\lambda \end{bmatrix} \mathbf{z} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\lambda_1 = 0 \quad \longrightarrow \quad \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \mathbf{z}_1 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} z_{11} \\ z_{21} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad z_{21} = -z_{11} \quad \mathbf{z}_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

$$\lambda_2 = 2 \quad \longrightarrow \quad \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \mathbf{z}_2 = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} z_{12} \\ z_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad z_{22} = z_{12} \quad \mathbf{z}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Diagonal Form:  $\mathbf{A}' = [\mathbf{B}^{-1} \mathbf{A} \mathbf{B}] = \begin{bmatrix} 1/2 & -1/2 \\ 1/2 & 1/2 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 2 \end{bmatrix}$

# Perceptron Learning Rule

**Learning Rule** or **Training Algorithm**: A procedure for modifying weights and biases of a network.

قانون یادگیری، الگوریتم تعلم

# Learning Rules

- Learning with a teacher:

Supervised Learning

(مثلاً آموزش پر سپترون)

- Learning without a teacher:

1. Reinforcement Learning

2. Unsupervised Learning

# Learning with a Teacher

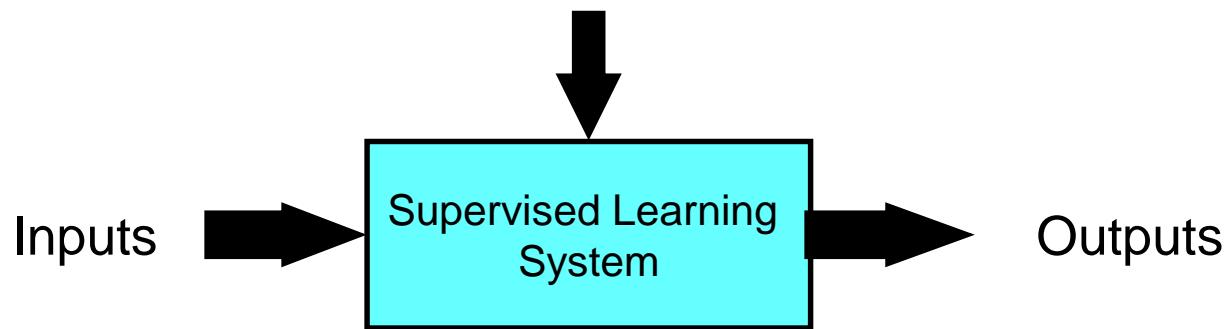
- **Supervised Learning**

- Network is provided with a set of examples of proper network behavior (inputs/targets)

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \dots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

- The desired response (target) represents the "optimum" action to be performed by the neural network. The network parameters are adjusted under the combined influence of the training vector and the error signal. The error signal is defined as the difference between the desired response and the actual response of the network.

Training Info = desired (target) outputs

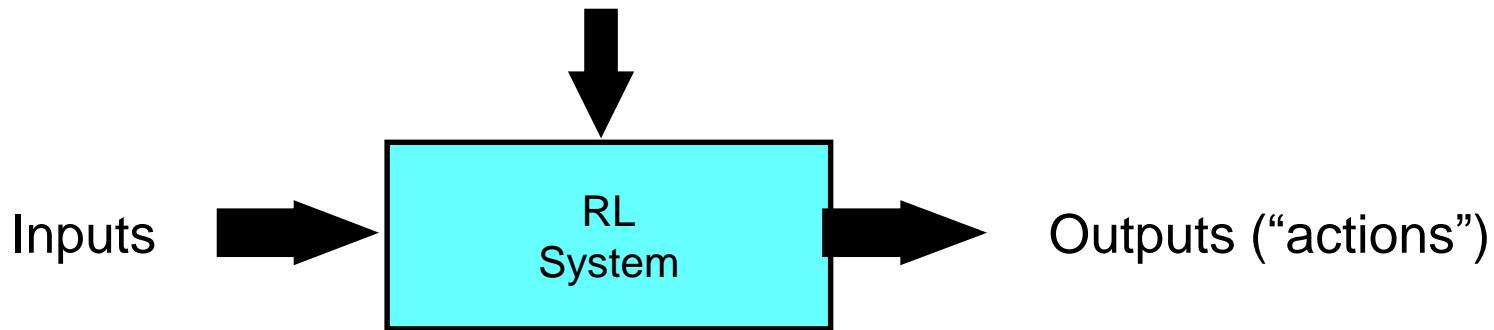


Error = (target output – actual output)

# Learning without a Teacher

- Reinforcement Learning
  - Network is only provided with a grade, or score, which indicates network performance.

Training Info = evaluations (“rewards” / “penalties”)



هدف: جمع کردن حداکثر پاداش ممکن

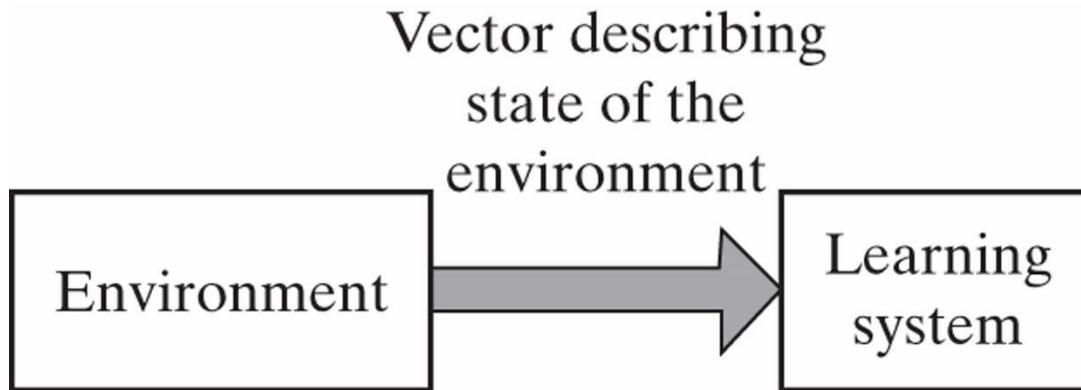
هیچگونه اطلاعات مربوط به گرادیان خطای موجود نیست.

حالت بعدی از روی عمل فعلی تعیین می‌شود.

یادگیری مبتنی بر سعی و خطا است.

# Learning without a Teacher

- Unsupervised Learning
  - Only network inputs are available to the learning algorithm.
  - We may use a competitive-learning rule.



## یادگیری نظارت شده

اگر هوا ابری باشد،  
باید چتری به همراه داشته باشی.

در این روش یک معلم یا ناظر وجود دارد که بهترین عمل در هر وضعیت را بلد است. این ناظر توصیه‌هایی را برای تصحیح شیوه‌ی عملکرد عامل، ارائه می‌دهد.

## یادگیری تقویتی

سابقا هوا ابری بود و تو چتری به همراه نداشتی؛

در نتیجه تو خیس شدی،

و این خیلی بد است.

در این نوع از یادگیری فیدبکی به صورت عبارات کمکی مثبت (پاداش) یا منفی (جریمه) به عامل یادگیرنده داده می‌شود. غالباً پاداش‌ها مقادیر اسکالاری همچون ۱- برای یک کار بد و ۱+ برای یک کار خوب هستند.

## یادگیری غیر نظارت شده

اگر آسمان ابری باشد، و من چتری با خود نداشته باشم،  
به احتمال زیاد خیس خواهم شد.

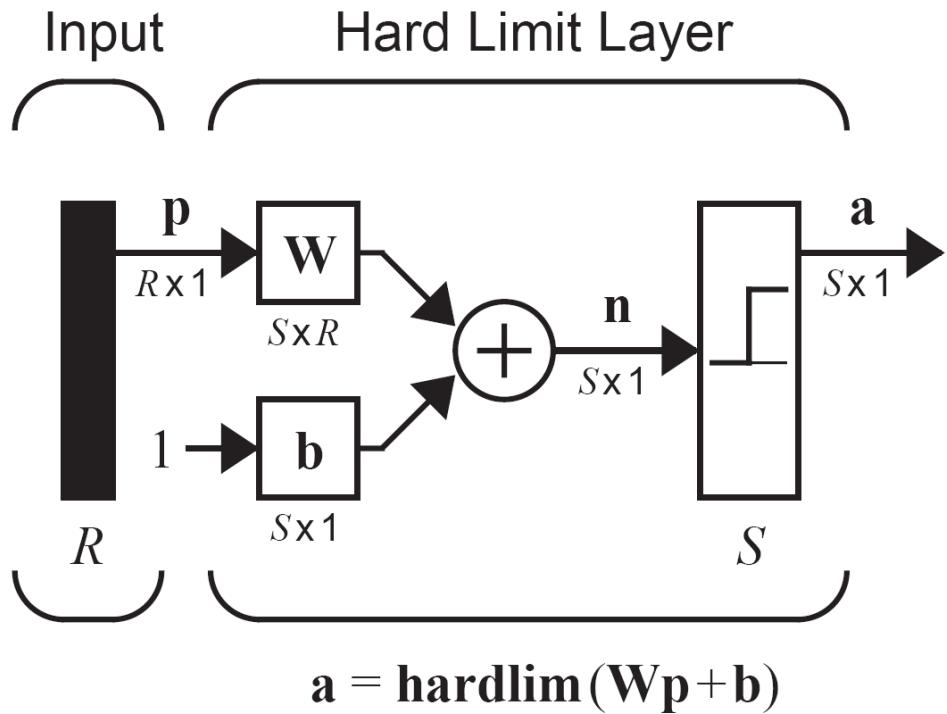
مشاهده‌ی نتیجه‌ی ترک کردن خانه، بدون چتر و در یک روز ابری، شکلی از فیدبک است. با این وجود، نتیجه‌هایی شبیه این، به طور مستقیم، پیشنهادی درباره‌ی انتخاب اعمال بهتر ندارند. این که عاملی ترجیح بدهد، که حتماً هنگام خروج از خانه چتری به همراه داشته باشد، کاملاً به این بستگی دارد که خیس شدن یا نشدن برای وی اهمیت داشته باشد.

مرجع مثال: درس  
یادگیری تقویتی،  
منصور فاتح

# Perceptron

- Perceptron was introduced by Frank Rosenblatt in the late 1950's (Rosenblatt, 1958) with a learning algorithm on it.

# Perceptron Architecture



$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$

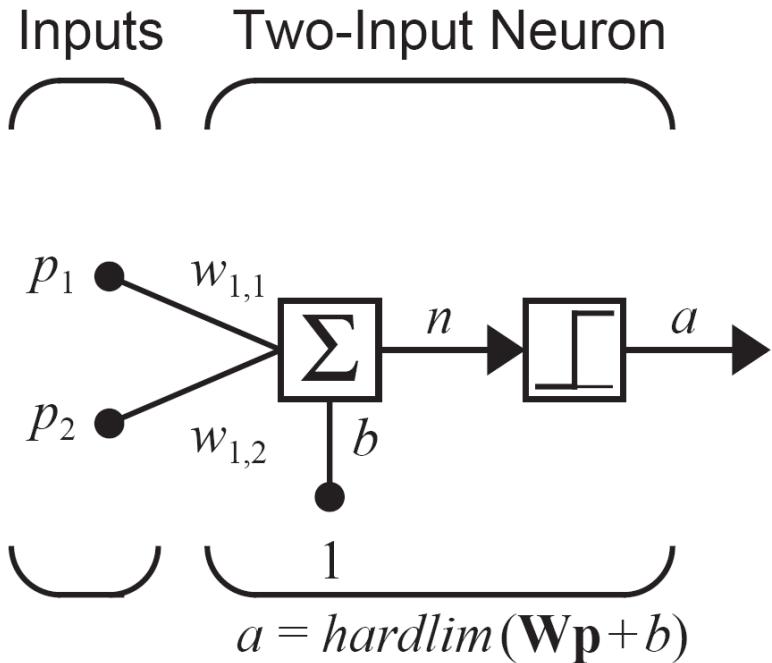
$$i\mathbf{w} = \begin{bmatrix} w_{i,1} \\ w_{i,2} \\ \vdots \\ w_{i,R} \end{bmatrix}$$

$$\mathbf{W} = \begin{bmatrix} 1\mathbf{w}^T \\ 2\mathbf{w}^T \\ \vdots \\ S\mathbf{w}^T \end{bmatrix}$$

$i\mathbf{w}$  is elements of  $i$ th row of  $\mathbf{W}$

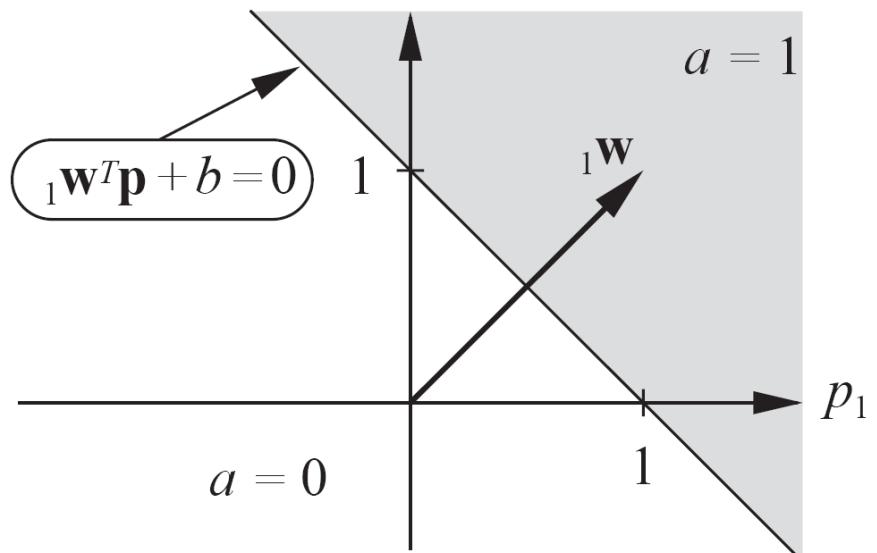
$$a_i = \text{hardlim}(n_i) = \text{hardlim}(i\mathbf{w}^T \mathbf{p} + b_i)$$

# Single-Neuron Perceptron



$$w_{1,1} = 1 \quad w_{1,2} = 1 \quad b = -1$$

$$p_2$$



$${}_1\mathbf{w}^T \mathbf{p} + b = 0$$

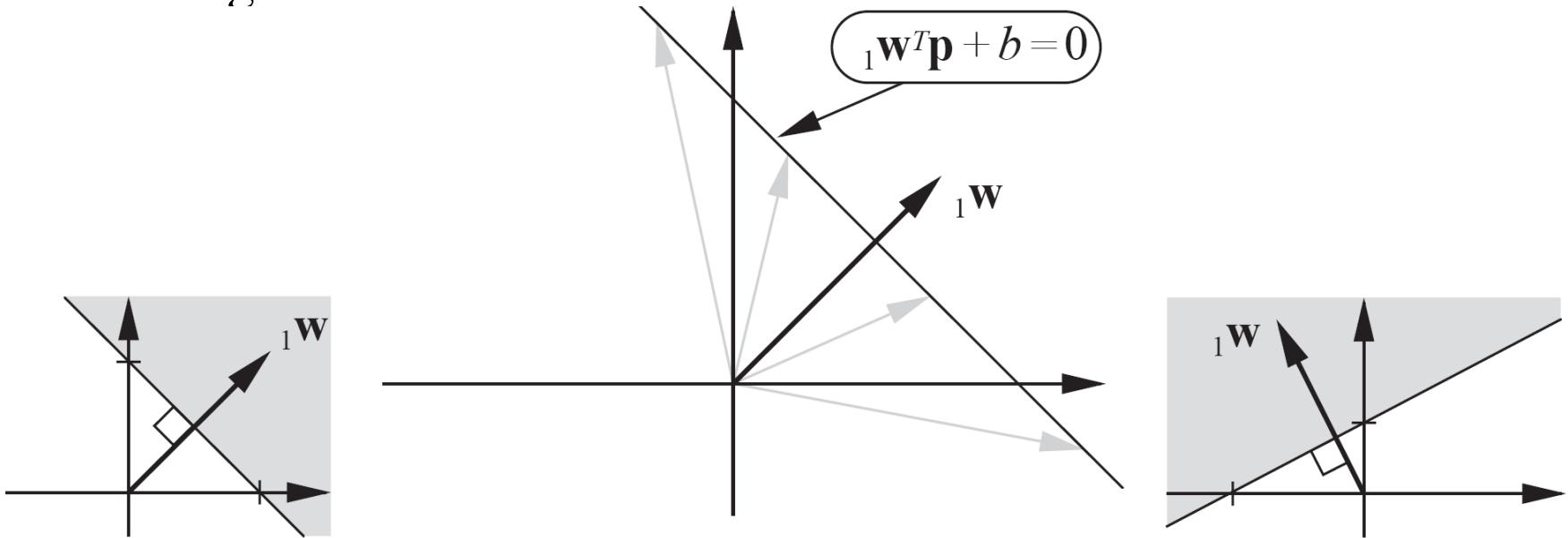
$$a = \text{hardlim}({}_1\mathbf{w}^T \mathbf{p} + b) = \text{hardlim}(w_{1,1}p_1 + w_{1,2}p_2 + b)$$

# Decision Boundary

$$_1\mathbf{w}^T \mathbf{p} + b = 0$$

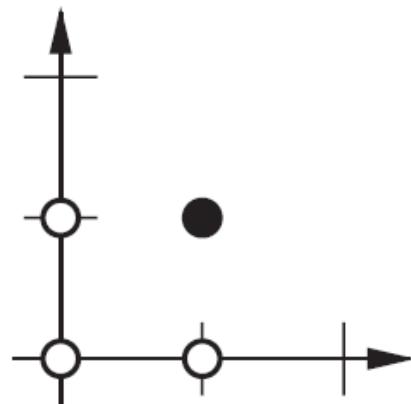
$$_1\mathbf{w}^T \mathbf{p} = -b$$

- All points on the decision boundary have the same inner product with the weight vector.
- Therefore they have the same projection onto the weight vector, and they must lie on a line orthogonal to the weight vector

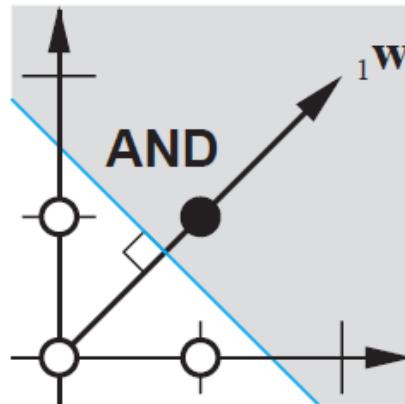


# Example - AND

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, t_1 = 0 \right\} \left\{ \mathbf{p}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, t_2 = 0 \right\} \left\{ \mathbf{p}_3 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, t_3 = 0 \right\} \left\{ \mathbf{p}_4 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, t_4 = 1 \right\}$$



# AND Solution



Weight vector should be orthogonal to the decision boundary.

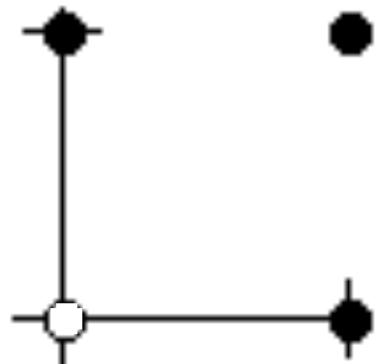
$$w_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$$

Pick a point on the decision boundary to find the bias.

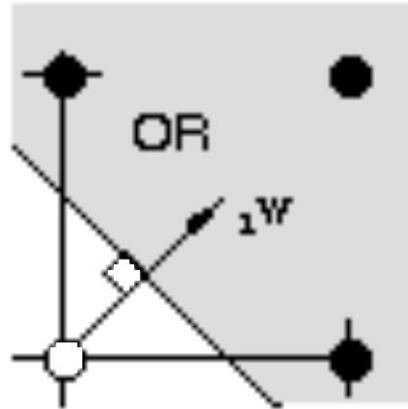
$$w_1^T p + b = \begin{bmatrix} 2 & 2 \end{bmatrix} \begin{bmatrix} 1.5 \\ 0 \end{bmatrix} + b = 3 + b = 0 \quad \Rightarrow \quad b = -3$$

# Example - OR

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, t_1 = 0 \right\} \quad \left\{ \mathbf{p}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, t_2 = 1 \right\} \quad \left\{ \mathbf{p}_3 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, t_3 = 1 \right\} \quad \left\{ \mathbf{p}_4 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, t_4 = 1 \right\}$$



# OR Solution



Weight vector should be orthogonal to the decision boundary.

$$w_1 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

Pick a point on the decision boundary to find the bias.

$$w_1^T p + b = [0.5 \ 0.5] \begin{bmatrix} 0 \\ 0.5 \end{bmatrix} + b = 0.25 + b = 0 \quad \Rightarrow \quad b = -0.25$$

# Multiple-Neuron Perceptron

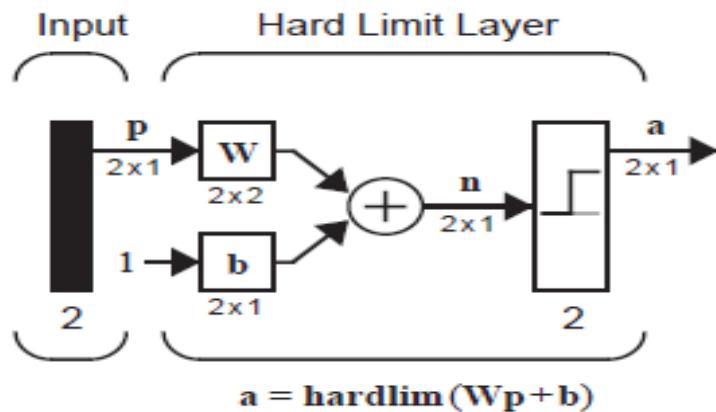
Each neuron will have its own decision boundary.

$$_i \mathbf{w}^T \mathbf{p} + b_i = 0$$

A single neuron can classify input vectors  
into two categories.

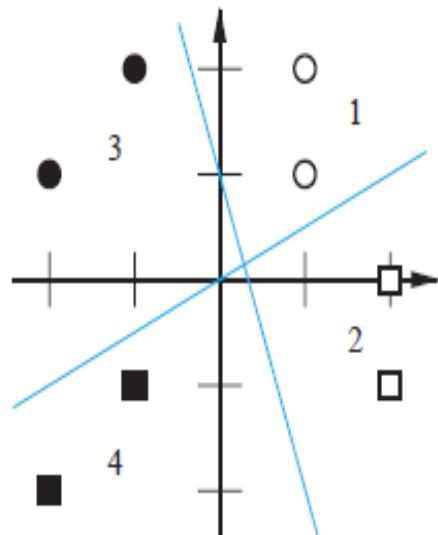
A multi-neuron perceptron can classify  
input vectors into  $2^S$  categories.

P4.3



$$\text{class 1: } \left\{ p_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, p_2 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right\}, \text{ class 2: } \left\{ p_3 = \begin{bmatrix} 2 \\ -1 \end{bmatrix}, p_4 = \begin{bmatrix} 2 \\ 0 \end{bmatrix} \right\},$$

$$\text{class 3: } \left\{ p_5 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, p_6 = \begin{bmatrix} -2 \\ 1 \end{bmatrix} \right\}, \text{ class 4: } \left\{ p_7 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, p_8 = \begin{bmatrix} -2 \\ -2 \end{bmatrix} \right\}$$



$${}_1^W = \begin{bmatrix} -3 \\ -1 \end{bmatrix} \text{ and } {}_2^W = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$

$$b_1 = -{}_1^W T p = -[-3 \ -1] \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 1,$$

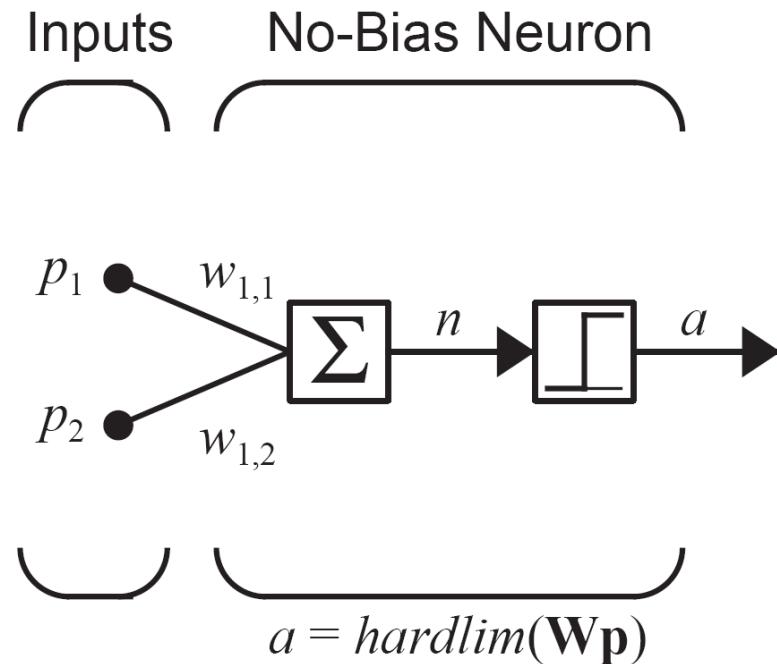
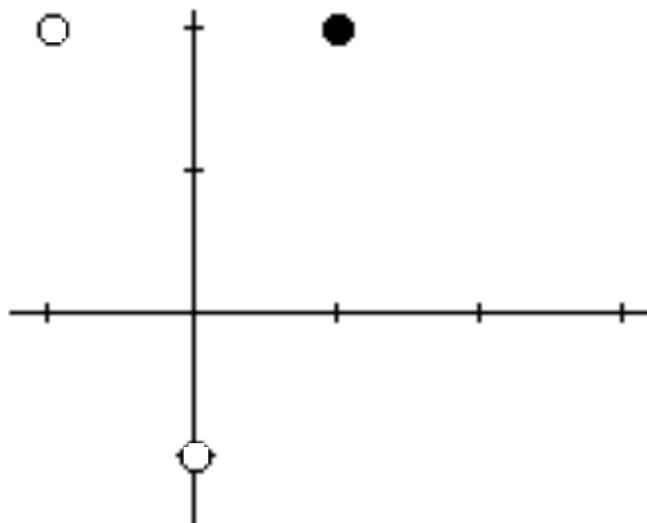
$$b_2 = -{}_2^W T p = -[1 \ -2] \begin{bmatrix} 0 \\ 0 \end{bmatrix} = 0.$$

# Learning Rule Test Problem

آزمایش بدون بایاس  $\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \dots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, t_1 = 1 \right\} \quad \left\{ \mathbf{p}_2 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, t_2 = 0 \right\} \quad \left\{ \mathbf{p}_3 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, t_3 = 0 \right\}$$

دایره تو پر کلاس  $t=1$  و دایره تو خالی کلاس  $t=0$  است.

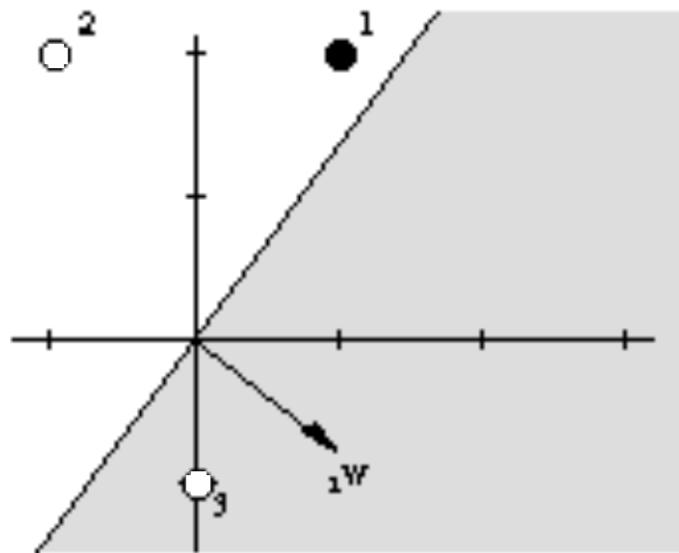


# Starting Point

Random initial weight:

$$_1\mathbf{w} = \begin{bmatrix} 1.0 \\ -0.8 \end{bmatrix}$$

Present  $\mathbf{p}_1$  to the network:



$$a = \text{hardlim}(_1\mathbf{w}^T \mathbf{p}_1) = \text{hardlim}\left(\begin{bmatrix} 1.0 & -0.8 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix}\right)$$

$a = \text{hardlim}(-0.6) = 0$       Incorrect Classification.

راه حل: بردار وزن باید در جهتی اصلاح شود که مرز تصمیم گیری بسمت بردار  $\mathbf{p}_1$  حرکت کند.

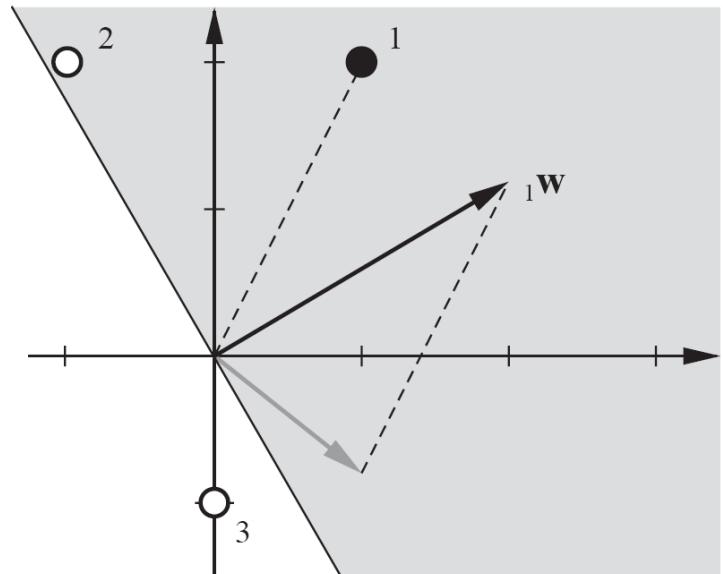
- Set  ${}_1\mathbf{w}$  to  $\mathbf{p}_1$        $\times$   
 – Not stable

اگر  ${}_1\mathbf{w}$  برابر  $\mathbf{p}_1$  و یا  $\mathbf{p}_2$  در نظر گرفته شود پاسخ پایداری حاصل نمی شود.

- Add  $\mathbf{p}_1$  to  ${}_1\mathbf{w}$        $\checkmark$

Tentative Rule: If  $t = 1$  and  $a = 0$ , then  ${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}$

$${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}_1 = \begin{bmatrix} 1.0 \\ -0.8 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2.0 \\ 1.2 \end{bmatrix}$$



# Second Input Vector

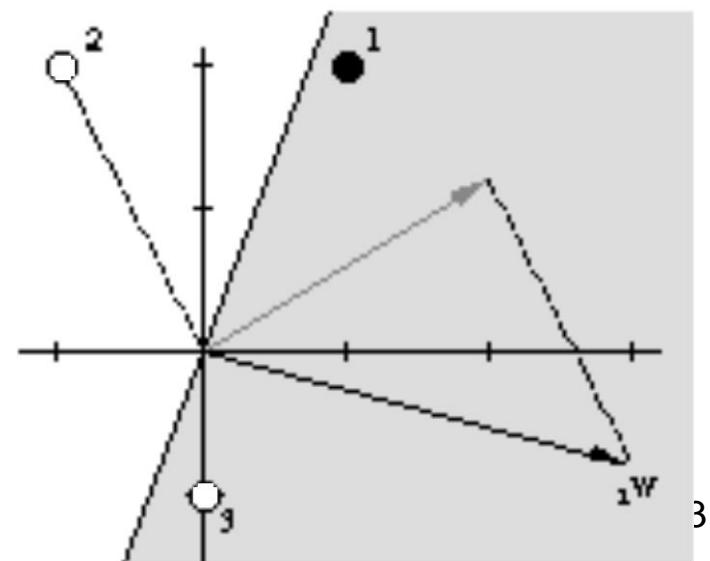
$$a = \text{hardlim}({}_1\mathbf{w}^T \mathbf{p}_2) = \text{hardlim}\left(\begin{bmatrix} 2.0 & 1.2 \end{bmatrix} \begin{bmatrix} -1 \\ 2 \end{bmatrix}\right)$$

$$a = \text{hardlim}(0.4) = 1 \quad (\text{Incorrect Classification})$$

لازم است بردار وزن از  $\mathbf{p}_2$  دور گردد.

Modification to Rule: If  $t = 0$  and  $a = 1$ , then  ${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}$

$${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}_2 = \begin{bmatrix} 2.0 \\ 1.2 \end{bmatrix} - \begin{bmatrix} -1 \\ 2 \end{bmatrix} = \begin{bmatrix} 3.0 \\ -0.8 \end{bmatrix}$$

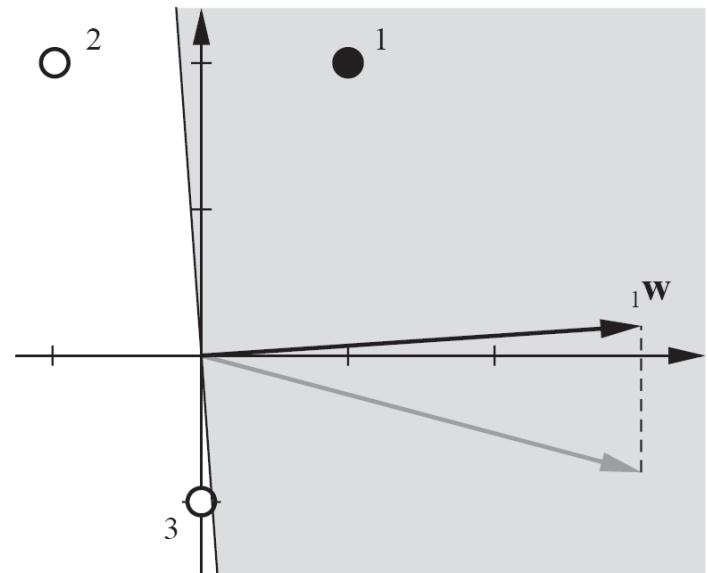


# Third Input Vector

$$a = \text{hardlim}({}_1\mathbf{w}^T \mathbf{p}_3) = \text{hardlim}\left(\begin{bmatrix} 3.0 & -0.8 \end{bmatrix} \begin{bmatrix} 0 \\ -1 \end{bmatrix}\right)$$

$$a = \text{hardlim}(0.8) = 1 \quad (\text{Incorrect Classification})$$

$${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}_3 = \begin{bmatrix} 3.0 \\ -0.8 \end{bmatrix} - \begin{bmatrix} 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 3.0 \\ 0.2 \end{bmatrix}$$



Patterns are now correctly classified.

If  $t = a$ , then  ${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old}$ .

# Unified Learning Rule

مجموعه قوانین

If  $t = 1$  and  $a = 0$ , then  ${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}$

If  $t = 0$  and  $a = 1$ , then  ${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}$

If  $t = a$ , then  ${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old}$

$$e = t - a$$

If  $e = 1$ , then  ${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}$

If  $e = -1$ , then  ${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}$

If  $e = 0$ , then  ${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old}$

Unified LR

$${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + e\mathbf{p} = {}_1\mathbf{w}^{old} + (t - a)\mathbf{p}$$

$$b^{new} = b^{old} + e$$

A bias is a weight with an input of 1.

# Multiple-Neuron Perceptrons

To update the  $i$ th row of the weight matrix:

$${}_i\mathbf{W}^{new} = {}_i\mathbf{W}^{old} + e_i \mathbf{p}$$

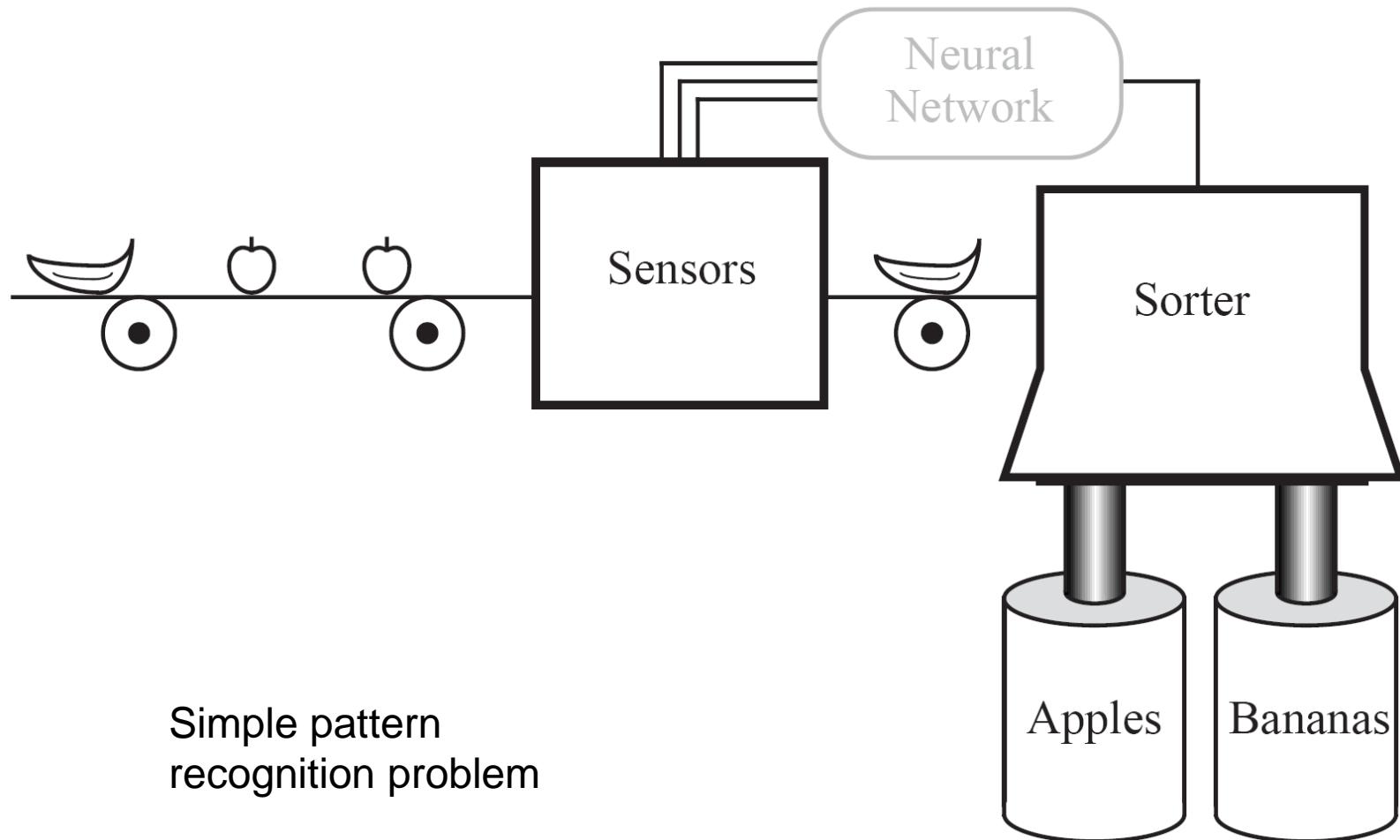
$$b_i^{new} = b_i^{old} + e_i$$

Matrix form:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e} \mathbf{p}^T$$

$$\mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e}$$

# Apple/Banana Example



# Prototype Vectors

Measurement  
Vector

$$\mathbf{p} = \begin{bmatrix} \text{shape} \\ \text{texture} \\ \text{weight} \end{bmatrix}$$

Shape: {1 : round ; -1 : elliptical}

Texture: {1 : smooth ; -1 : rough}

Weight: {1 : > 1 lb. ; -1 : < 1 lb.}

Prototype Banana

$$\mathbf{p}_1 = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}$$

Prototype Apple

$$\mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}$$

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}, t_1 = [1] \right\} \quad \left\{ \mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, t_2 = [0] \right\}$$

## Initial Weights

$$\mathbf{W} = \begin{bmatrix} 0.5 & -1 & -0.5 \end{bmatrix} \quad b = 0.5$$

### First Iteration

$$a = \text{hardlim}(\mathbf{W}\mathbf{p}_1 + b) = \text{hardlim}\left(\begin{bmatrix} 0.5 & -1 & -0.5 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} + 0.5\right)$$

$$a = \text{hardlim}(-0.5) = 0 \quad e = t_1 - a = 1 - 0 = 1$$

$$\mathbf{W}^{new} = \mathbf{W}^{old} + e\mathbf{p}^T = \begin{bmatrix} 0.5 & -1 & -0.5 \end{bmatrix} + (1)\begin{bmatrix} -1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} -0.5 & 0 & -1.5 \end{bmatrix}$$

$$b^{new} = b^{old} + e = 0.5 + (1) = 1.5$$

# Second Iteration

$$a = \text{hardlim}(\mathbf{W}\mathbf{p}_2 + b) = \text{hardlim}(\begin{bmatrix} -0.5 & 0 & -1.5 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + (1.5))$$

$$a = \text{hardlim}(2.5) = 1$$

$$e = t_2 - a = 0 - 1 = -1$$

$$\mathbf{W}^{new} = \mathbf{W}^{old} + e\mathbf{p}^T = \begin{bmatrix} -0.5 & 0 & -1.5 \end{bmatrix} + (-1) \begin{bmatrix} 1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} -1.5 & -1 & -0.5 \end{bmatrix}$$

$$b^{new} = b^{old} + e = 1.5 + (-1) = 0.5$$

# Test

$$a = \text{hardlim}(\mathbf{W}\mathbf{p}_1 + b) = \text{hardlim}(\begin{bmatrix} -1.5 & -1 & -0.5 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} + 0.5)$$

$$a = \text{hardlim}(1.5) = 1 = t_1$$

$$a = \text{hardlim}(\mathbf{W}\mathbf{p}_2 + b) = \text{hardlim}(\begin{bmatrix} -1.5 & -1 & -0.5 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + 0.5)$$

$$a = \text{hardlim}(-1.5) = 0 = t_2$$

Demonstrate program [nnd4pr](#)

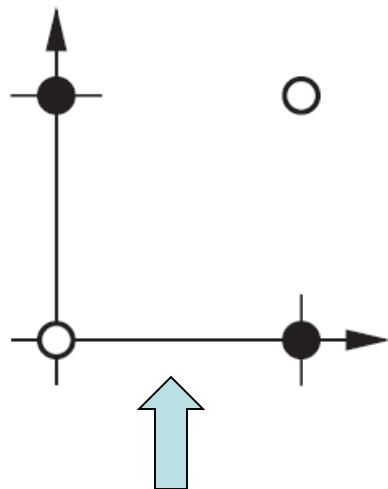
# Guarantee of Success: Novikoff (1963)

**Theorem 2.1:** Given training samples from two linearly separable classes, the perceptron training algorithm terminates after a finite number of steps, and correctly classifies all elements of the training set, irrespective of the initial random non-zero weight vector  $\mathbf{w}_0$ .

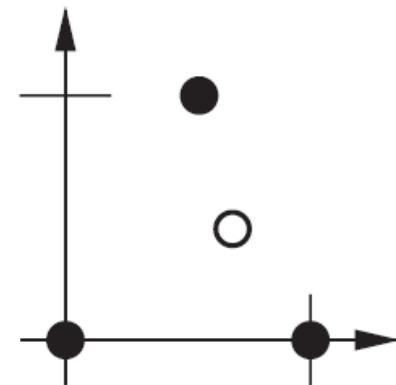
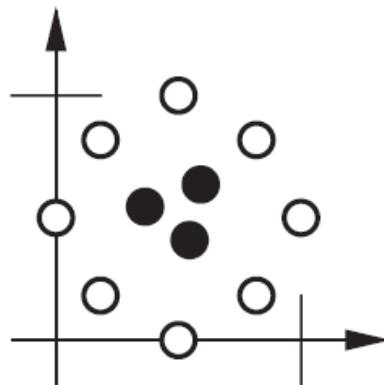
Limitation: Linear Separability تفکیک پذیری خطی

The perceptron can be used to classify input vectors that can be separated by a linear boundary.

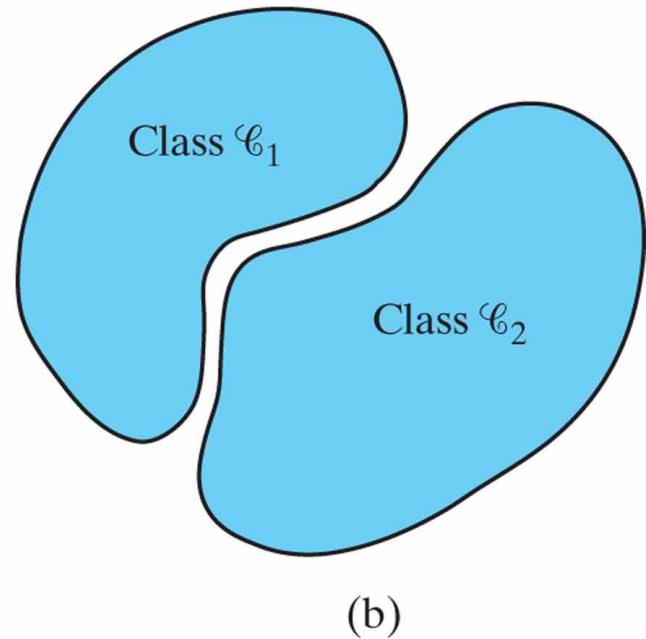
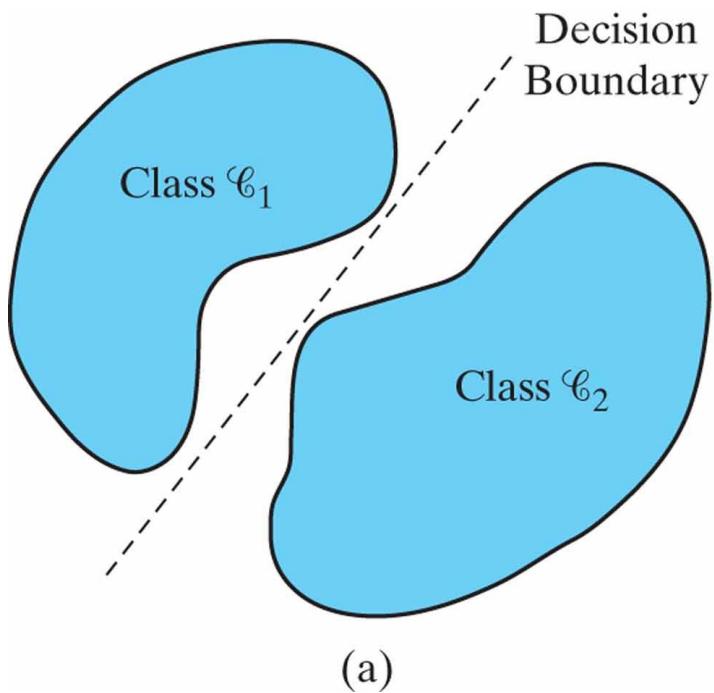
# Linearly Inseparable Problem



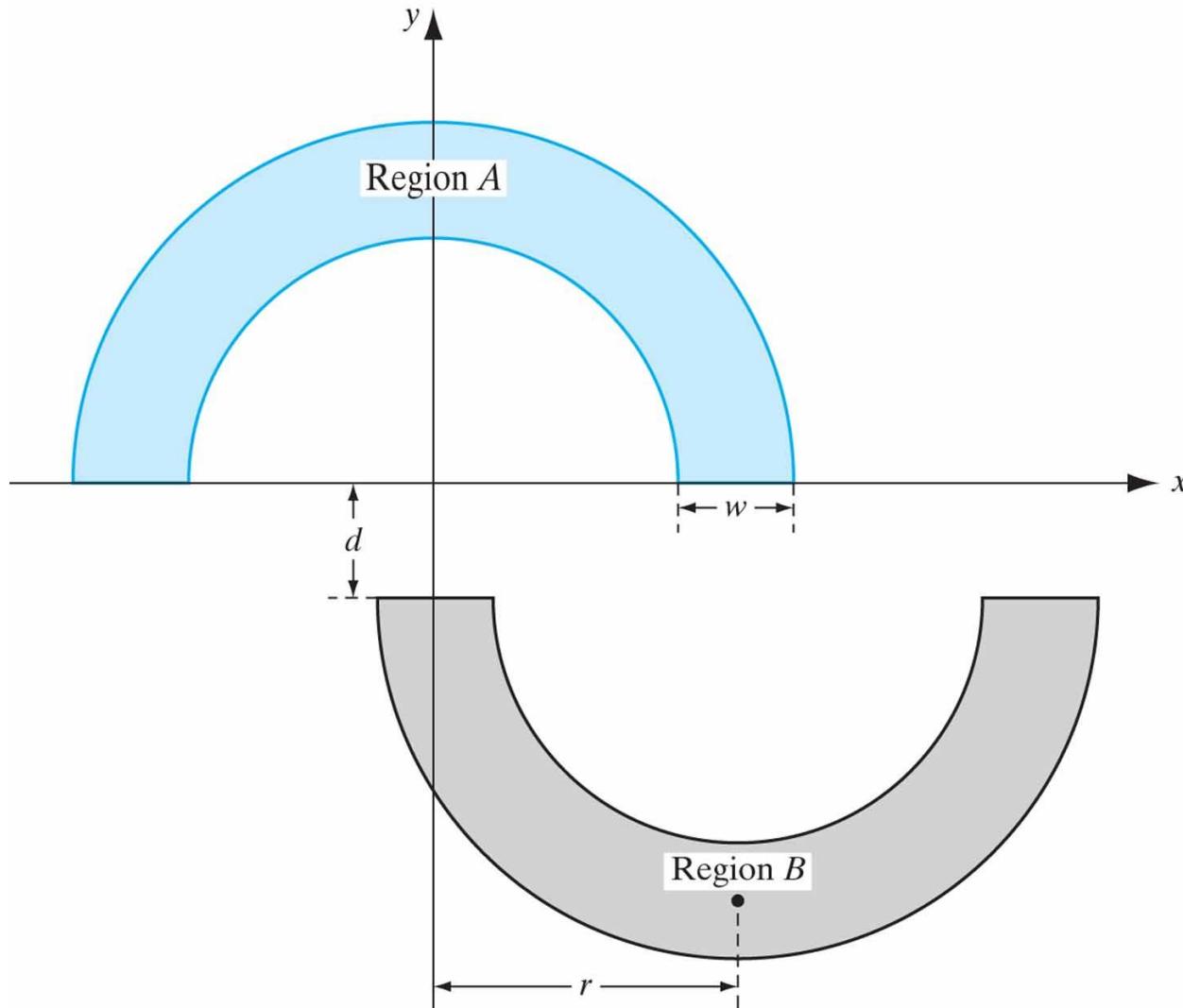
XOR function



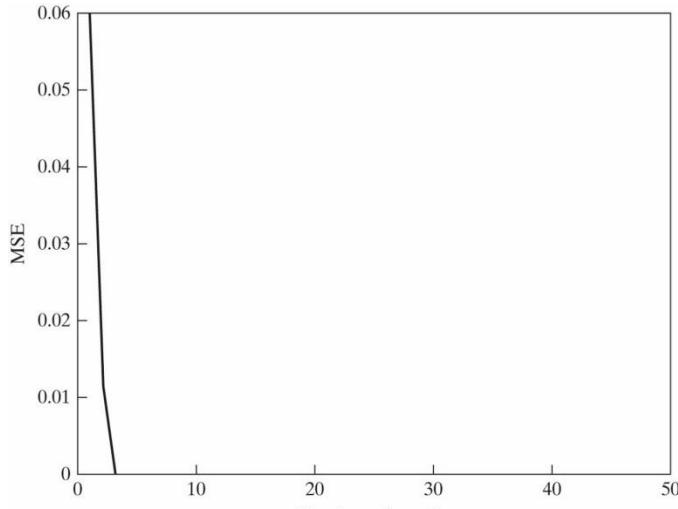
**Figure 1.4** (a) A pair of linearly separable patterns. (b) A pair of non-linearly separable.



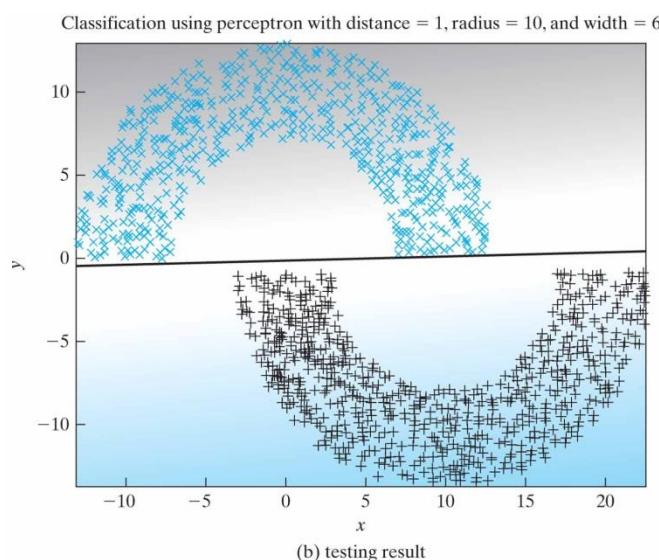
## The double-moon classification problem.



## Perceptron with the double-moon set at distance $d = 1$ .

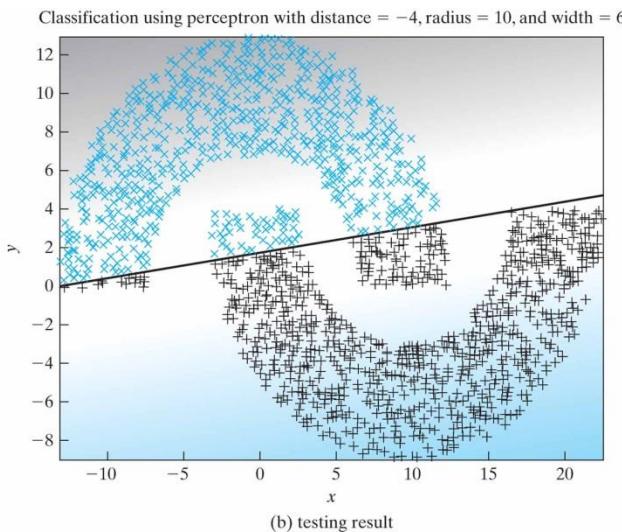
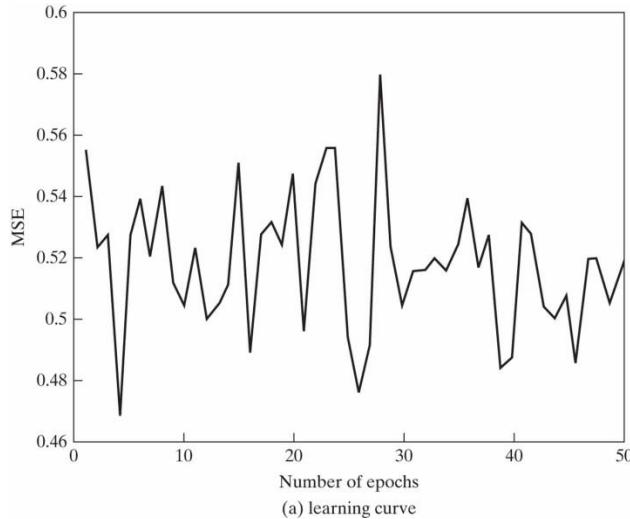


(a) learning curve



(b) testing result

## Perceptron with the double-moon set at distance $d = -4$ .

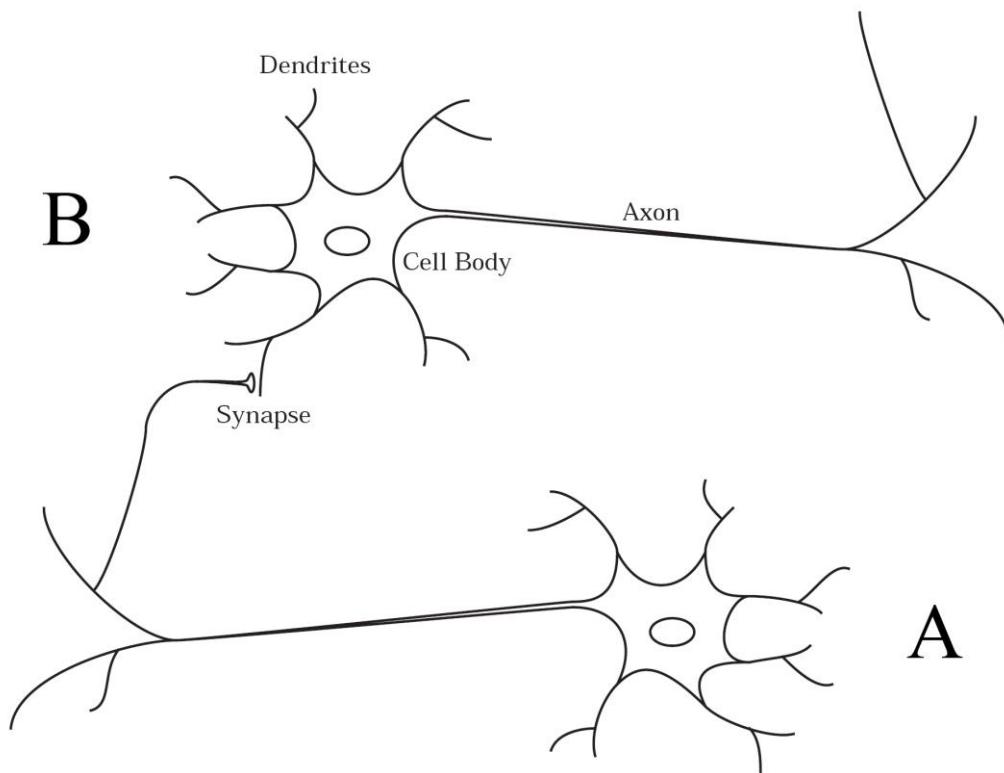


# Supervised Hebbian Learning

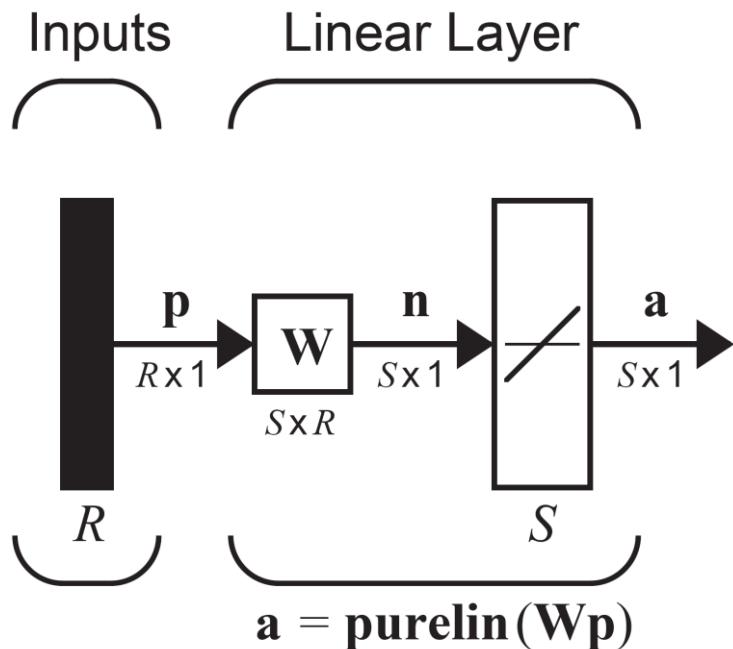
# Hebb's Postulate

“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 change takes place in one or both cells such that *A*’s efficiency, as one of the cells firing *B*, is increased.”

D. O. Hebb, 1949



# Linear Associator



$$\mathbf{a} = \mathbf{W}\mathbf{p} \quad a_i = \sum_{j=1}^R w_{ij}p_j$$

Training Set:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \dots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

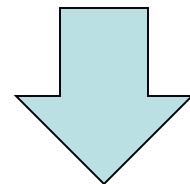
- LA introduced by Kohonen and Anderson independently.
- LA is an example of **Associative Memory**

# LEARNING TASKS

- **Pattern Association** تداعی معانی-اتحاد-پیوستگی
  - An associative memory is a brain like memory that learns by association:  
If  $\mathbf{p} = \mathbf{p}_q$  (input)  $\rightarrow \mathbf{a} = \mathbf{t}_q$  for  $q = 1, 2, \dots, Q$  & if  $\mathbf{p} = \mathbf{p}_q + \boldsymbol{\delta}$   
 $\rightarrow \mathbf{a} = \mathbf{t}_q$
  - Association takes one of two forms:  
**autoassociation** and **heteroassociation**
  - In autoassociation, a neural network is required to store a set of patterns (vectors) by repeatedly presenting them to the network. معمولاً یادگیری غیرنظرارتی

- Heteroassociation differs from auto association in that an arbitrary set of input patterns is paired with another arbitrary set of output patterns.

یادگیری نظارتی



مبحث این فصل

# Hebb Rule (1)

$$w_{ij}^{new} = w_{ij}^{old} + \alpha f_i(a_{iq})g_j(p_{jq})$$

- $p_{jq}$  is the  $j$ th element of the  $q$ th input vector  $\mathbf{p}_q$ .
- $a_{iq}$  is the  $i$ th element of the network output when the  $q$ th input vector is presented to the network.
- $\alpha$  is a positive constant, called learning rate.
- Changes in  $w_{ij}$  is proportional to a product of functions of the activities on either side of synapse.

# Hebb Rule (2)

Simplified Form:

$$w_{ij}^{new} = w_{ij}^{old} + \alpha a_{iq} p_{jq}$$

- This extends Hebb's postulate (we also increase the weight when both  $p_j$  and  $a_i$  negative and decrease when one of them is negative).
- The above rule is an unsupervised learning rule (Ch 15).

Supervised Form:

$$w_{ij}^{new} = w_{ij}^{old} + t_{iq} p_{jq}$$

- $t_{iq}$  is the  $i$ th element of the  $q$ th target vector  $\mathbf{t}_q$ . ( $\alpha=1$  for simplicity.)
- We are telling the algorithm what the network should do, rather than what it is currently doing.

# Batch Operation

Matrix Form  
of Rule:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{t}_q \mathbf{p}_q^T$$

When each of the  $Q$  input/output pair are applied once we have:

$$\mathbf{W} = \mathbf{t}_1 \mathbf{p}_1^T + \mathbf{t}_2 \mathbf{p}_2^T + \dots + \mathbf{t}_Q \mathbf{p}_Q^T = \sum_{q=1}^Q \mathbf{t}_q \mathbf{p}_q^T \quad (\text{Zero Initial Weights})$$

Matrix Form:

$$\mathbf{W} = [\mathbf{t}_1 \ \mathbf{t}_2 \ \dots \ \mathbf{t}_Q] \begin{bmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \\ \vdots \\ \mathbf{p}_Q^T \end{bmatrix} = \mathbf{T} \mathbf{P}^T$$
$$\mathbf{P} = [\mathbf{p}_1 \ \mathbf{p}_2 \ \dots \ \mathbf{p}_Q]$$
$$\mathbf{T} = [\mathbf{t}_1 \ \mathbf{t}_2 \ \dots \ \mathbf{t}_Q]$$

# Performance Analysis

Hebbian learning for the linear Associator. Assume  $\mathbf{p}_q$  vectors are *orthonormal*. If  $\mathbf{p}_k$  is input to the network, the output will be:

$$\mathbf{a} = \mathbf{W}\mathbf{p}_k = \left( \sum_{q=1}^Q \mathbf{t}_q \mathbf{p}_q^T \right) \mathbf{p}_k = \sum_{q=1}^Q \mathbf{t}_q (\mathbf{p}_q^T \mathbf{p}_k)$$

**Case I**, since input patterns are orthonormal.

$$(\mathbf{p}_q^T \mathbf{p}_k) = 1 \quad q = k \\ = 0 \quad q \neq k$$

Therefore the network output equals the target:  $\mathbf{a} = \mathbf{W}\mathbf{p}_k = \mathbf{t}_k$

**Case II**, input patterns are normalized, but not orthogonal.

$$\mathbf{a} = \mathbf{W}\mathbf{p}_k = \mathbf{t}_k + \boxed{\sum_{q \neq k} \mathbf{t}_q (\mathbf{p}_q^T \mathbf{p}_k)} \quad \text{Error}$$

The magnitude of the error will depend on the amount of correlation between the prototype input patterns.

# Example

Suppose that the prototype input/output vectors are

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} 0.5 \\ -0.5 \\ 0.5 \\ -0.5 \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \right\} \quad \left\{ \mathbf{p}_2 = \begin{bmatrix} 0.5 \\ 0.5 \\ -0.5 \\ -0.5 \end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}.$$

Two input vectors are orthonormal. The weight matrix would be

$$\mathbf{W} = \mathbf{T}\mathbf{P}^T = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 0.5 & -0.5 & 0.5 & -0.5 \\ 0.5 & 0.5 & -0.5 & -0.5 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \end{bmatrix}.$$

If we test this weight matrix on the two prototype inputs we find

$$\mathbf{W}\mathbf{p}_1 = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} 0.5 \\ -0.5 \\ 0.5 \\ -0.5 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix},$$

and

$$\mathbf{W}\mathbf{p}_2 = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} 0.5 \\ 0.5 \\ -0.5 \\ -0.5 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

Success!! The outputs of the network are equal to the targets.

# Example

|  |   |   |
|--|---|---|
| Banana   | Apple   | Normalized Prototype Patterns   |
| $\mathbf{p}_1 = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}$ | $\mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}$ | $\left\{ \mathbf{p}_1 = \begin{bmatrix} -0.5774 \\ 0.5774 \\ -0.5774 \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} -1 \end{bmatrix} \right\}$ $\left\{ \mathbf{p}_2 = \begin{bmatrix} 0.5774 \\ 0.5774 \\ -0.5774 \end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix} 1 \end{bmatrix} \right\}$ |

Weight Matrix (Hebb Rule):

$$\mathbf{W} = \mathbf{T}\mathbf{P}^T = \begin{bmatrix} -1 & 1 \end{bmatrix} \begin{bmatrix} -0.5774 & 0.5774 & -0.5774 \\ 0.5774 & 0.5774 & -0.5774 \end{bmatrix} = \begin{bmatrix} 1.1548 & 0 \end{bmatrix}$$

Tests:

$$\text{Banana } \mathbf{W}\mathbf{p}_1 = \begin{bmatrix} 1.1548 & 0 \end{bmatrix} \begin{bmatrix} -0.5774 \\ 0.5774 \\ -0.5774 \end{bmatrix} = \begin{bmatrix} -0.6668 \end{bmatrix}$$

$$\text{Apple } \mathbf{W}\mathbf{p}_2 = \begin{bmatrix} 0 & 1.1548 \end{bmatrix} \begin{bmatrix} 0.5774 \\ 0.5774 \\ -0.5774 \end{bmatrix} = \begin{bmatrix} 0.6668 \end{bmatrix}$$

The output are close, but do not quite match the target outputs<sub>12</sub>

# Pseudoinverse Rule

To reduce the error when the prototype input patterns are not orthogonal we may use pseudoinverse rule (among several procedures).

Recall that the task of the linear associator was to produce an output of  $\mathbf{t}_q$  for an input of  $\mathbf{p}_q$ . In other words

$$\mathbf{W}\mathbf{p}_q = \mathbf{t}_q \quad q = 1, 2, \dots, Q$$

If it is not possible to choose a weight matrix so that these equations are exactly satisfied, then we want them to be approximately satisfied.

Performance Index: (to be minimized)  $F(\mathbf{W}) = \sum_{q=1}^Q \|\mathbf{t}_q - \mathbf{W}\mathbf{p}_q\|^2$

When the input vectors are not orthogonal and we use the Hebb rule, then  $F(\mathbf{W})$  will not be zero, and it is not clear that  $F(\mathbf{W})$  will be minimized. So we obtain  $\mathbf{W}$  by pseudoinverse matrix.

Matrix Form:

$$\mathbf{T} = [\mathbf{t}_1 \ \mathbf{t}_2 \ \dots \ \mathbf{t}_Q] \quad \mathbf{P} = [\mathbf{p}_1 \ \mathbf{p}_2 \ \dots \ \mathbf{p}_Q]$$

$$F(\mathbf{W}) = \|\mathbf{T} - \mathbf{WP}\|^2 = \|\mathbf{E}\|^2$$

$$\|\mathbf{E}\|^2 = \sum_i \sum_j e_{ij}^2$$

Note that  $F(\mathbf{W})$  can be made zero if we can solve  $\mathbf{WP} = \mathbf{T}$

If an inverse exists for  $\mathbf{P}$ ,  $F(\mathbf{W})$  can be made zero:

$$\mathbf{W} = \mathbf{T}\mathbf{P}^{-1}$$

However, this is rarely possible. Normally the  $\mathbf{p}_q$  vectors (the columns of  $\mathbf{P}$ ) will be independent, but  $R$  (the dimension of  $\mathbf{p}_q$ ) will be larger than  $Q$  (the number of  $\mathbf{p}_q$  vectors). Therefore,  $\mathbf{P}$  will not be a square matrix, and no exact inverse will exist.

# Pseudoinverse Rule

When an inverse does not exist  $F(\mathbf{W})$  can be minimized using the pseudoinverse [Albert 72] :

$$\mathbf{W} = \mathbf{T}\mathbf{P}^+$$

where  $\mathbf{P}^+$  is the Moore-Penrose pseudoinverse.

A real matrix  $\mathbf{P}$  has a unique  
pseudoinverse that satisfies:

$$\mathbf{P}\mathbf{P}^+\mathbf{P}=\mathbf{P}$$

$$\mathbf{P}^+\mathbf{P}\mathbf{P}^+=\mathbf{P}^+$$

$$\mathbf{P}^+\mathbf{P}=(\mathbf{P}^+\mathbf{P})^T$$

$$\mathbf{P}\mathbf{P}^+=(\mathbf{P}\mathbf{P}^+)^T$$

# Pseudoinverse Rule

If  $\mathbf{P}$  is  $R \times Q$  and  $R > Q$  and the columns of  $\mathbf{P}$  are independent, then the pseudoinverse can be computed by

$$\mathbf{P}^+ = (\mathbf{P}^T \mathbf{P})^{-1} \mathbf{P}^T$$

If  $\mathbf{P}$  is  $R \times Q$  and  $R < Q$  and the columns of  $\mathbf{P}$  are independent, then the pseudoinverse can be computed by

$$\mathbf{P}^+ = \mathbf{P}^T (\mathbf{P} \mathbf{P}^T)^{-1}$$

# Relationship To The Hebb Rule

Hebb Rule

$$\mathbf{W} = \mathbf{T}\mathbf{P}^T$$

Pseudoinverse Rule:

$$\mathbf{W} = \mathbf{T}\mathbf{P}^+$$

$$\mathbf{P}^+ = (\mathbf{P}^T \mathbf{P})^{-1} \mathbf{P}^T$$

If the prototype patterns are orthonormal:

$$\mathbf{P}^T \mathbf{P} = \mathbf{I}$$

$$\mathbf{P}^+ = (\mathbf{P}^T \mathbf{P})^{-1} \mathbf{P}^T = \mathbf{P}^T$$

# Example

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \right\} \quad \left\{ \mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\} \quad \mathbf{W} = \mathbf{T}\mathbf{P}^+ = \begin{bmatrix} -1 & 1 \\ 1 & 1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} -1 & 1 \\ 1 & 1 \\ -1 & -1 \end{pmatrix}^+$$

**Note: normalizing the input are not needed.**

$$\mathbf{P}^+ = (\mathbf{P}^T \mathbf{P})^{-1} \mathbf{P}^T = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}^{-1} \begin{bmatrix} -1 & 1 & -1 \\ 1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} -0.5 & 0.25 & -0.25 \\ 0.5 & 0.25 & -0.25 \end{bmatrix}$$

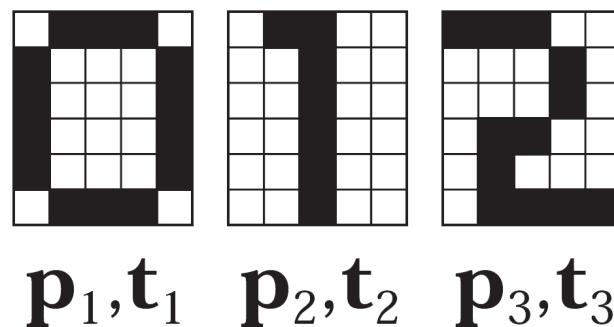
$$\mathbf{W} = \mathbf{T}\mathbf{P}^+ = \begin{bmatrix} -1 & 1 \end{bmatrix} \begin{bmatrix} -0.5 & 0.25 & -0.25 \\ 0.5 & 0.25 & -0.25 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$

$$\mathbf{W}\mathbf{p}_1 = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} -1 \end{bmatrix} \quad \mathbf{W}\mathbf{p}_2 = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 \end{bmatrix}$$

The network outputs exactly match the desired outputs.

# Autoassociative Memory

- In an Autoassociative memory the desired output vector is equal to the input vector (i.e.  $\mathbf{t}_q = \mathbf{p}_q$ ).
- An Autoassociative memory stores a set of patterns. We can recall these patterns even when corrupted patterns are provided as input.



$$\mathbf{p}_1 = \begin{bmatrix} -1 & 1 & 1 & 1 & 1 & -1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & -1 & \dots & 1 & -1 \end{bmatrix}^T$$

# Cont.

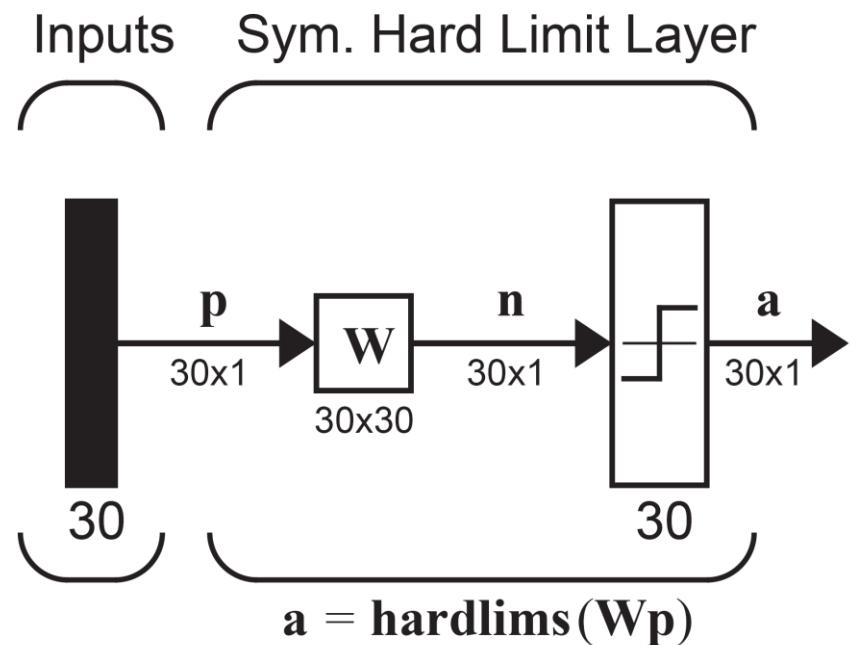
Using the Hebb rule we have:  $\mathbf{W} = \mathbf{p}_1\mathbf{p}_1^T + \mathbf{p}_2\mathbf{p}_2^T + \mathbf{p}_3\mathbf{p}_3^T$

Note that  $\mathbf{p}_q$  replaces  $\mathbf{t}_q$  in the related equation,

$$\mathbf{W} = \mathbf{t}_1\mathbf{p}_1^T + \mathbf{t}_2\mathbf{p}_2^T + \dots + \mathbf{t}_Q\mathbf{p}_Q^T = \sum_{q=1}^Q \mathbf{t}_q\mathbf{p}_q^T$$

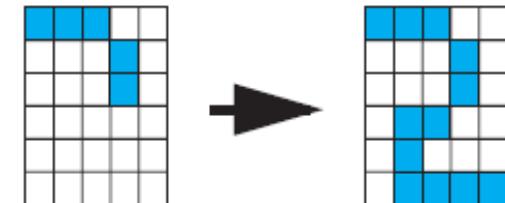
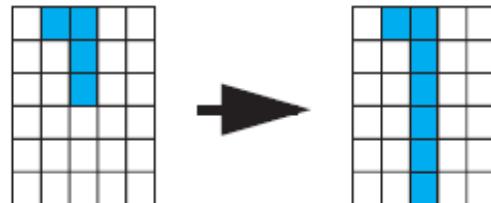
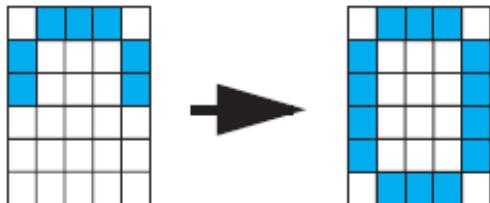
since this is Autoassociative memory.

Autoassociative Net for digit recog. Note the Linear Associator has been modified so that output can only take on values of -1 or 1.

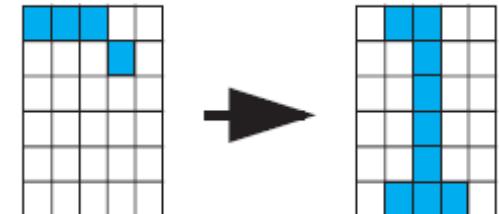
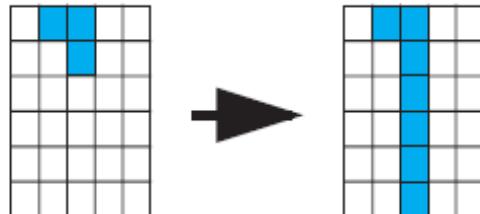
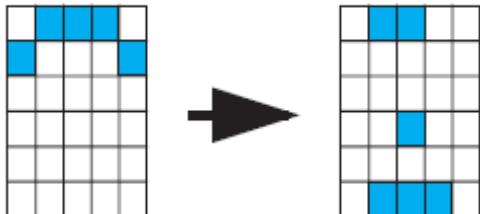


# Tests

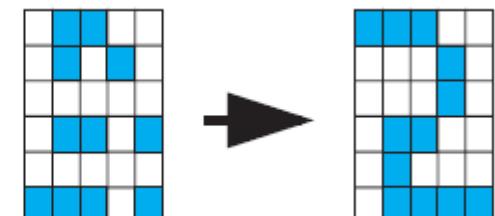
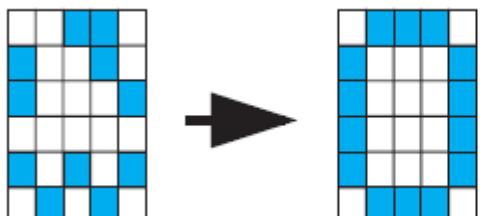
50% Occluded



67% Occluded



Noisy Patterns (7 pixels)



## مثال

- بازشناسی نویسه (کاراکتر)- الگوهای ورودی دو بعدی ...

- یک شبکه هب برای تشخیص الگوی «X» از الگوی «O»

- ## • يك مسئلة دسته بندى الگو؟

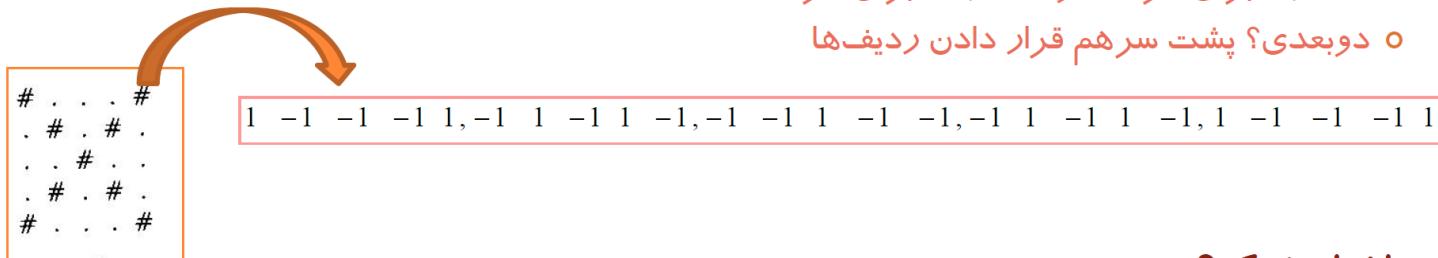
٥ دسته « $X$ » = خروجي مورد نظر و الگوي « $O$ » = خروجي «غير

۰ روش دیگر؟

- تبدیل الگوهای «O» و «X» به بردارهای ورودی؟

۰ مقدار ۱ برای هر «#» و مقدار ۱- برای هر «۰

## ۰ دو بعدی؟ پشت سر هم قرار دادن ردیف‌ها



- ساختار شبکه؟

## ٥ تعداد نرون‌های ورودی = برابر با تعداد ابعاد بردار الگوها = ٢٥

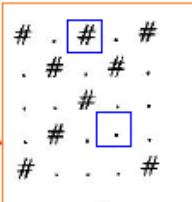
- قابلیت تعمیم شبکه

- تولید پاسخ منطقی شبکه برای الگوهای ورودی شبیه الگوهای آموزش اما نه کاملاً یکسان با آنها

- دو نوع تغییر در الگوی ورودی

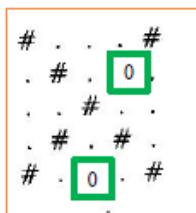
- «اشتباهات در داده‌ها»

علامت یک یا چند مؤلفه بردار ورودی قرینه شده و از ۱ به -۱، یا برعکس، ت.



- «داده‌های گم شده»

یک یا چند مؤلفه بردار ورودی به جای مقدار ۱ یا -۱ مقدار صفر دارند



# Variations of Hebbian Learning

Basic Rule:  $\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{t}_q \mathbf{p}_q^T$

Learning Rate:  $\mathbf{W}^{new} = \mathbf{W}^{old} + \alpha \mathbf{t}_q \mathbf{p}_q^T$

Forgetting factor:  $\mathbf{W}^{new} = \mathbf{W}^{old} + \alpha \mathbf{t}_q \mathbf{p}_q^T - \gamma \mathbf{W}^{old} = (1 - \gamma) \mathbf{W}^{old} + \alpha \mathbf{t}_q \mathbf{p}_q^T$

where  $\gamma$  is a positive constant less than one. As  $\gamma$  approaches zero, the learning law becomes the standard rule. As  $\gamma$  approaches one, the learning law quickly forgets old inputs and remembers only the most recent patterns. This keeps the weight matrix from growing without bound.

# Variations of Hebbian Learning

Delta Rule:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \alpha(\mathbf{t}_q - \mathbf{a}_q)\mathbf{p}_q^T$$

It is also known as the Widrow-Hoff algorithm, after the researchers who introduced it.

The delta rule adjusts the weights so as to minimize the mean square error (Ch. 10). For this reason it will produce the same results as the pseudoinverse rule, which minimizes the sum of squares of errors.

# Cont'

The advantage of the delta rule is that it can update the weights after each new input pattern is presented, whereas the Pseudo-inverse rule computes the weights in one step, after all of the input/target pairs are known. This sequential updating allows the delta rule to adapt to a changing environment.

Unsupervised:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \alpha \mathbf{a}_q \mathbf{p}_q^T$$

# **Performance Surfaces and Optimum Points**

# Classes Of Network Learning Laws

- Associative learning (as Hebbian learning of Ch 7,15).
- Competitive learning ( SOFM, LVQ, Ch 16).
- Performance learning (parameters are adjusted to optimize the performance of the network).(Ch10-14)
- .....
  - In this chapter we investigate performance surfaces and the conditions for the existence of minima and maxima of the performance surface. Why?
  - In some networks parameters ( $W \& b$ ) are adjusted in an effort to optimize the “*performance*” of the network.
    - Performance index to be defined
    - Search the parameter space (adjust the net weights and biases) in order to reduce the performance index.

# Taylor Series Expansion

- Let  $F(x)$  be the performance index.
- We assume the performance index is an analytic function.
- It can be represented by its Taylor series expansion (to approximate performance index, by limiting the expansion to a finite number of terms).

$$F(x) = F(x^*) + \frac{d}{dx}F(x) \Big|_{x=x^*} (x - x^*)$$

$$+ \frac{1}{2} \frac{d^2}{dx^2} F(x) \Bigg|_{x=x^*} (x - x^*)^2 + \dots$$

$$+ \frac{1}{n!} \frac{d^n}{dx^n} F(x) \Bigg|_{x=x^*} (x - x^*)^n + \dots$$

# Example

$$F(x) = e^{-x}$$

Taylor series of  $F(x)$  about  $x^*=0$ :

$$F(x) = e^{-x} = e^{-0} - e^{-0}(x-0) + \frac{1}{2}e^{-0}(x-0)^2 - \frac{1}{6}e^{-0}(x-0)^3 + \dots$$

$$F(x) = 1 - x + \frac{1}{2}x^2 - \frac{1}{6}x^3 + \dots$$

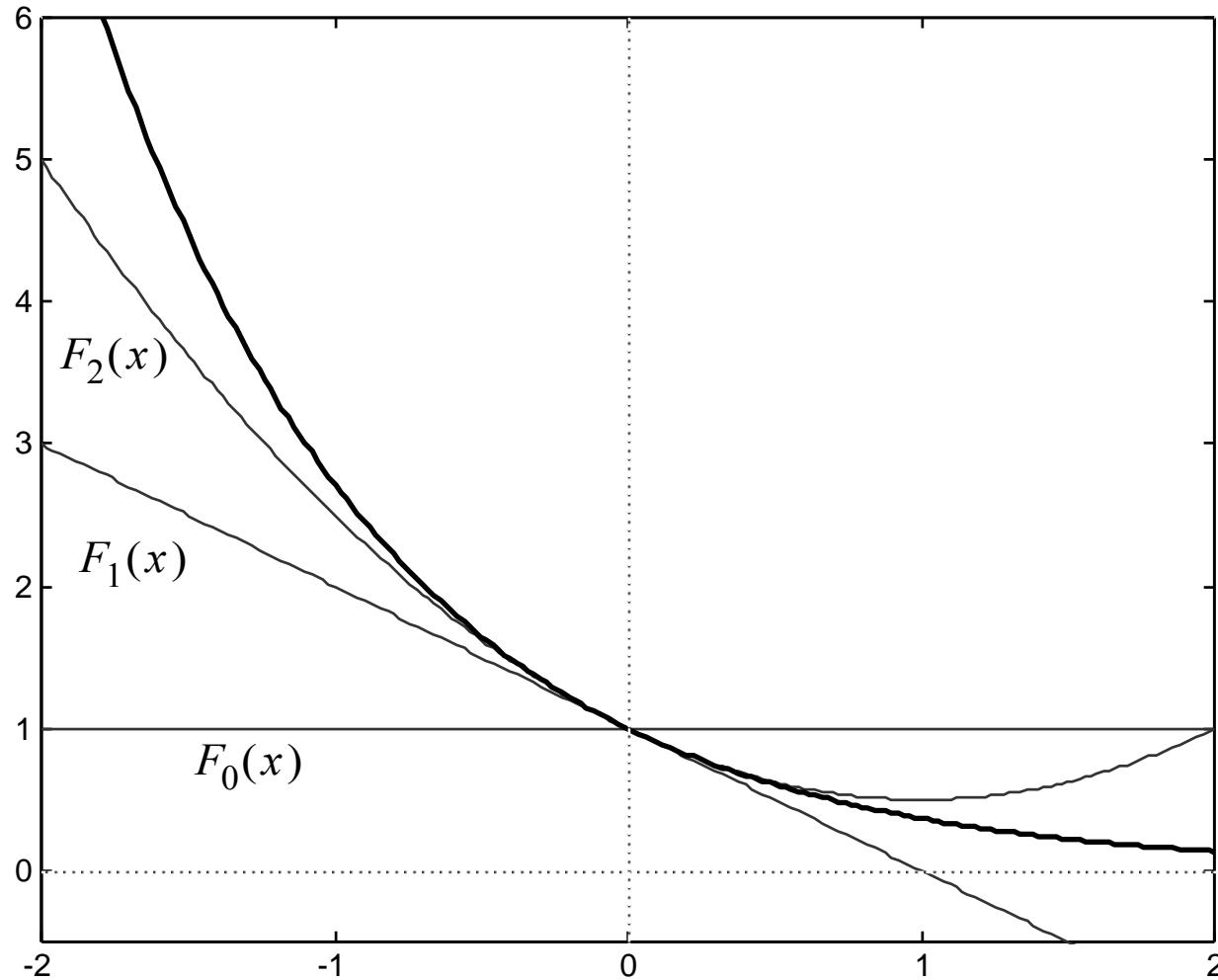
Taylor series approximations:

$$F(x) \approx F_0(x) = 1$$

$$F(x) \approx F_1(x) = 1 - x$$

$$F(x) \approx F_2(x) = 1 - x + \frac{1}{2}x^2$$

# Plot of Approximations



High order approximation is accurate over a wider range than the low order approximation. [nnd8ts1](#)

# Vector Case

$$F(\mathbf{x}) = F(x_1, x_2, \dots, x_n)$$

$$F(\mathbf{x}) = F(\mathbf{x}^*) + \frac{\partial}{\partial x_1} F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}^*} (x_1 - x_1^*) + \frac{\partial}{\partial x_2} F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}^*} (x_2 - x_2^*)$$

$$+ \dots + \frac{\partial}{\partial x_n} F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}^*} (x_n - x_n^*) + \frac{1}{2} \frac{\partial^2}{\partial x_1^2} F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}^*} (x_1 - x_1^*)^2$$

$$+ \frac{1}{2} \frac{\partial^2}{\partial x_1 \partial x_2} F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}^*} (x_1 - x_1^*)(x_2 - x_2^*) + \dots$$

# Matrix Form

$$F(\mathbf{x}) = F(\mathbf{x}^*) + \nabla F(\mathbf{x})^T \Big|_{\mathbf{x} = \mathbf{x}^*} (\mathbf{x} - \mathbf{x}^*)$$

$$+ \frac{1}{2} (\mathbf{x} - \mathbf{x}^*)^T \nabla^2 F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}^*} (\mathbf{x} - \mathbf{x}^*) + \dots$$

Gradient

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial x_n} F(\mathbf{x}) \end{bmatrix}$$

[nnd8ts2](#)

Hessian

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2}{\partial x_1^2} F(\mathbf{x}) & \frac{\partial^2}{\partial x_1 \partial x_2} F(\mathbf{x}) & \dots & \frac{\partial^2}{\partial x_1 \partial x_n} F(\mathbf{x}) \\ \frac{\partial^2}{\partial x_2 \partial x_1} F(\mathbf{x}) & \frac{\partial^2}{\partial x_2^2} F(\mathbf{x}) & \dots & \frac{\partial^2}{\partial x_2 \partial x_n} F(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2}{\partial x_n \partial x_1} F(\mathbf{x}) & \frac{\partial^2}{\partial x_n \partial x_2} F(\mathbf{x}) & \dots & \frac{\partial^2}{\partial x_n^2} F(\mathbf{x}) \end{bmatrix}$$

# Directional Derivatives

First derivative (slope) of  $F(\mathbf{x})$  along  $x_i$  axis:

$$\partial F(\mathbf{x}) / \partial x_i$$

( $i$ th element of gradient)

Second derivative (curvature) of  $F(\mathbf{x})$  along  $x_i$  axis:

$$\partial^2 F(\mathbf{x}) / \partial x_i^2$$

( $i,i$  element of Hessian)

First derivative (slope) of  $F(\mathbf{x})$  along vector  $\mathbf{p}$ :

$$\frac{\mathbf{p}^T \nabla F(\mathbf{x})}{\|\mathbf{p}\|}$$

Second derivative (curvature) of  $F(\mathbf{x})$  along vector  $\mathbf{p}$ :

$$\frac{\mathbf{p}^T \nabla^2 F(\mathbf{x}) \mathbf{p}}{\|\mathbf{p}\|^2}$$

# Example

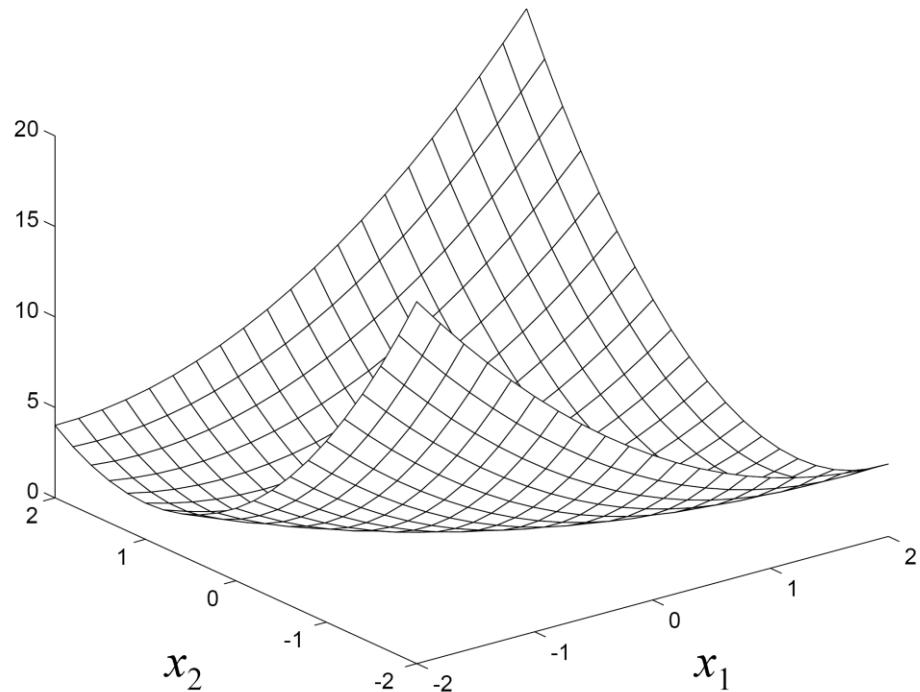
$$F(\mathbf{x}) = x_1^2 + 2x_1x_2 + 2x_2^2$$

$$\mathbf{x}^* = \begin{bmatrix} 0.5 \\ 0 \end{bmatrix} \quad \mathbf{p} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

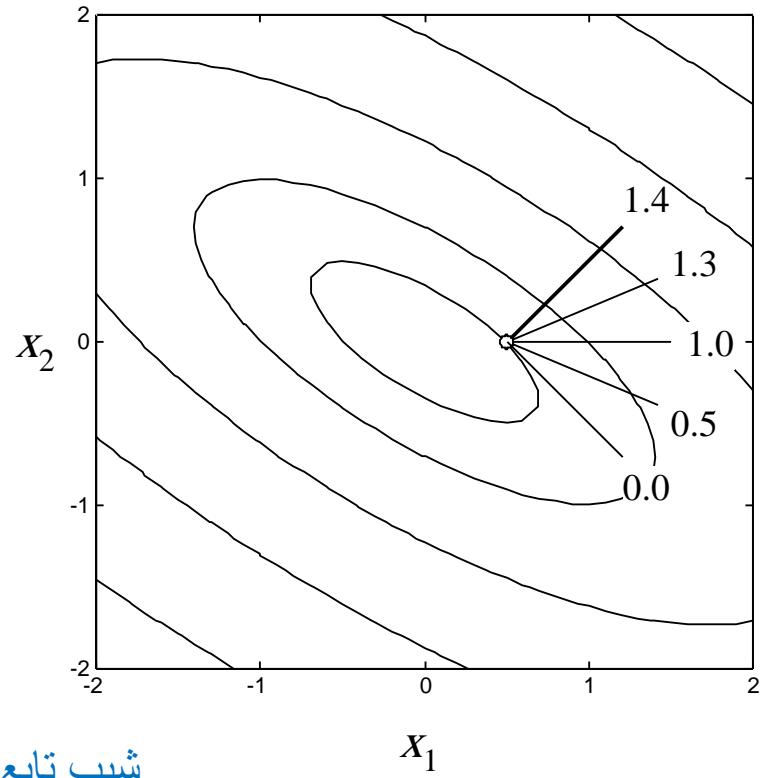
$$\nabla F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}^*} = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} \Bigg|_{\mathbf{x} = \mathbf{x}^*} = \begin{bmatrix} 2x_1 + 2x_2 \\ 2x_1 + 4x_2 \end{bmatrix} \Bigg|_{\mathbf{x} = \mathbf{x}^*} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\frac{\mathbf{p}^T \nabla F(\mathbf{x})}{\|\mathbf{p}\|} = \frac{\begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}}{\sqrt{\left\| \begin{bmatrix} 1 \\ -1 \end{bmatrix} \right\|^2}} = \frac{\begin{bmatrix} 0 \end{bmatrix}}{\sqrt{2}} = 0$$

# Plots



Directional  
Derivatives



شیب تابع در جهت گرادیان، ماکزیمم و در جهت عمود بر آن، صفر است.

nnd8dd

# Minima

## Strong Minimum

The point  $\mathbf{x}^*$  is a strong minimum of  $F(\mathbf{x})$  if a scalar  $\delta > 0$  exists, such that  $F(\mathbf{x}^*) < F(\mathbf{x}^* + \Delta\mathbf{x})$  for all  $\Delta\mathbf{x}$  such that  $\delta > \|\Delta\mathbf{x}\| > 0$ .

## Global Minimum

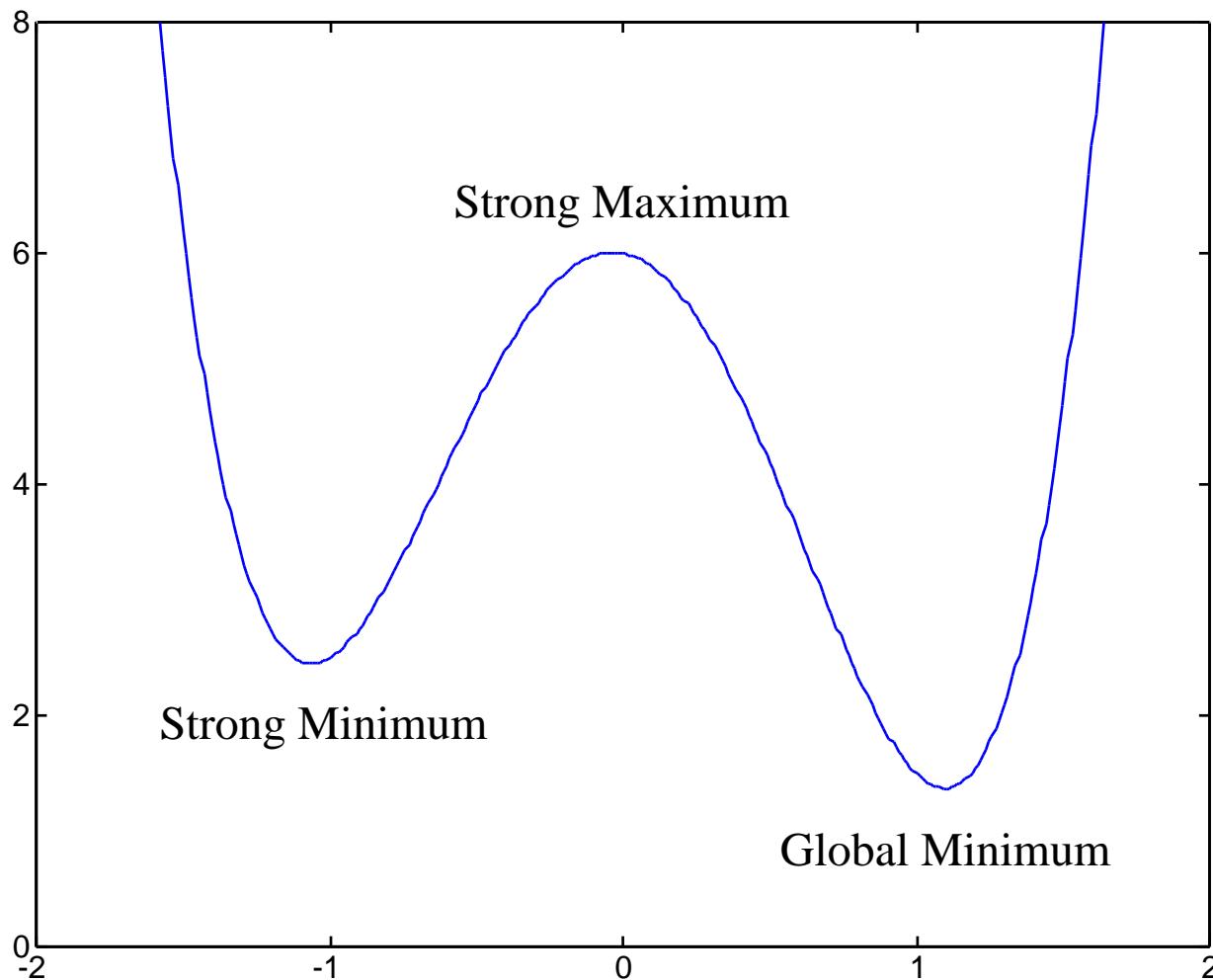
The point  $\mathbf{x}^*$  is a unique global minimum of  $F(\mathbf{x})$  if  $F(\mathbf{x}^*) < F(\mathbf{x}^* + \Delta\mathbf{x})$  for all  $\Delta\mathbf{x} \neq 0$ .

## Weak Minimum

The point  $\mathbf{x}^*$  is a weak minimum of  $F(\mathbf{x})$  if it is not a strong minimum, and a scalar  $\delta > 0$  exists, such that  $F(\mathbf{x}^*) \leq F(\mathbf{x}^* + \Delta\mathbf{x})$  for all  $\Delta\mathbf{x}$  such that  $\delta > \|\Delta\mathbf{x}\| > 0$ .

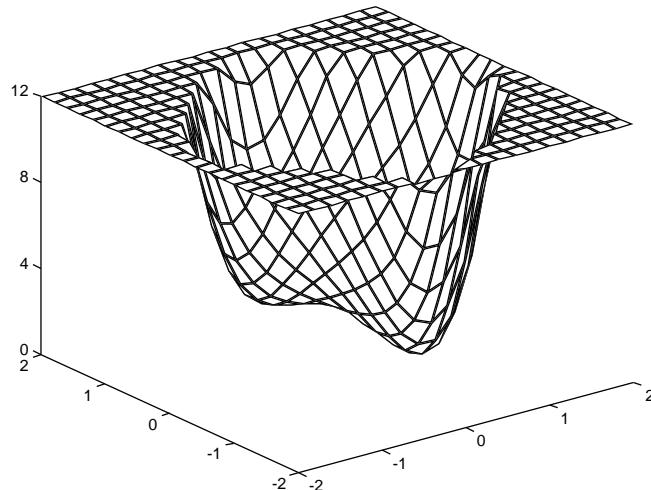
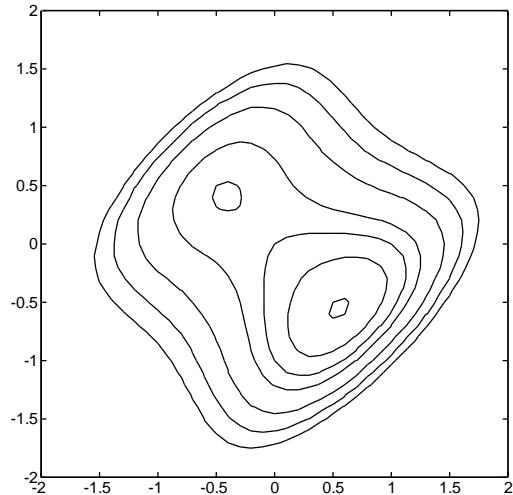
# Scalar Example

$$F(x) = 3x^4 - 7x^2 - \frac{1}{2}x + 6$$



# Vector Example (1)

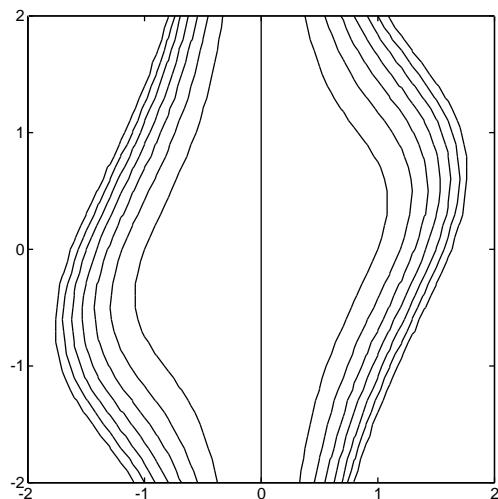
$$F(\mathbf{x}) = (x_2 - x_1)^4 + 8x_1x_2 - x_1 + x_2 + 3$$



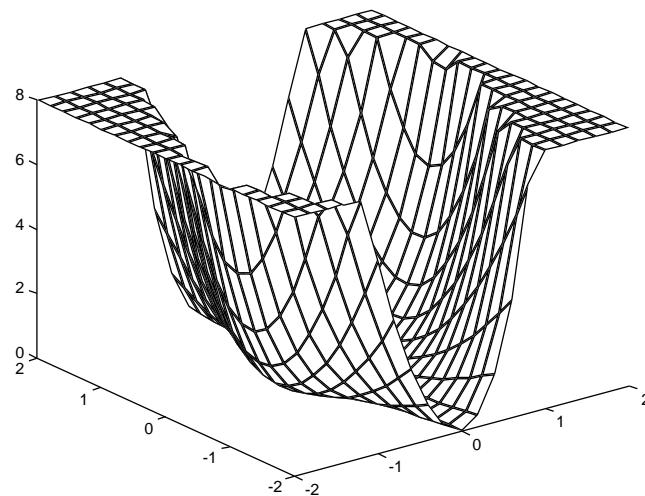
- Contour plot shows that the function has two strong local minimum points: (-0.42,0.42) and (0.55,-0.55) (the 2<sup>nd</sup> is global).
- Point (-0.13,0.13) is called saddle point. Along  $x_1=-x_2$  is a local max and along a line orthogonal to that line is a local min.
- [nnd8ts2](#)

# Vector Example (2)

$$F(\mathbf{x}) = (x_1^2 - 1.5x_1x_2 + 2x_2^2)x_1^2$$



- Any point along the line  $x_1=0$  is a weak minimum.



# First-Order Optimality Condition

$$F(\mathbf{x}) = F(\mathbf{x}^* + \Delta\mathbf{x}) = F(\mathbf{x}^*) + \nabla F(\mathbf{x})^T \Big|_{\mathbf{x} = \mathbf{x}^*} \Delta\mathbf{x} + \frac{1}{2} \Delta\mathbf{x}^T \nabla^2 F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}^*} \Delta\mathbf{x} + \dots$$

$$\Delta\mathbf{x} = \mathbf{x} - \mathbf{x}^*$$

For small  $\Delta\mathbf{x}$ :

If  $\mathbf{x}^*$  is a minimum, this implies:

$$F(\mathbf{x}^* + \Delta\mathbf{x}) \cong F(\mathbf{x}^*) + \nabla F(\mathbf{x})^T \Big|_{\mathbf{x} = \mathbf{x}^*} \Delta\mathbf{x}$$

$$\nabla F(\mathbf{x})^T \Big|_{\mathbf{x} = \mathbf{x}^*} \Delta\mathbf{x} \geq 0$$

If  $\nabla F(\mathbf{x})^T \Big|_{\mathbf{x} = \mathbf{x}^*} \Delta\mathbf{x} > 0$  then  $F(\mathbf{x}^* - \Delta\mathbf{x}) \cong F(\mathbf{x}^*) - \nabla F(\mathbf{x})^T \Big|_{\mathbf{x} = \mathbf{x}^*} \Delta\mathbf{x} < F(\mathbf{x}^*)$

But this would imply that  $\mathbf{x}^*$  is not a minimum. Therefore

$$\nabla F(\mathbf{x})^T \Big|_{\mathbf{x} = \mathbf{x}^*} \Delta\mathbf{x} = 0$$

Since this must be true for every  $\Delta\mathbf{x}$ ,

$$\boxed{\nabla F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}^*} = \mathbf{0}}$$

- Therefore the gradient must be zero at a minimum point.
- This is a first-order necessary (but not sufficient) condition for  $x^*$  to be a local minimum.
- Any points that satisfy this condition are called *stationary* points.

# Second-order Condition

If the first-order condition is satisfied (zero gradient), then

$$F(\mathbf{x}^* + \Delta\mathbf{x}) = F(\mathbf{x}^*) + \frac{1}{2}\Delta\mathbf{x}^T \nabla^2 F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}^*} \Delta\mathbf{x} + \dots$$

A strong minimum will exist at  $\mathbf{x}^*$  if  $\Delta\mathbf{x}^T \nabla^2 F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}^*} \Delta\mathbf{x} > 0$   
for any  $\Delta\mathbf{x} \neq 0$ .

Therefore the Hessian matrix must be **positive definite**.

A matrix  $\mathbf{A}$  is positive definite if:

$$\mathbf{z}^T \mathbf{A} \mathbf{z} > 0 \quad \text{for any } \mathbf{z} \neq 0.$$

This is a **sufficient** condition for optimality.

A **necessary** condition is that the Hessian matrix be positive semidefinite. A matrix  $\mathbf{A}$  is positive semidefinite if:

$$\mathbf{z}^T \mathbf{A} \mathbf{z} \geq 0 \quad \text{for any } \mathbf{z}.$$

By testing the eigenvalues of the matrix:

- If the eigenvalues are positive, then the matrix is positive definite.
- If all eigenvalues are nonnegative, then the matrix is positive semidefinite.
- A positive definite Hessian matrix is a 2<sup>nd</sup>-order, sufficient condition for a strong minimum to exist. It is not a necessary condition.
- A minimum can still be strong if the 2<sup>nd</sup>-order term of the Taylor series is zero, but the 3<sup>rd</sup>-order term is positive.
- Therefore the 2<sup>nd</sup>-order, necessary condition for a strong minimum is that the Hessian matrix be positive semidefinite.

# Summary

- The first-order and second-order necessary condition for  $\mathbf{x}^*$  to be a minimum, strong or weak, of  $F(\mathbf{x})$  are:

$\nabla F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^*} = \mathbf{0}$  and  $\nabla^2 F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^*}$  positive semidefinite.

- The 1<sup>st</sup> and 2<sup>nd</sup> order sufficient conditions for  $\mathbf{x}^*$  to be a strong minimum point of  $F(\mathbf{X})$  are

$\nabla F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^*} = \mathbf{0}$  and  $\nabla^2 F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^*}$  positive definite.

# Example

$$F(\mathbf{x}) = x_1^2 + 2x_1x_2 + 2x_2^2 + x_1$$

$$\nabla F(\mathbf{x}) = \begin{bmatrix} 2x_1 + 2x_2 + 1 \\ 2x_1 + 4x_2 \end{bmatrix} = \mathbf{0} \quad \longrightarrow \quad \mathbf{x}^* = \begin{bmatrix} -1 \\ 0.5 \end{bmatrix}$$

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix} \quad (\text{Not a function of } \mathbf{x} \text{ in this case.})$$

To test the definiteness, check the eigenvalues of the Hessian. If the eigenvalues are all greater than zero, the Hessian is positive definite.

$$|\nabla^2 F(\mathbf{x}) - \lambda \mathbf{I}| = \left| \begin{bmatrix} 2-\lambda & 2 \\ 2 & 4-\lambda \end{bmatrix} \right| = \lambda^2 - 6\lambda + 4 = (\lambda - 0.76)(\lambda - 5.24)$$

$$\lambda = 0.76, 5.24$$

**Both eigenvalues are positive, therefore strong minimum.** 21

# Quadratic Functions

$$F(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{d}^T \mathbf{x} + c \quad (\text{Symmetric } \mathbf{A})$$

Note: If  $\mathbf{A}$  is not symmetric we always can find another symmetric matrix to replace and get the same  $F(\mathbf{x})$ .

Gradient and Hessian:

Useful properties of gradients :

$$\nabla(\mathbf{h}^T \mathbf{x}) = \nabla(\mathbf{x}^T \mathbf{h}) = \mathbf{h} \quad (\text{for constant vector } \mathbf{h})$$

$$\nabla \mathbf{x}^T \mathbf{Q} \mathbf{x} = \mathbf{Q} \mathbf{x} + \mathbf{Q}^T \mathbf{x} = 2\mathbf{Q} \mathbf{x} \quad (\text{for symmetric } \mathbf{Q})$$

Gradient of Quadratic Function:

$$\nabla F(\mathbf{x}) = \mathbf{A} \mathbf{x} + \mathbf{d}$$

Hessian of Quadratic Function:

$$\nabla^2 F(\mathbf{x}) = \mathbf{A}$$

Higher derivatives are zero, then 3 terms of Taylor series give an exact representation of the function. We say all analytic functions behave like quadratics over a small neighborhood.

# General shape of the quadratic function

Consider a quadratic function which has a stationary point at the origin, and whose value there is zero.

$$F(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x}$$

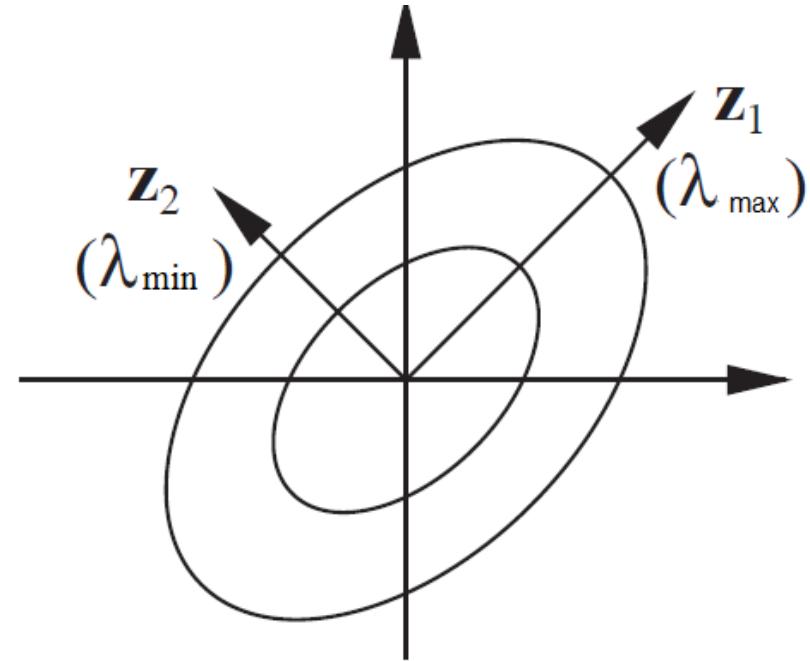
the second derivative of a function in the direction of an arbitrary vector is given by

$$\frac{\mathbf{p}^T \nabla^2 F(\mathbf{x}) \mathbf{p}}{\|\mathbf{p}\|^2} = \frac{\mathbf{p}^T \mathbf{A} \mathbf{p}}{\|\mathbf{p}\|^2}$$

It can be proved that

$$\lambda_{min} \leq \frac{\mathbf{p}^T \mathbf{A} \mathbf{p}}{\|\mathbf{p}\|^2} \leq \lambda_{max}$$

$$\frac{\mathbf{z}_{max}^T \mathbf{A} \mathbf{z}_{max}}{\|\mathbf{z}_{max}\|^2} = \lambda_{max}$$



The eigenvalues represent curvature (2<sup>nd</sup> derivatives) along the eigenvectors (the principal axes).

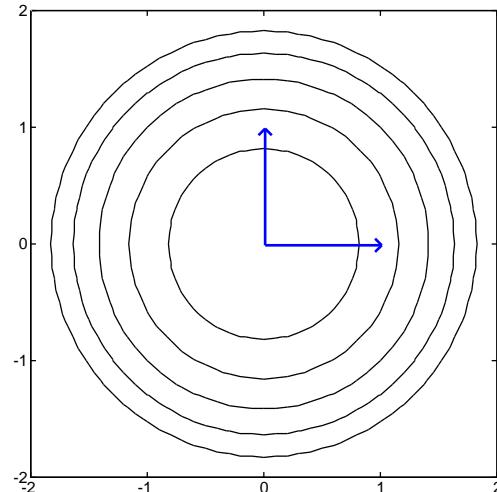
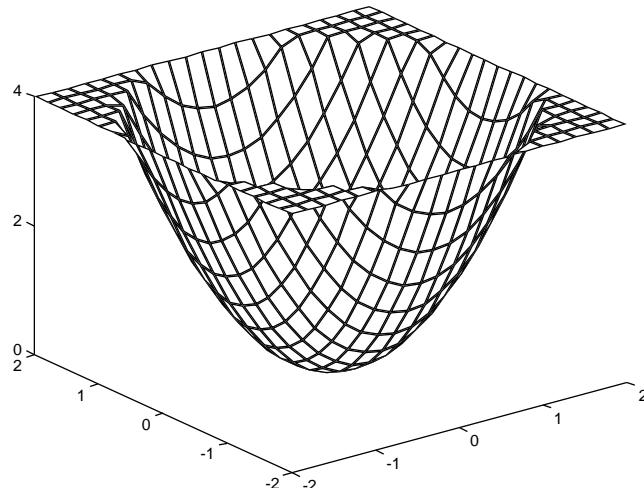
Exp: When both eigenvalues are positive.

# Circular Hollow

$$F(\mathbf{x}) = x_1^2 + x_2^2 = \frac{1}{2} \mathbf{x}^T \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \mathbf{x}$$

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \quad \lambda_1 = 2 \quad \mathbf{z}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \lambda_2 = 2 \quad \mathbf{z}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

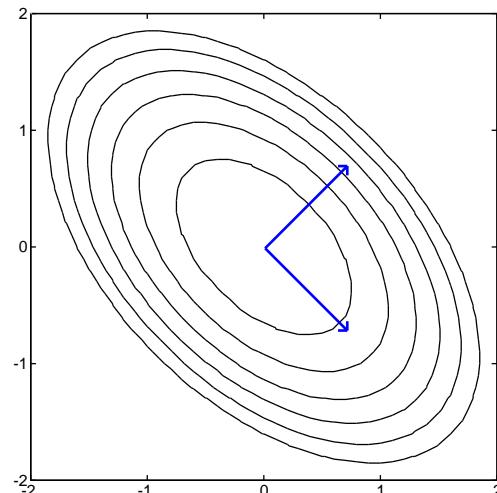
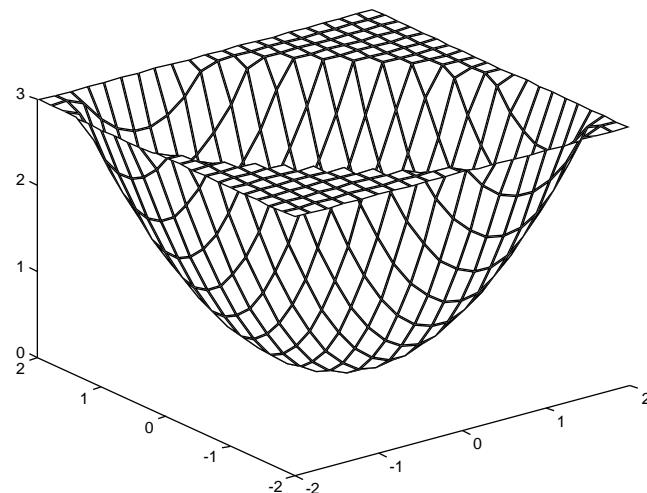
(Any two independent vectors in the plane would work.)



# Elliptical Hollow

$$F(\mathbf{x}) = x_1^2 + x_1 x_2 + x_2^2 = \frac{1}{2} \mathbf{x}^T \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \mathbf{x}$$

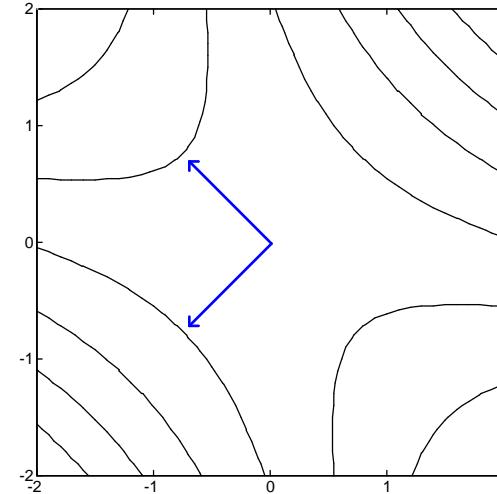
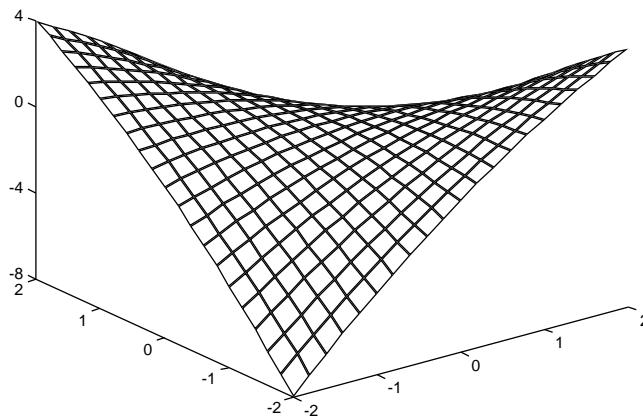
$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \quad \lambda_1 = 1 \quad \mathbf{z}_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \quad \lambda_2 = 3 \quad \mathbf{z}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$



# Elongated Saddle

$$F(\mathbf{x}) = -\frac{1}{4}x_1^2 - \frac{3}{2}x_1x_2 - \frac{1}{4}x_2^2 = \frac{1}{2}\mathbf{x}^T \begin{bmatrix} -0.5 & -1.5 \\ -1.5 & -0.5 \end{bmatrix} \mathbf{x}$$

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} -0.5 & -1.5 \\ -1.5 & -0.5 \end{bmatrix} \quad \lambda_1 = 1 \quad \mathbf{z}_1 = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \quad \lambda_2 = -2 \quad \mathbf{z}_2 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$$

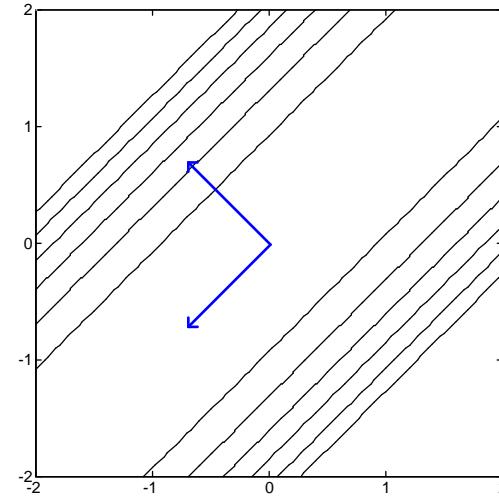
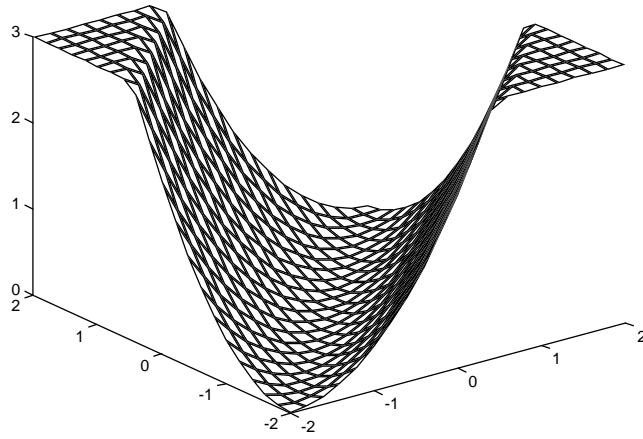


The stationary point  $\mathbf{X}^*=[0 \ 0]^T$  is a saddle point. The Hessian matrix is indefinite.

# Stationary Valley

$$F(\mathbf{x}) = \frac{1}{2}x_1^2 - x_1 x_2 + \frac{1}{2}x_2^2 = \frac{1}{2}\mathbf{x}^T \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \mathbf{x}$$

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \quad \lambda_1 = 1 \quad \mathbf{z}_1 = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \quad \lambda_2 = 0 \quad \mathbf{z}_2 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$$



The Hessian matrix is positive semidefinite. We have a weak minimum along the line  $x_1=x_2$  (corresponding to the 2<sup>nd</sup> eigenvector.) [nnd8qf](#)

# Quadratic Function Summary

- If the eigenvalues of the Hessian matrix are all positive, the function will have a single strong minimum.
- If the eigenvalues are all negative, the function will have a single strong maximum.
- If some eigenvalues are positive and other eigenvalues are negative, the function will have a single saddle point.
- If the eigenvalues are all nonnegative, but some eigenvalues are zero, then the function will either have a weak minimum or will have no stationary point.
- If the eigenvalues are all nonpositive, but some eigenvalues are zero, then the function will either have a weak maximum or will have no stationary point.

- Note: For simplicity we assumed that stationary point was at the origin and had a value zero.
- If  $c \neq 0 \rightarrow$  function is increased by  $c$  at every point.
- If  $\mathbf{d} \neq \mathbf{0}$  and  $\mathbf{A}$  is invertible  $\rightarrow$  Shape of contours do not change but the stationary point moves to:

$$\mathbf{X}^* = -\mathbf{A}^{-1}\mathbf{d}$$

- If  $\mathbf{d} \neq \mathbf{0}$  and  $\mathbf{A}$  is not invertible  $\rightarrow$  there could be no stationary points.

# Performance Optimization

- We use the Taylor series expansion to develop algorithms to locate the optimum points.
- We discuss three categories of optimization algorithm.
  - *Steepest Descent*
  - *Newton's Method*
  - *Conjugate Gradient*
- In Ch 10 we apply these algorithms to the training of neural networks.

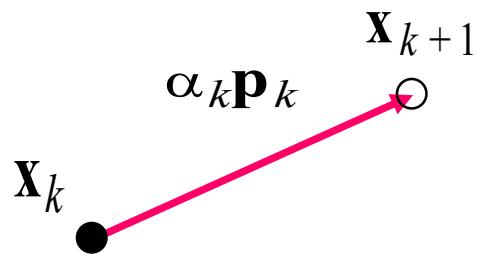
# Basic Optimization Algorithm

- We develop algorithm to search the parameter space and locate minimum points of the surface i.e. find the optimum weights and biases for a given network.
- We are going to develop algorithms to optimize a performance index  $F(\mathbf{x})$  (precisely to find  $\mathbf{x}$  that minimizes  $F(\mathbf{x})$ ).

We begin from some initial guess  $\mathbf{x}_0$  and then update our guess in stages as:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$

or  $\Delta \mathbf{x}_k = (\mathbf{x}_{k+1} - \mathbf{x}_k) = \alpha_k \mathbf{p}_k$



$\mathbf{p}_k$  - Search Direction

$\alpha_k$  - Learning Rate

We discuss 3 possibilities for search direction  $\mathbf{p}_k$  and variety of ways to select  $\alpha_k$ .

# 1- Steepest Descent

Choose the next step so that the function decreases:

$$F(\mathbf{x}_{k+1}) < F(\mathbf{x}_k)$$

How can we choose a direction  $\mathbf{p}_k$ , so that for small learning rate  $\alpha_k$  we will move downhill in this way?

For small changes in  $\mathbf{x}$  we can approximate  $F(\mathbf{x})$ :

$$F(\mathbf{x}_{k+1}) = F(\mathbf{x}_k + \Delta\mathbf{x}_k) \approx F(\mathbf{x}_k) + \mathbf{g}_k^T \Delta\mathbf{x}_k \quad \text{1st order}$$

Where  $\mathbf{g}_k$  is the gradient evaluated at the old guess  $\mathbf{x}_k$ :

$$\mathbf{g}_k \equiv \nabla F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}_k}$$

If we want the function to decrease i. e.  $F(\mathbf{x}_{k+1}) < F(\mathbf{x}_k)$

$$\mathbf{g}_k^T \Delta \mathbf{x}_k = \alpha_k \mathbf{g}_k^T \mathbf{p}_k < 0$$

We select an  $\alpha_k$  small positive number so  $\mathbf{g}_k^T \mathbf{p}_k < 0$

Any  $\mathbf{p}_k$  that satisfies this eq. is called a descent direction.

We can maximize the decrease (*steepest descent*) by choosing:

$$\mathbf{p}_k = -\mathbf{g}_k$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k$$

$\alpha_k$ :

1. We can choose a fixed or variable (like  $1/k$ ) value.
2. By minimizing  $F(\mathbf{x})$  with respect to  $\alpha_k$  along the line  $\mathbf{x}_k - \alpha_k \mathbf{g}_k$

# Example

$$F(\mathbf{x}) = x_1^2 + 2x_1x_2 + 2x_2^2 + x_1$$

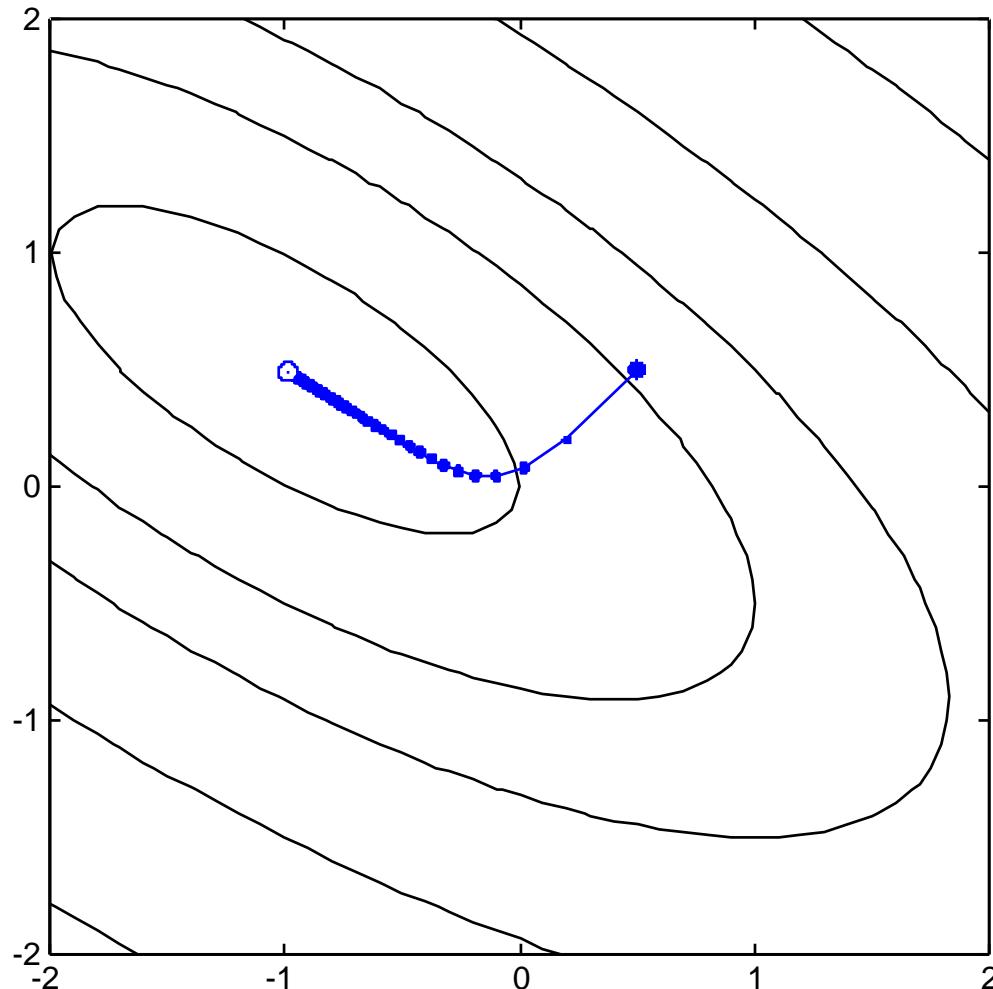
$$\mathbf{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \quad \alpha = 0.1$$

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 2x_1 + 2x_2 + 1 \\ 2x_1 + 4x_2 \end{bmatrix} \quad \mathbf{g}_0 = \nabla F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}_0} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

$$\mathbf{x}_1 = \mathbf{x}_0 - \alpha \mathbf{g}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - 0.1 \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 0.2 \\ 0.2 \end{bmatrix}$$

$$\mathbf{x}_2 = \mathbf{x}_1 - \alpha \mathbf{g}_1 = \begin{bmatrix} 0.2 \\ 0.2 \end{bmatrix} - 0.1 \begin{bmatrix} 1.8 \\ 1.2 \end{bmatrix} = \begin{bmatrix} 0.02 \\ 0.08 \end{bmatrix}$$

# Plot



What is the effect of increasing learning rate?

# Stable Learning Rates (Quadratic)

$$F(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{d}^T \mathbf{x} + c$$

$$\nabla F(\mathbf{x}) = \mathbf{A} \mathbf{x} + \mathbf{d}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \mathbf{g}_k = \mathbf{x}_k - \alpha(\mathbf{A}\mathbf{x}_k + \mathbf{d}) \longrightarrow \mathbf{x}_{k+1} = \underbrace{[\mathbf{I} - \alpha\mathbf{A}]\mathbf{x}_k}_{\text{Stability is determined}} - \alpha\mathbf{d}$$

This is a linear dynamic system. →

Stability is determined  
by the eigenvalues of  
this matrix.

$$[\mathbf{I} - \alpha\mathbf{A}]\mathbf{z}_i = \mathbf{z}_i - \alpha\mathbf{A}\mathbf{z}_i = \mathbf{z}_i - \underbrace{\alpha\lambda_i\mathbf{z}_i}_{\text{(}\lambda_i\text{- eigenvalue of A)}} = (1 - \underbrace{\alpha\lambda_i}_{\text{Eigenvalues}})\mathbf{z}_i$$

( $\lambda_i$ - eigenvalue of  $\mathbf{A}$ )

Eigenvalues  
of  $[\mathbf{I} - \alpha\mathbf{A}]$ .

Stability Requirement:

$$|(1 - \alpha\lambda_i)| < 1 \quad \alpha < \frac{2}{\lambda_i}$$

$$\alpha < \frac{2}{\lambda_{max}}$$

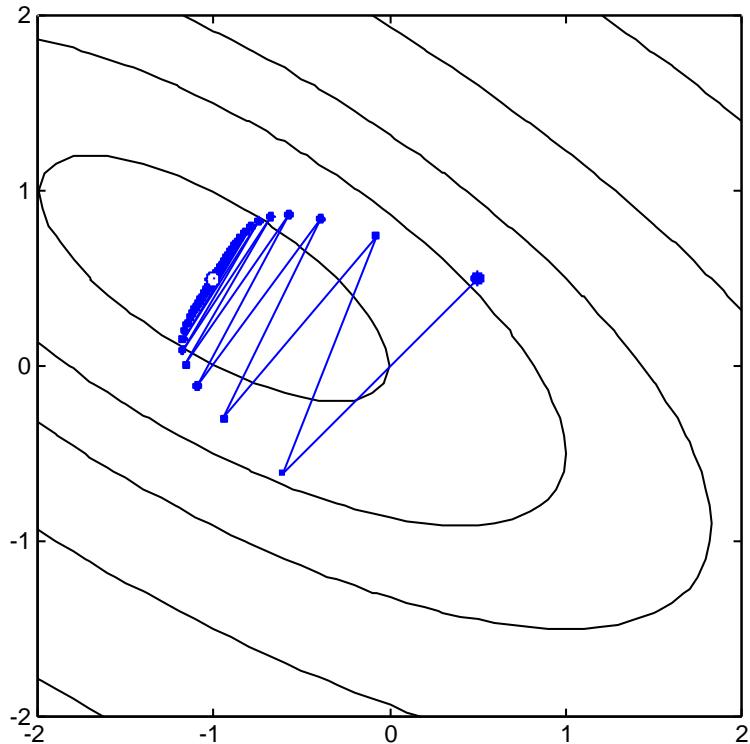
If the gradient is changing too fast we may jump past the minimum point so far that the gradient at the new location will be larger in magnitude (but opposite direction than the gradient at the old location).

**Exp:**  $A = \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix}$   $\left\{ (\lambda_1 = 0.764, \mathbf{z}_1 = \begin{bmatrix} 0.851 \\ -0.526 \end{bmatrix}), (\lambda_2 = 5.24, \mathbf{z}_2 = \begin{bmatrix} 0.526 \\ 0.851 \end{bmatrix}) \right\}$

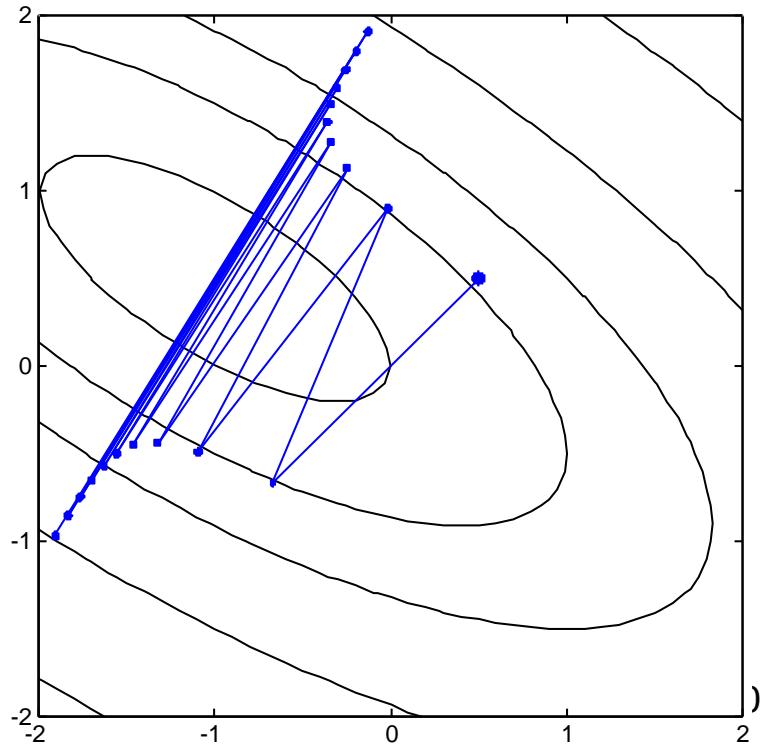
$$\alpha < \frac{2}{\lambda_{max}} = \frac{2}{5.24} = 0.38$$

[nnd9sda](#)

$$\alpha = 0.37$$



$$\alpha = 0.39$$



# Minimizing Along a Line

Choose  $\alpha_k$  to minimize  $F(\mathbf{x}_k + \alpha_k \mathbf{p}_k)$

$$\frac{d}{d\alpha_k} (F(\mathbf{x}_k + \alpha_k \mathbf{p}_k)) = \nabla F(\mathbf{x})^T \Big|_{\mathbf{x} = \mathbf{x}_k} \mathbf{p}_k + \alpha_k \mathbf{p}_k^T \nabla^2 F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}_k} \mathbf{p}_k$$

Set equal zero  $\rightarrow$

$$\alpha_k = - \frac{\nabla F(\mathbf{x})^T \Big|_{\mathbf{x} = \mathbf{x}_k} \mathbf{p}_k}{\mathbf{p}_k^T \nabla^2 F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}_k} \mathbf{p}_k} = - \frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{p}_k^T \mathbf{A}_k \mathbf{p}_k}$$

where

$$\mathbf{A}_k \equiv \nabla^2 F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}_k}$$

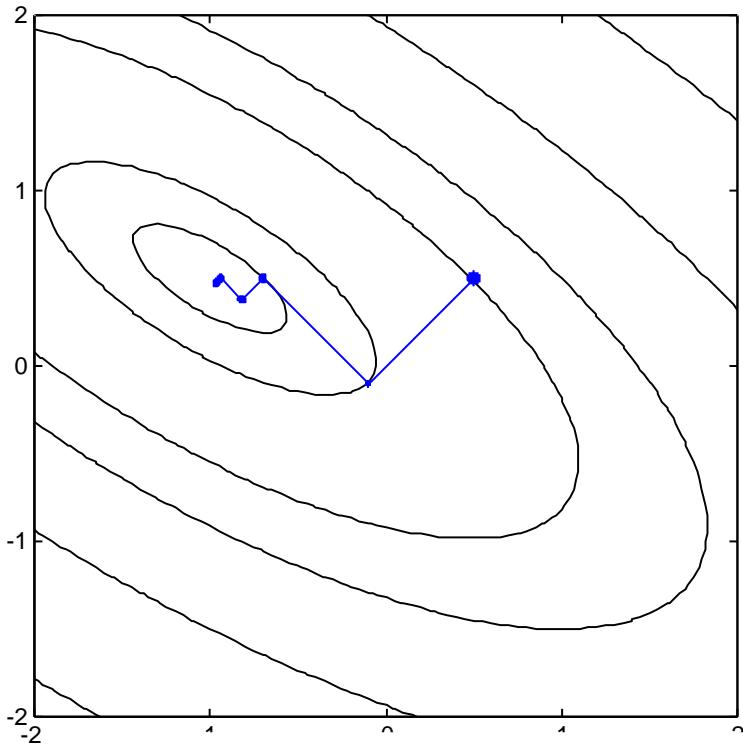
# Example

$$F(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 1 & 0 \end{bmatrix} \mathbf{x} \quad \mathbf{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 2x_1 + 2x_2 + 1 \\ 2x_1 + 4x_2 \end{bmatrix} \quad \mathbf{p}_0 = -\mathbf{g}_0 = -\nabla F(\mathbf{x}) \Big|_{\mathbf{X} = \mathbf{x}_0} = \begin{bmatrix} -3 \\ -3 \end{bmatrix}$$

$$\alpha_0 = -\frac{\begin{bmatrix} 3 & 3 \end{bmatrix} \begin{bmatrix} -3 \\ -3 \end{bmatrix}}{\begin{bmatrix} -3 & -3 \end{bmatrix} \begin{bmatrix} 2 & 2 \end{bmatrix} \begin{bmatrix} -3 \\ -3 \end{bmatrix}} = 0.2 \quad \mathbf{x}_1 = \mathbf{x}_0 - \alpha_0 \mathbf{g}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - 0.2 \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} -0.1 \\ -0.1 \end{bmatrix}$$

# Plot



Note that the successive steps of the algorithm are orthogonal. Why does this happen?

$$\frac{d}{d\alpha_k} F(\mathbf{x}_k + \alpha_k \mathbf{p}_k) = \frac{d}{d\alpha_k} F(\mathbf{x}_{k+1}) = \nabla F(\mathbf{x})^T \Big|_{\mathbf{x} = \mathbf{x}_{k+1}} \frac{d}{d\alpha_k} [\mathbf{x}_k + \alpha_k \mathbf{p}_k]$$

$$= \nabla F(\mathbf{x})^T \Big|_{\mathbf{x} = \mathbf{x}_{k+1}} \mathbf{p}_k = \mathbf{g}_{k+1}^T \mathbf{p}_k$$

## 2- Newton's Method

The steepest descent algorithm was based on the 1<sup>st</sup>-order Taylor series expansion. Newton's method is based on the 2<sup>nd</sup>-order Taylor series.

$$F(\mathbf{x}_{k+1}) = F(\mathbf{x}_k + \Delta\mathbf{x}_k) \approx F(\mathbf{x}_k) + \mathbf{g}_k^T \Delta\mathbf{x}_k + \frac{1}{2} \Delta\mathbf{x}_k^T \mathbf{A}_k \Delta\mathbf{x}_k$$

Take the gradient of this second-order approximation and set it equal to zero to find the stationary point.

$$\mathbf{g}_k + \mathbf{A}_k \Delta\mathbf{x}_k = \mathbf{0} \implies \Delta\mathbf{x}_k = -\mathbf{A}_k^{-1} \mathbf{g}_k \implies \boxed{\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1} \mathbf{g}_k}$$

So actually we locate the stationary point of this quadratic approximation function and set to  $F(\mathbf{x})$ .

# Example

$$F(\mathbf{x}) = x_1^2 + 2x_1x_2 + 2x_2^2 + x_1$$

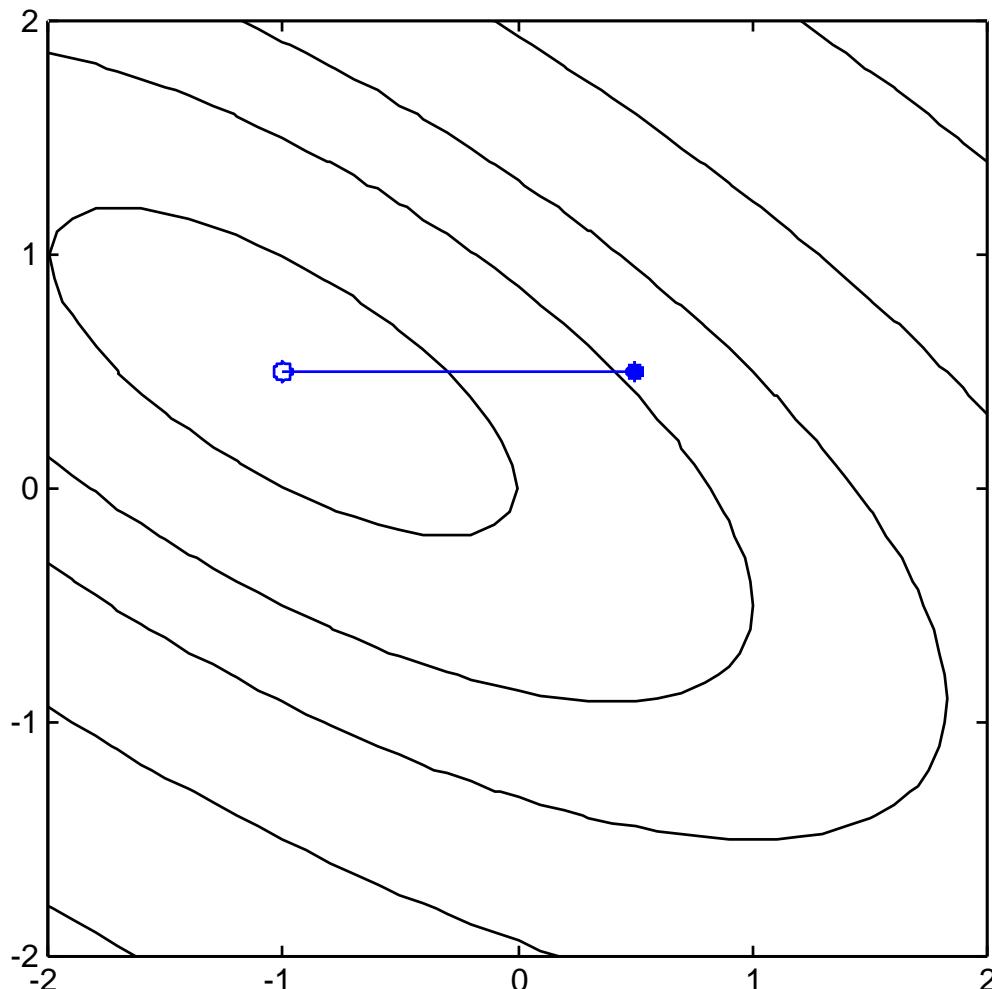
$$\mathbf{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 2x_1 + 2x_2 + 1 \\ 2x_1 + 4x_2 \end{bmatrix} \quad \mathbf{g}_0 = \nabla F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}_0} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

$$\mathbf{A} = \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix}$$

$$\mathbf{x}_1 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix}^{-1} \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - \begin{bmatrix} 1 & -0.5 \\ -0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - \begin{bmatrix} 1.5 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 0.5 \end{bmatrix}$$

# Plot



This method will always find the minimum of a quadratic function in *one* step.

- This is because Newton's method is designed to approximate a function as quadratic and then locate the stationary point of the quadratic approximation.
- If the original function is quadratic (with a strong minimum) it will be minimized in one step.
- If the function  $F(x)$  is not quadratic, then Newton's method will not generally converge in one step. In fact, we cannot be sure that it will converge at all, since this will depend on the function and the initial guess.

# Non-Quadratic Example

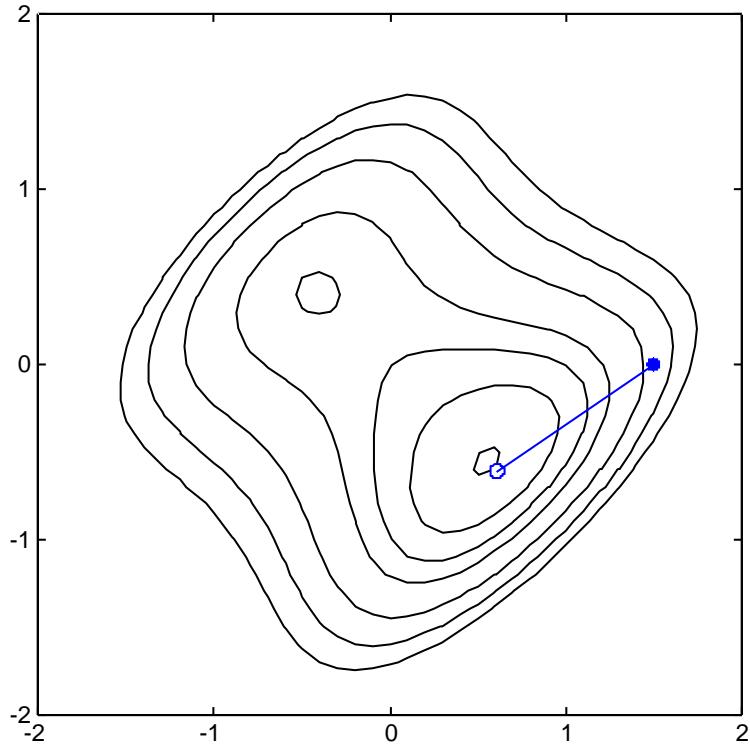
$$F(\mathbf{x}) = (x_2 - x_1)^4 + 8x_1x_2 - x_1 + x_2 + 3$$

Stationary Points:  $\mathbf{x}^1 = \begin{bmatrix} -0.42 \\ 0.42 \end{bmatrix}$  Local minimum

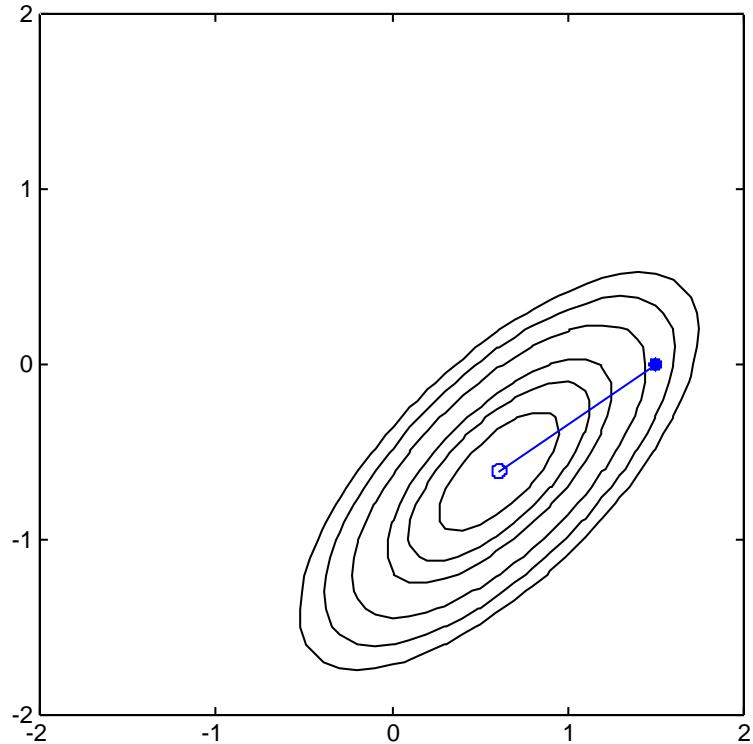
$\mathbf{x}^2 = \begin{bmatrix} -0.13 \\ 0.13 \end{bmatrix}$  Saddle point

$\mathbf{x}^3 = \begin{bmatrix} 0.55 \\ -0.55 \end{bmatrix}$  Strong global minimum

We apply Newton's method to this problem starting from the initial point  $\mathbf{x}_0 = [1.5 \quad 0]^T$

$F(\mathbf{x})$ 

First iteration –original function

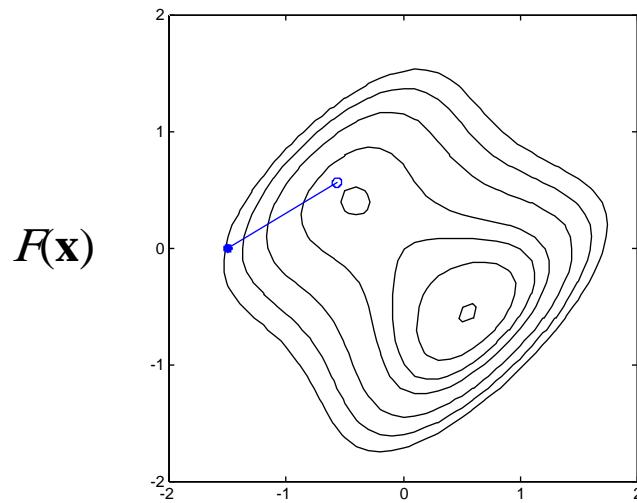
 $F_2(\mathbf{x})$ 

Quadratic approx. at the initial guess.

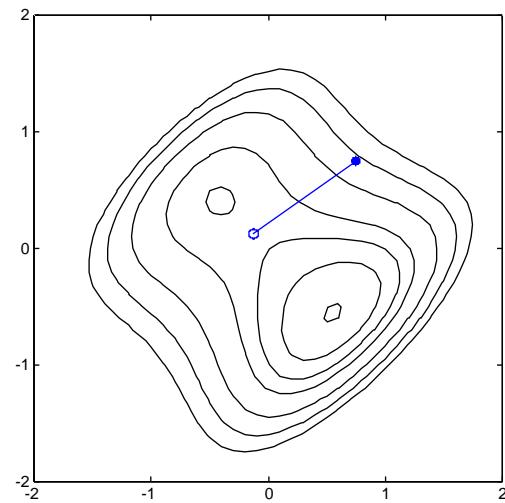
The function is not minimized in one step, since it is not quadratic. If we take two more steps toward the global point we will converge to within 0.01 of the global minimum.

# Different Initial Conditions

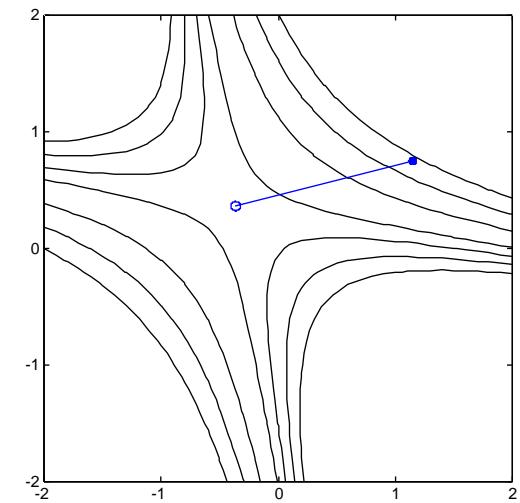
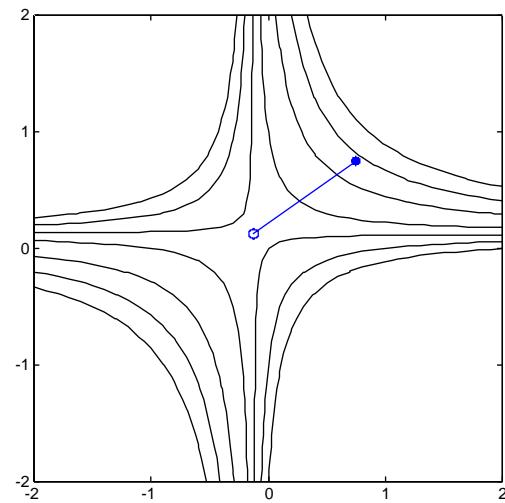
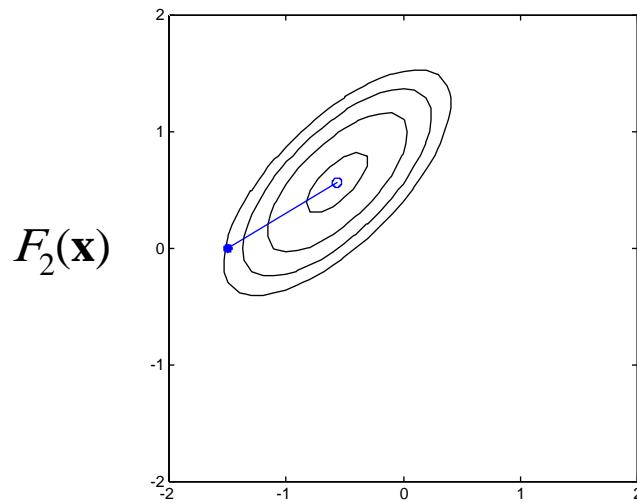
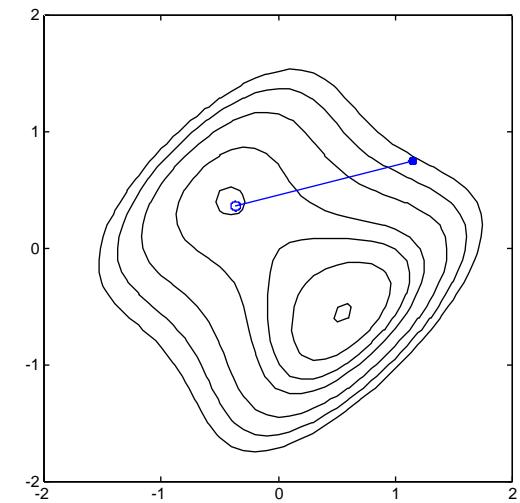
$$\mathbf{x}_0 = [-1.5 \ 0]^T$$



$$\mathbf{x}_0 = [0.75 \ 0.75]^T$$



$$\mathbf{x}_0 = [1.15 \ 0.75]^T$$



nnd9nm, nnd9sd

- Newton's method usually produces faster convergence than steepest descent.
- Newton's method can be quite complex.
- Convergence to saddle point is very unlikely with SD.
- In Newton's it is possible for the algorithm to oscillate or diverge, but SD is guaranteed to converge if  $\alpha$  is not too large or if we perform a linear minimization at each stage.
- In Ch 12 a variation of Newton's method that is well suited to *NN* training is discussed. It eliminates the divergence problem by using SD steps whenever divergence begins to occur.
- Newton's method requires computation and storage of the Hessian Matrix as well as its inverse.

# 3- Conjugate Vectors

- Newton's method has quadratic termination. It minimizes quadratic function exactly in a finite number of iterations.
- When the number of parameters,  $n$ , is large it is difficult to compute and store the Hessian matrix ( $2^{\text{nd}}$  derivatives). We would like to have methods that require only  $1^{\text{st}}$  derivatives but still have quadratic termination.
- The search direction of SD algorithm with linear searches at each iteration, are orthogonal.

# Conjugate Vectors (Cont.)

Is there a set of search directions that will guarantee quadratic termination?

$$F(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{d}^T \mathbf{x} + c$$

A set of vectors is mutually conjugate with respect to a positive definite Hessian matrix  $\mathbf{A}$  if

$$\mathbf{p}_k^T \mathbf{A} \mathbf{p}_j = 0 \quad k \neq j$$

There are an infinite number of mutually conjugate vectors that span a given  $n$ -dimensional space. If  $\mathbf{A}$  is equal to the identity matrix, conjugacy is equivalent to the usual notion of orthogonality.

One set of conjugate vectors consists of the eigenvectors of  $\mathbf{A}$ .

$$\mathbf{z}_k^T \mathbf{A} \mathbf{z}_j = \lambda_j \mathbf{z}_k^T \mathbf{z}_j = 0 \quad k \neq j$$

Note: The eigenvectors of symmetric matrices are orthogonal.

- Therefore the eigenvectors are both conjugate and orthogonal.
- We can minimize a quadratic function exactly by searching along the eigenvectors of the Hessian matrix, since they form the principal axes of the function contours. So we must find the Hessian matrix.
- But we look for an algorithm that doesn't need 2<sup>nd</sup> derivative.

# For Quadratic Functions

- It can be shown that if we search along any set of conjugate directions  $\{\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_n\}$  then the exact minimum of any quadratic function with  $n$  parameters will be reached at most  $n$  searches.
- How can we construct these conjugate search directions?

$$\nabla F(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{d} \quad \nabla^2 F(\mathbf{x}) = \mathbf{A}$$

The change in the gradient at iteration  $k$  is

$$\Delta \mathbf{g}_k = \mathbf{g}_{k+1} - \mathbf{g}_k = (\mathbf{A}\mathbf{x}_{k+1} + \mathbf{d}) - (\mathbf{A}\mathbf{x}_k + \mathbf{d}) = \mathbf{A}\Delta\mathbf{x}_k$$

where

$$\Delta \mathbf{x}_k = (\mathbf{x}_{k+1} - \mathbf{x}_k) = \alpha_k \mathbf{p}_k$$

$\alpha_k$  is chosen to minimizes  $F(\mathbf{x})$  in the direction  $\mathbf{p}_k$ .

The conjugacy conditions can be rewritten

$$\alpha_k \mathbf{p}_k^T \mathbf{A} \mathbf{p}_j = \Delta \mathbf{x}_k^T \mathbf{A} \mathbf{p}_j = \Delta \mathbf{g}_k^T \mathbf{p}_j = 0 \quad k \neq j$$

This does not require knowledge of the Hessian matrix.

We have restated the conjugacy conditions in terms of the **changes in the gradient** at successive iterations.

# Forming Conjugate Directions

The search directions will be conjugate if they are orthogonal to the changes in the gradients.

Choose the initial search direction (arbitrary) as the negative of the gradient.

$$\mathbf{p}_0 = -\mathbf{g}_0$$

At each iteration we should construct a vector  $\mathbf{p}_k$  that is orthogonal to  $\{\Delta\mathbf{g}_0, \Delta\mathbf{g}_1, \dots, \Delta\mathbf{g}_{k-1}\}$ .

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

$$\beta_k = \frac{\Delta \mathbf{g}_{k-1}^T \mathbf{g}_k}{\Delta \mathbf{g}_{k-1}^T \mathbf{p}_{k-1}} \quad (1)$$

or

$$\beta_k = \frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}} \quad (2)$$

or

$$\beta_k = \frac{\Delta \mathbf{g}_{k-1}^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}} \quad (3)$$

# Conjugate Gradient Algorithm

1. The first search direction is the negative of the gradient.

$$\mathbf{p}_0 = -\mathbf{g}_0$$

2. Take a step according to Eq:

$$\Delta \mathbf{x}_k = (\mathbf{x}_{k+1} - \mathbf{x}_k) = \alpha_k \mathbf{p}_k$$

Select the learning rate to minimize along the line. We will discuss general linear minimization techniques in Ch12. For quadratic functions we can use Eq:

$$\alpha_k = - \frac{\nabla F(\mathbf{x})^T \Big|_{\mathbf{x} = \mathbf{x}_k} \mathbf{p}_k}{\mathbf{p}_k^T \nabla^2 F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}_k} \mathbf{p}_k} = - \frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{p}_k^T \mathbf{A}_k \mathbf{p}_k}$$

# CG Algorithm (Cont.)

3. Select the next search direction using

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

Calculate  $\beta_k$  from one of three equations.

4. If the algorithm has not converged, return to second step.

➤ A quadratic function will be minimized in  $n$  steps.

# Example

$$F(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 1 & 0 \end{bmatrix} \mathbf{x}$$

$$\mathbf{x}^* = \begin{bmatrix} -1 \\ 0.5 \end{bmatrix} \quad \mathbf{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 2x_1 + 2x_2 + 1 \\ 2x_1 + 4x_2 \end{bmatrix} \quad \mathbf{p}_0 = -\mathbf{g}_0 = -\nabla F(\mathbf{x}) \Big|_{\mathbf{X} = \mathbf{x}_0} = \begin{bmatrix} -3 \\ -3 \end{bmatrix}$$

$$\alpha_0 = -\frac{\begin{bmatrix} 3 & 3 \end{bmatrix} \begin{bmatrix} -3 \\ -3 \end{bmatrix}}{\begin{bmatrix} -3 & -3 \end{bmatrix} \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} -3 \\ -3 \end{bmatrix}} = 0.2 \quad \mathbf{x}_1 = \mathbf{x}_0 - \alpha_0 \mathbf{g}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - 0.2 \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} -0.1 \\ -0.1 \end{bmatrix}$$

Which is equivalent to the first step of SD with minimization along a line.

## Example (Cont.)

$$\mathbf{g}_1 = \nabla F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}_1} = \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} -0.1 \\ -0.1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.6 \\ -0.6 \end{bmatrix}$$

$$\beta_1 = \frac{\mathbf{g}_1^T \mathbf{g}_1}{\mathbf{g}_0^T \mathbf{g}_0} = \frac{\begin{bmatrix} 0.6 & -0.6 \end{bmatrix} \begin{bmatrix} 0.6 \\ -0.6 \end{bmatrix}}{\begin{bmatrix} 3 & 3 \end{bmatrix} \begin{bmatrix} 3 \\ 3 \end{bmatrix}} = \frac{0.72}{18} = 0.04 \quad \text{Fletcher-Reeves}$$

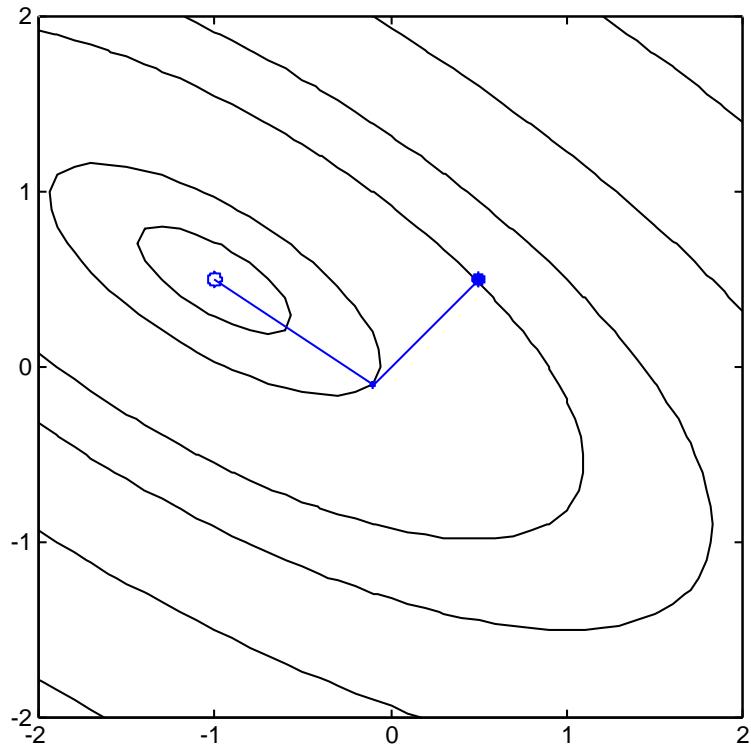
$$\mathbf{p}_1 = -\mathbf{g}_1 + \beta_1 \mathbf{p}_0 = \begin{bmatrix} -0.6 \\ 0.6 \end{bmatrix} + 0.04 \begin{bmatrix} -3 \\ -3 \end{bmatrix} = \begin{bmatrix} -0.72 \\ 0.48 \end{bmatrix}$$

$$\alpha_1 = -\frac{\begin{bmatrix} 0.6 & -0.6 \end{bmatrix} \begin{bmatrix} -0.72 \\ 0.48 \end{bmatrix}}{\begin{bmatrix} -0.72 & 0.48 \end{bmatrix} \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} -0.72 \\ 0.48 \end{bmatrix}} = -\frac{-0.72}{0.576} = 1.25$$

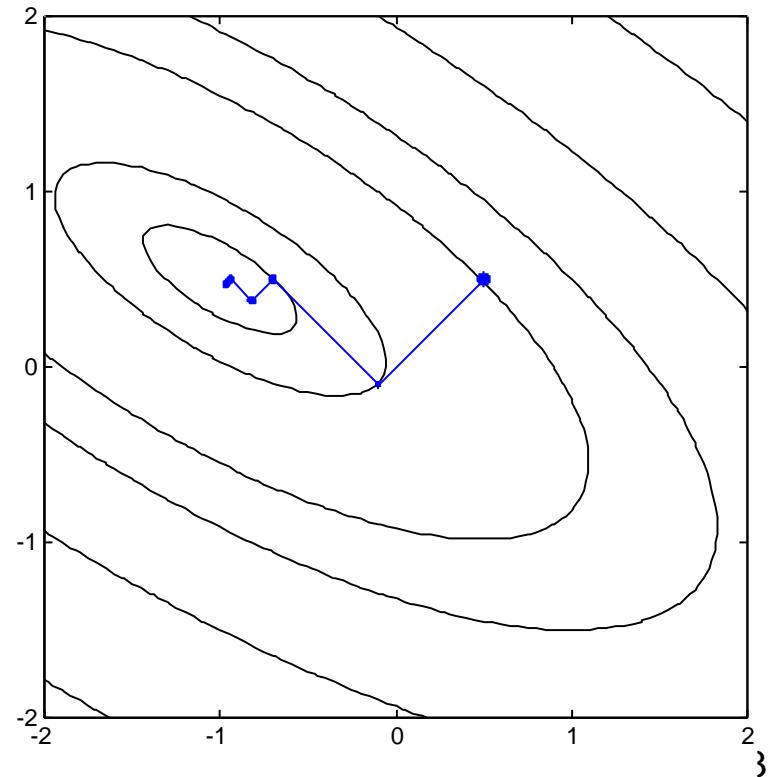
$$\mathbf{x}_2 = \mathbf{x}_1 + \alpha_1 \mathbf{p}_1 = \begin{bmatrix} -0.1 \\ -0.1 \end{bmatrix} + 1.25 \begin{bmatrix} -0.72 \\ 0.48 \end{bmatrix} = \begin{bmatrix} -1 \\ 0.5 \end{bmatrix} \quad \mathbf{x}^* = \begin{bmatrix} -1 \\ 0.5 \end{bmatrix}$$

# Plots

Conjugate Gradient



Steepest Descent



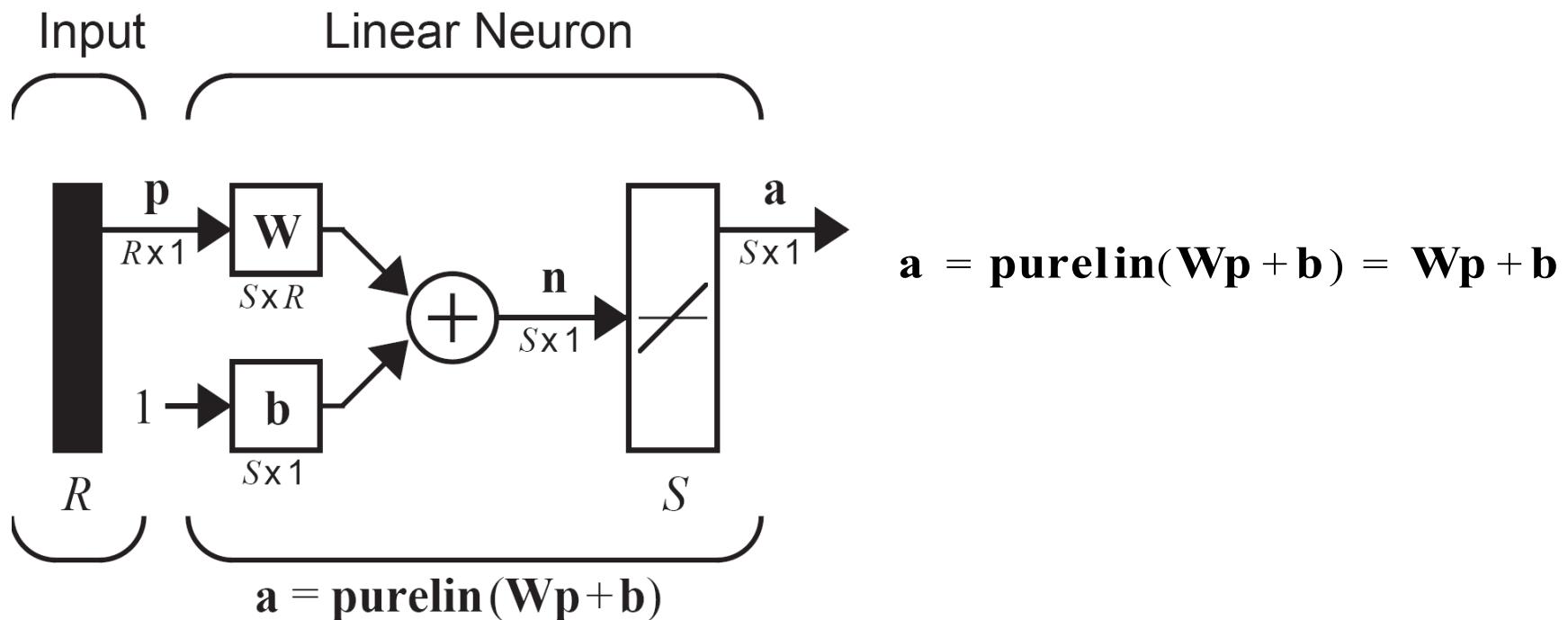
nnd9mc

# Widrow-Hoff Learning (LMS Algorithm)

- In 1960 Widrow and Hoff introduced ADALINE (ADaptive LInear NEuron) network.
- Its learning rule is called LMS (Least Mean Square) algorithm.
- ADALINE is similar to the perceptron, except that its transfer function is linear, instead of hard limiting.

- Both have the same limitations; They can only solve linearly separable problems.
- The LMS algorithm found many more practical uses than the perceptron (like most long distance phone lines use ADALINE network for echo cancellation).

# ADALINE Network



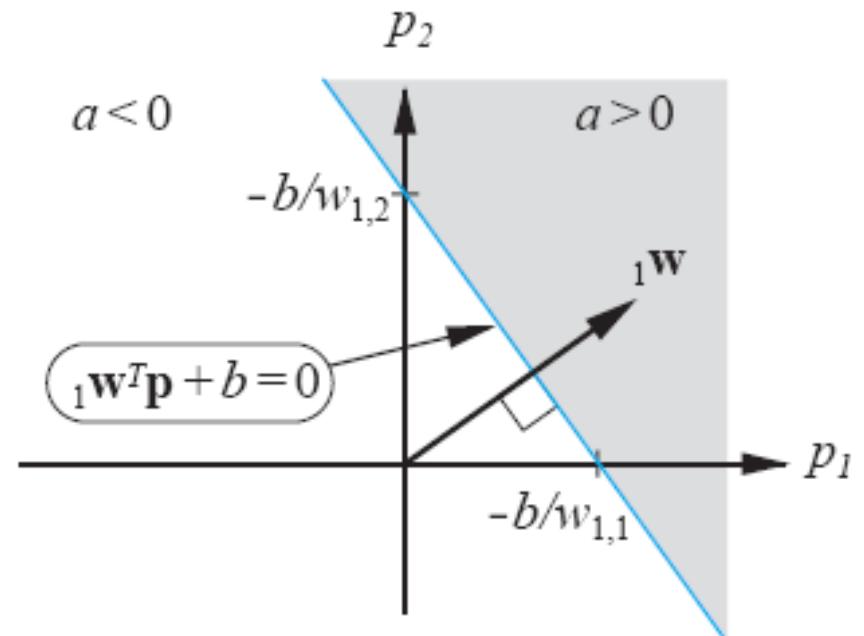
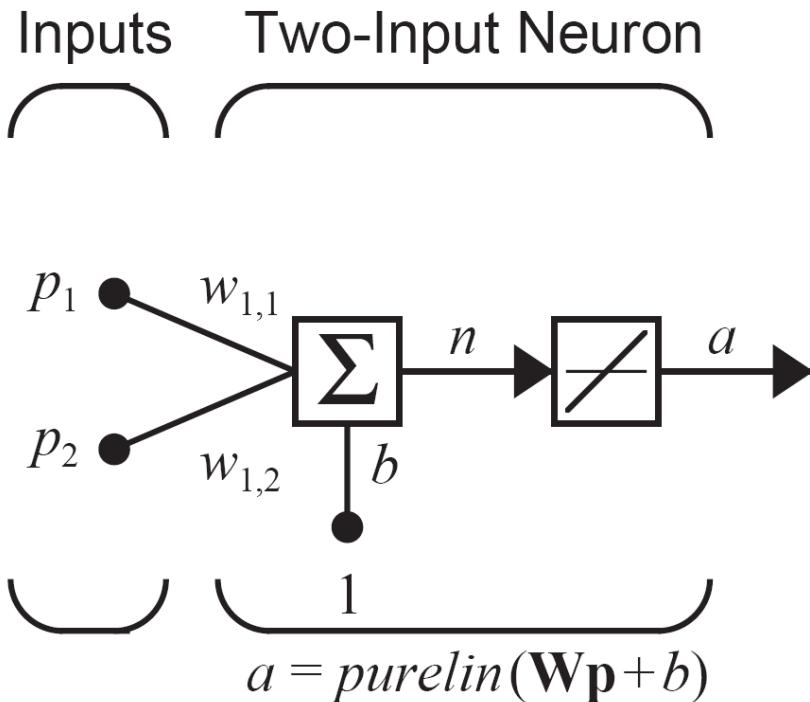
$$a_i = \text{purelin}(n_i) = \text{purelin}({}_i\mathbf{w}^T \mathbf{p} + b_i) = {}_i\mathbf{w}^T \mathbf{p} + b_i$$

${}_i\mathbf{w}$  is made up of the elements of the  $i$ th row of  $\mathbf{W}$ :

$${}_i\mathbf{w} = \begin{bmatrix} w_{i,1} \\ w_{i,2} \\ \vdots \\ w_{i,R} \end{bmatrix}$$

4

# Two-Input ADALINE



$$a = \text{purelin}(n) = \text{purelin}(\mathbf{1w}^T \mathbf{p} + b) = \mathbf{1w}^T \mathbf{p} + b$$

$$a = \mathbf{1w}^T \mathbf{p} + b = w_{1,1}p_1 + w_{1,2}p_2 + b$$

The ADALINE like perceptron has a *decision boundary*, which is determined by the input vectors for which the net input  $n$  is *zero*.

# Mean Square Error

The LMS algorithm is an example of supervised training.

Training Set:  $\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \dots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$

Input:  $\mathbf{p}_q$       Target:  $\mathbf{t}_q$

Notation:

$$\mathbf{x} = \begin{bmatrix} \mathbf{w} \\ 1 \\ b \end{bmatrix} \quad \mathbf{z} = \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} \quad a = {}_1 \mathbf{w}^T \mathbf{p} + b \quad \longrightarrow \quad a = \mathbf{x}^T \mathbf{z}$$

Mean Square Error:

$$F(\mathbf{x}) = E[e^2] = E[(t - a)^2] = E[(t - \mathbf{x}^T \mathbf{z})^2]$$

The expectation is taken over all sets of input/target pairs,

# Error Analysis

$$F(\mathbf{x}) = E[e^2] = E[(t - a)^2] = E[(t - \mathbf{x}^T \mathbf{z})^2]$$

$$F(\mathbf{x}) = E[t^2 - 2t\mathbf{x}^T \mathbf{z} + \mathbf{x}^T \mathbf{z} \mathbf{z}^T \mathbf{x}]$$

$$F(\mathbf{x}) = E[t^2] - 2\mathbf{x}^T E[t\mathbf{z}] + \mathbf{x}^T E[\mathbf{z} \mathbf{z}^T] \mathbf{x}$$

This can be written in the following convenient form:

$$F(\mathbf{x}) = c - 2\mathbf{x}^T \mathbf{h} + \mathbf{x}^T \mathbf{R} \mathbf{x}$$

where

$$c = E[t^2] \quad \mathbf{h} = E[t\mathbf{z}] \quad \mathbf{R} = E[\mathbf{z} \mathbf{z}^T] \quad ,$$

- The vector  $\mathbf{h}$  gives the cross-correlation between the input vector and its associated target.
- $\mathbf{R}$  is the input correlation matrix.
- The diagonal elements of this matrix are equal to the mean square values of the elements of the input vectors.

*The mean square error for the ADALINE Network is a quadratic function:*

$$F(\mathbf{x}) = c + \mathbf{d}^T \mathbf{x} + \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x}$$

$$\mathbf{d} = -2\mathbf{h} \quad \mathbf{A} = 2\mathbf{R}$$

# Stationary Point

Hessian Matrix:  $\mathbf{A} = 2\mathbf{R}$

- The correlation matrix  $\mathbf{R}$  must be at least positive semidefinite. Really it can be shown that all correlation matrices are either positive definite or positive semidefinite. If there are any zero eigenvalues, the performance index will either have a weak minimum or else no stationary point (depending on  $\mathbf{d} = -2\mathbf{h}$ ), otherwise there will be a unique global minimum  $\mathbf{x}^*$  (see Ch8).

$$\nabla F(\mathbf{x}) = \nabla \left( c + \mathbf{d}^T \mathbf{x} + \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} \right) = \mathbf{d} + \mathbf{A} \mathbf{x} = -2\mathbf{h} + 2\mathbf{R} \mathbf{x}$$

Stationary point:  $-2\mathbf{h} + 2\mathbf{R} \mathbf{x} = \mathbf{0}$

If  $\mathbf{R}$  (the correlation matrix) is positive definite:

$$\mathbf{x}^* = \mathbf{R}^{-1} \mathbf{h}$$

- If we could calculate the statistical quantities  $\mathbf{h}$  and  $\mathbf{R}$ , we could find the minimum point directly from above equation.
- But it is not desirable or convenient to calculate  $\mathbf{h}$  and  $\mathbf{R}$ . So ...

# Approximate Steepest Descent

Approximate mean square error (one sample):

$$\hat{F}(\mathbf{x}) = (t(k) - a(k))^2 = e^2(k)$$

Expectation of the squared error has been replaced by the squared error at iteration k.

Approximate (stochastic) gradient:

$$\hat{\nabla} F(\mathbf{x}) = \nabla e^2(k)$$

$$[\nabla e^2(k)]_j = \frac{\partial e^2(k)}{\partial w_{1,j}} = 2e(k) \frac{\partial e(k)}{\partial w_{1,j}} \quad j = 1, 2, \dots, R$$

$$[\nabla e^2(k)]_{R+1} = \frac{\partial e^2(k)}{\partial b} = 2e(k) \frac{\partial e(k)}{\partial b}$$

# Approximate Gradient Calculation

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial [t(k) - a(k)]}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}} [t(k) - (w_1^T p(k) + b)]$$

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}} \left[ t(k) - \left( \sum_{i=1}^R w_{1,i} p_i(k) + b \right) \right]$$

Where  $p_i(k)$  is the  $i$ th elements of the input vector at  $k$ th iteration.

$$\frac{\partial e(k)}{\partial w_{1,j}} = -p_j(k) \quad \frac{\partial e(k)}{\partial b} = -1$$

$$\hat{\nabla} F(\mathbf{x}) = \nabla e^2(k) = -2e(k)\mathbf{z}(k)$$

- Now we can see the beauty of approximating the mean square error by the single error at iteration  $k$  as in:

$$\hat{F}(\mathbf{x}) = (t(k) - a(k))^2 = e^2(k)$$

- This approximation to  $\nabla F(\mathbf{x})$  can now be used in the Steepest descent algorithm.

## LMS Algorithm

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \nabla F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}_k}$$

If we substitute  $\hat{\nabla}F(\mathbf{x})$  for  $\nabla F(\mathbf{x})$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + 2\alpha e(k) \mathbf{z}(k)$$

$$_1\mathbf{w}(k+1) = _1\mathbf{w}(k) + 2\alpha e(k) \mathbf{p}(k)$$

$$b(k+1) = b(k) + 2\alpha e(k)$$

These last two equations make up the LMS algorithm.  
Also called *Delta Rule* or the *Widrow-Hoff* learning  
algorithm.

# Multiple-Neuron Case

$$_i\mathbf{w}(k+1) = _i\mathbf{w}(k) + 2\alpha e_i(k)\mathbf{p}(k)$$

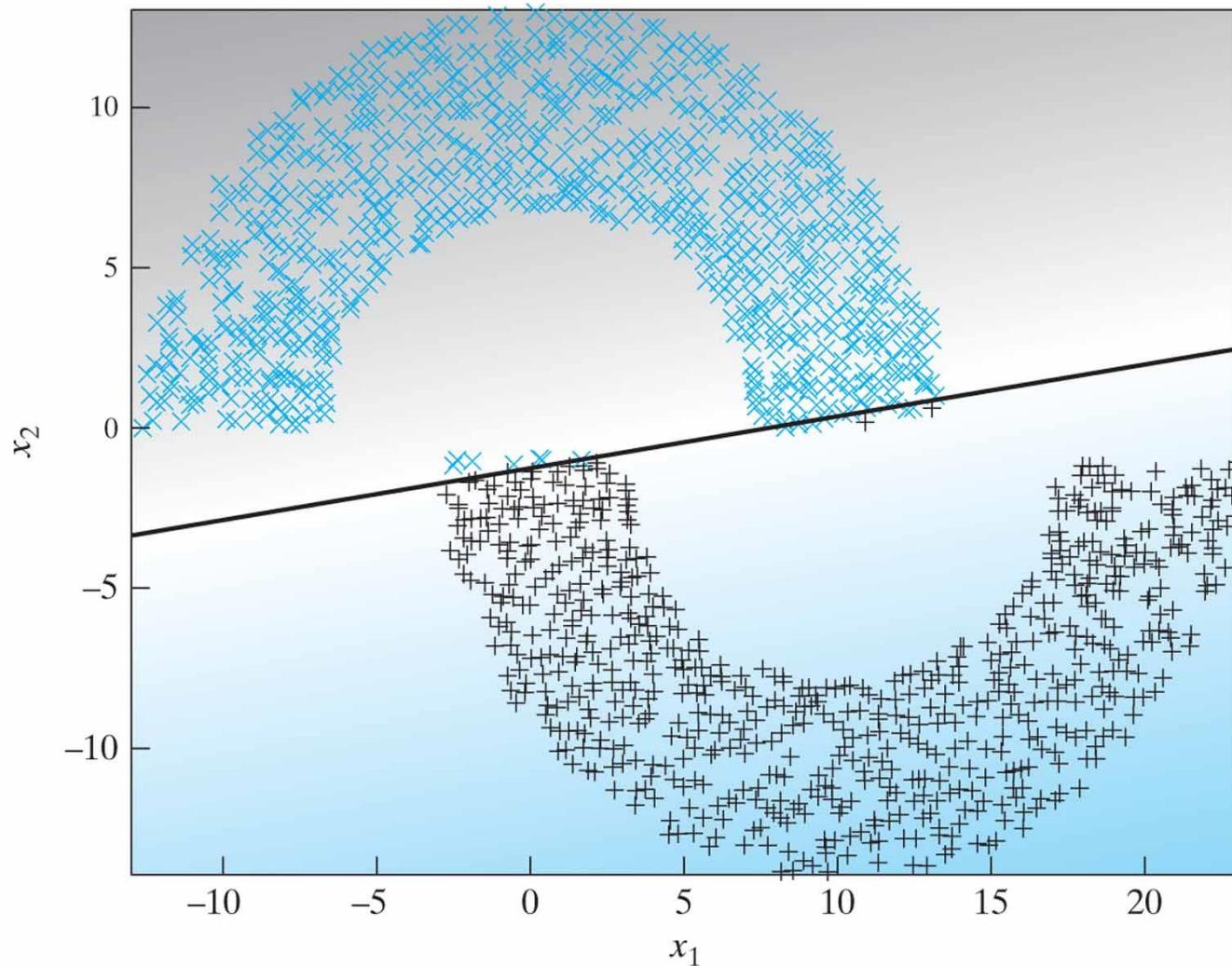
$$b_i(k+1) = b_i(k) + 2\alpha e_i(k)$$

Matrix Form:

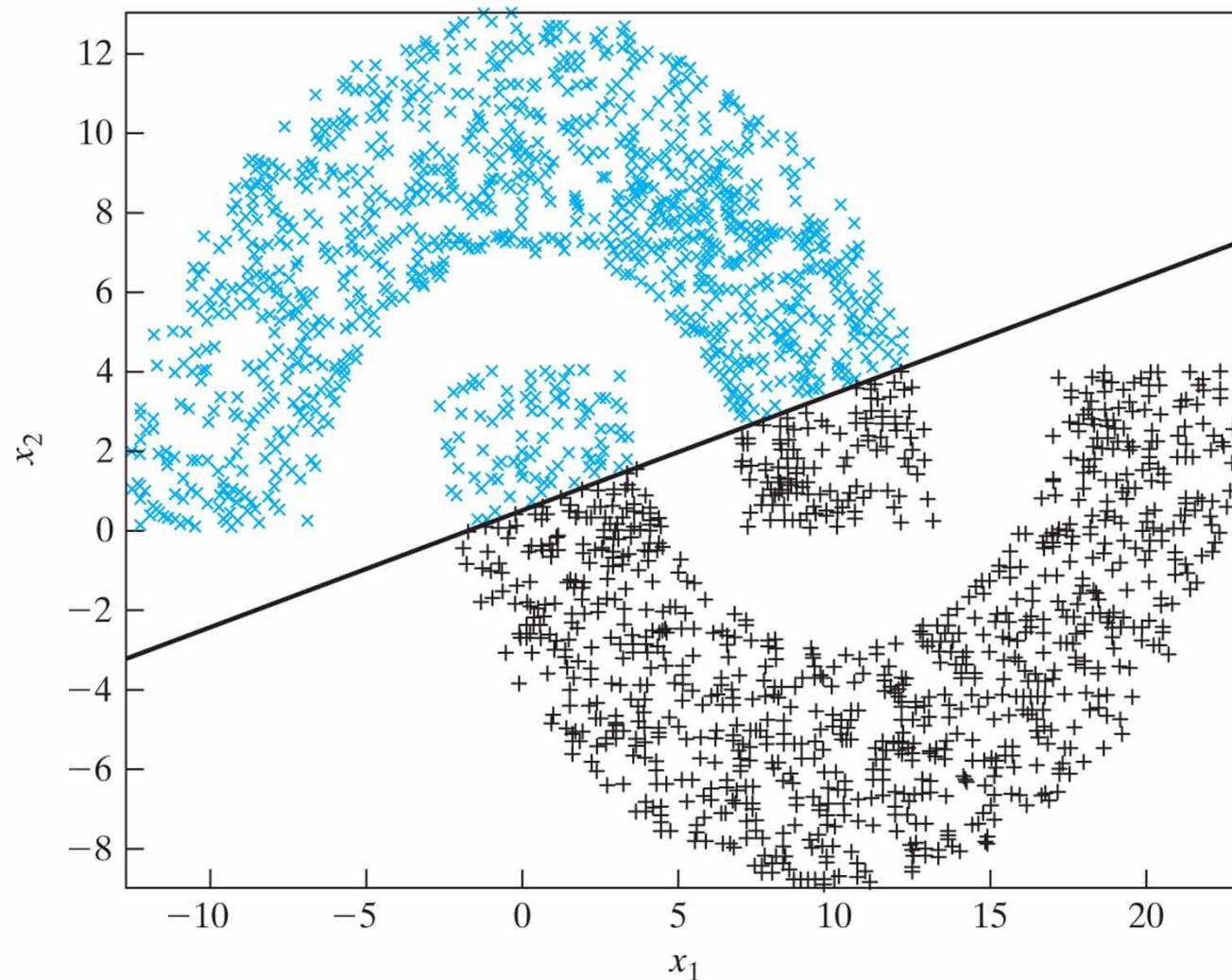
$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha \mathbf{e}(k) \mathbf{p}^T(k)$$

$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha \mathbf{e}(k)$$

Classification using LMS with distance = 1, radius = 10, and width = 6



Classification using LMS with distance = -4, radius = 10, and width = 6



# Stability condition

Note: we have the same condition as the SD algorithm. In SD we use the Hessian Matrix  $\mathbf{A}$ , here we use the input correlation matrix  $\mathbf{R}$  (Recall that  $\mathbf{A}=2\mathbf{R}$ ).

$$1 - 2\alpha\lambda_i > -1$$

(where  $\lambda_i$  is an eigenvalue of  $\mathbf{R}$ )

$$\alpha < 1/\lambda_i \quad \text{for all } i$$

$$0 < \alpha < 1/\lambda_{max}$$

Thus the LMS solution, obtained by applying one input at a time, is the same as the minimum mean square solution of  $\mathbf{x}^* = \mathbf{R}^{-1}\mathbf{h}$

# Example

$$\text{Banana} \quad \left\{ \mathbf{p}_1 = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \right\} \quad \text{Apple} \quad \left\{ \mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}$$

If inputs are generated randomly with equal probability, the input correlation matrix is:

$$\mathbf{R} = E[\mathbf{p}\mathbf{p}^T] = \frac{1}{2}\mathbf{p}_1\mathbf{p}_1^T + \frac{1}{2}\mathbf{p}_2\mathbf{p}_2^T$$

$$\mathbf{R} = \frac{1}{2} \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} \begin{bmatrix} -1 & 1 & -1 \end{bmatrix}^T + \frac{1}{2} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} \begin{bmatrix} 1 & 1 & -1 \end{bmatrix}^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & -1 & 1 \end{bmatrix}$$

$$\lambda_1 = 1.0, \quad \lambda_2 = 0.0, \quad \lambda_3 = 2.0 \quad \alpha < \frac{1}{\lambda_{max}} = \frac{1}{2.0} = 0.5$$

We take  $\alpha=0.2$  (Note: Practically it is difficult to calculate  $\mathbf{R}$  and  $\alpha$ . We choose them by trial and error).

# Iteration One

Banana

$$a(0) = \mathbf{W}(0)\mathbf{p}(0) = \mathbf{W}(0)\mathbf{p}_1 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} = 0$$

$\mathbf{W}(0)$  is selected arbitrarily.

$$e(0) = t(0) - a(0) = t_1 - a(0) = -1 - 0 = -1$$

$$\mathbf{W}(1) = \mathbf{W}(0) + 2\alpha e(0)\mathbf{p}^T(0)$$

$$\mathbf{W}(1) = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} + 2(0.2)(-1) \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}^T = \begin{bmatrix} 0.4 & -0.4 & 0.4 \end{bmatrix}$$

# Iteration Two

Apple     $a(1) = \mathbf{W}(1)\mathbf{p}(1) = \mathbf{W}(1)\mathbf{p}_2 = \begin{bmatrix} 0.4 & -0.4 & 0.4 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} = -0.4$

$$e(1) = t(1) - a(1) = t_2 - a(1) = 1 - (-0.4) = 1.4$$

$$\mathbf{W}(2) = \begin{bmatrix} 0.4 & -0.4 & 0.4 \end{bmatrix} + 2(0.2)(1.4) \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}^T = \begin{bmatrix} 0.96 & 0.16 & -0.16 \end{bmatrix}$$

# Iteration Three

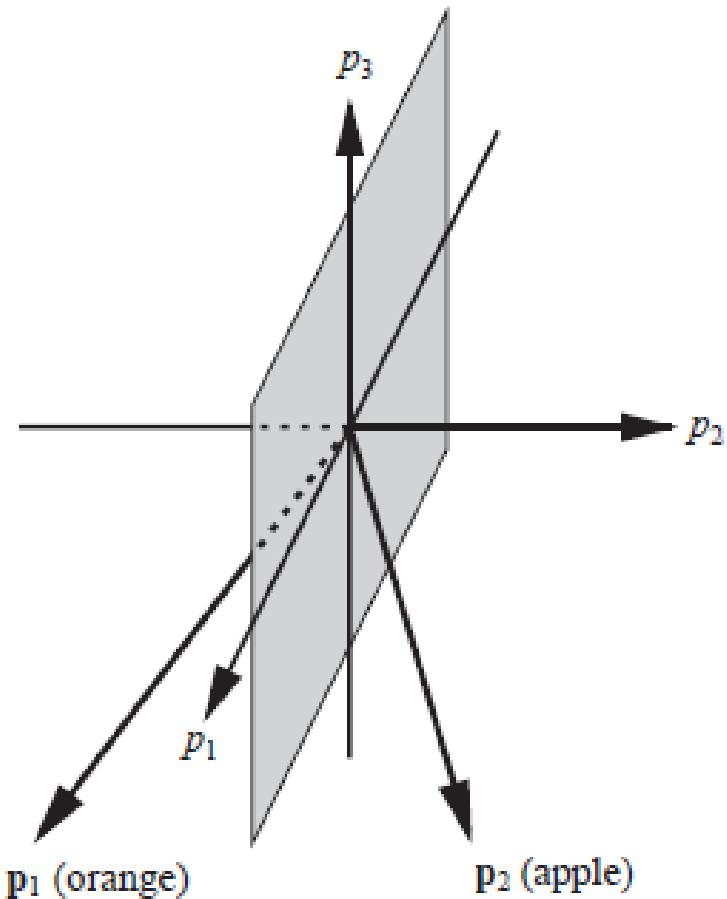
$$a(2) = \mathbf{W}(2)\mathbf{p}(2) = \mathbf{W}(2)\mathbf{p}_1 = \begin{bmatrix} 0.96 & 0.16 & -0.16 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} = -0.64$$

$$e(2) = t(2) - a(2) = t_1 - a(2) = -1 - (-0.64) = -0.36$$

$$\mathbf{W}(3) = \mathbf{W}(2) + 2\alpha e(2)\mathbf{p}^T(2) = \begin{bmatrix} 1.1040 & 0.0160 & -0.0160 \end{bmatrix}$$

$$\mathbf{W}(\infty) = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$

# Decision Boundary



# Some general comments on the learning process

- Computationally, the learning process goes through all training examples (an epoch) number of times, until a **stopping criterion** is reached.
- The convergence process can be monitored with the plot of the mean-squared error function  $F(\mathbf{W}(k))$ .

# The popular stopping criteria are:

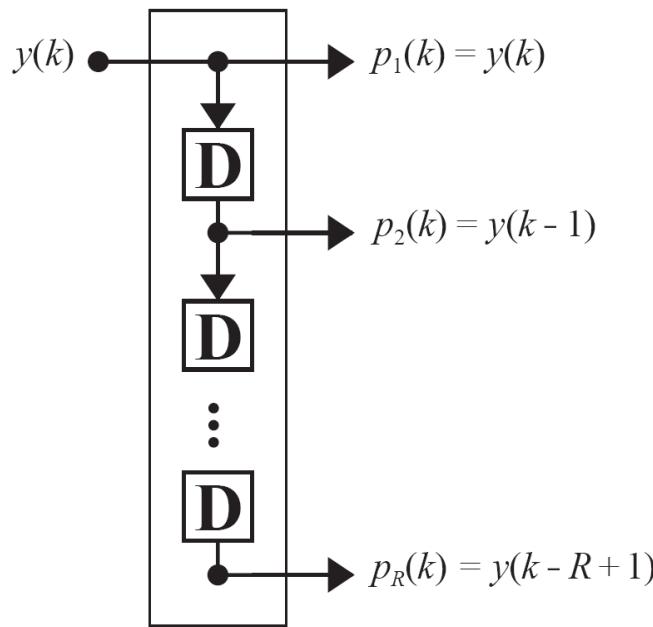
- the mean-squared error is sufficiently small:  $F(\mathbf{W}(k)) < \varepsilon$
- The rate of change of the mean-squared error is sufficiently small:

$$\frac{\partial F(\mathbf{W}(k))}{\partial k} < \varepsilon$$

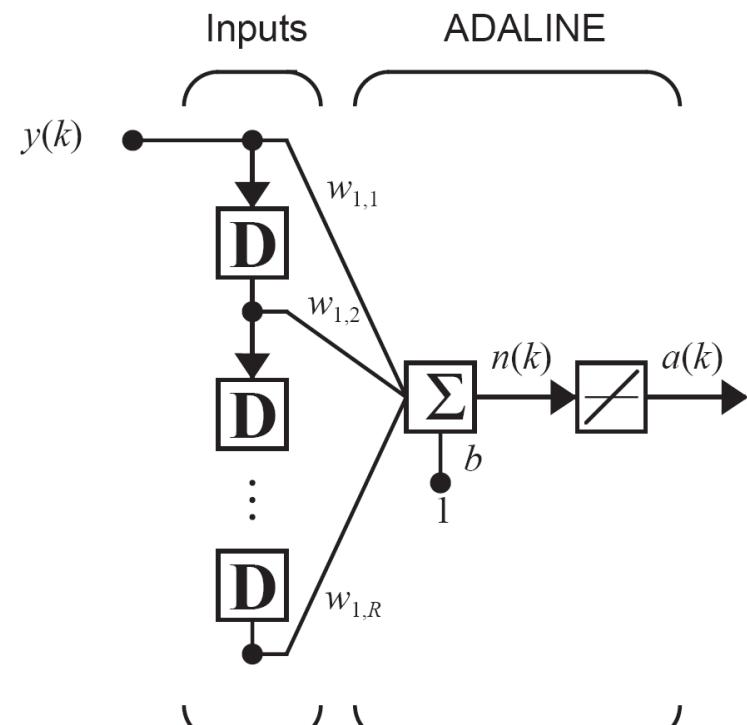
# Adaptive Filtering

ADALINE is one of the most widely used NNs in practical applications. One of the major application areas has been Adaptive Filtering.

Tapped Delay Line



Adaptive Filter

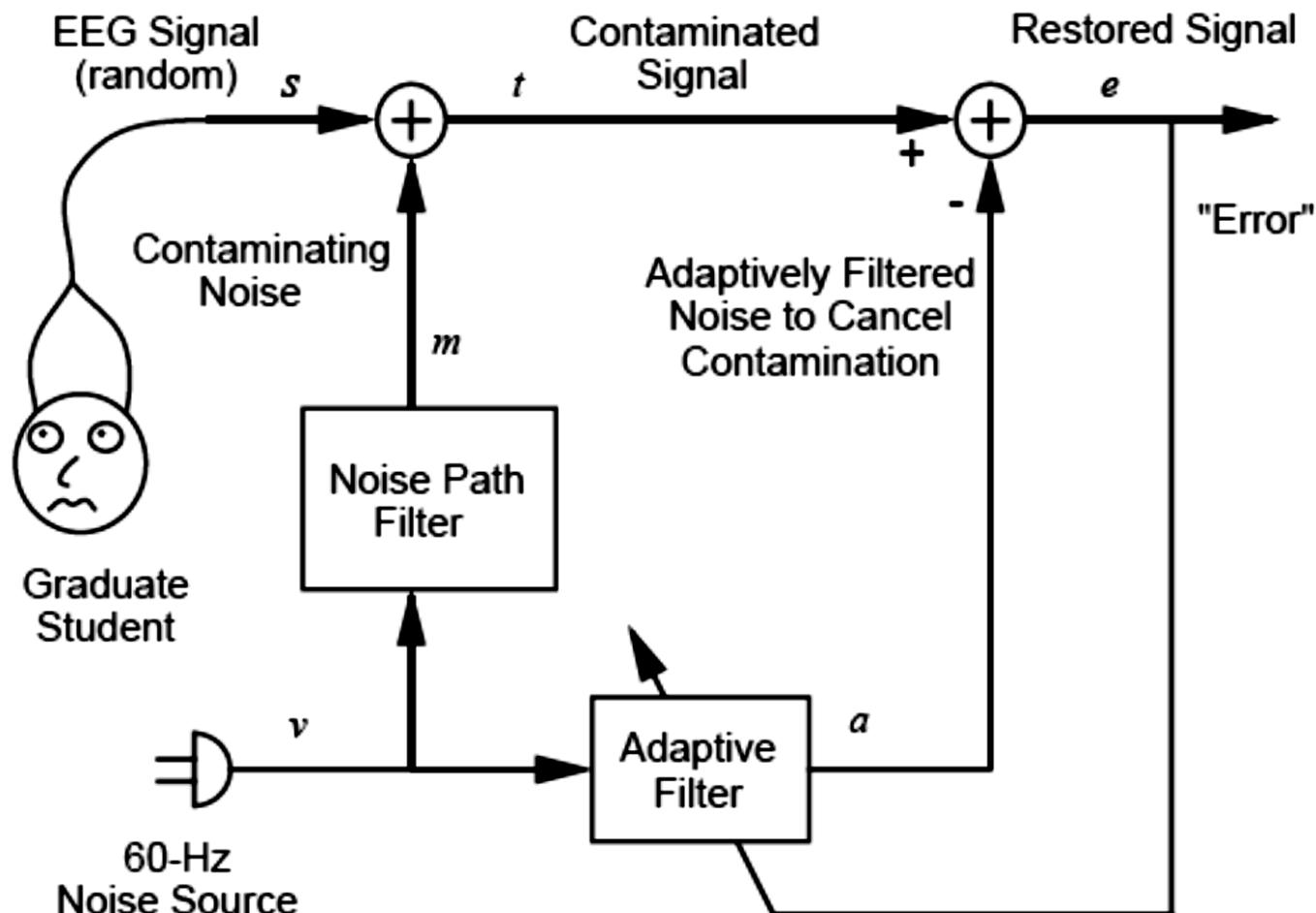


$$a(k) = \text{purelin}(\mathbf{W}\mathbf{p}(k) + b)$$

$$a(k) = \text{purelin}(\mathbf{W}\mathbf{p} + b) = \sum_{i=1}^R w_{1,i}y(k-i+1) + b$$

In Digital Signal Processing (DSP) language we recognize this network as a finite impulse response (FIR) filter.

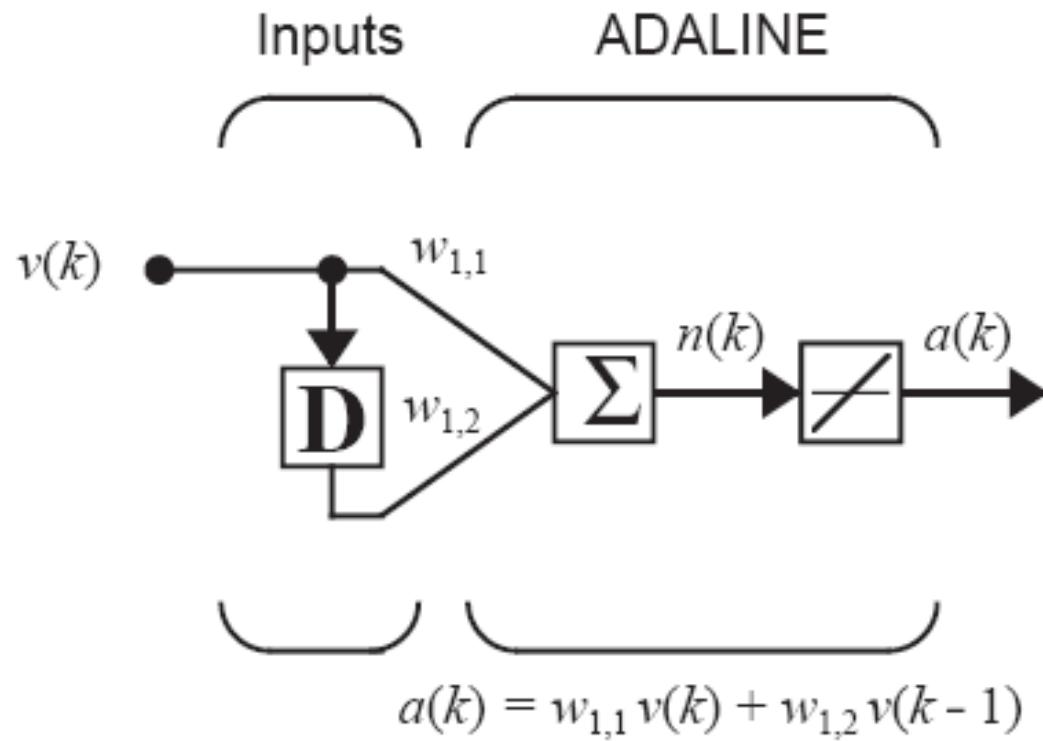
# Example: Noise Cancellation



Adaptive Filter Adjusts to Minimize Error (and in doing this removes 60-Hz noise from contaminated signal)

# Noise Cancellation Adaptive Filter

Two-input filter can attenuate and phase-shift the noise  $v$  in the desired way.



# Correlation Matrix

To Analyze this system we need to find the input correlation matrix  $\mathbf{R}$  and the input/target cross-correlation vector  $\mathbf{h}$ .

$$\mathbf{R} = E[\mathbf{z}\mathbf{z}^T] \quad \mathbf{h} = E[t\mathbf{z}]$$

$$\mathbf{z}(k) = \begin{bmatrix} v(k) \\ v(k-1) \end{bmatrix} \quad t(k) = s(k) + m(k)$$

$$\mathbf{R} = \begin{bmatrix} E[v^2(k)] & E[v(k)v(k-1)] \\ E[v(k-1)v(k)] & E[v^2(k-1)] \end{bmatrix}$$

$$\mathbf{h} = \begin{bmatrix} E[(s(k) + m(k))v(k)] \\ E[(s(k) + m(k))v(k-1)] \end{bmatrix}$$

- We must define the noise signal  $v$ , the EEG signal  $s$ , and the filtered noise  $m$ , to be able to obtain specific values.
  - We assume: The EEG signal is a white (Uncorrelated from one time step to the next) random signal uniformly distributed between the values -0.2 and +0.2, the noise source (60 Hz sine wave sampled at 180 Hz) is given by
- $$v(k) = 1.2 \sin\left(\frac{2\pi k}{3}\right)$$
- And the filtered noise that contaminates the EEG is the noise attenuated by a factor 1.0 and shifted in phase by  $-3\pi/4$ :

$$m(k) = 1.2 \sin\left(\frac{2\pi k}{3} - \frac{3\pi}{4}\right)$$

$$E[v^2(k)] = (1.2)^2 \frac{1}{3} \sum_{k=1}^3 \left( \sin\left(\frac{2\pi k}{3}\right) \right)^2 = (1.2)^2 0.5 = 0.72$$

$$E[v^2(k-1)] = E[v^2(k)] = 0.72$$

$$E[v(k)v(k-1)] = \frac{1}{3} \sum_{k=1}^3 \left( 1.2 \sin\frac{2\pi k}{3} \right) \left( 1.2 \sin\frac{2\pi(k-1)}{3} \right)$$

$$= (1.2)^2 0.5 \cos\left(\frac{2\pi}{3}\right) = -0.36$$

$$\mathbf{R} = \begin{bmatrix} 0.72 & -0.36 \\ -0.36 & 0.72 \end{bmatrix}$$

# Stationary Point

$$E[(s(k) + m(k))v(k)] = E[s(k)v(k)] + \underbrace{E[m(k)v(k)]}_0$$

The 1<sup>st</sup> term is zero because  $s(k)$  and  $v(k)$  are independent and zero mean.

$$E[m(k)v(k)] = \frac{1}{3} \sum_{k=1}^3 \left( 1.2 \sin\left(\frac{2\pi k}{3} - \frac{3\pi}{4}\right) \right) \left( 1.2 \sin\frac{2\pi k}{3} \right) = -0.51$$

Now we find the 2<sup>nd</sup> element of  $\mathbf{h}$ :

$$E[(s(k) + m(k))v(k-1)] = E[s(k)v(k-1)] + \underbrace{E[m(k)v(k-1)]}_0$$

$$E[m(k)v(k-1)] = \frac{1}{3} \sum_{k=1}^3 \left(1.2 \sin\left(\frac{2\pi k}{3} - \frac{3\pi}{4}\right)\right) \left(1.2 \sin\frac{2\pi(k-1)}{3}\right) = 0.70$$

$$\mathbf{h} = \begin{bmatrix} E[(s(k) + m(k))v(k)] \\ E[(s(k) + m(k))v(k-1)] \end{bmatrix} \quad \longrightarrow \quad \mathbf{h} = \begin{bmatrix} -0.51 \\ 0.70 \end{bmatrix}$$

$$\mathbf{x}^* = \mathbf{R}^{-1}\mathbf{h} = \begin{bmatrix} 0.72 & -0.36 \\ -0.36 & 0.72 \end{bmatrix}^{-1} \begin{bmatrix} -0.51 \\ 0.70 \end{bmatrix} = \begin{bmatrix} -0.30 \\ 0.82 \end{bmatrix}$$

Now, what kind of error will we have at the minimum solution?

# Performance Index

$$F(\mathbf{x}) = c - 2\mathbf{x}^T \mathbf{h} + \mathbf{x}^T \mathbf{R} \mathbf{x}$$

We have just found  $\mathbf{x}^*$ ,  $\mathbf{R}$  and  $\mathbf{h}$ . To find  $c$  we have

$$c = E[t^2(k)] = E[(s(k) + m(k))^2]$$

$$c = E[s^2(k)] + 2E[s(k)m(k)] + E[m^2(k)]$$

The middle term is zero because  $s(k)$  and  $m(k)$  are independent and zero mean.

$$E[s^2(k)] = \frac{1}{0.4} \int_{-0.2}^{0.2} s^2 ds = \frac{1}{3(0.4)} s^3 \Big|_{-0.2}^{0.2} = 0.0133$$

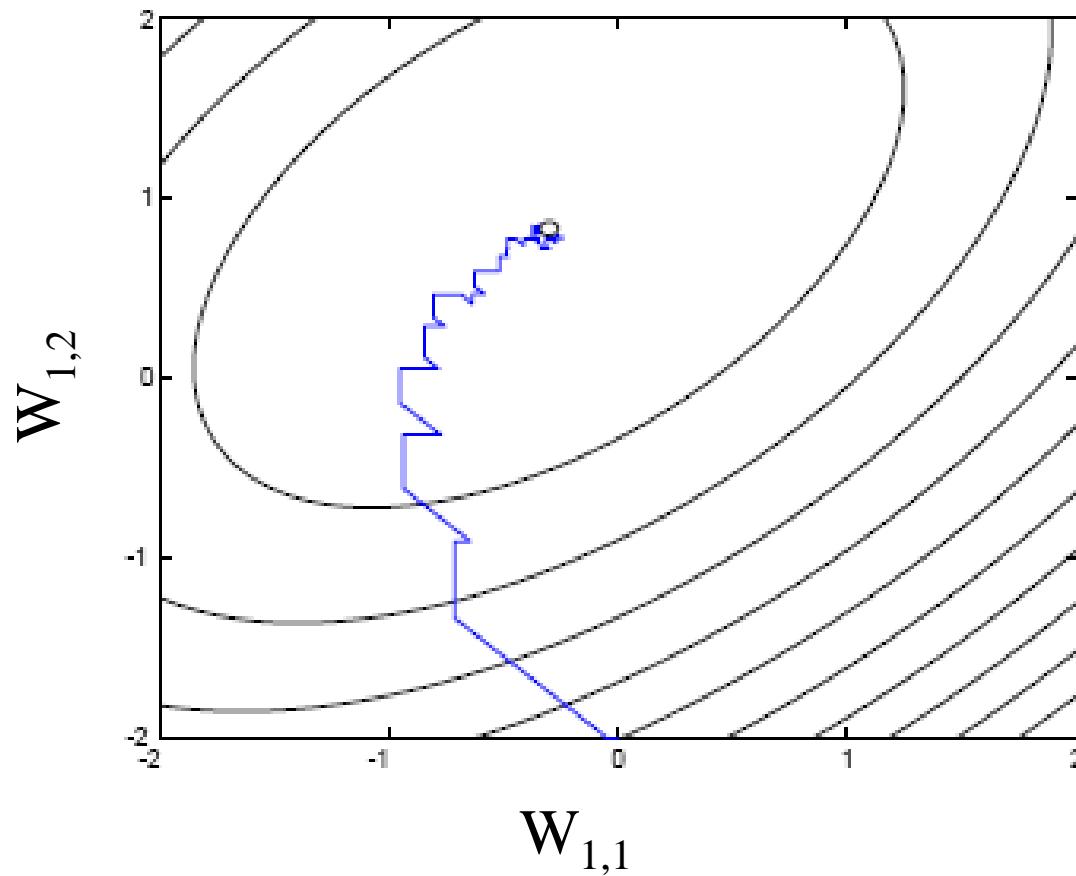
$$E[m^2(k)] = \frac{1}{3} \sum_{k=1}^3 \left\{ 1.2 \sin\left(\frac{2\pi}{3} - \frac{3\pi}{4}\right) \right\}^2 = 0.72$$

$$c = 0.0133 + 0.72 = 0.7333$$

$$F(\mathbf{x}^*) = 0.7333 - 2(0.72) + 0.72 = 0.0133$$

The minimum mean square error is the same as the mean square value of the EEG signal. This is what we expected, since the “**error**” of this adaptive noise canceller is in fact the reconstructed EEG Signal.

# LMS Response for $\alpha = 0.1$



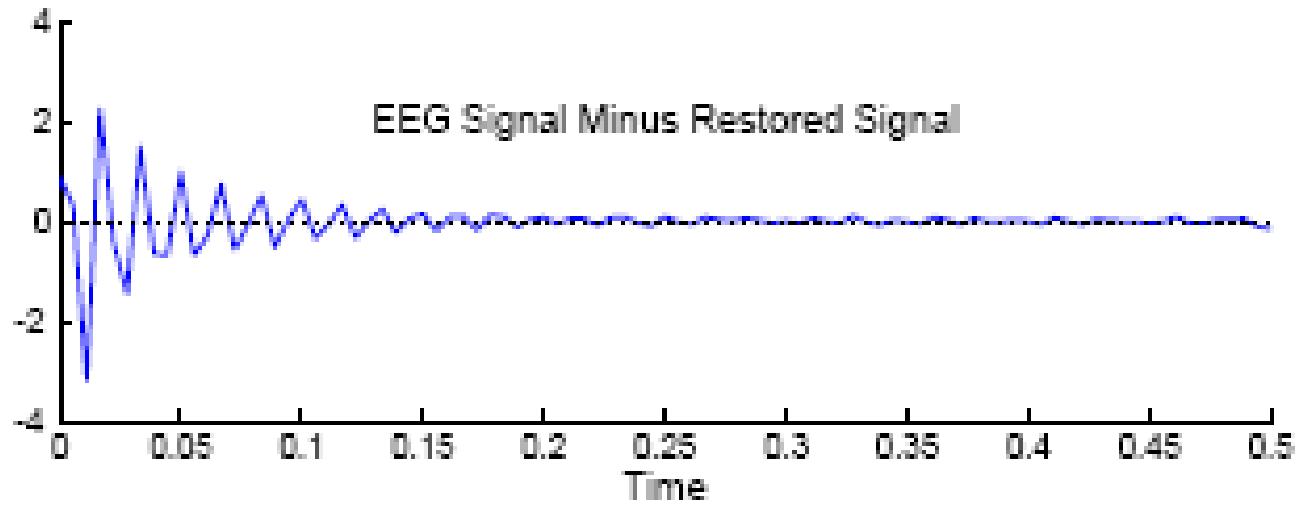
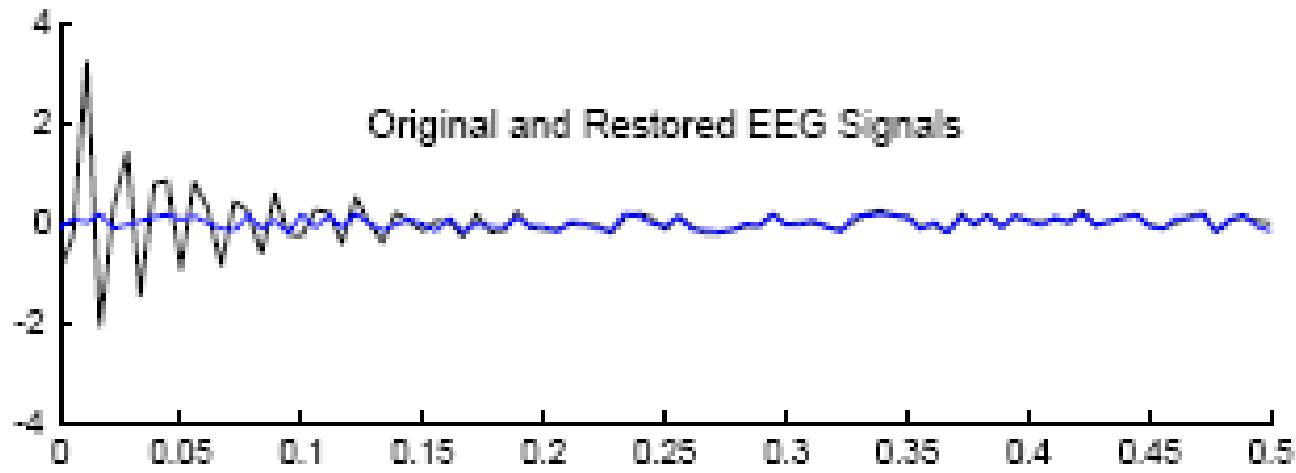
LMS trajectory looks like noisy version of steepest descent.

Note that the contours in this figure reflect the fact that the eigenvalues and the eigenvectors of the Hessian matrix  $\mathbf{A}=2\mathbf{R}$  are

$$\lambda_1 = 2.16, \quad z_1 = \begin{bmatrix} -0.7071 \\ 0.7071 \end{bmatrix}, \quad \lambda_2 = 0.75, \quad z_2 = \begin{bmatrix} -0.7071 \\ -0.7071 \end{bmatrix}$$

If the learning rate is decreased, the LMS trajectory is smoother, but the learning proceed more slowly.

Note that  $\alpha_{\max}$  is  $2/2.16=0.926$  for stability.



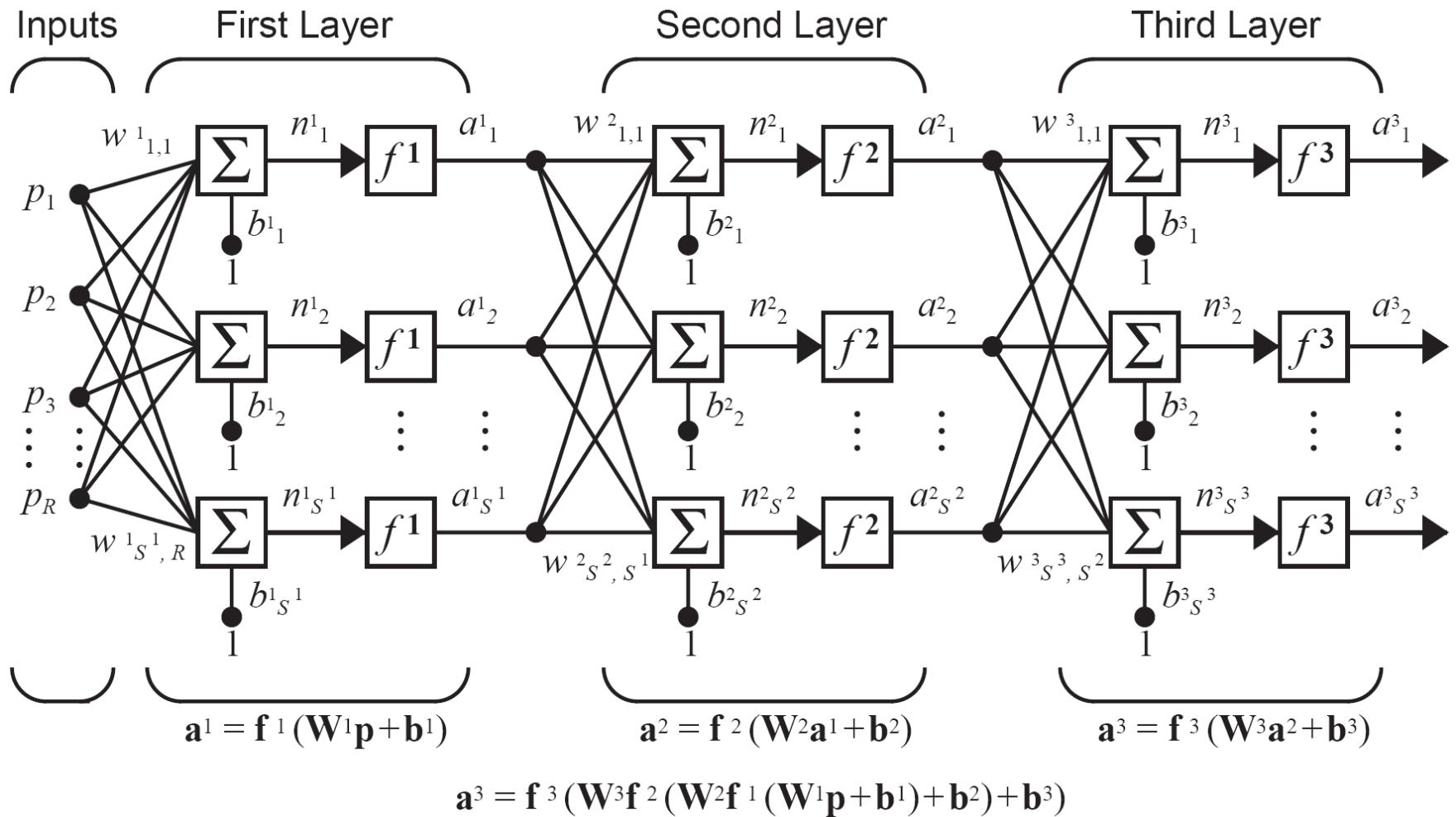
Note that error does not go to zero, because the LMS algorithm is approximate steepest descent; it uses an estimate of the gradient, not the true gradient.

[nnd10eeg](#)

# Backpropagation

- Backpropagation is a generalization of LMS algorithm and can be used to train multilayer networks.
- So backpropagation is an approximate steepest descent algorithm, in which the performance index is mean square error.
- In *multilayer* networks with *nonlinear* transfer functions, the relationship between the network weights and the error is more complex than a single-layer *linear* network.

# Multilayer Perceptron

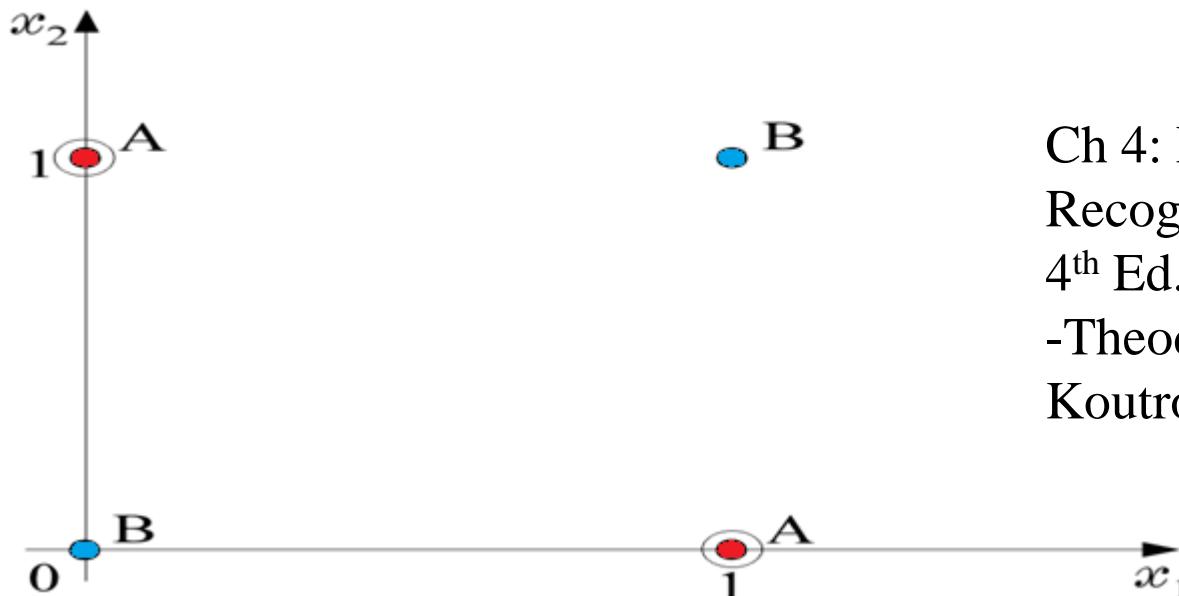


R – S<sup>1</sup> – S<sup>2</sup> – S<sup>3</sup> Network

# Non Linear Classifiers

- The XOR problem

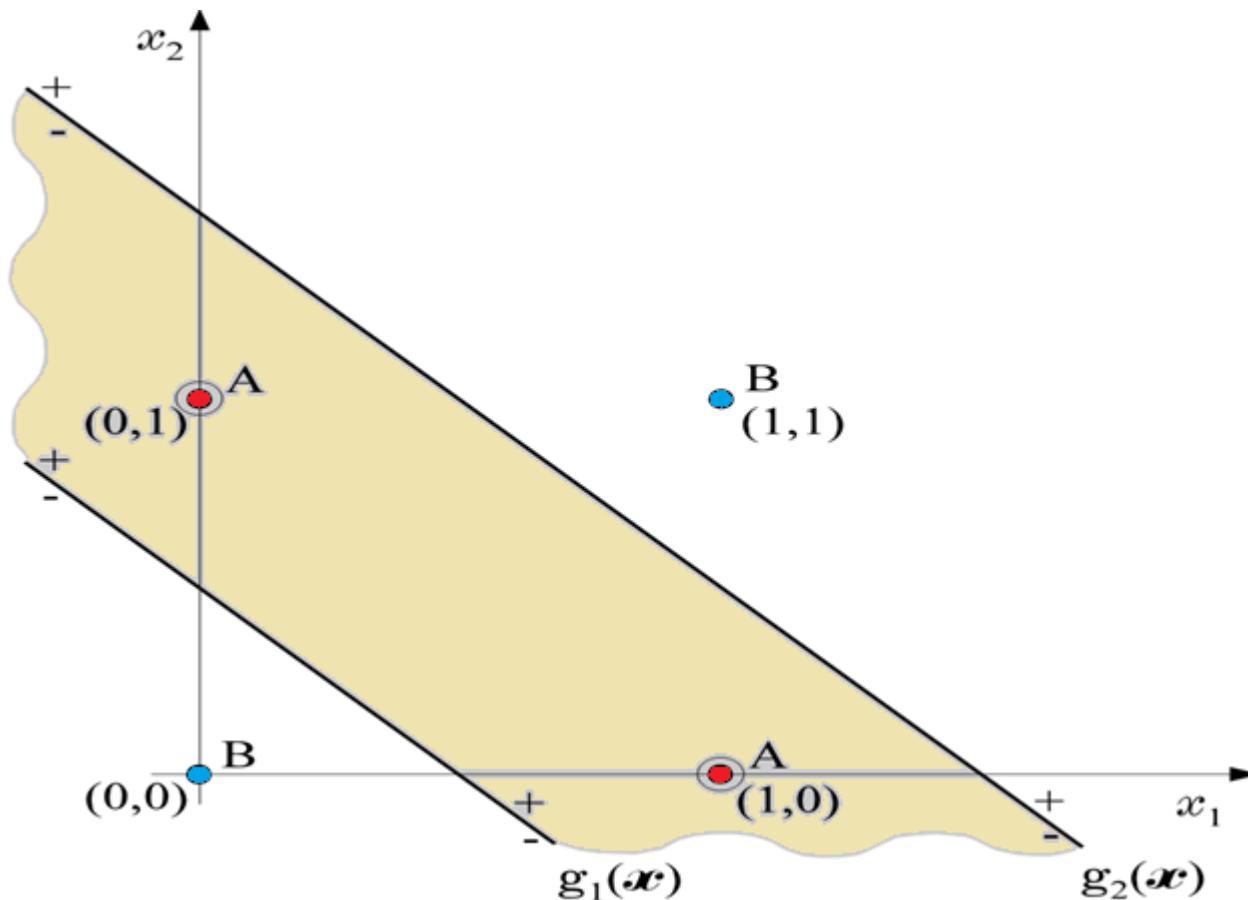
| <b><math>x_1</math></b> | <b><math>x_2</math></b> | <b>XOR</b> | <b>Class</b> |
|-------------------------|-------------------------|------------|--------------|
| 0                       | 0                       | 0          | B            |
| 0                       | 1                       | 1          | A            |
| 1                       | 0                       | 1          | A            |
| 1                       | 1                       | 0          | B            |



Ch 4: Pattern  
Recognition.  
 $4^{\text{th}}$  Ed.  
-Theodoridis-  
Koutroumbas

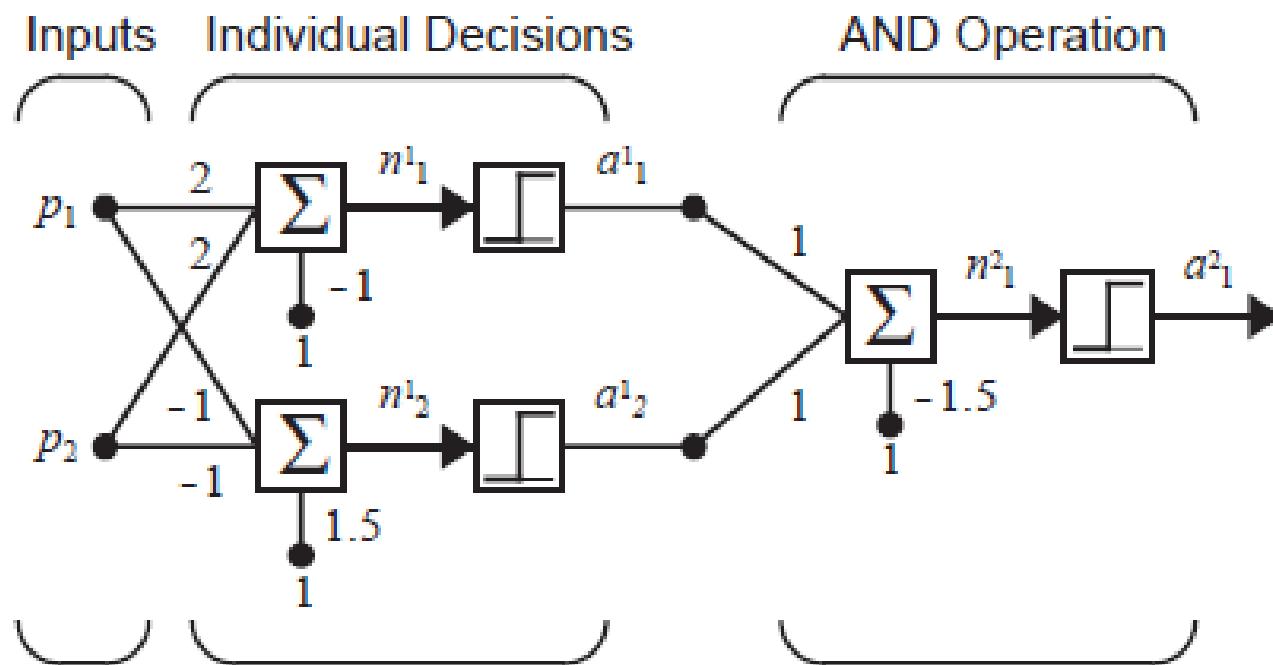
## • The Two-Layer Perceptron

- For the XOR problem, draw **two**, instead, of one lines

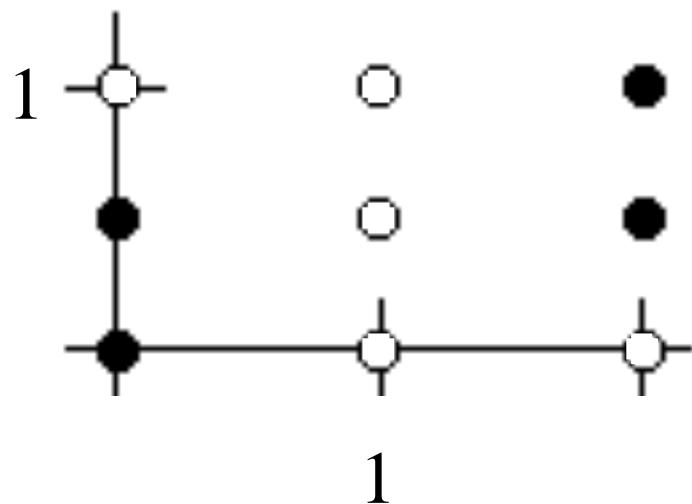


- Computations of the first phase perform a **mapping** that **transforms** the **nonlinearly separable** problem to a **linearly separable** one.

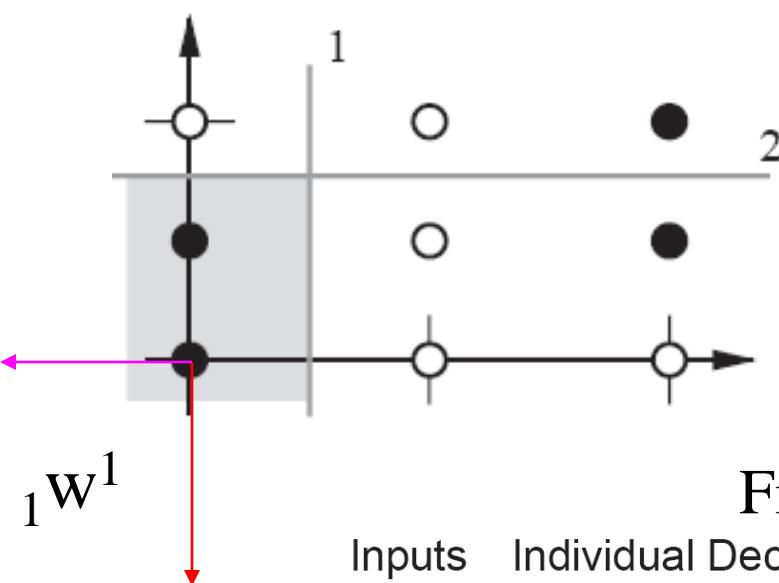
– The architecture



# Example



# Elementary Decision Boundaries



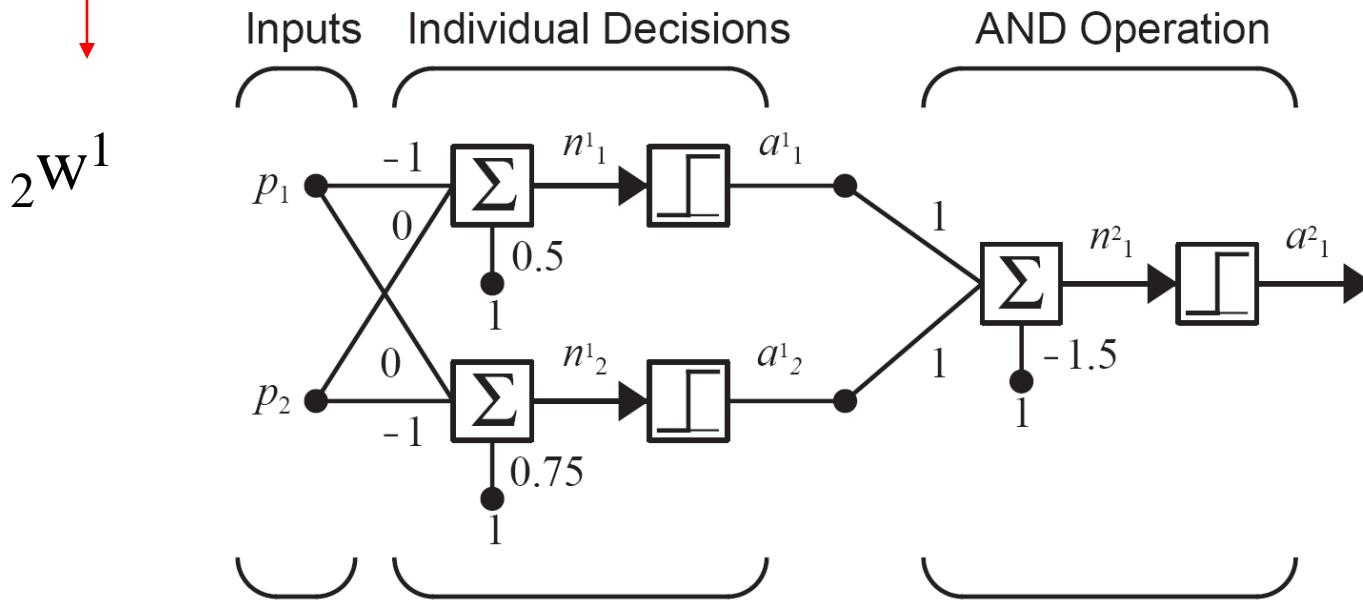
First Boundary:

$$a_1^1 = \text{hardlim}(\begin{bmatrix} -1 & 0 \end{bmatrix} \mathbf{p} + 0.5)$$

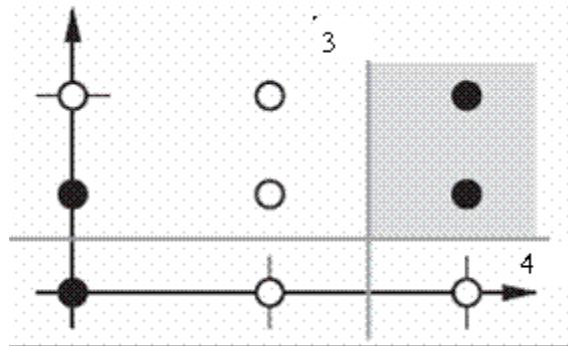
Second Boundary:

$$a_2^1 = \text{hardlim}(\begin{bmatrix} 0 & -1 \end{bmatrix} \mathbf{p} + 0.75)$$

First Subnetwork



# Elementary Decision Boundaries



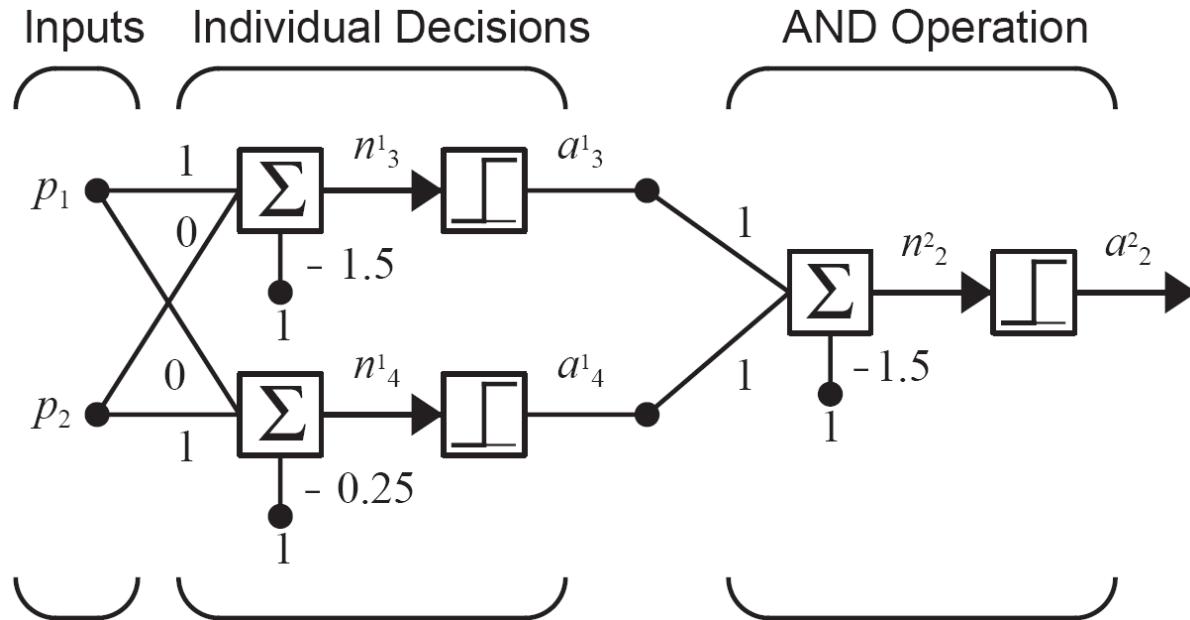
Third Boundary:

$$a_3^1 = \text{hardlim}(\begin{bmatrix} 1 & 0 \end{bmatrix} \mathbf{p} - 1.5)$$

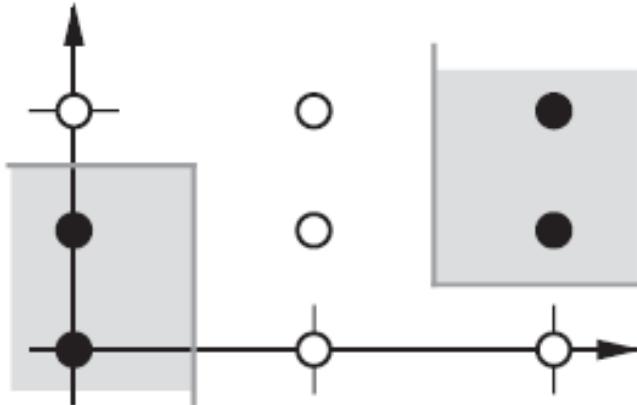
Fourth Boundary:

$$a_4^1 = \text{hardlim}(\begin{bmatrix} 0 & 1 \end{bmatrix} \mathbf{p} - 0.25)$$

Second Subnetwork



# Total Network



$$\mathbf{W}^1 = \begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

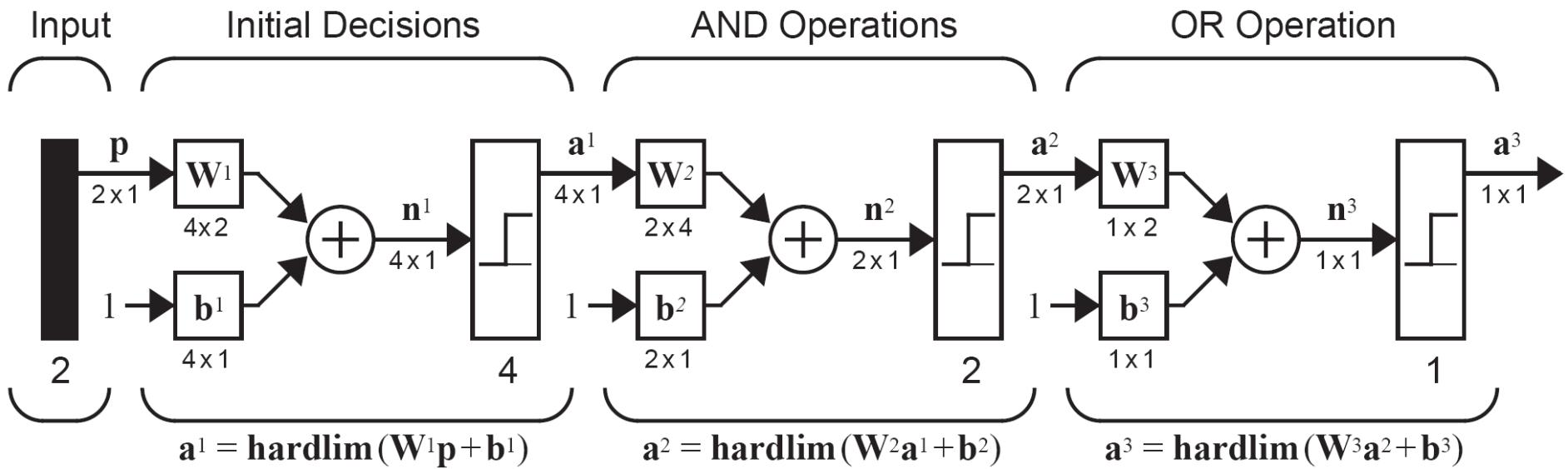
$$\mathbf{b}^1 = \begin{bmatrix} 0.5 \\ 0.75 \\ -1.5 \\ -0.25 \end{bmatrix}$$

$$\mathbf{W}^2 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

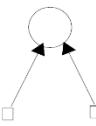
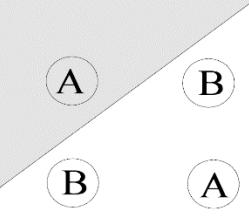
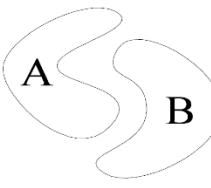
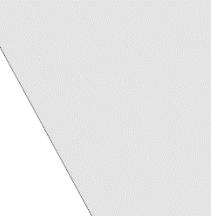
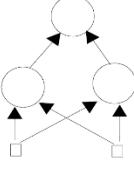
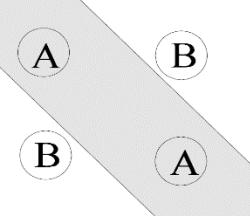
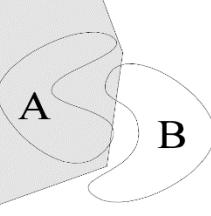
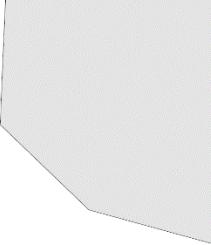
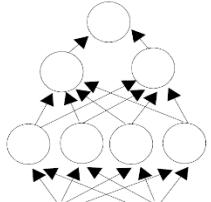
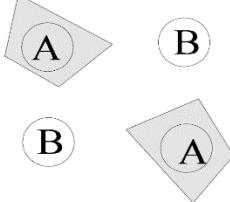
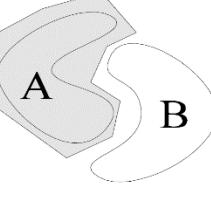
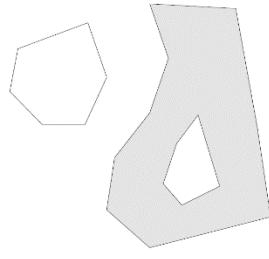
$$\mathbf{b}^2 = \begin{bmatrix} -1.5 \\ -1.5 \end{bmatrix}$$

$$\mathbf{W}^3 = \begin{bmatrix} 1 & 1 \end{bmatrix}$$

$$\mathbf{b}^3 = \begin{bmatrix} -0.5 \end{bmatrix}$$

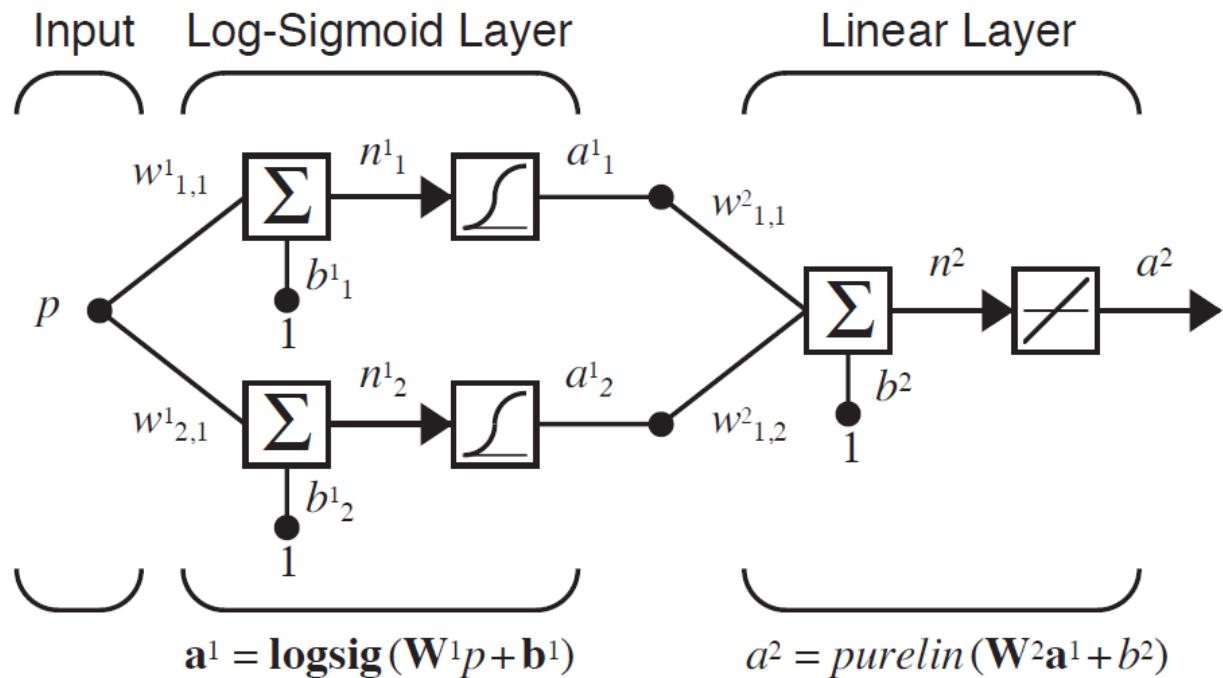


The types of decision regions that can be formed by single and multilayer perceptrons with one and two layers of hidden layers are given in the Figure 6.5 [Lipmann 87].

| STRUCTURE  | TYPES OF DECISION REGIONS  | EXCLUSIVE OR PROBLEM  | MOST GENERAL REGION SHAPES   |
|--|--|---|--|
|   |   |   |   |
|   |   |   |   |
|  |  |  |  |

# Function Approximation Example

- In control systems the objective is to find an appropriate feedback function that maps from measured outputs to control inputs.
- In adaptive filtering the objective is to find a function that maps from delayed values of an input signal to an appropriate output signal. The following example illustrate the flexibility of the multilayer perceptron for implementing functions.
- We consider the 1-2-1 network; the transfer function of the first layer is log-sigmoid and the transfer function of the 2<sup>nd</sup> layer is linear.



$$f^1(n) = \frac{1}{1 + e^{-n}}$$

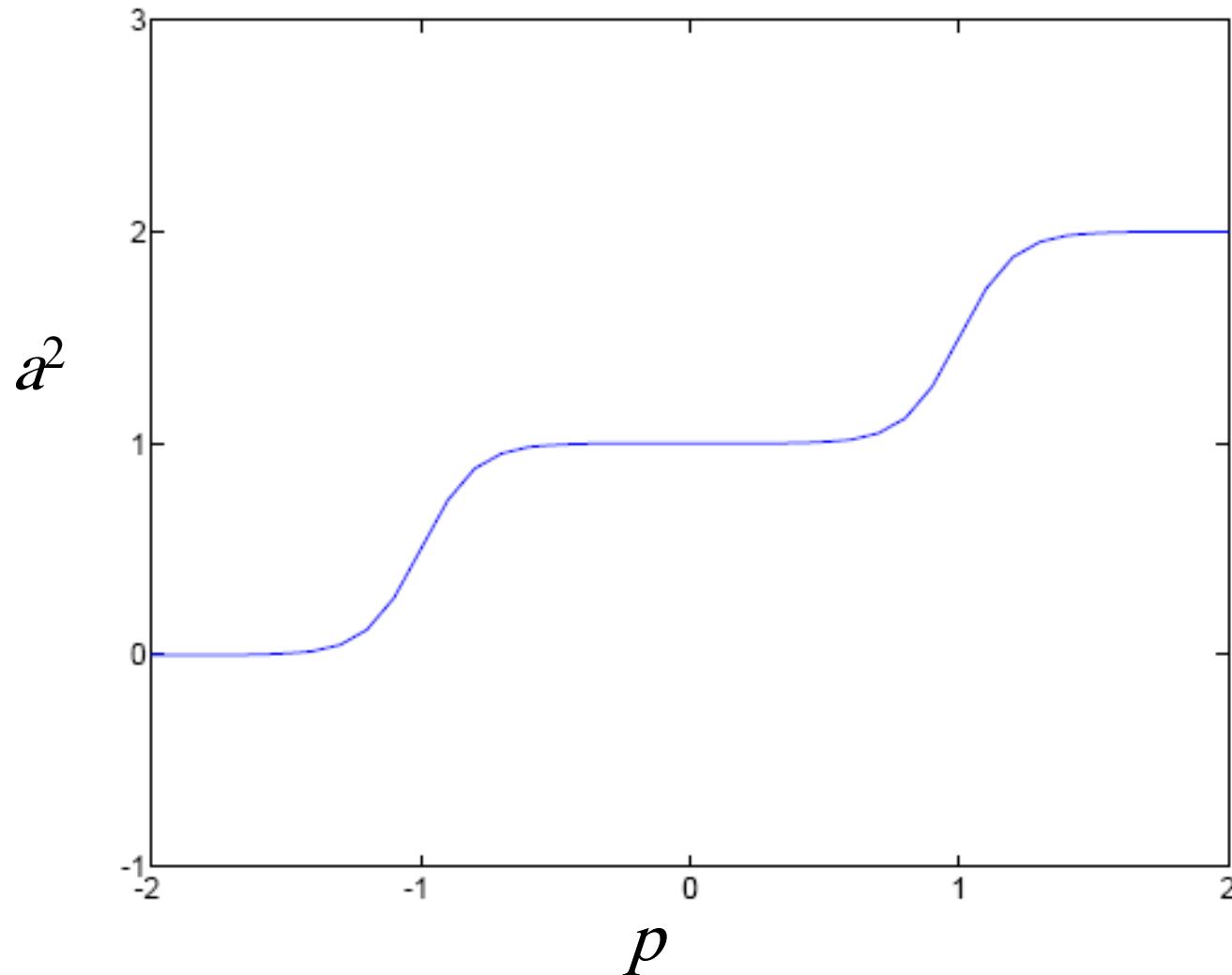
$$f^2(n) = n$$

Suppose that the nominal parameter values are:

$$w_{1,1}^1 = 10 \quad w_{2,1}^1 = 10 \quad b_1^1 = -10 \quad b_2^1 = 10$$

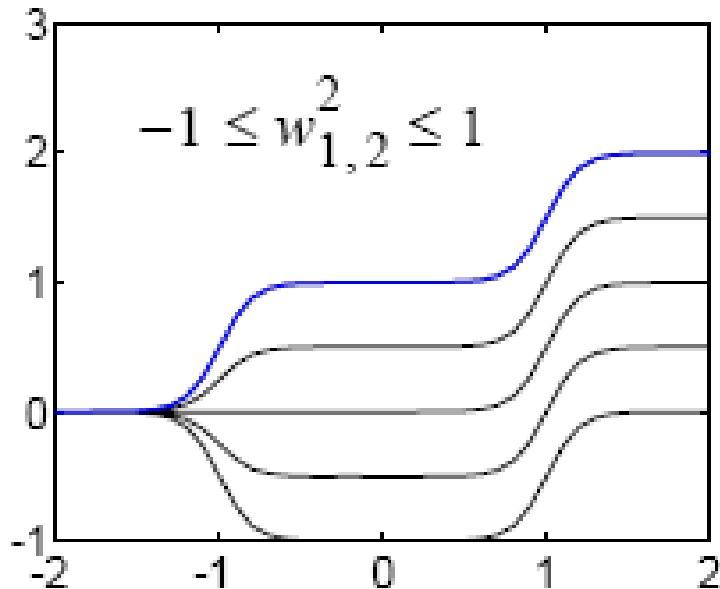
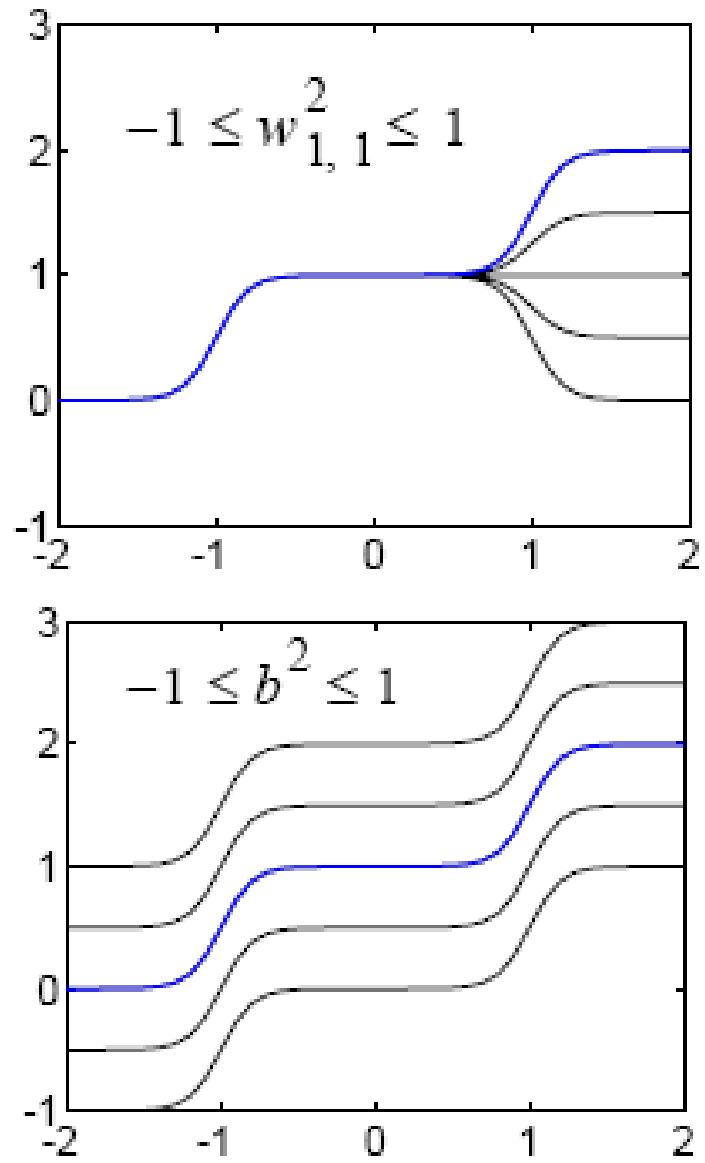
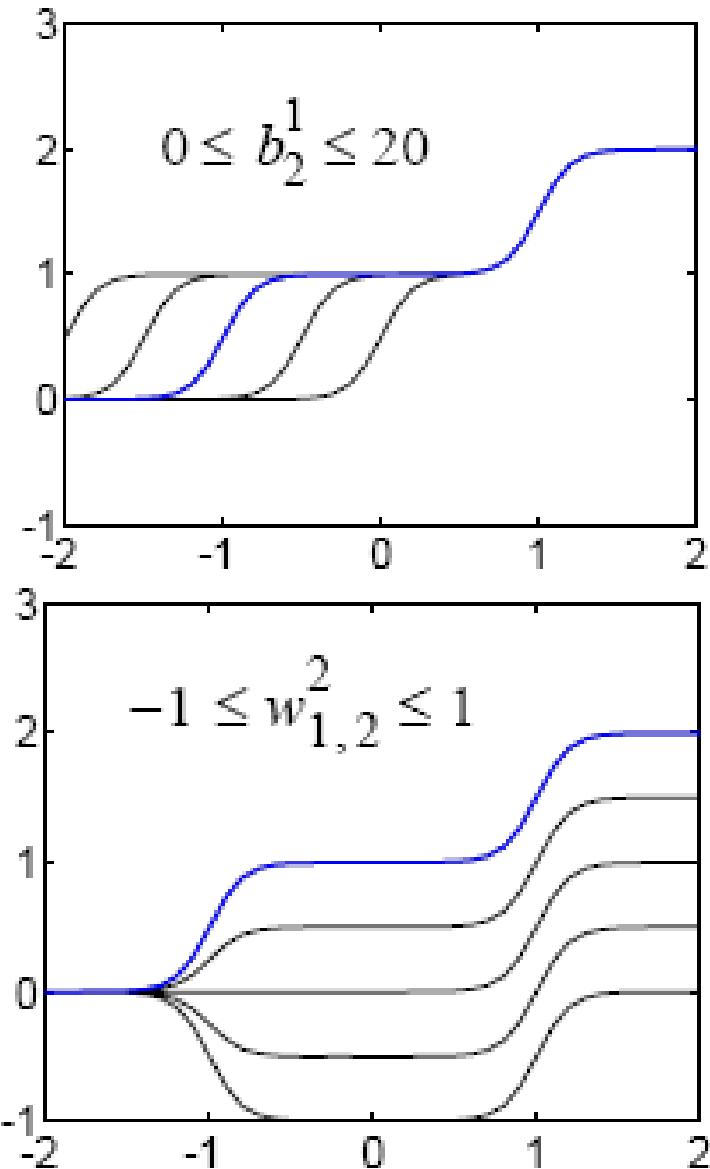
$$w_{1,1}^2 = 1 \quad w_{1,2}^2 = 1 \quad b^2 = 0$$

# Nominal Response

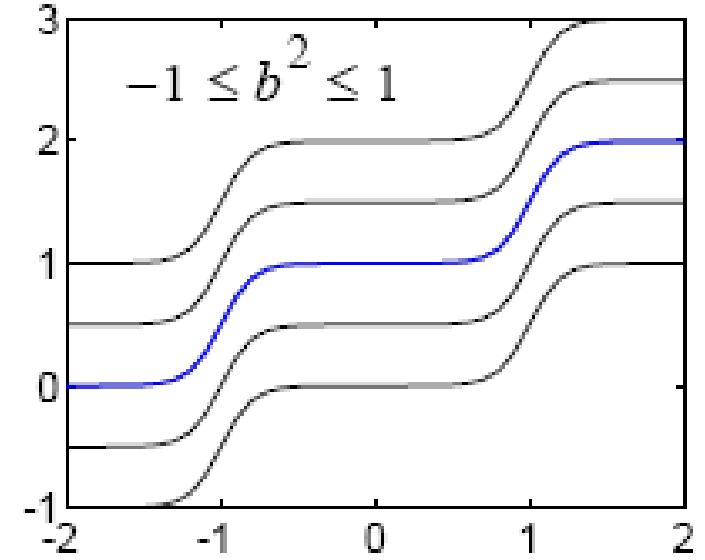


We can change the shape and location of log-sigmoid neurons. 14

# Parameter Variations



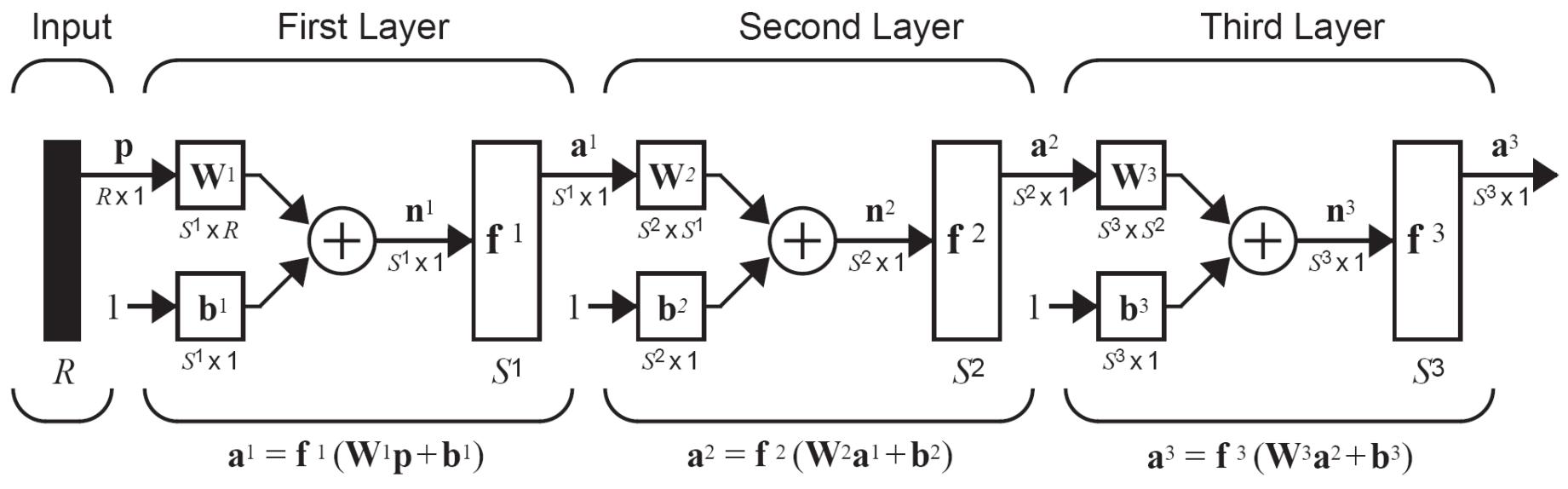
[nnd11nf](#)



# Results

- The multilayer network is very flexible.
- It can be shown that two-layer networks, with sigmoid transfer functions in the hidden layer and linear transfer functions in the output layer, can approximate virtually any function of interest to any degree of accuracy, provided sufficiently many hidden units are available [HoSt89].

# Multilayer Network



$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}) \quad m=0, 1, \dots, M-1$$

$$\mathbf{a}^0 = \mathbf{p} \quad \text{The first layer}$$

$$\mathbf{a} = \mathbf{a}^M \quad \text{The last layer}$$

# Performance Index

Training Set

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \dots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Mean Square Error (the performance index)

$$F(\mathbf{x}) = E[e^2] = E[(t - a)^2]$$

Vector Case

$$F(\mathbf{x}) = E[\mathbf{e}^T \mathbf{e}] = E[(\mathbf{t} - \mathbf{a})^T (\mathbf{t} - \mathbf{a})]$$

Approximate Mean Square Error

(Single Sample; where the expectation of the squared error has been replaced by the squared error at iteration  $k$ .)

$$\hat{F}(\mathbf{x}) = (\mathbf{t}(k) - \mathbf{a}(k))^T (\mathbf{t}(k) - \mathbf{a}(k)) = \mathbf{e}^T(k) \mathbf{e}(k)$$

## Approximate Steepest Descent

$$w_{i,j}^m(k+1) = w_{i,j}^m(k) - \alpha \frac{\partial \hat{F}}{\partial w_{i,j}^m} \quad b_i^m(k+1) = b_i^m(k) - \alpha \frac{\partial \hat{F}}{\partial b_i^m}$$

Where  $\alpha$  is the learning rate.

## Chain Rule

(the computation of the partial derivatives)

$$\frac{df(n(w))}{dw} = \frac{df(n)}{dn} \times \frac{dn(w)}{dw}$$

Example:

$$f(n) = \cos(n) \quad n = e^{2w} \quad f(n(w)) = \cos(e^{2w})$$

$$\frac{df(n(w))}{dw} = \frac{df(n)}{dn} \times \frac{dn(w)}{dw} = (-\sin(n))(2e^{2w}) = (-\sin(e^{2w}))(2e^{2w})$$

## Application to Gradient Calculation

$$\frac{\partial \hat{F}}{\partial w_{i,j}^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^m} \quad \frac{\partial \hat{F}}{\partial b_i^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m}$$

The 2<sup>nd</sup> terms can be easily computed, since the net input to layer  $m$  is an explicit function of the weights and bias in that layer.

# Gradient Calculation

$$S^{m-1}$$
$$n_i^m = \sum_{j=1}^{S^{m-1}} w_{i,j}^m a_j^{m-1} + b_i^m$$
$$\frac{\partial n_i^m}{\partial w_{i,j}^m} = a_j^{m-1} \quad \frac{\partial n_i^m}{\partial b_i^m} = 1$$

If we define *sensitivity* as:

$$s_i^m \equiv \frac{\partial \hat{F}}{\partial n_i^m}$$

(the sensitivity of  $\hat{F}$  to changes is the  $i$ th element of the net input at later  $m$ )

$$\frac{\partial \hat{F}}{\partial w_{i,j}^m} = s_i^m a_j^{m-1} \quad \frac{\partial \hat{F}}{\partial b_i^m} = s_i^m$$

- We can now express the approximate **Steepest Descent** algorithm as:

$$w_{i,j}^m(k+1) = w_{i,j}^m(k) - \alpha s_i^m a_j^{m-1}$$

$$b_i^m(k+1) = b_i^m(k) - \alpha s_i^m$$

In matrix form:

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha \mathbf{s}^m$$

Where

$$\mathbf{s}^m \equiv \frac{\partial \hat{F}}{\partial \mathbf{n}^m} = \begin{bmatrix} \frac{\partial \hat{F}}{\partial n_1^m} \\ \frac{\partial \hat{F}}{\partial n_2^m} \\ \vdots \\ \frac{\partial \hat{F}}{\partial n_{S^m}^m} \end{bmatrix}$$

Compare with LMS algorithm

$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha \mathbf{e}(k) \mathbf{p}^T(k)$$

$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha \mathbf{e}(k)$$

Next Step: Compute the Sensitivities (Backpropagation)

# Jacobian Matrix

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} \equiv \begin{bmatrix} \frac{\partial n_1^{m+1}}{\partial n_1^m} & \frac{\partial n_1^{m+1}}{\partial n_2^m} & \dots & \frac{\partial n_1^{m+1}}{\partial n_{S^m}^m} \\ \vdots & \vdots & & \vdots \\ \frac{\partial n_{S^m+1}^{m+1}}{\partial n_1^m} & \frac{\partial n_{S^m+1}^{m+1}}{\partial n_2^m} & \dots & \frac{\partial n_{S^m+1}^{m+1}}{\partial n_{S^m}^m} \end{bmatrix}$$

$i, j$  element of  
the matrix

$$\frac{\partial n_i^{m+1}}{\partial n_j^m} = \frac{\partial \left( \sum_{l=1}^{S^m} w_{i,l}^{m+1} a_l^m + b_i^{m+1} \right)}{\partial n_j^m} = w_{i,j}^{m+1} \frac{\partial a_j^m}{\partial n_j^m} \quad 24$$

$$\frac{\partial n_i^{m+1}}{\partial n_j^m} = w_{i,j}^{m+1} \frac{\partial f^m(n_j^m)}{\partial n_j^m} = w_{i,j}^{m+1} \dot{f}^m(n_j^m)$$

$$\dot{f}^m(n_j^m) = \frac{\partial f^m(n_j^m)}{\partial n_j^m}$$

Therefore the Jacobian matrix can be written

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} = \mathbf{w}^{m+1} \dot{\mathbf{F}}^m(\mathbf{n}^m)$$

Where

$$\dot{\mathbf{F}}^m(\mathbf{n}^m) = \begin{bmatrix} \bullet^m \\ f(n_1^m) & 0 & \dots & 0 \\ 0 & \bullet^m & f(n_2^m) & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \bullet^m & f(n_{S^m}^m) \end{bmatrix}$$

We can now write out the recurrence relation for the sensitivity by using the chain rule in matrix form.

# Backpropagation (Sensitivities)

$$\mathbf{S}^m = \frac{\partial \hat{\mathbf{F}}}{\partial \mathbf{n}^m} = \left( \frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} \right)^T \frac{\partial \hat{\mathbf{F}}}{\partial \mathbf{n}^{m+1}} = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \frac{\partial \hat{\mathbf{F}}}{\partial \mathbf{n}^{m+1}}$$

$$\mathbf{S}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m) (\mathbf{W}^{m+1})^T \mathbf{s}^{m+1}$$

The sensitivities are computed by starting at the last layer, and then propagating backwards through the network to the first layer.

$$\mathbf{s}^M \rightarrow \mathbf{s}^{M-1} \rightarrow \dots \rightarrow \mathbf{s}^2 \rightarrow \mathbf{s}^1$$

- Note that the backpropagation algorithm uses the same approximate SD technique that we used in the LMS algorithm.
- The only complication is that in order to compute the sensitivities.

## Initialization (Last Layer)

$$s_i^M = \frac{\partial \hat{F}}{\partial n_i^M} = \frac{\partial(\mathbf{t} - \mathbf{a})^T(\mathbf{t} - \mathbf{a})}{\partial n_i^M} = \frac{\partial \sum_{j=1}^{S^M} (t_j - a_j)^2}{\partial n_i^M} = -2(t_i - a_i) \frac{\partial a_i}{\partial n_i^M}$$

Now, since

$$\frac{\partial a_i}{\partial n_i^M} = \frac{\partial a_i^M}{\partial n_i^M} = \frac{\partial f^M(n_i^M)}{\partial n_i^M} = \dot{f}^M(n_i^M)$$

We can write

$$S_i^M = -2(t_i - a_i) \dot{f}^M(n_i^M)$$

$$\mathbf{S}^M = -2 \dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t} - \mathbf{a})$$

# Summary

The first Step: Forward Propagation

$$\mathbf{a}^0 = \mathbf{p}$$

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1} \mathbf{a}^m + \mathbf{b}^{m+1})$$

$$m = 0, 2, \dots, M-1$$

$$\mathbf{a} = \mathbf{a}^M$$

The second step: Backpropagation

$$\mathbf{S}^M = -2 \dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t} - \mathbf{a})$$

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m) (\mathbf{W}^{m+1})^T \mathbf{s}^{m+1}$$

$$m = M-1, \dots, 2, 1$$

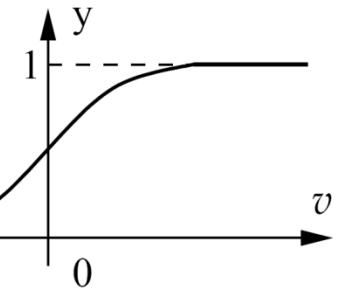
The final step: Weight Update

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha \mathbf{s}^m(\mathbf{a}^{m-1})^T$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha \mathbf{s}^m$$

## Sigmoidal unipolar:

$$y = \varphi(v) = \frac{1}{1 + e^{-\beta v}} = \frac{1}{2}(\tanh(\beta v/2) - 1)$$

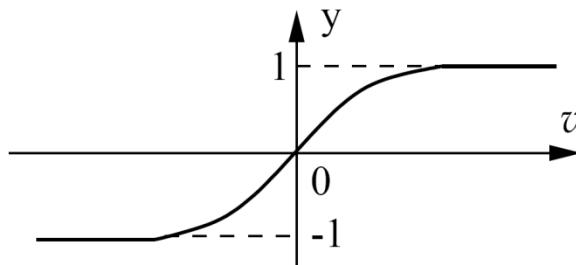


The derivative of the unipolar sigmoidal function:

$$y' = \frac{d\varphi}{dv} = \beta \frac{e^{-\beta v}}{(1 + e^{-\beta v})^2} = \beta y(1 - y)$$

## Sigmoidal bipolar:

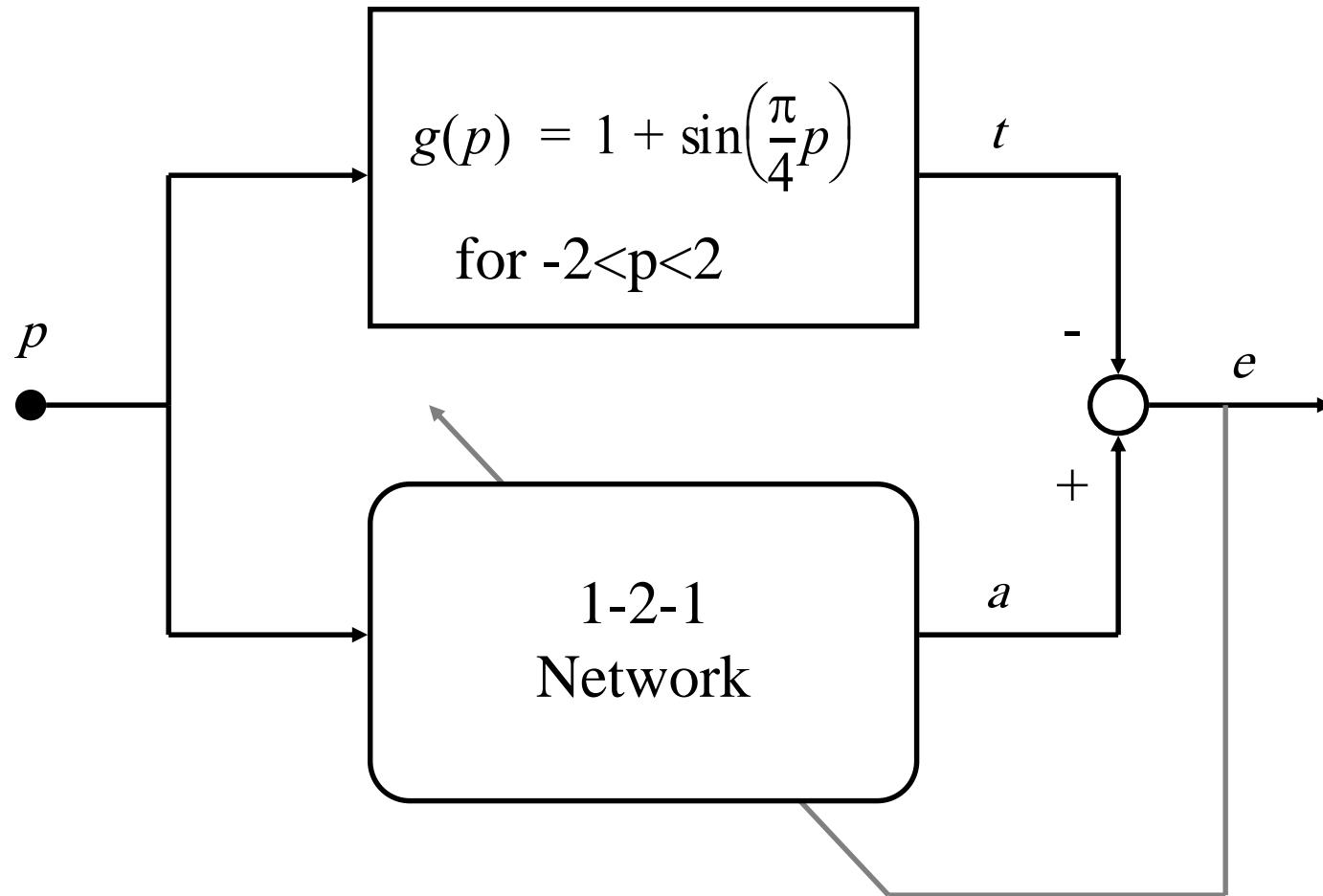
$$\varphi(v) = \tanh(\beta v)$$



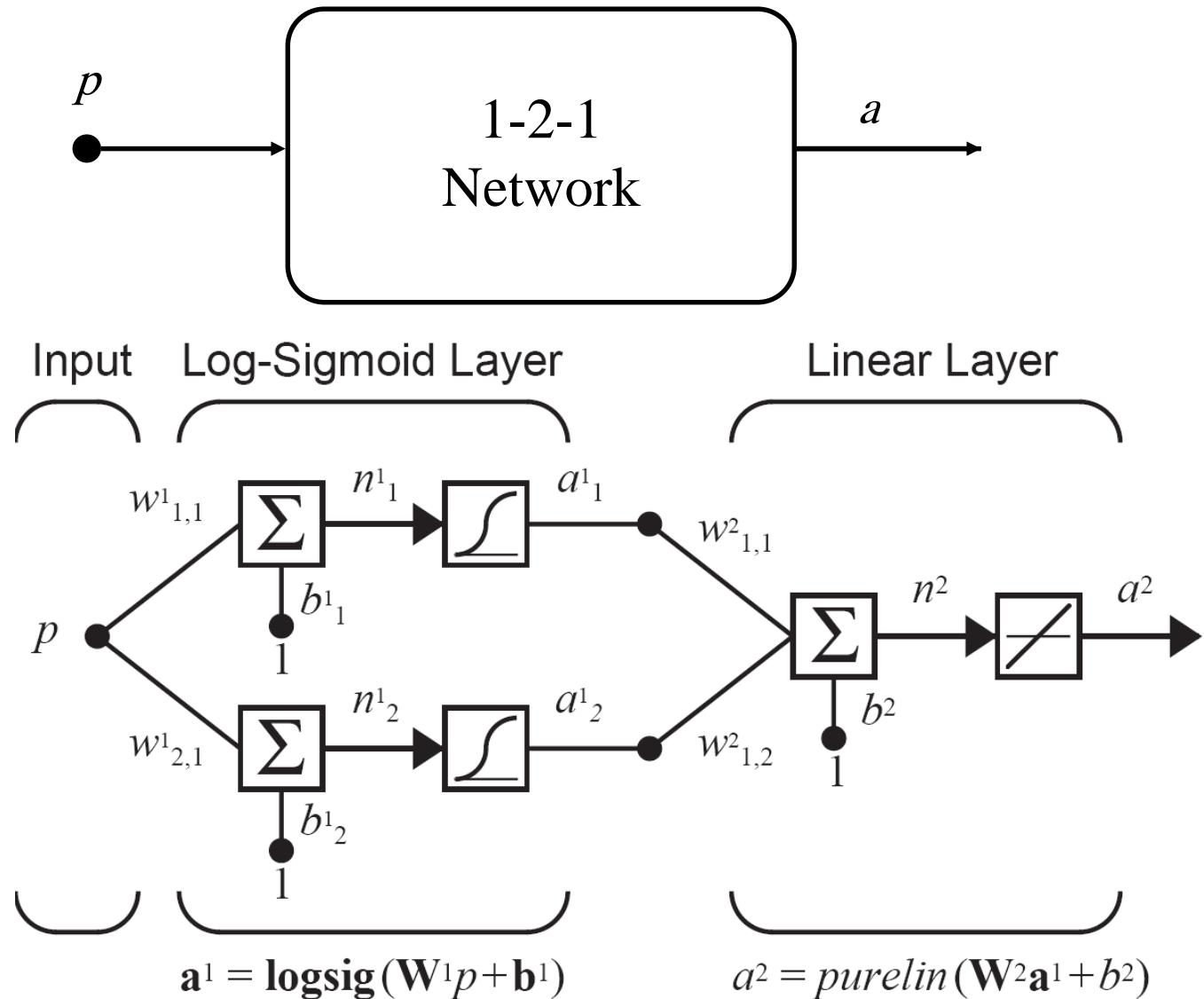
The derivative of the bipolar sigmoidal function:

$$y' = \frac{d\varphi}{dv} = \frac{4\beta e^{2\beta v}}{(e^{2\beta v} + 1)^2} = \beta(1 - y^2)$$

# Example: Function Approximation

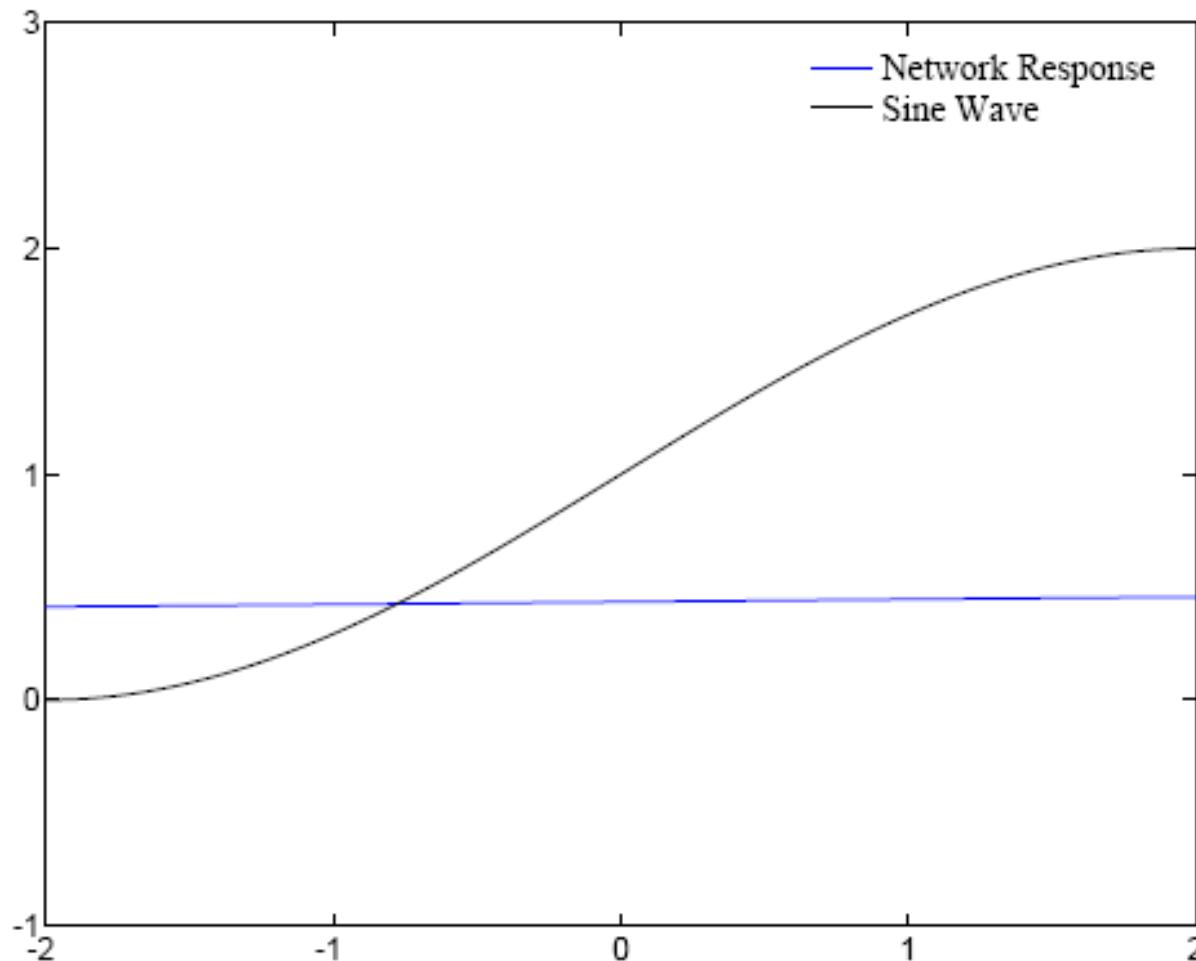


# Network



# Initial Conditions

$$\mathbf{W}^1(0) = \begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix}, \mathbf{b}^1(0) = \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix}, \mathbf{W}^2(0) = \begin{bmatrix} 0.09 & -0.17 \end{bmatrix}, \mathbf{b}^2(0) = \begin{bmatrix} 0.48 \end{bmatrix}$$



# Forward Propagation

To start algorithm we choose  $p=1$ :

$$a^0 = p = 1$$

The output of the first layer is:

$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{W}^1 \mathbf{a}^0 + \mathbf{b}^1) = \text{logsig}\left(\begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix} + \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix}\right) = \text{logsig}\left(\begin{bmatrix} -0.75 \\ -0.54 \end{bmatrix}\right)$$

$$\mathbf{a}^1 = \begin{bmatrix} \frac{1}{1 + e^{-0.75}} \\ \frac{1}{1 + e^{0.54}} \end{bmatrix} = \begin{bmatrix} 0.321 \\ 0.368 \end{bmatrix}$$

The output of the second layer is:

$$a^2 = f^2(\mathbf{W}^2 \mathbf{a}^1 + \mathbf{b}^2) = \text{purelin} ([0.09 \quad -0.17] \begin{bmatrix} 0.321 \\ 0.368 \end{bmatrix} + [0.48]) = [0.446]$$

The error would be:

$$e = t - a = \left\{ 1 + \sin\left(\frac{\pi}{4}p\right) \right\} - a^2 = \left\{ 1 + \sin\left(\frac{\pi}{4}1\right) \right\} - 0.446 = 1.261$$

The next stage of the algorithm is to backpropagate the sensitivities.

# Transfer Function Derivatives

$$\dot{f}^1(n) = \frac{d}{dn} \left( \frac{1}{1+e^{-n}} \right) = \frac{e^{-n}}{(1+e^{-n})^2} = \left( 1 - \frac{1}{1+e^{-n}} \right) \left( \frac{1}{1+e^{-n}} \right) = (1-a^1)(a^1)$$

$$\dot{f}^2(n) = \frac{d}{dn}(n) = 1$$

# Backpropagation

We now start from the 2<sup>nd</sup> layer

$$\mathbf{S}^2 = -2 \dot{\mathbf{F}}^2 (\mathbf{n}^2) (\mathbf{t} - \mathbf{a}) = -2 \left[ \dot{f}^2 (n^2) \right] (1.261)$$

$$= -2[1](1.261) = -2.522$$

The first layer sensitivity is computed by back-propagating the sensitivity from the 2<sup>nd</sup> layer:

$$\mathbf{s}^1 = \dot{\mathbf{F}}^1 (\mathbf{n}^1) (\mathbf{W}^2)^T \mathbf{s}^2 = \begin{bmatrix} (1 - a_1^1)(a_1^1) & 0 \\ 0 & (1 - a_2^1)(a_2^1) \end{bmatrix} \begin{bmatrix} 0.09 \\ -0.17 \end{bmatrix} \begin{bmatrix} -2.522 \end{bmatrix}$$

$$\mathbf{s}^1 = \begin{bmatrix} (1 - 0.321)(0.321) & 0 \\ 0 & (1 - 0.368)(0.368) \end{bmatrix} \begin{bmatrix} 0.09 \\ -0.17 \end{bmatrix} \begin{bmatrix} -2.522 \end{bmatrix}$$

$$\mathbf{s}^1 = \begin{bmatrix} 0.218 & 0 \\ 0 & 0.233 \end{bmatrix} \begin{bmatrix} -0.227 \\ 0.429 \end{bmatrix} = \begin{bmatrix} -0.0495 \\ 0.0997 \end{bmatrix}$$

## Weight Update

This is the final stage. For simplicity we use  $\alpha=0.1$

$$\alpha = 0.1$$

$$\mathbf{W}^2(1) = \mathbf{W}^2(0) - \alpha \mathbf{s}^2(\mathbf{a}^1)^T = \begin{bmatrix} 0.09 & -0.17 \end{bmatrix} - 0.1 \begin{bmatrix} -2.52 \\ 2 \end{bmatrix} \begin{bmatrix} 0.32 & 1 \\ 0.36 & 8 \end{bmatrix}$$

$$\mathbf{W}^2(1) = \begin{bmatrix} 0.171 & -0.0772 \end{bmatrix}$$

$$\mathbf{b}^2(1) = \mathbf{b}^2(0) - \alpha \mathbf{s}^2 = \begin{bmatrix} 0.48 \\ -2.522 \end{bmatrix} - 0.1 \begin{bmatrix} 0.0997 \\ 0.0997 \end{bmatrix} = \begin{bmatrix} 0.732 \\ -0.420 \end{bmatrix}$$

$$\mathbf{W}^1(1) = \mathbf{W}^1(0) - \alpha \mathbf{s}^1 (\mathbf{a}^0)^T = \begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix} - 0.1 \begin{bmatrix} -0.0495 \\ 0.0997 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -0.265 \\ -0.420 \end{bmatrix}$$

$$\mathbf{b}^1(1) = \mathbf{b}^1(0) - \alpha \mathbf{s}^1 = \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix} - 0.1 \begin{bmatrix} -0.0495 \\ 0.0997 \end{bmatrix} = \begin{bmatrix} -0.475 \\ -0.140 \end{bmatrix}$$

- This completes the first iteration of BP algorithm.
- We next proceed to choose another input  $p$  and perform another iteration of algorithm.
- We continue to iterate until the difference between the network response and the target function reaches some acceptable level.

# Batch vs. Incremental Training

- The algorithm described above is the stochastic gradient descent algorithm, which involves “on-line” or *incremental training*, in which the network weights and biases are updated after each input is presented.
- It is also possible to perform *batch training*, in which the complete gradient is computed (after all inputs are applied to the network) before the weights and biases are updated.
- Performance Index:

$$F(\mathbf{x}) = E[\mathbf{e}^T \mathbf{e}] = E[(\mathbf{t} - \mathbf{a})^T (\mathbf{t} - \mathbf{a})] = \frac{1}{Q} \sum_{q=1}^Q (\mathbf{t}_q - \mathbf{a}_q)^T (\mathbf{t}_q - \mathbf{a}_q).$$

The total gradient of this performance index:

$$\nabla F(\mathbf{x}) = \nabla \left\{ \frac{1}{Q} \sum_{q=1}^Q (\mathbf{t}_q - \mathbf{a}_q)^T (\mathbf{t}_q - \mathbf{a}_q) \right\} = \frac{1}{Q} \sum_{q=1}^Q \nabla \{ (\mathbf{t}_q - \mathbf{a}_q)^T (\mathbf{t}_q - \mathbf{a}_q) \}$$

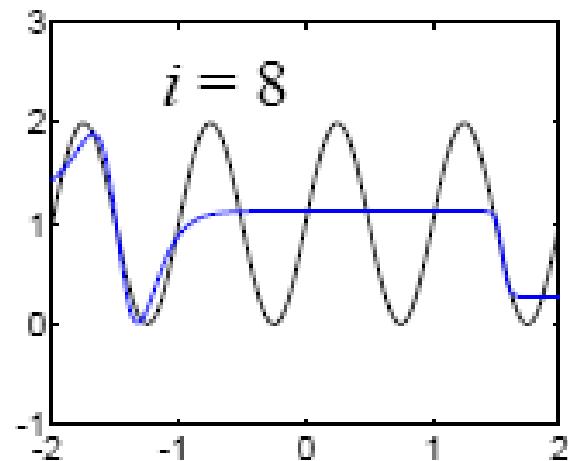
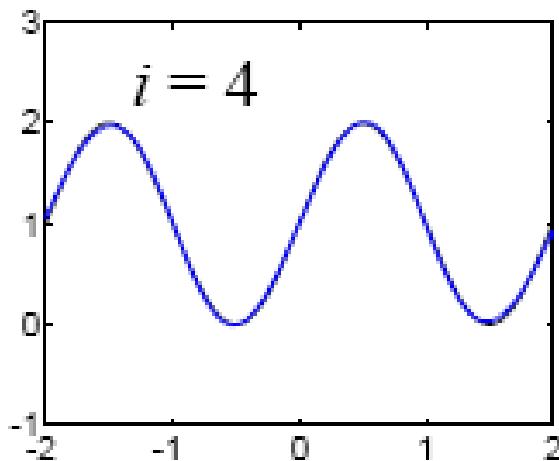
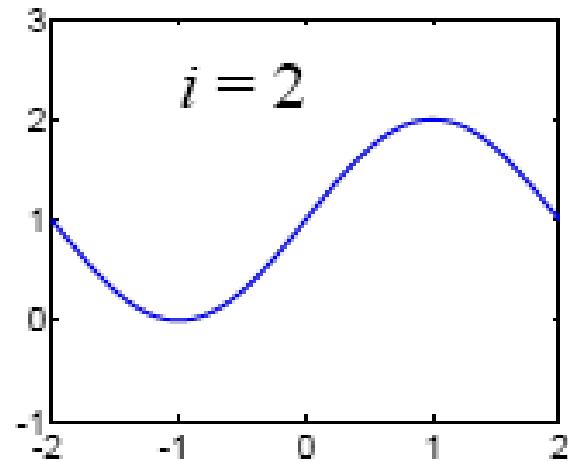
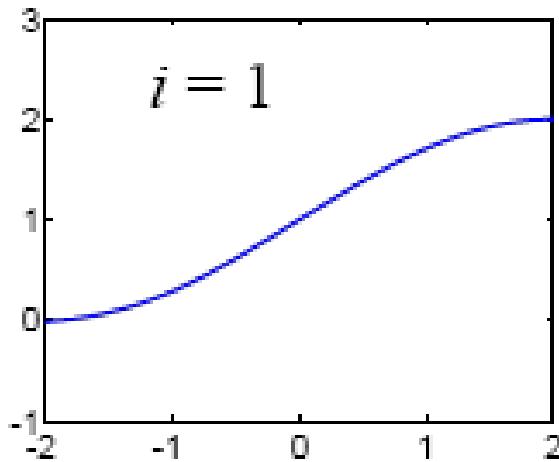
The update equations for the batch steepest descent algorithm are:

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \frac{\alpha}{Q} \sum_{q=1}^Q \mathbf{s}_q^m (\mathbf{a}_q^{m-1})^T,$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \frac{\alpha}{Q} \sum_{q=1}^Q \mathbf{s}_q^m.$$

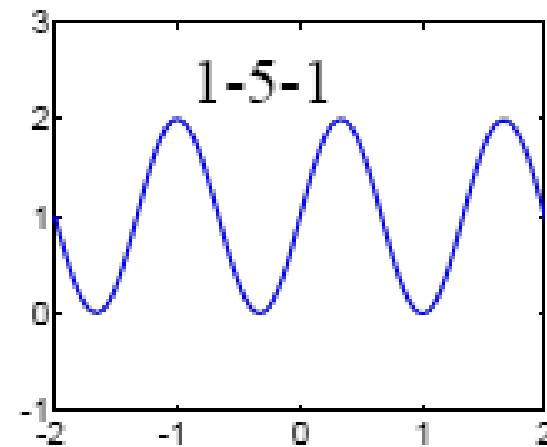
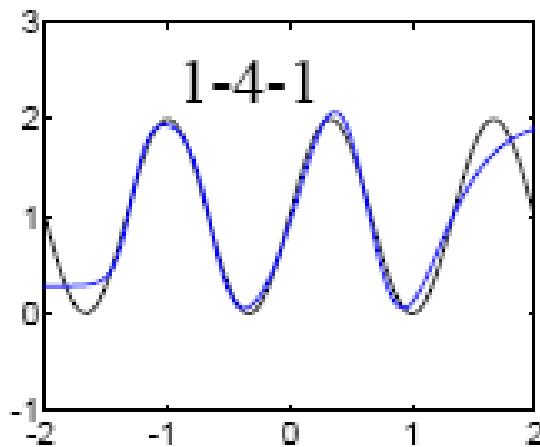
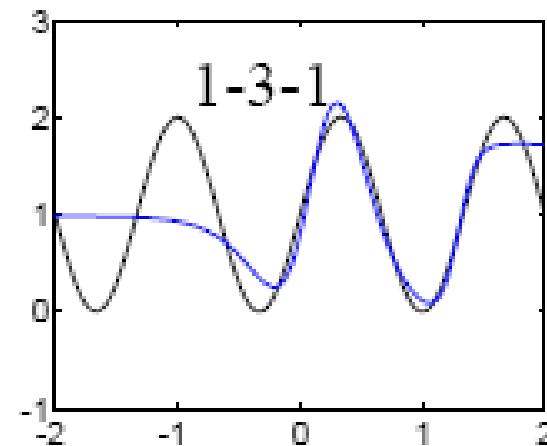
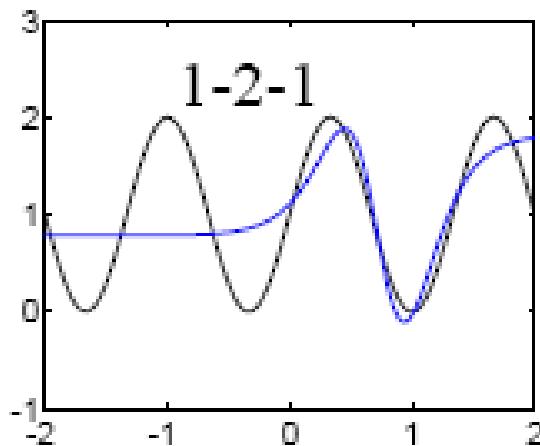
# Choice of Architecture

$$g(p) = 1 + \sin\left(\frac{i\pi}{4}p\right) \quad \text{for } -2 < p < 2 \quad \text{1-3-1 Network}$$



# Choice of Network Architecture

$$g(p) = 1 + \sin\left(\frac{6\pi}{4}p\right) \quad \text{for } -2 < p < 2$$

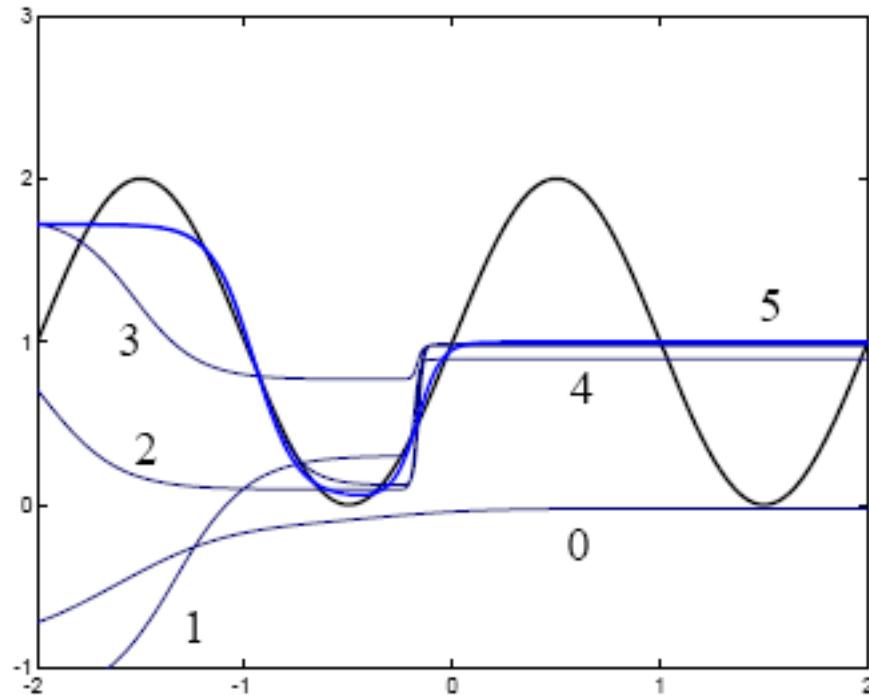
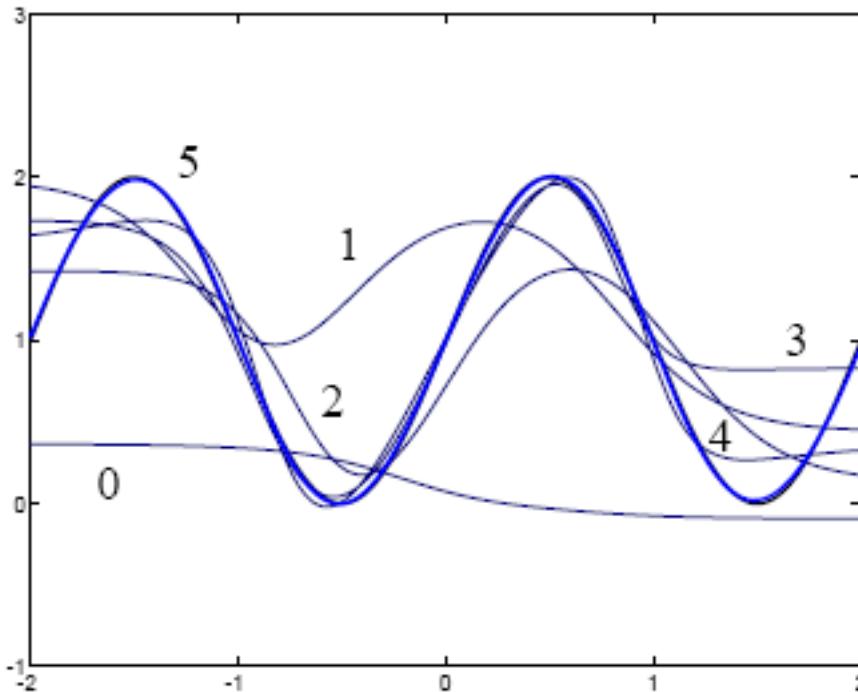


- **Summary:** A 1-S<sup>1</sup>-1 network, with sigmoid neurons in the hidden layer and linear neurons in the output layer, can produce a response that is a superposition of S<sup>1</sup> sigmoid functions.
- To approximate a function that has a large number of reflection points, a large number of neurons in the hidden layer is needed. [nnd11fa](#)

# Different initial conditions

$$g(p) = 1 + \sin(\pi p)$$

1-3-1



The numbers indicate sequence of iterations. There are many iterations between those shown.

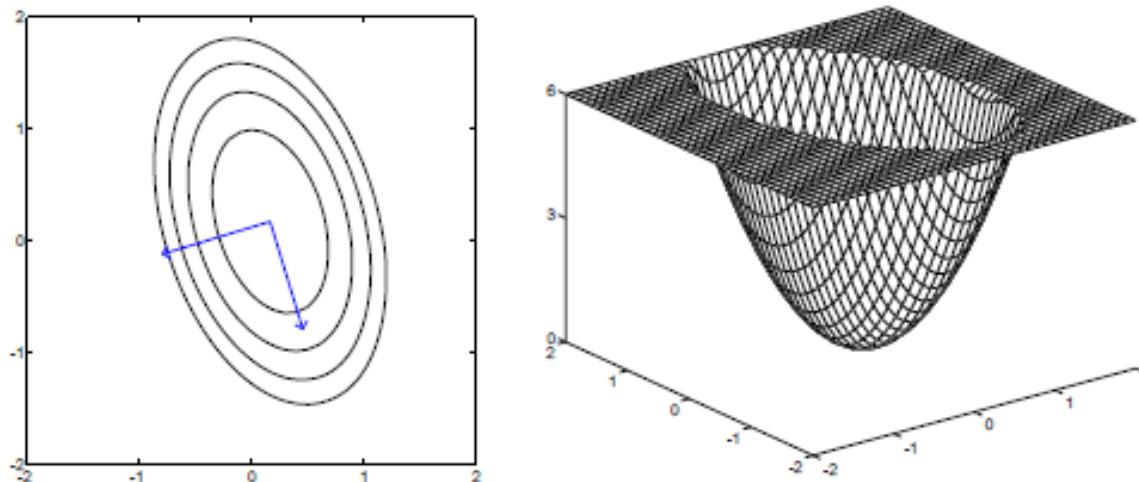
*Left:* Solution minimizes MSE

*Right:* Solution does not minimize MSE.

# Convergence

- Sometimes the network is capable of approximating the function, but the learning algorithm does not produce network parameters that produce an accurate approximation.
- It is best to try several different initial conditions in order to ensure that an optimum solution has been obtained.

- Note that this result could not have occurred with the LMS algorithm. The MSE performance index for the ADALINE network is a quadratic function with a single minimum point .
- The MSE for multilayer network is generally much more complex and has many local minima.

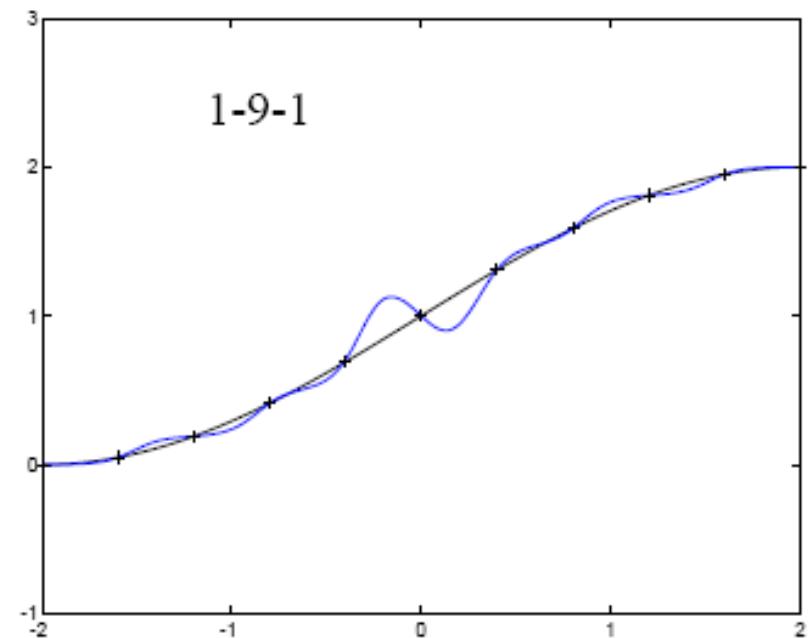
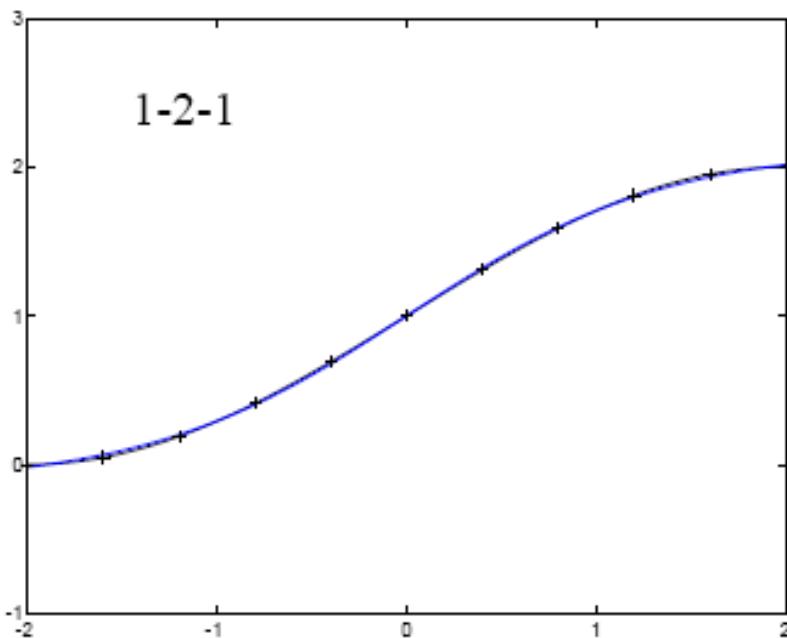


# Generalization

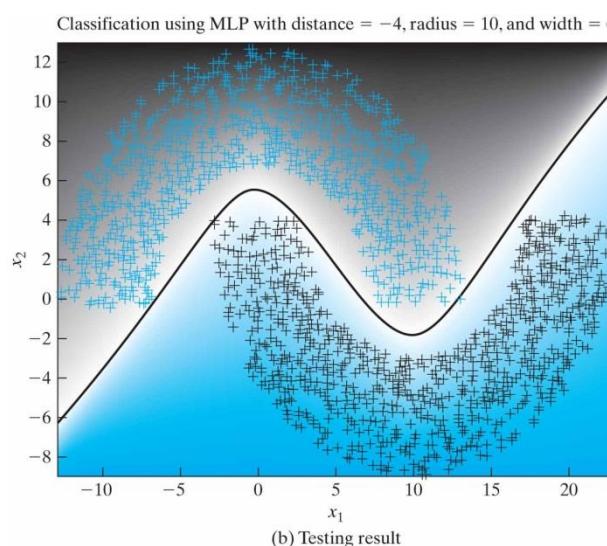
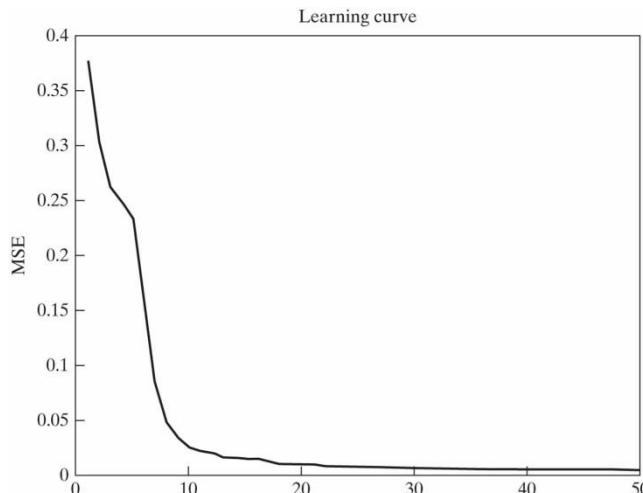
$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \dots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

It is important that the network successfully generalize what it has learned (by finite number of examples) to total population.

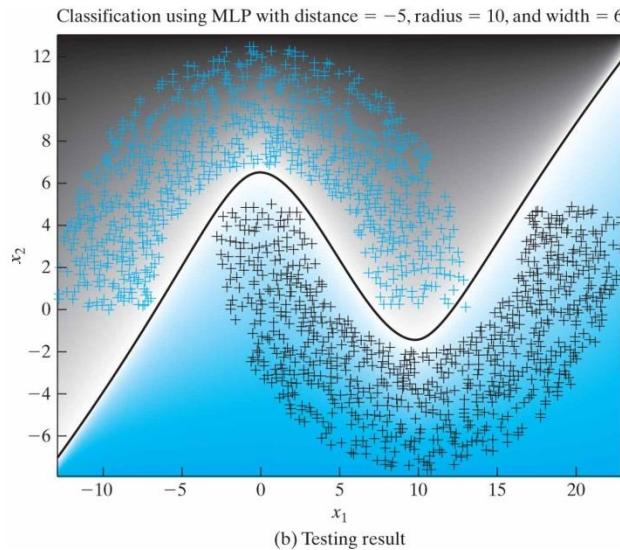
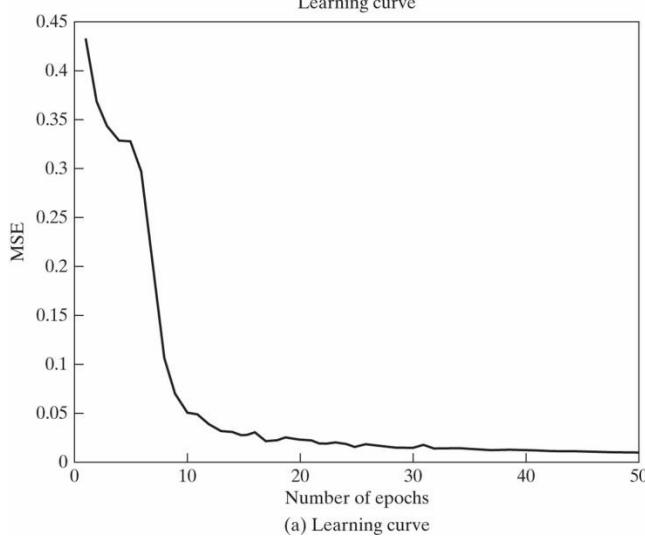
$$g(p) = 1 + \sin\left(\frac{\pi}{4}p\right) \quad p = -2, -1.6, -1.2, \dots, 1.6, 2$$



**Figure 4.12** Results of the computer experiment on the back-propagation algorithm applied to the MLP with distance  $d = -4$ .



**Figure 4.13** Results of the computer experiment on the back-propagation algorithm applied to the MLP with distance  $d = -5$ .



# Strengths of BP Nets

- **Great representation power**
  - Any function can be represented by a BP net (multi-layer feed-forward net with non-linear hidden units)
- **Wide applicability of BP learning**
  - Only requires that a good set of training samples is available).
  - Tolerates noise and missing data in training samples
- **Easy to implement**
- **Good generalization power**

# Deficiencies of BP Nets

- Learning often takes a **long time** to converge
  - Complex functions often need hundreds or thousands of epochs
- The net is essentially a **black box**
  - It may provide a desired mapping between input and output vectors ( $p$ ,  $a$ ) but does not have the information of why a particular  $p$  is mapped to a particular  $a$ .
- Gradient descent approach only guarantees to reduce the total error to a **local minimum**. ( $F$  may be reduced to zero)
  - Cannot escape from the local minimum error state.

- **Generalization** is not guaranteed even if the error is reduced to zero
  - Over-fitting/over-training problem: trained net fits the training samples perfectly ( $F$  reduced to 0) but it does not give accurate outputs for inputs not in the training set.
- **Network paralysis** with sigmoid activation function (فلج شدن شبکه)
  - Saturation regions because of the sigmoid Transfer functions.

$$f(x) = 1 / (1 + e^{-sx}),$$

$$f'(x) = s f(x)(1 - f(x))$$

# Some Practical Considerations

- **Training samples:**
  - Quality and quantity of training samples determines the quality of learning results
  - Samples must be good representatives of the problem space
    - Random sampling
    - Proportional sampling (with prior knowledge of the problem space)
  - Binary vs. bipolar
    - Bipolar representation uses training samples more efficiently; no learning will occur when input is zero with binary rep.

# Variations on Backpropagation

- The basic backpropagation algorithm is too slow for most practical applications. It may take days or weeks of computer time.
- We demonstrate why the backpropagation algorithm is slow in converging.
- We saw the steepest descent is the slowest minimization method.
- The conjugate gradient algorithm and Newton's method generally provide faster convergence.

# Variations

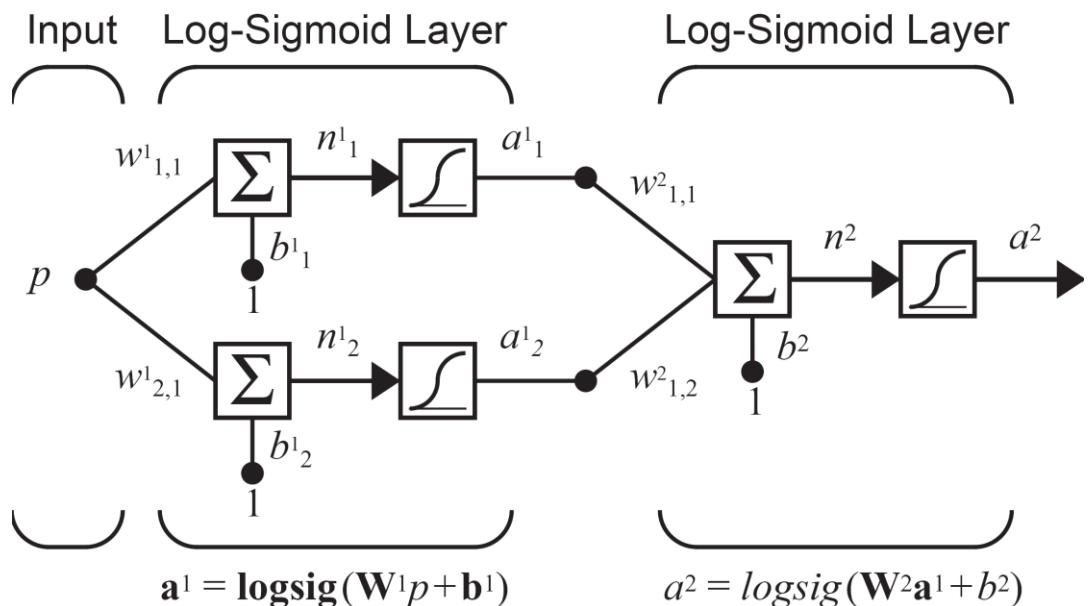
- Heuristic modifications
  - Momentum
  - Variable learning rate
- Standard numerical optimization
  - Conjugate gradient
  - Newton's method (Levenberg-Marquardt)

# Drawbacks of BP

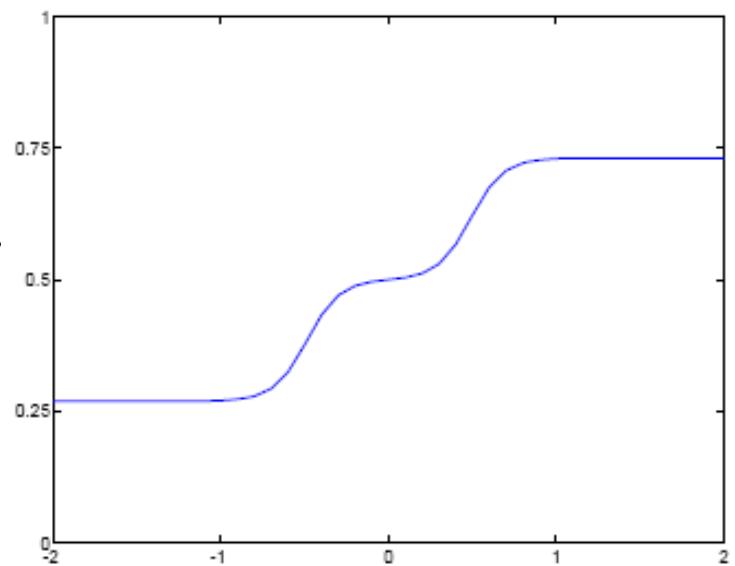
- We saw that the LMS algorithm is guaranteed to converge to a solution that minimizes the mean squared error, so long as the learning rate is not too large.
- Steepest Descent backpropagation (SDBP) is a generalization of the LMS algorithm.
- Multilayer nonlinear Net → many local minimum points → the curvature can vary widely in different regions of the parameter space.

# Performance Surface Example

Network Architecture



Nominal Function

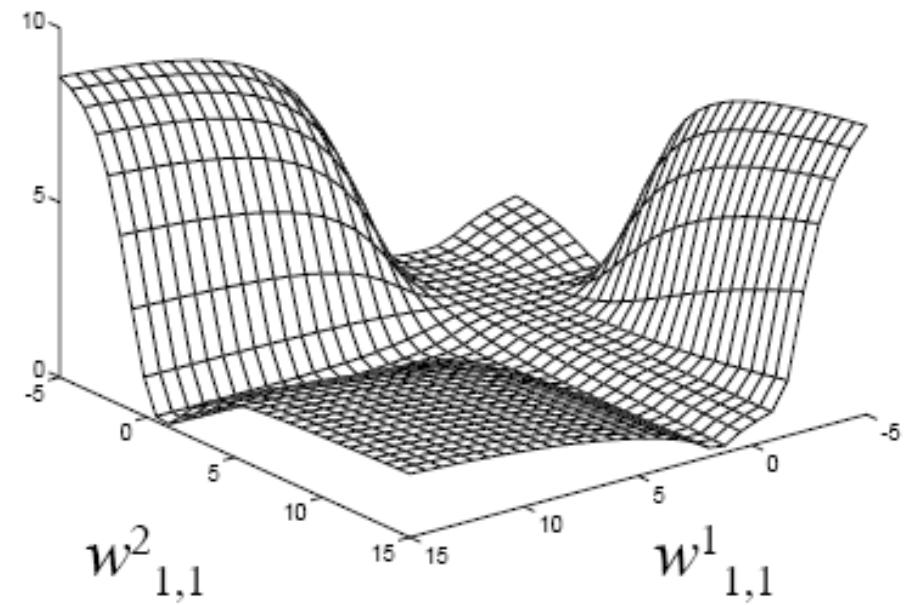
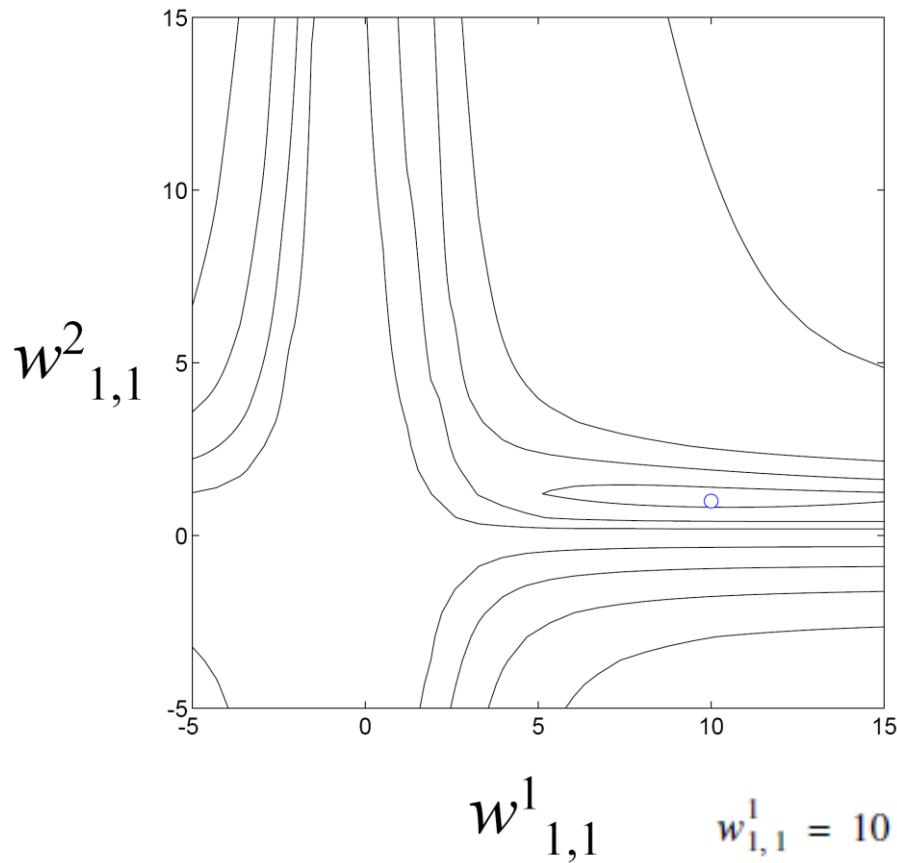


Parameter Values

$$w_{1,1}^1 = 10 \quad w_{2,1}^1 = 10 \quad b_1^1 = -5 \quad b_2^1 = 5$$

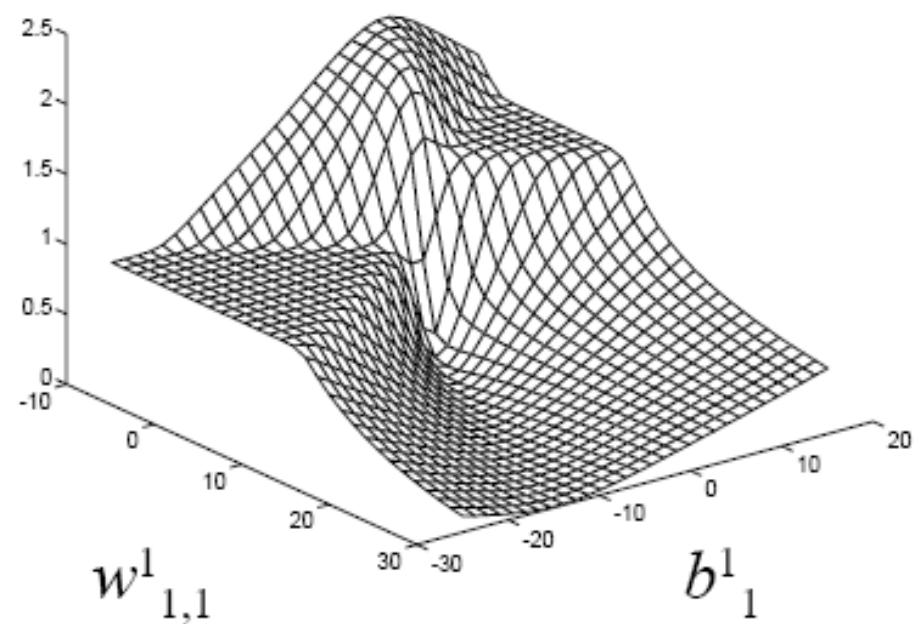
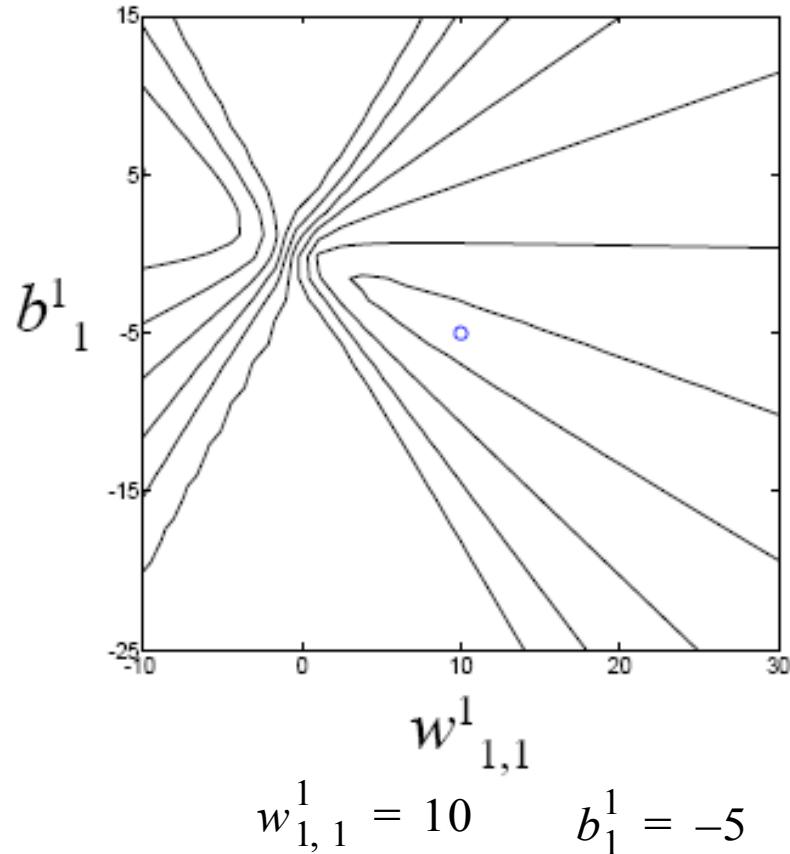
$$w_{1,1}^2 = 1 \quad w_{1,2}^2 = 1 \quad b^2 = -1$$

# Squared Error vs. $w_{1,1}^1$ and $w_{1,1}^2$



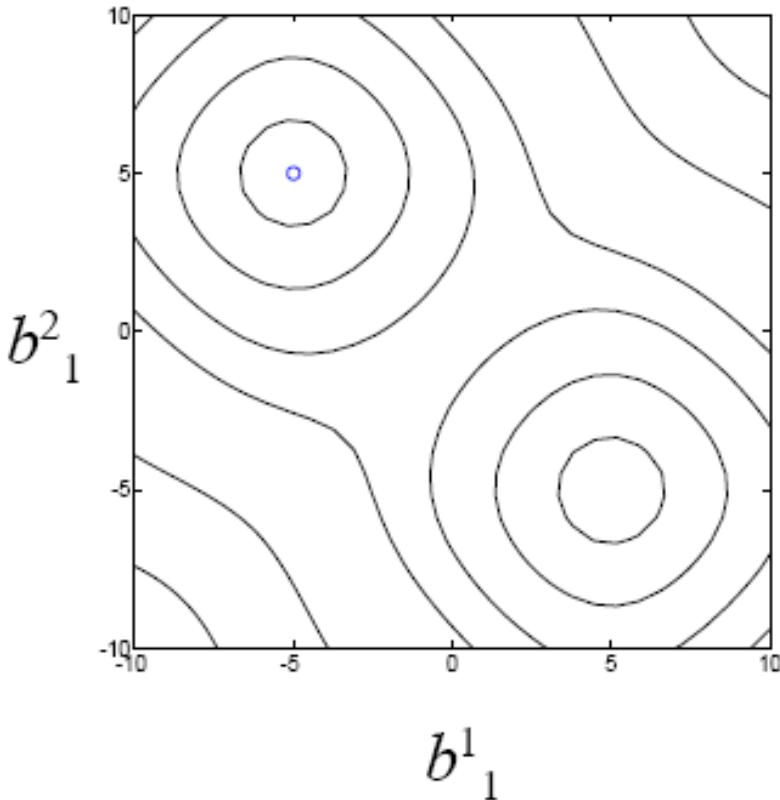
The curvature varies drastically over the parameter space. So it is difficult to choose an appropriate learning rate for SD algorithm.

# Squared Error vs. $w_{1,1}^1$ and $b_1^1$



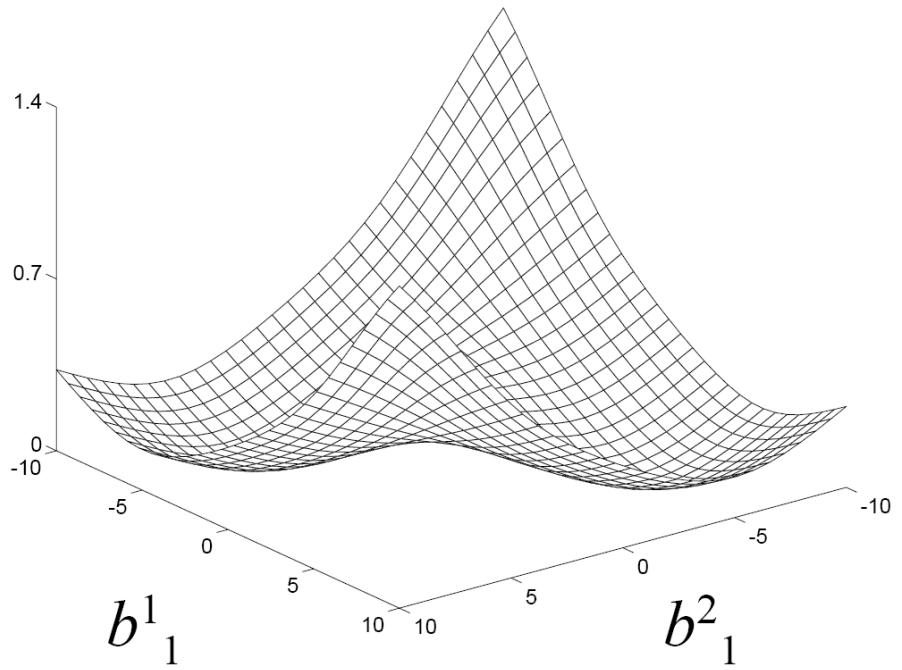
flat regions of the performance surface should not be unexpected, given the sigmoid transfer functions used by the networks. The sigmoid is very flat for large inputs.

# Squared Error vs. $b_1^1$ and $b_2^1$



$$b_1^1 = -5$$

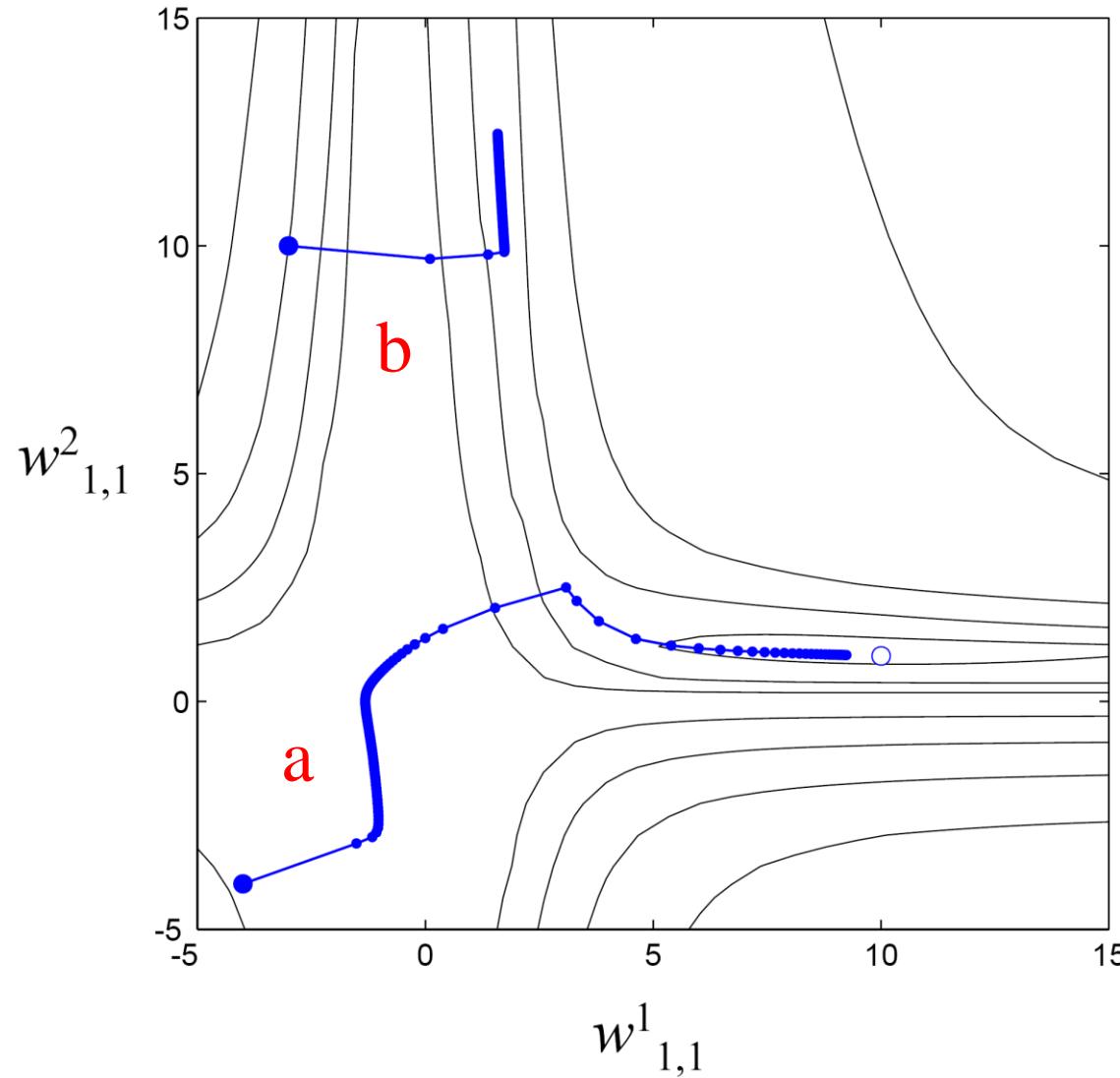
$$b_2^1 = 5$$



It is because of this characteristic of neural networks that we do not set the initial weights and biases to zero.

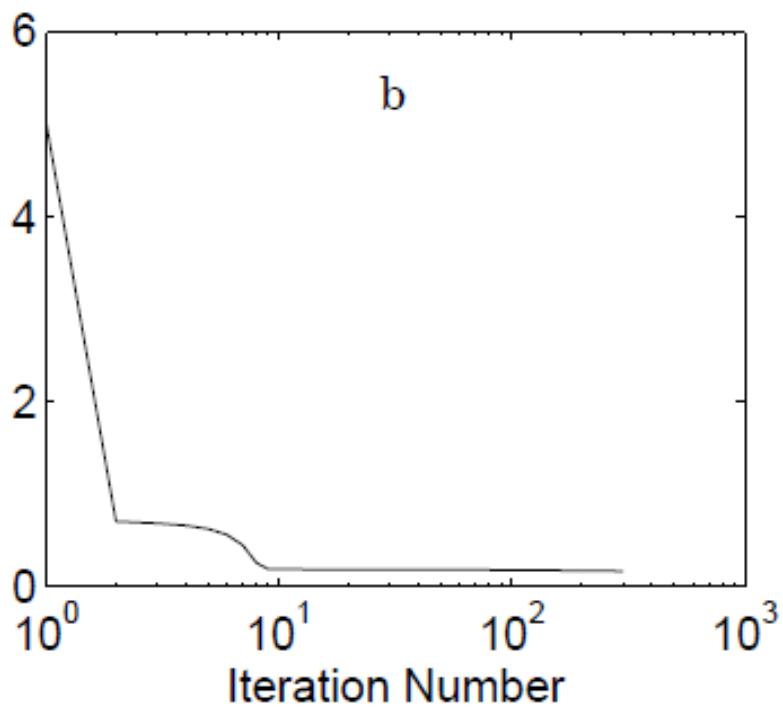
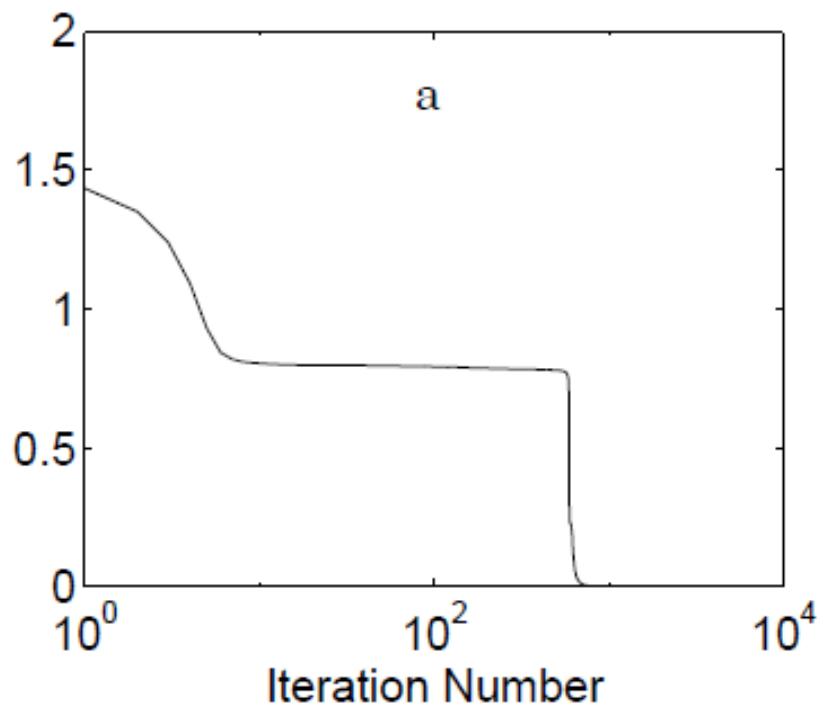
# Convergence Example

- We use a variation of the standard algorithm, called **batching**.
- In batching mode the parameters are updated only after the entire training set has been presented.
- The gradients calculated at each training example are averaged together to produce a more accurate estimate of the gradient.
  - Smoothing the training sample outliers
  - Learning independent of the order of sample presentations
  - Usually slower than in sequential mode

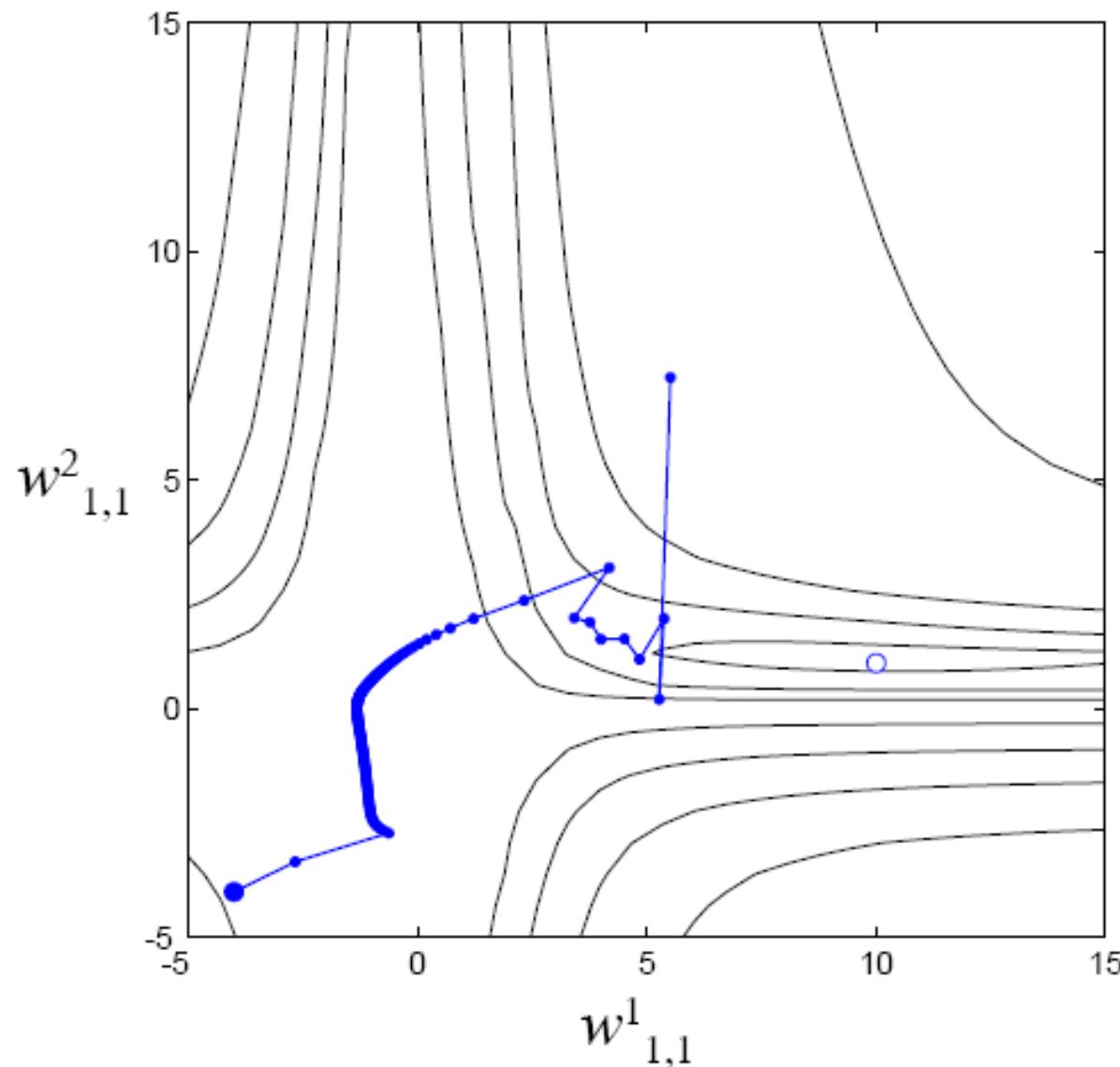


a: converge to the optimal solution, but the convergence is slow.

b: converge to a local minimum ( $w_{1,1}^1 = 0.88$ ,  $w_{1,1}^2 = 38.6$ ). 10



# Learning Rate Too Large



nnd12sd1

nnd12sd2

# Momentum

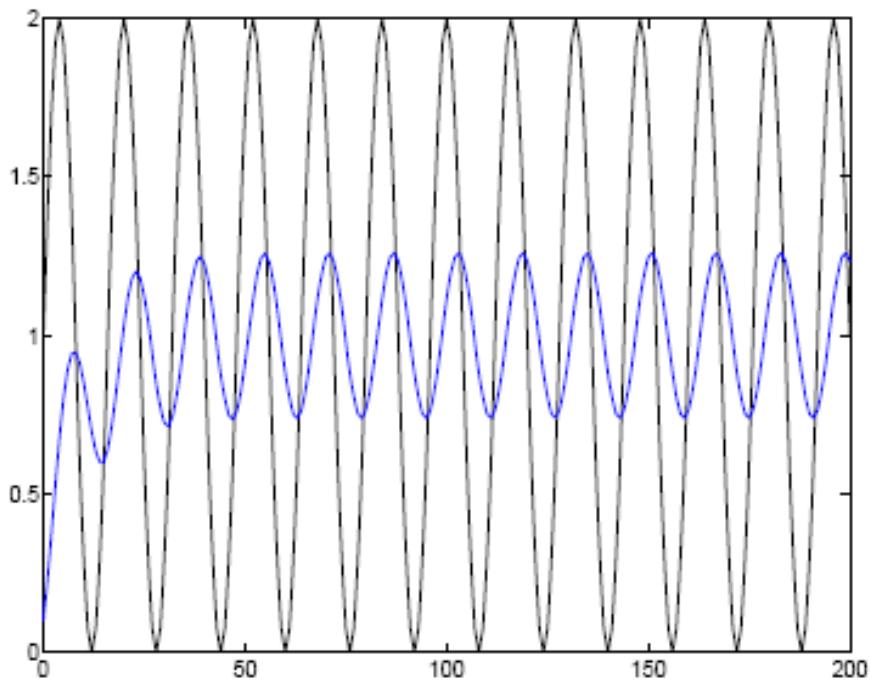
## Filter

$$y(k) = \gamma y(k-1) + (1-\gamma)w(k) \quad 0 \leq \gamma < 1$$

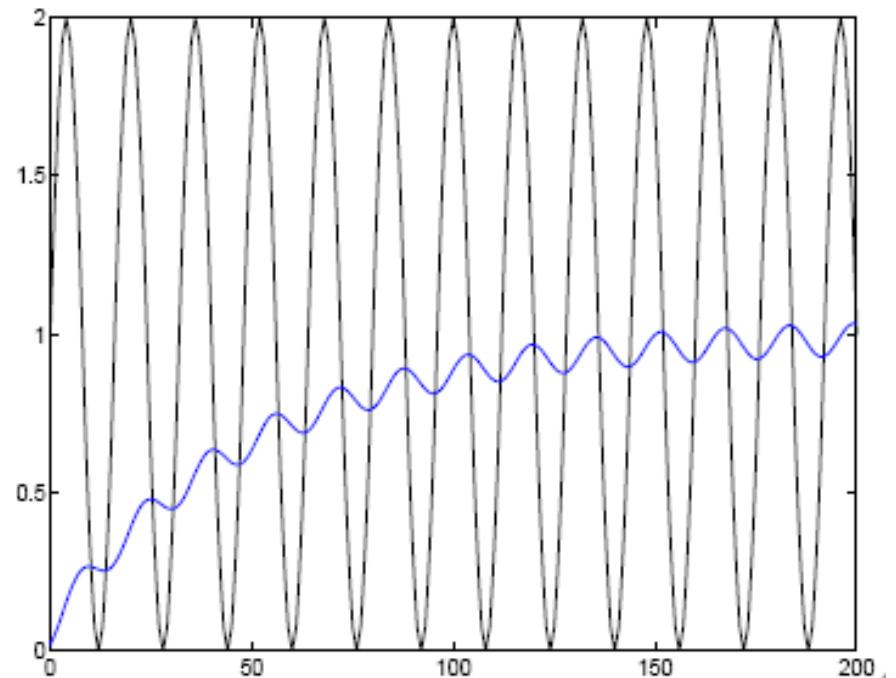
## Example

$$w(k) = 1 + \sin\left(\frac{2\pi k}{16}\right)$$

$\gamma = 0.9$



$\gamma = 0.98$



# Observations

- The oscillation of the filter output is less than the oscillation in the filter input (low pass filter).
- As  $\gamma$  is increased the oscillation in the filter output is reduced.
- The average filter output is the same as average filter input, although as  $\gamma$  is increased the filter output is slower to respond.
- To summarize, the filter tends to reduce the amount of oscillation, while still tracking the average value.

# Momentum Backpropagation

Steepest Descent Backpropagation  
(SDBP)

$$\Delta \mathbf{W}^m(k) = -\alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T$$

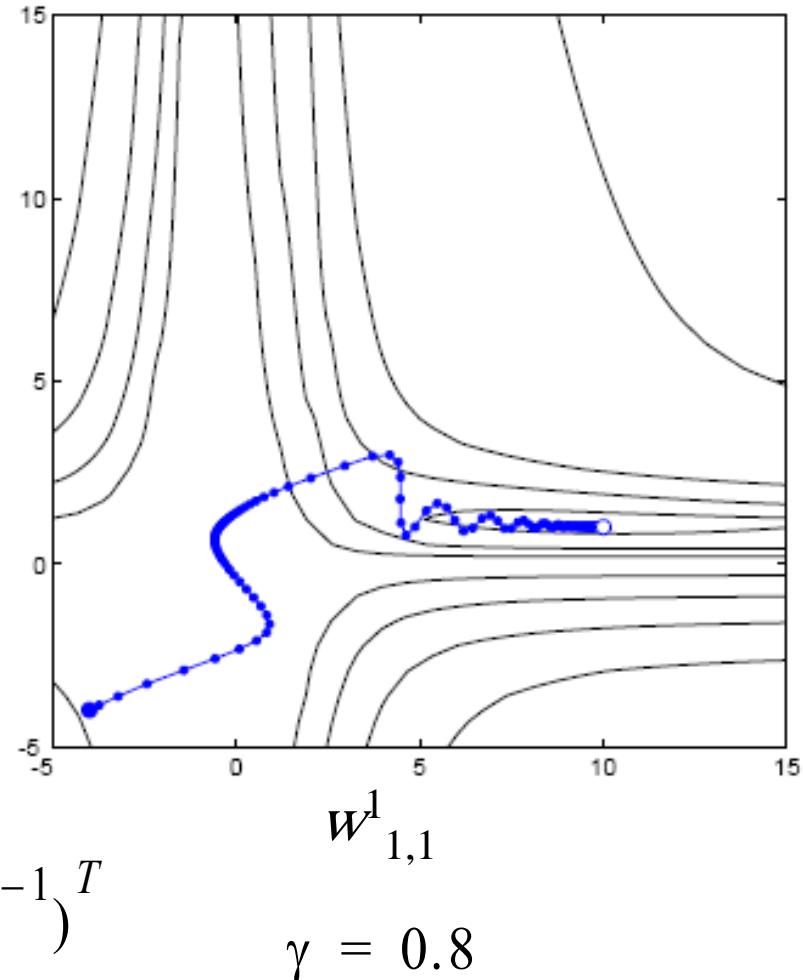
$$\Delta \mathbf{b}^m(k) = -\alpha \mathbf{s}^m$$

Momentum Backpropagation  
(MOBP)

$$\Delta \mathbf{W}^m(k) = \gamma \Delta \mathbf{W}^m(k-1) - (1-\gamma) \alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T$$

$$\gamma = 0.8$$

$$\Delta \mathbf{b}^m(k) = \gamma \Delta \mathbf{b}^m(k-1) - (1-\gamma) \alpha \mathbf{s}^m$$



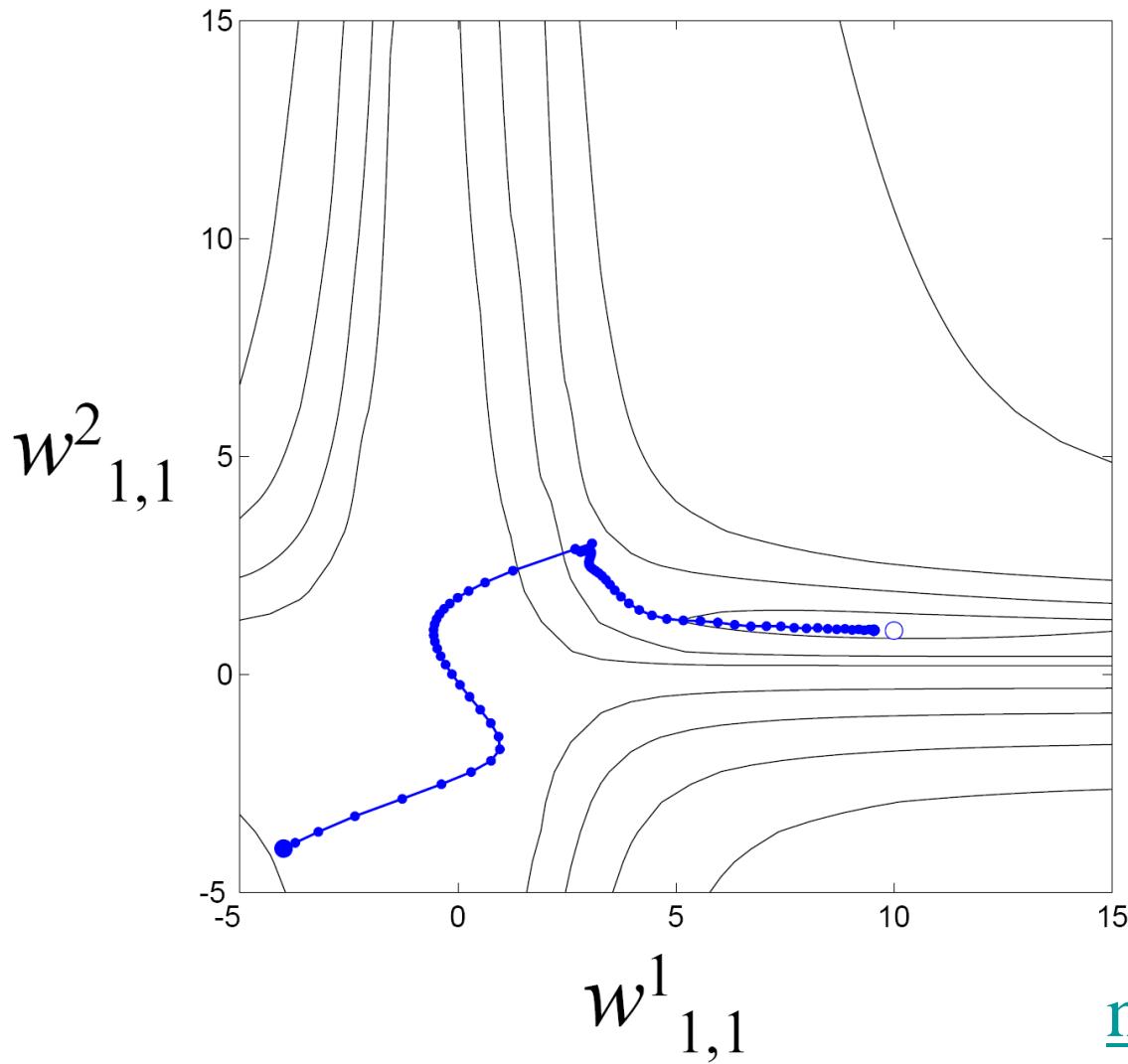
- The batching form of MOBP, in which the parameters are updated only after the entire example set has been presented.
- The same initial condition and learning rate has been used as in the previous example, in which the algorithm was not stable.
- The algorithm now is stable and it tends to accelerate convergence when the trajectory is moving in a consistent direction.

nnd12mo

# Variable Learning Rate (VLBP)

- If the squared error (over the entire training set) increases by more than some set percentage  $\zeta$  after a weight update, then the weight update is discarded, the learning rate is multiplied by some factor ( $0 < \rho < 1$ ), and the momentum coefficient  $\gamma$  is set to zero.
- If the squared error decreases after a weight update, then the weight update is accepted and the learning rate is multiplied by some factor  $\eta > 1$ . If  $\gamma$  has been previously set to zero, it is reset to its original value.
- If the squared error increases by less than  $\zeta$ , then the weight update is accepted, but the learning rate and the momentum coefficient are unchanged.

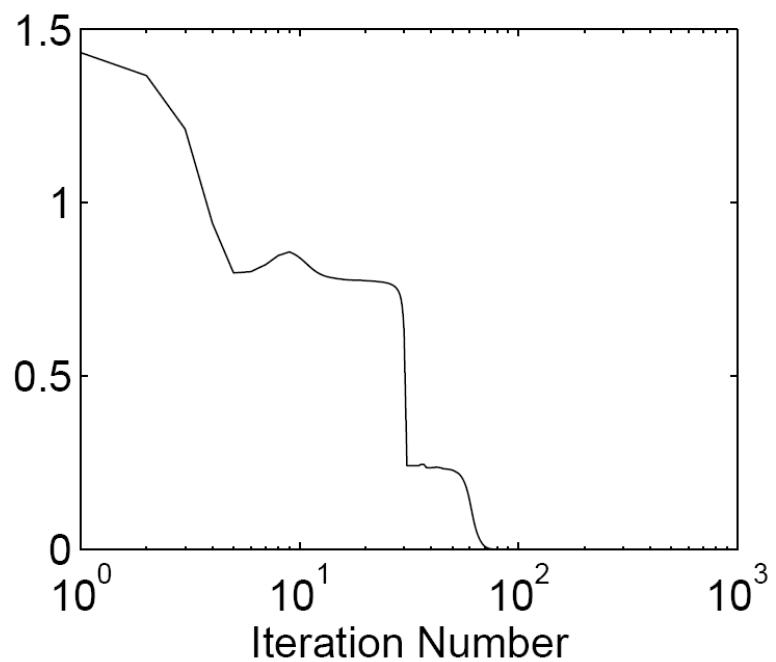
# Example



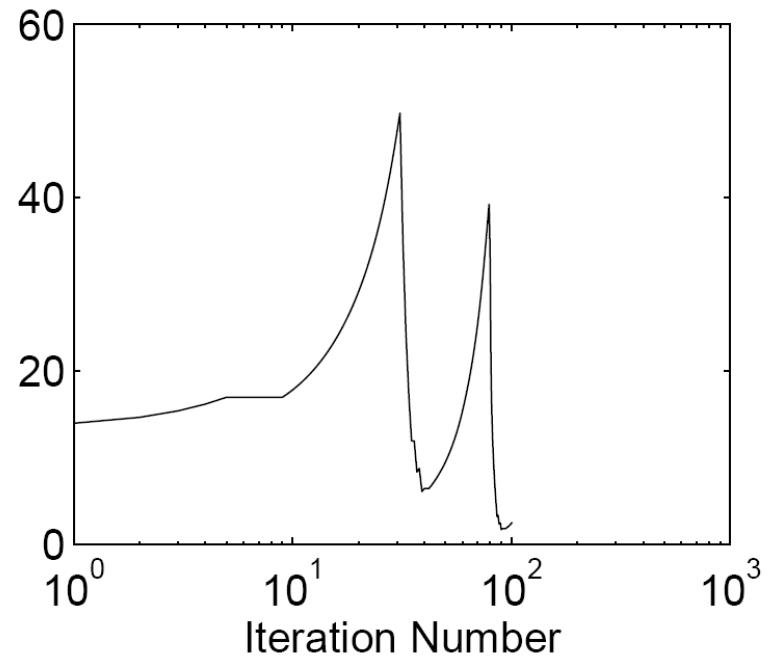
$$\begin{aligned}\eta &= 1.05 \\ \rho &= 0.7 \\ \zeta &= 4\%\end{aligned}$$

nnd12vl

**Squared Error**



**$\alpha$  Learning Rate**



Convergence Characteristics Of Variable Learning Rate

# Conjugate Gradient

- We saw SD is the simplest optimization method but is often slow in converging.
- Newton's method is much faster, but requires that the Hessian matrix and its inverse be calculated.
- The conjugate gradient is a compromise; it does not require the calculation of 2<sup>nd</sup> derivatives, and yet it still has the quadratic convergence property.
- Now we describe how the conjugate gradient algorithm can be used to train multilayer network.
- This algorithm is called Conjugate Gradient Backpropagation (CGBP).

# Review Of CG Algorithm

1. The first search direction is steepest descent.

$$\mathbf{p}_0 = -\mathbf{g}_0 \quad \mathbf{g}_k \equiv \nabla F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{X}_k}$$

2. Take a step and choose the learning rate to minimize the function along the search direction.

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$

3. Select the next search direction according to:

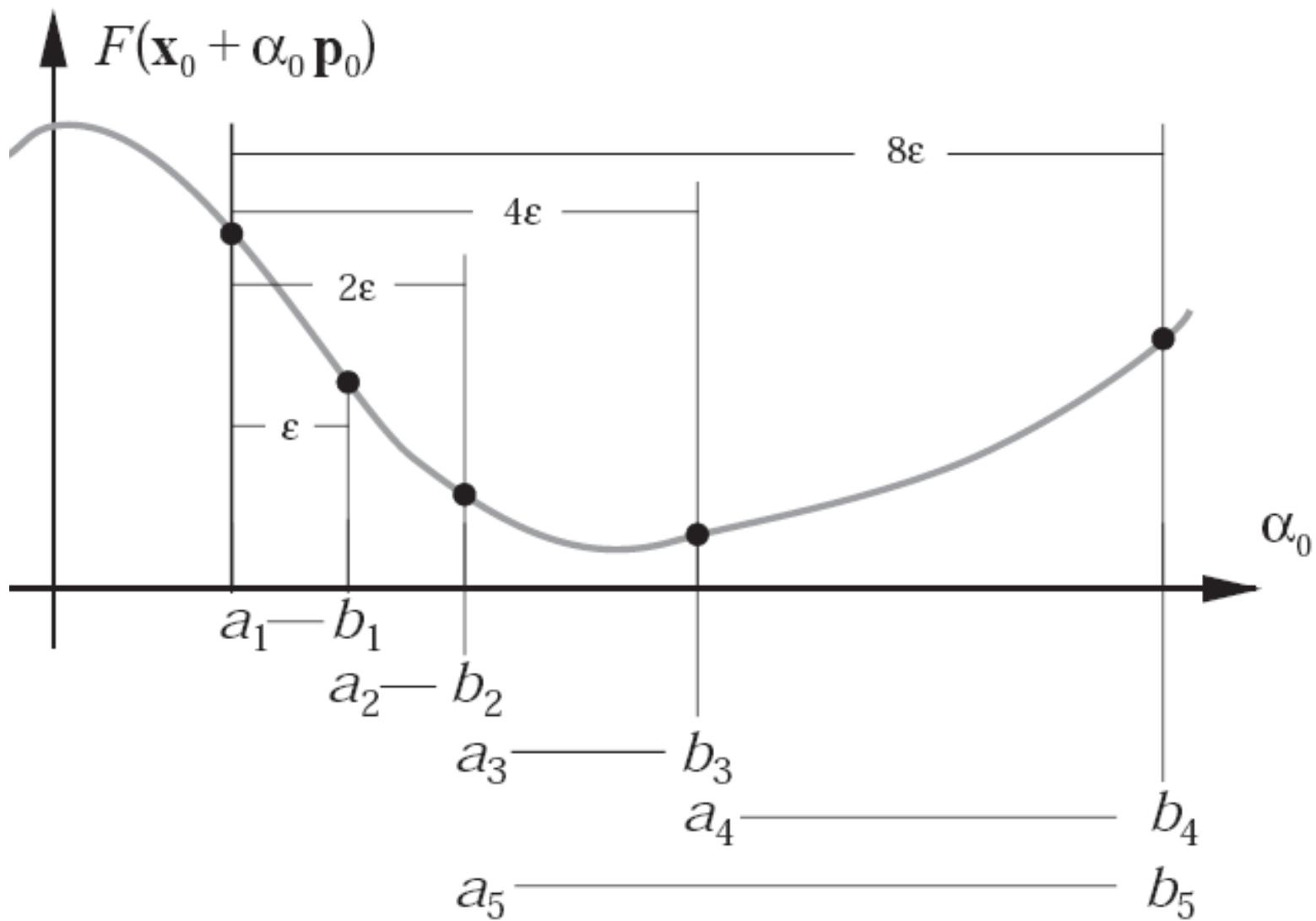
$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

where

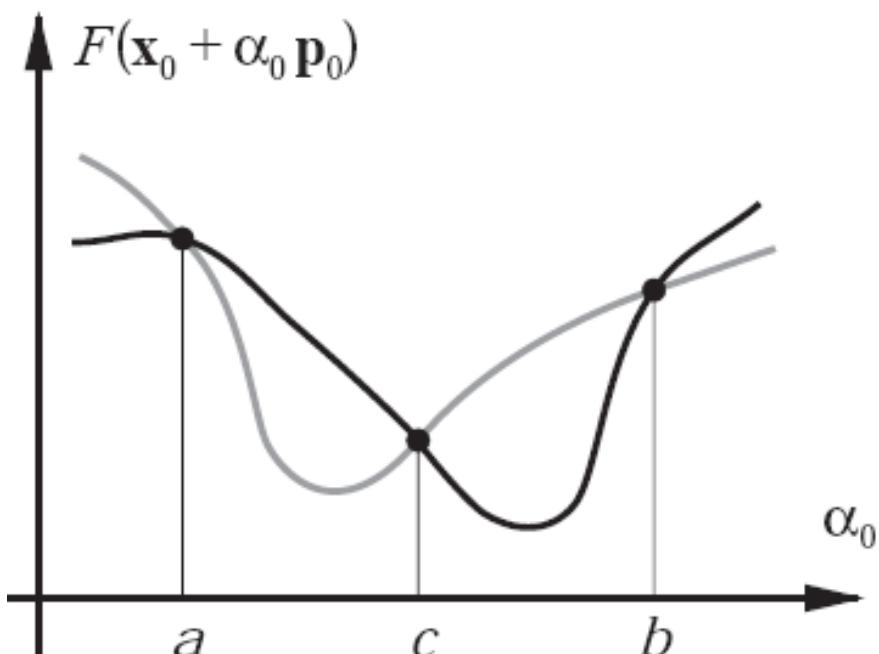
$$\beta_k = \frac{\Delta \mathbf{g}_{k-1}^T \mathbf{g}_k}{\Delta \mathbf{g}_{k-1}^T \mathbf{p}_{k-1}} \quad \text{or} \quad \beta_k = \frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}} \quad \text{or} \quad \beta_k = \frac{\Delta \mathbf{g}_{k-1}^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

- This cannot be applied to neural network training, because the performance index is not quadratic.
  - We cannot use
 
$$\alpha_k = -\frac{\nabla F(\mathbf{x})^T \Big|_{\mathbf{x} = \mathbf{x}_k} \mathbf{p}_k}{\mathbf{p}_k^T \nabla^2 F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}_k} \mathbf{p}_k} = -\frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{p}_k^T \mathbf{A}_k \mathbf{p}_k}$$
 to minimize the function along a line.
  - The exact minimum will not normally be reached in a finite number of steps, and therefore the algorithm will need to be reset after some set number of iterations.
- Locating the minimum of a function
  - Interval location
  - Interval reduction

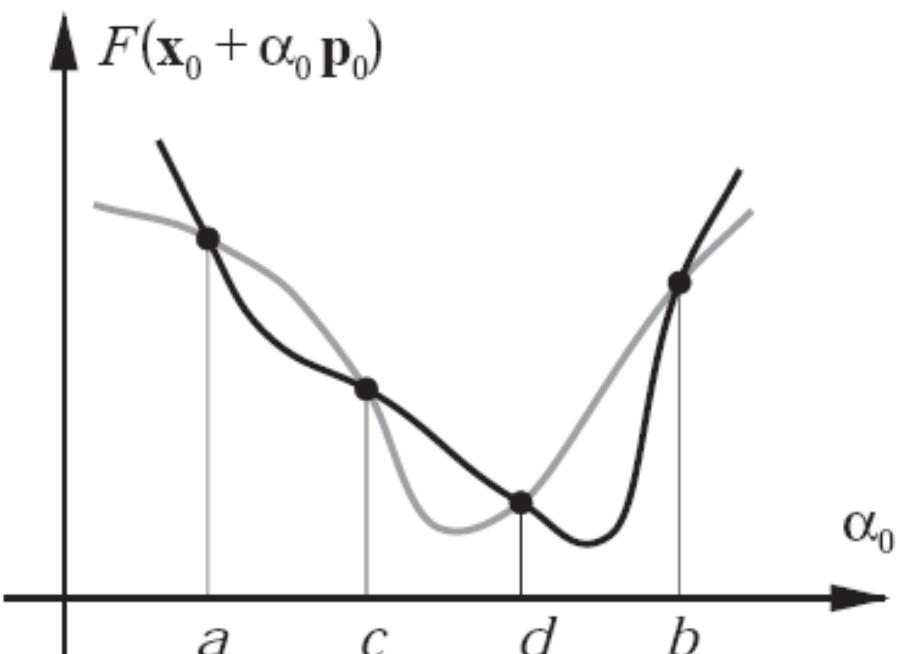
# Interval Location



# Interval Reduction



(a) Interval is not reduced.



(b) Minimum must occur between  $c$  and  $b$ .

# Golden Section Search

$\tau=0.618$

Set  $c_1 = a_1 + (1-\tau)(b_1 - a_1)$ ,  $F_c = F(c_1)$   
 $d_1 = b_1 - (1-\tau)(b_1 - a_1)$ ,  $F_d = F(d_1)$

For  $k=1, 2, \dots$  repeat

If  $F_c < F_d$  then

Set  $a_{k+1} = a_k$ ;  $b_{k+1} = d_k$ ;  $d_{k+1} = c_k$   
 $c_{k+1} = a_{k+1} + (1-\tau)(b_{k+1} - a_{k+1})$   
 $F_d = F_c$ ;  $F_c = F(c_{k+1})$

else

Set  $a_{k+1} = c_k$ ;  $b_{k+1} = b_k$ ;  $c_{k+1} = d_k$   
 $d_{k+1} = b_{k+1} - (1-\tau)(b_{k+1} - a_{k+1})$   
 $F_c = F_d$ ;  $F_d = F(d_{k+1})$

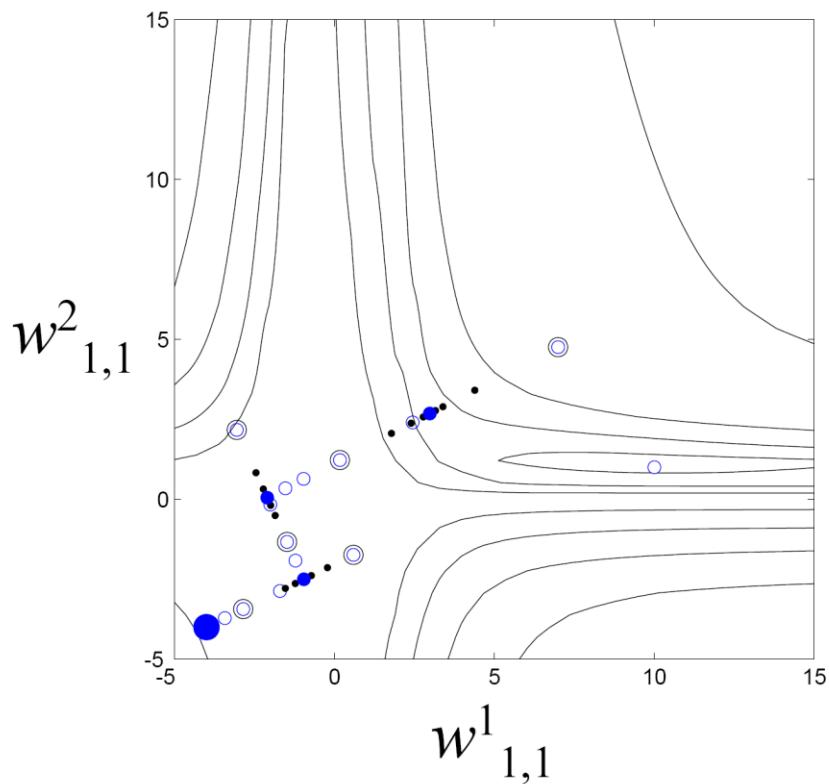
end

end until  $b_{k+1} - a_{k+1} < tolerance$

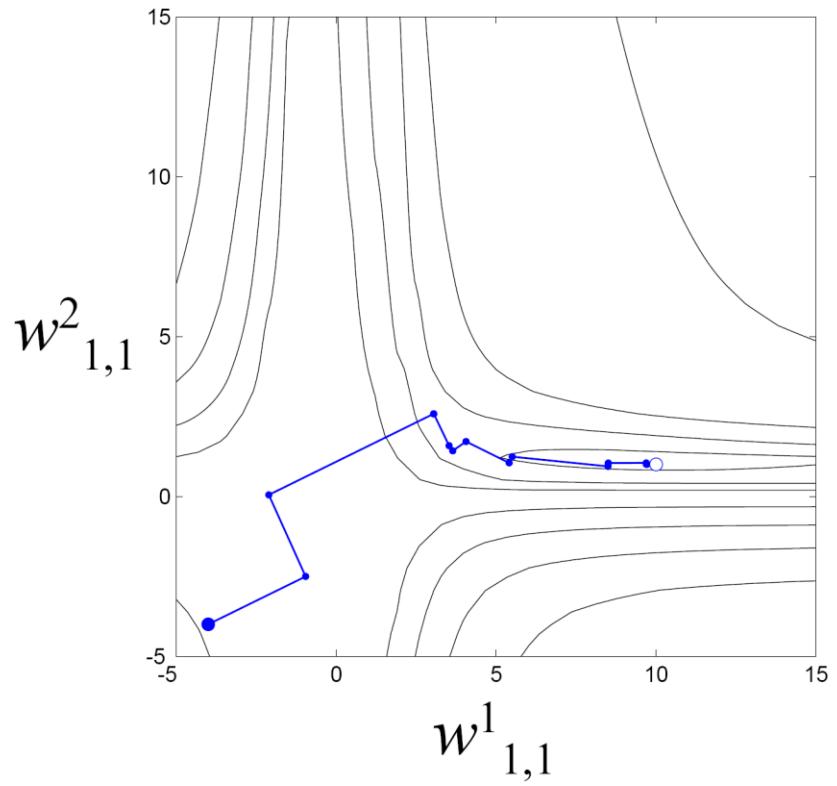
- For quadratic functions the algorithm will converge to the minimum in at most  $n$  (# of parameters) iterations; this normally does not happen for multilayer networks.
- The development of the CG algorithm does not indicate what search direction to use once a cycle of  $n$  iterations has been completed.
- The simplest method is to reset the search direction to the steepest descent direction after  $n$  iterations.
- In the following function approximate example we use the BP algorithm to compute the gradient and the CG algorithm to determine the weight updates. This is a batch mode algorithm.

# Conjugate Gradient BP (CGBP)

Intermediate Steps



Complete Trajectory



[nnd12ls](#) [nnd12cg](#)

# Newton's Method

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1} \mathbf{g}_k$$

$$\mathbf{A}_k \equiv \nabla^2 F(\mathbf{x}) \Big|_{\mathbf{X} = \mathbf{x}_k} \quad \mathbf{g}_k \equiv \nabla F(\mathbf{x}) \Big|_{\mathbf{X} = \mathbf{x}_k}$$

If the performance index is a sum of squares function:

$$F(\mathbf{x}) = \sum_{i=1}^N v_i^2(\mathbf{x}) = \mathbf{v}^T(\mathbf{x}) \mathbf{v}(\mathbf{x})$$

then the  $j$ th element of the gradient is

$$[\nabla F(\mathbf{x})]_j = \frac{\partial F(\mathbf{x})}{\partial x_j} = 2 \sum_{i=1}^N v_i(\mathbf{x}) \frac{\partial v_i(\mathbf{x})}{\partial x_j}$$

# Matrix Form

The gradient can be written in matrix form:

$$\nabla F(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{v}(\mathbf{x})$$

where  $\mathbf{J}$  is the Jacobian matrix:

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial v_1(\mathbf{x})}{\partial x_1} & \frac{\partial v_1(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial v_1(\mathbf{x})}{\partial x_n} \\ \frac{\partial v_2(\mathbf{x})}{\partial x_1} & \frac{\partial v_2(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial v_2(\mathbf{x})}{\partial x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial v_N(\mathbf{x})}{\partial x_1} & \frac{\partial v_N(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial v_N(\mathbf{x})}{\partial x_n} \end{bmatrix} \quad N \times n$$

Now we want to find the **Hessian** matrix

$$[\nabla^2 F(\mathbf{x})]_{k,j} = \frac{\partial^2 F(\mathbf{x})}{\partial x_k \partial x_j} = 2 \sum_{i=1}^N \left\{ \frac{\partial v_i(\mathbf{x})}{\partial x_k} \frac{\partial v_i(\mathbf{x})}{\partial x_j} + v_i(\mathbf{x}) \frac{\partial^2 v_i(\mathbf{x})}{\partial x_k \partial x_j} \right\}$$

$$[\nabla F(\mathbf{x})]_j = \frac{\partial F(\mathbf{x})}{\partial x_j} = 2 \sum_{i=1}^N v_i(\mathbf{x}) \frac{\partial v_i(\mathbf{x})}{\partial x_j}$$

$$\nabla^2 F(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x}) + 2\mathbf{S}(\mathbf{x})$$

where

$$\mathbf{S}(\mathbf{x}) = \sum_{i=1}^N v_i(\mathbf{x}) \nabla^2 v_i(\mathbf{x})$$

# Gauss-Newton Method

Approximate the Hessian matrix as:

$$\nabla^2 F(\mathbf{x}) \cong 2\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x})$$

(if we assume that  $\mathbf{S}(\mathbf{x})$  is small)

We had:

$$\left\{ \begin{array}{l} \nabla F(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{v}(\mathbf{x}) \\ \mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1}\mathbf{g}_k \end{array} \right.$$

Newton's method becomes:

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{x}_k - [2\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k)]^{-1} 2\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k) \\ &= \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k)]^{-1} \mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k) \end{aligned}$$

# Levenberg-Marquardt

Gauss-Newton approximates the Hessian by:

$$\mathbf{H} = \mathbf{J}^T \mathbf{J}$$

This matrix may be singular, but can be made invertible as follows:

$$\mathbf{G} = \mathbf{H} + \mu \mathbf{I}$$

If the eigenvalues and eigenvectors of  $\mathbf{H}$  are:

$$\{\lambda_1, \lambda_2, \dots, \lambda_n\}$$

$$\{\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_n\}$$

then

Eigenvalues of  $\mathbf{G}$

$$\mathbf{G}\mathbf{z}_i = [\mathbf{H} + \mu \mathbf{I}]\mathbf{z}_i = \mathbf{H}\mathbf{z}_i + \mu \mathbf{z}_i = \lambda_i \mathbf{z}_i + \mu \mathbf{z}_i = \overbrace{(\lambda_i + \mu)}^{\text{Eigenvalues of } \mathbf{G}} \mathbf{z}_i$$

$\mathbf{G}$  can be made positive definite by increasing  $\mu$  until  $\lambda_i + \mu > 0$  for all  $i$ .

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \mu_k \mathbf{I}]^{-1} \mathbf{J}^T(\mathbf{x}_k) \mathbf{v}(\mathbf{x}_k)$$

# Adjustment of $\mu_k$

As  $\mu_k \rightarrow 0$ , LM becomes Gauss-Newton.

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k)]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k)$$

As  $\mu_k \rightarrow \infty$ , LM becomes Steepest Descent with small learning rate.

$$\mathbf{x}_{k+1} \cong \mathbf{x}_k - \frac{1}{\mu_k} \mathbf{J}^T(\mathbf{x}_k) \mathbf{v}(\mathbf{x}_k) = \mathbf{x}_k - \frac{1}{2\mu_k} \nabla F(\mathbf{x})$$

Therefore, begin with a small  $\mu_k$  to use Gauss-Newton and speed convergence. If a step does not yield a smaller  $F(\mathbf{x})$ , then repeat the step with an increased  $\mu_k$  until  $F(\mathbf{x})$  is decreased.  $F(\mathbf{x})$  must decrease eventually, since we will be taking a very small step in the steepest descent direction.

# Application To Multilayer Network

The performance index for the multilayer network is:

$$\text{→ } F(\mathbf{x}) = \sum_{q=1}^Q (\mathbf{t}_q - \mathbf{a}_q)^T (\mathbf{t}_q - \mathbf{a}_q) = \sum_{q=1}^Q \mathbf{e}_q^T \mathbf{e}_q = \sum_{q=1}^Q \sum_{j=1}^{S^M} (e_{j,q})^2 = \sum_{i=1}^N (v_i)^2$$

Equal probability

Where  $e_{j,q}$  is the  $j$ th element of the error for the  $q$ th input/target pair.

This is similar to performance index, for which LM was designed.

In standard BP we compute the derivatives of the *squared errors*, with respect to weights and biases. To create matrix  $\mathbf{J}$  we need to compute the derivatives of *errors*.

The error vector is:

$$\mathbf{v}^T = \begin{bmatrix} v_1 & v_2 & \dots & v_N \end{bmatrix} = \begin{bmatrix} e_{1,1} & e_{2,1} & \dots & e_{S^M,1} & e_{1,2} & \dots & e_{S^M,Q} \end{bmatrix}$$

The parameter vector is:

$$\mathbf{x}^T = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix} = \begin{bmatrix} w_{1,1}^1 & w_{1,2}^1 & \dots & w_{S^1,R}^1 & b_1^1 & \dots & b_{S^1}^1 & w_{1,1}^2 & \dots & b_{S^M}^M \end{bmatrix}$$

The dimensions of the two vectors are:

$$N = Q \times S^M , \quad n = S^1(R+1) + S^2(S^1+1) + \dots + S^M(S^{M-1}+1)$$

If we make these substitutions into the Jacobian matrix for multilayer network training we have:

# Jacobian Matrix

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial e_{1,1}}{\partial w_{1,1}^1} & \frac{\partial e_{1,1}}{\partial w_{1,2}^1} & \dots & \frac{\partial e_{1,1}}{\partial w_{S^1,R}^1} & \frac{\partial e_{1,1}}{\partial b_1^1} & \dots \\ \vdots & \vdots & & \vdots & \vdots & \\ \frac{\partial e_{2,1}}{\partial w_{1,1}^1} & \frac{\partial e_{2,1}}{\partial w_{1,2}^1} & \dots & \frac{\partial e_{2,1}}{\partial w_{S^1,R}^1} & \frac{\partial e_{2,1}}{\partial b_1^1} & \dots \\ \vdots & \vdots & & \vdots & \vdots & \\ \frac{\partial e_{S^M,1}}{\partial w_{1,1}^1} & \frac{\partial e_{S^M,1}}{\partial w_{1,2}^1} & \dots & \frac{\partial e_{S^M,1}}{\partial w_{S^1,R}^1} & \frac{\partial e_{S^M,1}}{\partial b_1^1} & \dots \\ \frac{\partial e_{1,2}}{\partial w_{1,1}^1} & \frac{\partial e_{1,2}}{\partial w_{1,2}^1} & \dots & \frac{\partial e_{1,2}}{\partial w_{S^1,R}^1} & \frac{\partial e_{1,2}}{\partial b_1^1} & \dots \\ \vdots & \vdots & & \vdots & \vdots & \end{bmatrix} N \times n$$

# Computing The Jacobian

SDBP computes terms like:

$$\frac{\partial \hat{F}(\mathbf{x})}{\partial x_l} = \frac{\partial \mathbf{e}_q^T \mathbf{e}_q}{\partial x_l}$$

using the chain rule:

$$\frac{\partial \hat{F}}{\partial w_{i,j}^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^m}$$

where the sensitivity

$$s_i^m \equiv \frac{\partial \hat{F}}{\partial n_i^m}$$

is computed using backpropagation.

For the Jacobian we need to compute terms like:

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial x_l}$$

# Marquardt Sensitivity

If we define a Marquardt sensitivity:

$$\tilde{s}_{i,h}^m \equiv \frac{\partial v_h}{\partial n_{i,q}^m} = \frac{\partial e_{k,q}}{\partial n_{i,q}^m}, \quad h \stackrel{\Delta}{=} (q-1)S^M + k$$

We can compute the Jacobian as follows:

weight

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial w_{i,j}^m} = \frac{\partial e_{k,q}}{\partial n_{i,q}^m} \times \frac{\partial n_{i,q}^m}{\partial w_{i,j}^m} = \tilde{s}_{i,h}^m \times \frac{\partial n_{i,q}^m}{\partial w_{i,j}^m} = \tilde{s}_{i,h}^m \times a_{j,q}^{m-1}$$

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial b_i^m} = \frac{\partial e_{k,q}}{\partial n_{i,q}^m} \times \frac{\partial n_{i,q}^m}{\partial b_i^m} = \tilde{s}_{i,h}^m \times \frac{\partial n_{i,q}^m}{\partial b_i^m} = \tilde{s}_{i,h}^m$$

# Computing the Sensitivities

Initialization

$$\tilde{s}_{i,h}^M = \frac{\partial v_h}{\partial n_{i,q}^M} = \frac{\partial e_{k,q}}{\partial n_{i,q}^M} = \frac{\partial(t_{k,q} - a_{k,q}^M)}{\partial n_{i,q}^M} = -\frac{\partial a_{k,q}^M}{\partial n_{i,q}^M}$$
$$\tilde{S}_{i,h}^M = \begin{cases} \dot{f}(n_{i,q}^M) & \text{for } i = k \\ 0 & \text{for } i \neq k \end{cases}$$

Therefore when the input  $\mathbf{p}_q$  has been applied to the network and the corresponding network output  $\mathbf{a}_q^M$  has been computed, the LMBP is initialized with

$$\tilde{\mathbf{S}}_q^M = -\dot{\mathbf{F}}(n_q^M)$$

Where

$$\dot{\mathbf{F}}^m(\mathbf{n}^m) = \begin{bmatrix} \dot{f}^m(n_1^m) & 0 & \dots & 0 \\ 0 & \dot{f}^m(n_2^m) & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & \dot{f}^m(n_{S^m}^m) \end{bmatrix}$$

Each column of the matrix  $\tilde{\mathbf{S}}_q^M$  must be backpropagated through the network using the following equation (Ch11) to produce one row of the Jacobian matrix.

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m) (\mathbf{W}^{m+1})^T \mathbf{s}^{m+1}$$

The columns can also be backpropagated together using

$$\tilde{\mathbf{S}}_q^m = \dot{\mathbf{F}}^m(\mathbf{n}_q^m)(\mathbf{W}^{m+1})^T \tilde{\mathbf{S}}_q^{m+1}$$

The total Marquardt sensitivity matrices for each layer are then created by augmenting the matrices computed for each input:

$$\tilde{\mathbf{S}}^m = [\tilde{\mathbf{S}}_1^m \quad \tilde{\mathbf{S}}_2^m \quad \dots \quad \tilde{\mathbf{S}}_Q^m]$$

Note that for each input we will backpropagate  $\mathbf{S}^M$  sensitivity vectors. Because we compute the derivatives of each **individual** error, rather than the derivative of **the sum of squares of the errors**. For every input we have  $\mathbf{S}^M$  errors. For each error there will be one row of the Jacobian matrix.

- After the sensitivities have been backpropagated, the Jacobian matrix is computed using:

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial w_{i,j}^m} = \frac{\partial e_{k,q}}{\partial n_{i,q}^m} \times \frac{\partial n_{i,q}^m}{\partial w_{i,j}^m} = \tilde{s}_{i,h}^m \times \frac{\partial n_{i,q}^m}{\partial w_{i,j}^m} = \tilde{s}_{i,h}^m \times a_{j,q}^{m-1}$$

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial b_i^m} = \frac{\partial e_{k,q}}{\partial n_{i,q}^m} \times \frac{\partial n_{i,q}^m}{\partial b_i^m} = \tilde{s}_{i,h}^m \times \frac{\partial n_{i,q}^m}{\partial b_i^m} = \tilde{s}_{i,h}^m$$

# LMBP (summarized)

- Present all inputs to the network and compute the corresponding network outputs and the errors. Compute the sum of squared errors over all inputs.

$$\mathbf{e}_q = \mathbf{t}_q - \mathbf{a}_q^M$$

$$F(\mathbf{x}) = \sum_{q=1}^Q (\mathbf{t}_q - \mathbf{a}_q)^T (\mathbf{t}_q - \mathbf{a}_q) = \sum_{q=1}^Q \mathbf{e}_q^T \mathbf{e}_q = \sum_{q=1}^Q \sum_{j=1}^{S^M} (e_{j,q})^2 = \sum_{i=1}^N (v_i)^2$$

- Compute the Jacobian matrix. Calculate the sensitivities with the backpropagation algorithm, after initializing. Augment the individual matrices into the Marquardt sensitivities. Compute the elements of the Jacobian matrix.

$$\tilde{\mathbf{S}}_q^M = -\overset{\bullet}{\mathbf{F}}^M(\mathbf{n}_q^M)$$

$$\tilde{\mathbf{S}}_q^m = \overset{\bullet}{\mathbf{F}}^m(\mathbf{n}_q^m)(\mathbf{W}^{m+1})^T \tilde{\mathbf{S}}_q^{m+1}, \quad m = M-1, \dots, 2, 1$$

$$\tilde{\mathbf{S}}^m = \begin{bmatrix} \tilde{\mathbf{S}}_1^m & \tilde{\mathbf{S}}_2^m & \dots & \tilde{\mathbf{S}}_{\mathcal{Q}}^m \end{bmatrix}$$

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial w_{i,j}^m} = \frac{\partial e_{k,q}}{\partial n_{i,q}^m} \times \frac{\partial n_{i,q}^m}{\partial w_{i,j}^m} = \tilde{s}_{i,h}^m \times \frac{\partial n_{i,q}^m}{\partial w_{i,j}^m} = \tilde{s}_{i,h}^m \times a_{j,q}^{m-1}$$

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial b_i^m} = \frac{\partial e_{k,q}}{\partial n_{i,q}^m} \times \frac{\partial n_{i,q}^m}{\partial b_i^m} = \tilde{s}_{i,h}^m \times \frac{\partial n_{i,q}^m}{\partial b_i^m} = \tilde{s}_{i,h}^m$$

- Solve the following Eq. to obtain the change in the weights.

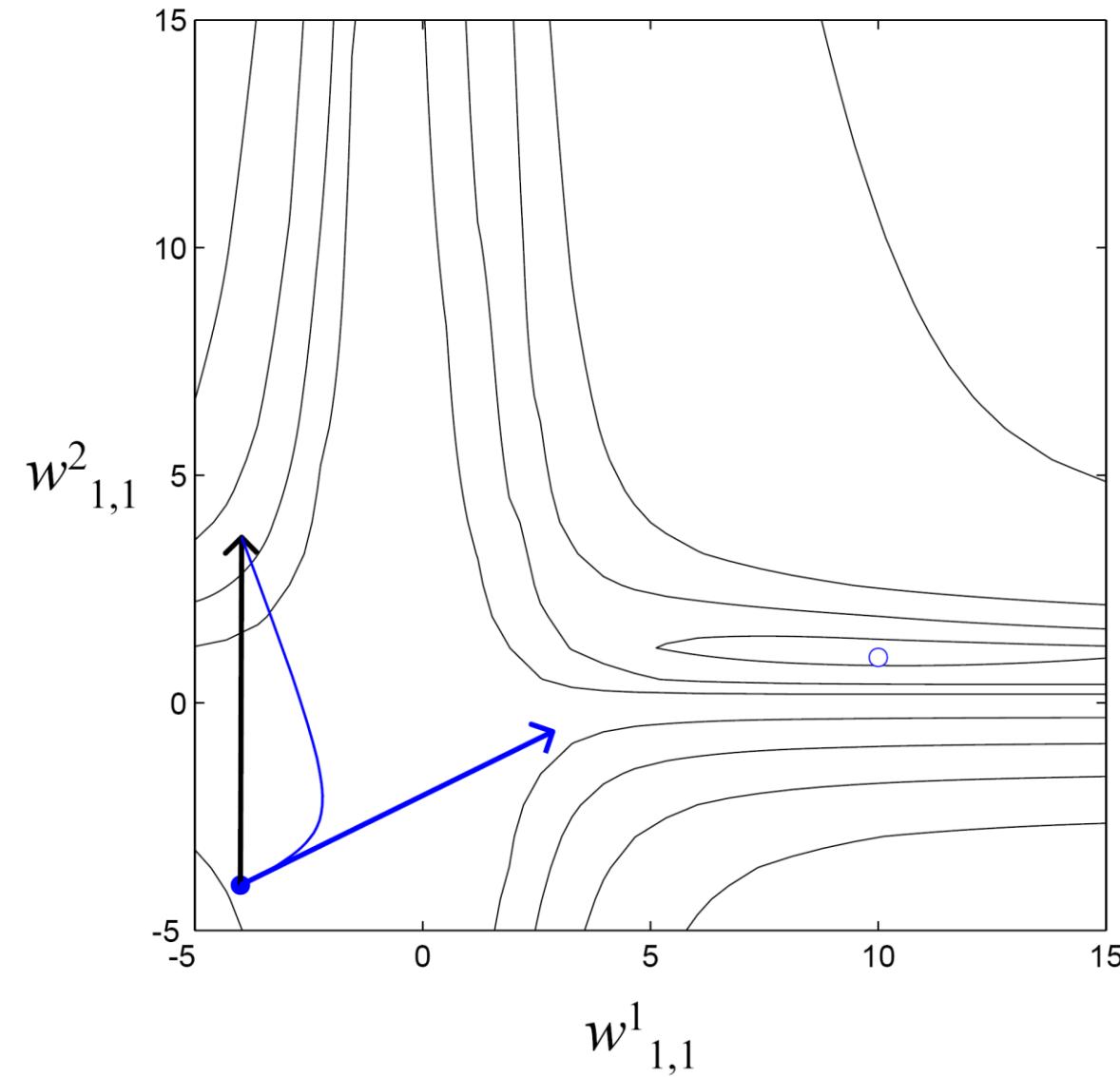
$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \mu_k \mathbf{I}]^{-1} \mathbf{J}^T(\mathbf{x}_k) \mathbf{v}(\mathbf{x}_k) \quad \Delta \mathbf{x}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$$

- Recompute the sum of squared errors with the new weights. If this new sum of squares is smaller than that computed in step 1, then divide  $\mu_k$  by  $v$ , update the weights and go back to step 1. If the sum of squares is not reduced, then multiply  $\mu_k$  by  $v$  and go back to step 3.

The algorithm is assumed to have converged when the norm of the gradient is less than some predetermined value, or when the sum of squares has been reduced to some error goal.

See P12.5 for a numerical illustration of Jacobian computation.

# Example LMBP Step

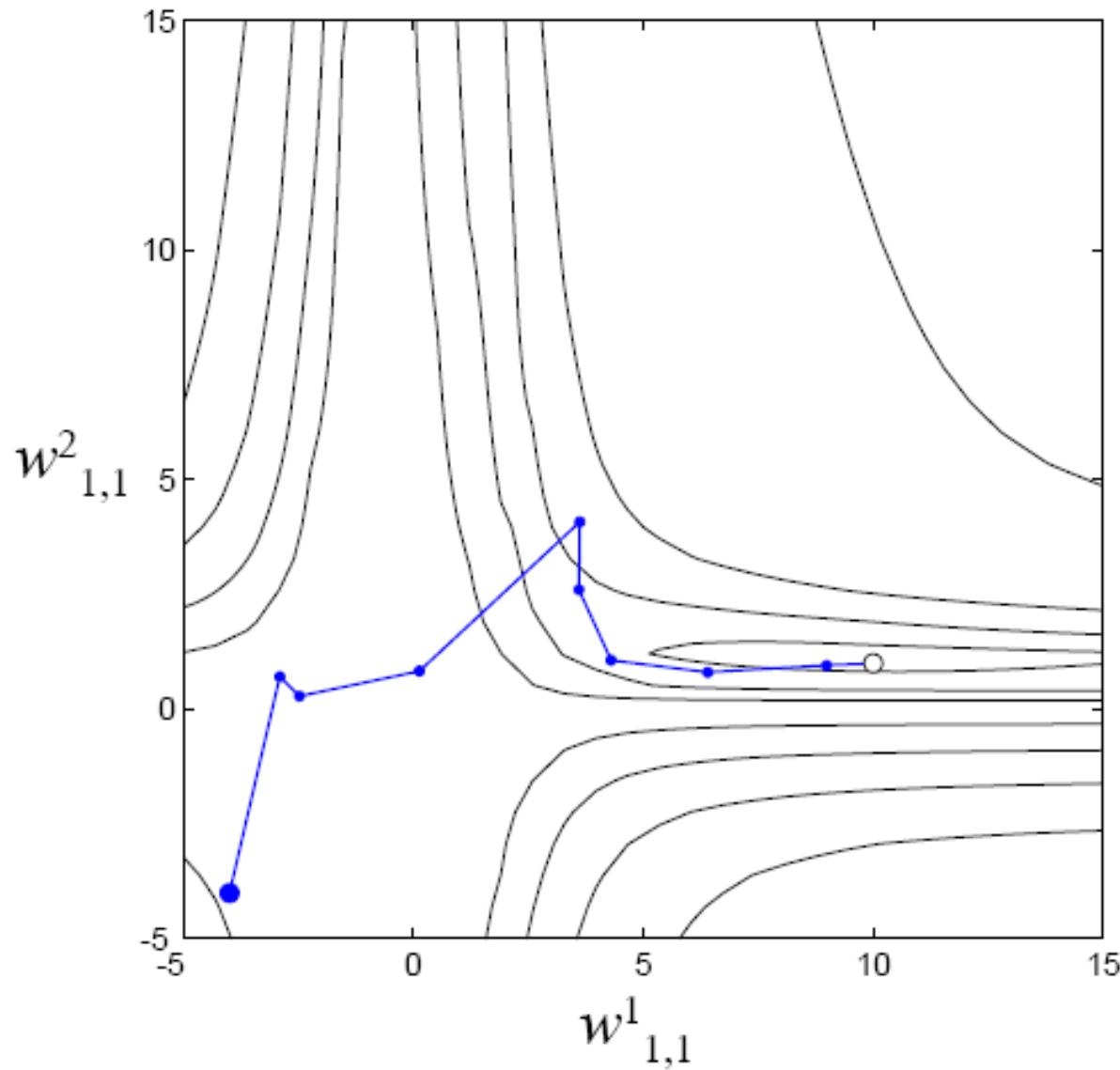


**Black arrow:** small  $\mu_k$   
(Gauss-Newton direction)

**Blue arrow :** large  $\mu_k$   
(SD direction)

**Blue curve:** LM for  
intermediate  $\mu_k$

# LMBP Trajectory



nnd12ms

nnd12m

Storage requirement:  
 $n \times n$  for Hessian  
matrix



# Generalization



- The network input-output mapping is accurate for the training data and for test data never seen before.
- The network interpolates well.

# Cause of Overfitting



Poor generalization is caused by using a network that is too complex (too many neurons/parameters). To have the best performance we need to find the least complex network that can represent the data



Find the simplest model that explains the data.

# Problem Statement



Training Set

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \dots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Underlying Function

$$\mathbf{t}_q = \mathbf{g}(\mathbf{p}_q) + \boldsymbol{\varepsilon}_q$$

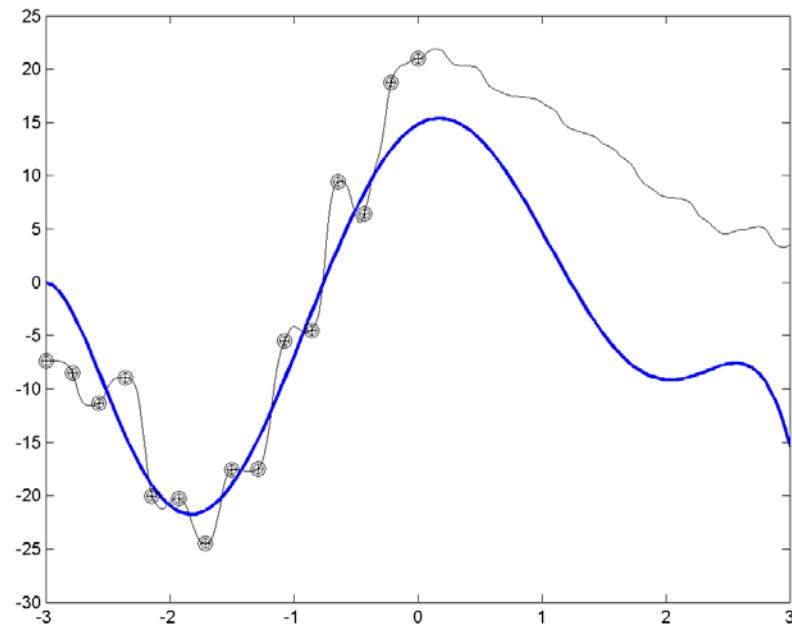
Performance Function

$$F(\mathbf{x}) = E_D = \sum_{q=1}^Q (\mathbf{t}_q - \mathbf{a}_q)^T (\mathbf{t}_q - \mathbf{a}_q)$$

# Poor Generalization



Overfitting      Extrapolation

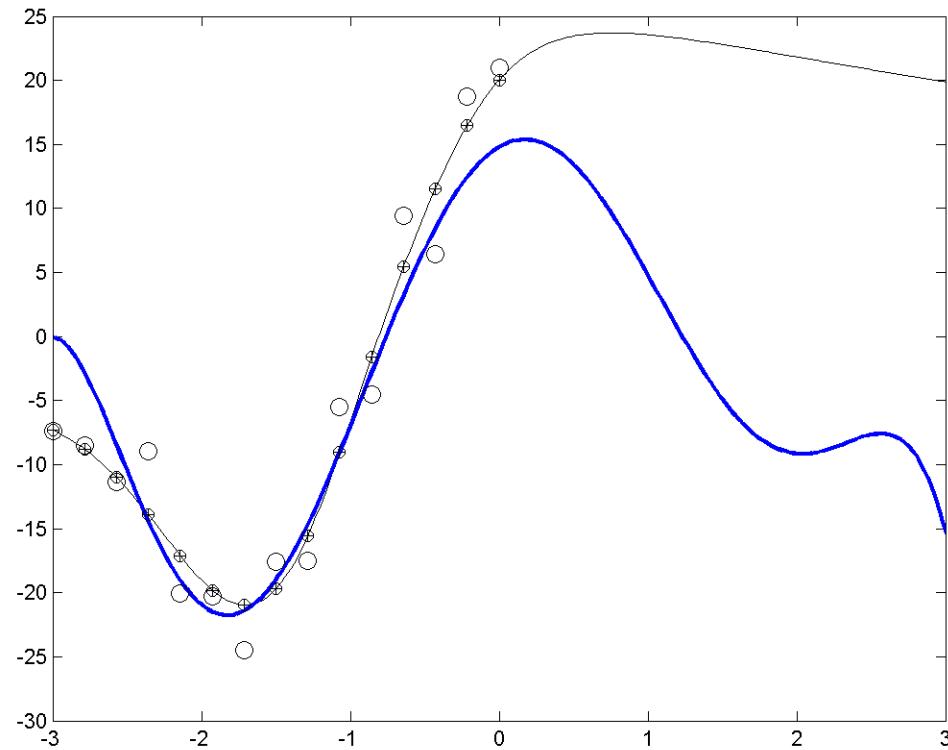


Interpolation

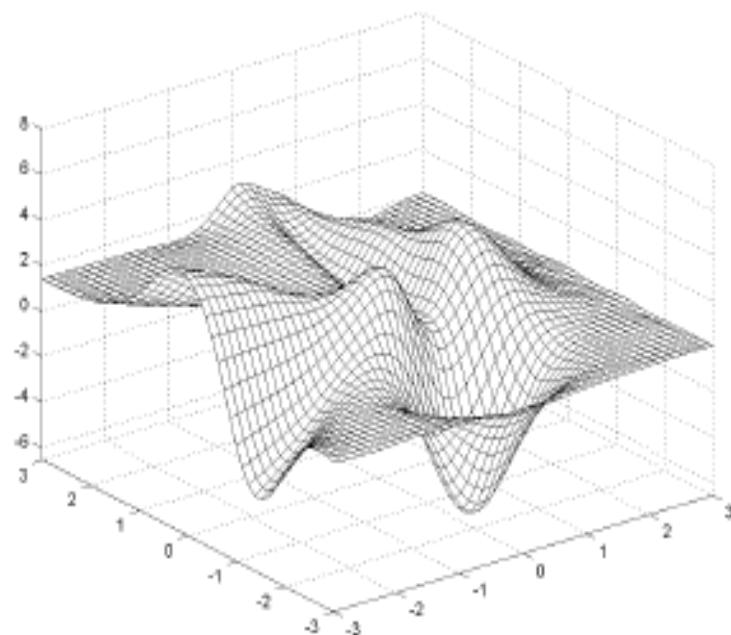
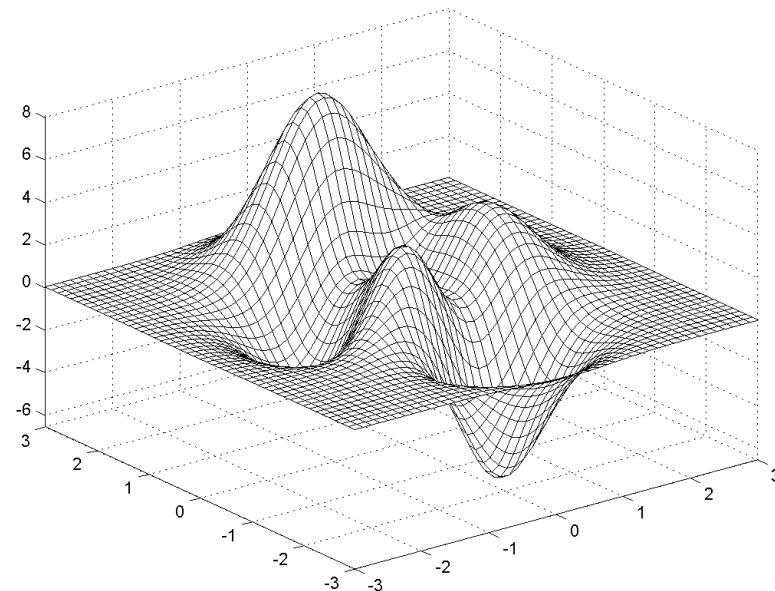
# Good Generalization



Interpolation      Extrapolation



# Extrapolation in 2-D





## Test Set

- Part of the available data is set aside during the training process.
- After training, the network error on the test set is used as a measure of generalization ability.
- The test set must never be used in any way to train the network.
- The test set must be representative of all situations for which the network will be used.



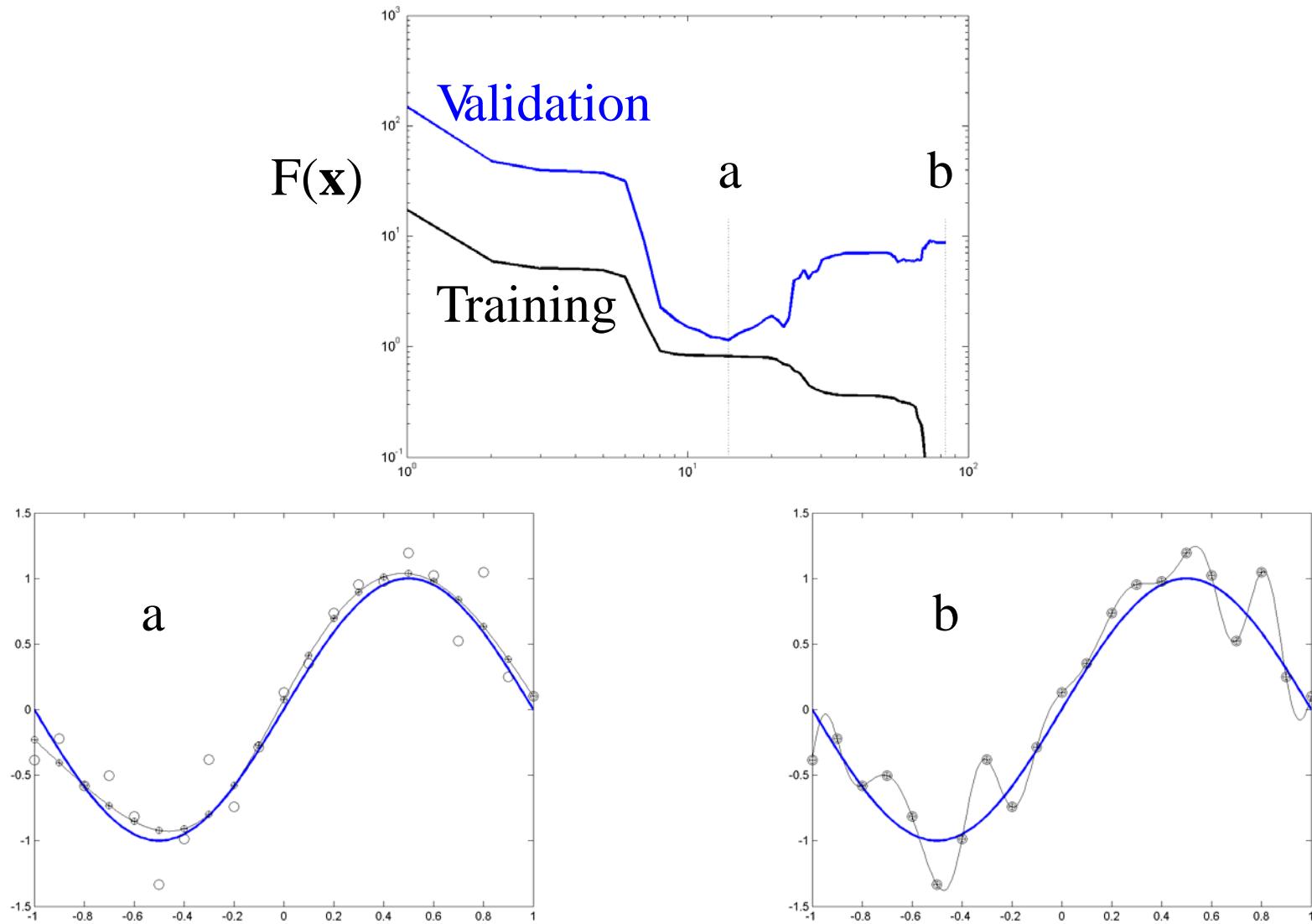
- Pruning (removing neurons) until the performance is degraded.
- Growing (adding neurons) until the performance is adequate.
- Validation Methods
- Regularization

# Early Stopping



- Break up data into training, *validation*, and test sets.
- Use only the training set to compute gradients and determine weight updates.
- Compute the performance on the validation set at each iteration of training.
- Stop training when the performance on the validation set goes up for a specified number of iterations.
- Use the weights which achieved the lowest error on the validation set.

# Early Stopping Example



# Regularization



## Standard Performance Measure

$$F = E_D$$

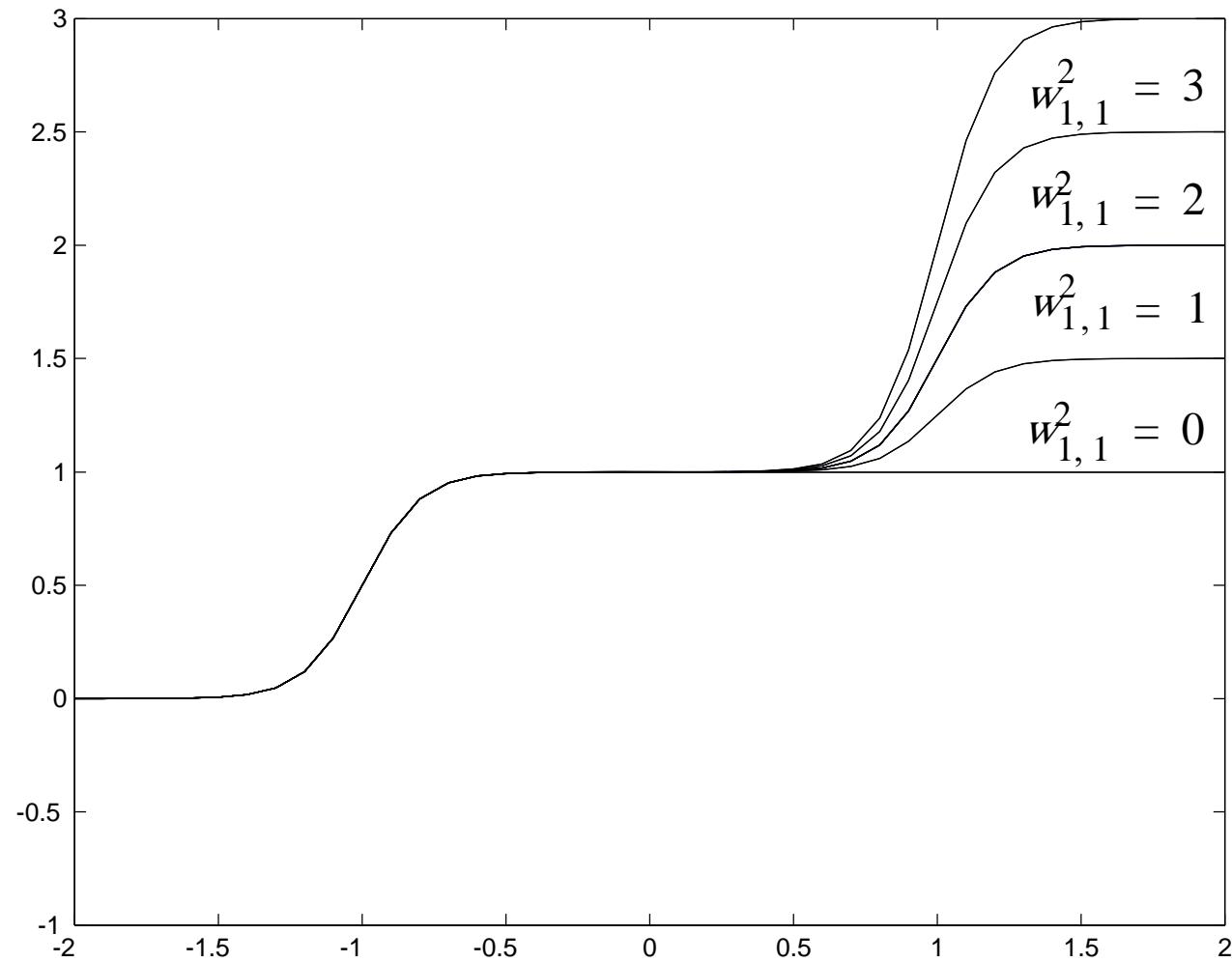
## Performance Measure with Regularization

$$F = \beta E_D + \alpha E_W = \beta \sum_{q=1}^Q (\mathbf{t}_q - \mathbf{a}_q)^T (\mathbf{t}_q - \mathbf{a}_q) + \alpha \sum_{i=1}^n x_i^2$$

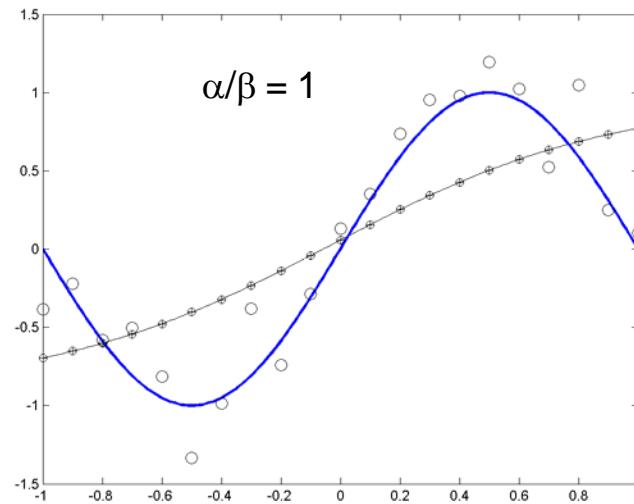
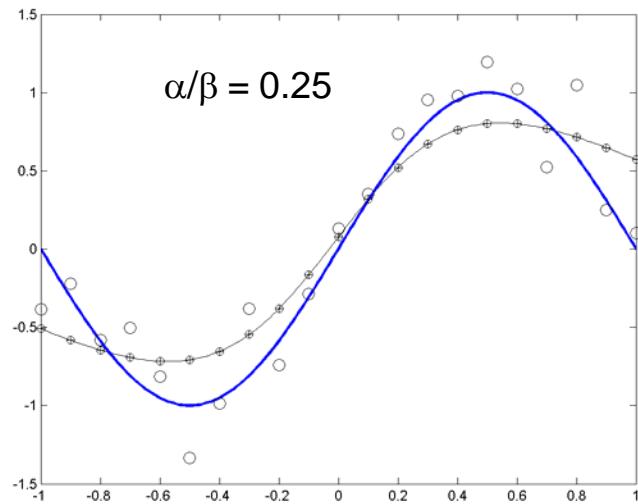
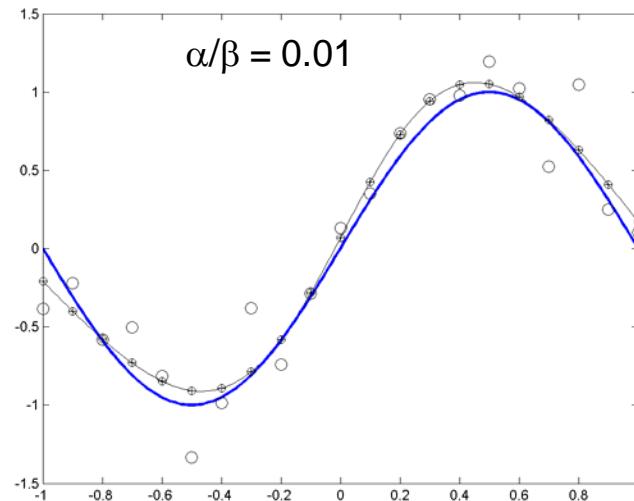
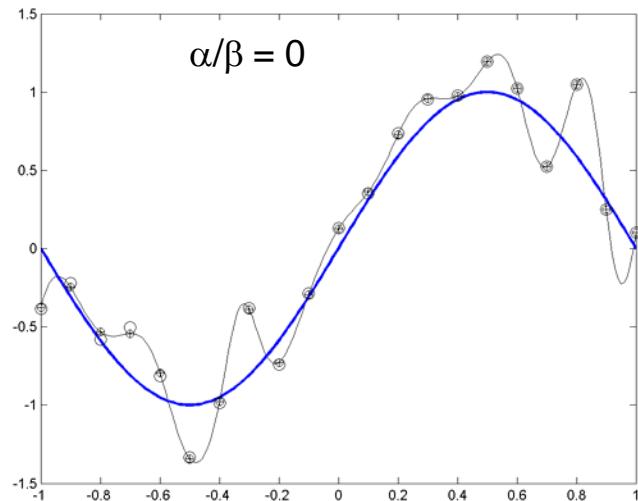
Complexity Penalty

(Smaller weights means a smoother function.)

# Effect of Weight Changes



# Effect of Regularization



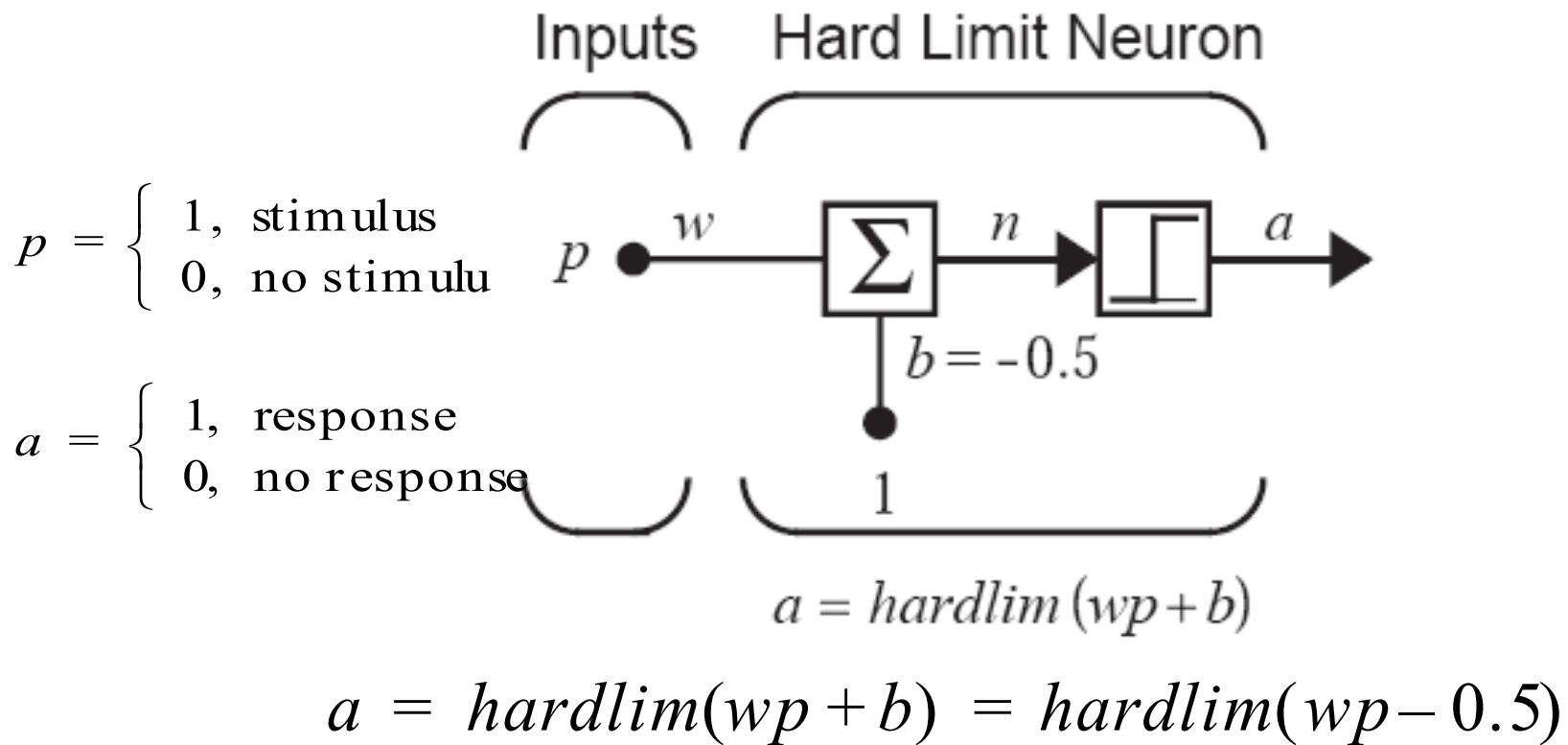
# Associative Learning

- We discuss a collection of simple rules that allow unsupervised learning.
- These rules give networks the ability to learn associations between patterns that occur together frequently. → pattern recognition and recall.
- How associations can be represented by a network?
- How a network can learn new associations?
- An association is any link between a system's input and output such that when a pattern A (*stimulus*) is presented to the system it will respond with pattern B (*response*).

# Hebb's Postulate

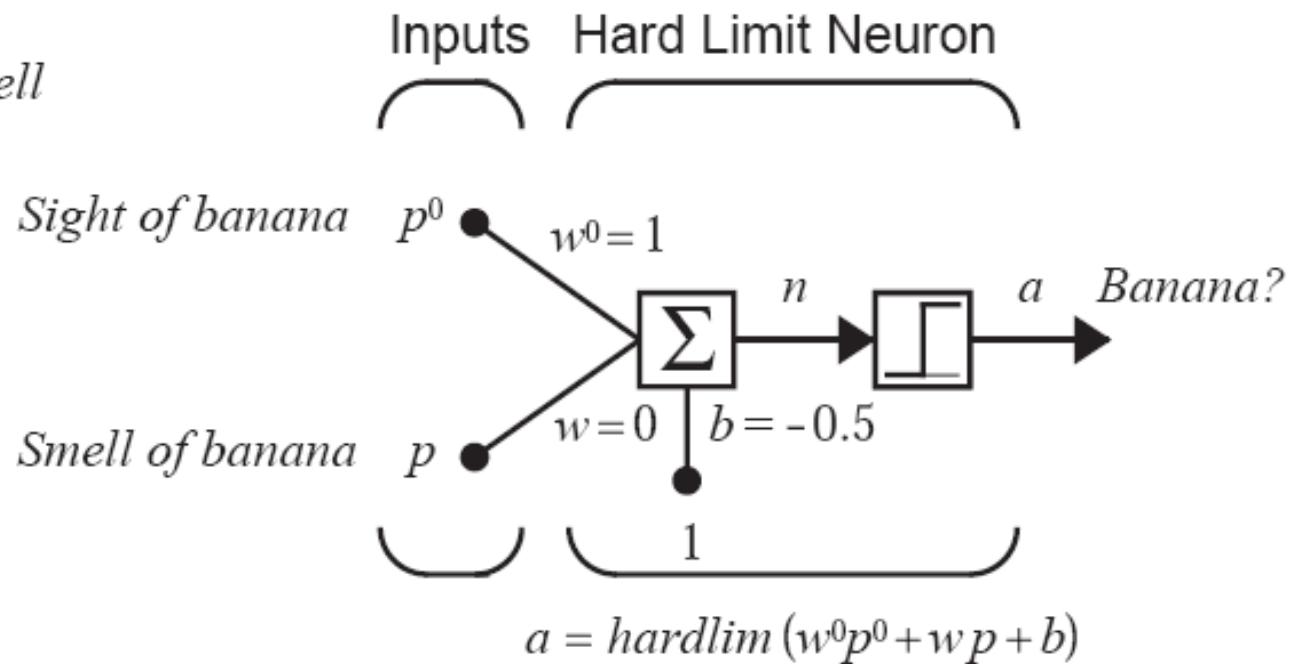
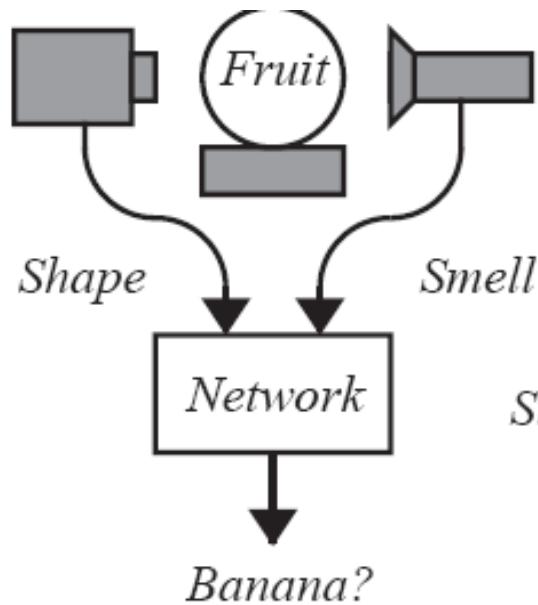
“When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A’s efficiency, as one of the cells firing B, is increased.”

# Simple Associative Network



The network will respond to the stimulus only if  $w$  is greater than  $-b$  (in this case 0.5).

# Banana Associator



Unconditioned Stimulus (~dog's food)

$$p^0 = \begin{cases} 1, & \text{shape detected} \\ 0, & \text{shape not detected} \end{cases}$$

Conditioned Stimulus (~bell)

$$p = \begin{cases} 1, & \text{smell detected} \\ 0, & \text{smell not detected} \end{cases}$$

- One set of inputs will represent the *unconditioned stimulus*.
- Another set of inputs will represent the *conditioned stimulus*.
- We will represent the unconditioned stimulus as  $p_0$  and the conditioned stimulus simply as  $p$ . For our purposes we will assume that the weights associated with  $p_0$  are fixed, but that the weights associated with  $p$  are adjusted according to the relevant learning rule.

# Unsupervised Hebb Rule

$$w_{ij}(q) = w_{ij}(q - 1) + \alpha a_i(q)p_j(q)$$

$\alpha$  dictates how many times a stimulus and response must occur together before an association is made.

Vector Form:

$$\mathbf{W}(q) = \mathbf{W}(q - 1) + \alpha \mathbf{a}(q) \mathbf{p}^T(q)$$

Training Sequence       $\mathbf{p}(1), \mathbf{p}(2), \dots, \mathbf{p}(Q)$

# Banana Recognition Example

Initial Weights:

$$w^0 = 1, w(0) = 0$$

Training Sequence:  $\{p^0(1) = 0, p(1) = 1\}, \{p^0(2) = 1, p(2) = 1\}, \dots$

$$\alpha = 1 \rightarrow w(q) = w(q-1) + a(q)p(q)$$

First Iteration (sight fails):

$$\begin{aligned} a(1) &= \text{hardlim}(w^0 p^0(1) + w(0)p(1) - 0.5) \\ &= \text{hardlim}(1 \times 0 + 0 \times 1 - 0.5) = 0 \quad (\text{no response}) \end{aligned}$$

$$w(1) = w(0) + a(1)p(1) = 0 + 0 \cdot 1 = 0$$

# Example

Second Iteration (sight works):

$$\begin{aligned}a(2) &= \text{hardlim}(w^0 p^0(2) + w(1)p(2) - 0.5) \\&= \text{hardlim}(1 \times 1 + 0 \times 1 - 0.5) = 1 \quad (\text{banana})\end{aligned}$$

$$w(2) = w(1) + a(2)p(2) = 0 + 1 \cdot 1 = 1$$

Third Iteration (sight fails):

$$\begin{aligned}a(3) &= \text{hardlim}(w^0 p^0(3) + w(2)p(3) - 0.5) \\&= \text{hardlim}(1 \cdot 0 + 1 \cdot 1 - 0.5) = 1 \quad (\text{banana})\end{aligned}$$

$$w(3) = w(2) + a(3)p(3) = 1 + 1 \cdot 1 = 2$$

Banana will now be detected if either sensor works.

# Problems with Hebb Rule

- Weights can become arbitrarily large (in biological systems synapses cannot grow without bound).
- There is no mechanism for weights to decrease. If the inputs or outputs of a Hebb network experience any noise, every weight will grow until the net responds to any stimulus.

# Hebb Rule with Decay

$$\mathbf{W}(q) = \mathbf{W}(q-1) + \alpha \mathbf{a}(q) \mathbf{p}^T(q) - \gamma \mathbf{W}(q-1)$$

$$\mathbf{W}(q) = (1 - \gamma) \mathbf{W}(q-1) + \alpha \mathbf{a}(q) \mathbf{p}^T(q)$$

As decay rate ( $\gamma$ ) approaches one, the learning law quickly forgets old inputs and remembers only the most recent patterns. This keeps the weight matrix from growing without bound, which can be demonstrated by setting both  $a_i$  and  $p_j$  to 1:

$$w_{ij}^{max} = (1 - \gamma) w_{ij}^{max} + \alpha a_i p_j$$

$$w_{ij}^{max} = (1 - \gamma) w_{ij}^{max} + \alpha$$

$$w_{ij}^{max} = \frac{\alpha}{\gamma}$$

# Example: Banana Associator

$$\alpha = 1$$

$$\gamma = 0.1$$

First Iteration (sight fails):

$$\begin{aligned} a(1) &= \text{hardlim}(w^0 p^0(1) + w(0)p(1) - 0.5) \\ &= \text{hardlim}(1 \times 0 + 0 \times 1 - 0.5) = 0 \quad (\text{no response}) \end{aligned}$$

$$w(1) = w(0) + a(1)p(1) - 0.1w(0) = 0 + 0 \cdot 1 - 0.1(0) = 0$$

Second Iteration (sight works):

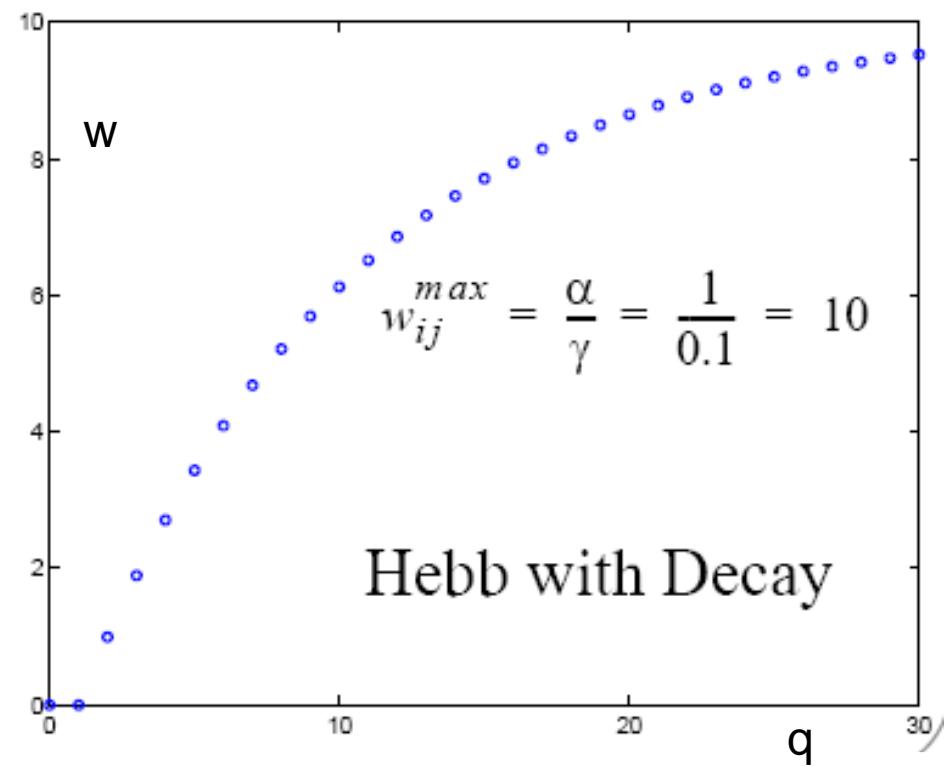
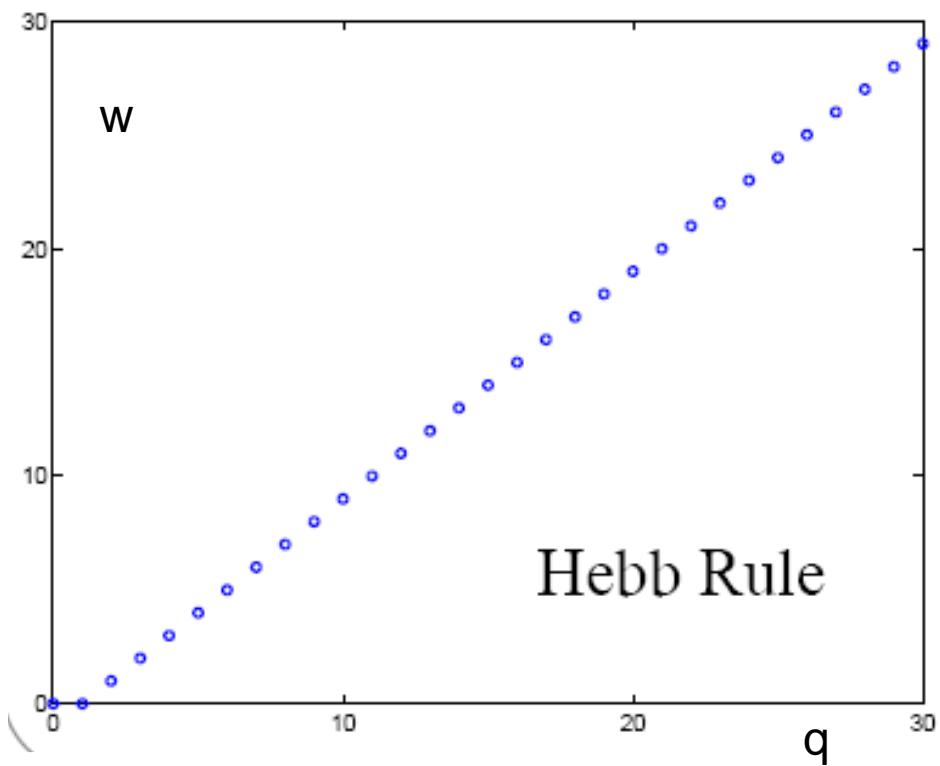
$$\begin{aligned} a(2) &= \text{hardlim}(w^0 p^0(2) + w(1)p(2) - 0.5) \\ &= \text{hardlim}(1 \times 1 + 0 \times 1 - 0.5) = 1 \quad (\text{banana}) \end{aligned}$$

$$w(2) = w(1) + a(2)p(2) - 0.1w(1) = 0 + 1 \cdot 1 - 0.1(0) = 1 \quad 12$$

### Third Iteration (sight fails):

$$\begin{aligned}a(3) &= \text{hardlim}(w^0 p^0(3) + w(2)p(3) - 0.5) \\&= \text{hardlim}(1 \cdot 0 + 1 \cdot 1 - 0.5) = 1 \quad (\text{banana})\end{aligned}$$

$$w(3) = w(2) + a(3)p(3) - 0.1w(3) = 1 + 1 \cdot 1 - 0.1(1) = 1.9$$



# Problem of Hebb with Decay

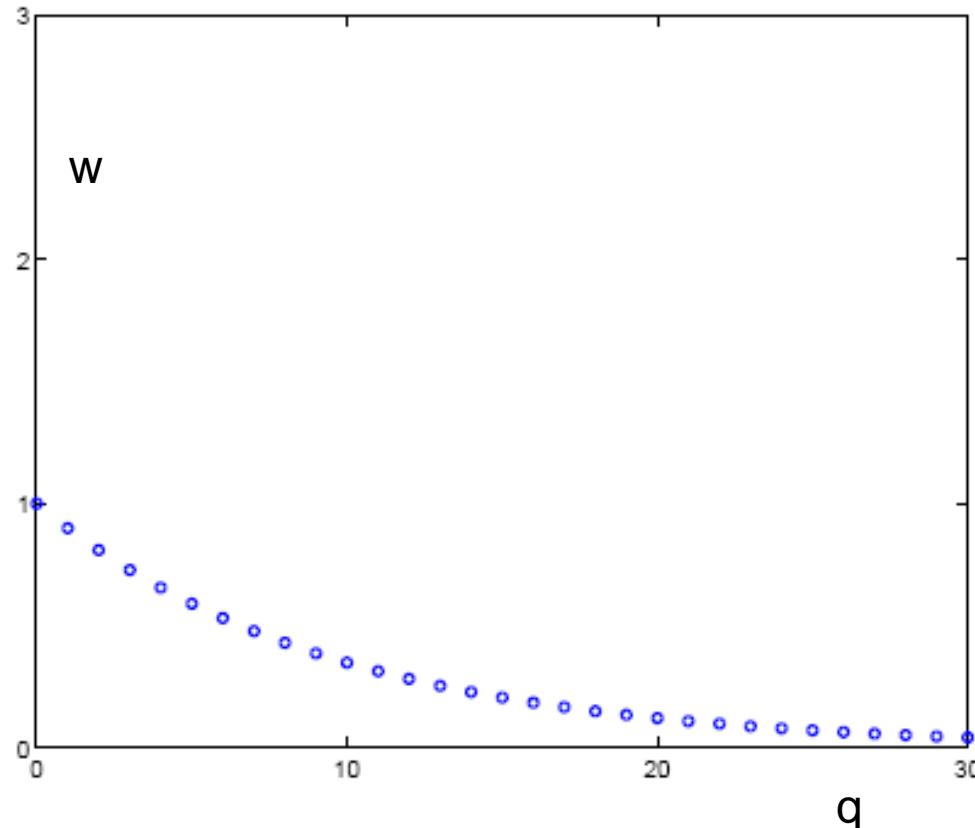
- Associations will decay away if stimuli are not occasionally presented.

If  $a_i = 0$ , then

$$w_{ij}(q) = (1 - \gamma)w_{ij}(q - 1)$$

If  $\gamma = 0.1$ , this becomes

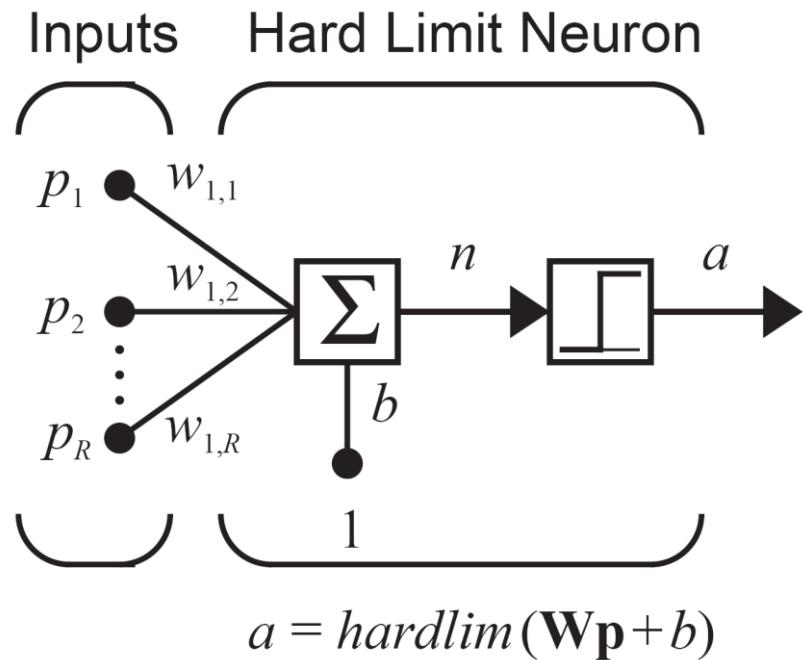
$$w_{ij}(q) = (0.9)w_{ij}(q - 1)$$



Therefore the weight decays by 10% at each iteration where there is no stimulus.

# Instar (Recognition Network)

We considered associations between scalar inputs and outputs. Now we examine a neuron that has a vector input.



Instar is similar with ADALINE, perceptron and linear Associator. <sup>15</sup>

# Instar Operation

$$a = \text{hardlim}(\mathbf{W}\mathbf{p} + b) = \text{hardlim}(\mathbf{w}_1^T \mathbf{p} + b)$$

The instar will be active when

$$\mathbf{w}_1^T \mathbf{p} \geq -b$$

or

$$\mathbf{w}_1^T \mathbf{p} = \|\mathbf{w}_1\| \|\mathbf{p}\| \cos \theta \geq -b$$

For normalized vectors, the largest inner product occurs when the angle between the weight vector and the input vector is zero → the input vector is equal to the weight vector.

The rows of a weight matrix represent patterns to be recognized.

# Vector Recognition

If we set

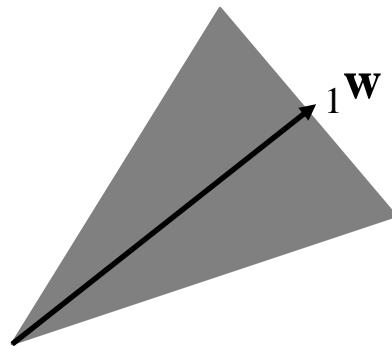
$$b = -\|_1 \mathbf{w} \| \| \mathbf{p} \|$$

the instar will only be active when  $\theta=0$ .

If we set

$$b > -\|_1 \mathbf{w} \| \| \mathbf{p} \|$$

the instar will be active for a range of angles.



As  $b$  is increased, the more patterns there will be (over a wider range of  $\theta$ ) which will activate the instar.

# Instar Rule

Original Hebb rule:  $w_{ij}(q) = w_{ij}(q-1) + \alpha a_i(q)p_j(q)$

With decay:  $w_{ij}(q) = w_{ij}(q-1) + \alpha a_i(q)p_j(q) - \gamma w_{ij}(q-1)$

Modify so that learning and forgetting will only occur when the neuron is active - Instar Rule:

$$w_{ij}(q) = w_{ij}(q-1) + \alpha a_i(q)p_j(q) - \gamma a_i(q)w_{ij}(q-1)$$

or

$$w_{ij}(q) = w_{ij}(q-1) + \alpha a_i(q)(p_j(q) - w_{ij}(q-1))$$

Vector Form:

$$_i\mathbf{w}(q) = _i\mathbf{w}(q-1) + \alpha a_i(q)(\mathbf{p}(q) - _i\mathbf{w}(q-1))$$

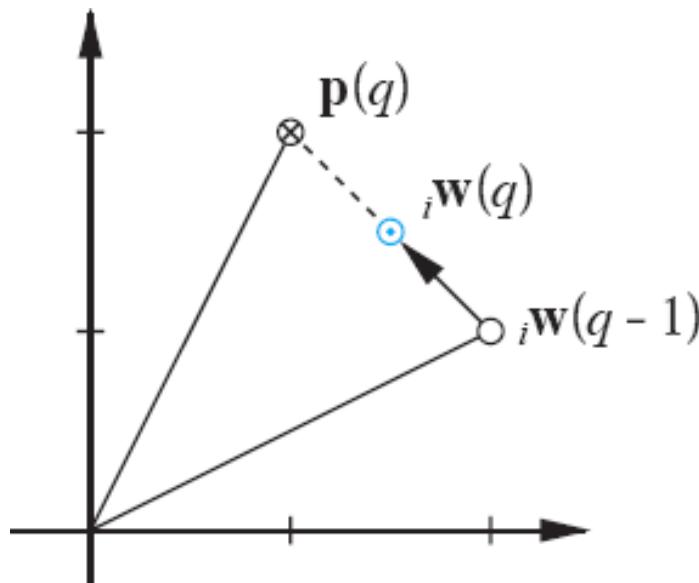
# Graphical Representation

For the case where the instar is active ( $a_i = 1$ ):

$${}_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{w}(q-1))$$

or

$${}_i\mathbf{w}(q) = (1 - \alpha){}_i\mathbf{w}(q-1) + \alpha\mathbf{p}(q)$$

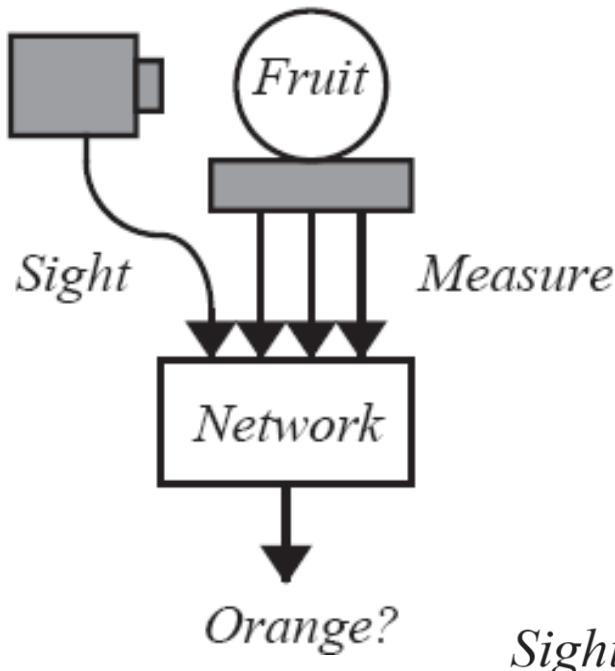


For the case where the instar is inactive ( $a_i = 0$ ):

$${}_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1)$$

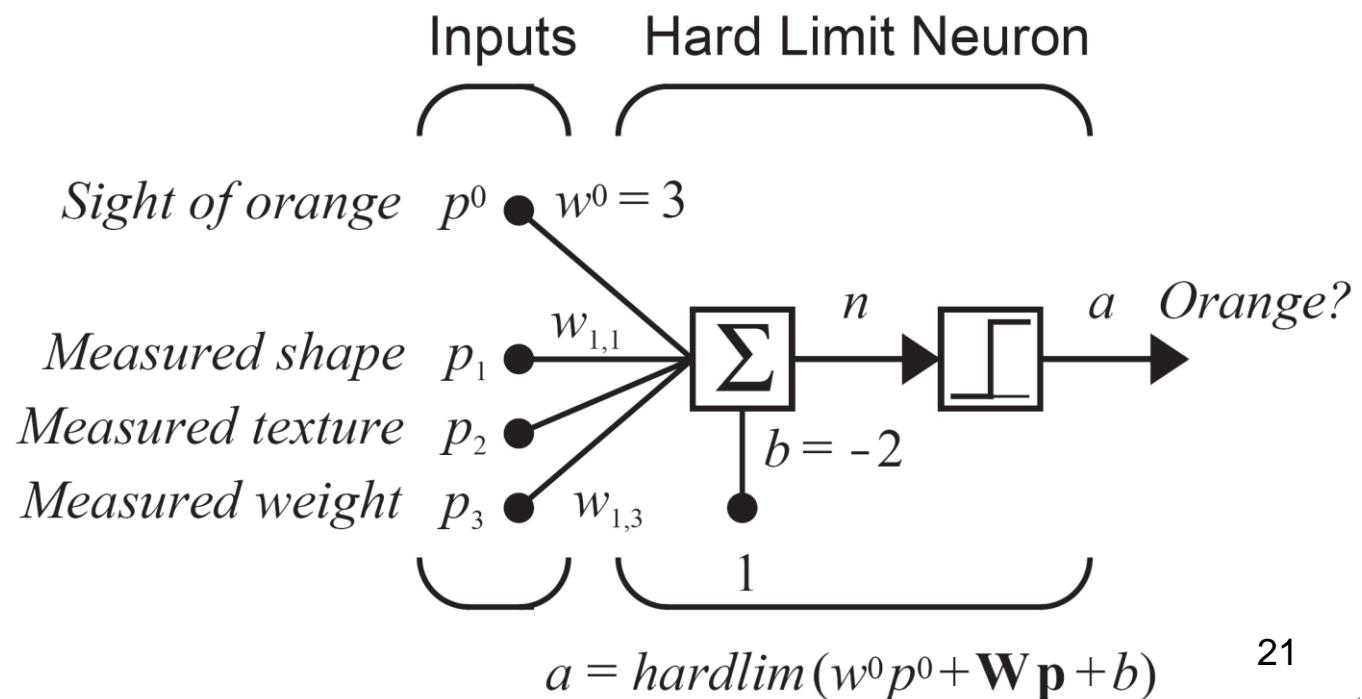
- When the instar is active the weight vector is moved toward the input vector along a line between the old weight vector and the input vector. If  $\alpha=1$  the weight vector is equal to the input vector (max. movement).
- One useful feature of the instar rule is that if the input vectors are normalized, then  $,_i\mathbf{w}$  will also be normalized once it has learned a particular vector  $\mathbf{p}$ .
- This rule not only minimizes forgetting, but results in normalized weight vectors, if the input vectors are normalized.

# Example



$$p^0 = \begin{cases} 1, & \text{orange detected visually} \\ 0, & \text{orange not detected} \end{cases}$$

$$\mathbf{p} = \begin{bmatrix} \text{shape} \\ \text{texture} \\ \text{weight} \end{bmatrix} \quad p_j = \pm 1 \rightarrow \|\mathbf{p}\| = \sqrt{3} \rightarrow b = -2 > -\|\mathbf{p}\|^2$$



# Training

$$\mathbf{W}(0) = {}_1\mathbf{w}^T(0) = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$

The network should not respond to any combination of fruit measurements, so the measurement weights will start with values of 0.

$$\left\{ p^0(1) = 0, \mathbf{p}(1) = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \right\}, \left\{ p^0(2) = 1, \mathbf{p}(2) = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \right\}, \dots$$

Assumption: the visual system only operates correctly on even time steps.

First Iteration ( $\alpha=1$ ):

$$a(1) = \text{hardlim}(w^0 p^0(1) + \mathbf{W}\mathbf{p}(1) - 2)$$

$$a(1) = \text{hardlim}\left(3 \cdot 0 + \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} - 2\right) = 0 \quad (\text{no response})$$

$${}_1\mathbf{w}(1) = {}_1\mathbf{w}(0) + a(1)(\mathbf{p}(1) - {}_1\mathbf{w}(0)) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + 0 \left( \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \right) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

# Further Training

$$a(2) = \text{hardlim}(w^0 p^0(2) + \mathbf{W}\mathbf{p}(2) - 2) = \text{hardlim} \left( 3 \times 1 + \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} - 2 \right) = 1 \quad (\text{orange})$$

$$_1\mathbf{w}(2) = _1\mathbf{w}(1) + a(2)(\mathbf{p}(2) - _1\mathbf{w}(1)) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + 1 \left( \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \right) = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}$$

$$a(3) = \text{hardlim}(w^0 p^0(3) + \mathbf{W}\mathbf{p}(3) - 2) = \text{hardlim} \left( 3 \times 0 + \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} - 2 \right) = 1 \quad (\text{orange})$$

$$_1\mathbf{w}(3) = _1\mathbf{w}(2) + a(3)(\mathbf{p}(3) - _1\mathbf{w}(2)) = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} + 1 \left( \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} - \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \right) = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}$$

nnd15is

Orange will now be detected if either set of sensors works.

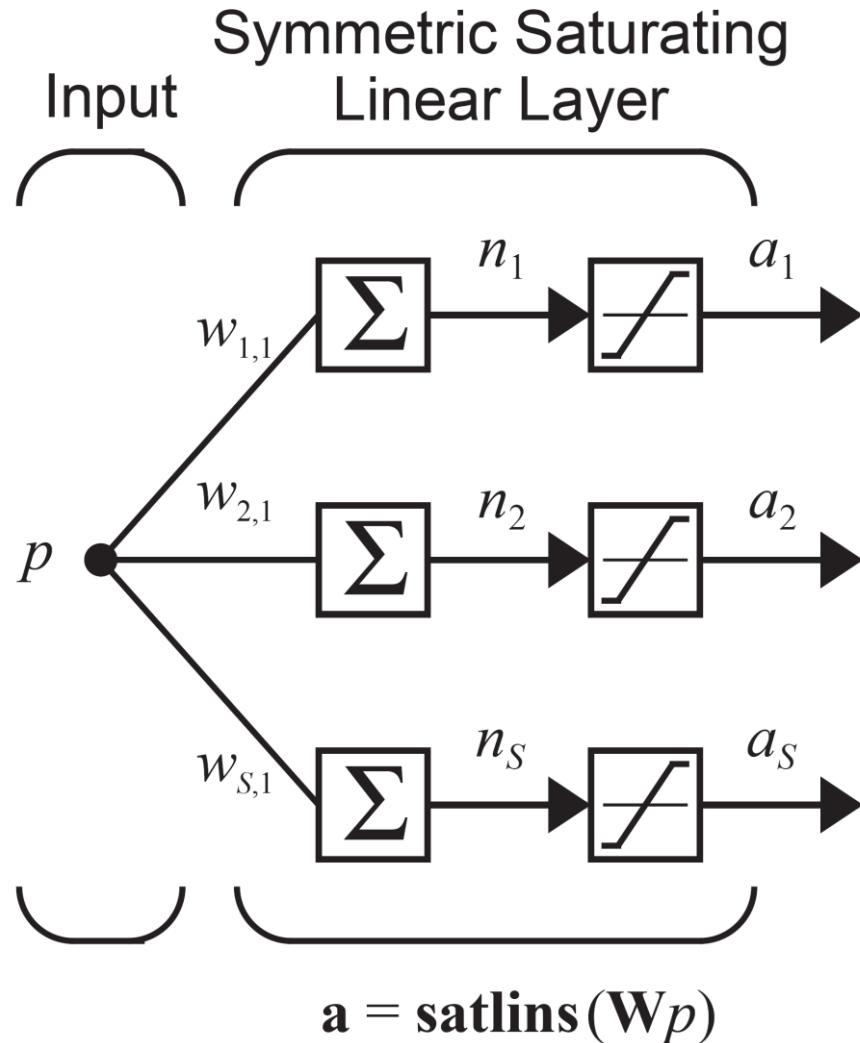
# Kohonen Rule

$$_i \mathbf{w}(q) = _i \mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - _i \mathbf{w}(q-1)), \quad \text{for } i \in X(q)$$

- This is an associative learning rule like instar, so it is suitable for recognition. Unlike the instar rule, learning is not proportional to the neuron's output  $a_i(q)$ .
- Learning occurs when the neuron's index  $i$  is a member of the set  $X(q)$ . We will see in Ch 16 that this can be used to train all neurons in a given neighborhood (e.g. training self-organizing feature map NNs).
- If we define  $X(q)$  as the set of all  $i$  such that  $a_i(q) = 1$ , this rule will be as instar rule.

# Outstar (Recall Network)

- The instar network with a vector input and a scalar output can perform pattern recognition by associating a particular vector stimulus with a response.
- The outstar has a scalar input and a vector output. It can perform pattern recall by associating a stimulus with a vector response.



The symmetric saturating function is chosen because this network is used to recall a vector containing values -1 or +1.

# Outstar Operation

Suppose we want the outstar to recall a certain pattern  $\mathbf{a}^*$  whenever the input  $p=1$  is presented to the network. Let

$$\mathbf{W} = \mathbf{a}^*$$

Then, when  $p=1$

$$\mathbf{a} = \text{\texttt{satlins}}(\mathbf{W}p) = \text{\texttt{satlins}}(\mathbf{a}^* \cdot 1) = \mathbf{a}^*$$

and the pattern is correctly recalled.

The columns of a weight matrix represent patterns to be recalled.

# Outstar Rule

For the instar rule we made the weight decay term of the Hebb rule proportional to the output of the network. For the outstar rule we make the weight decay term proportional to the input of the network.

$$w_{ij}(q) = w_{ij}(q-1) + \alpha a_i(q)p_j(q) - \gamma p_j(q)w_{ij}(q-1)$$

If we make the decay rate  $\gamma$  equal to the learning rate  $\alpha$ ,

$$w_{ij}(q) = w_{ij}(q-1) + \alpha(a_i(q) - w_{ij}(q-1))p_j(q)$$

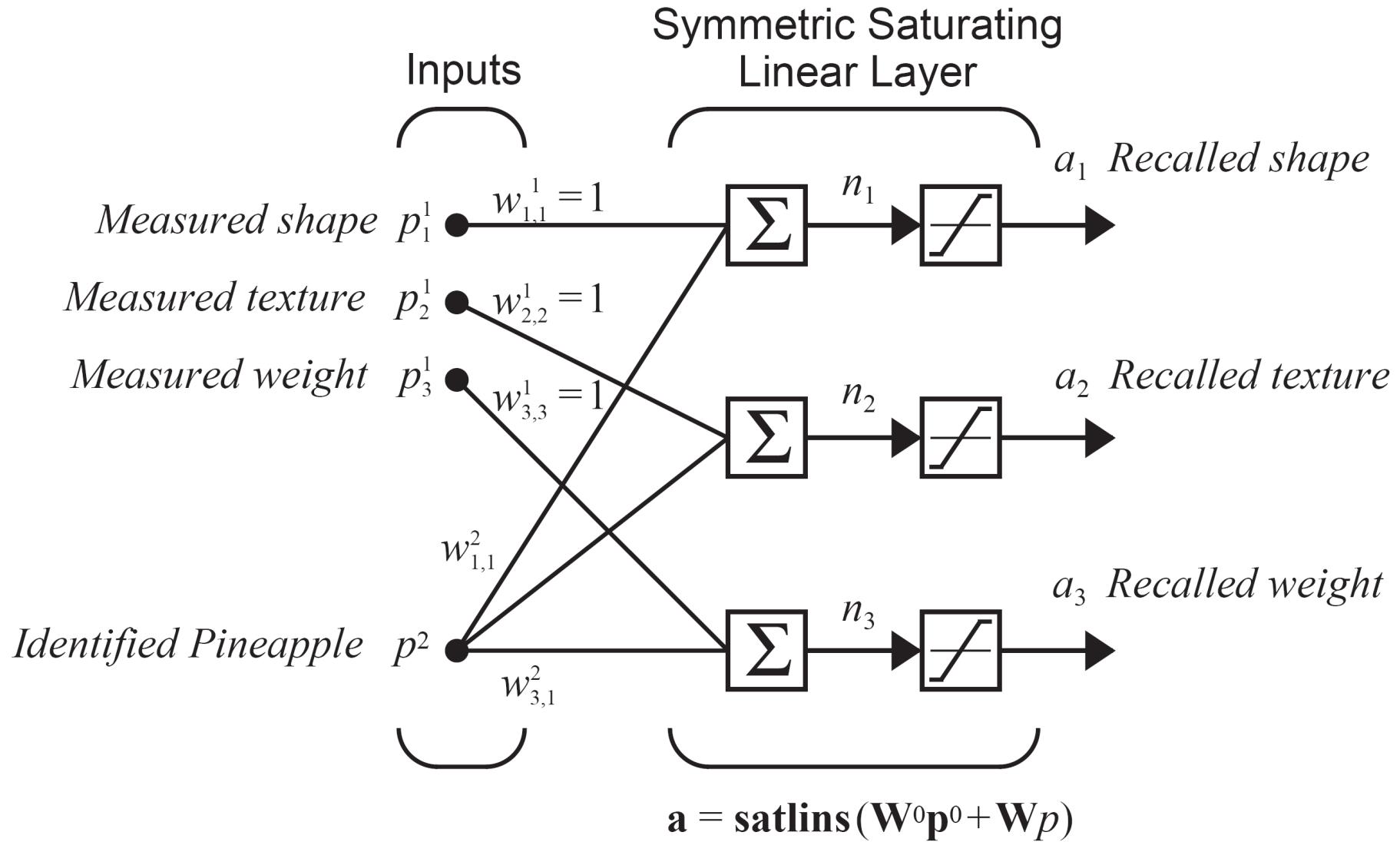
The outstar rule has properties complementary to the instar rule. Learning occurs whenever  $p_j$  is nonzero (instead of  $a_i$ ). When learning occurs, column  $\mathbf{w}_j$  moves toward the output vector.

Vector Form:

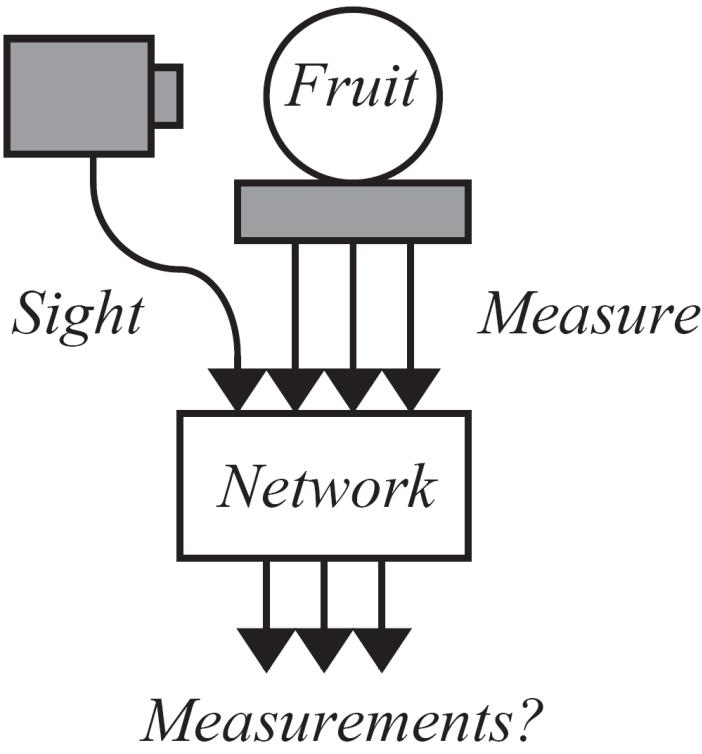
$$\mathbf{w}_j(q) = \mathbf{w}_j(q-1) + \alpha(\mathbf{a}(q) - \mathbf{w}_j(q-1))p_j(q)$$

where  $\mathbf{w}_j$  is the  $j$ th column of the matrix  $\mathbf{W}$ .

# Example - Pineapple Recall



# Definitions



$$\mathbf{a} = \text{satlins}(\mathbf{W}^0 \mathbf{p}^0 + \mathbf{W}p)$$

$$\mathbf{W}^0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{p}^0 = \begin{bmatrix} shape \\ texture \\ weight \end{bmatrix}$$

$$\mathbf{p}^{pineapple} = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix}$$

$$p = \begin{cases} 1, & \text{if a pineapple can be seen} \\ 0, & \text{otherwise} \end{cases}$$

- The weight matrix for unconditioned stimulus  $\mathbf{W}^0$ , is set to the identity matrix, so that any set of measurements  $\mathbf{p}^0$  (with  $\pm 1$  values) will be copied to the output  $\mathbf{a}$ .
- The weight matrix for the conditioned stimulus,  $\mathbf{W}$ , is set to zero initially, so that a 1 on  $p$  will not generate a response.
- $\mathbf{W}$  will be updated with the outstar rule using  $\alpha=1$ :

$$\mathbf{w}_j(q) = \mathbf{w}_j(q-1) + (\mathbf{a}(q) - \mathbf{w}_j(q-1)) p_j(q)$$

# Iteration 1

$$\left\{ \mathbf{p}^0(1) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, p(1) = 1 \right\}, \left\{ \mathbf{p}^0(2) = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix}, p(2) = 1 \right\}, \dots \text{ Assumption: measured values are available only on even iterations.}$$

$$\alpha = 1$$

$$\mathbf{a}(1) = \mathbf{satline}\left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} 1\right) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad (\text{no response})$$

$$\mathbf{w}_1(1) = \mathbf{w}_1(0) + (\mathbf{a}(1) - \mathbf{w}_1(0)) p(1) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}\right) 1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

# Convergence

$$\mathbf{a}(2) = \text{satlins} \left( \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} 1 \right) = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix} \quad (\text{measurements given})$$

$$\mathbf{w}_1(2) = \mathbf{w}_1(1) + (\mathbf{a}(2) - \mathbf{w}_1(1))p(2) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \left( \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \right) 1 = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix}$$

$$\mathbf{a}(3) = \text{satlins} \left( \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix} 1 \right) = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix} \quad (\text{measurements recalled})$$

- The network is now able to recall the measurements of the pineapple when it sees it, even though the measurements system fails.
- From now on, the weights will no longer change values unless a pineapple is seen with different measurements.

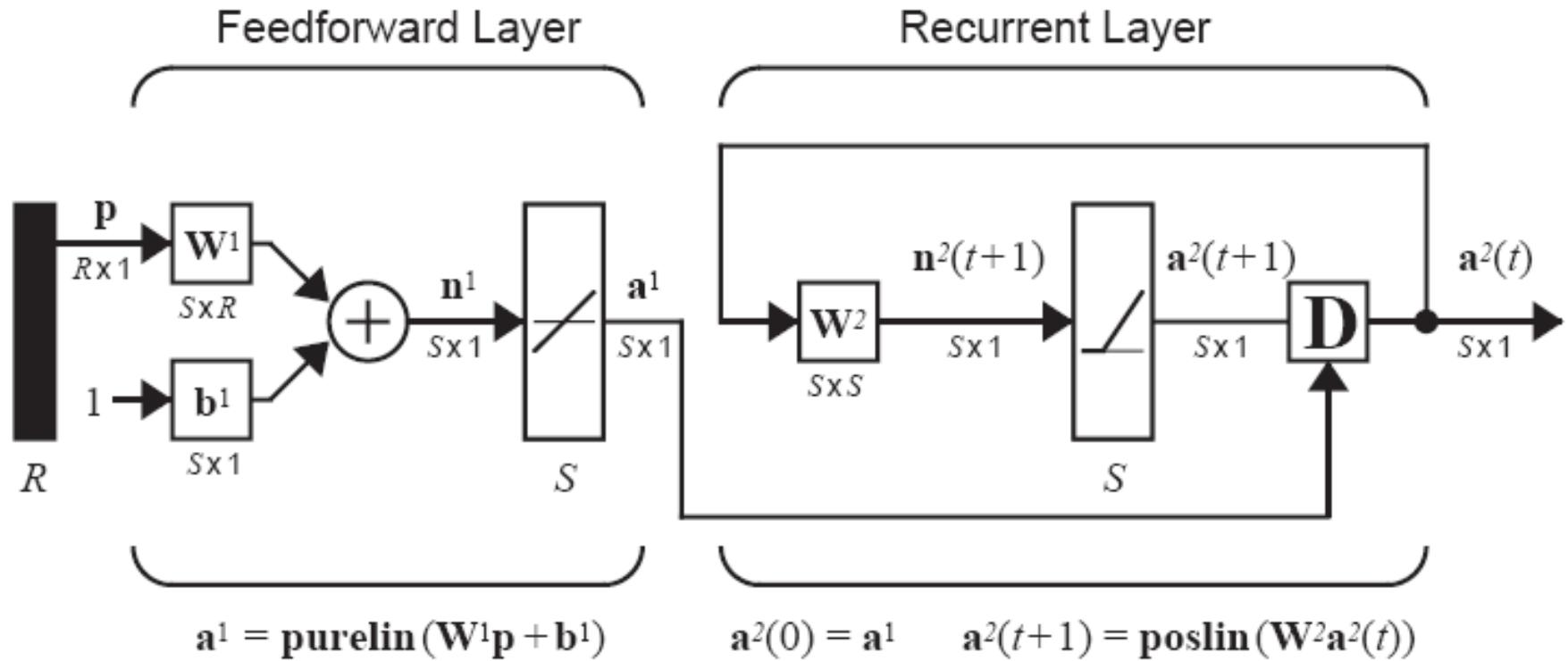
$$\mathbf{w}_1(3) = \mathbf{w}_1(2) + (\mathbf{a}(2) - \mathbf{w}_1(2))p(2) = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix} + \left( \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix} - \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix} \right) \mathbf{1} = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix}$$

nnd15os

# Competitive Networks

- We discuss networks that are very similar in structure and operation to the Hamming network.
- In Hamming Network the neurons in the output layer compete with each other to determine a winner.
- The winner indicates which prototype pattern is most representative of the input pattern.
- Unlike the Hamming network they use the associative learning rules of Ch 15 to adaptively learn to classify patterns.
- The competitive network, the feature map and the learning vector quantization (LVQ) networks are discussed.

# Hamming Network



- The 1<sup>st</sup> layer which is a layer of instars, performs a correlation between the input vector and the prototype vectors.
- The 2<sup>nd</sup> layer performs a competition to determine which of the prototype vectors is closest to the input vector.

# Layer 1 (Correlation)

We want the network to recognize the following prototype vectors:

$$\{\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_Q\}$$

- A single instar is able to recognize only one pattern. In order to allow multiple patterns to be classified we need to have multiple instars.

The first layer weight matrix and bias vector are given by:

$$\mathbf{W}^1 = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \\ \vdots \\ {}_S\mathbf{w}^T \end{bmatrix} = \begin{bmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \\ \vdots \\ \mathbf{p}_Q^T \end{bmatrix} \quad \mathbf{b}^1 = \begin{bmatrix} R \\ R \\ \vdots \\ R \end{bmatrix}$$

- Each row of  $\mathbf{W}^1$  represents a prototype vector which we want to recognize, and each elements of  $\mathbf{b}^1$  is set equal to the # of elements in each input vector ( $R$ ).
- $S$  (number of neurons) is equal to  $Q$ .

The response of the first layer is:

$$\mathbf{a}^1 = \mathbf{W}^1 \mathbf{p} + \mathbf{b}^1 = \begin{bmatrix} \mathbf{p}_1^T \mathbf{p} + R \\ \mathbf{p}_2^T \mathbf{p} + R \\ \vdots \\ \mathbf{p}_Q^T \mathbf{p} + R \end{bmatrix}$$

*Hamming Distance (HD)*

- These inner products indicate how close each of the prototype patterns is close to the input vector.
- The prototype closest to the input vector produces the largest response.

# Layer 2 (Competition)

- In the layer 2 we want to decide which prototype is closest to the input, so we used competition layer instead of *hardlim* transfer function.
- The output of the first layer indicate the correlation between the prototype patterns and the input vector.

$$\mathbf{a}^2(0) = \mathbf{a}^1$$

*The second layer is initialized with the output of the first layer.*

- The 2<sup>nd</sup> layer output is updated according to the following recurrent relation:

$$\mathbf{a}^2(t+1) = \mathbf{poslin}(\mathbf{W}^2 \mathbf{a}^2(t))$$

$$w_{ij}^2 = \begin{cases} 1, & \text{if } i = j \\ -\varepsilon, & \text{otherwise} \end{cases} \quad 0 < \varepsilon < \frac{1}{S-1}$$

- This matrix produces lateral inhibition, in which the output of each neuron has an inhibitory effect on all of the other neurons.

$$a_i^2(t+1) = poslin\left(a_i^2(t) - \varepsilon \sum_{j \neq i} a_j^2(t)\right)$$

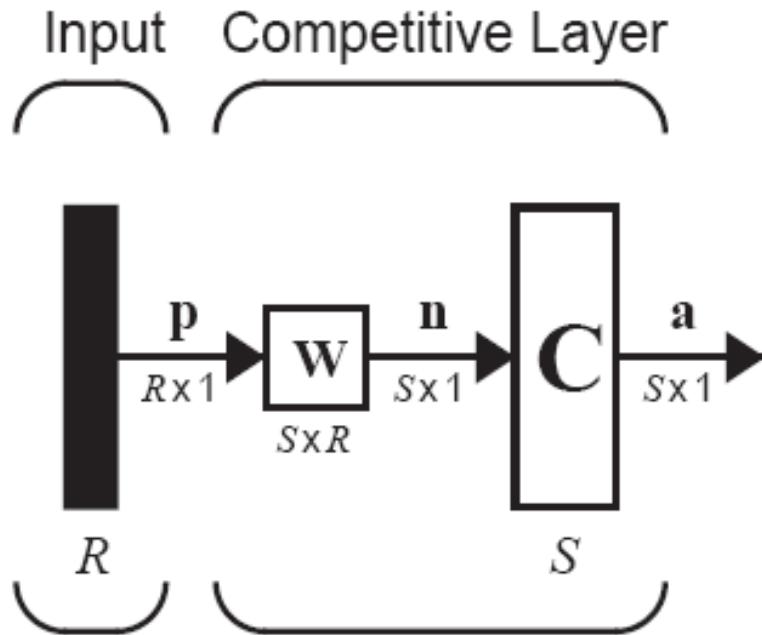
*The neuron with the largest initial condition will win the competition.*

- The output of the neuron with the largest initial condition will decrease more slowly than the outputs of the other neurons.

- The index of the 2<sup>nd</sup> layer neuron with a stable positive output is the index of the prototype vector that best matched the input.
- This is called a winner-take-all competition, since only one neuron will have a nonzero output.
- Hamming net is a pure classifier not an autoassociative memory.

nnd3hamc

# Competitive Layer



$$\mathbf{a} = \text{compet}(\mathbf{W}\mathbf{p})$$

$$\mathbf{n} = \mathbf{W}\mathbf{p} = \begin{bmatrix} 1\mathbf{w}^T \\ 2\mathbf{w}^T \\ \vdots \\ S\mathbf{w}^T \end{bmatrix} \mathbf{p} = \begin{bmatrix} 1\mathbf{w}^T \mathbf{p} \\ 2\mathbf{w}^T \mathbf{p} \\ \vdots \\ S\mathbf{w}^T \mathbf{p} \end{bmatrix} = \begin{bmatrix} L^2 \cos \theta_1 \\ L^2 \cos \theta_2 \\ \vdots \\ L^2 \cos \theta_S \end{bmatrix}$$

Assuming vectors have normalized lengths of L

$$\mathbf{a} = \text{compet}(\mathbf{n})$$

$$a_i = \begin{cases} 1, & i = i^* \\ 0, & i \neq i^* \end{cases}$$

$$n_{i^*} \geq n_i, \forall i \text{ and } i^* \leq i, \forall n_i = n_{i^*}$$

# Competitive Learning

- We can now design a competitive network classifier by setting the rows of  $\mathbf{W}$  to the desired prototype vectors.
- We would like to have a learning rule that could be used to train the weights in a competitive network, without knowing the prototype vectors.
- Clustering of input data  $\rightarrow$  grouping of similar objects and separating of dissimilar ones.

## Instar Rule

$$_i\mathbf{w}(q) = _i\mathbf{w}(q-1) + \alpha a_i(q)(\mathbf{p}(q) - _i\mathbf{w}(q-1))$$

- For the competitive network, the winning neuron has an output of 1, and the other neurons have an output of 0.
- For the competitive network,  $\mathbf{a}$  is only nonzero for the winning neuron ( $i=i^*$ ). So, we can get the same results using the Kohonen rule.

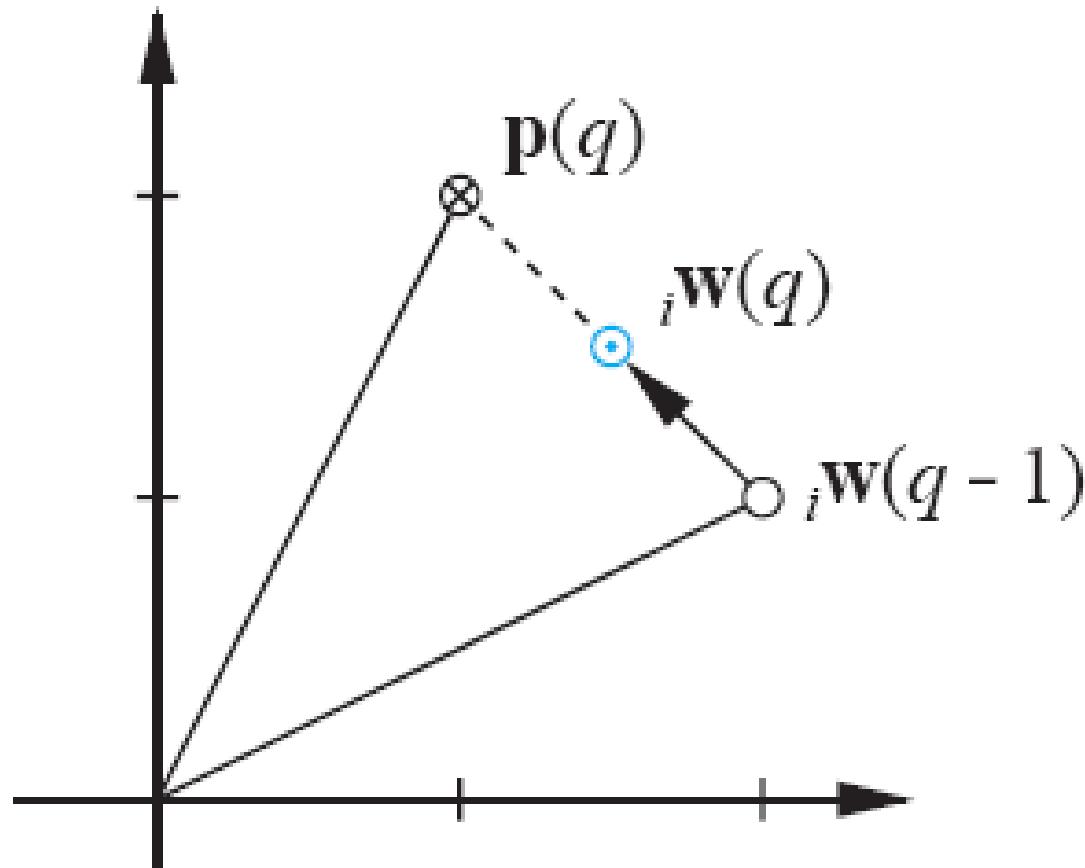
### Kohonen Rule

$${}_{i^*}\mathbf{w}(q) = {}_{i^*}\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{w}(q-1))$$

$${}_{i^*}\mathbf{w}(q) = (1 - \alpha) {}_{i^*}\mathbf{w}(q-1) + \alpha \mathbf{p}(q)$$

$${}_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) \quad i \neq i^*$$

# Graphical Representation



$${}_i^*\mathbf{w}(q) = {}_i^*\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i^*\mathbf{w}(q-1))$$

$${}_i^*\mathbf{w}(q) = (1 - \alpha){}_i^*\mathbf{w}(q-1) + \alpha\mathbf{p}(q)$$

Exp:

Let's use the six vectors in Figure 14.4 to demonstrate how a competitive layer learns to classify vectors.

Here are the six vectors:

$$\mathbf{p}_1 = \begin{bmatrix} -0.1961 \\ 0.9806 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} 0.1961 \\ 0.9806 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} 0.9806 \\ 0.1961 \end{bmatrix}$$
$$\mathbf{p}_4 = \begin{bmatrix} 0.9806 \\ -0.1961 \end{bmatrix}, \mathbf{p}_5 = \begin{bmatrix} -0.5812 \\ -0.8137 \end{bmatrix}, \mathbf{p}_6 = \begin{bmatrix} -0.8137 \\ -0.5812 \end{bmatrix}$$

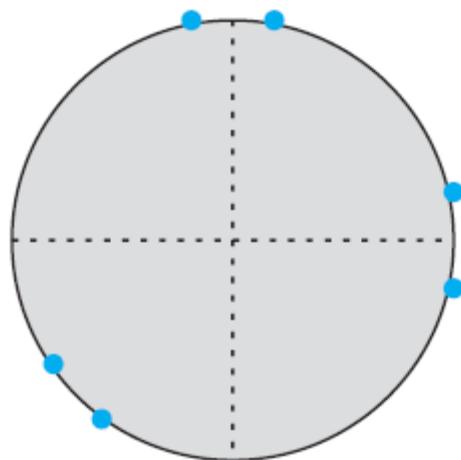
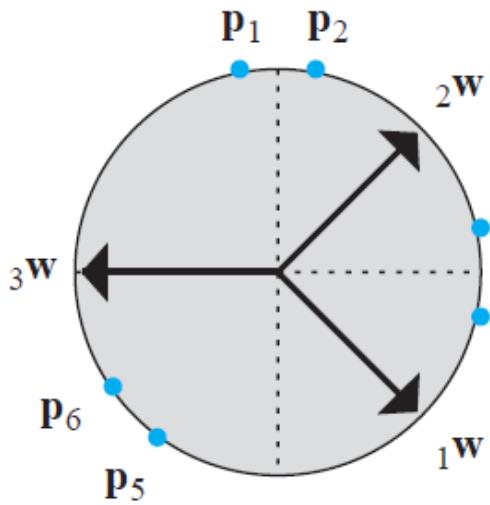


Figure 16.4 Sample Input Vectors

Our competitive network will have three neurons, and therefore it can classify vectors into three classes. Here are the "randomly" chosen normalized initial weights:

$${}_1\mathbf{w} = \begin{bmatrix} 0.7071 \\ -0.7071 \end{bmatrix}, {}_2\mathbf{w} = \begin{bmatrix} 0.7071 \\ 0.7071 \end{bmatrix}, {}_3\mathbf{w} = \begin{bmatrix} -1.0000 \\ 0.0000 \end{bmatrix}, \mathbf{W} = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \\ {}_3\mathbf{w}^T \end{bmatrix}$$



The data vectors are shown, with the weight vectors displayed as arrows.  
Let's present the vector  $p_2$  to the network:

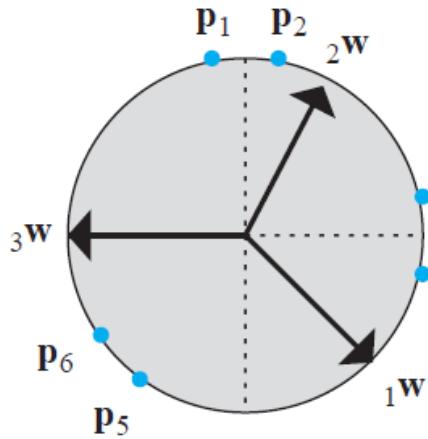
$$\begin{aligned}
 \mathbf{a} &= \text{compet}(\mathbf{W}\mathbf{p}_2) = \text{compet} \left( \begin{bmatrix} 0.7071 & -0.7071 \\ 0.7071 & 0.7071 \\ -1.0000 & 0.0000 \end{bmatrix} \begin{bmatrix} 0.1961 \\ 0.9806 \end{bmatrix} \right) \\
 &= \text{compet} \left( \begin{bmatrix} -0.5547 \\ 0.8321 \\ -0.1961 \end{bmatrix} \right) = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}.
 \end{aligned}$$

The second neuron's weight vector was closest to  $\mathbf{p}_2$  so it won the competition (  $i^* = 2$  ) and output a 1.

We now apply the Kohonen learning rule to the winning neuron with a learning rate of  $\alpha = 0.5$ .

$${}_2\mathbf{w}^{new} = {}_2\mathbf{w}^{old} + \alpha(\mathbf{p}_2 - {}_2\mathbf{w}^{old})$$

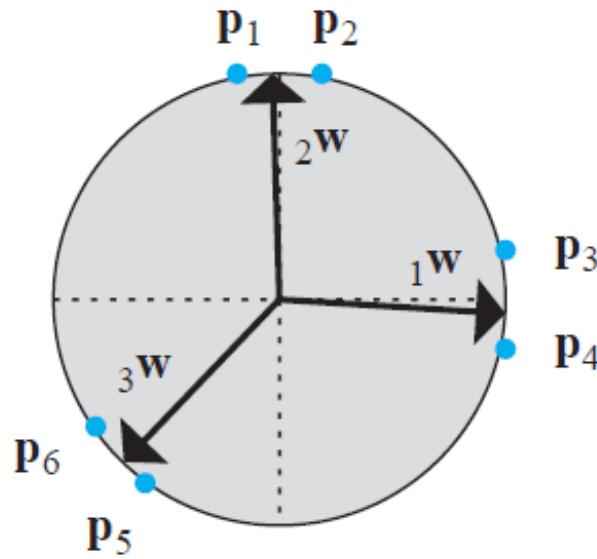
$$= \begin{bmatrix} 0.7071 \\ 0.7071 \end{bmatrix} + 0.5 \left( \begin{bmatrix} 0.1961 \\ 0.9806 \end{bmatrix} - \begin{bmatrix} 0.7071 \\ 0.7071 \end{bmatrix} \right) = \begin{bmatrix} 0.4516 \\ 0.8438 \end{bmatrix}$$



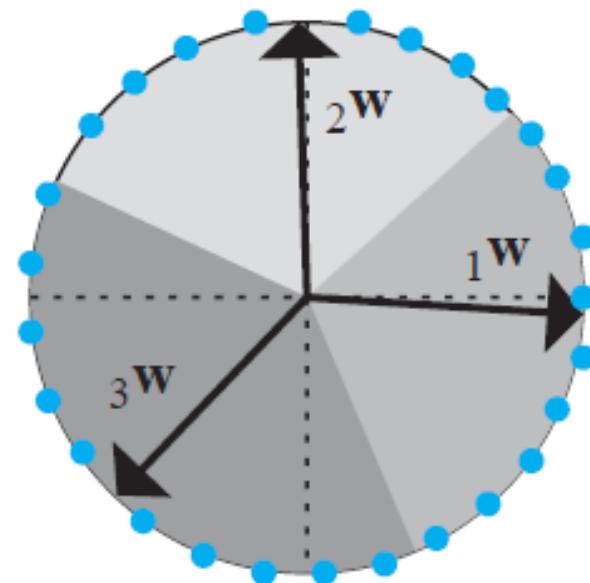
The Kohonen rule moves  ${}_2\mathbf{w}$  closer to  $\mathbf{p}_2$  as can be seen in the diagram. If we continue choosing input vectors at random and presenting them to the network, then at each iteration the weight vector closest to the input vector will move toward that vector.

Eventually, each weight vector will point at a different cluster of input vectors. Each weight vector becomes a prototype for a different cluster.

Once the network has learned to cluster the input vectors, it will classify new vectors accordingly. The diagram uses shading to show which region each neuron will respond to. The competitive layer assigns each input vector  $p$  to one of these classes by producing an output of 1 for the neuron whose weight vector is closest to  $p$ .

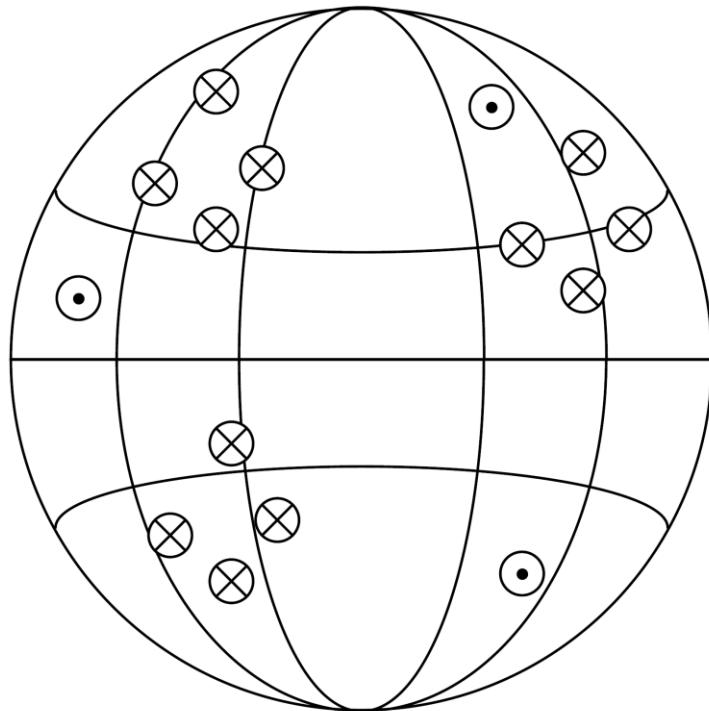


nnd14cl

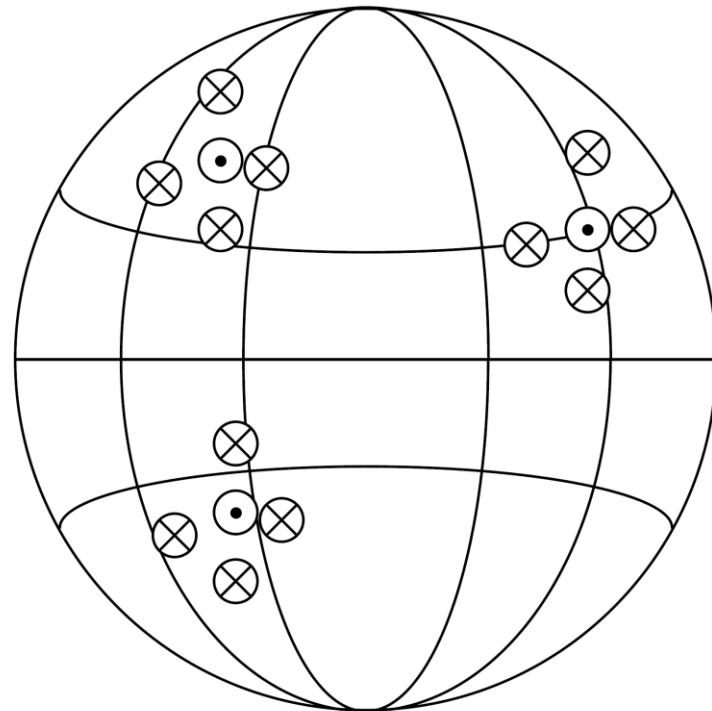


# Typical Convergence (Clustering)

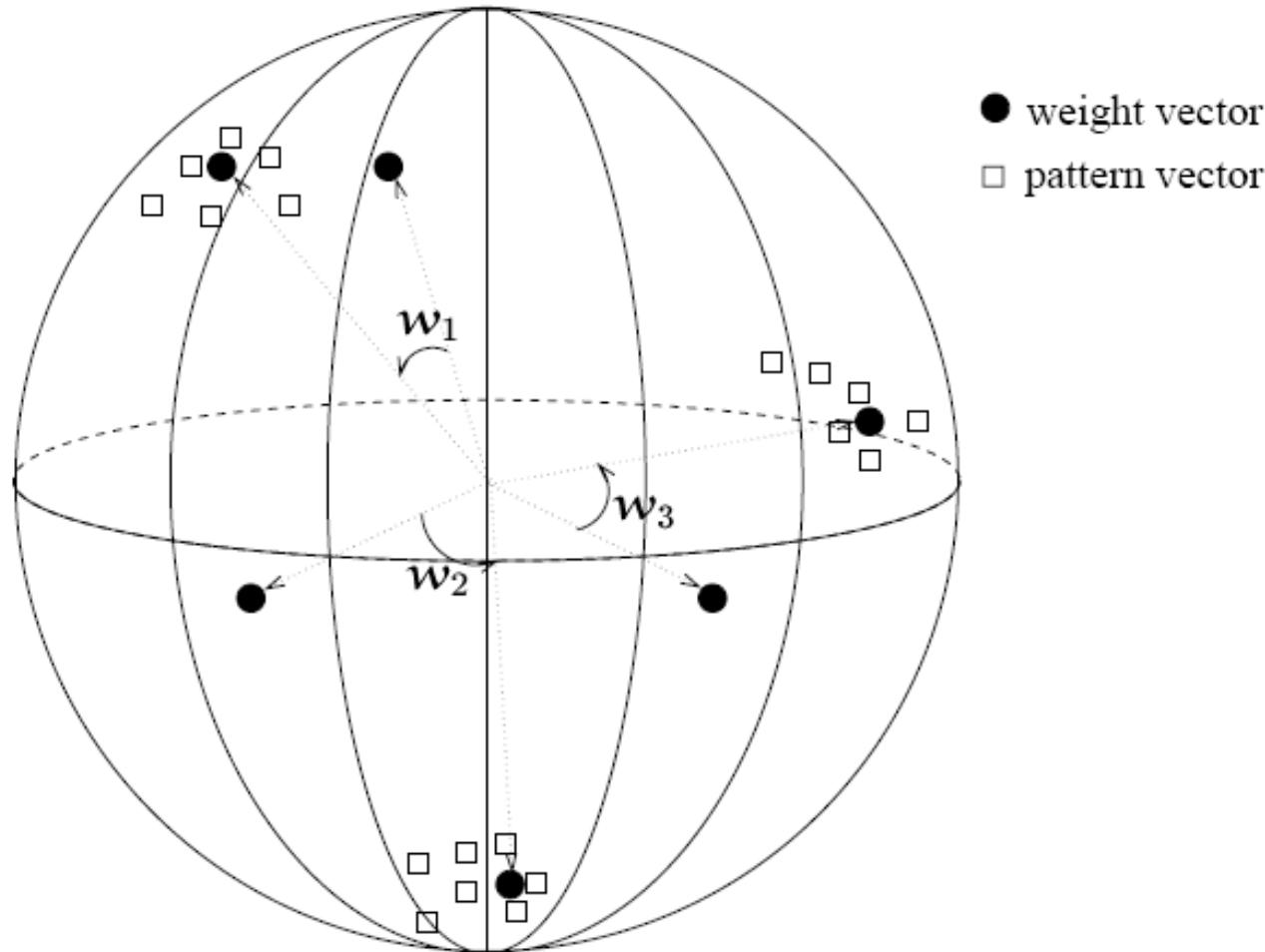
● Weights  
⊗ Input Vectors



Before Training

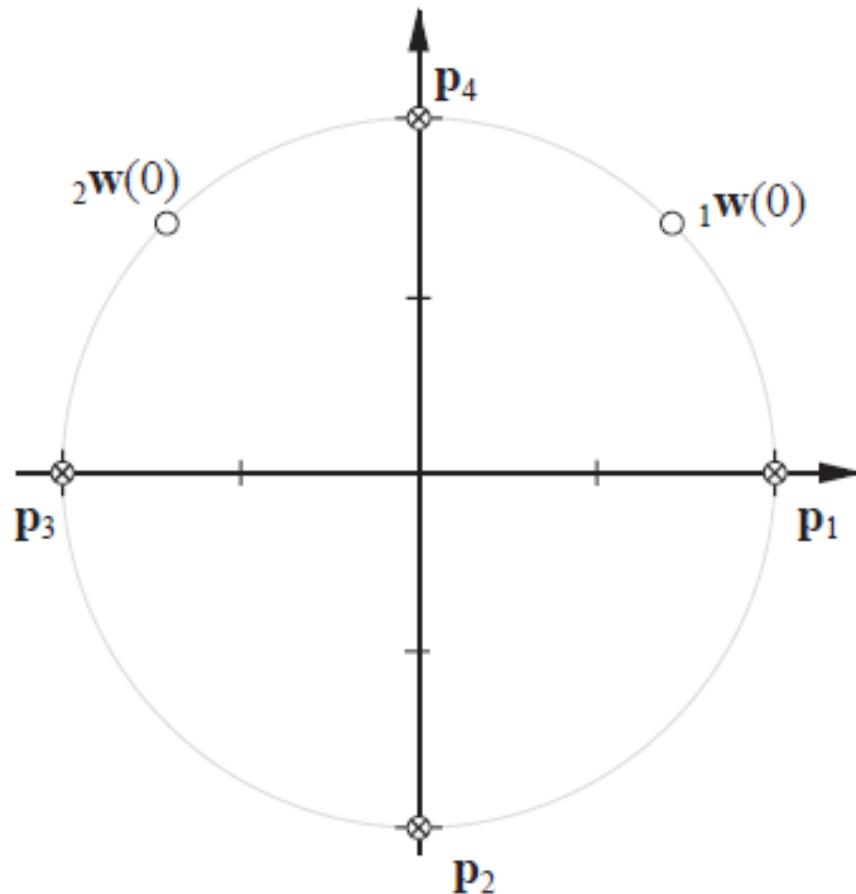


After Training

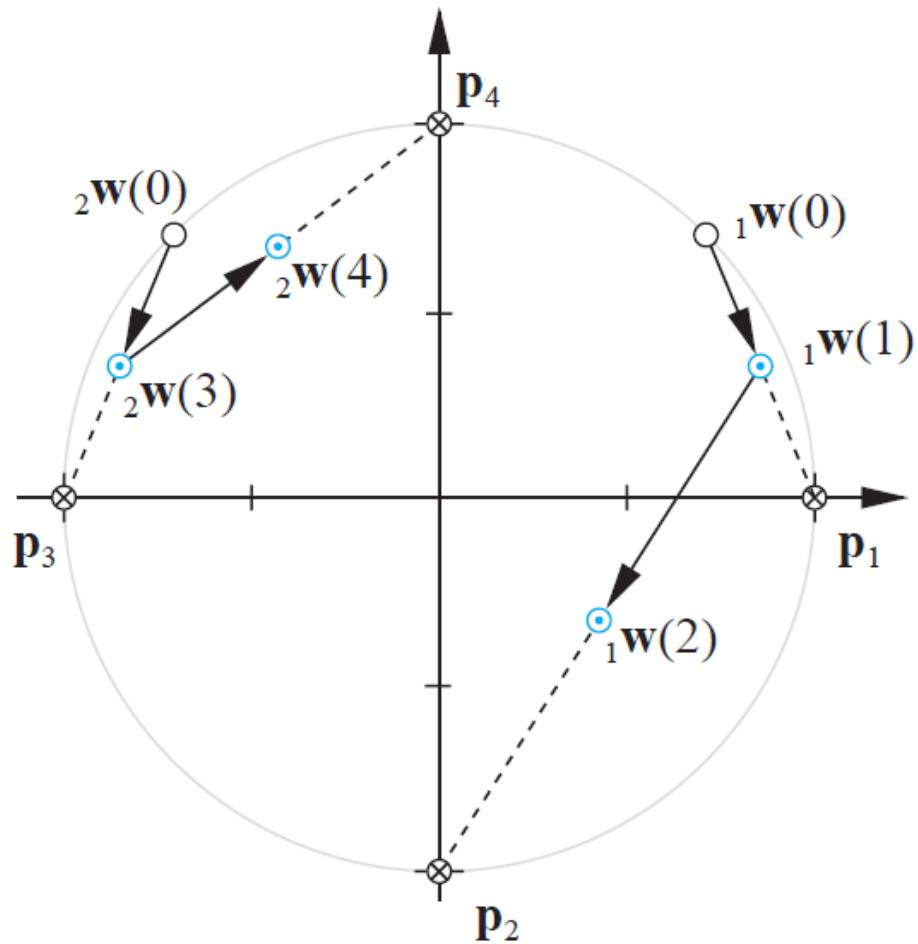


Example of clustering in 3D with normalized vectors, which all lie on the unity sphere. The three weight vectors are rotated towards the centers of gravity of the three different input clusters.

# Example

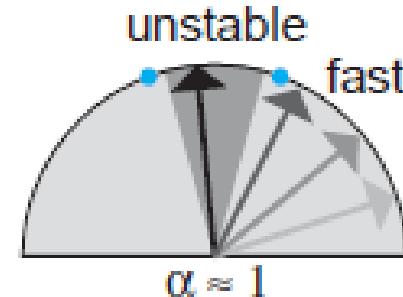
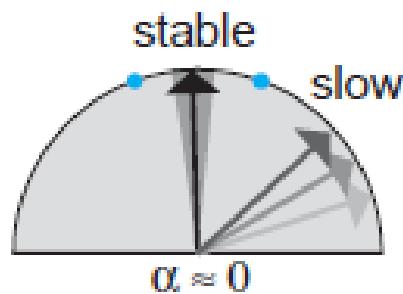


# Four Iterations



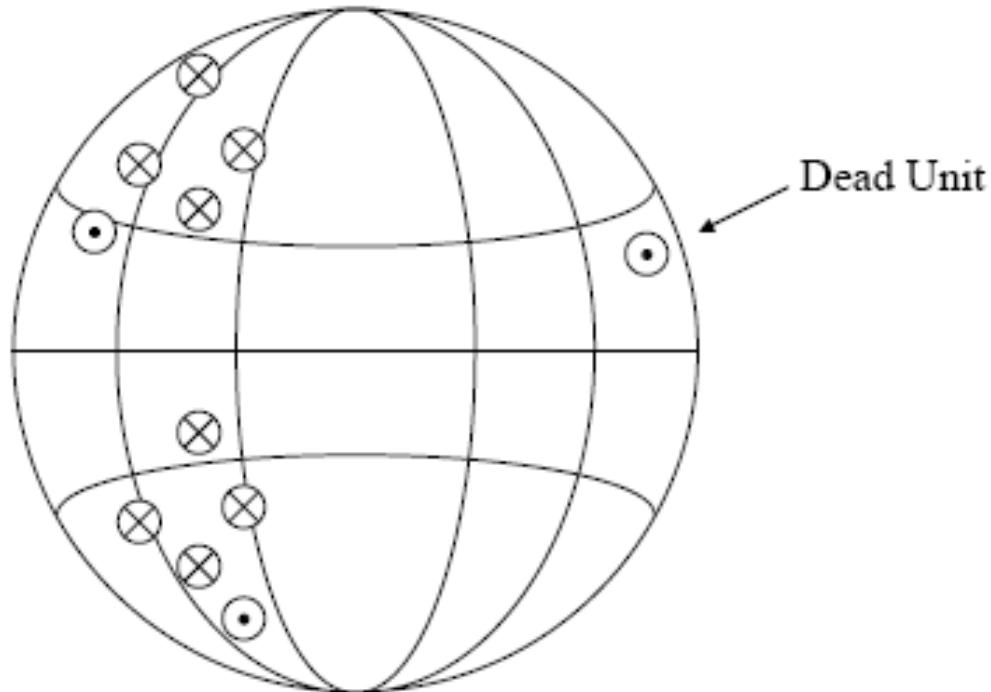
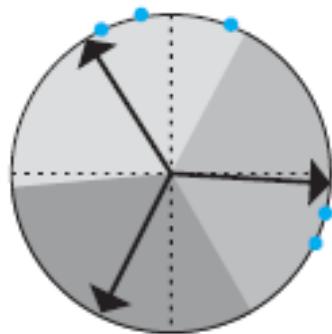
# Stability

- The choice of learning rate forces a trade-off between the speed of learning and the stability of the final weight vectors. A learning rate near zero results in slow learning. However, once a weight vector reaches the center of a cluster it will tend to stay close to the center.



# Dead Units

One problem with competitive learning is that neurons with initial weights far from any input vector may never win.



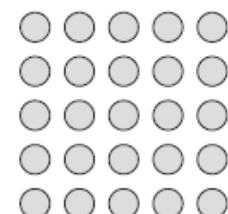
**Solution:** Add a negative bias to each neuron, and increase the magnitude of the bias as the neuron wins. This will make it harder to win if a neuron has won often. This is called a “**conscience.**” 23

# Other problems

- A competitive layer always has as many classes as it has neurons. This may not be acceptable for some applications, especially when the number of clusters is not known in advance.
- For competitive layers, each class consists of a convex region of the input space. Competitive layers cannot form classes with nonconvex regions or classes that are union of unconnected regions.
- Some of these problems can be solved by SOM and LVQ networks

# Competitive Layers in Biology

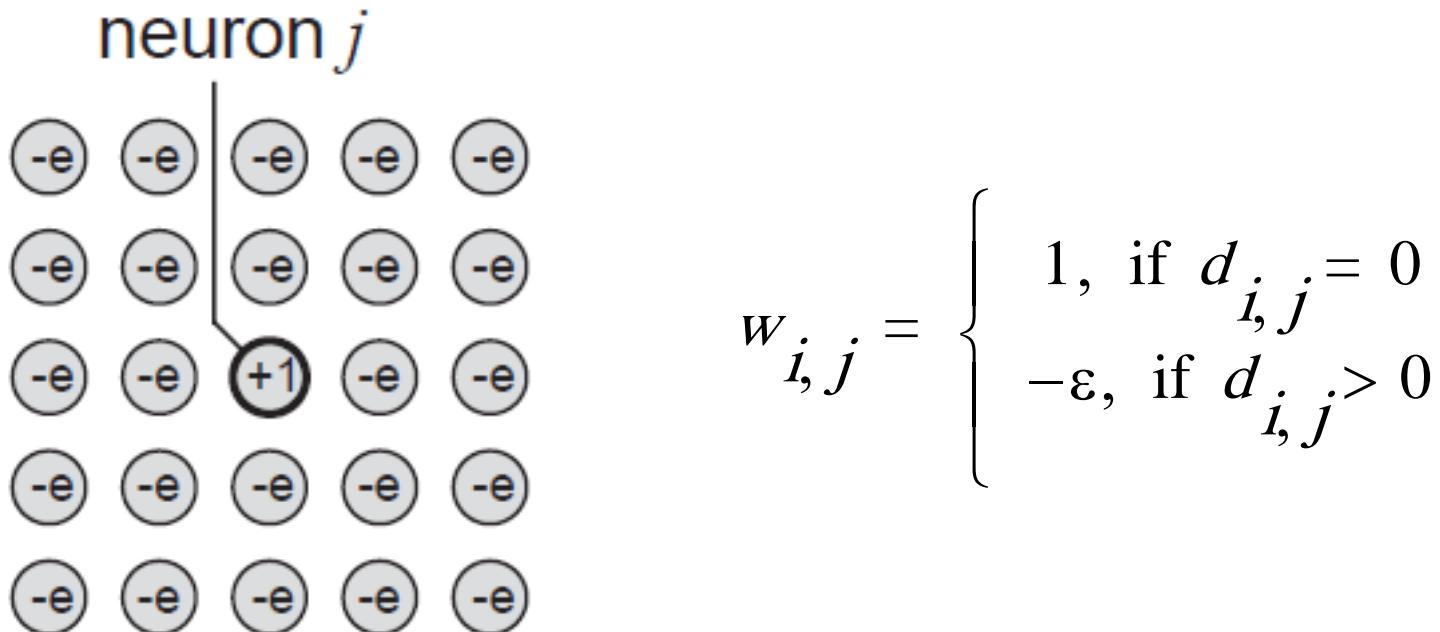
- How neurons are physically organized within a layer (the topology of the network)?
- In biological neural networks, Often weights vary as a function of the distance between the neurons they connect.



Weights in the competitive layer of the Hamming network:

$$w_{i,j} = \begin{cases} 1, & \text{if } i = j \\ -\varepsilon, & \text{if } i \neq j \end{cases}$$

Weights assigned based on distance:

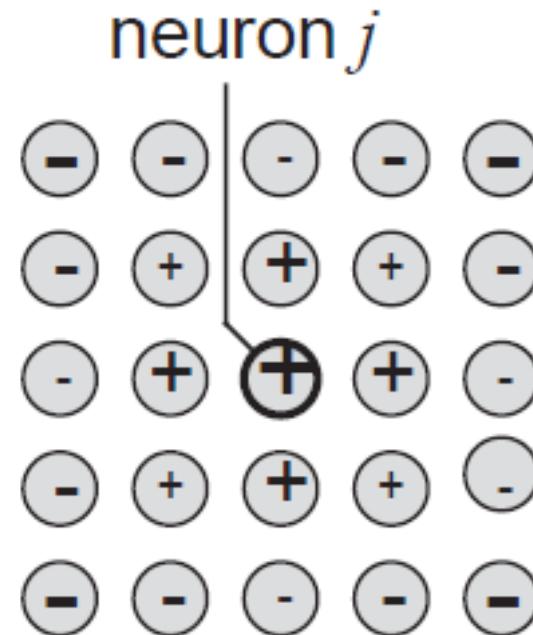
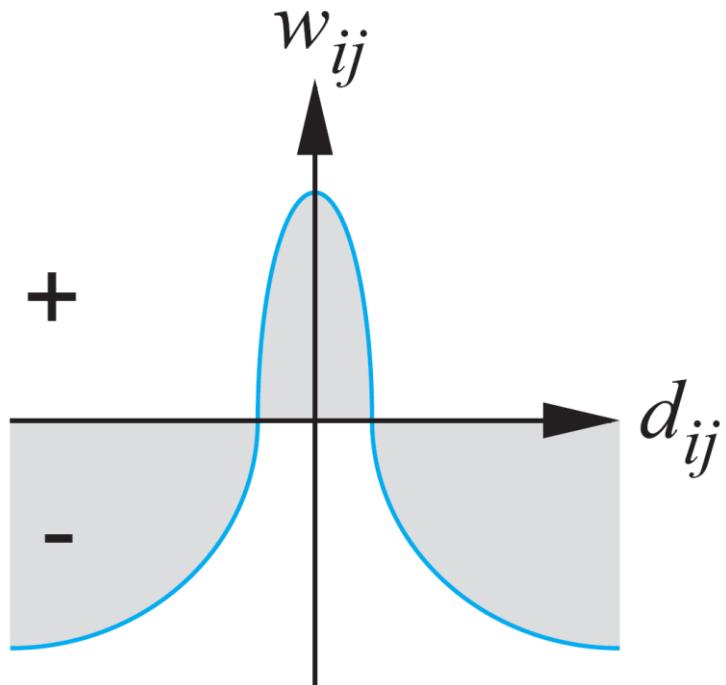


## On-Center/Off-Surround Connections for Competition

- Each neuron reinforces itself (center), while inhibiting all other neurons (surround).

- In biology a neuron reinforces not only itself, but also those neurons close to it.
- Typically, the transition from reinforcement to inhibition occurs smoothly as the distance between neurons increases.

# Mexican-Hat Function



# Feature Maps

- In order to emulate the activity bubbles of biological systems, Kohonen designed the following simplification.
- His self-organizing feature map (SOFM) network first determines the winning neuron  $i^*$  using the same procedure as the competitive layer. Next, the weight vectors for all neurons within a certain neighborhood of the winning neuron are updated using the Kohonen rule.

Update weight vectors in a neighborhood of the winning neuron.

$$\begin{aligned}_i \mathbf{w}(q) &= {}_i \mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i \mathbf{w}(q-1)) \\ {}_i \mathbf{w}(q) &= (1 - \alpha) {}_i \mathbf{w}(q-1) + \alpha \mathbf{p}(q) \quad i \in N_{i^*}(d)\end{aligned}$$

Where the neighborhood  $N_{i^*}(d)$  contains the indices for all of the neurons that lie within a radius of the winning neuron  $i^*$ :

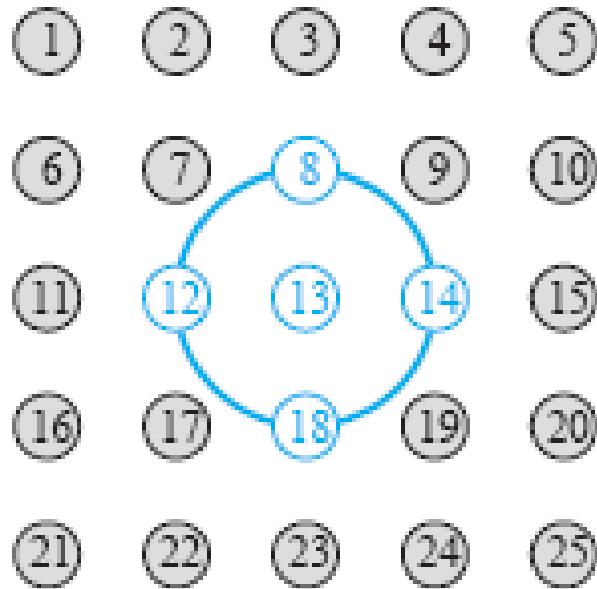
$$N_i(d) = \{j, d_{i,j} \leq d\}$$

When a vector  $\mathbf{p}$  is presented, the weights of the winning neuron and its neighbors will move toward  $\mathbf{p}$ .

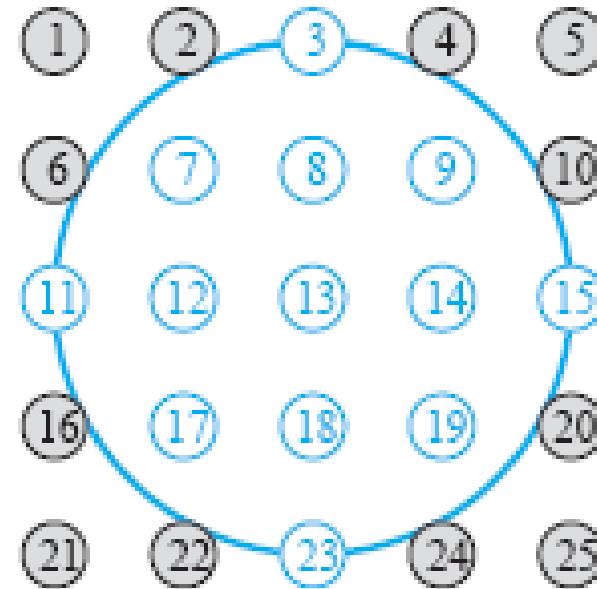
The result is that after many presentation, neighboring neurons will have learned vector similar to each other.

$$N_{13}(1) = \{8, 12, 13, 14, 18\}$$

$$N_{13}(2) = \{3, 7, 8, 9, 11, 12, 13, 14, 15, 17, 18, 19, 23\}$$



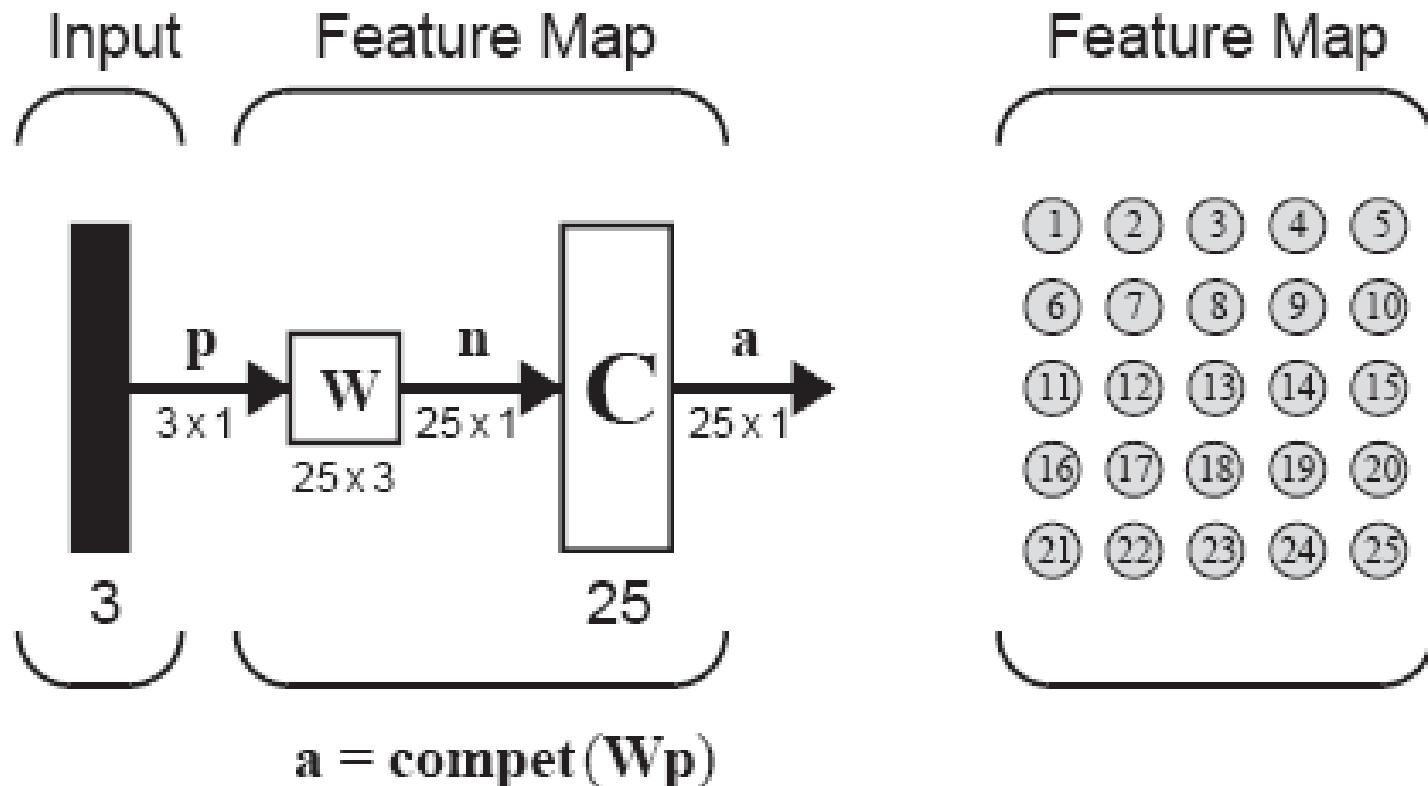
$N_{13}(1)$



$N_{13}(2)$

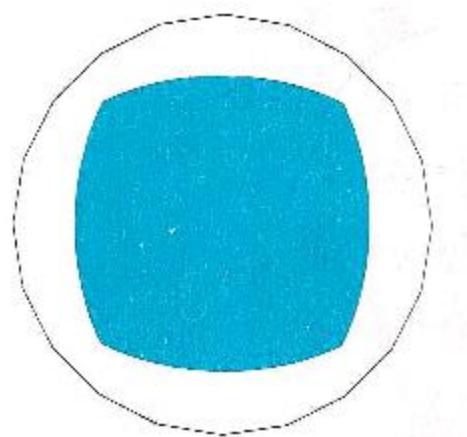
# Example

Although this example demonstrates 2D topology for neurons, we may have one dimensional or 3 or more dimensional arrangement.

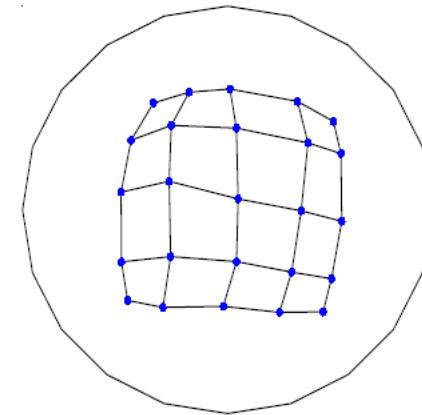
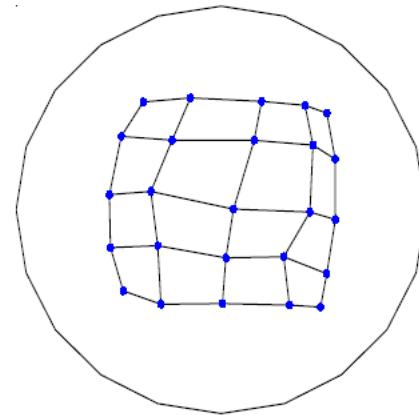
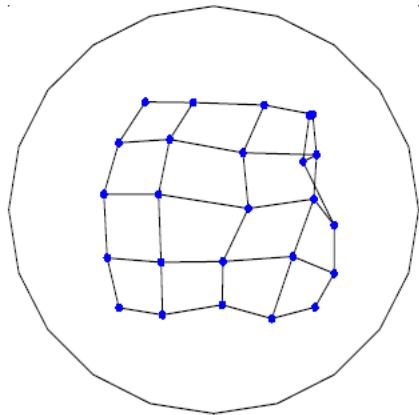
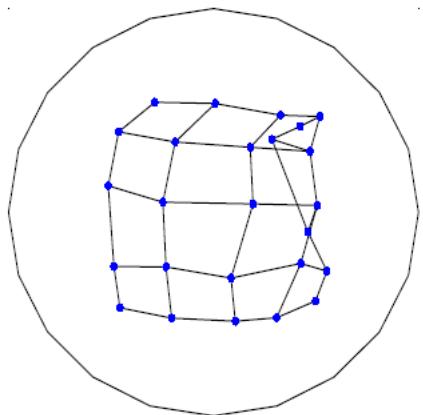
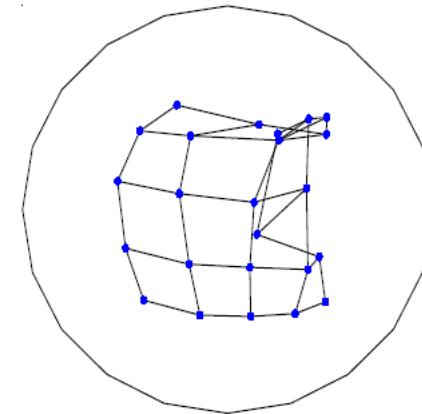
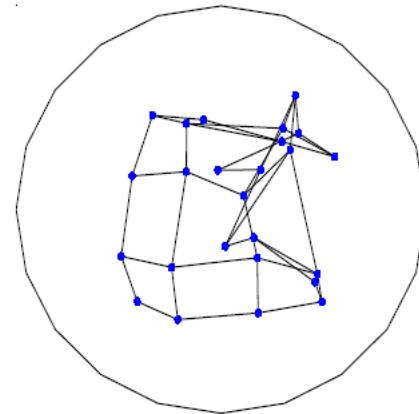
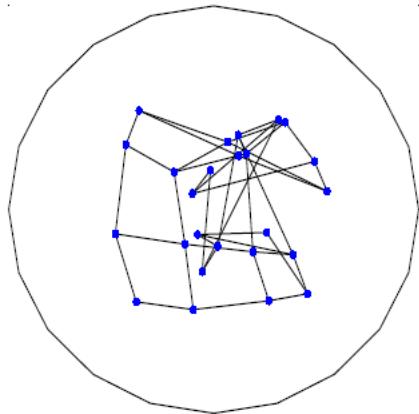
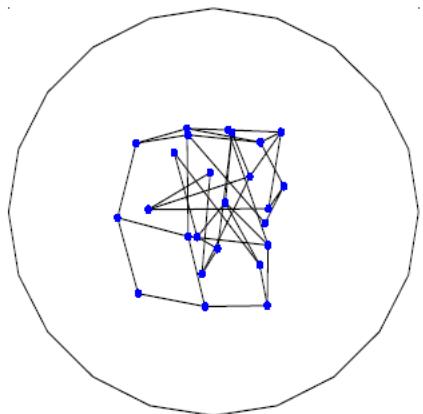


- We assume all weights are normalized, so each 3D vectors for the feature map can be represented by a dot on the sphere. Dots of neighboring neurons are connected by lines so you can see how the physical topology of the network is arranged in the input space.
- Each time a vector is presented, the neuron with the closest weight vector will win the competition. The winning neuron and its neighbors move their weight vectors closer to the input vector (and therefore to each other). We used neighborhood with radius 1.

- Input randomly drawn data from the blue area
- Adjust the nodes locations so that they represent the data



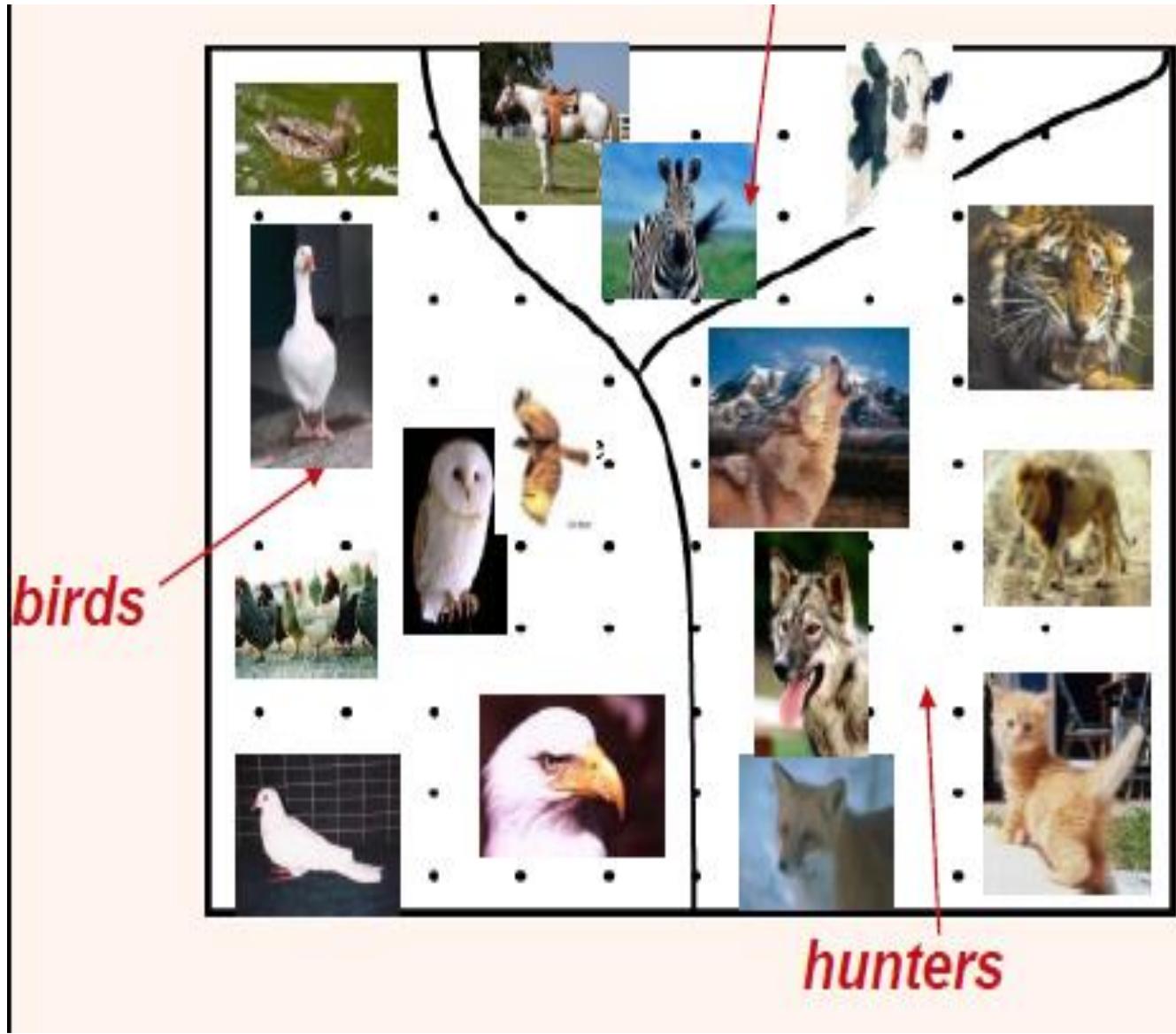
# All iterations (250 Iterations per Diagram)

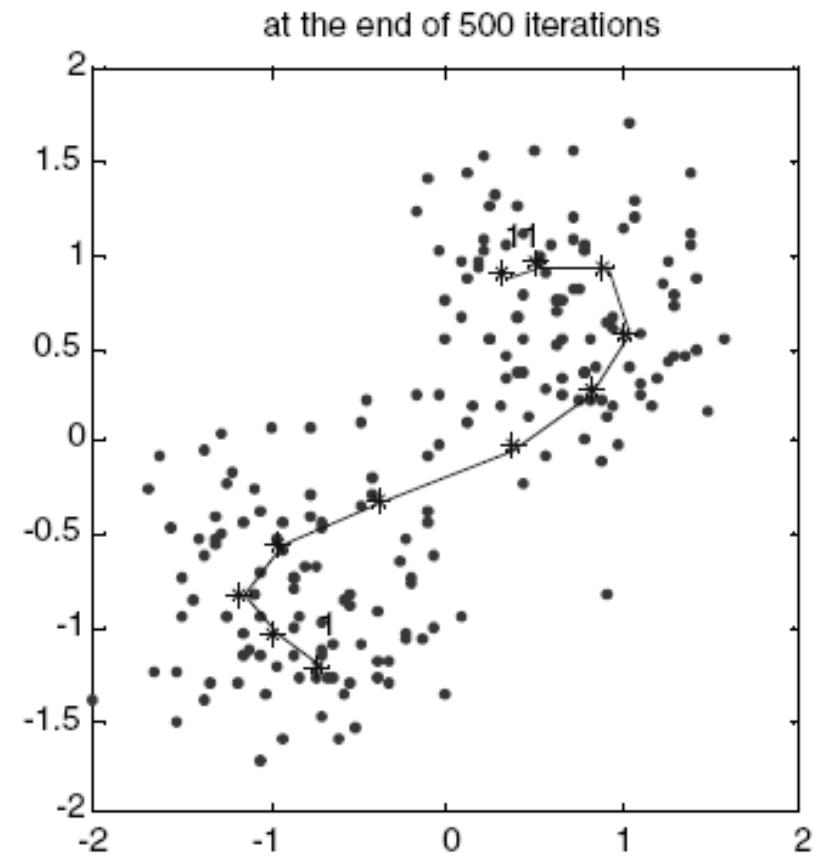
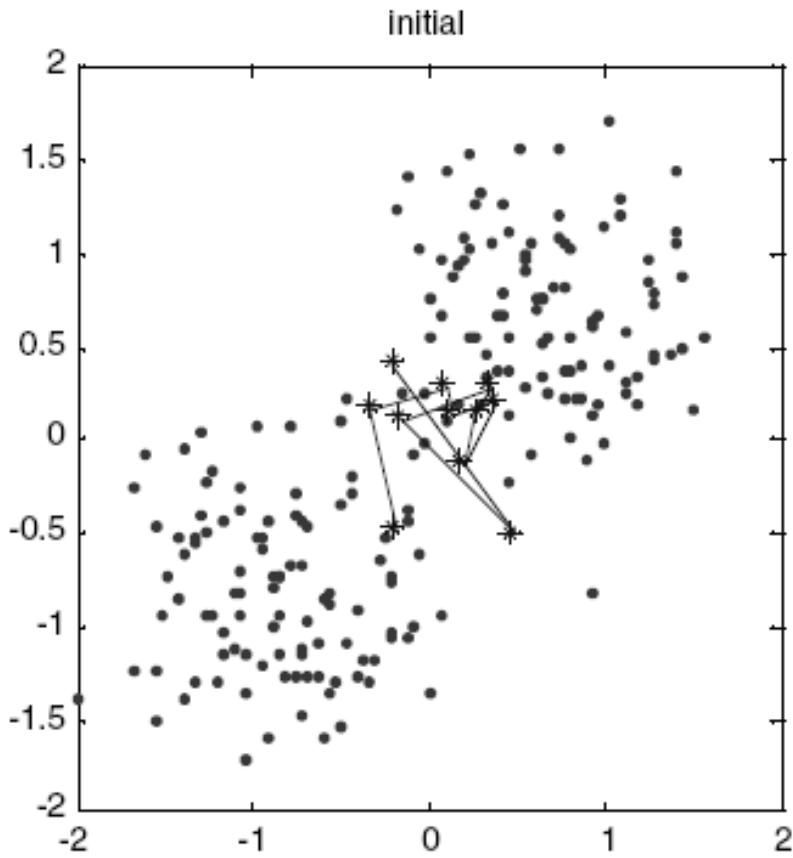


# Animal Example

TABLE 9.2 Animal Names and Their Attributes

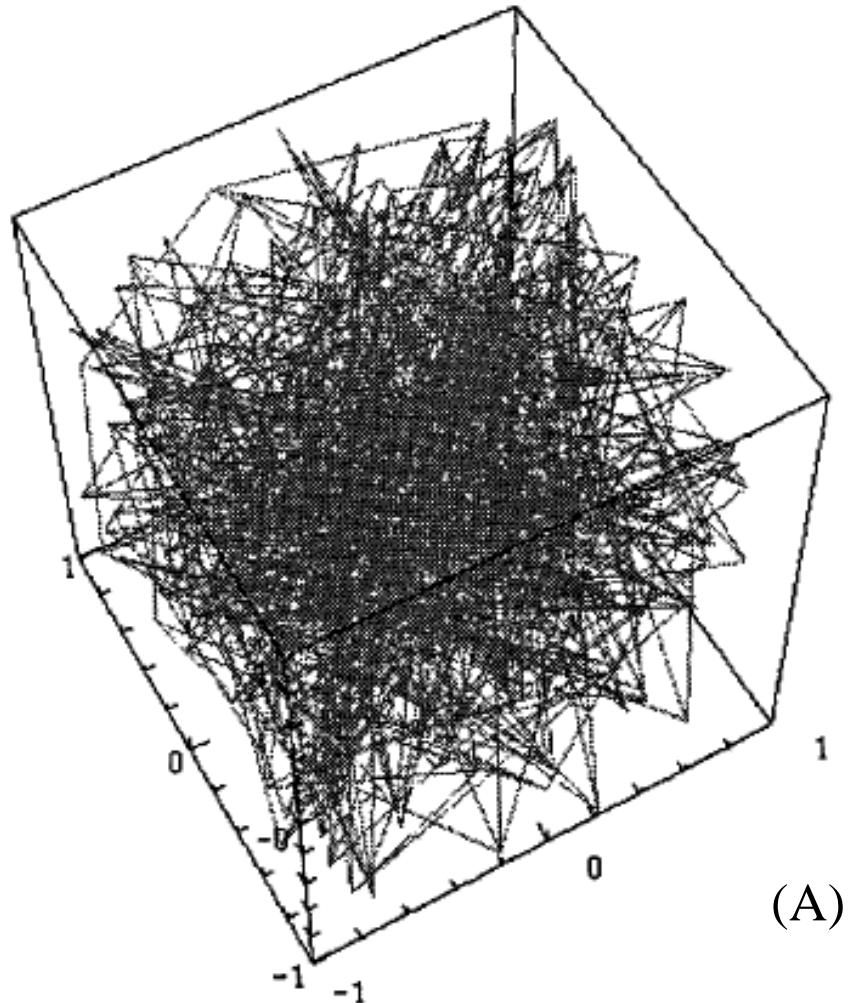
| Animal      | Dove     | Hen | Duck | Goose | Owl | Hawk | Eagle | Fox | Dog | Wolf | Cat | Tiger | Lion | Horse | Zebra | Cow |
|-------------|----------|-----|------|-------|-----|------|-------|-----|-----|------|-----|-------|------|-------|-------|-----|
| is          | { small  | 1   | 1    | 1     | 1   | 1    | 0     | 0   | 0   | 0    | 1   | 0     | 0    | 0     | 0     | 0   |
|             | medium   | 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   | 1     | 1    | 1     | 1     | 1   |
| has         | { 2 legs | 1   | 1    | 1     | 1   | 1    | 1     | 0   | 0   | 0    | 0   | 0     | 0    | 0     | 0     | 0   |
|             | 4 legs   | 0   | 0    | 0     | 0   | 0    | 0     | 1   | 1   | 1    | 1   | 1     | 1    | 1     | 1     | 1   |
|             | hair     | 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    | 1     | 1     | 1   |
|             | mane     | 0   | 0    | 0     | 0   | 0    | 0     | 0   | 0   | 1    | 0   | 0     | 1    | 1     | 1     | 0   |
|             | feathers | 1   | 1    | 1     | 1   | 1    | 1     | 0   | 0   | 0    | 0   | 0     | 0    | 0     | 0     | 0   |
| likes<br>to | { hunt   | 0   | 0    | 0     | 0   | 1    | 1     | 1   | 0   | 1    | 1   | 1     | 1    | 0     | 0     | 0   |
|             | run      | 0   | 0    | 0     | 0   | 0    | 0     | 0   | 1   | 1    | 0   | 1     | 1    | 1     | 1     | 0   |
|             | fly      | 1   | 0    | 0     | 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   |





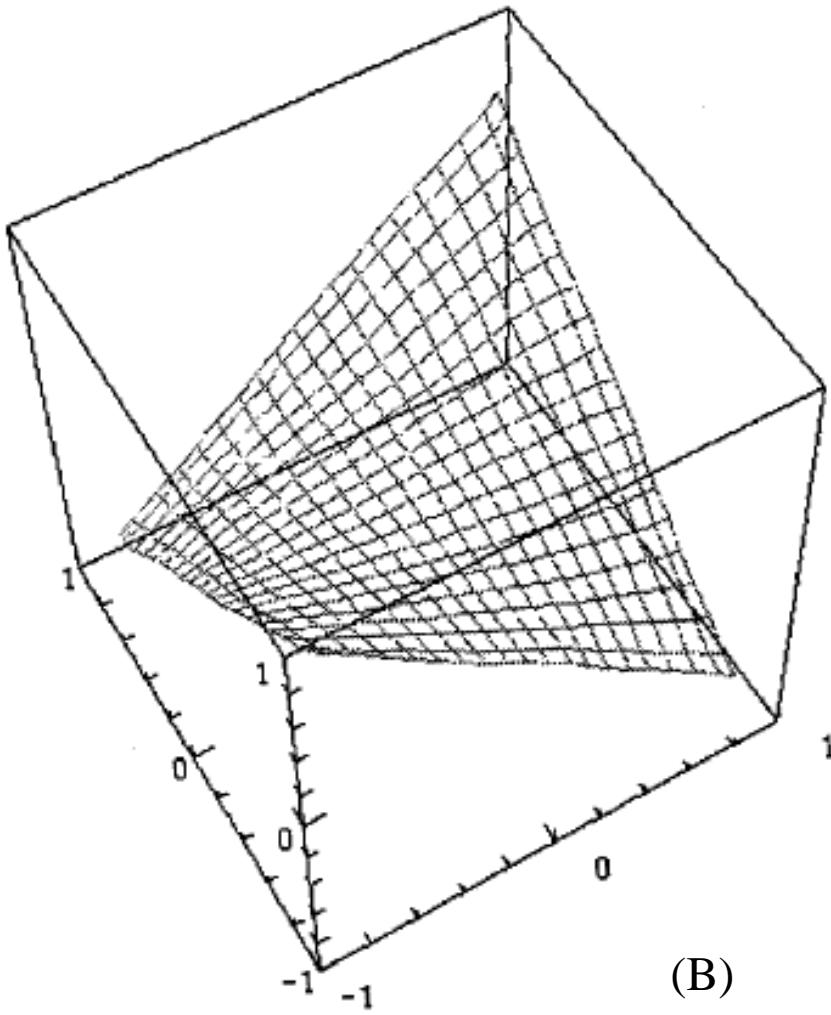
The neurons are initially placed randomly in the feature space as shown to the left of the figure. After 500 iterations, they are distributed evenly to represent the underlying feature vector distribution.

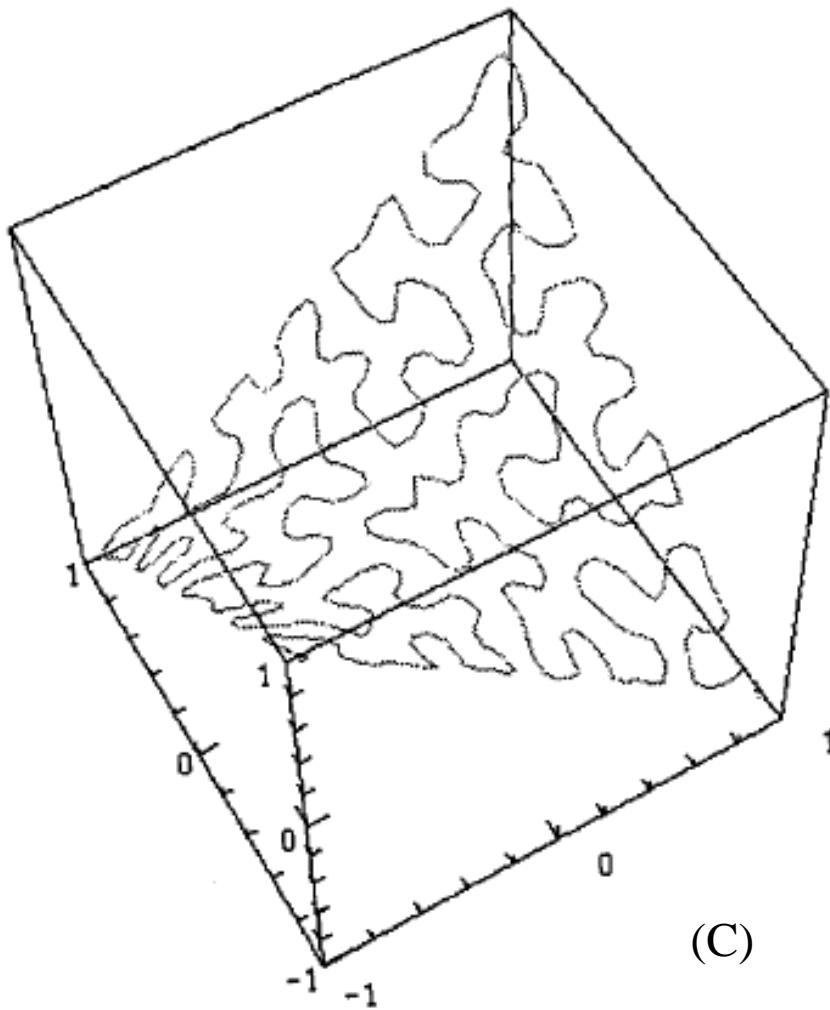
Well trained net should have same topology as that in the physical space, and will reflect the properties of the training set.



(A)

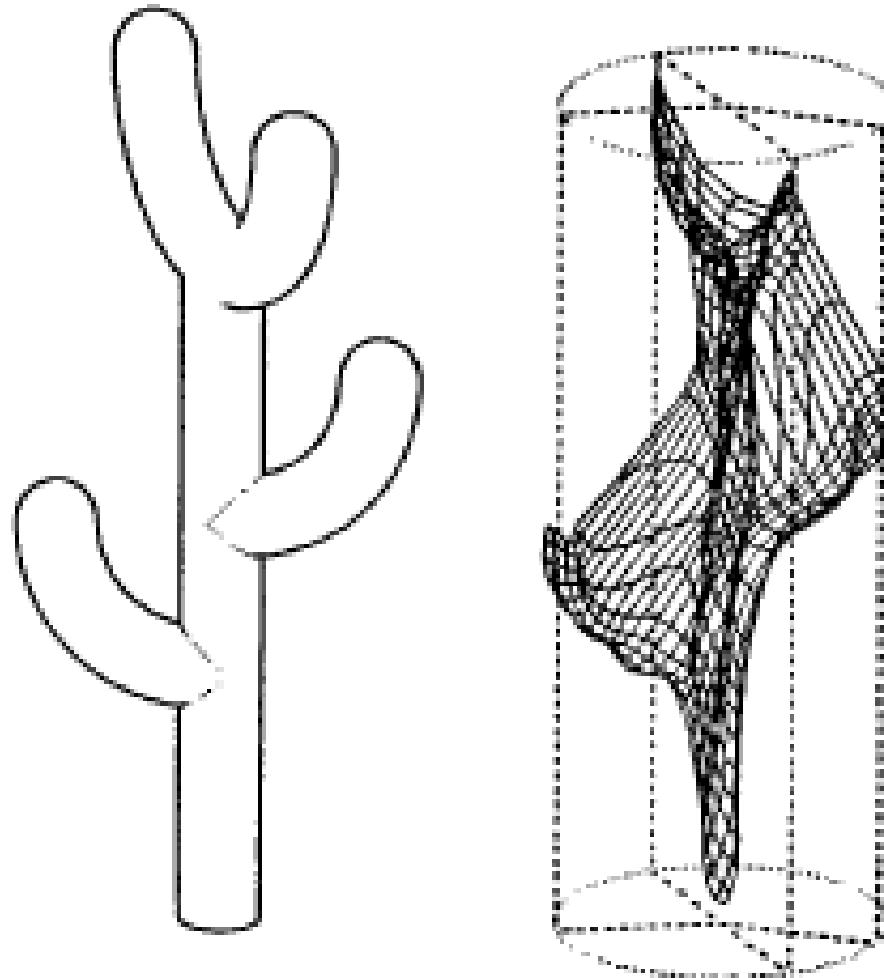
A disordered initial state





*Dimension conflict:* when replacing the two-dimensional feature map of parts *A* and *B* with a one-dimensional chain, a space-filling curve results.

# Three dimensional example



- One method to improve the performance of the feature map is to **vary the size of the neighborhoods** during training. Initially, the neighborhood size,  $d$  is set large. As training progresses,  $d$  is gradually reduced, until it only includes the winning neuron. This speeds up self-organizing .
- The **learning rate** can also be **varied** over time. An initial rate of 1 allows neurons to quickly learn presented vectors. During training, the learning rate is decreased asymptotically toward 0, so that learning becomes stable.
- Another alteration that speeds self-organization is to have the winning neuron use a **larger learning rate** than the neighboring neurons.

nnd14fm1

nnd14fm2

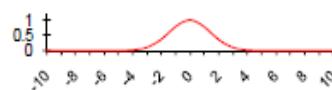
# Neighborhood

- ⌘ Typical choice for the neighborhood function satisfying the desired criteria is the Gaussian:

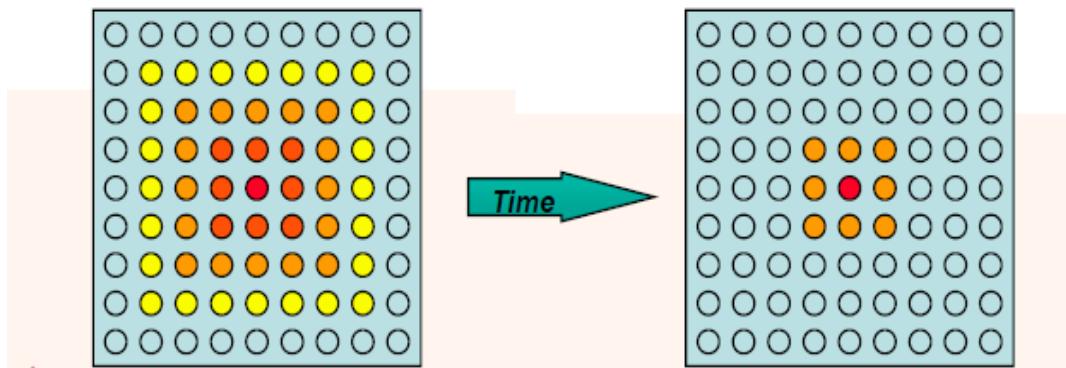
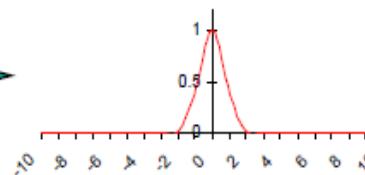
$$T_{i,i^*} = \exp\left(-\frac{d_{i^*,i}^2}{2\sigma^2}\right)$$

where  $i^*$  is the index of the winning neuron;

$d_{j,i}$  is the topologic distance between nodes  $i$  and  $j$



Time →



# Neighborhood

- ⌘ Neighborhood gets smaller in time

$$\sigma^2 = \sigma_0^2 \exp\left(-\frac{2q}{\tau_\sigma}\right)$$

where  $\sigma_0$  is the initial value of  $\sigma$   
 $q=0,1,2\dots$   $\tau_\sigma$  is a time constant

$$\alpha(q) = \alpha_0 \exp\left(-\frac{q}{\tau_\alpha}\right)$$

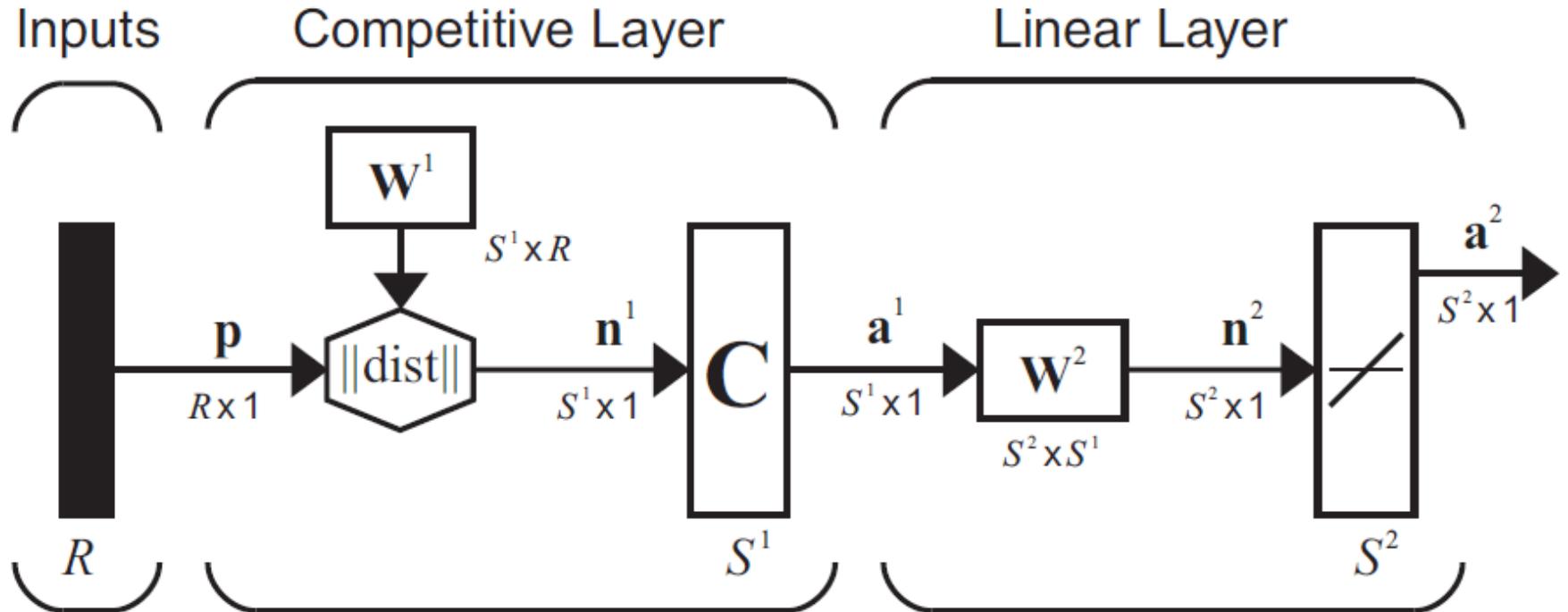
- ⌘ Variable Learning rate

## Weight Update

$$_i \mathbf{w}(q) = _i \mathbf{w}(q-1) + \alpha_0 \exp\left(-\frac{q}{\tau_\alpha}\right) \exp\left(-\frac{d_{i^*,i}^2}{2\sigma_0^2 \exp\left(-\frac{2q}{\tau_\sigma}\right)}\right) (\mathbf{p}(q) - _i \mathbf{w}(q-1))$$

# Learning Vector Quantization

- The LVQ network is a hybrid network. It uses both **unsupervised** and **supervised** learning to form classifications.
- Each neuron in the first layer is assigned to a class, with several neurons often assigned to the same class.
- Each class is then assigned to one neuron in the second layer.
- Each neuron in the 1<sup>st</sup> layer of the LVQ net learn a prototype vector, which allows to classify a region of the input space.



$$n_i^1 = - \left\| {}_i \mathbf{w}^1 - \mathbf{p} \right\|$$

$$\mathbf{a}^2 = \mathbf{W}^2 \mathbf{a}^1$$

$$\mathbf{a}^1 = \mathbf{compet}(\mathbf{n}^1)$$

- The net input is **NOT** computed by taking an inner product of the prototype vectors with the input. Instead, the net input is the negative of the distance between the prototype vectors and the input. (Normalization of vectors is not needed.)

The net input of the 1<sup>st</sup> layer of the LVQ is:  $n_i^1 = -\|_i \mathbf{w}^1 - \mathbf{p}\|$

Vector from

$$\mathbf{n}^1 = -\begin{bmatrix} \|\mathbf{w}^1 - \mathbf{p}\| \\ \|\mathbf{w}^2 - \mathbf{p}\| \\ \vdots \\ \|\mathbf{w}^{S^1} - \mathbf{p}\| \end{bmatrix}$$

The output of the 1<sup>st</sup> layer of the LVQ is:  $\mathbf{a}^1 = \mathbf{compet}(\mathbf{n}^1)$

- The neuron whose weight vector is closest to the input will output a 1 and the other will output 0.

# Subclass

- For the LVQ network, the winning neuron in the first layer indicates the **subclass** which the input vector belongs to. There may be several different neurons (subclasses) which make up each class.
- The second layer of the LVQ network combines subclasses into a single class. The columns of  $\mathbf{W}^2$  represent subclasses, and the rows represent classes.  $\mathbf{W}^2$  has a single 1 in each column, with the other elements set to zero. The row in which the 1 occurs indicates which class the appropriate subclass belongs to.

$$(w_{k,i}^2 = 1) \Rightarrow \text{subclass } i \text{ is a part of class } k$$

# Example

$$\mathbf{W}^2 = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

- Subclasses 1, 3 and 4 belong to class 1.
- Subclass 2 belongs to class 2.
- Subclasses 5 and 6 belong to class 3.

A single-layer competitive network can create convex classification regions. The second layer of the LVQ network can combine the convex regions to create more complex categories.

# LVQ Learning

- LVQ learning combines competitive learning with supervision. It requires a training set of examples of proper network behavior.

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \dots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

- Each target vector must contain only zeros, except for a single 1. The row in which the 1 appears indicates the class to which the input belongs.
- All elements of  $\mathbf{W}^2$  are set to zero, except for the following:

If hidden neuron  $i$  is to be assigned to class  $k$  then set  $w_{ki}^2 = 1$ .

- At each iteration, an input vector  $\mathbf{p}$  is presented to the network, and the distance from  $\mathbf{p}$  to each prototype vector is computed. The hidden neurons compete, neuron  $i^*$  wins the competition, and the  $i^*$ th element of  $\mathbf{a}^1$  is set to 1. Next,  $\mathbf{a}^1$  is multiplied by  $\mathbf{W}^2$  to get the final output  $\mathbf{a}^2$ . Kohonen rule can be used as well.
- If the input pattern is classified correctly, then move the winning weight toward the input vector according to the Kohonen rule.

$${}_{i^*}\mathbf{w}^1(q) = {}_{i^*}\mathbf{w}^1(q-1) + \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{w}^1(q-1)) \quad a_{k^*}^2 = t_{k^*} = 1$$

- If the input pattern is classified incorrectly, then move the winning weight away from the input vector.

$${}_{i^*}\mathbf{w}^1(q) = {}_{i^*}\mathbf{w}^1(q-1) - \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{w}^1(q-1)) \quad a_{k^*}^2 = 1 \neq t_{k^*} = 0$$

# Example

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right\} \quad \left\{ \mathbf{p}_2 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \right\} \quad \left\{ \mathbf{p}_3 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \mathbf{t}_3 = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \right\} \quad \left\{ \mathbf{p}_4 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \mathbf{t}_4 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right\}$$

$$\mathbf{W}^1(0) = \begin{bmatrix} (\mathbf{1}\mathbf{w}^1)^T \\ (\mathbf{2}\mathbf{w}^1)^T \\ (\mathbf{3}\mathbf{w}^1)^T \\ (\mathbf{4}\mathbf{w}^1)^T \end{bmatrix} = \begin{bmatrix} 0.25 & 0.75 \\ 0.75 & 0.75 \\ 1 & 0.25 \\ 0.5 & 0.25 \end{bmatrix}$$

↑  
Random values

$$\mathbf{W}^2 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

# First Iteration

$$\mathbf{a}^1 = \mathbf{compet}(\mathbf{n}^1) = \mathbf{compet} \begin{pmatrix} -\|_1 \mathbf{w}^1 - \mathbf{p}_1 \| \\ -\|_2 \mathbf{w}^1 - \mathbf{p}_1 \| \\ -\|_3 \mathbf{w}^1 - \mathbf{p}_1 \| \\ -\|_4 \mathbf{w}^1 - \mathbf{p}_1 \| \end{pmatrix}$$

$$\mathbf{a}^1 = \mathbf{compet} \begin{pmatrix} -\left\| \begin{bmatrix} 0.25 & 0.75 \end{bmatrix}^T - \begin{bmatrix} 0 & 1 \end{bmatrix}^T \right\| \\ -\left\| \begin{bmatrix} 0.75 & 0.75 \end{bmatrix}^T - \begin{bmatrix} 0 & 1 \end{bmatrix}^T \right\| \\ -\left\| \begin{bmatrix} 1.00 & 0.25 \end{bmatrix}^T - \begin{bmatrix} 0 & 1 \end{bmatrix}^T \right\| \\ -\left\| \begin{bmatrix} 0.50 & 0.25 \end{bmatrix}^T - \begin{bmatrix} 0 & 1 \end{bmatrix}^T \right\| \end{pmatrix} = \mathbf{compet} \begin{pmatrix} -0.354 \\ -0.791 \\ -1.25 \\ -0.901 \end{pmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The 1<sup>st</sup> hidden neuron has the closest weight to  $\mathbf{p}_1$ .

# Second Layer

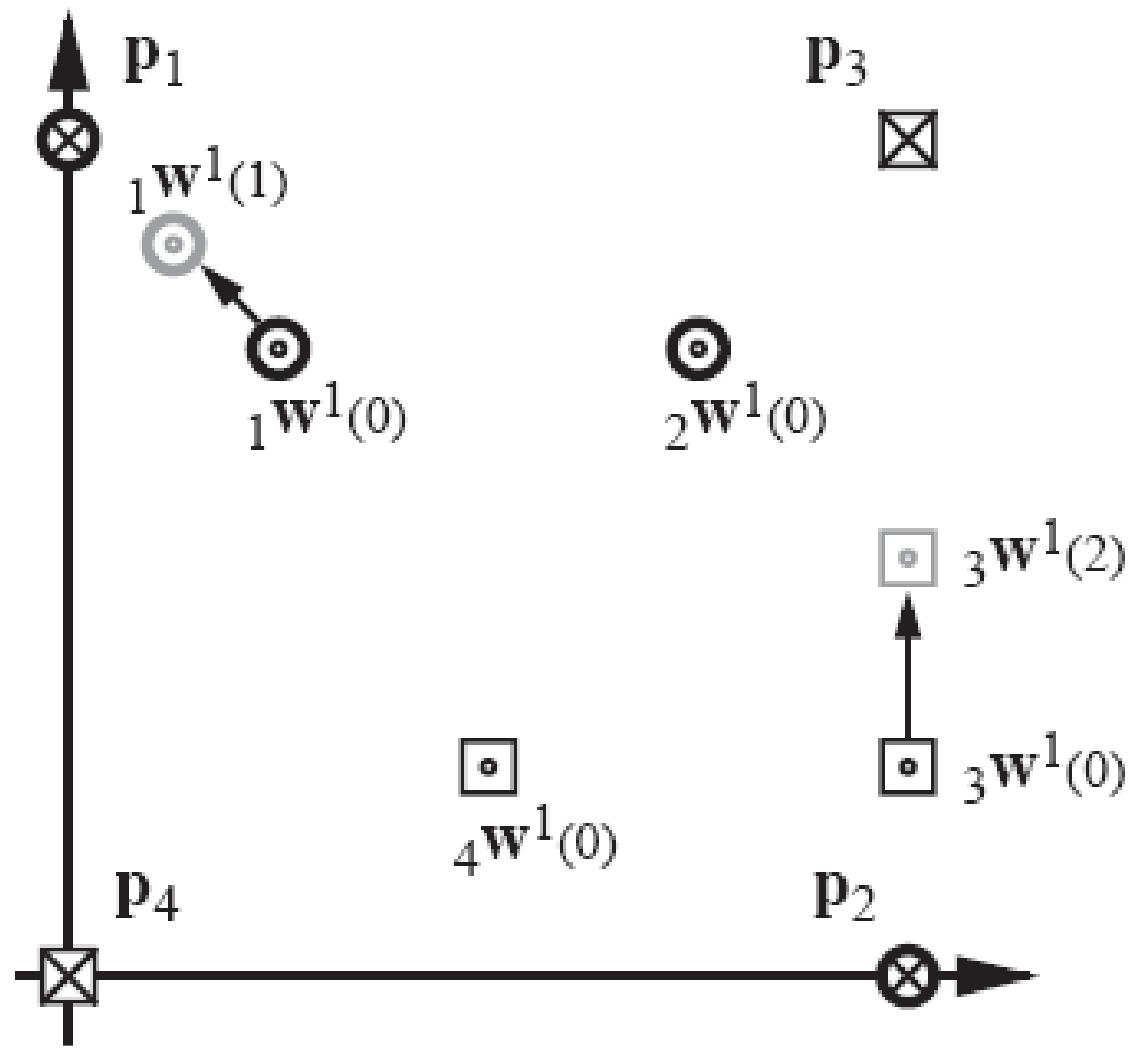
$$\mathbf{a}^2 = \mathbf{W}^2 \mathbf{a}^1 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

- This output indicates that  $\mathbf{p}_1$  is a member of class 1.
- This is the correct class, therefore the weight vector is moved toward the input vector.

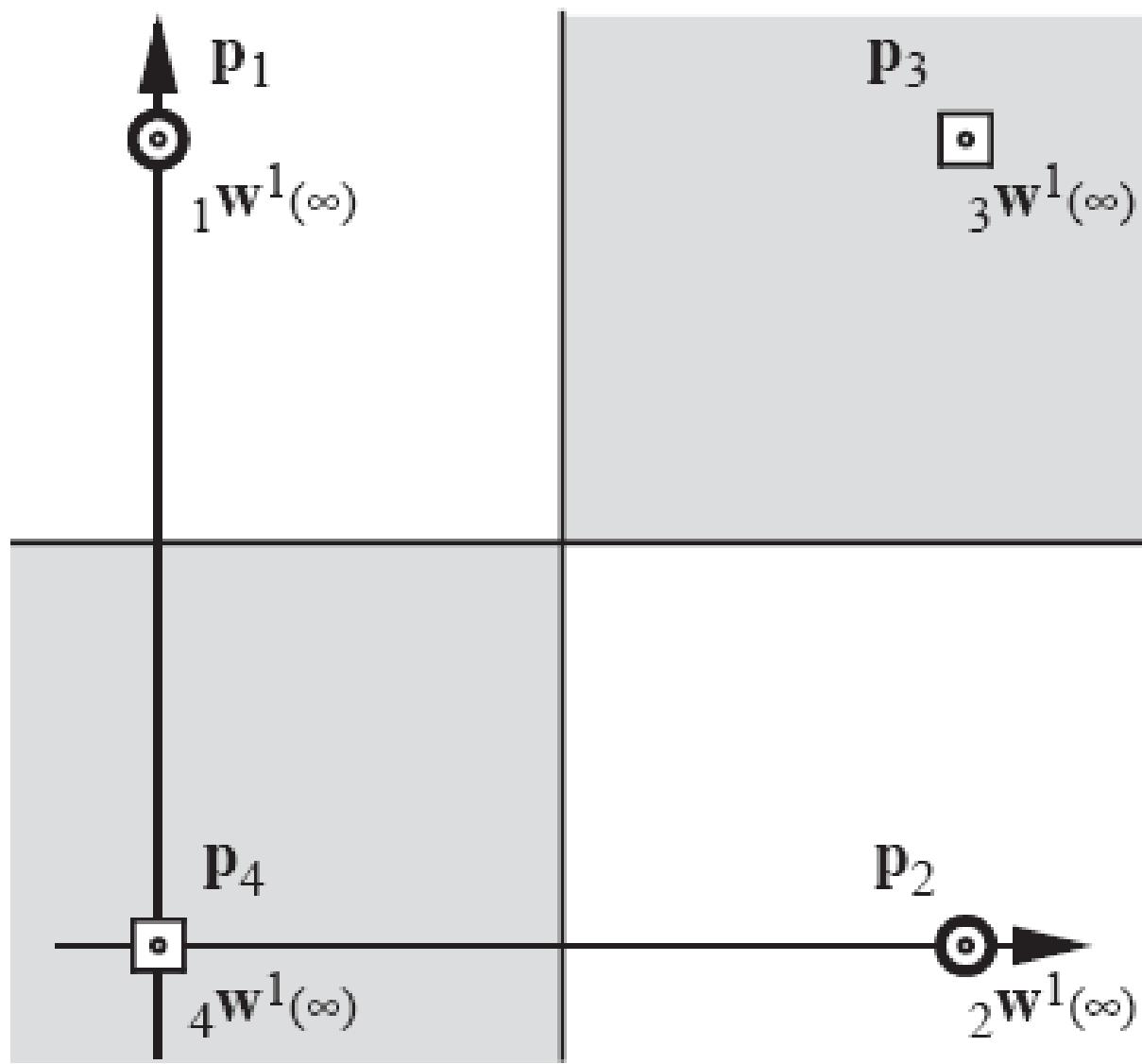
$$_1\mathbf{w}^1(1) = _1\mathbf{w}^1(0) + \alpha(\mathbf{p}_1 - _1\mathbf{w}^1(0))$$

$$_1\mathbf{w}^1(1) = \begin{bmatrix} 0.25 \\ 0.75 \end{bmatrix} + 0.5 \left( \begin{bmatrix} 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 0.25 \\ 0.75 \end{bmatrix} \right) = \begin{bmatrix} 0.125 \\ 0.875 \end{bmatrix}$$

# Figure



# Final Decision Regions

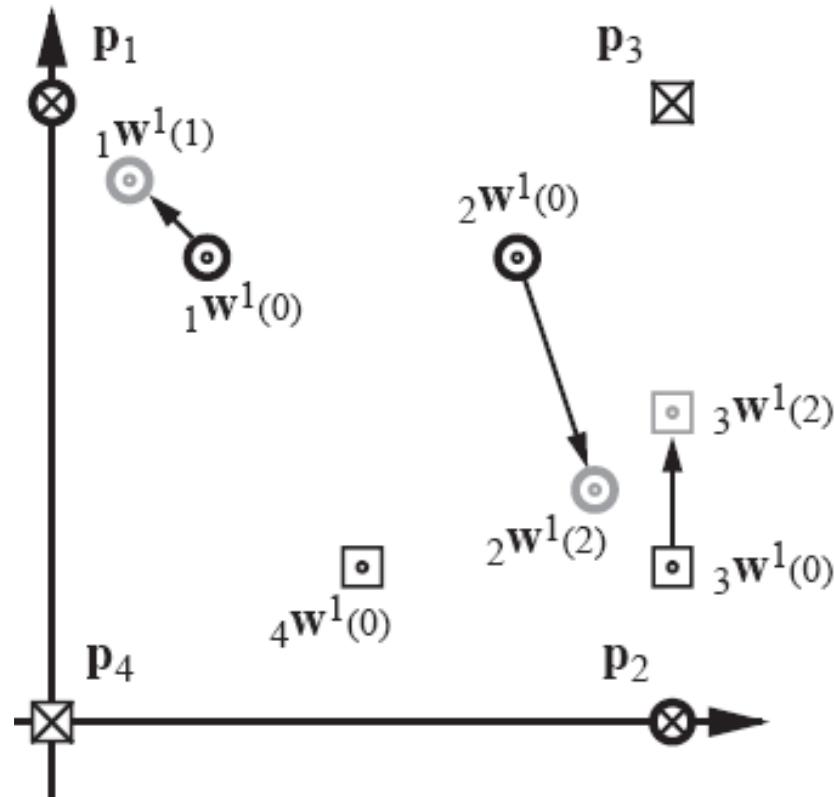


# LVQ2

- Occasionally a hidden neuron in an LVQ network can have initial weight values that stop it from ever winning the competition. The result is a dead neuron that never does anything useful. This problem is solved with the use of a “conscience” mechanism.
- If the winning neuron in the hidden layer incorrectly classifies the current input, we move its weight vector away from the input vector, as before. However, we also adjust the weights of the closest neuron to the input vector that does classify it properly. The weights for this second neuron should be moved toward the input vector.

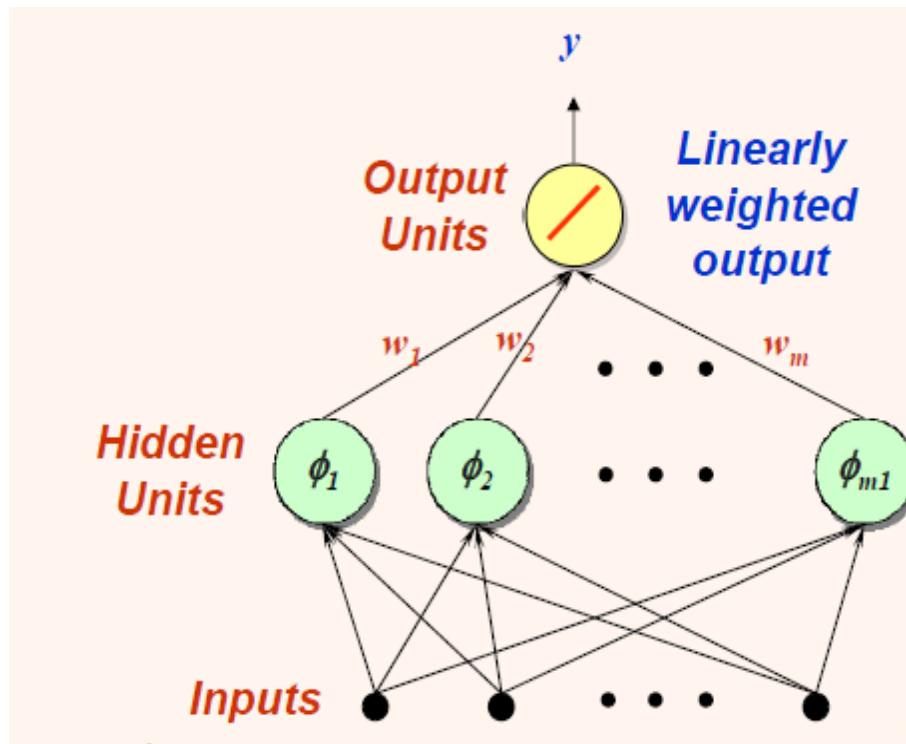
- When the network correctly classifies an input vector, the weights of only one neuron are moved toward the input vector. However, if the input vector is incorrectly classified, the weights of two neurons are updated, one weight vector is moved away from the input vector, and the other one is moved toward the input vector. The resulting algorithm is called **LVQ2**.

# LVQ2 Example



[nnd14lv1](#) [nnd14lv2](#)

# Radial Basis Networks



$$f(\mathbf{x}) = \sum_{i=1}^{m_1} w_i \phi_i(\mathbf{x})$$

$$\phi_i(\mathbf{x}) = \phi(||\mathbf{x} - \mathbf{x}_i||)$$

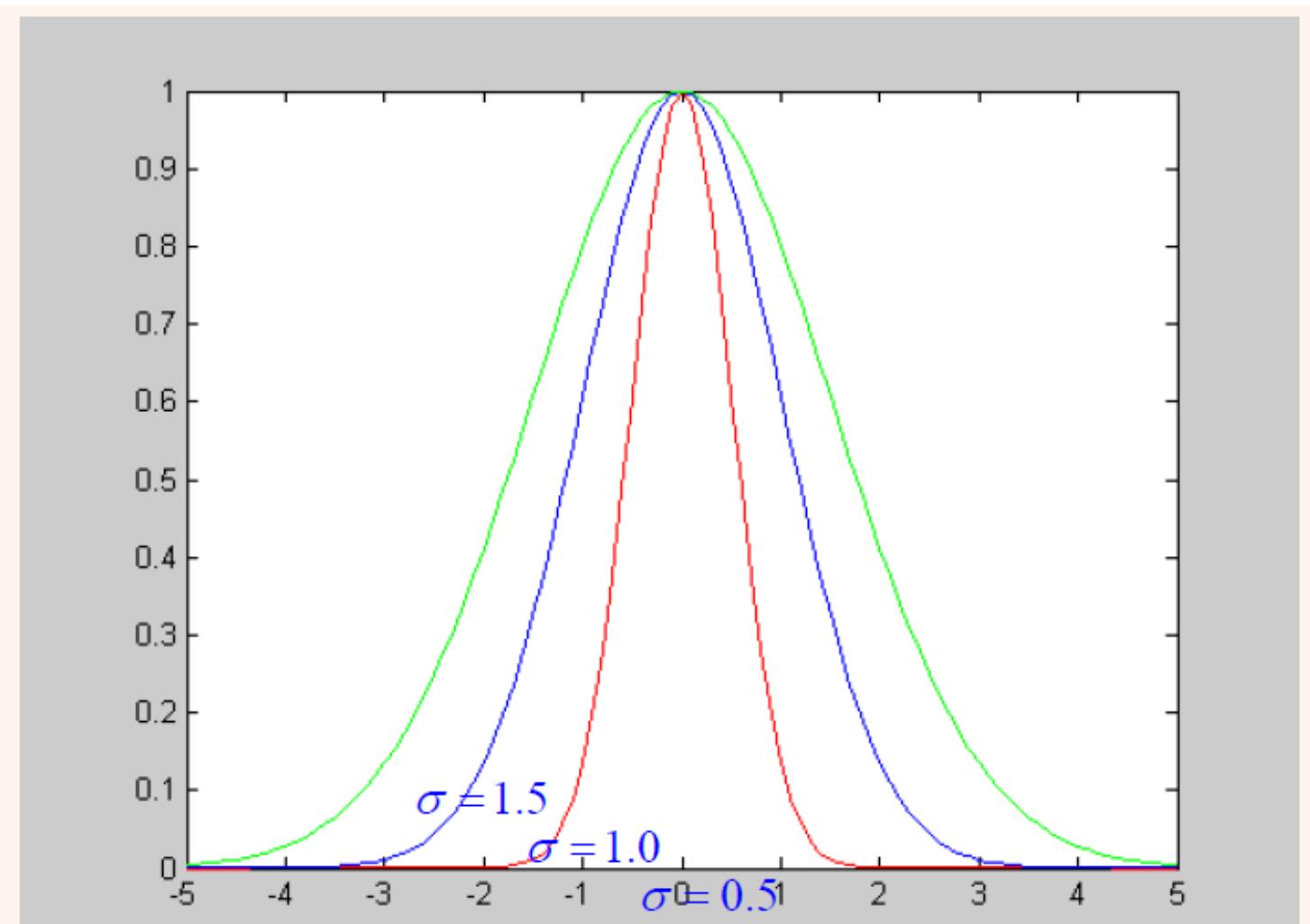
Arrows point from the equation to three boxes below:

- Left box: **دستگاه**
- Middle box: **نحوه**
- Right box: **مرز**

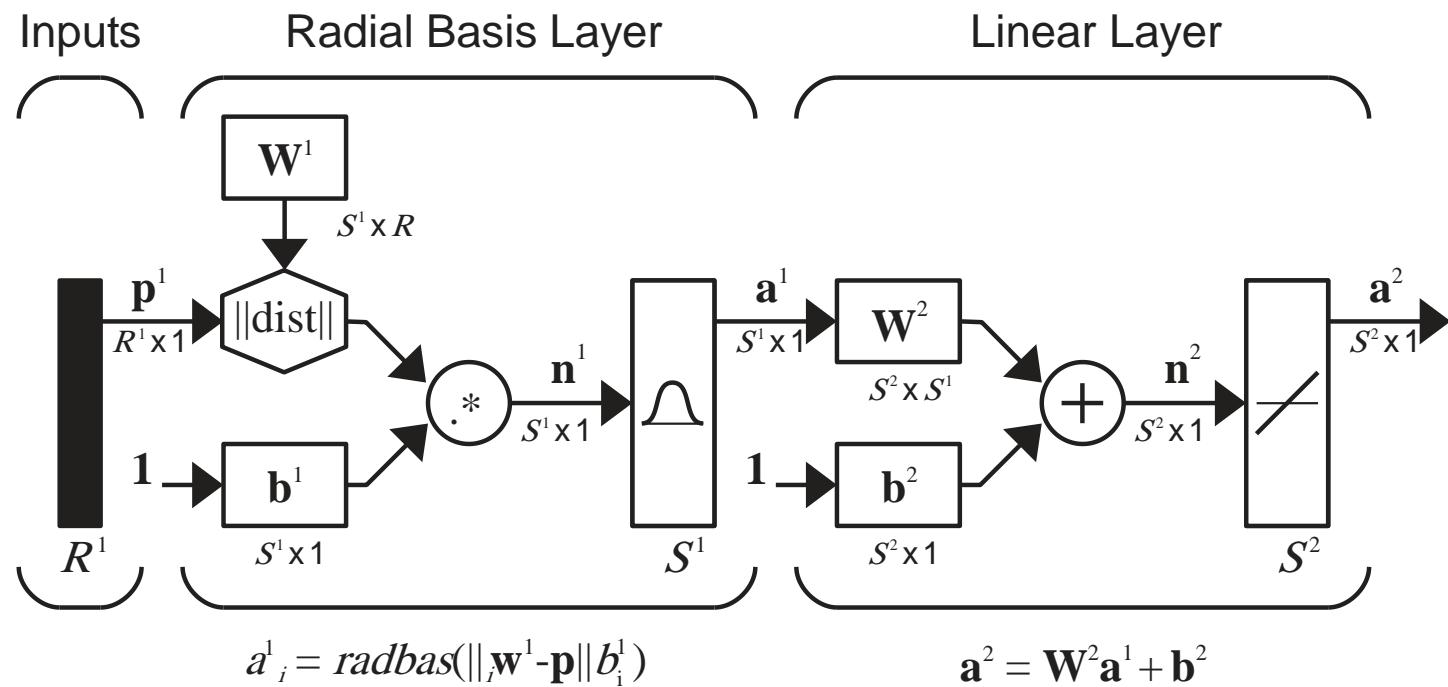
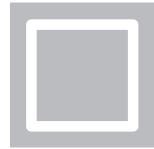
# Gaussian Function

$$\phi(r) = e^{-\frac{r^2}{2\sigma^2}} \quad \sigma > 0 \quad \text{and} \quad r \in \Re$$

$$r = \|x - x_j\|$$



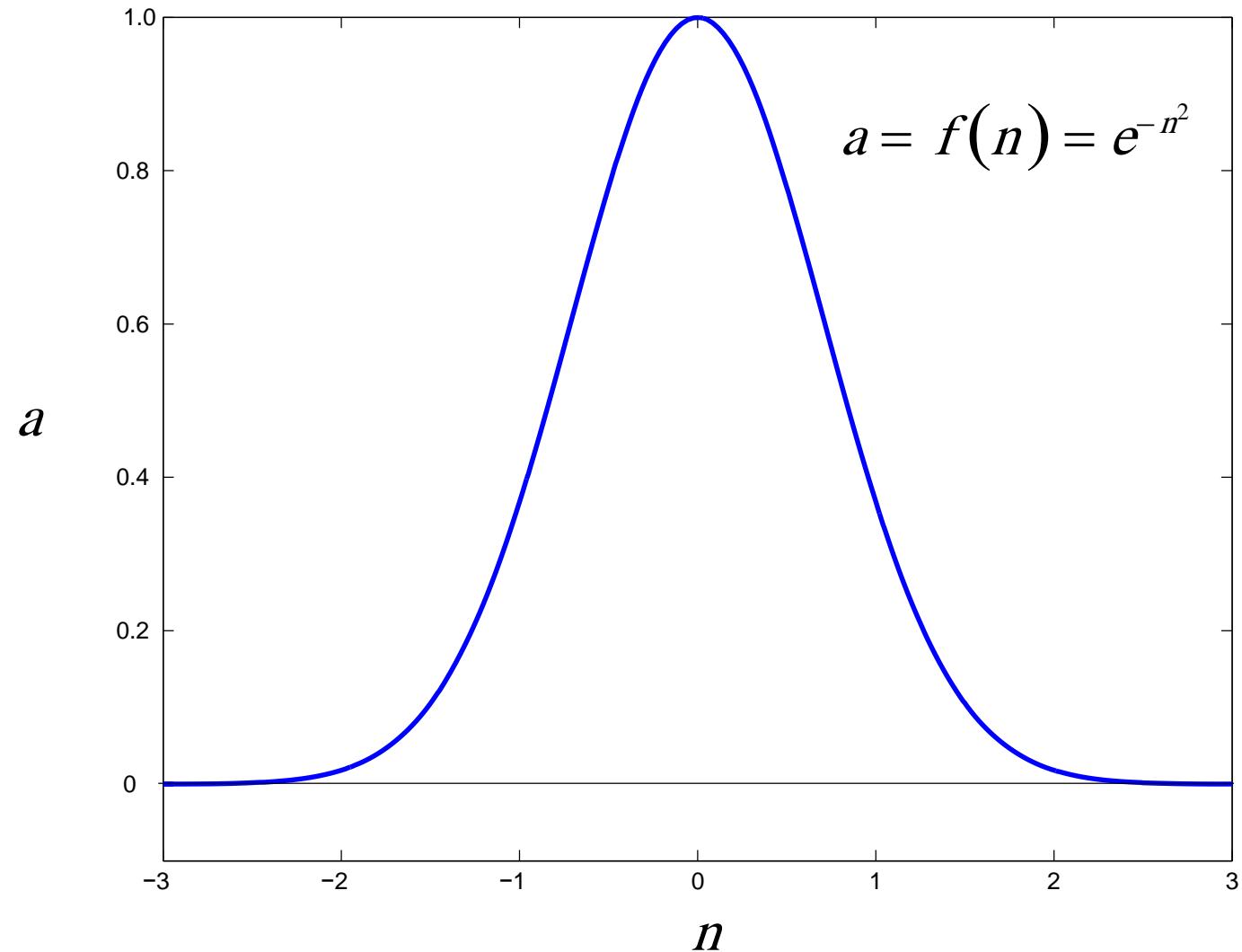
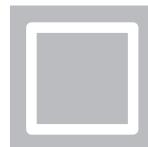
# Radial Basis Network



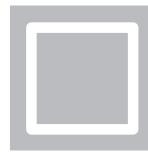
$$n_i^1 = \|\mathbf{p} - \mathbf{w}^1\| b_i^1 \quad b = 1 / (\sigma \sqrt{2}) \quad a = f(n) = e^{-n^2}$$

The first layer weight vectors  $i\mathbf{w}^1$  are called “centers” of the basis functions.

Bias → standard deviation, variance or spread constant

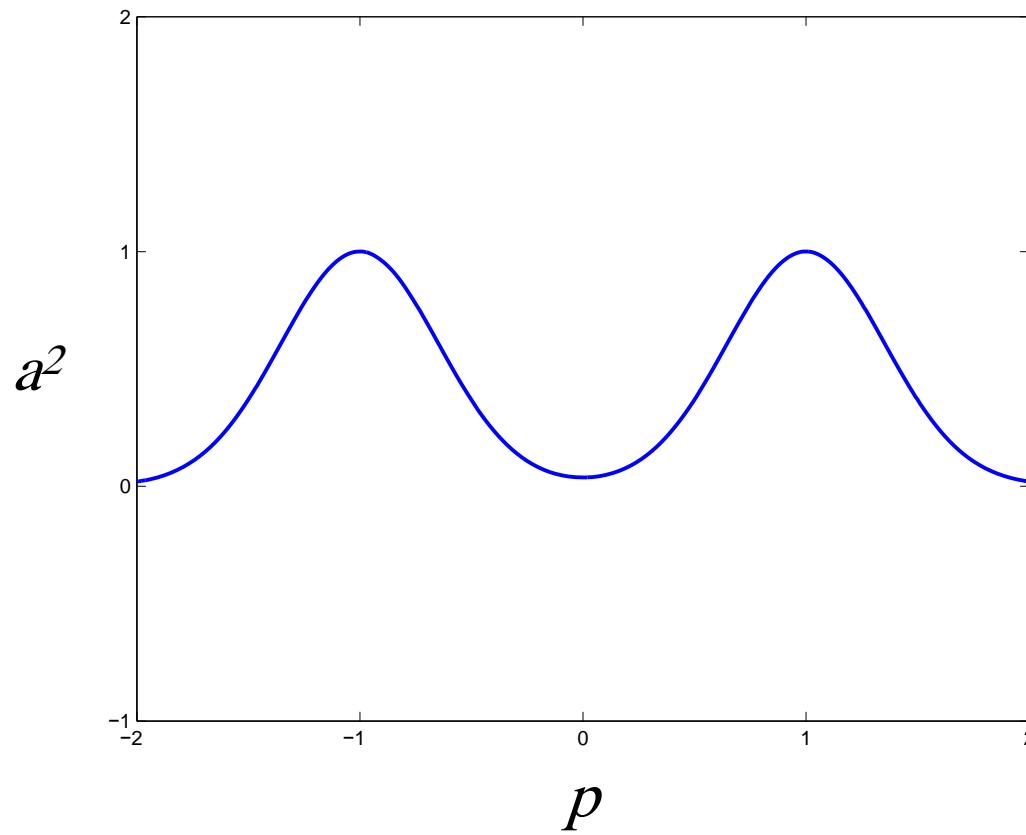


# Example Network Function

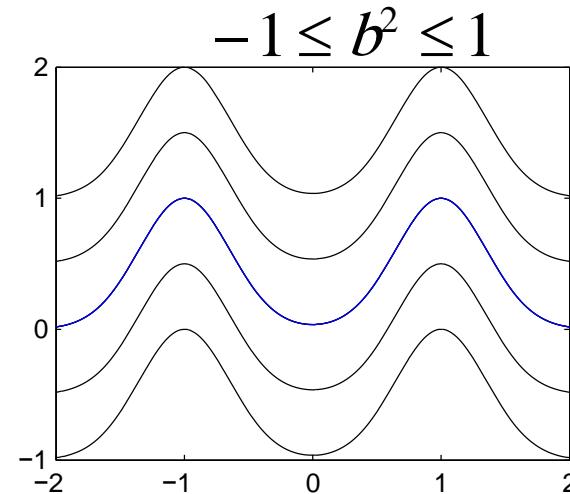
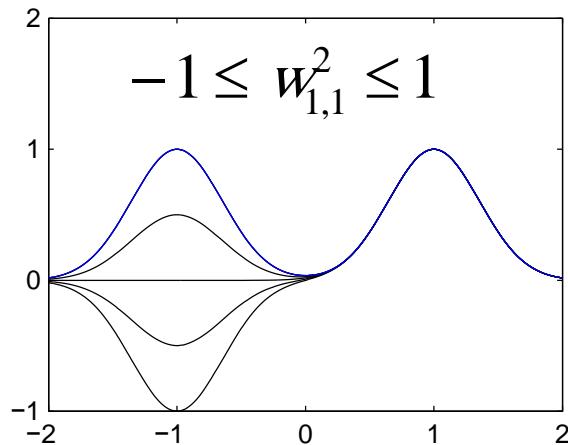
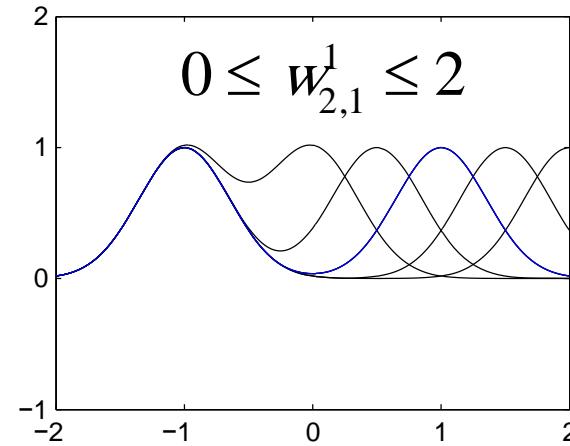
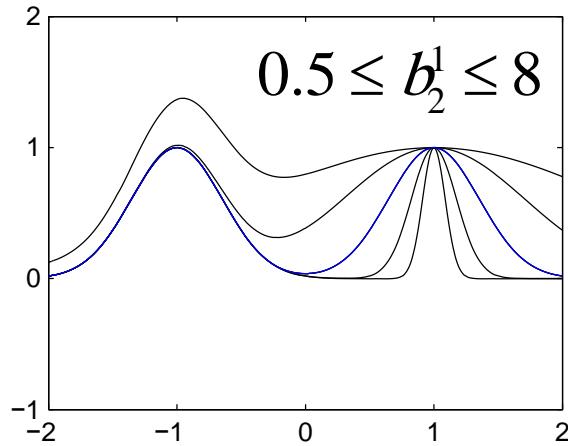


$$w_{1,1}^1 = -1, w_{2,1}^1 = 1, b_1^1 = 2, b_2^1 = 2$$

$$w_{1,1}^2 = 1, w_{1,2}^2 = 1, b^2 = 0$$



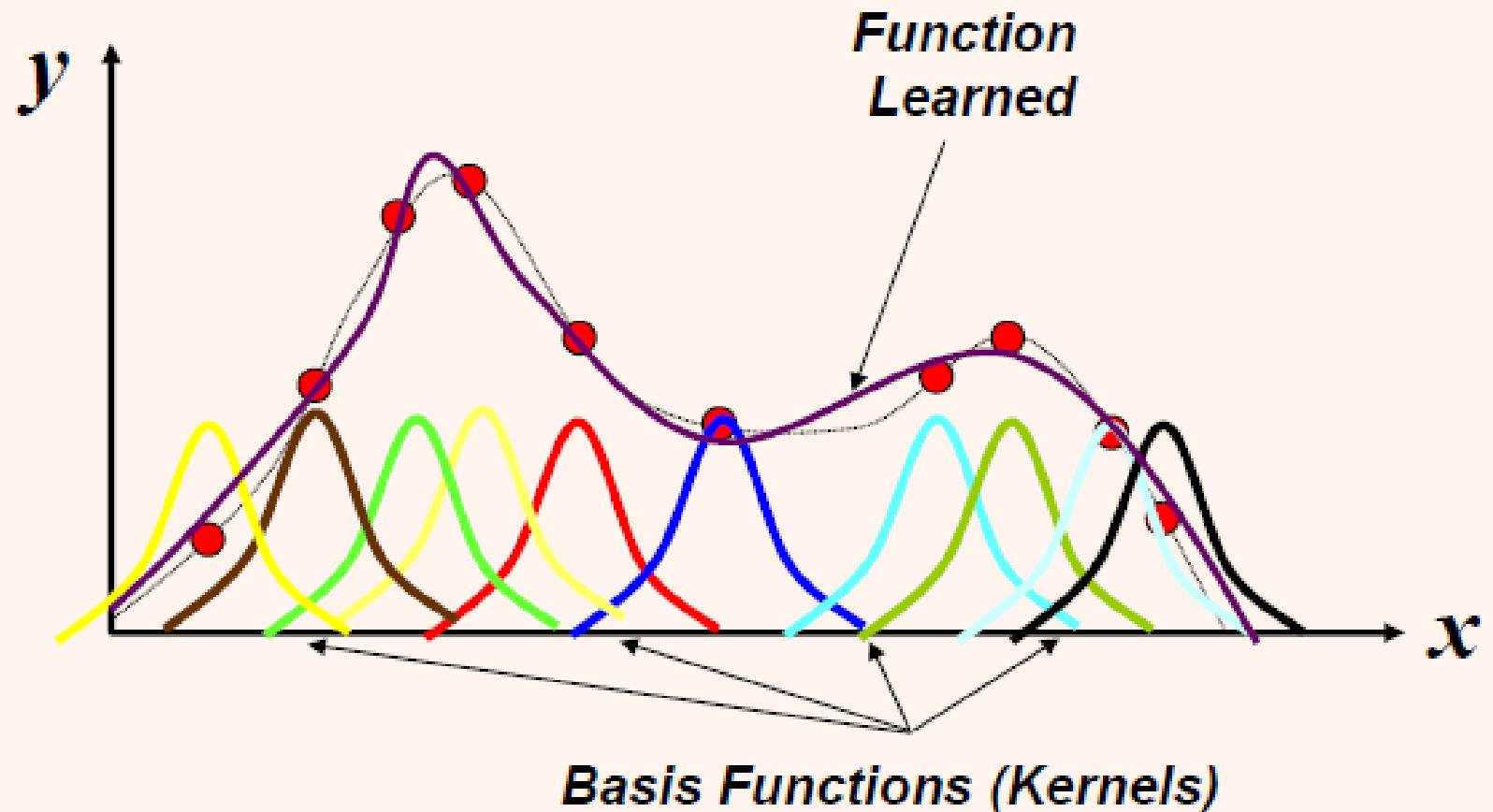
# Parameter Variations



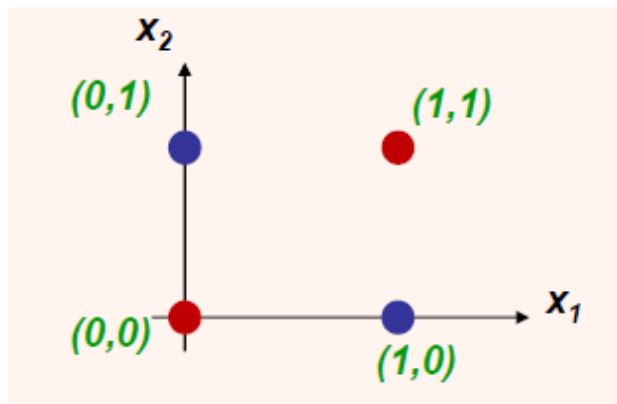
If we have enough neurons in the first layer of the RBF network, we can approximate virtually any function of interest!

# Function approximation

$$y = f(\mathbf{x}) = \sum_{i=1}^{m_1} w_i \phi_i(\mathbf{x})$$



## حل مسالهی XOR با شبکه‌ی RBF



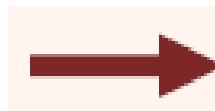
$t_1 = (1,1)$  and  $t_2 = (0,0)$

نگاشت به صفر شده در یک گروه  
 $(1,1)$  و  $(0,0)$   
 نگاشت به یک شده در گروه دیگر قرار می‌گیرند  
 $(1,0)$  و  $(0,1)$

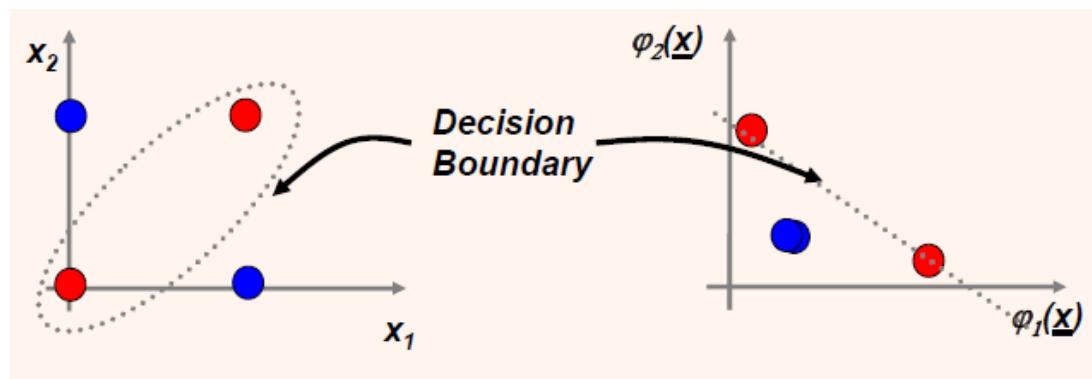
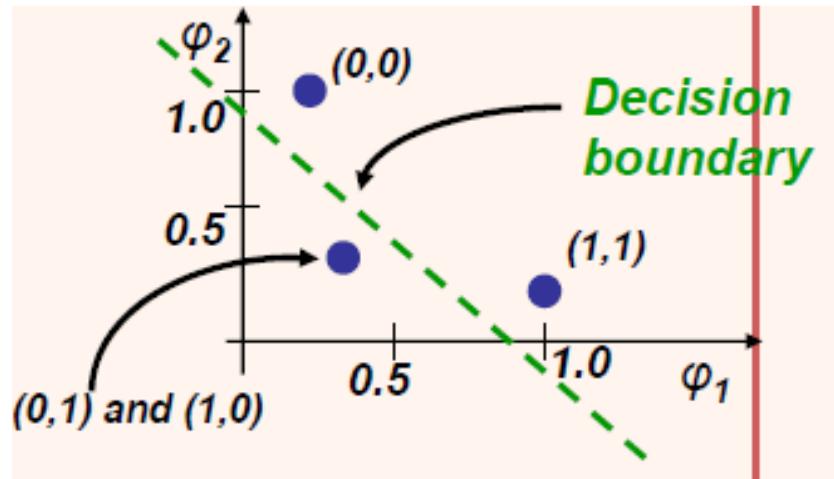
$$\varphi_1(\|x - t_1\|) = e^{-\|x-t_1\|^2}$$

$$\varphi_2(\|x - t_2\|) = e^{-\|x-t_2\|^2}$$

| $x_1$ | $x_2$ |
|-------|-------|
| 0     | 0     |
| 0     | 1     |
| 1     | 0     |
| 1     | 1     |



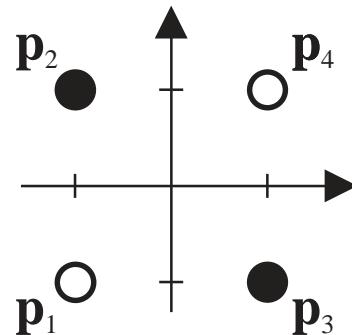
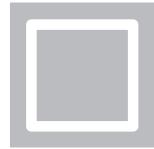
| $\varphi_1(\underline{x})$ | $\varphi_2(\underline{x})$ |
|----------------------------|----------------------------|
| 0.13                       | 1                          |
| 0.36                       | 0.36                       |
| 0.36                       | 0.36                       |
| 1                          | 0.13                       |



$$W^T \varphi(x) > 0 \quad \rightarrow \quad x \in A_1$$

$$W^T \varphi(x) < 0 \quad \rightarrow \quad x \in A_2$$

# Pattern Recognition Problem



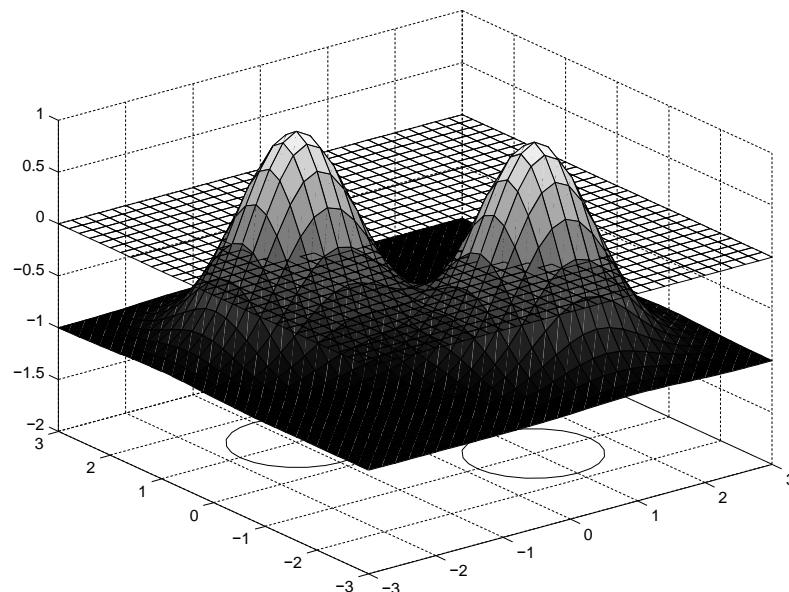
$$\text{Category 1 : } \left\{ \mathbf{p}_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \right\} \quad \text{Category 2 : } \left\{ \mathbf{p}_1 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, \mathbf{p}_4 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}$$

The idea will be to have the network produce outputs greater than zero when the input is near patterns  $\mathbf{p}_2$  or  $\mathbf{p}_3$ , and outputs less than zero for all other inputs.

# Radial Basis Solution

Choose centers at  $\mathbf{p}_2$  and  $\mathbf{p}_3$ :

$$\mathbf{W}^1 = \begin{bmatrix} \mathbf{p}_2^T \\ \mathbf{p}_3^T \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix}$$



Choose bias to be :1

$$\mathbf{b}^1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

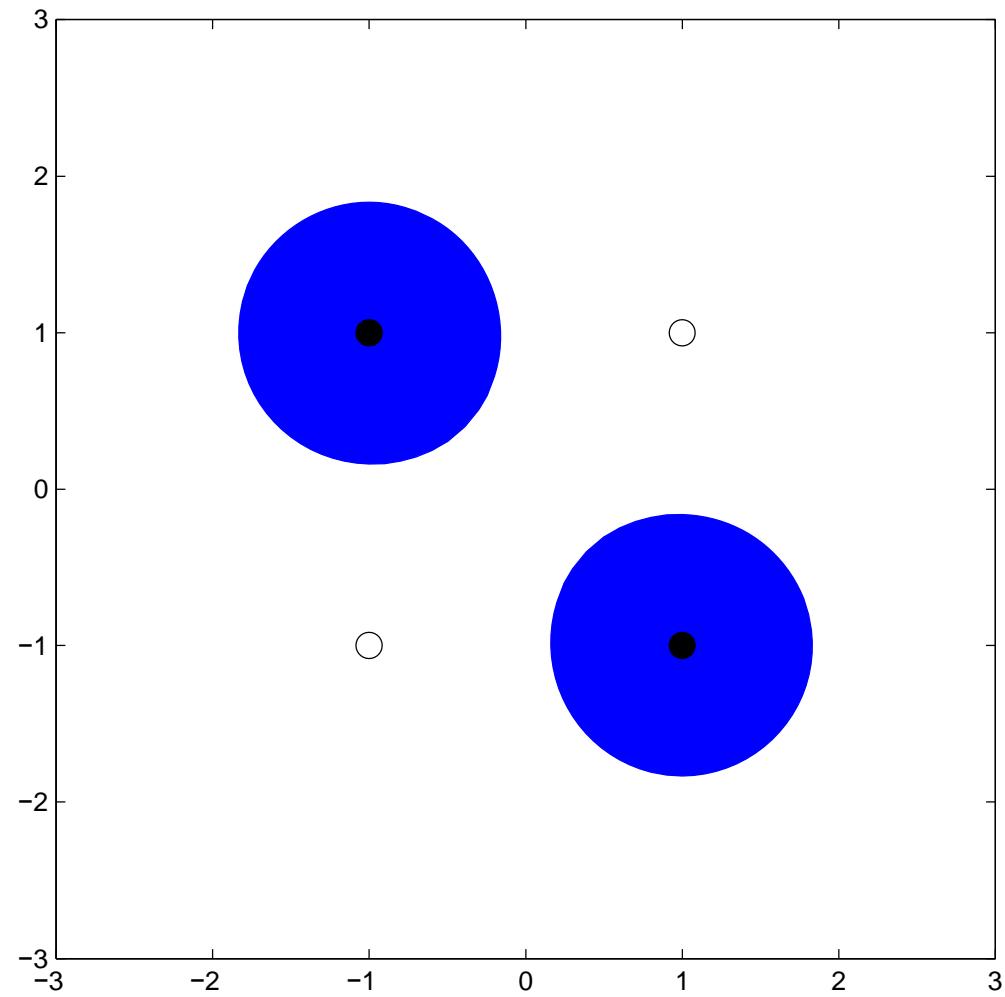
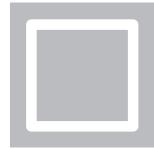
This will cause the following reduction in the basis functions where they meet:

$$a = e^{-n^2} = e^{-(\sqrt{2})^2} = e^{-2} = 0.1353$$

Choose the second layer bias to produce negative outputs, unless we are near  $\mathbf{p}_2$  and  $\mathbf{p}_3$ . Choose second layer weights so that output moves above 0 near  $\mathbf{p}_2$  and  $\mathbf{p}_3$ .

$$\mathbf{W}^2 = [2 \quad 2], b^2 = [-1]$$

# Final Decision Regions



nnd17pc

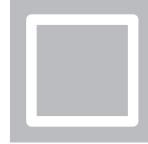
# Global Versus Local

- Multilayer networks create a distributed representation.
  - All sigmoid or linear transfer functions overlap in their activity.
- Radial basis networks create local representations.
  - Each basis function is only active over a small region.
- The global approach requires fewer neurons.  
The local approach is susceptible to the “**curse of dimensionality**.”
- The local approach leads to faster training and is suitable for adaptive methods.

# Radial Basis Training

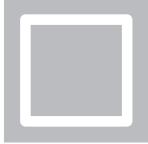
- Backpropagation (gradient-based) algorithms can also be used for radial basis networks.  
In RBF networks there tend to be many more unsatisfactory local minima in the error surfaces than in those of MLP networks. So gradient-based algorithms are often unsatisfactory for the complete training of RBF networks.
- Radial basis network training generally consists of two Stages:
  - 1) During the first stage, the weights and biases in the first layer are set. This can involve unsupervised training or even random selection of the weights.
  - 2) The weights and biases in the second layer are found during the second stage. This usually involves linear least squares, or LMS for adaptive training.

# Radial Basis Training



- The simplest of the two-stage algorithms arranges the centers (first layer weights) in a grid pattern throughout the input range and then chooses a constant bias so that the basis functions have some degree of overlap.
- One of the drawbacks of the RBF network, especially when the centers are selected on a grid, is that they suffer from **the curse of dimensionality**.
- Another method for selecting the centers is to select some random subset of the input vectors in the training set. This procedure is not optimal.
- A more efficient approach is to use a method such as the Kohonen competitive layer or the feature map, described in Chapter 16, to cluster the input space.

# Assume Fixed First Layer



We begin with the case where the first layer weights (centers) are fixed. Assume they are set on a grid, or randomly set. For random weights, the bias can be

$$b_i^1 = \frac{\sqrt{S^1}}{d_{\max}}$$

The training data is given by

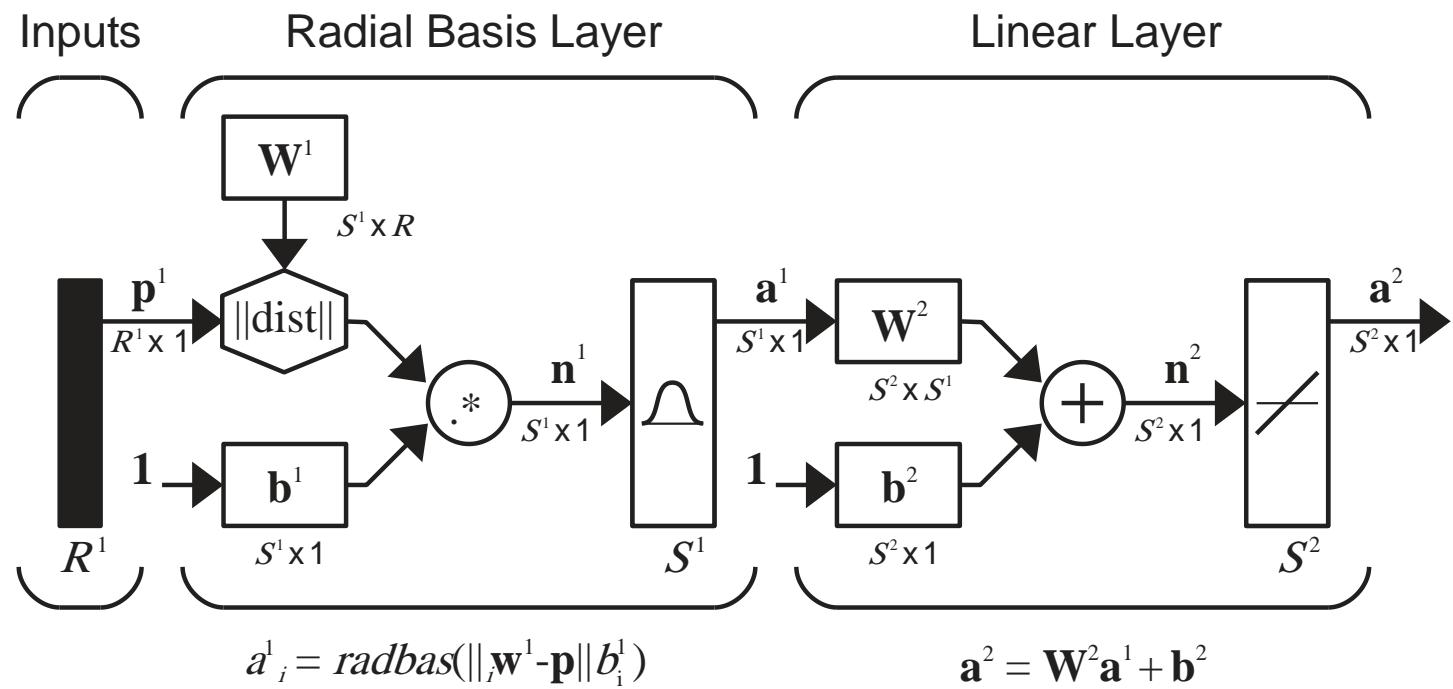
$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \dots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

With first layer weights and biases fixed, the first layer output can be computed:

$$n_{i,q}^1 = \|\mathbf{p}_q - \mathbf{w}^1\| b_i^1 \quad \mathbf{a}_q^1 = \text{radbas}(\mathbf{n}_q^1)$$

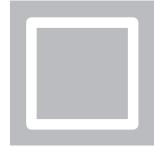
This provides a training set for the second layer:

$$\{\mathbf{a}_1^1, \mathbf{t}_1\}, \{\mathbf{a}_2^1, \mathbf{t}_2\}, \dots, \{\mathbf{a}_Q^1, \mathbf{t}_Q\}$$



$$n_i^1 = \|\mathbf{p} - \mathbf{w}^1\| b_i^1 \quad b = 1 / (\sigma \sqrt{2}) \quad a = f(n) = e^{-n^2}$$

# Linear Least Squares (2<sup>nd</sup> Layer)



$$\mathbf{a}^2 = \mathbf{W}^2 \mathbf{a}^1 + \mathbf{b}^2$$

$$F(\mathbf{x}) = \sum_{q=1}^Q (\mathbf{t}_q - \mathbf{a}_q^2)^T (\mathbf{t}_q - \mathbf{a}_q^2)$$

برای سادگی، خروجی را اسکالر در نظر  
میگیریم:

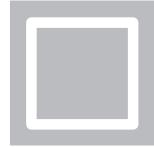
$$\mathbf{x} = \begin{bmatrix} \mathbf{w}^2 \\ b^2 \end{bmatrix}$$

$$\mathbf{z}_q = \begin{bmatrix} \mathbf{a}_q^1 \\ 1 \end{bmatrix}$$

$$a_q^2 = (\mathbf{w}^2)^T \mathbf{a}_q^1 + b^2 = \mathbf{x}^T \mathbf{z}_q$$

$$F(\mathbf{x}) = \sum_{q=1}^Q (\mathbf{t}_q - \mathbf{x}^T \mathbf{z}_q)^T (\mathbf{t}_q - \mathbf{x}^T \mathbf{z}_q)$$

# Matrix Form



$$\mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_Q \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} {}^T \mathbf{u}_1 \\ {}^T \mathbf{u}_2 \\ \vdots \\ {}^T \mathbf{u}_Q \end{bmatrix} = \begin{bmatrix} {}^T \mathbf{z}_1 \\ {}^T \mathbf{z}_2 \\ \vdots \\ {}^T \mathbf{z}_Q \end{bmatrix}, \quad \mathbf{e} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_Q \end{bmatrix}.$$

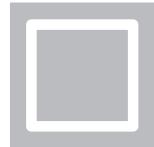
$$\mathbf{e} = \mathbf{t} - \mathbf{Ux} \quad F(\mathbf{x}) = (\mathbf{t} - \mathbf{Ux})^T (\mathbf{t} - \mathbf{Ux})$$

To prevent  
overfitting;  
Regularization

$$F(\mathbf{x}) = \beta E_D + \alpha E_W = \beta \sum_{q=1}^Q (\mathbf{t}_q - \mathbf{a}_q)^T (\mathbf{t}_q - \mathbf{a}_q) + \alpha \sum_{i=1}^n x_i^2,$$

$$\begin{aligned} \rho &= \alpha / \beta \quad F(\mathbf{x}) = (\mathbf{t} - \mathbf{Ux})^T (\mathbf{t} - \mathbf{Ux}) + \rho \sum_{i=1}^n x_i^2 = (\mathbf{t} - \mathbf{Ux})^T (\mathbf{t} - \mathbf{Ux}) + \rho \mathbf{x}^T \mathbf{x} \\ &= \mathbf{t}^T \mathbf{t} - 2 \mathbf{t}^T \mathbf{Ux} + \mathbf{x}^T \mathbf{U}^T \mathbf{Ux} + \rho \mathbf{x}^T \mathbf{x} \\ &= \mathbf{t}^T \mathbf{t} - 2 \mathbf{t}^T \mathbf{Ux} + \mathbf{x}^T [\mathbf{U}^T \mathbf{U} + \rho \mathbf{I}] \mathbf{x} \end{aligned}$$

# Linear Least Squares Solution



$$\begin{aligned}
 F(\mathbf{x}) &= \mathbf{t}^T \mathbf{t} - 2\mathbf{t}^T \mathbf{U} \mathbf{x} + \mathbf{x}^T [\mathbf{U}^T \mathbf{U} + \rho \mathbf{I}] \mathbf{x} \\
 &= c + \mathbf{d}^T \mathbf{x} + \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} \quad (\text{Quadratic Function})
 \end{aligned}$$

$$c = \mathbf{t}^T \mathbf{t}, \mathbf{d} = -2\mathbf{U}^T \mathbf{t} \text{ and } \mathbf{A} = 2[\mathbf{U}^T \mathbf{U} + \rho \mathbf{I}].$$

$$\begin{aligned}
 \nabla F(\mathbf{x}) &= \nabla \left( c + \mathbf{d}^T \mathbf{x} + \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} \right) = \mathbf{d} + \mathbf{A} \mathbf{x} \\
 &= -2\mathbf{U}^T \mathbf{t} + 2[\mathbf{U}^T \mathbf{U} + \rho \mathbf{I}] \mathbf{x} = 0
 \end{aligned}$$

$$[\mathbf{U}^T \mathbf{U} + \rho \mathbf{I}] \mathbf{x}^* = \mathbf{U}^T \mathbf{t}$$

$$\mathbf{x}^* = [\mathbf{U}^T \mathbf{U} + \rho \mathbf{I}]^{-1} \mathbf{U}^T \mathbf{t}$$

# Example

$$g(p) = 1 + \sin\left(\frac{\pi}{4} p\right) \text{ for } -2 \leq p \leq 2$$

$$p = \{-2, -1.2, -0.4, 0.4, 1.2, 2\}$$

$$t = \{0, 0.19, 0.69, 1.3, 1.8, 2\}$$

We will choose the basis function centers to be spaced equally throughout the input range: -2, 0 and 2.

$$\mathbf{W}^1 = \begin{bmatrix} -2 \\ 0 \\ 2 \end{bmatrix}, \mathbf{b}^1 = \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \end{bmatrix}$$

## Example (cont.)

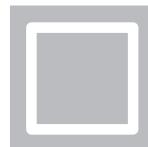
$$n_{i,q}^1 = \|P_{q-i} w^1\| b_i^1 \quad \mathbf{a}_q^1 = \mathbf{radbas}(\mathbf{n}_q^1)$$

$$\mathbf{a}^1 = \left\{ \begin{bmatrix} 1 \\ 0.368 \\ 0.018 \end{bmatrix}, \begin{bmatrix} 0.852 \\ 0.698 \\ 0.077 \end{bmatrix}, \begin{bmatrix} 0.527 \\ 0.961 \\ 0.237 \end{bmatrix}, \begin{bmatrix} 0.237 \\ 0.961 \\ 0.527 \end{bmatrix}, \begin{bmatrix} 0.077 \\ 0.698 \\ 0.852 \end{bmatrix}, \begin{bmatrix} 0.018 \\ 0.368 \\ 1 \end{bmatrix} \right\}$$

$$\mathbf{U}^T = \begin{bmatrix} 1 & 0.852 & 0.527 & 0.237 & 0.077 & 0.018 \\ 0.368 & 0.698 & 0.961 & 0.961 & 0.698 & 0.368 \\ 0.018 & 0.077 & 0.237 & 0.527 & 0.852 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

$$\mathbf{t}^T = [0 \ 0.19 \ 0.69 \ 1.3 \ 1.8 \ 2]$$

# Example (cont.)

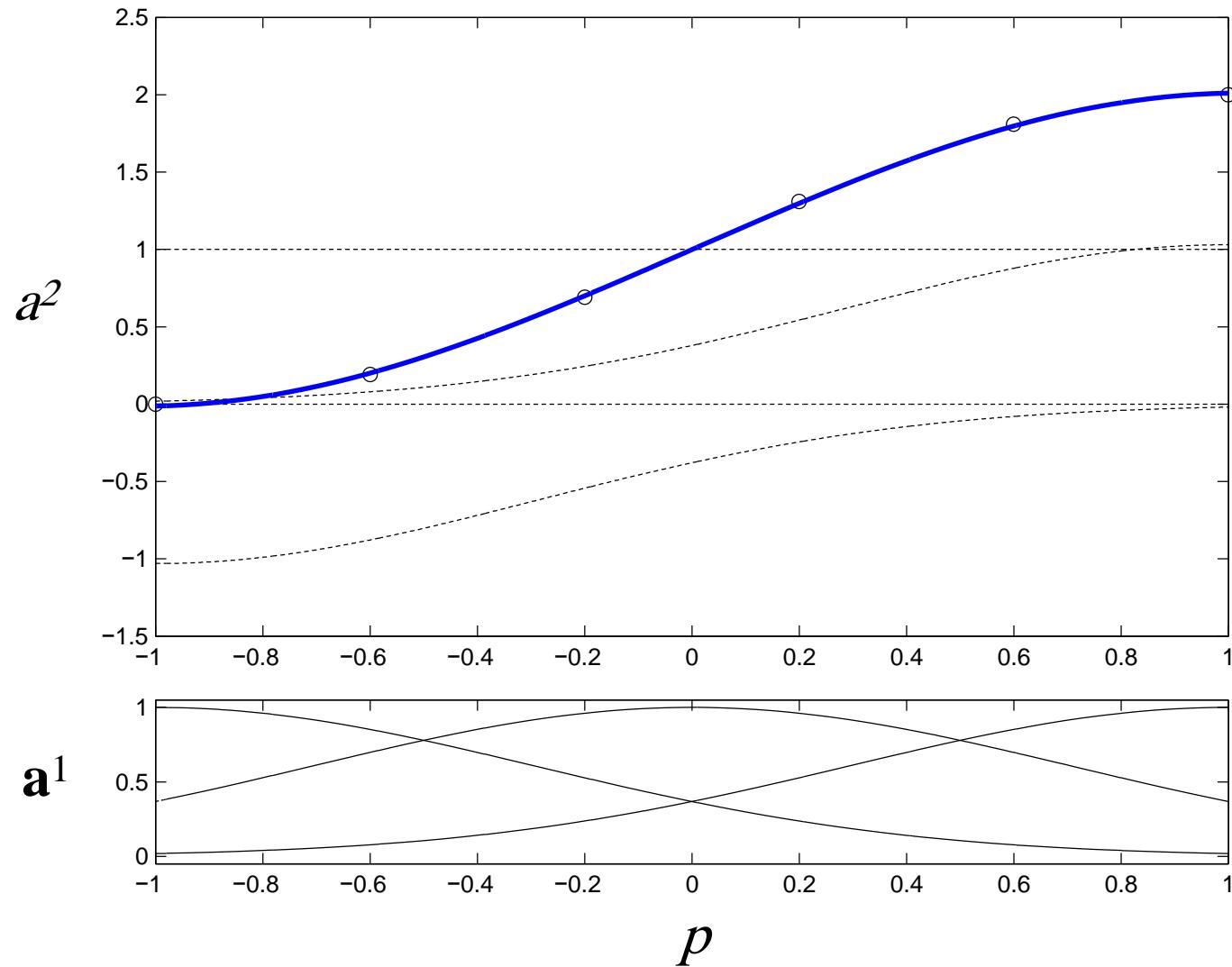


$$\mathbf{x}^* = [\mathbf{U}^T \mathbf{U} + \rho \mathbf{I}]^{-1} \mathbf{U}^T \mathbf{t} \quad \text{Pseudo Inverse}$$

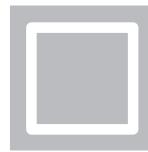
$$\rho=0 \quad \mathbf{x}^* = \begin{bmatrix} 2.07 & 1.76 & 0.42 & 2.71 \\ 1.76 & 3.09 & 1.76 & 4.05 \\ 0.42 & 1.76 & 2.07 & 2.71 \\ 2.71 & 4.05 & 2.71 & 6 \end{bmatrix}^{-1} \begin{bmatrix} 1.01 \\ 4.05 \\ 4.41 \\ 6 \end{bmatrix} = \begin{bmatrix} -1.03 \\ 0 \\ 1.03 \\ 1 \end{bmatrix}$$

$$\mathbf{W}^2 = \begin{bmatrix} -1.03 & 0 & 1.03 \end{bmatrix}, \mathbf{b}^2 = \begin{bmatrix} 1 \end{bmatrix}.$$

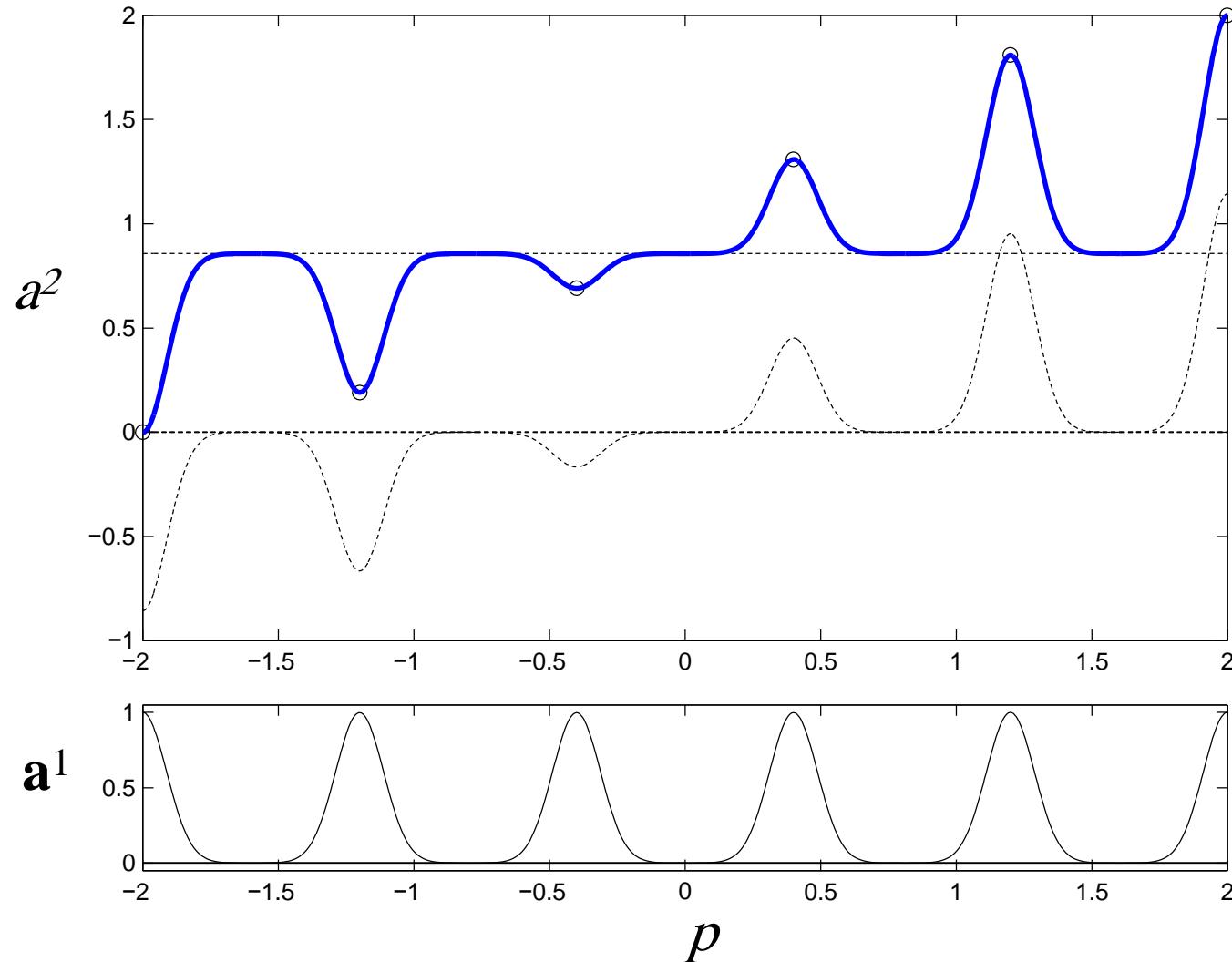
# Example (cont.)



# Bias Too Large



six basis functions and six data points,



$$\mathbf{b}^1 = \begin{bmatrix} 8 \\ 8 \\ 8 \end{bmatrix}$$

nnd17lls

## استفاده از الگوریتم LMS برای تعیین وزن های لایه دوم

$$F(\mathbf{x}) = \sum_{q=1}^Q (\mathbf{t}_q - \mathbf{a}_q^2)^T (\mathbf{t}_q - \mathbf{a}_q^2)$$

$$\mathbf{x} = \begin{bmatrix} {}_1\mathbf{w}^2 \\ b^2 \end{bmatrix} \quad \mathbf{z}_q = \begin{bmatrix} \mathbf{a}_q^1 \\ 1 \end{bmatrix} \quad a_q^2 = ({}_1\mathbf{w}^2)^T \mathbf{a}_q^1 + b^2 = \mathbf{x}^T \mathbf{z}_q$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \nabla F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}_k}$$

$$\hat{F}(\mathbf{x}) = (t(k) - a(k))^2 = e^2(k)$$

$$\hat{\nabla} F(\mathbf{x}) = \nabla e^2(k) = -2e(k)\mathbf{z}(k)$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + 2\alpha e(k)\mathbf{z}(k)$$

- Cluster the input space using a competitive layer (or SOFM).  
Use the cluster centers as basis function centers.

Training set  $\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \dots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$  Clustering on  $\{\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_Q\}$

$${}_{i^*}\mathbf{w}^1(q) = {}_{i^*}\mathbf{w}^1(q-1) + \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{w}^1(q-1)) \text{ the Kohonen learning rule}$$

For each neuron (basis function), locate the  $n_c$  input vectors from the training set that are closest to the corresponding weight vector (center).

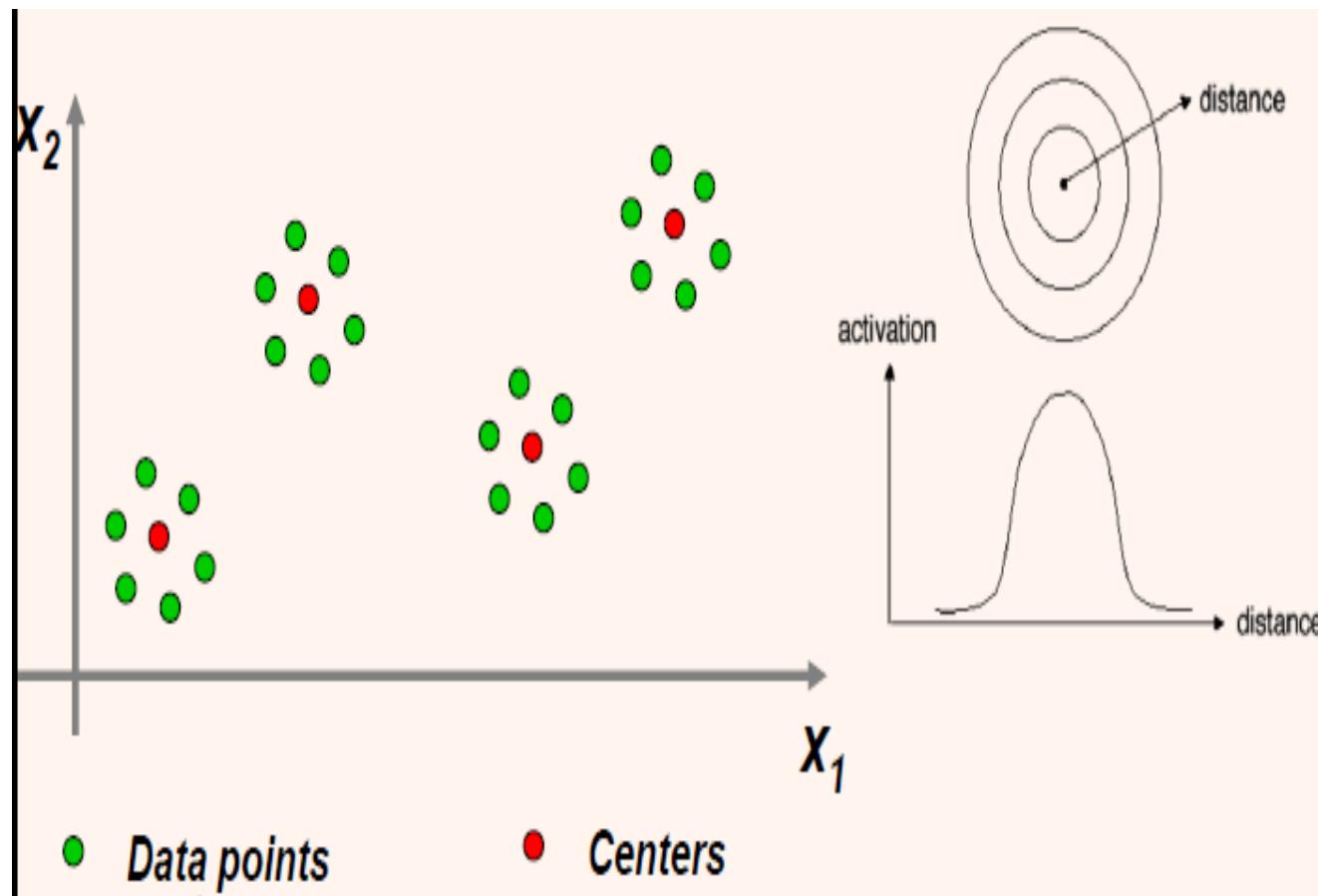
- The bias can be computed from the variation in each cluster:

the average  
distance between  
the center and its  
neighbors:

$$dist_i = \frac{1}{n_c} \left( \sum_{j=1}^{n_c} \left\| \mathbf{p}_j^i - {}_i\mathbf{w}^1 \right\|^2 \right)^{\frac{1}{2}}$$

$$b_i^1 = \frac{1}{\sqrt{2} dist_i}$$

# Clustering

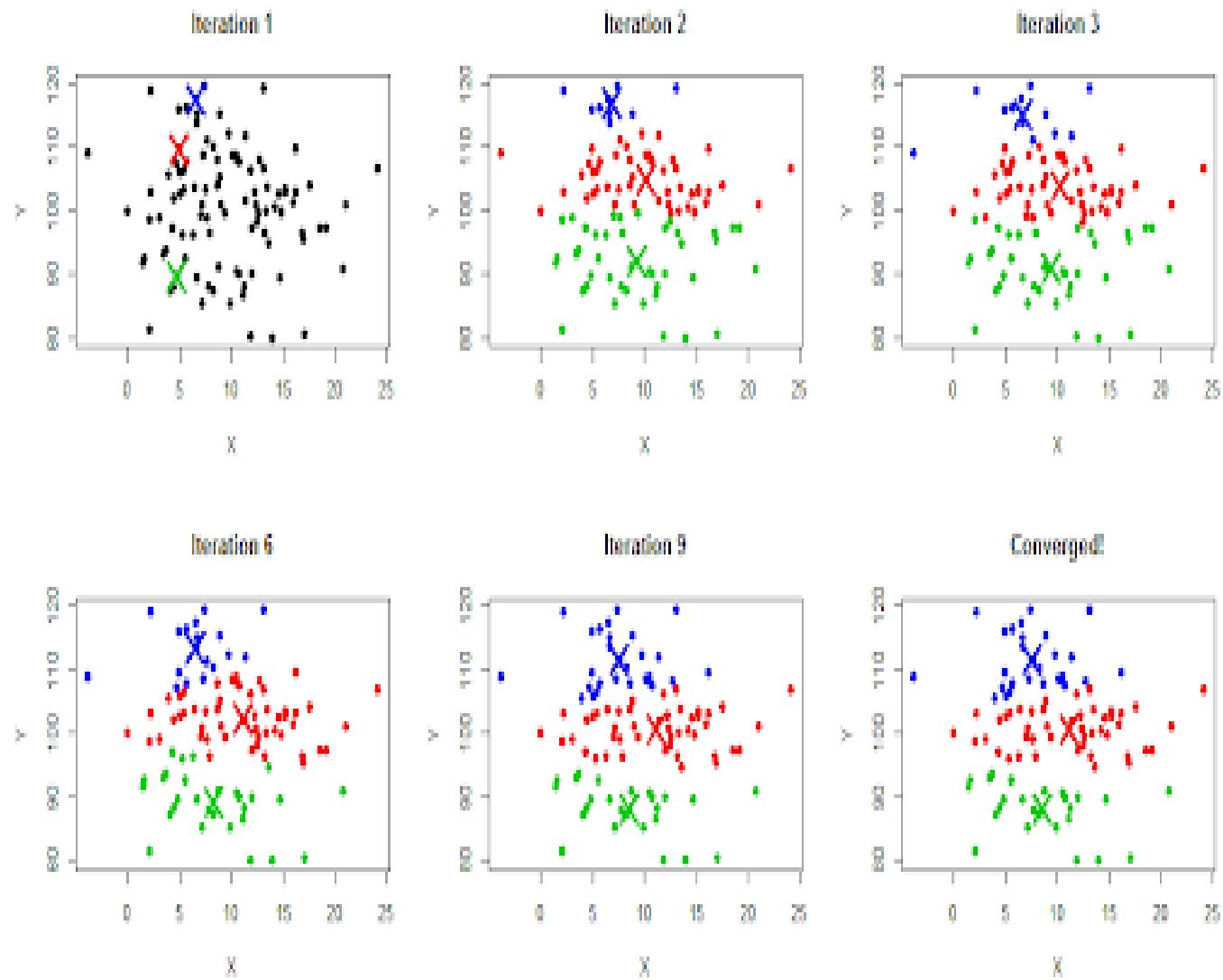


## روش دیگر برای تعیین وزن های لایه اول(مراکز توابع گوسین)

### الگوریتم k-means:(ویکی پدیا)

۱. ابتدا  $k$  میانگین یعنی  $\left(\mu_1^{(0)}, \mu_2^{(0)}, \dots, \mu_k^{(0)}\right)$  را که نماینده خوشها هستند، بصورت تصادفی مقدار دهی می‌کنیم.
۲. سپس، این دو مرحله بین را به تناوب چندین بار اجرا می‌کنیم تا میانگین‌ها به یک ثبات کافی برسند و یا مجموع واریانس‌های خوشها تغییر چندانی نکند:
  - از میانگین‌ها  $k$  خوش می‌سازیم، خوش نام در زمان  $t$  تمام داده‌هایی هستند که از لحاظ اقلیدسی کمترین فاصله را با میانگین  $\mu_i^{(t)}$  یعنی میانگین نام در زمان  $t$  دارند.<sup>[V]</sup> به زبان ریاضی خوش نام در زمان  $t$  برابر خواهد بود با:
$$S_i^{(t)} = \{x_p : \|x_p - \mu_i^{(t)}\|^2 \leq \|x_p - \mu_j^{(t)}\|^2 \forall j, 1 \leq j \leq k\}.$$
  - ۳. حال میانگین‌ها را بر اساس این خوش‌های جدید به این شکل بروز می‌کنیم:<sup>[A]</sup>
$$\mu_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j.$$
  - ۴. در نهایت میانگین‌های مرحله آخر (در زمان  $T$ ) یعنی  $\left(\mu_1^{(T)}, \mu_2^{(T)}, \dots, \mu_k^{(T)}\right)$  خوشها را نمایندگی خواهند کرد.

منظور از  $\mu_i$  میانگین خوش  $S_i$  و  $|S_i|$  تعداد اعضای خوش نام است.

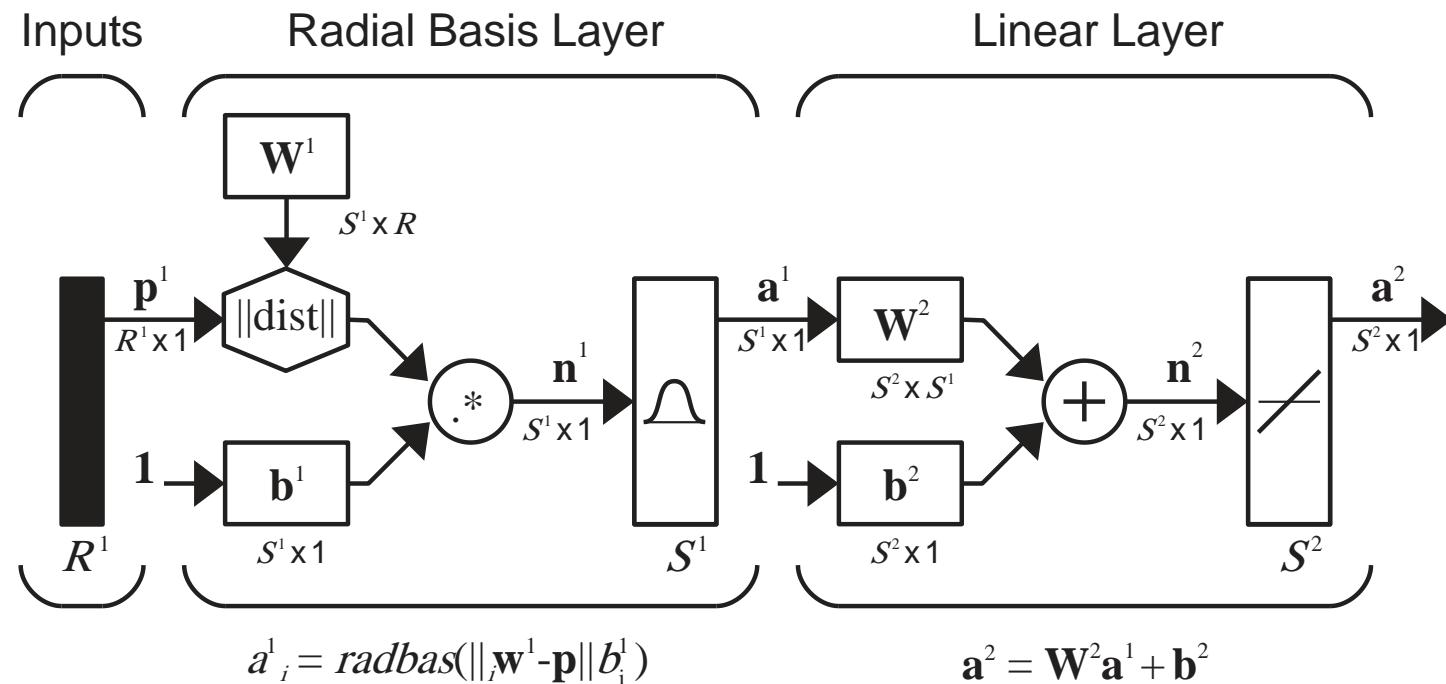


## Backpropagation (gradient method)



The derivation of the gradient for RBF networks follows the same pattern as the gradient development for MLP networks.

$$\hat{F}(\mathbf{x}) = (t(k) - a(k))^2 = e^2(k)$$



$$n_i^1 = \|\mathbf{p} - \mathbf{w}^1\| b_i^1$$

$$b = 1 / (\sigma \sqrt{2})$$

$$a = f(n) = e^{-n^2}$$

nnd17no

# Differences

$$n_i^1 = \|\mathbf{p} - {}_i\mathbf{w}^1\| b_i^1 = b_i^1 \sqrt{\sum_{j=1}^{S^1} (p_j - w_{i,j}^1)^2}$$

$$\frac{\partial n_i^1}{\partial w_{i,j}^1} = b_i^1 \frac{1/2}{\sqrt{\sum_{j=1}^{S^1} (p_j - w_{i,j}^1)^2}} 2(p_j - w_{i,j}^1)(-1) = \frac{b_i^1 (w_{i,j}^1 - p_j)}{\|\mathbf{p} - {}_i\mathbf{w}^1\|}, \quad \frac{\partial n_i^1}{\partial b_i^1} = \|\mathbf{p} - {}_i\mathbf{w}^1\|$$

$$\frac{\partial \hat{F}}{\partial w_{i,j}^1} = s_i^1 \frac{b_i^1 (w_{i,j}^1 - p_j)}{\|\mathbf{p} - {}_i\mathbf{w}^1\|}, \quad \frac{\partial \hat{F}}{\partial b_i^1} = s_i^1 \|\mathbf{p} - {}_i\mathbf{w}^1\|$$

Classification using RBF with distance = -6, radius = 10, and width = 6

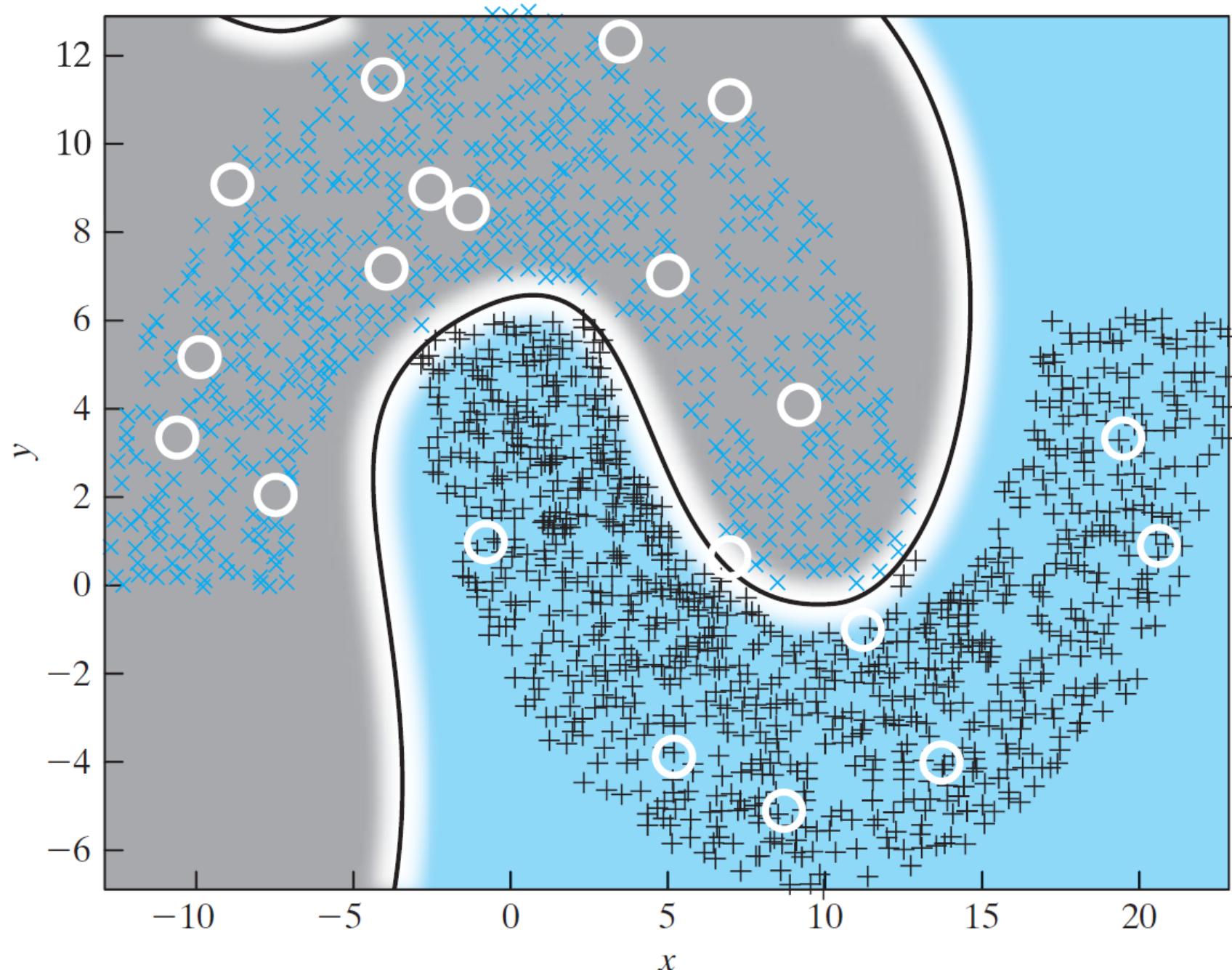
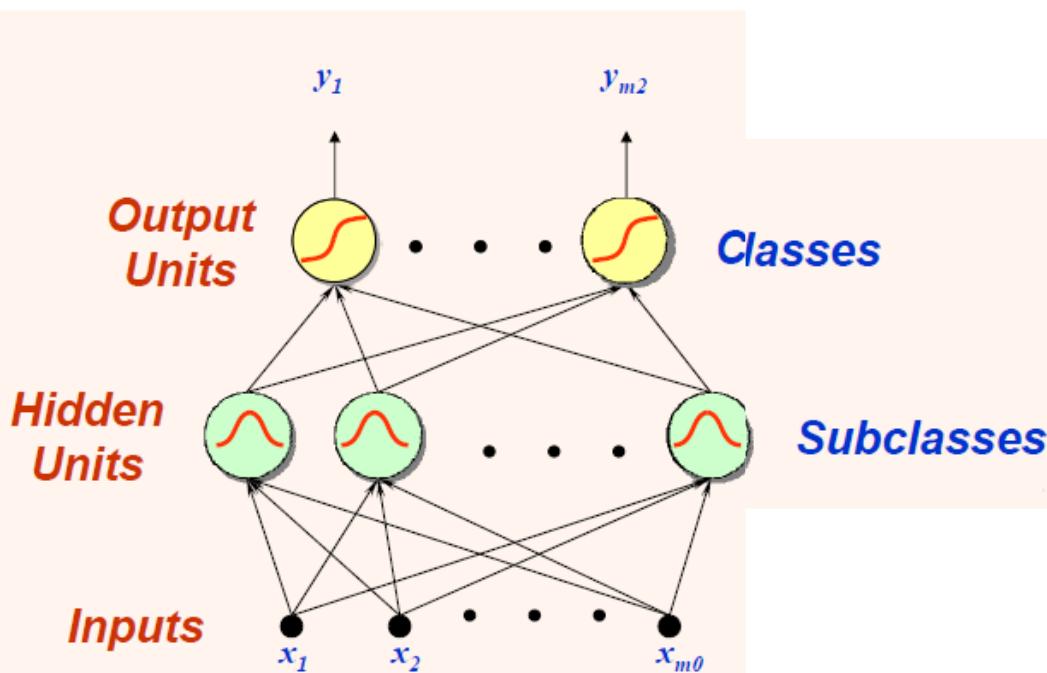


FIGURE 5.6 RBF network trained with K-means and RLS algorithms for distanced  $d=-6$ .

# ساختارهای متداول برای شبکه RBF

ساختار شبکه برای classification



ساختار شبکه برای تخمین تابع

