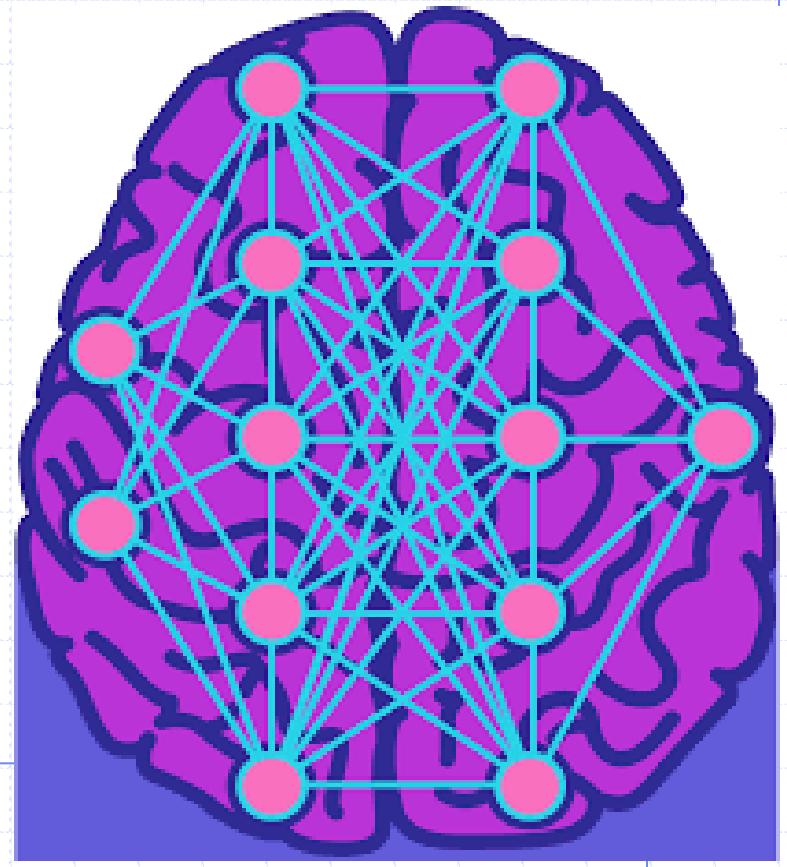
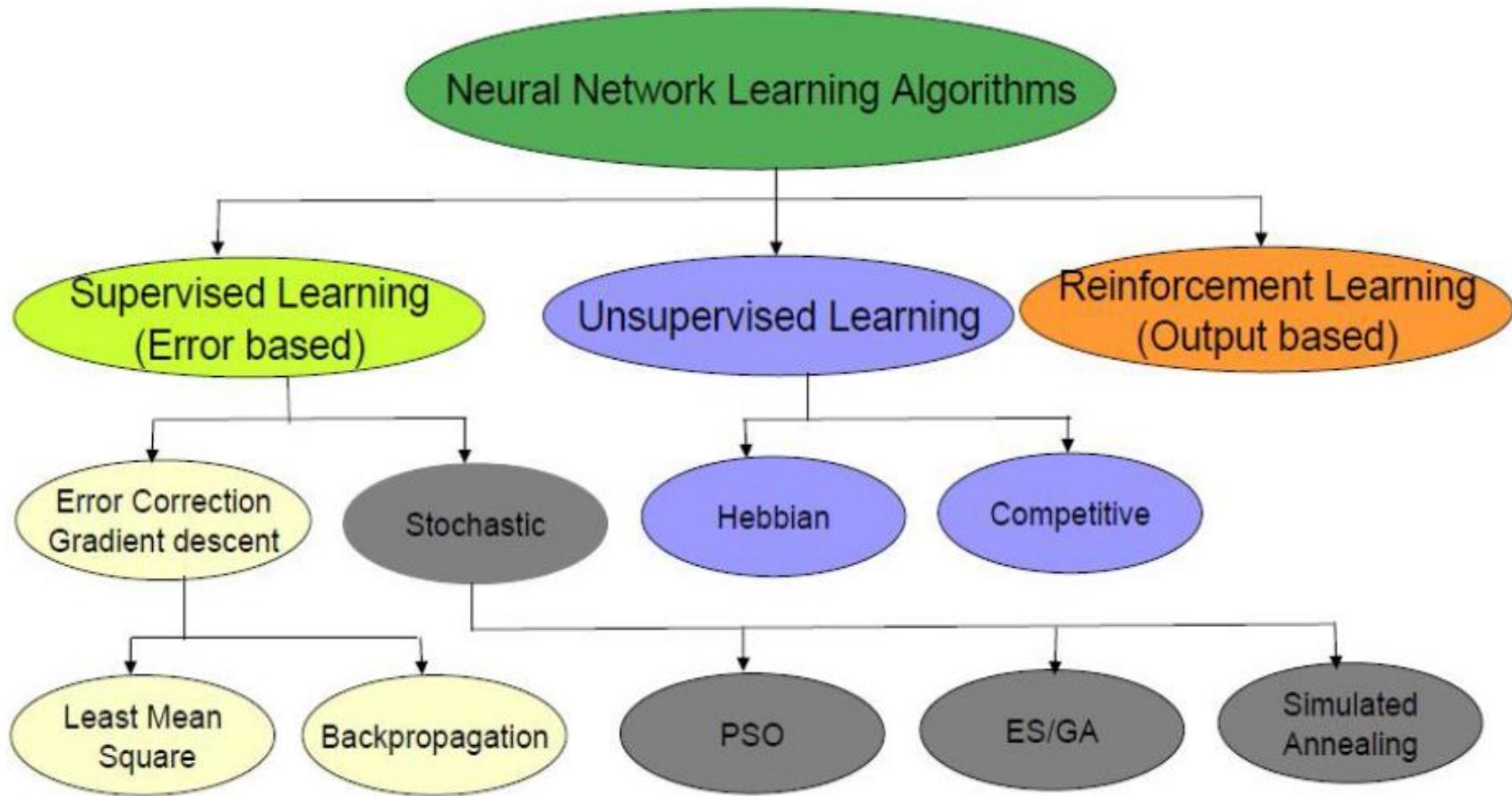


ITF309

Artificial Neural Networks



Classification of Learning Algorithms



CHAPTER 10

Widrow-Hoff *Learning*

Adaline Network

(ADApative LInear NEuron)

Adaline Inventors

Bernard Widrow and Marcian (Ted) Hoff



Bernard Widrow,
Professor Emeritus of E.E.,
Stanford University



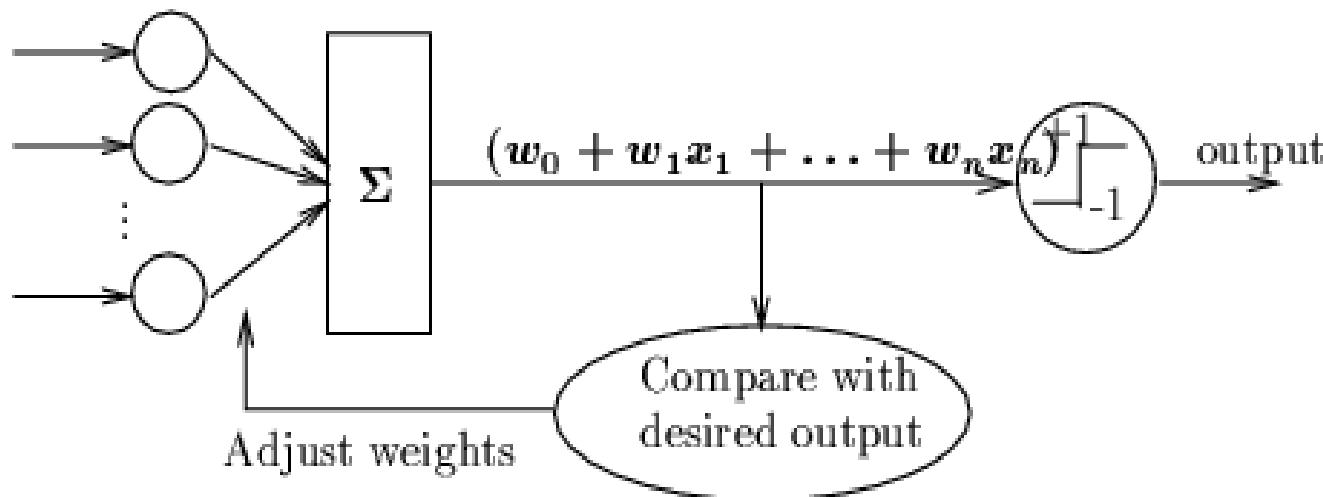
Marcian Hoff
Co-inventor of Patent 3,821,715
*Microprocessor Concept and
Architecture*

Objectives

- ◆ Widrow-Hoff learning is an **approximate steepest descent algorithm**, in which the **performance index** is **mean square error**.
- ◆ It is widely used today in many **signal processing applications**.
- ◆ It is precursor to the **backpropagation algorithm** for **multilayer networks**.

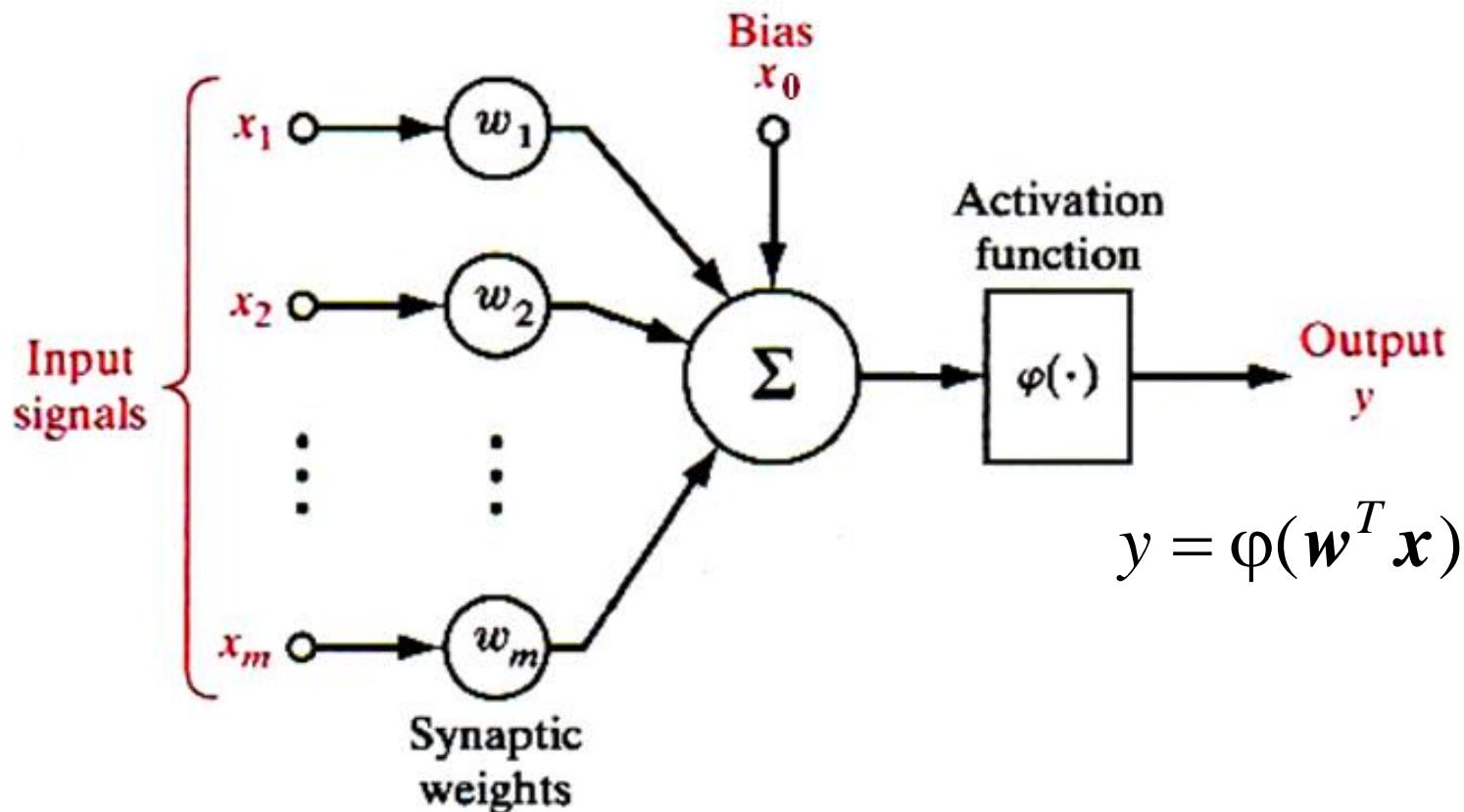
Adaline

- By Widrow and Hoff (~1960)
 - The same architecture of perceptrons



- Learning method: **delta rule** (another way of error driven), also called Widrow-Hoff learning rule
Try to reduce the mean squared error (MSE) between the net input and the desired output

Neuron model:



Adaline: Neuron model with linear active function

$$\varphi(x) = x \quad \therefore y = \varphi(w^T x) = w^T x$$

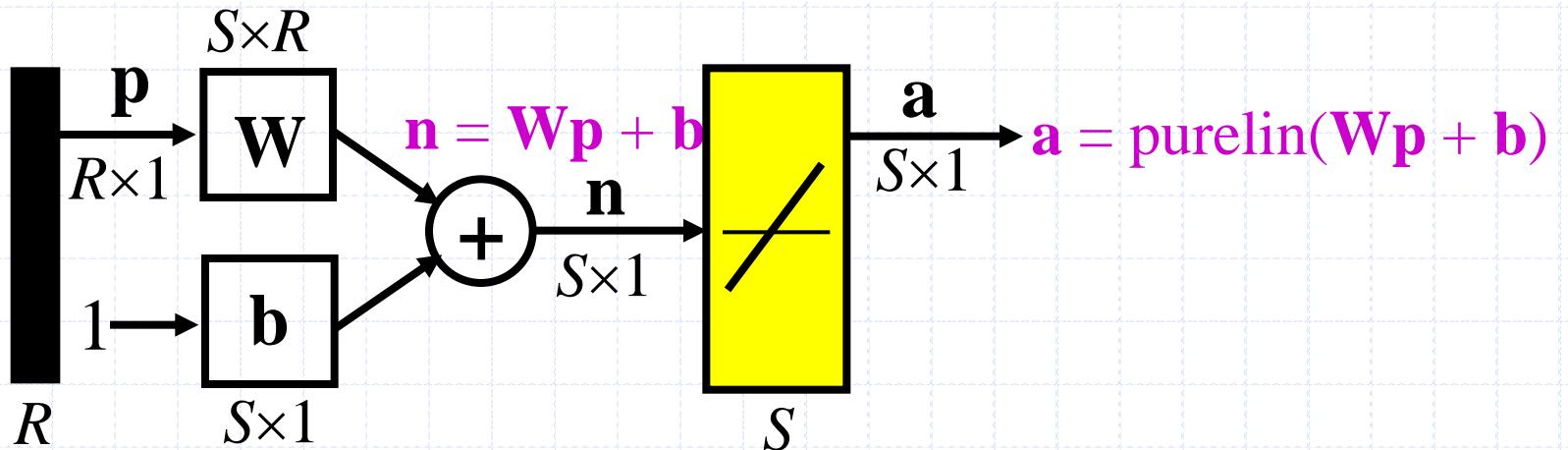
Alternate Rule Names

- Because the Adaline rule minimizes MSE, it is sometimes called the **“LMS rule”** [LMS = “least mean square”].
- The term **“Delta rule”** is also sometimes used, although this will be seen to be a rule for a more general class of networks.
- The **“Widrow-Hoff”** rule is also used.

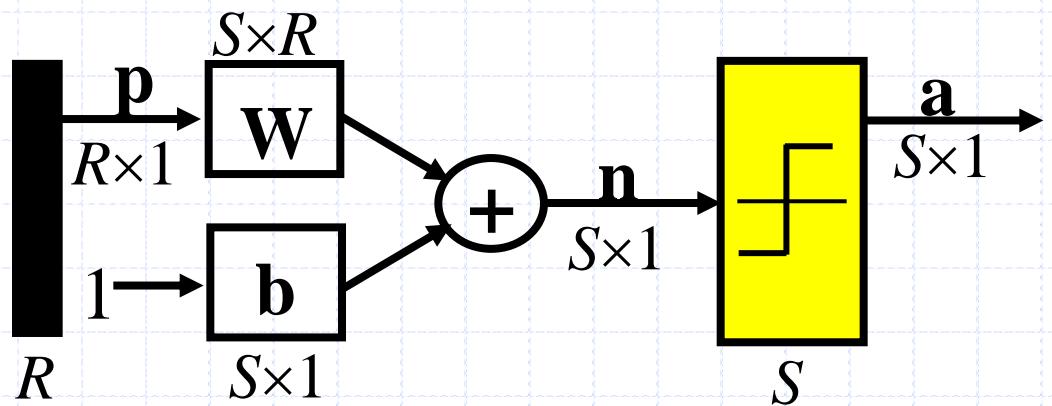
ADALINE Network

- ◆ **ADALINE (Adaptive Linear Neuron) network** and its learning rule, **LMS (Least Mean Square)** algorithm are proposed by Widrow and Marcian Hoff in 1960.
- ◆ Both **ADALINE network** and the **perceptron** suffer from the same inherent limitation: they can only solve **linearly separable problems.**
- ◆ The **LMS algorithm** minimizes mean square error (MSE), and therefore tries to *move the decision boundaries as far from the training patterns as possible.*

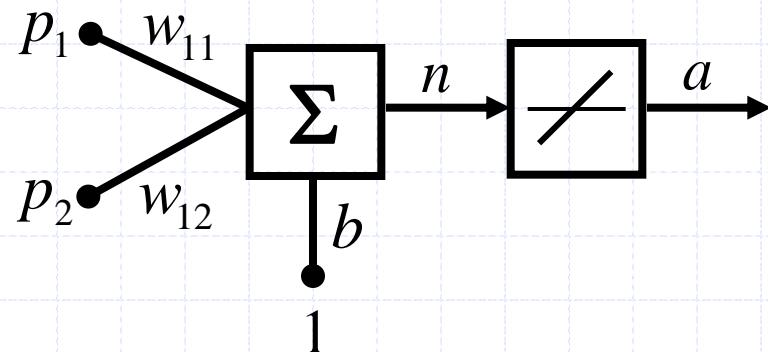
ADALINE Network



Single-layer
perceptron

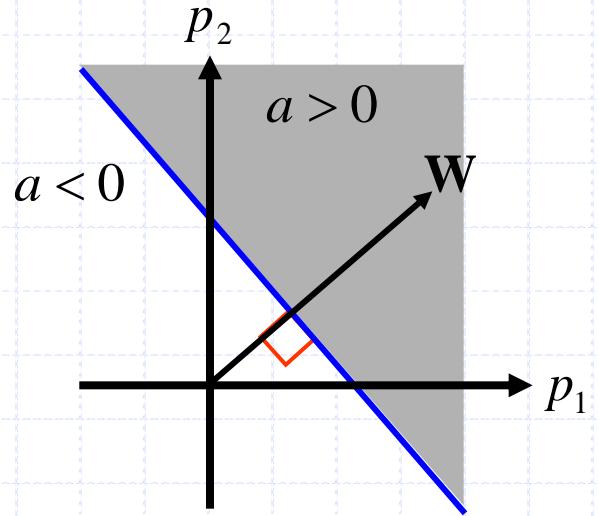


Single ADALINE



$$\mathbf{p} = \begin{bmatrix} p_1 \\ p_2 \end{bmatrix}, \mathbf{W} = \begin{bmatrix} w_{11} & w_{12} \end{bmatrix}$$
$$\Rightarrow n = \mathbf{W}\mathbf{p} + b$$
$$a = \text{purelin}(n) = n$$

- ◆ Set $n = 0$, then $\mathbf{W}\mathbf{p} + b = 0$ specifies a **decision boundary**.
- ◆ The **ADALINE** can be used to classify objects into **two categories** if they are **linearly separable**.



Mean Square Error

- ◆ The LMS algorithm is an example of supervised training.
- ◆ The LMS algorithm will adjust the weights and biases of the ADALINE in order to minimize the mean square error, where the error is the difference between the target output (t_q) and the network output (p_q).

$$\mathbf{x} = \begin{bmatrix} {}^1\mathbf{w} \\ b \end{bmatrix}, \mathbf{z} = \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} \Rightarrow a = {}^1\mathbf{w}^T \mathbf{p} + b = \mathbf{x}^T \mathbf{z}$$

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

$E[\cdot]$: expected value

Performance Optimization

- ◆ Develop algorithms to **optimize** a performance index $F(\mathbf{x})$, where the word “optimize” will mean to find the value of \mathbf{x} that **minimizes** $F(\mathbf{x})$.
- ◆ The optimization algorithms are **iterative** as

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

\mathbf{p}_k : a search direction

α_k : positive learning rate, which determines the length of the step

\mathbf{x}_0 : initial guess

Taylor Series Expansion

◆ Taylor series:

◆ Vector case:

Gradient & Hessian

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

◆ Gradient: $\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) & \frac{\partial}{\partial x_2} F(\mathbf{x}) & \cdots & \frac{\partial}{\partial x_n} F(\mathbf{x}) \end{bmatrix}^T$

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

Steepest Descent

- ◆ Goal: The function $F(\mathbf{x})$ can **decrease** at each iteration, i.e.,
$$F(\mathbf{x}_{k+1}) < F(\mathbf{x}_k)$$

- ◆ Central idea: first-order Taylor series expansion

$$F(\mathbf{x}_{k+1}) = F(\mathbf{x}_k + \Delta\mathbf{x}) \approx F(\mathbf{x}_k) + \mathbf{g}_k^T \mathbf{x}_k, \quad \boxed{\mathbf{g}_k \equiv \nabla F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}_k}}$$

$$F(\mathbf{x}_{k+1}) < F(\mathbf{x}_k) \Rightarrow \mathbf{g}_k^T \Delta\mathbf{x}_k = \alpha_k \mathbf{g}_k^T \mathbf{p}_k < 0 \Rightarrow \mathbf{g}_k^T \mathbf{p}_k < 0$$

- ◆ Any vector \mathbf{p}_k that satisfies $\mathbf{g}_k^T \mathbf{p}_k < 0$ is called a **descent direction**.

- ◆ A vector that points in the **steepest descent direction** is

- ◆ **Steepest descent:**

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

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

$$\Delta\mathbf{x} \propto -\mathbf{g}_k$$

Delta Learning Rule

- ◆ ADALINE: $a = \phi(\mathbf{z}, \mathbf{x}) = \sum_j^R w_j p_j + b = \mathbf{x}^T \mathbf{z}$ $\begin{cases} \mathbf{x} = [\mathbf{w} \quad b]^T \\ \mathbf{z} = [\mathbf{p} \quad 1]^T \end{cases}$
- ◆ Least-Squares-Error Criterion:

$$\text{minimize } E = \frac{1}{2} (t - a)^2$$

$$\text{◆ Gradient: } \frac{\partial E}{\partial w_j} = \frac{\partial E}{\partial a} \frac{\partial a}{\partial w_j} = -(t - a) p_j$$

$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial a} \frac{\partial a}{\partial b} = -(t - a)$$

- ◆ Delta learning rule:

$$\Delta \mathbf{x} \propto -\frac{\partial E}{\partial \mathbf{x}}$$

$$w_j(k+1) = w_j(k) + \alpha(t - a)p_j$$

$$b(k+1) = b(k) + \beta(t - a)$$

LMS Algorithm

- ◆ LMS algorithm is to **locate the minimum point**.
- ◆ Use an **approximate steepest descent algorithm** to estimate the **gradient**.
- ◆ Estimate the mean square error $F(\mathbf{x})$ by
$$\hat{F}(\mathbf{x}) = (t(k) - a(k))^2 = e^2(k)$$
- ◆ Estimated gradient: $\hat{\nabla}F(\mathbf{x}) = \nabla e^2(k)$

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

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

LMS Algorithm

- * $a(k) = \mathbf{W}\mathbf{p}(k) + b = [w_1 \quad w_2 \quad \cdots \quad w_R] \begin{bmatrix} p_1(k) \\ p_2(k) \\ \vdots \\ p_R(k) \end{bmatrix} = \sum_{i=1}^R w_i p_i(k) + b$
- * $e(k) = t(k) - a(k) = t(k) - \left(\sum_{i=1}^R w_i p_i(k) + b \right)$
- * $\frac{\partial e(k)}{\partial w_j} = -p_j(k), \quad \frac{\partial e(k)}{\partial b} = -1$
- * $\hat{\nabla}F(\mathbf{x}) = \nabla e^2(k) = -2e(k)\mathbf{z}(k)$

LMS Algorithm

- * The steepest descent algorithm with constant learning rate α is $\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \nabla F(\mathbf{x}) \Big|_{\mathbf{x}=\mathbf{x}_k}$
- * $\hat{\nabla}F(\mathbf{x}) = \nabla e^2(k) = -2e(k)\mathbf{z}(k) \Rightarrow \mathbf{x}_{k+1} = \mathbf{x}_k + 2\alpha e(k)\mathbf{z}(k)$

* Matrix notation of LMS algorithm:

$$\begin{aligned}\mathbf{W}(k+1) &= \mathbf{W}(k) + 2\alpha \mathbf{e}(k) \mathbf{p}^T(k) \\ \mathbf{b}(k+1) &= \mathbf{b}(k) + 2\alpha \mathbf{e}(k)\end{aligned}$$

- * The LMS algorithm is also referred to as the delta rule or the Widrow-Hoff learning algorithm.

Orange/Apple Example

Start, arbitrary, with **all the weights set to zero**, and then will apply input p_1, p_2, p_1, p_2 , etc., in that order, calculating the new weights after each input is presented.

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



$$a(0) = \mathbf{W}(0)\mathbf{p}(0) = \mathbf{W}(0)\mathbf{p}_1 = 0 \Rightarrow e(0) = t(0) - a(0) = t_1 - a(0) = -1$$



Orange/Apple Example

$$a(2) = \mathbf{W}(2)\mathbf{p}(2) = \mathbf{W}(2)\mathbf{p}_1 = -0.64$$

$$\Rightarrow e(2) = t(2) - a(2) = t_1 - a(2) = -0.36$$

$$\mathbf{W}(3) = \mathbf{W}(2) + 2\alpha e(2)\mathbf{p}^T(2) = [0.0160 \quad 1.1040 \quad -0.0160].$$

$$\boxed{\mathbf{W}(\infty) = [0 \quad 1 \quad 0].}$$

- * This decision boundary **falls halfway** between the two reference patterns. The **perceptron rule** did NOT produce such a boundary,
- * The **perceptron rule** stops as soon as the patterns are **correctly classified**, even though some patterns may be close to the boundaries. The **LMS algorithm minimizes the mean square error.**

Solved Problem P10.2

*Category I: $\mathbf{p}_1 = [1 \ 1]^T, \mathbf{p}_2 = [-1 \ -1]^T$

Category II: $\mathbf{p}_3 = [2 \ 2]^T$

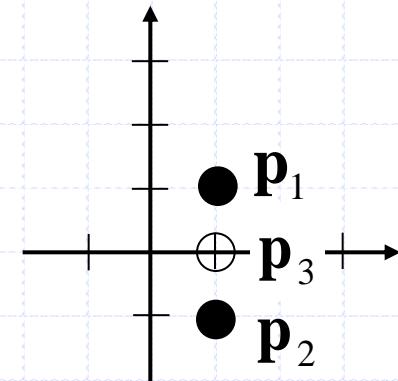
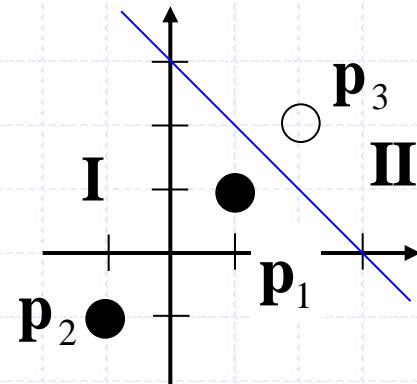
Since they are linear separable, we can design an ADALINE network to make such a distinction.

As shown in figure, $b = 3$, $w_{11} = -1$, $w_{12} = -1$

*Category III: $\mathbf{p}_1 = [1 \ 1]^T, \mathbf{p}_2 = [1 \ -1]^T$

Category IV: $\mathbf{p}_3 = [1 \ 0]^T$

They are NOT linear separable, so an ADALINE network CANNOT distinguish between them.



Solved Problem P10.4

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

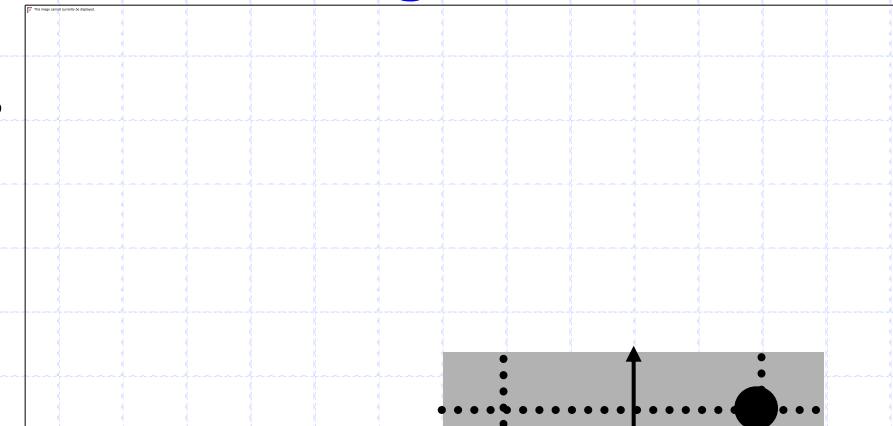
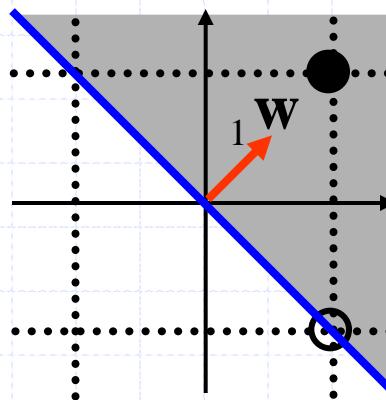
* Train the network using the LMS algorithm, with the initial guess set to zero and a learning rate $\alpha = 0.25$.

$$a(0) = \text{purelin} \left[\begin{bmatrix} 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right] = 0,$$

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

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

$$= \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

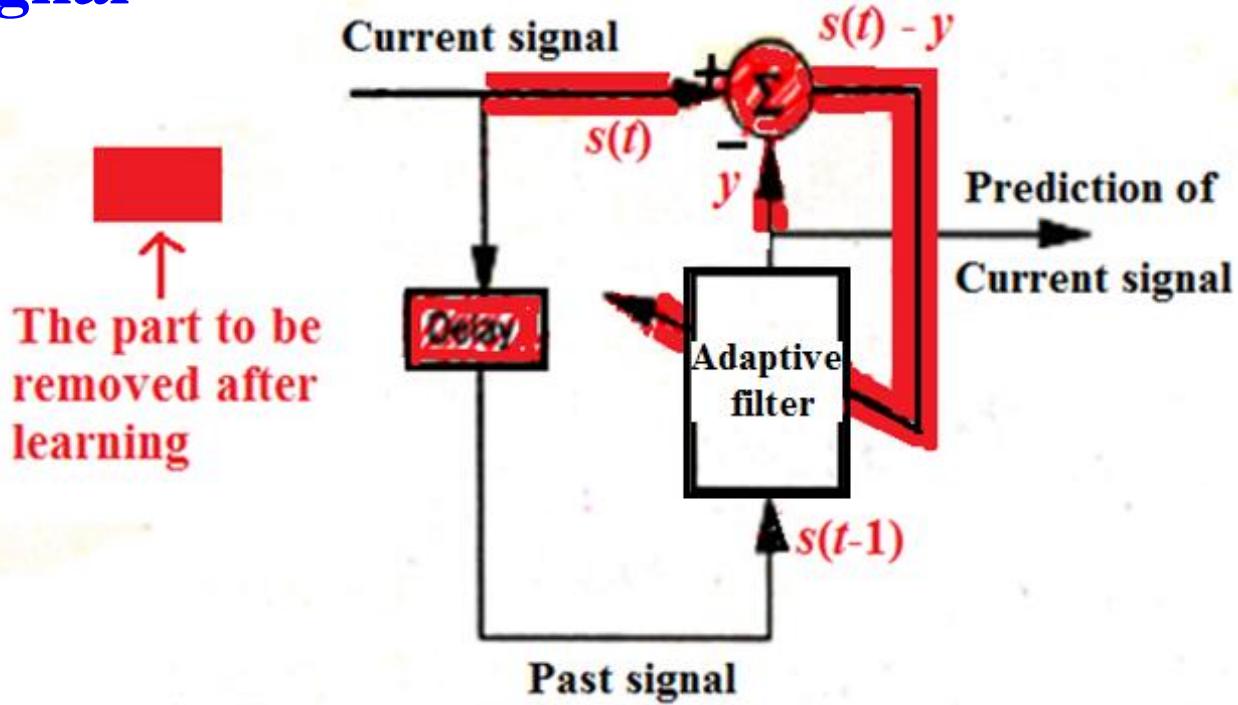


Applications

- ◆ **Noise** cancellation system to remove 60-Hz noise from EEG signal (Fig. 10.6)
- ◆ **Echo** cancellation system in long distance telephone lines (Fig. 10.10)
- ◆ Filtering engine noise from pilot's voice signal (Fig. P10.8)

Applications

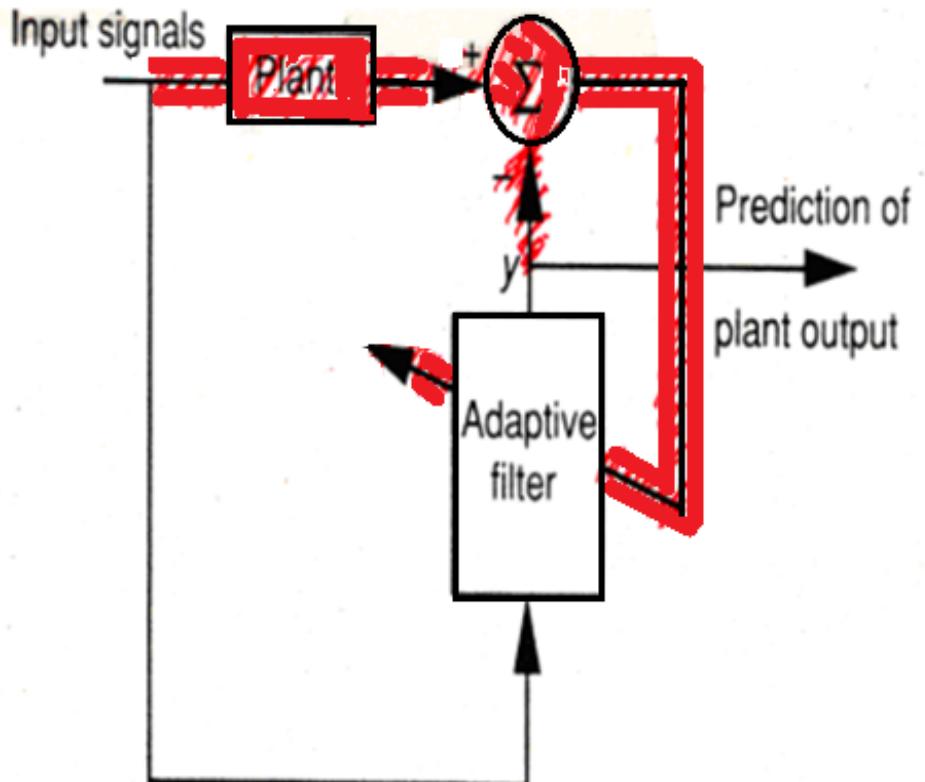
Predict Signal



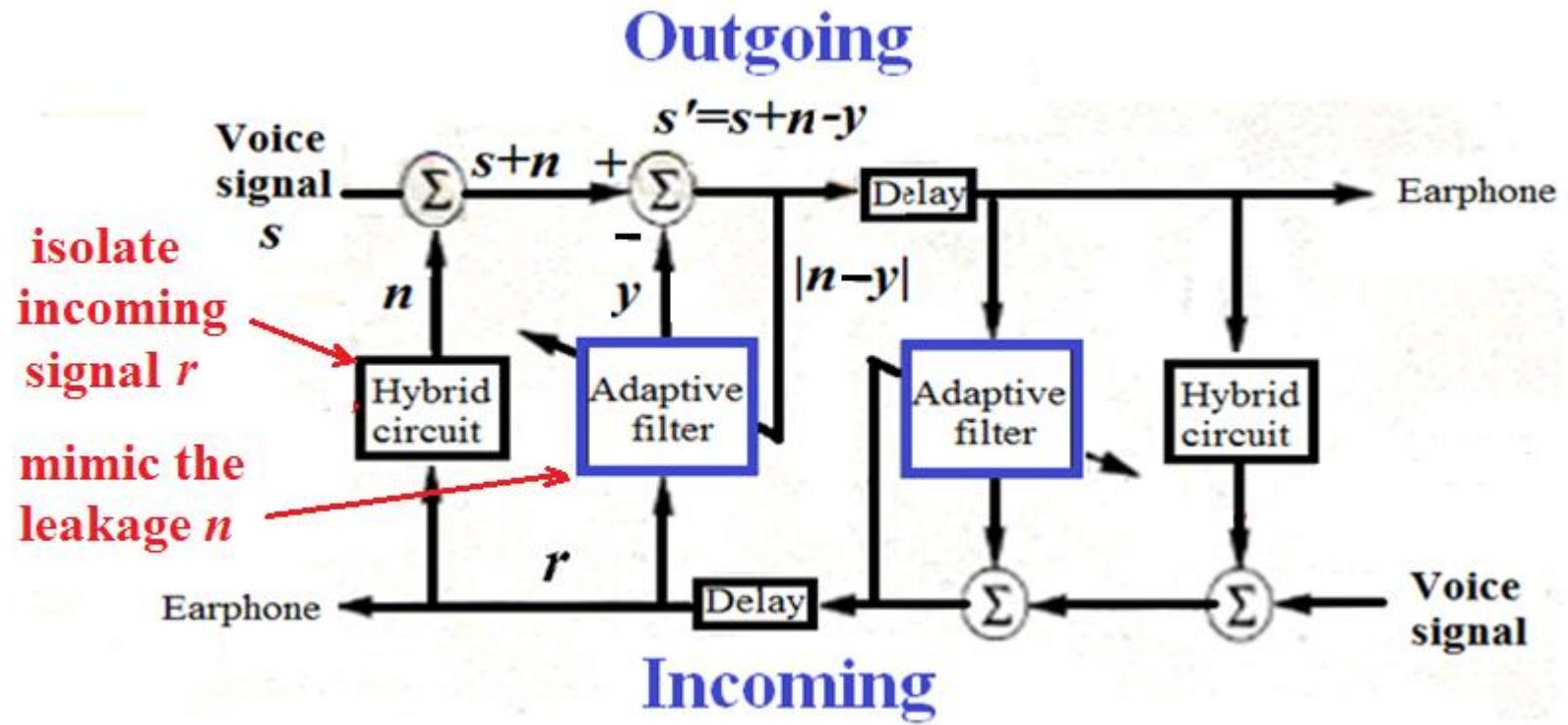
An adaptive filter is trained to predict signal. The Signal used to train the filter is a delayed actual signal. The expected output is the current signal.

Reproduce Signal

- An adaptive filter is used to model a plant.
- Inputs to the filter are the same as those to the plant. The filter adjusts its weights based on the difference between its output and the output of the plant.



Echo Cancellation in Telephone Circuits



s : outgoing voice, r : incoming voice

n : noise (leakage of r)

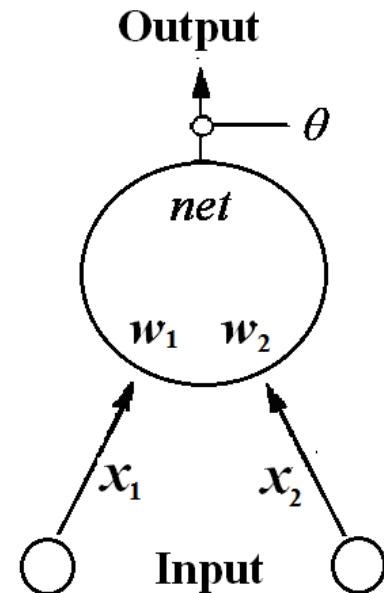
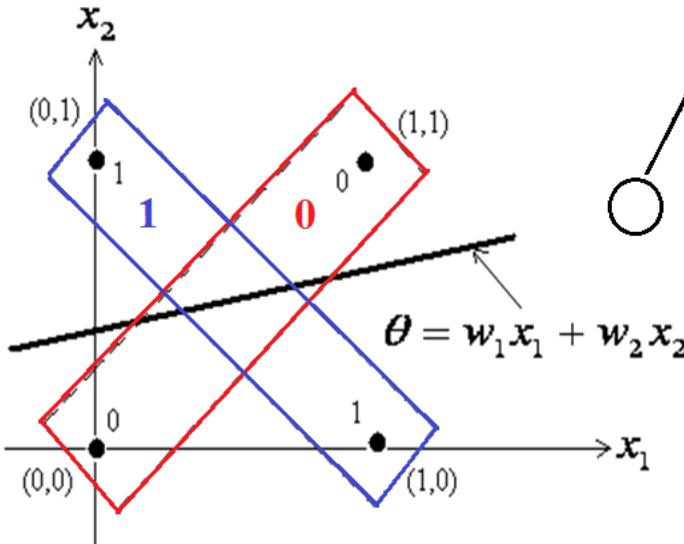
y : the output of the filter mimics n

2.4 Madaline : Many adaline

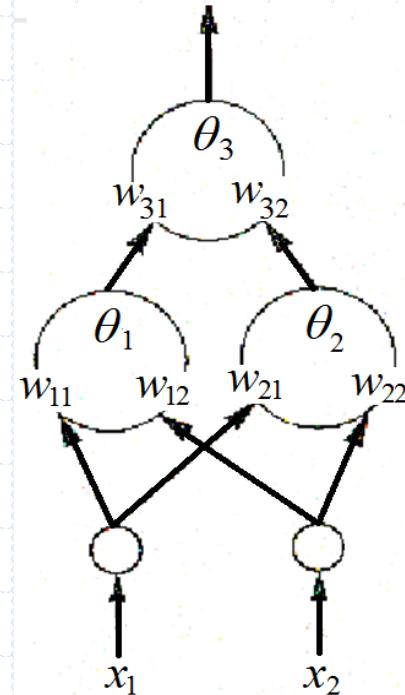
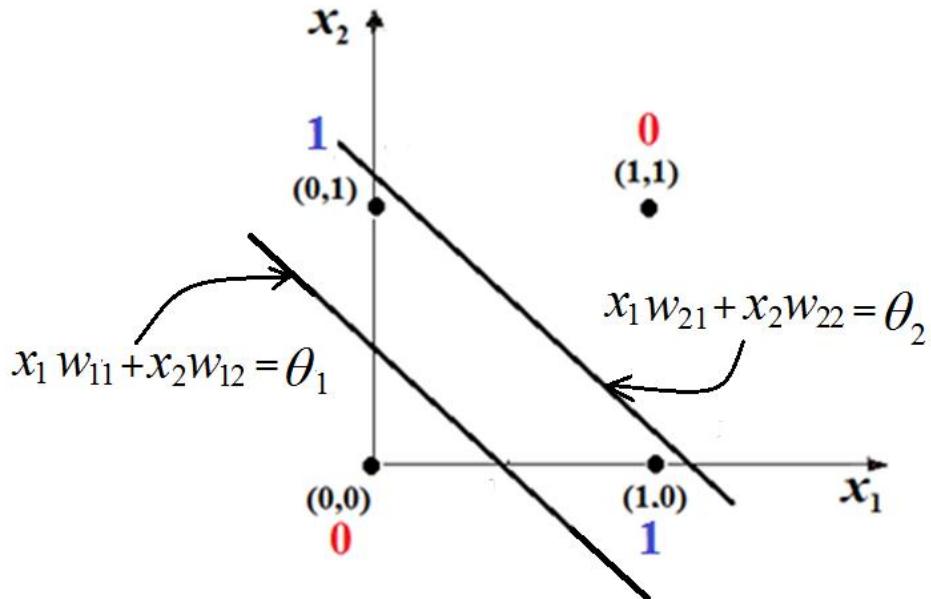
XOR function

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

This problem cannot be solved by an adaline.



Reason: $w_1x_1 + w_2x_2 = \theta$ specifies is a line in the (x_1, x_2) plane.



The two neurons in the hidden layer provides two lines that can separate the plane into three regions. The two regions containing $(0,0)$ and $(1,1)$ are associated with the network output of 0. The central region is associated with the network output of 1.

Gradient Descent

- **Gradient descent (GD)**...(not the first but used most)
- GD is aimed to find the **weight** values that minimize *Error*
- GD requires the definition of an error (or objective) function to measure the neuron's error in approximating the target

$$f_j = \psi \left(\sum_{i=1}^{n+1} w_{ji} x_i \right) \quad \text{Error} = \sum_{p=1}^P (t_p - f_p)^2$$

Where t_p and f_p are respectively the target and actual output for patterns p

The updated weights:

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

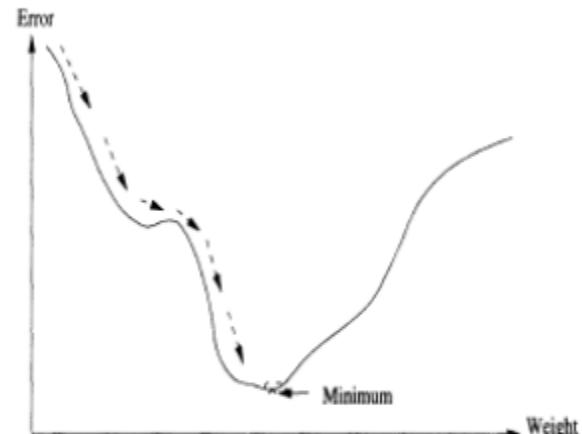
$$\text{with } \Delta w_i(t) = \eta \left(-\frac{\partial E}{\partial w_i} \right)$$

$$\text{where } \frac{\partial E}{\partial w_i} = -2(t_p - f_p) \frac{\partial f}{\partial u_p} x_{i,p}$$

where η :learning rate
 $w_i(t+1)$:new weights

3/18/2024

Analogy: Suppose we want to come down (descend) from a high hill (higher error) to a low valley (lower error). We move along the negative gradient or slopes. By doing so, we take the steepest path to the downhill valley → steepest descent algorithm

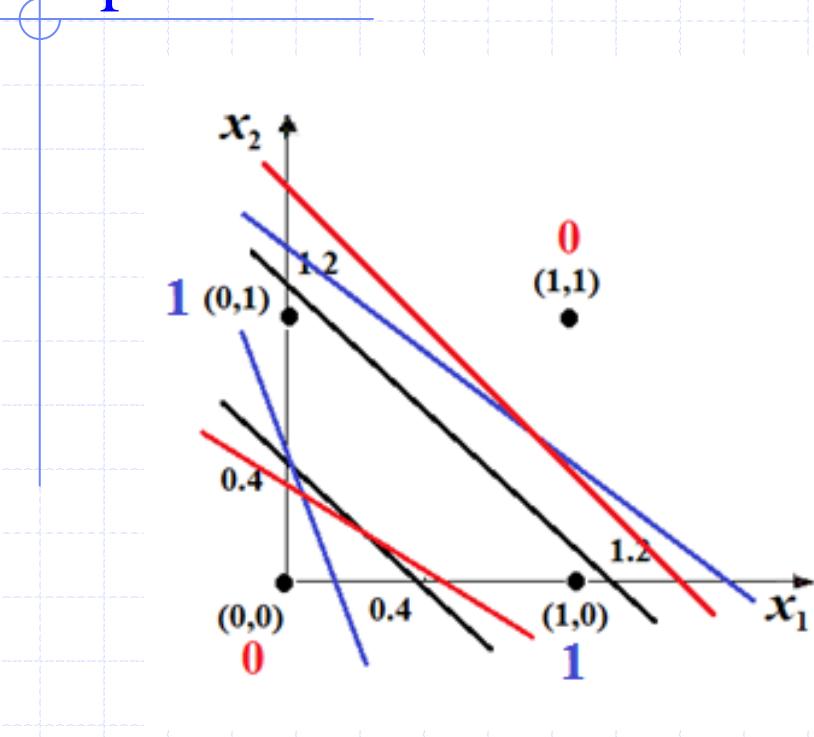


The calculation of the partial derivative of E with respect to u_p (the net input for pattern p) presents a problem for all discontinuous activation functions, such as the step and ramp functions

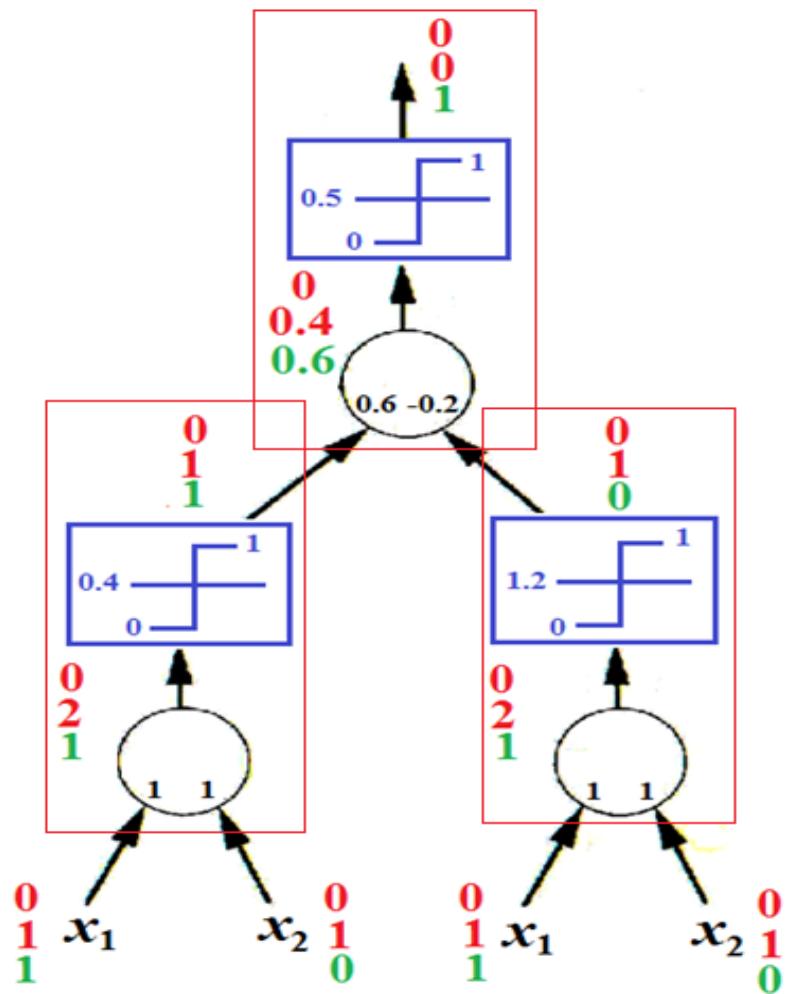


Widrow-Hoff learning rule

There are many solution pairs of lines.



One solution:



Multiple Adaptive Linear Neuron (Madaline)

Madaline which stands for Multiple Adaptive Linear Neuron, is a network which consists of many Adalines in parallel.

It will have a single output unit. Some important points about Madaline are as follows –

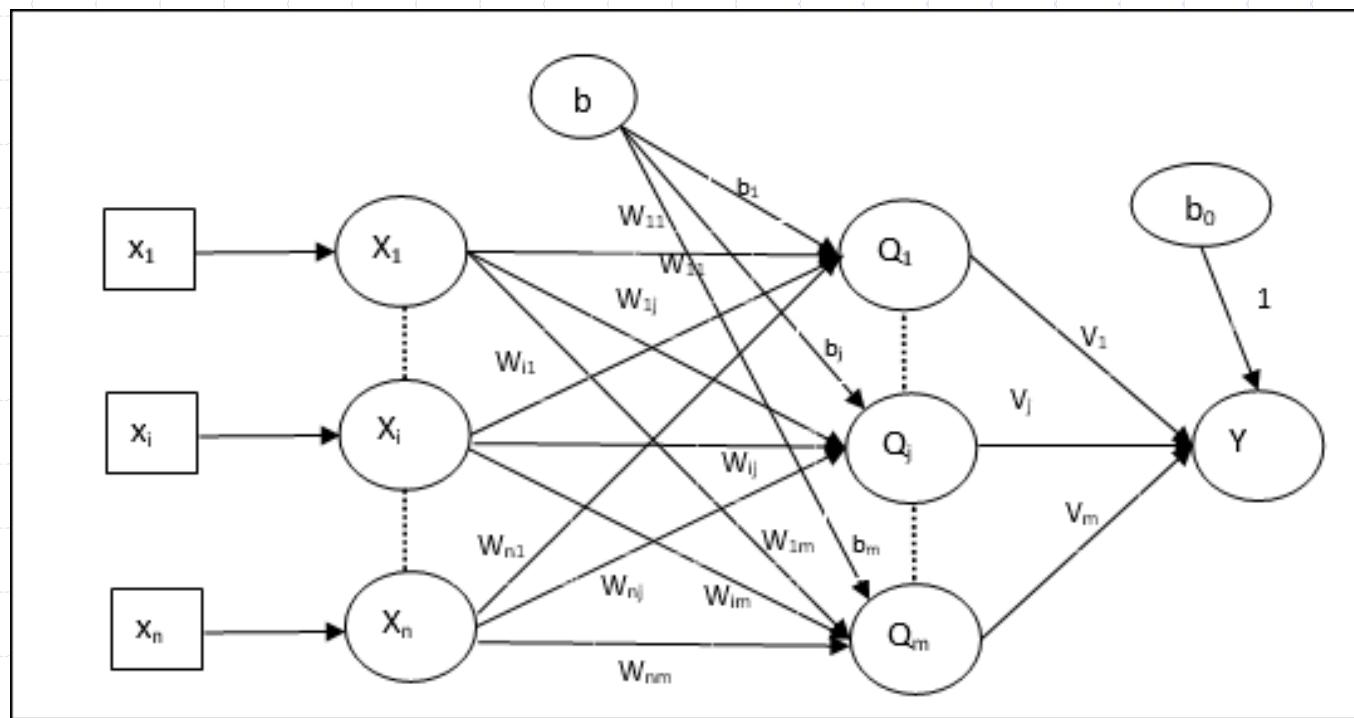
It is just like a multilayer perceptron, where Adaline will act as a hidden unit between the input and the Madaline layer.

The weights and the bias between the input and Adaline layers, as in we see in the Adaline architecture, are adjustable.

Training can be done with the help of Delta rule.

Architecture

The architecture of Madaline consists of “ n ” neurons of the input layer, “ m ” neurons of the Adaline layer, and 1 neuron of the Madaline layer. The Adaline layer can be considered as the hidden layer as it is between the input layer and the output layer, i.e. the Madaline layer.



Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function. To find a local minimum of a function using gradient descent, one takes steps proportional to the *negative* of the gradient (or of the approximate gradient) of the function at the current point. If instead one takes steps proportional to the *positive* of the gradient, one approaches a local maximum of that function; the procedure is then known as **gradient ascent**.

Gradient descent is also known as **steepest descent**.

The method of *Steepest Descent* is the simplest of the gradient methods. The choice of direction is where f decreases most quickly, which is in the direction opposite to $\nabla f(\mathbf{x}_i)$. The search starts at an arbitrary point \mathbf{x}_0 and then slide down the gradient, until we are close enough to the solution. In other words, the iterative procedure is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \lambda_k \nabla f(\mathbf{x}_k) = \mathbf{x}_k - \lambda_k \mathbf{g}(\mathbf{x}_k), \quad (4.7)$$

where $\mathbf{g}(\mathbf{x}_k)$ is the gradient at one given point.

Widrow-hoff Least-Means-Square (LMS)

Assume that $f = u_p$

The weights are updated using:

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

$$\text{with } \Delta w_i(t) = \eta \left(-\frac{\partial E}{\partial w_i} \right)$$

$$\text{where } \frac{\partial E}{\partial w_i} = -2(t_p - f_p) \frac{\partial f}{\partial u_p} x_{i,p} = -2(t_p - f_p) \cdot 1 \cdot x_{i,p} = -2(t_p - f_p) x_{i,p}$$

$$w_i(t+1) = w_i(t) + 2\eta(t_p - f_p)x_{i,p}$$

One of the first algorithms used to train multiple adaptive linear neurons
(Madaline) [Widrow 1987, Widrow and Lehr 1990]

Example

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

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

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

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

$$\lambda_1 = 1.0, \quad \lambda_2 = 0.0, \quad \lambda_3 = 2.0$$

$$\alpha < \frac{1}{\lambda_{max}} = \frac{1}{2.0} = 0.5$$

Iteration One

Banana

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

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

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

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

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

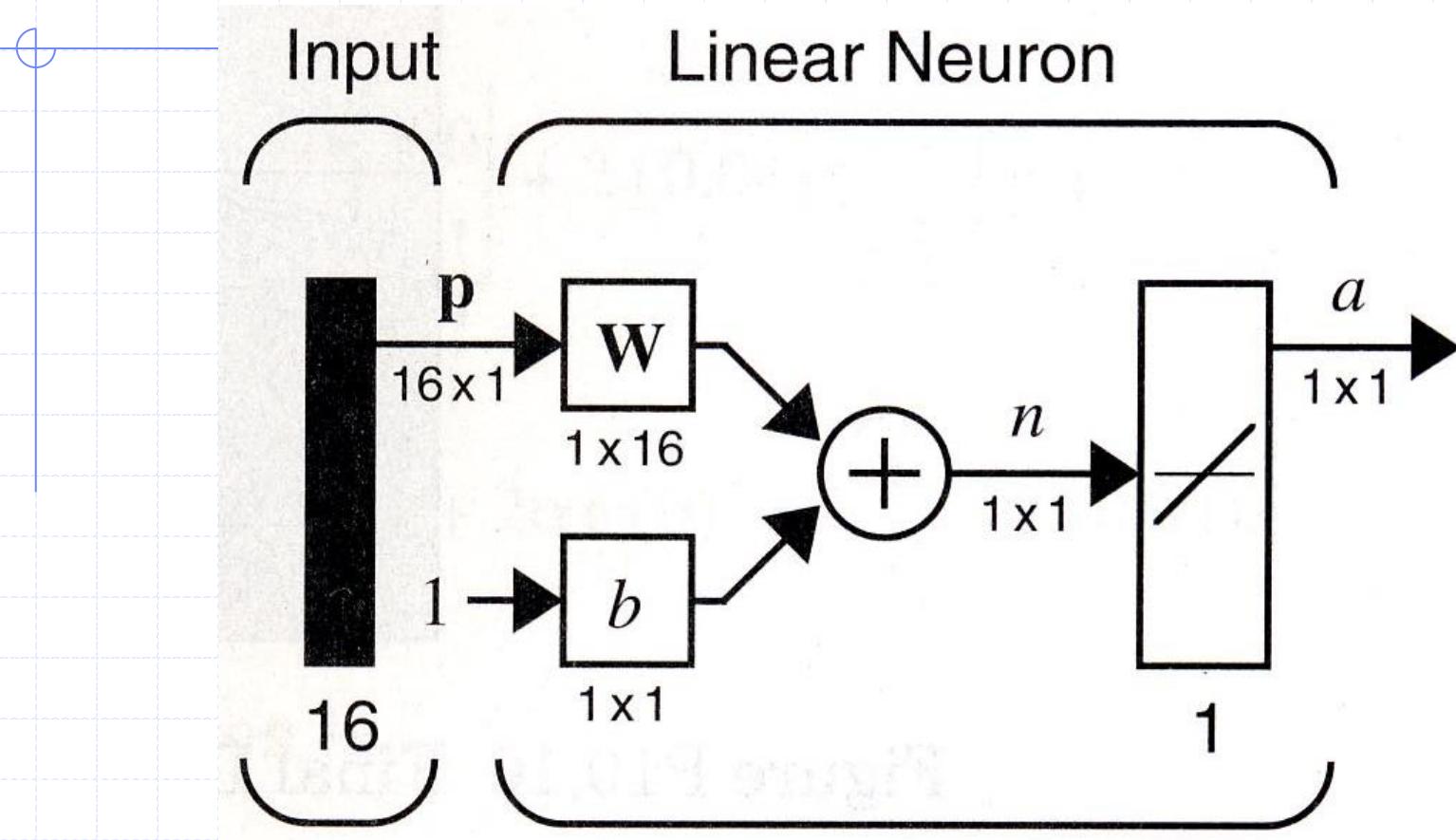
$$a(2) = \mathbf{W}(2)\mathbf{p}(2) = \mathbf{W}(2)\mathbf{p}_1 = \begin{bmatrix} 0.96 & 0.16 & -0.16 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} = -0.64$$

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

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

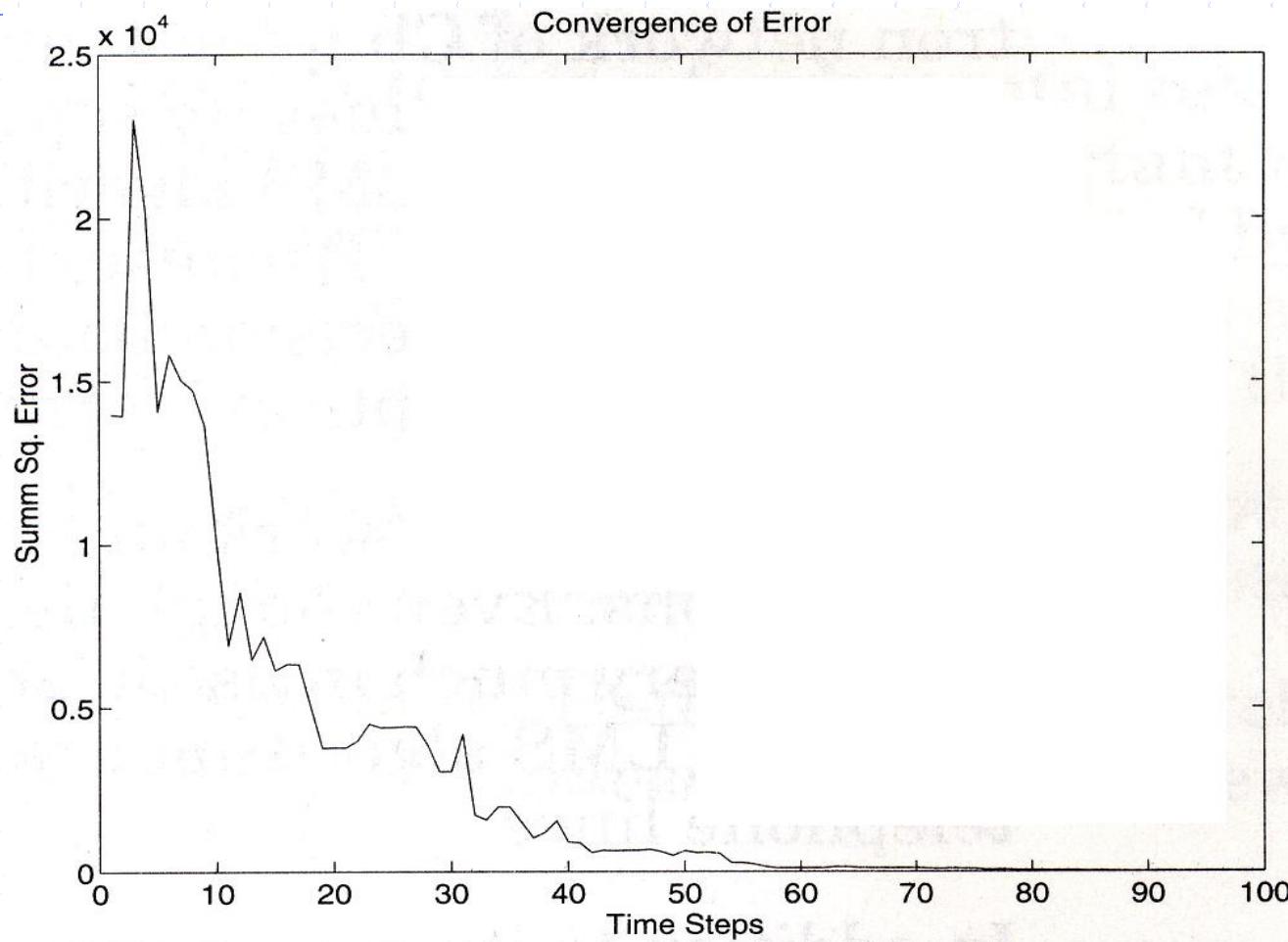
$$\mathbf{W}(\infty) = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$

Another Pattern Recognition Example – cont.



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

Another Pattern Recognition Example – cont.



Adaline Algorithm

- ◆ **Step 0:** initialize the weights to small random values and select a learning rate, (η)
- ◆ **Step 1:** set each input vector x_i , with target output, t .
- ◆ **Step 2:** compute the neuron inputs $y_{in} = b + x_i w_i$
- ◆ **Step 3:** use the delta rule to update the bias and weights
 $b \text{ (new)} = b \text{ (old)} + \text{delta}$
 $w_i \text{ (new)} = w_i \text{ (old)} + \text{delta}$
- ◆ **Step 4:** stop if the **largest weight change across all the training samples is less than a specified tolerance**, otherwise cycle through the training set again

The Learning Rate

- ◆ The performance of an ADALINE neuron depends heavily on the choice of the learning rate
 - if it is too large the system will not converge
 - if it is too small the convergence will take to long
- ◆ Typically, η is selected by trial and error
 - typical range: $0.01 < \eta < 10.0$
 - often start at 0.1
 - sometimes it is suggested that:
$$0.1 < n \eta < 1.0$$
where n is the number of inputs

Comparison of Perceptron and Adaline

	Perceptron	Adaline
Architecture	Single-layer	Single-layer
Neuron model	Non-linear	linear
Learning algorithm	Minimize number of misclassified examples	Minimize total error
Application	Linear classification	Linear classification regression

Weight Adjustments/Updates

Two types of supervised learning algorithms exist, based on when/how weights are updated:

- **Stochastic/Delta/(online) learning**, where the NN weights are adjusted after each pattern presentation. In this case the next input pattern is selected randomly from the training set, to prevent any bias that may occur due to the sequences in which patterns occur in the training set.
- **Batch/(offline) learning**, where the NN weight changes are accumulated and used to adjust weights only after all training patterns have been presented

Summary of Adaline LMS learning

Step1: Initialize weights and thresholds: Set the w_{ij} and the b_{ij} to small random values in the range [-1,+1].

Step2: For each pattern pair (p_k, t_k) do:

(i) Input p_k and calculate the actual output:

$$a_j = \sum w_{ij} p_i + b_j$$

(ii) Compute the error:

$$e^2(k) = (t_k - a_k)^2$$

(iii) Adjust the weights:

$$\Delta w_{ij} = \pm 2\alpha e_i p_i \text{ where } (e_i = t_i - a_i)$$

Gradient descent!

Step 3: Repeat step 2 until the error correction is sufficiently low or zero.

WBS WS06-07 14

Training Algorithm

Step 1 – Initialize the following to start the training –

- Weights
- Bias
- Learning rate α

For easy calculation and simplicity, weights and bias must be set equal to 0 and the learning rate must be set equal to 1.

Step 2 – Continue step 3-8 when the stopping condition is not true.

Step 3 – Continue step 4-6 for every bipolar training pair **s:t**.

Step 4 – Activate each input unit as follows –

$$x_i = s_i \quad (i = 1 \text{ to } n)$$

Step 5 – Obtain the net input with the following relation –

$$y_{in} = b + \sum_i^n x_i w_i$$

Here ‘**b**’ is bias and ‘**n**’ is the total number of input neurons.

Step 6 – Apply the following activation function to obtain the final output –

$$f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} \geq 0 \\ -1 & \text{if } y_{in} < 0 \end{cases}$$

Step 7 – Adjust the weight and bias as follows –

Case 1 – if $y \neq t$ then,

$$w_i(\text{new}) = w_i(\text{old}) + \alpha(t - y_{in})x_i$$

$$b(\text{new}) = b(\text{old}) + \alpha(t - y_{in})$$

Case 2 – if $y = t$ then,

$$w_i(\text{new}) = w_i(\text{old})$$

$$b(\text{new}) = b(\text{old})$$

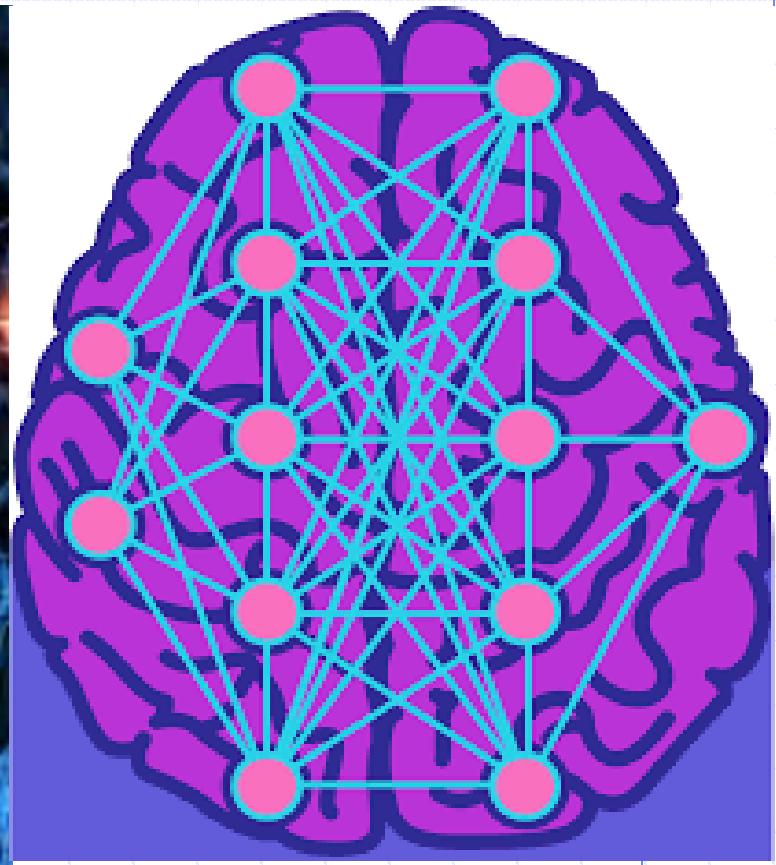
Here ‘y’ is the actual output and ‘t’ is the desired/target output.

$(t - y_{in})$ is the computed error.

Step 8 – Test for the stopping condition, which will happen when there is no change in weight or the highest weight change occurred during training is smaller than the specified tolerance.

ITF309

Artificial Neural Networks



CHAPTER 11

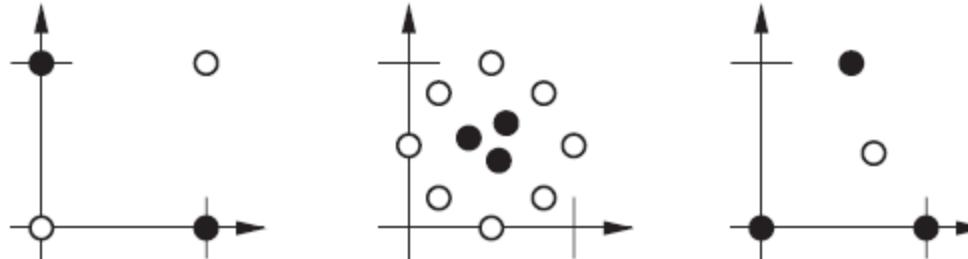
Back-Propagation

Limitations of perceptron

- ◆ The perceptron learning rule is guaranteed to converge to a solution in a finite number of steps, so long as a solution exists.
- ◆ The two classes must be linearly separable.

Limitations perceptron

- ◆ Linearly non-separable problems



Try drawing a straight line between two classes vectors

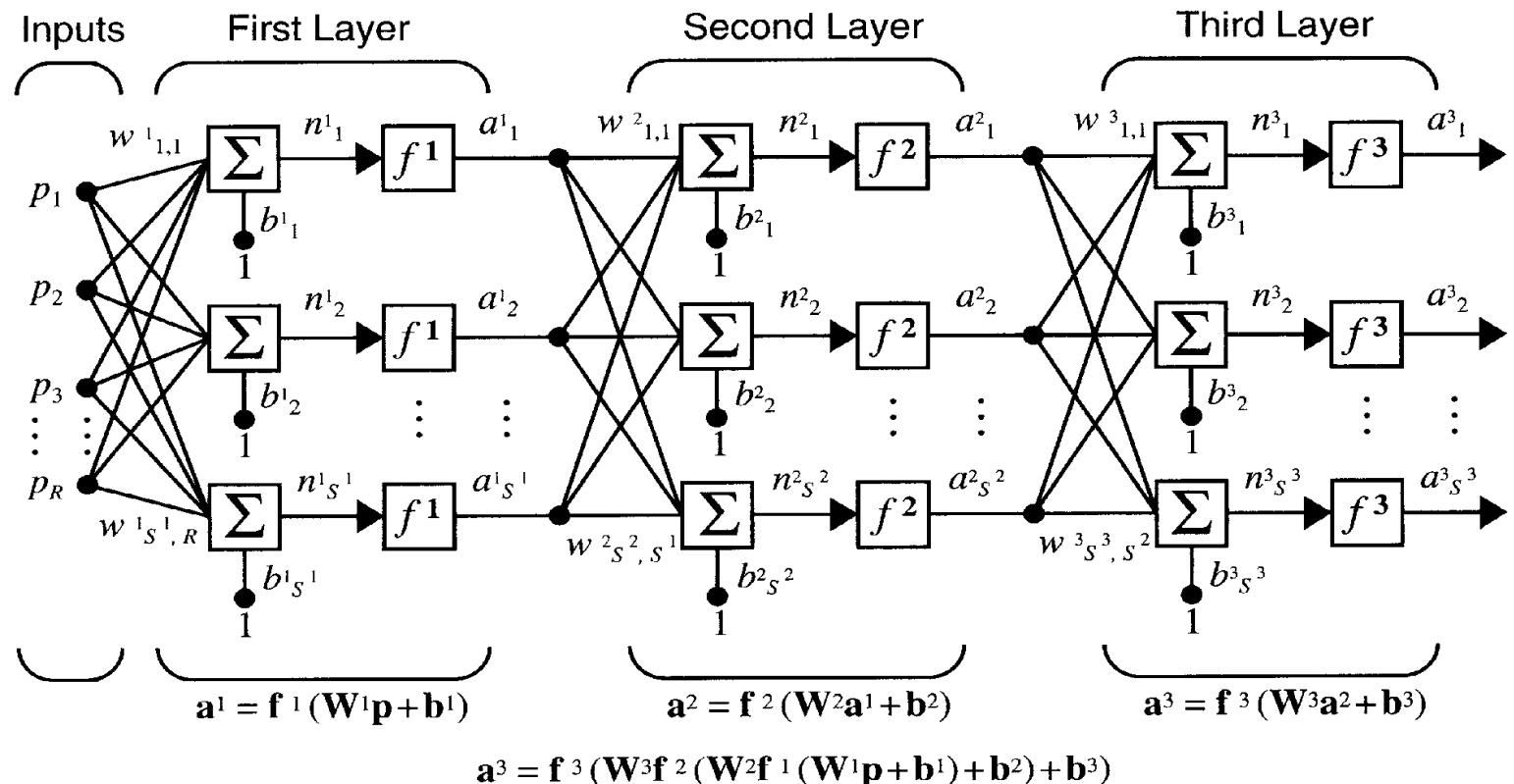
Objectives

- ◆ A generalization of the LMS algorithm, called **backpropagation**, can be used to train ***multilayer networks***.
- ◆ Backpropagation is an ***approximate steepest descent algorithm***, in which the performance index is **mean square error**.
- ◆ In order to calculate the **derivatives**, we need to use ***the chain rule of calculus***.

Motivation

- ◆ The **perceptron learning** and the **LMS algorithm** were designed to train **single-layer** perceptron-like networks.
- ◆ They are only able to solve **linearly separable classification** problems.
- ◆ The **multilayer perceptron**, trained by the **backpropagation algorithm**, is currently the most widely used neural network.

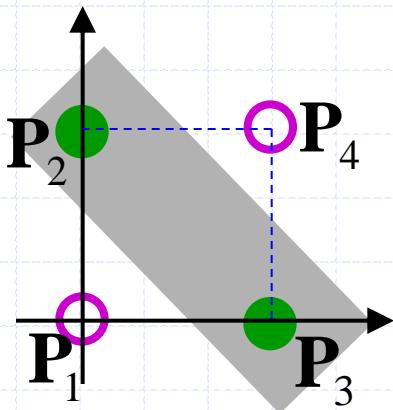
Three-Layer Network



Number of neurons in each layer: $R - S^1 - S^2 - S^3$

Pattern Classification: XOR gate

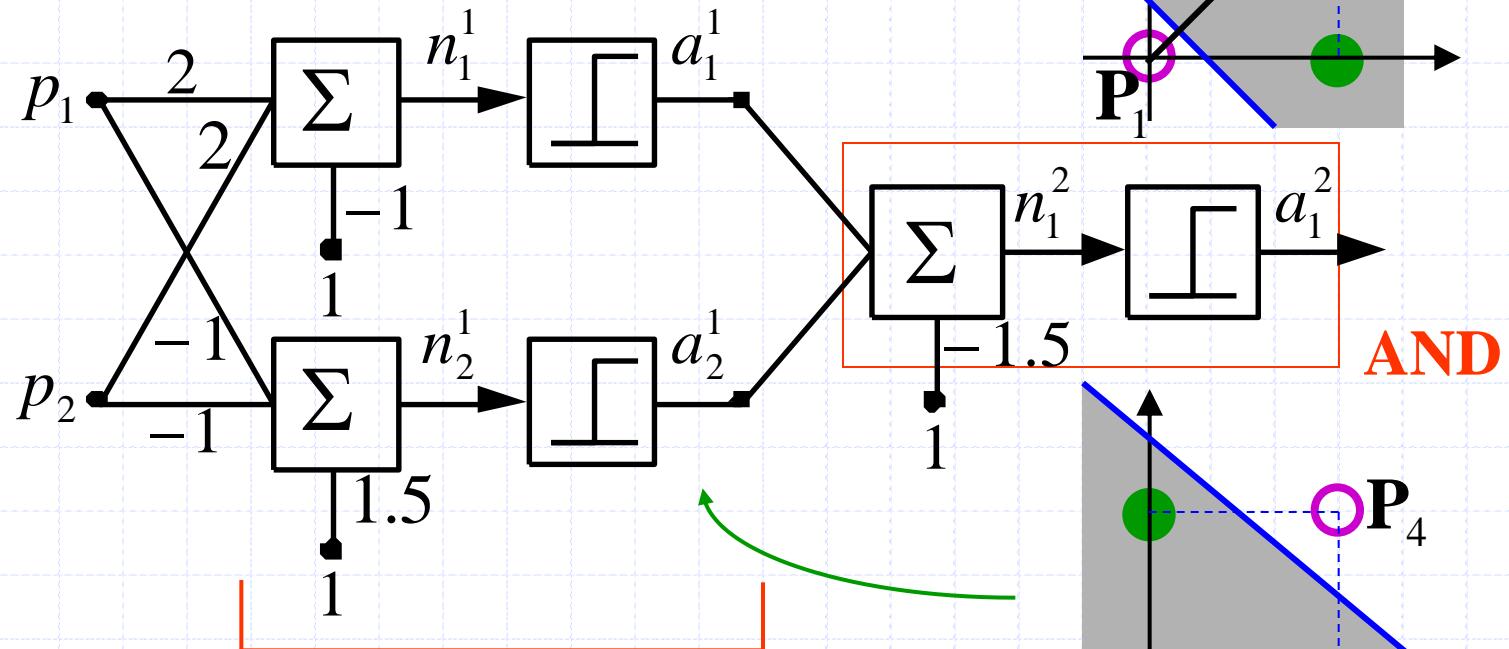
- ◆ The limitations of the single-layer perceptron (Minsky & Papert, 1969)



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

Two-Layer XOR Network

◆ Two-layer, 2-2-1 network



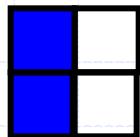
Individual Decisions

EELU ITF309 Neural Network

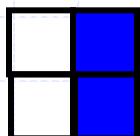
Lecture 7

Solved Problem P11.1

- ◆ Design a multilayer network to distinguish these categories.



$$\mathbf{p}_1 = [1 \quad 1 \quad -1 \quad -1]^T$$

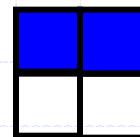


$$\mathbf{p}_2 = [-1 \quad -1 \quad 1 \quad 1]^T$$

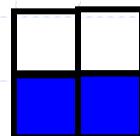
Class I

$$\mathbf{W}\mathbf{p}_1 + b > 0$$

$$\mathbf{W}\mathbf{p}_2 + b > 0$$



$$\mathbf{p}_3 = [1 \quad -1 \quad 1 \quad -1]^T$$



$$\mathbf{p}_4 = [-1 \quad 1 \quad -1 \quad 1]^T$$

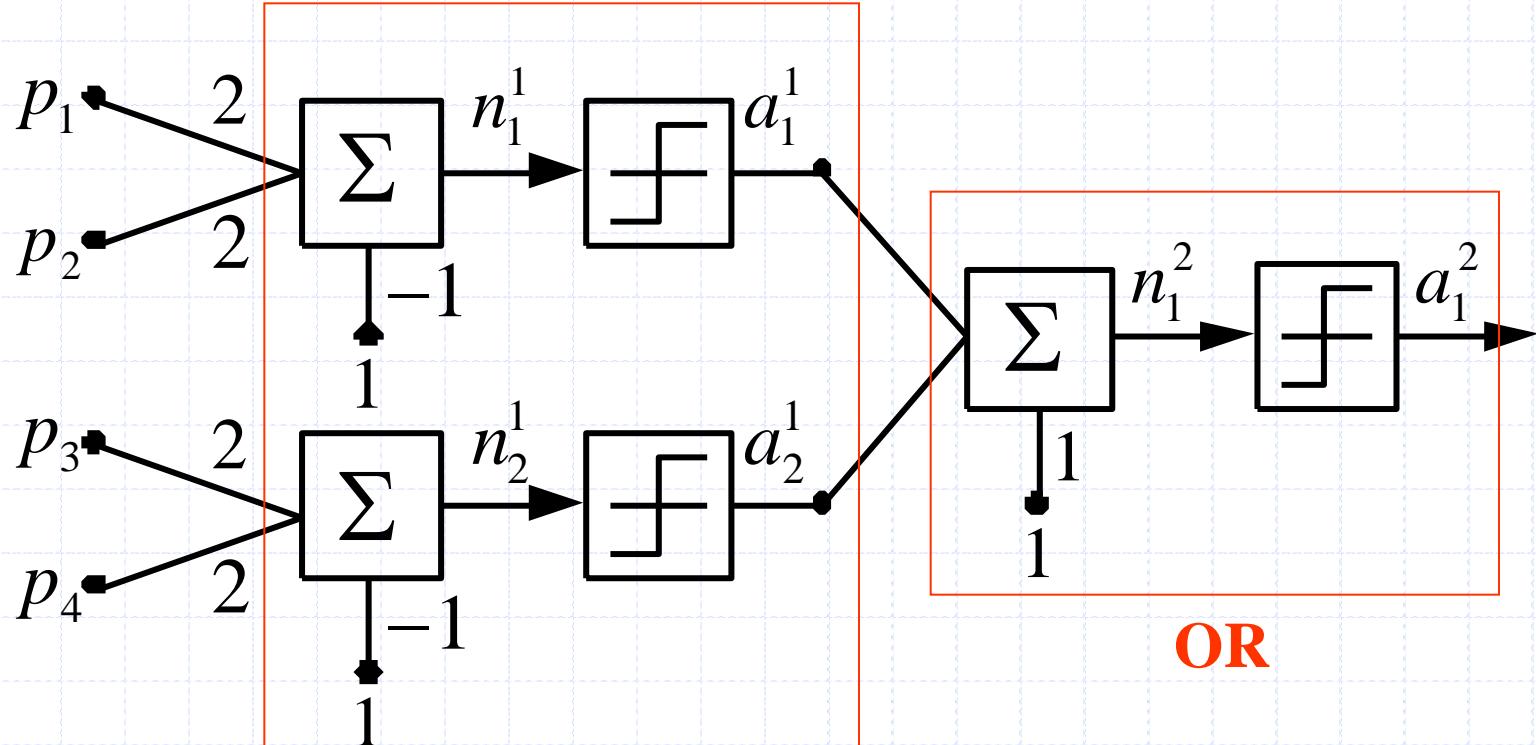
Class II

$$\mathbf{W}\mathbf{p}_3 + b < 0$$

$$\mathbf{W}\mathbf{p}_4 + b < 0$$

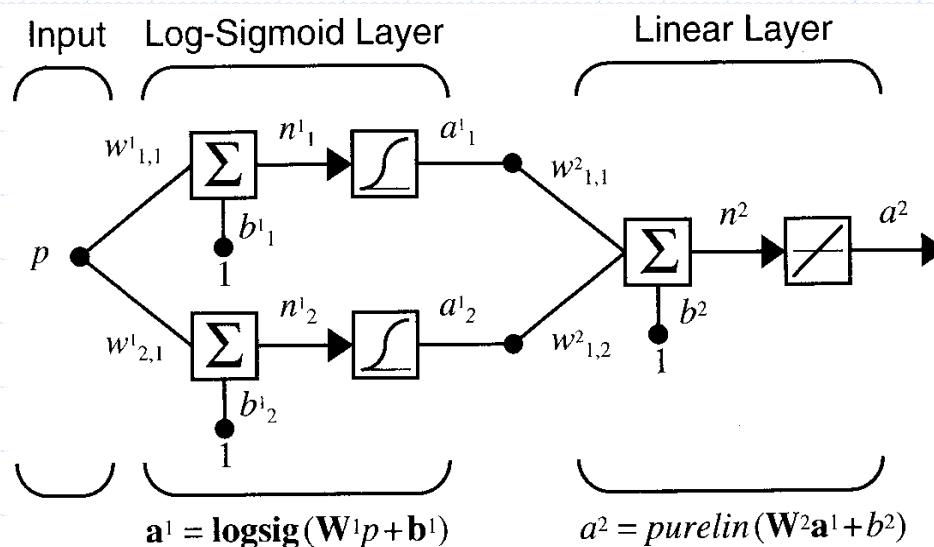
There is no hyperplane that can separate these two categories.

Solution of Problem P11.1



Function Approximation

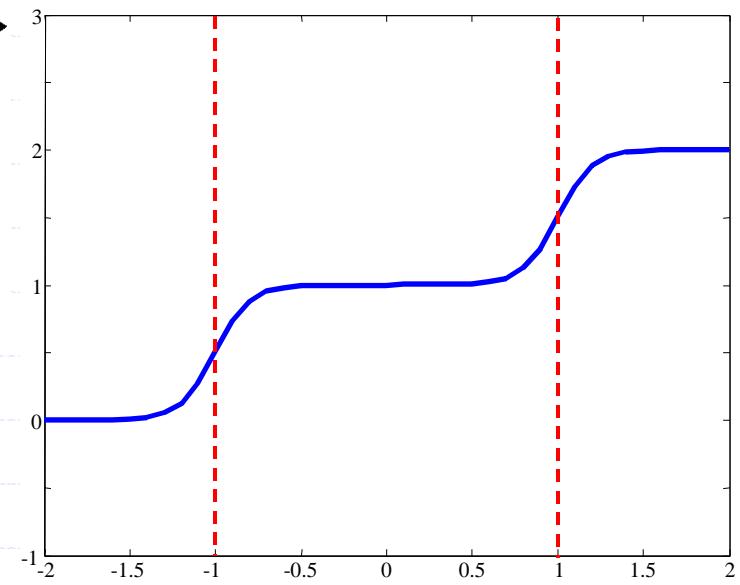
◆ Two-layer, 1-2-1 network



$$w_1^1 = 10, w_2^1 = 10, b_1^1 = -10, b_2^1 = 10.$$

$$w_1^2 = 1, w_2^2 = 1, b^2 = 0.$$

$$f^1(n) = \frac{1}{1+e^{-n}}, f^2(n) = n$$



Function Approximation

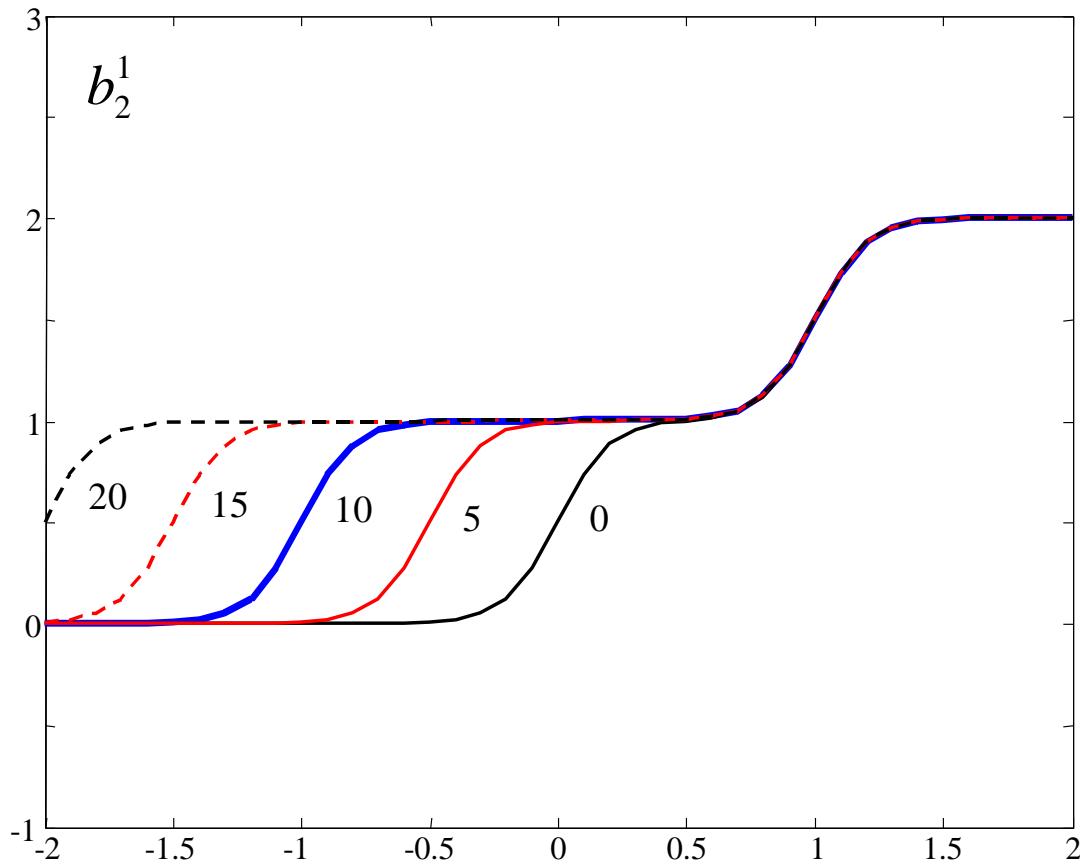
- ◆ The **centers** of the steps occur where the ***net input*** to a neuron in ***the first layer*** is **zero**.

$$n_1^1 = w_1^1 p + b_1^1 = 0 \Rightarrow p = -b_1^1 / w_1^1 = -(-10)/10 = 1$$

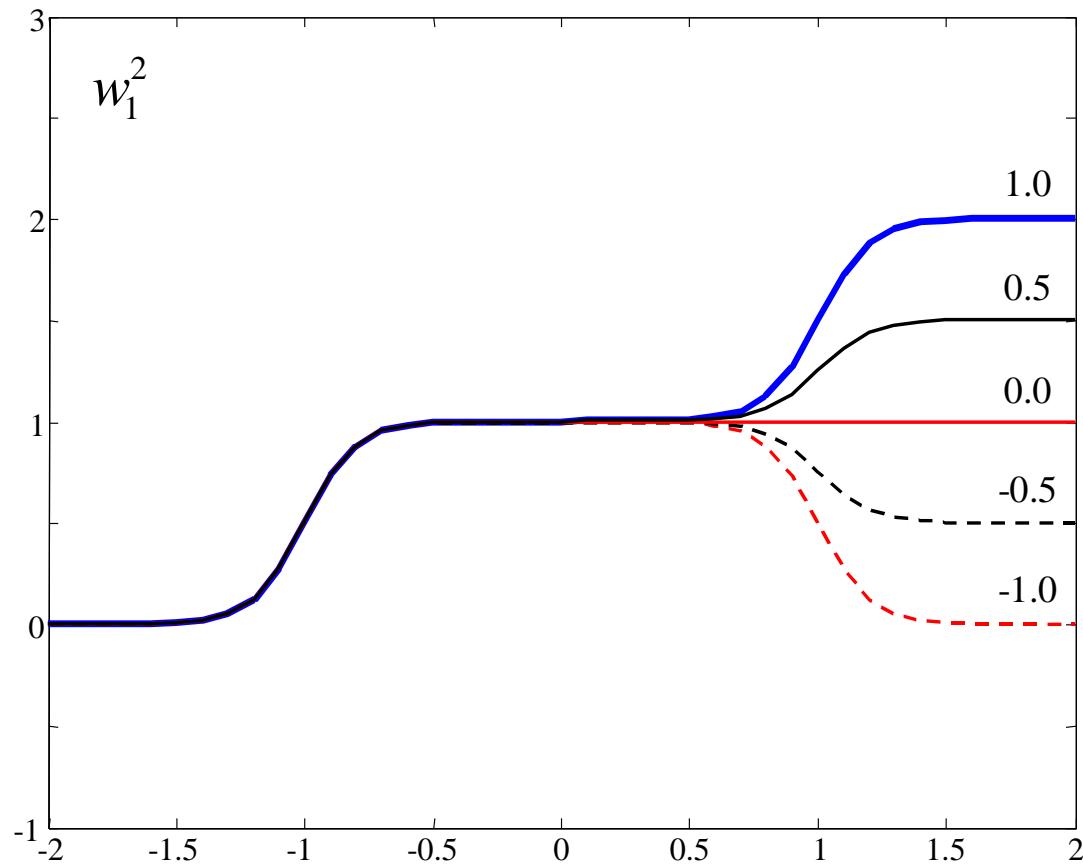
$$n_2^1 = w_2^1 p + b_2^1 = 0 \Rightarrow p = -b_2^1 / w_2^1 = -10/10 = -1$$

- ◆ The **steepness** of each step can be adjusted by ***changing*** the ***network weights***.

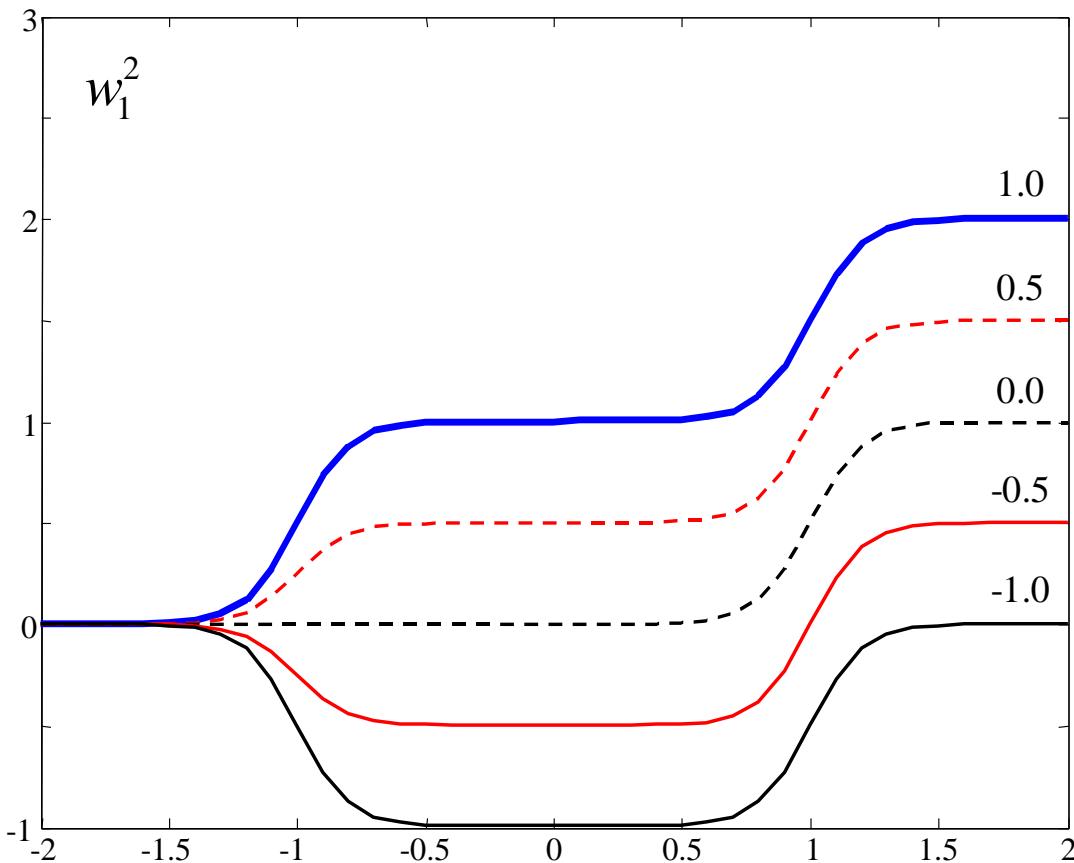
Effect of Parameter Changes



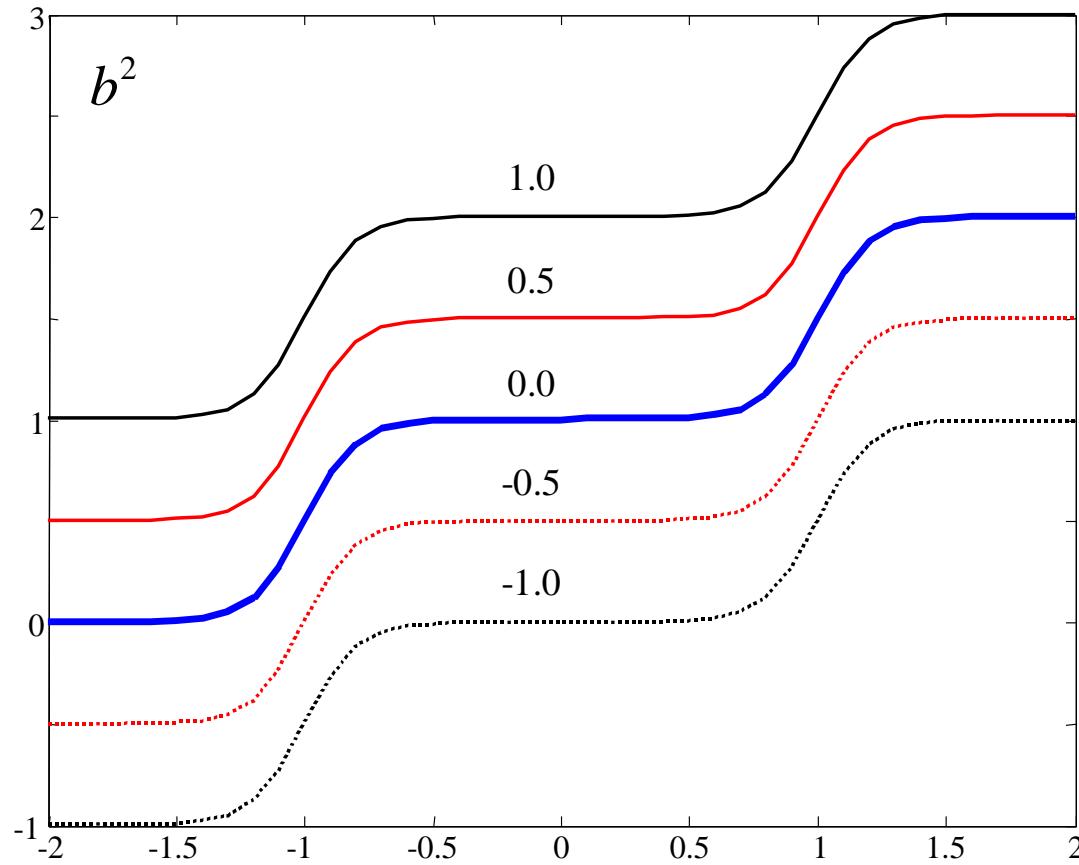
Effect of Parameter Changes



Effect of Parameter Changes



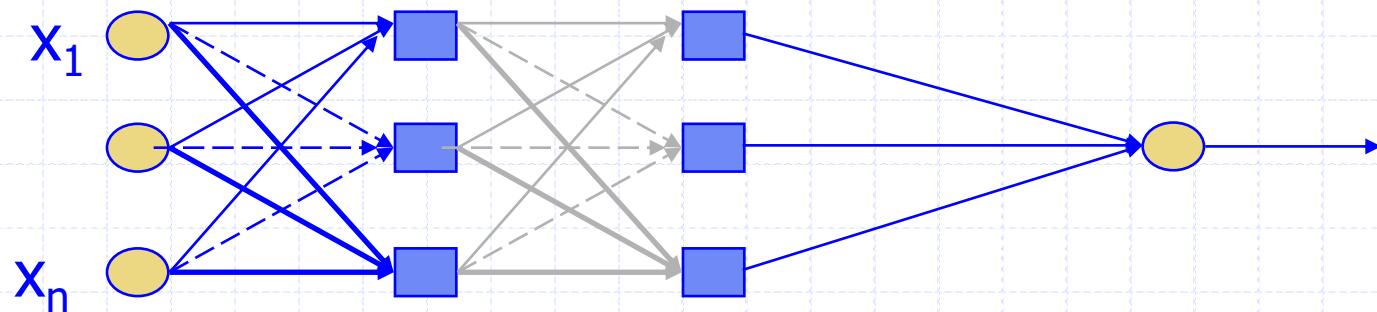
Effect of Parameter Changes



Function Approximation

- ◆ 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 accuracy, provided sufficiently many hidden units are available.

Backpropagation Algorithm



The *backpropagation* algorithm was used to train the *multi-layer perception* MLP

MLP used to describe any general Feedforward (no recurrent connections) Neural Network FNN

However, we will concentrate on nets with units arranged in layers

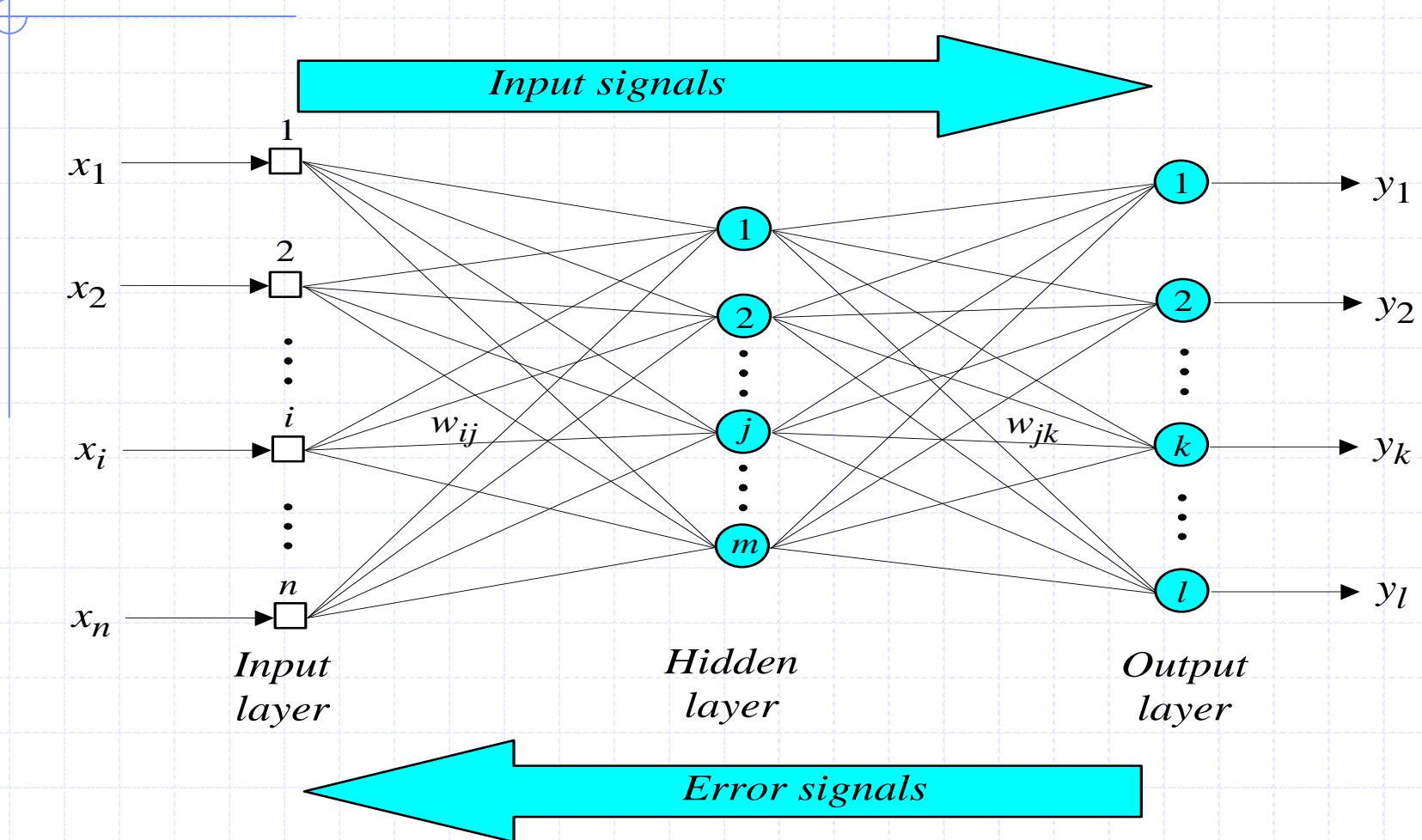
Architecture of BP Nets

- Multi-layer, feed-forward networks have the following characteristics:
 - They must have at least one hidden layer
 - Hidden units must be non-linear units (usually with sigmoid activation functions).
 - Fully connected between units in two consecutive layers, but no connection between units within one layer.
 - For a net with only one hidden layer, each hidden unit receives input from all input units and sends output to all output units
 - Number of output units need not equal number of input units
 - Number of hidden units per layer can be more or less than input or output units.

Training a BackPropagation Net

- **Feed-forward training of input patterns**
 - each input node receives a signal, which is broadcast to all of the hidden units
 - each hidden unit computes its activation which is broadcast to all output nodes
- **Back propagation of errors**
 - each output node compares its activation with the desired output
 - based on these differences, the error is propagated back to all previous nodes *Delta Rule*
- **Adjustment of weights**
 - weights of all links computed simultaneously based on the errors that were propagated back

Three-layer back-propagation neural network



Generalized delta rule

- Delta rule only works for the output layer.
- Backpropagation, or the generalized delta rule, is a way of creating desired values for hidden layers

Description of Training BP Net: Feedforward Stage

1. Initialize weights with small, random values
2. While stopping condition is not true
 - for each training pair (input/output):
 - ◆ each input unit broadcasts its value to all hidden units
 - ◆ each hidden unit sums its input signals & applies activation function to compute its output signal
 - ◆ each hidden unit sends its signal to the output units
 - ◆ each output unit sums its input signals & applies its activation function to compute its output signal

Training BP Net: Backpropagation stage

3. Each output computes its error term, its own weight correction term and its bias(threshold) correction term & sends it to layer below
4. Each hidden unit sums its delta inputs from above & multiplies by the derivative of its activation function; it also computes its own weight correction term and its bias correction term

Training a Back Prop Net: Adjusting the Weights

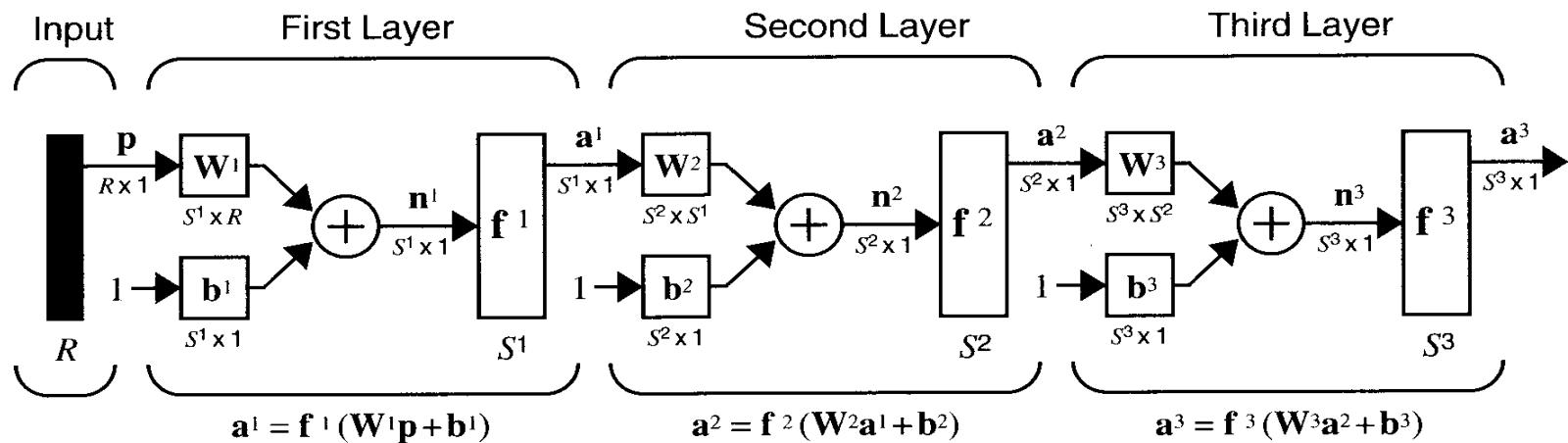
5. Each output unit updates its weights and bias
6. Each hidden unit updates its weights and bias
 - Each training cycle is called an epoch. The weights are updated in each cycle
 - It is not analytically possible to determine where the global minimum is. Eventually the algorithm stops in a low point, which may just be a local minimum.

Backpropagation Algorithm

- ◆ For multilayer networks the outputs of one layer becomes the input to the following layer.

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}), m=0,1,2,\dots, M-1$$

$$\mathbf{a}_0 = \mathbf{p}, \quad \mathbf{a} = \mathbf{a}^M$$



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

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

- ◆ Approximate Steepest Descent Algorithm

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

Chain Rule

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

◆ If $f(n) = e^n$ and $n = 2w$, so that $f(n(w)) = e^{2w}$.

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

◆ Approximate mean square error:

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

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

$$b_i^m(k+1) = b_i^m(k) - \alpha \frac{\partial \hat{F}}{\partial b_i^m} = b_i^m(k) - \alpha \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m}$$

Sensitivity & Gradient

- ◆ The net input to the i th neurons of layer m :

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

- ◆ The **sensitivity** of \hat{F} to changes in the i th element of the net input at layer m : $s_i^m \equiv \partial \hat{F} / \partial n_i^m$

- ◆ **Gradient:** $\frac{\partial \hat{F}}{\partial w_{i,j}^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^m} = s_i^m \cdot a_j^{m-1}$

$$\frac{\partial \hat{F}}{\partial b_i^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m} = s_i^m \cdot 1 = s_i^m$$

Steepest Descent Algorithm

- ◆ The **steepest descent algorithm** for the approximate mean square error:

$$w_{i,j}^m(k+1) = w_{i,j}^m(k) - \alpha \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^m} = w_{i,j}^m(k) - \alpha s_i^m a_j^{m-1}$$

$$b_i^m(k+1) = b_i^m(k) - \alpha \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m} = b_i^m(k) - \alpha s_i^m$$

- ◆ Matrix form:

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

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

$$\mathbf{s}^m \equiv \frac{\partial F}{\partial \mathbf{n}^m} =$$

$$\begin{bmatrix} \frac{\partial F}{\partial n_1^m} \\ \frac{\partial F}{\partial n_2^m} \\ \vdots \\ \frac{\partial F}{\partial n_{S^m}^m} \end{bmatrix}$$

BP the Sensitivity

- ◆ Backpropagation: a **recurrence relationship** in which the sensitivity at layer m is computed from the sensitivity at layer $m+1$.

- ◆ Jacobian matrix:

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} = \begin{bmatrix} \frac{\partial n_1^{m+1}}{\partial n_1^m} & \frac{\partial n_1^{m+1}}{\partial n_2^m} & \dots & \frac{\partial n_1^{m+1}}{\partial n_{s^m}^m} \\ \frac{\partial n_2^{m+1}}{\partial n_1^m} & \frac{\partial n_2^{m+1}}{\partial n_2^m} & \dots & \frac{\partial n_2^{m+1}}{\partial n_{s^m}^m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial n_{s^{m+1}}^{m+1}}{\partial n_1^m} & \frac{\partial n_{s^{m+1}}^{m+1}}{\partial n_2^m} & \dots & \frac{\partial n_{s^{m+1}}^{m+1}}{\partial n_{s^m}^m} \end{bmatrix}.$$

Matrix Repression

- ◆ The i,j element of Jacobian matrix

$$\begin{aligned}\frac{\partial n_i^{m+1}}{\partial n_j^m} &= \frac{\partial \left(\sum_{l=1}^{s^m} w_{i,l}^{m+1} a_l^m + b_i^{m+1} \right)}{\partial n_j^m} = w_{i,j}^{m+1} \frac{\partial a_j^m}{\partial n_j^m} = w_{i,j}^{m+1} \frac{\partial f^m(n_j^m)}{\partial n_j^m} \\ &= w_{i,j}^{m+1} \dot{f}^m(n_j^m).\end{aligned}$$

- ◆ $\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} = \mathbf{W}^{m+1} \dot{\mathbf{F}}^m(\mathbf{n}^m),$

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

Recurrence Relation

- ◆ The recurrence relation for the sensitivity

$$\begin{aligned}\mathbf{s}^m &= \frac{\partial \hat{\mathbf{F}}}{\partial \mathbf{n}^m} = \left(\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} \right)^T \frac{\partial \hat{\mathbf{F}}}{\mathbf{n}^{m+1}} = \dot{\mathbf{F}}(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \frac{\partial \hat{\mathbf{F}}}{\partial \mathbf{n}^{m+1}} \\ &= \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \mathbf{s}^{m+1}.\end{aligned}$$

- ◆ The **sensitivities** are *propagated backward* through the network from the **last** layer to the **first** layer.

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

Backpropagation Algorithm

◆ At the **final** layer:

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

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

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

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

Summary

- ◆ The **first step** is to propagate the **input forward** through the network:

$$\mathbf{a}_0 = \mathbf{p}$$

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}), m = 0, 1, 2, \dots, M-1$$

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

- ◆ The **second step** is to propagate the **sensitivities backward** through the network:

- Output layer:

- Hidden layer: $\mathbf{s}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t} - \mathbf{a})$

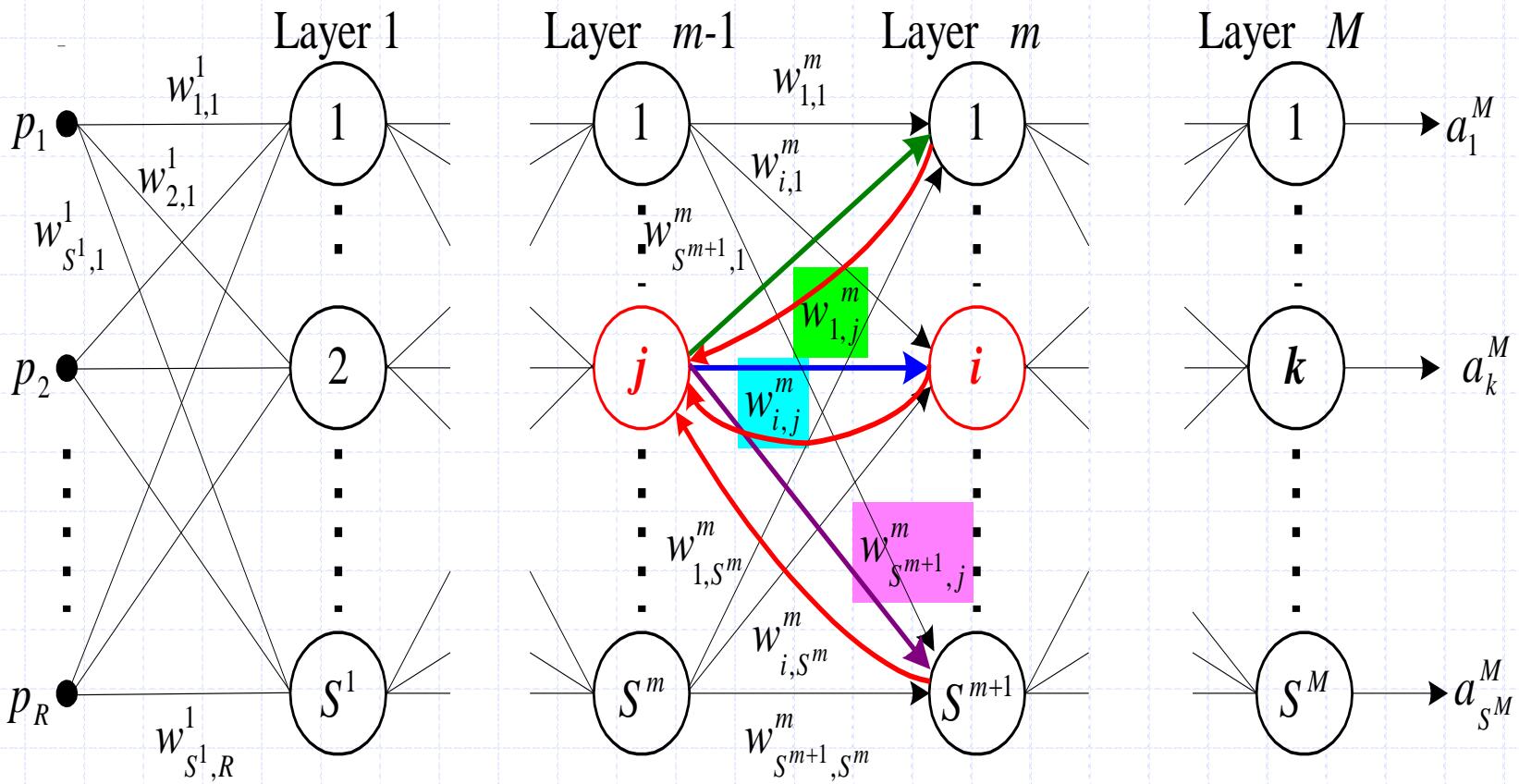
$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \mathbf{s}^{m+1}, m = M-1, \dots, 2, 1$$

- ◆ The **final step** is to **update the weights and biases**:

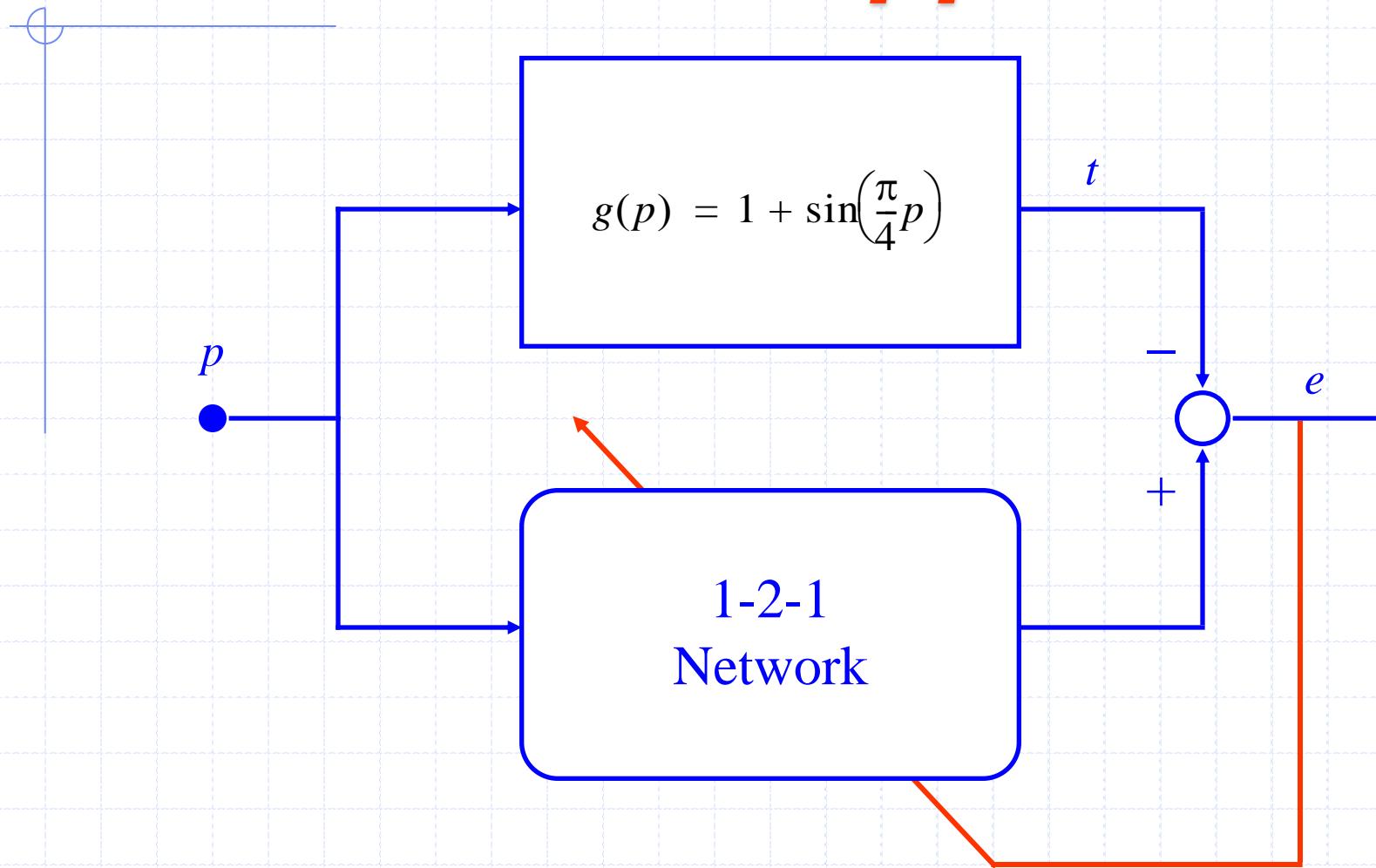
$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T$$

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

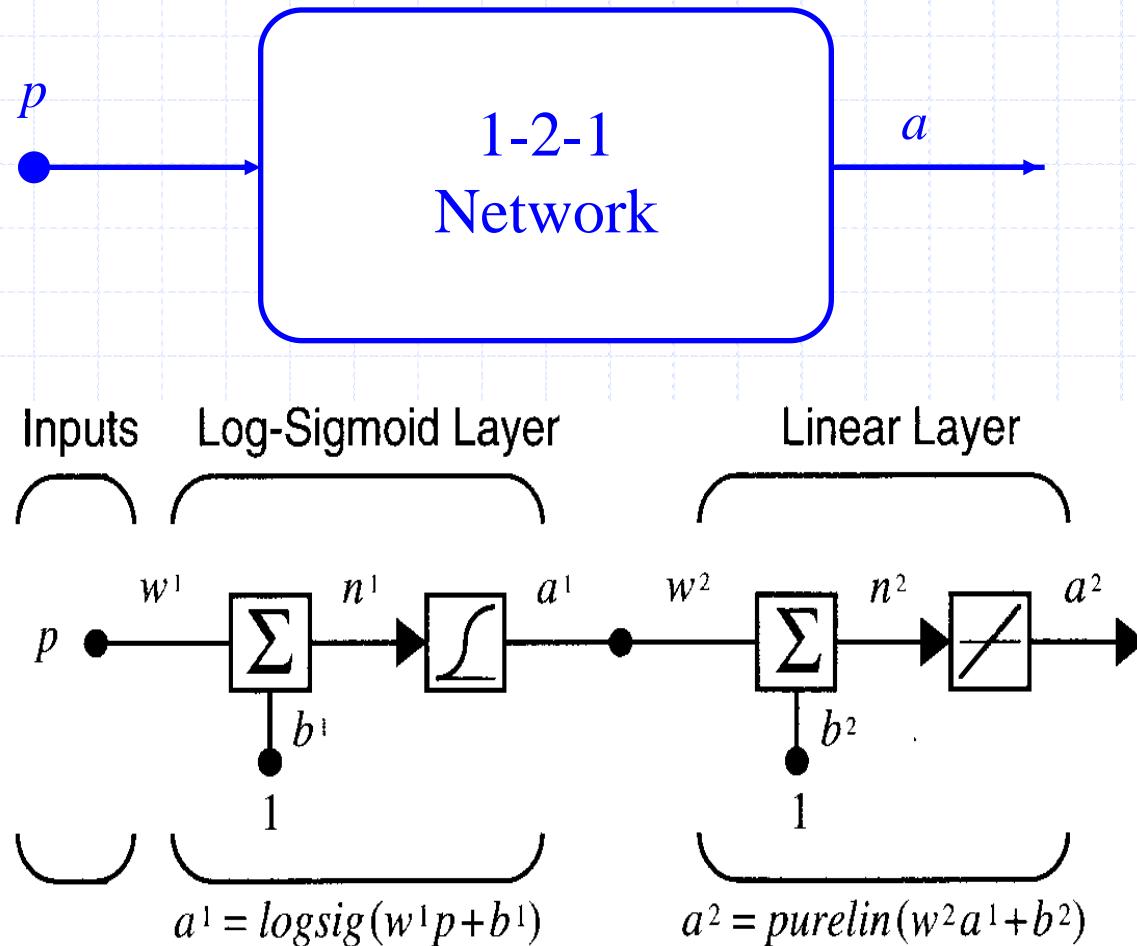
BP Neural Network



Ex: Function Approximation



Network Architecture



Initial Values

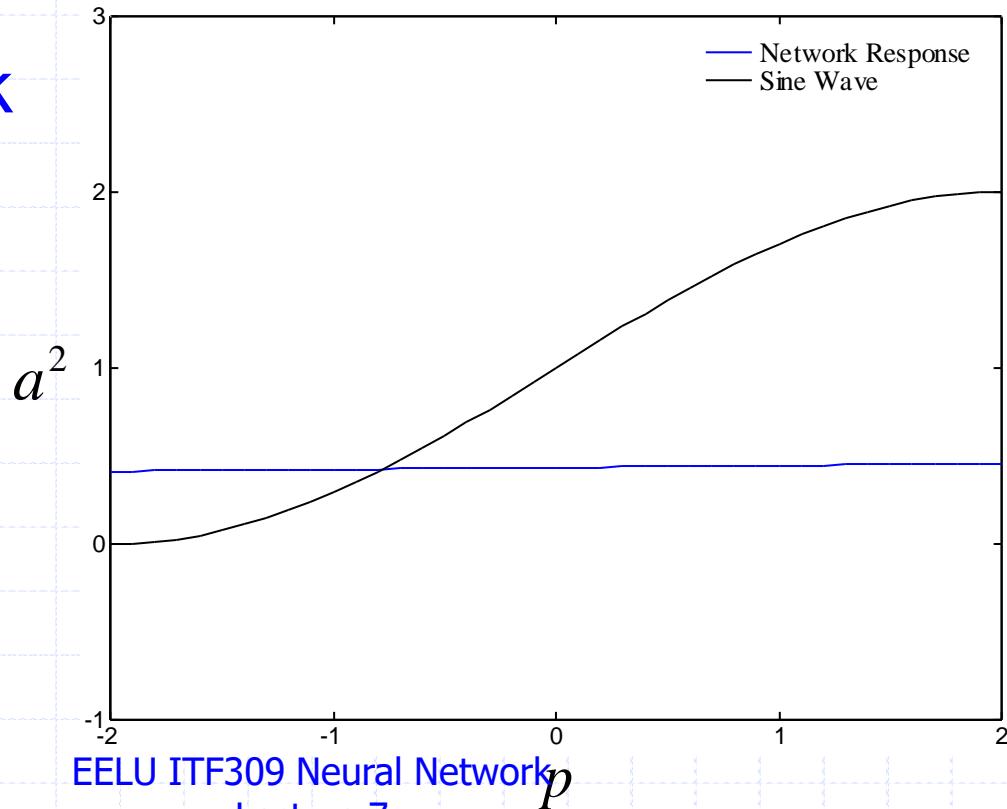
$$\mathbf{W}^1(0) = \begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix}$$

$$\mathbf{b}^1(0) = \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix}$$

$$\mathbf{W}^2(0) = \begin{bmatrix} 0.09 & -0.17 \end{bmatrix}$$

$$\mathbf{b}^2(0) = \begin{bmatrix} 0.48 \end{bmatrix}$$

Initial Network
Response:



Forward Propagation

Initial input: $a^0 = p = 1$

Output of the 1st layer:

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

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

Output of the 2nd layer:

$$a^2 = f^2(W^2 a^1 + b^2) = \text{purelin} \left(\begin{bmatrix} 0.09 & -0.17 \end{bmatrix} \begin{bmatrix} 0.321 \\ 0.368 \end{bmatrix} + \begin{bmatrix} 0.48 \end{bmatrix} \right) = \begin{bmatrix} 0.446 \end{bmatrix}$$

error:

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

Transfer Func. Derivatives

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

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

Backpropagation

◆ The second layer sensitivity:

$$\begin{aligned}s^2 &= -2\dot{\mathbf{F}}^2(n^2)(\mathbf{t} - \mathbf{a}) = -2[\dot{f}^2(n^2)]e \\ &= -2 \cdot 1 \cdot 1.261 = -2.522\end{aligned}$$

◆ The first layer sensitivity:

$$\begin{aligned}\mathbf{s}^1 &= \dot{\mathbf{F}}^1(\mathbf{n}^1)(\mathbf{W}^2)^T \mathbf{s}^2 = \begin{bmatrix} (1-a_1^1)(a_1^1) & 0 \\ 0 & (1-a_2^1)(a_2^1) \end{bmatrix} \begin{bmatrix} w_{1,1}^2 \\ w_{1,2}^2 \end{bmatrix} \mathbf{s}^2 \\ &= \begin{bmatrix} (1-0.321) \cdot 0.321 & 0 \\ 0 & (1-0.368) \cdot 0.368 \end{bmatrix} \begin{bmatrix} 0.09 \\ -0.17 \end{bmatrix} [-2.522] \\ &= \begin{bmatrix} -0.0495 \\ 0.0997 \end{bmatrix}\end{aligned}$$

Weight Update

- ◆ Learning rate $\alpha = 0.1$

$$\begin{aligned}\mathbf{W}^2(1) &= \mathbf{W}^2(0) - \alpha \mathbf{s}^2 (\mathbf{a}^1)^T \\ &= [0.09 \quad -0.17] - 0.1[-2.522][0.321 \quad 0.368] \\ &= [0.171 \quad -0.0772]\end{aligned}$$

$$\mathbf{b}^2(1) = \mathbf{b}^2(0) - \alpha \mathbf{s}^2 = [0.48] - 0.1[-2.522] = [0.732]$$

$$\begin{aligned}\mathbf{W}^1(1) &= \mathbf{W}^1(0) - \alpha \mathbf{s}^1 (\mathbf{a}^0)^T \\ &= [-0.27] - 0.1 \begin{bmatrix} -0.0495 \\ 0.0997 \end{bmatrix}[1] = \begin{bmatrix} -0.265 \\ -0.420 \end{bmatrix} \\ \mathbf{b}^1(1) &= \mathbf{b}^1(0) - \alpha \mathbf{s}^1 = \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix} - 0.1 \begin{bmatrix} -0.0495 \\ 0.0997 \end{bmatrix} = \begin{bmatrix} -0.475 \\ -0.140 \end{bmatrix}\end{aligned}$$

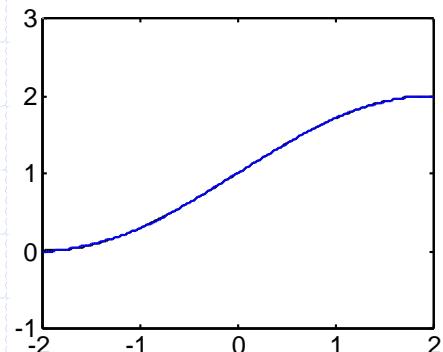
Choice of Network Structure

- ◆ Multilayer networks can be used to approximate almost any function, if we have **enough neurons** in the **hidden layers**.
- ◆ We ***cannot*** say, in general, how many **layers** or how many **neurons** are necessary for adequate performance.

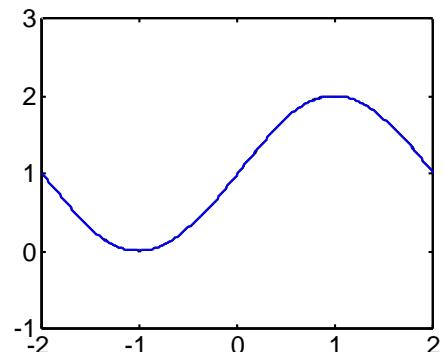
Illustrated Example 1

$$g(p) = 1 + \sin\left(\frac{i\pi}{4} p\right)$$

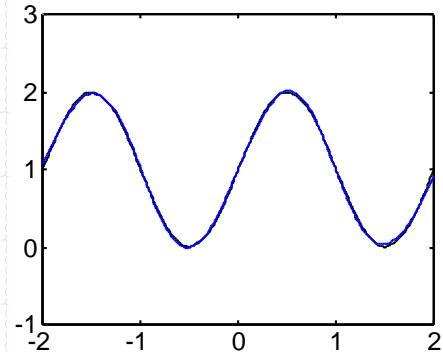
1-3-1 Network



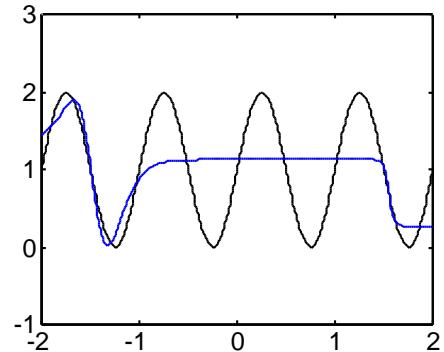
$i = 1$



$i = 2$



$i = 4$

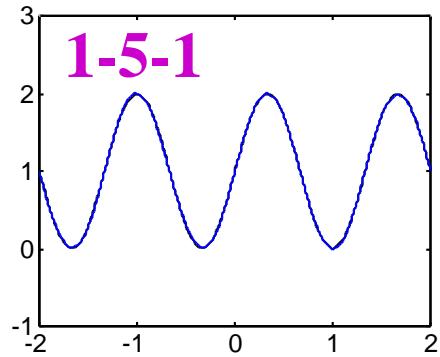
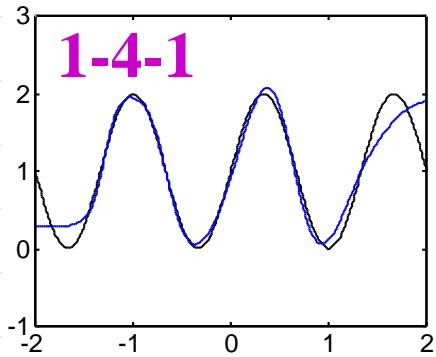
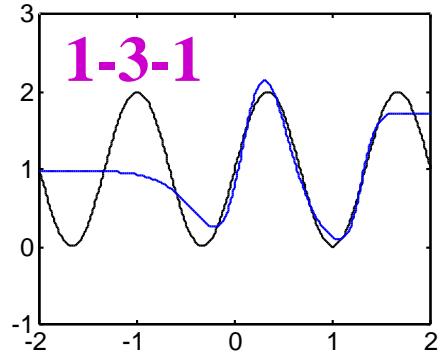
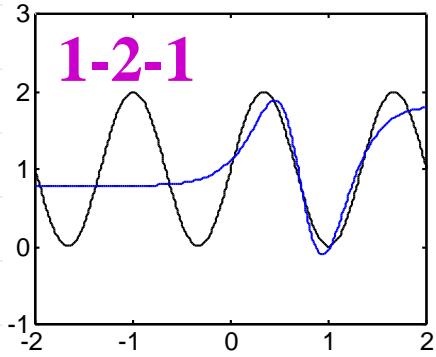


$i = 8$

Illustrated Example 2

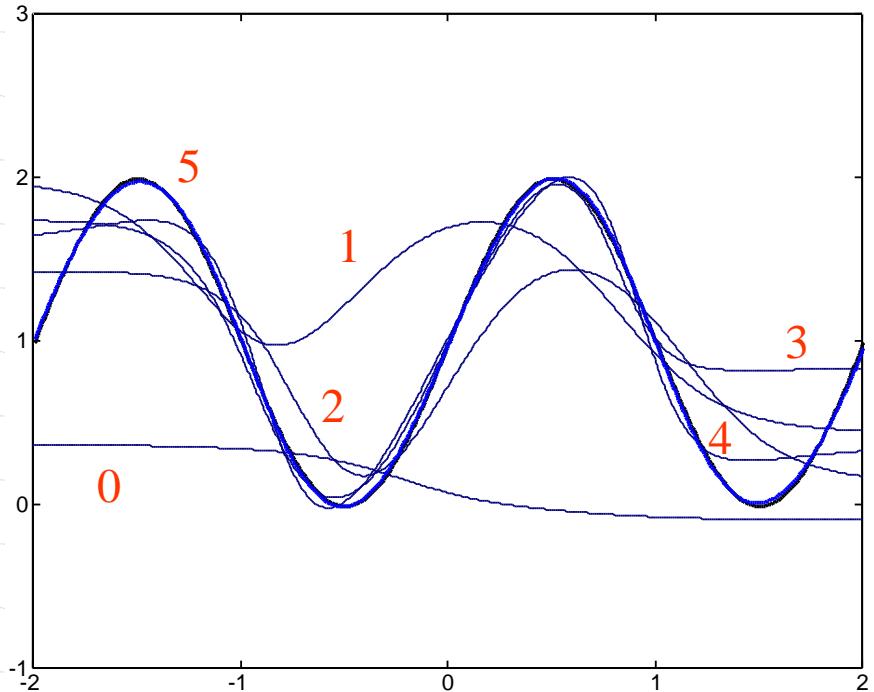
$$g(p) = 1 + \sin\left(\frac{6\pi}{4} p\right)$$

$$-2 \leq p \leq 2$$



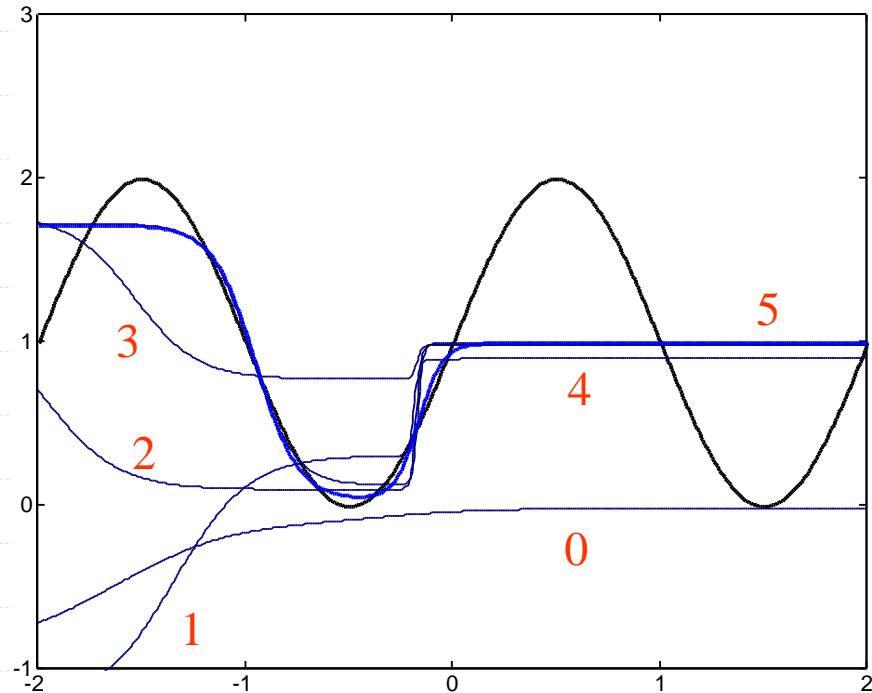
Convergence

$$g(p) = 1 + \sin(\pi p) \quad -2 \leq p \leq 2$$



Convergence to Global Min.

The **numbers** to each curve indicate the sequence of iterations.



Convergence to Local Min.

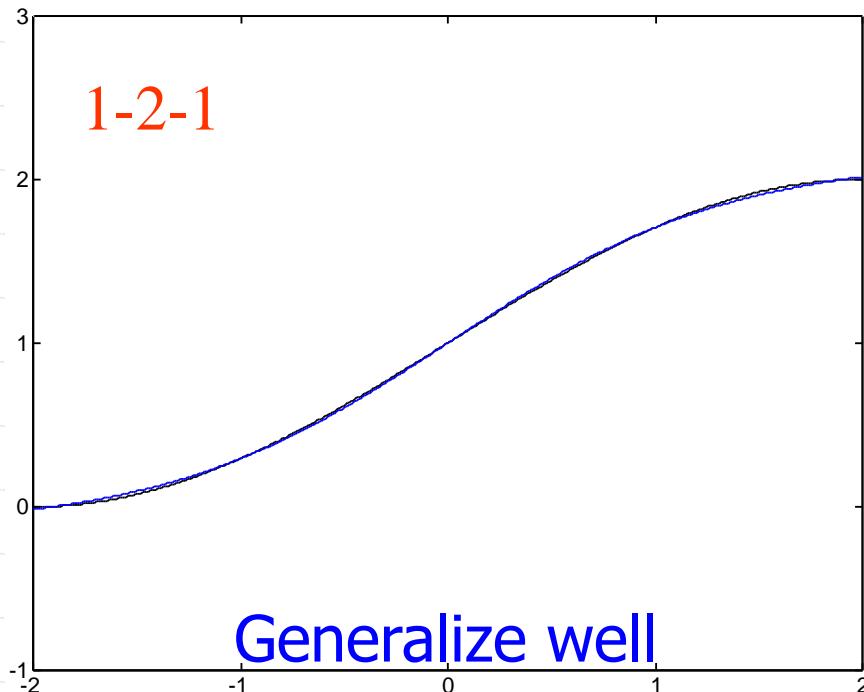
Generalization

- ◆ In most cases the multilayer network is trained with a finite number of examples of proper network behavior: $\{p_1, t_1\}, \{p_2, t_2\}, \dots, \{p_Q, t_Q\}$
- ◆ This **training set** is normally representative of a much larger class of possible input/output pairs.
- ◆ *Can the network successfully generalize what it has learned to the total population?*

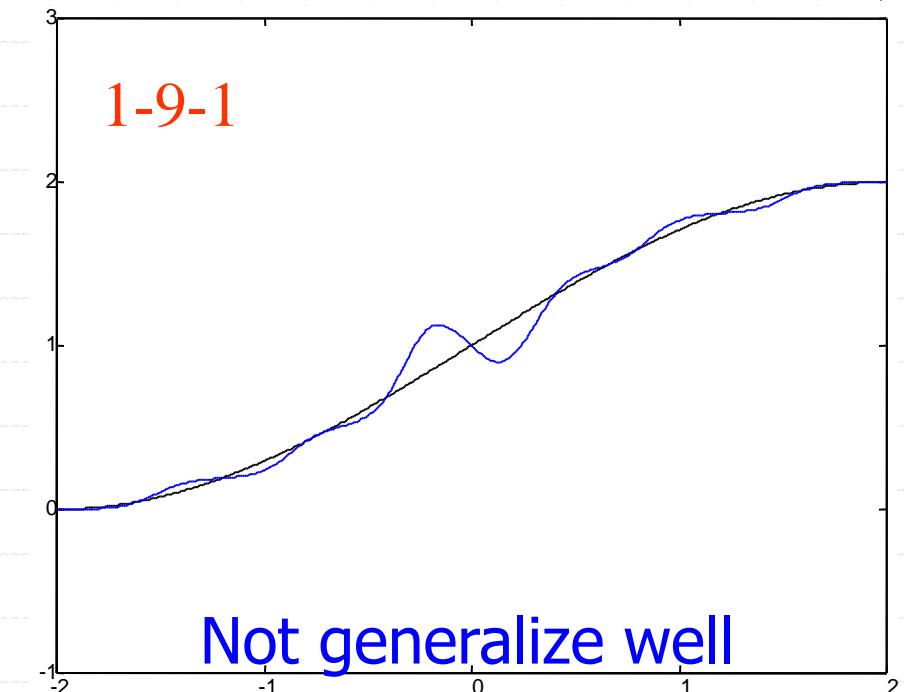
Generalization Example

$$g(p) = 1 + \sin\left(\frac{\pi}{4}p\right)$$

$$p = -2, -1.6, -1.2, \dots, 1.6, 2$$



Generalize well

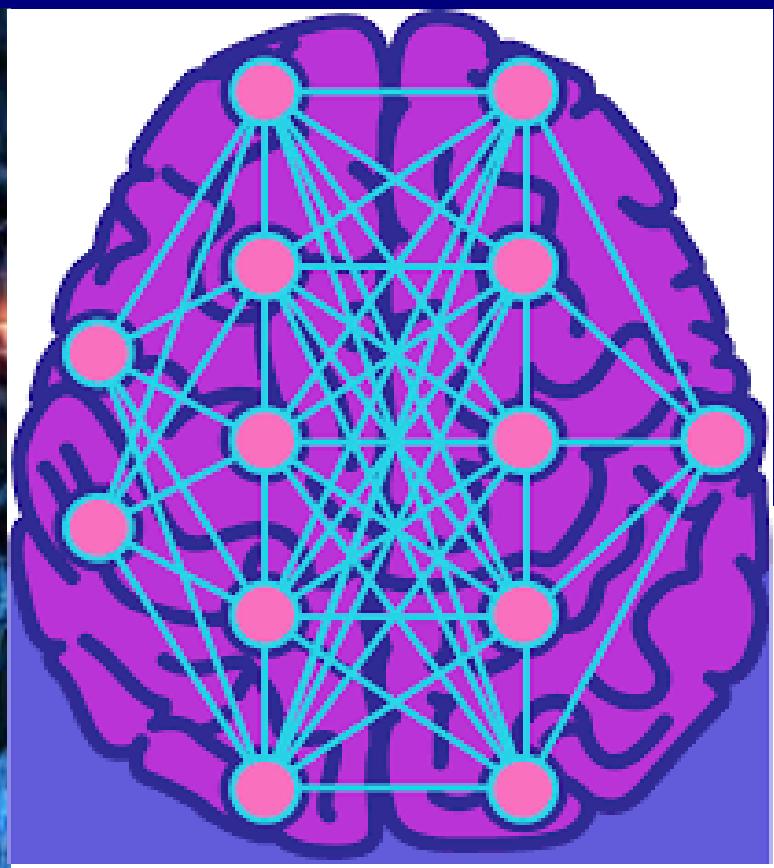


Not generalize well

For a network to be able to generalize, it should have fewer parameters than there are data points in the training set.

ITF309

Artificial Neural Networks



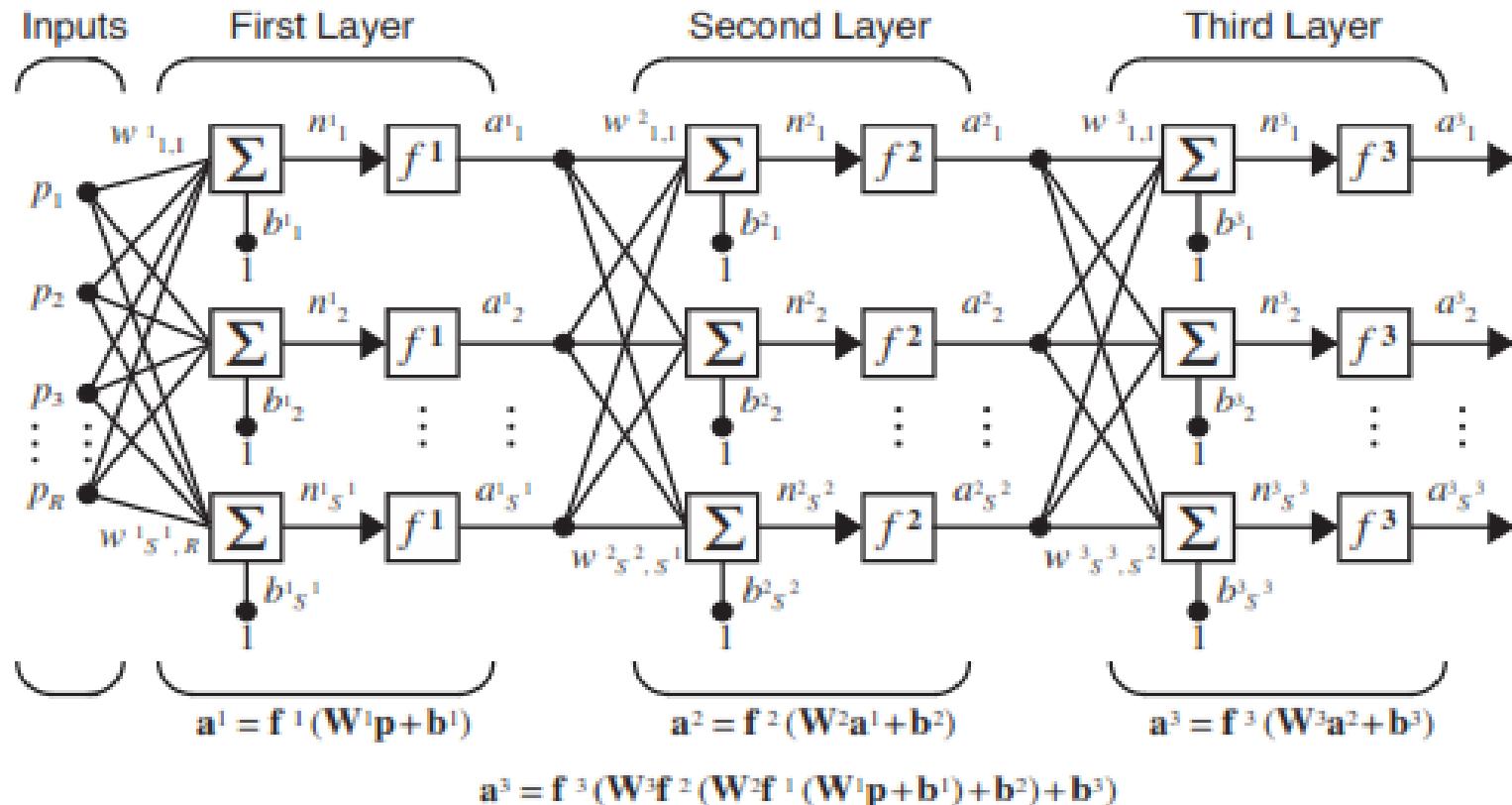
CHAPTER 11

Back-Propagation



Backpropagation

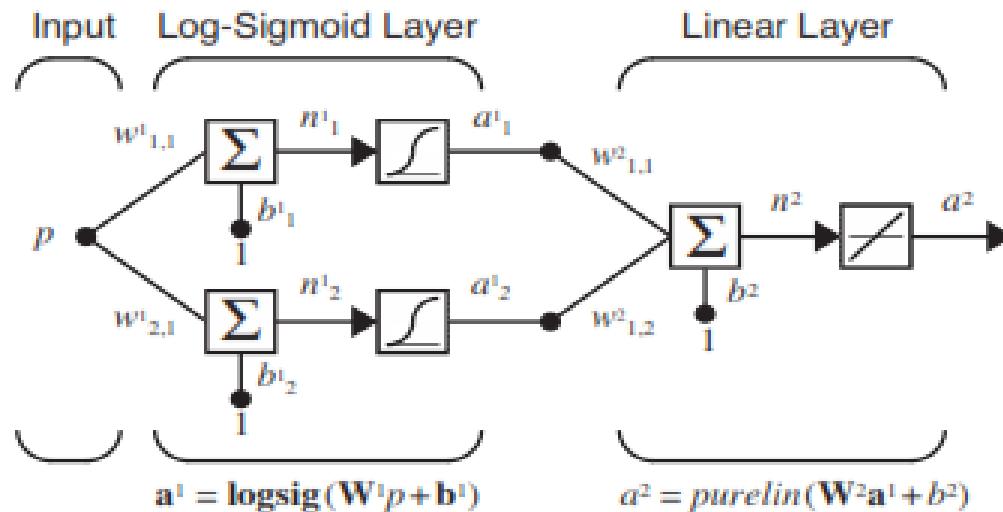
Multilayer Perceptron



R – S¹ – S² – S³ Network

Function Approximation

The following example will illustrate the flexibility of the multilayer perceptron for implementing functions.



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

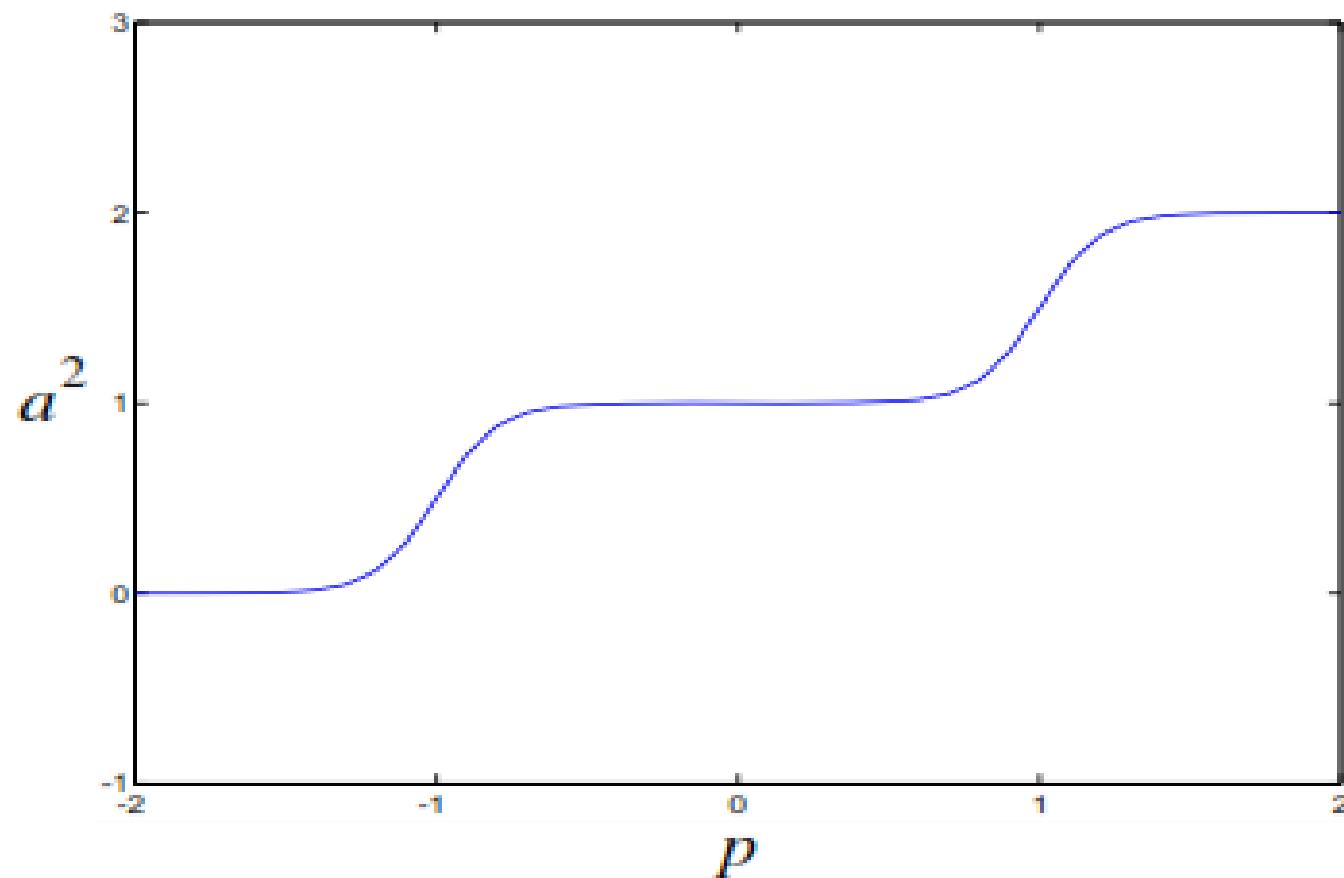
$$f^2(n) = n$$

Nominal Parameter Values

$$w^1_{1,1} = 10 \quad w^1_{2,1} = 10 \quad b^1_1 = -10 \quad b^1_2 = 10$$

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

The network output response for these parameters as the input P is varied over the range [-2, 2].



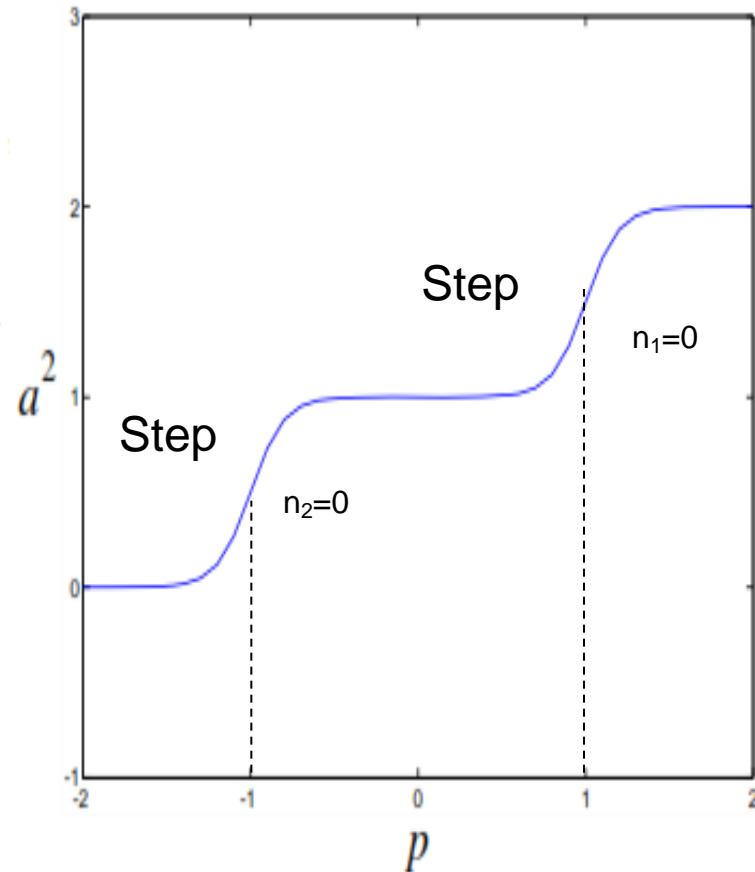
Notice that the response consists of two steps, one for each of the log-sigmoid neurons in the first layer. By adjusting the network parameters we can change the shape and location of each step.

The **centers of the steps** occur where the net input to a neuron in the first layer is zero:

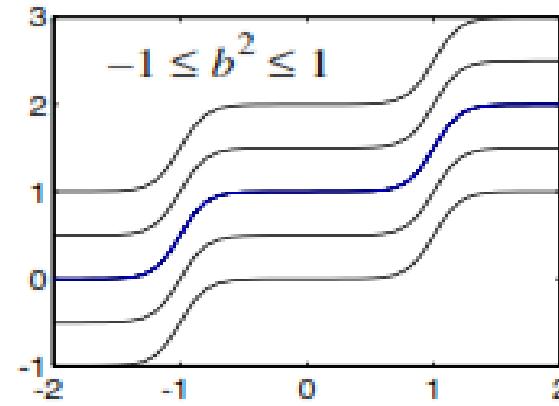
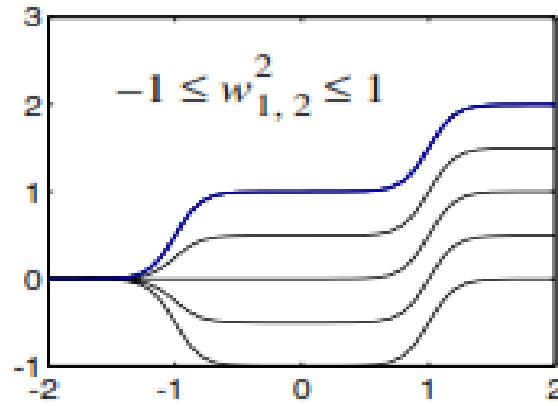
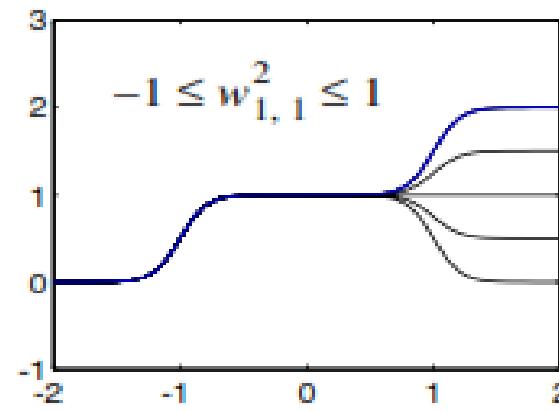
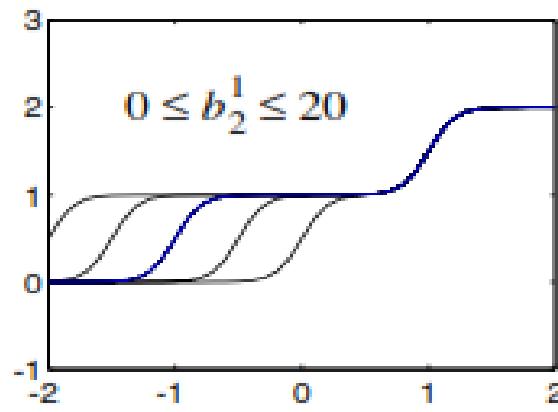
$$n_1^1 = w_{1,1}^1 p + b_1^1 = 0 \Rightarrow p = -\frac{b_1^1}{w_{1,1}^1} = -\frac{-10}{10} = 1$$

$$n_2^1 = w_{2,1}^1 p + b_2^1 = 0 \Rightarrow p = -\frac{b_2^1}{w_{2,1}^1} = -\frac{10}{10} = -1$$

The steepness of each step can be adjusted by changing the network weights.



The blue curve is the nominal response. The other curves correspond to the network response when one parameter at a time is varied over the following ranges

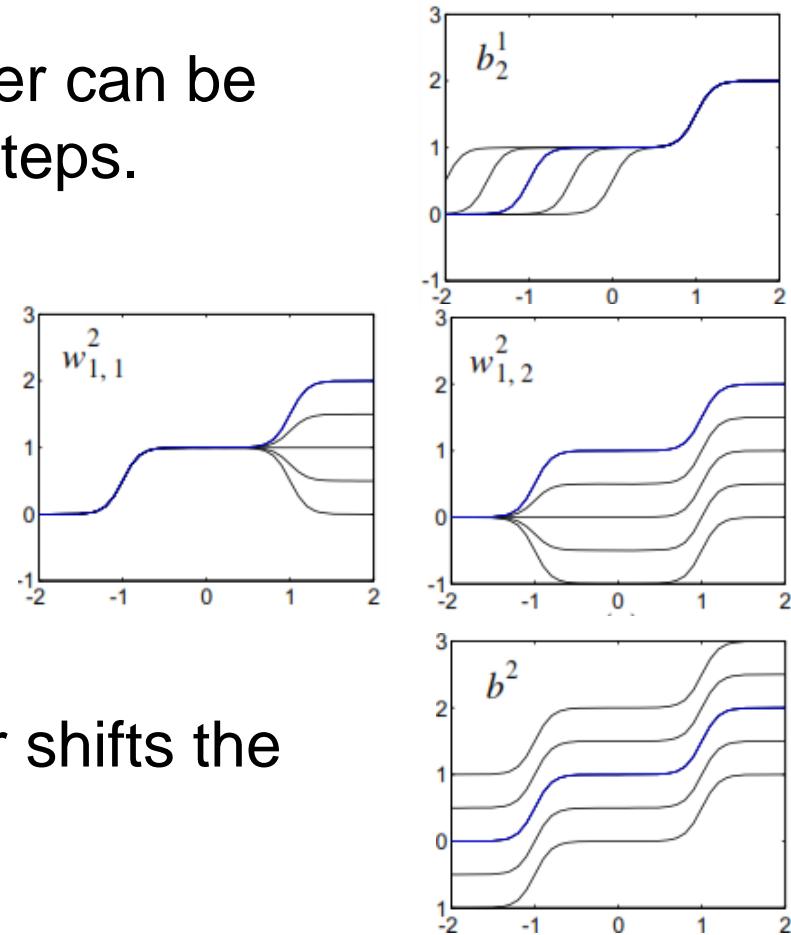


Effect of Parameter Changes on Network Response

The biases in the first (hidden) layer can be used to locate the position of the steps.

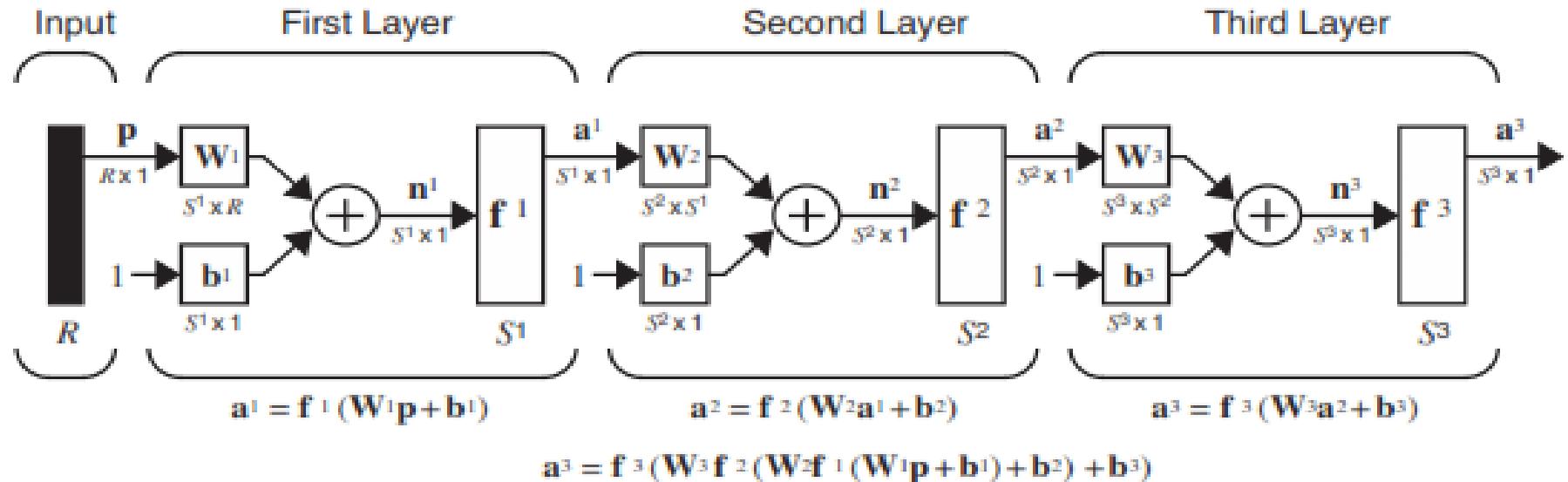
The weights determine the slope of the steps.

The bias in the second (output) layer shifts the entire network response up or down.



- In fact, any 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.
- The next step is to develop an algorithm to train such networks.

The Backpropagation Algorithm



Multilayer network

$$a^{m+1} = f^{m+1}(W^{m+1} a^m + b^{m+1}) \quad m = 0, 2, \dots, M-1$$

The neurons in the first layer receive external inputs:

$$a^0 = p$$

M is the number of layers in the net

The outputs of the neurons in the last layer are considered the net outputs $a = a^M$

Performance Index of Backpropagation

- The performance index of the algorithms is **the mean square error**. (Least Mean Square error or LMS).
- The weights are adjusted, using a gradient descent method, so as to minimize the mean square error.
- The algorithm is provided with a set of examples of proper network behavior:

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

- where \mathbf{p}_q is an input to the network, and \mathbf{t}_q is the corresponding target output.
- As each input is applied to the network, the network output is compared to the target.

- The algorithm should adjust the network parameters in order to minimize the mean square error:

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

- where $E[]$ denotes the expected value, and \mathbf{x} is the vector of network weights and biases

$$\mathbf{x} = \begin{bmatrix} \mathbf{w} \\ \mathbf{b} \end{bmatrix}$$

- If the network has multiple outputs this generalizes to the **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 (the expectation replaced by single iteration k)

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

Where $\hat{F}(\mathbf{x})$ is the estimated performance index.

Approximate Steepest Descent

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

- Now we come to the difficult part – the computation of the partial derivatives.
- For a single-layer linear network these partial derivatives are conveniently computed.
- For the multilayer network the error is not an **explicit** function of the weights in the hidden layers, therefore these derivatives are not computed so easily.
- Because the **error is an indirect function of the weights** in the hidden layers, we will use the **chain rule** of calculus to calculate the derivatives.
- To review the chain rule, suppose that we have a function f that is an **explicit function only of the variable n** .
- We want to take the **derivative of f** with respect to a **third variable w** . The chain rule is then:

Chain Rule

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

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

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

$$\frac{\partial \hat{F}}{\partial b_i^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m}$$

Gradient Calculation

- The second term in each of these equations can be easily computed, since the net input to layer m is an explicit function of the weights and bias in that layer:

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



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

Sensitivity s_i^m : the sensitivity of \hat{F} to changes in the i th element of the net input n at layer m

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

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

$$\frac{\partial \hat{F}}{\partial b_i^m} = s_i^m$$

Steepest Descent

- 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 this becomes:

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

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

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

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

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

Next Step: Compute the Sensitivities (Backpropagation)

Backpropagating the Sensitivities

- It now remains for us to compute the sensitivities \mathbf{s}^m , which requires another application of the chain rule.
- It is this process that gives us the term *backpropagation*, because it describes a recurrence relationship in which the sensitivity at layer m is computed from the sensitivity at layer $m+1$.
- To derive the recurrence relationship for the sensitivities, we will use the following **Jacobian matrix**:

Jacobian Matrix

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

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

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

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

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

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

Backpropagation (Sensitivities)

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

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

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

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

Initialization Last Layer

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

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

$$s_i^M = -2(t_i - a_i) f^M(n_i^M)$$

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

Summary

Forward Propagation

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

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1} \mathbf{a}^m + \mathbf{b}^{m+1}) \quad m = 0, 2, \dots, M-1$$

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

Backpropagation

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

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)({\mathbf{W}^{m+1}}^T \mathbf{s}^{m+1}) \quad m = M-1, \dots, 2, 1$$

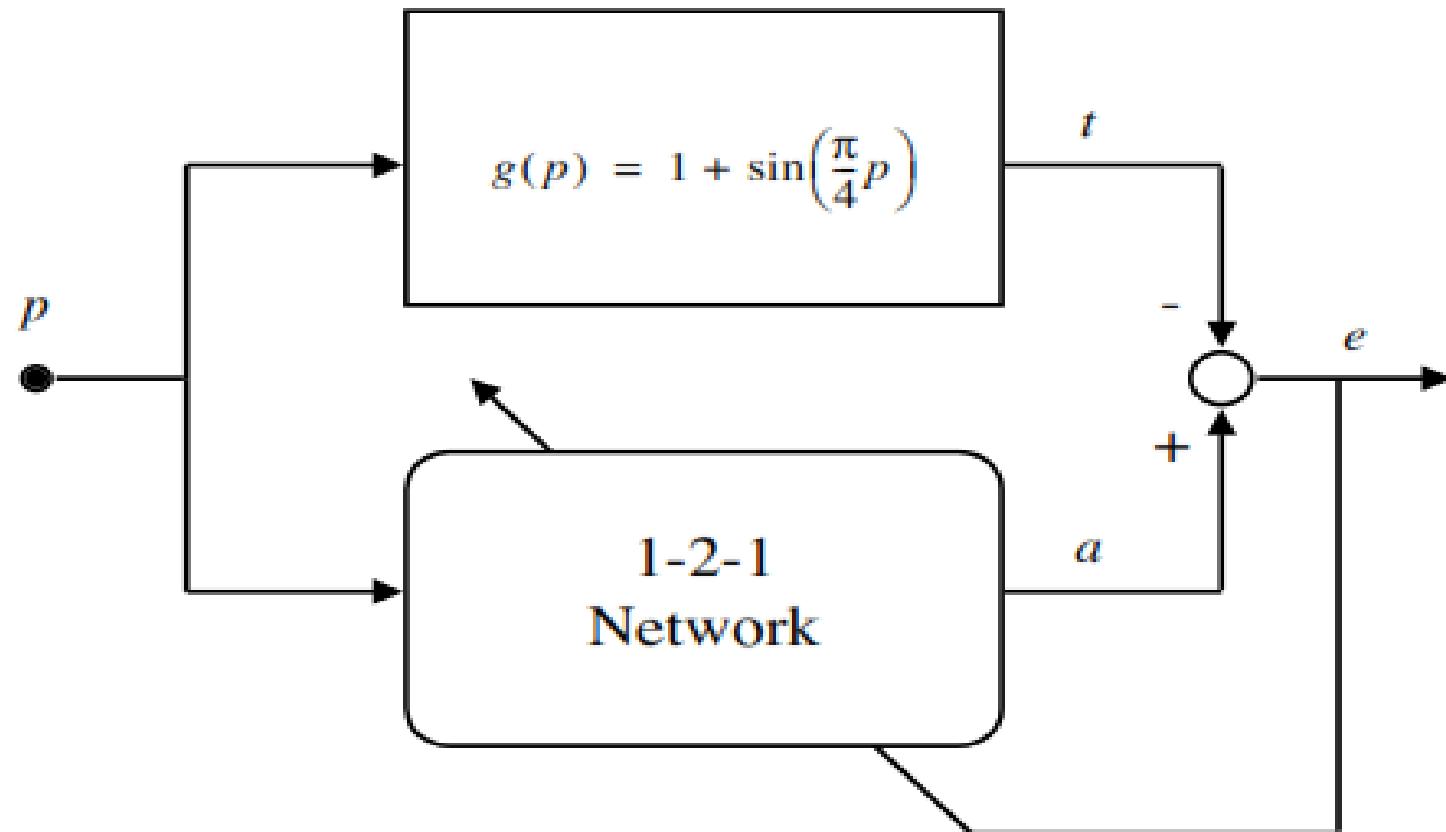
Weight Update

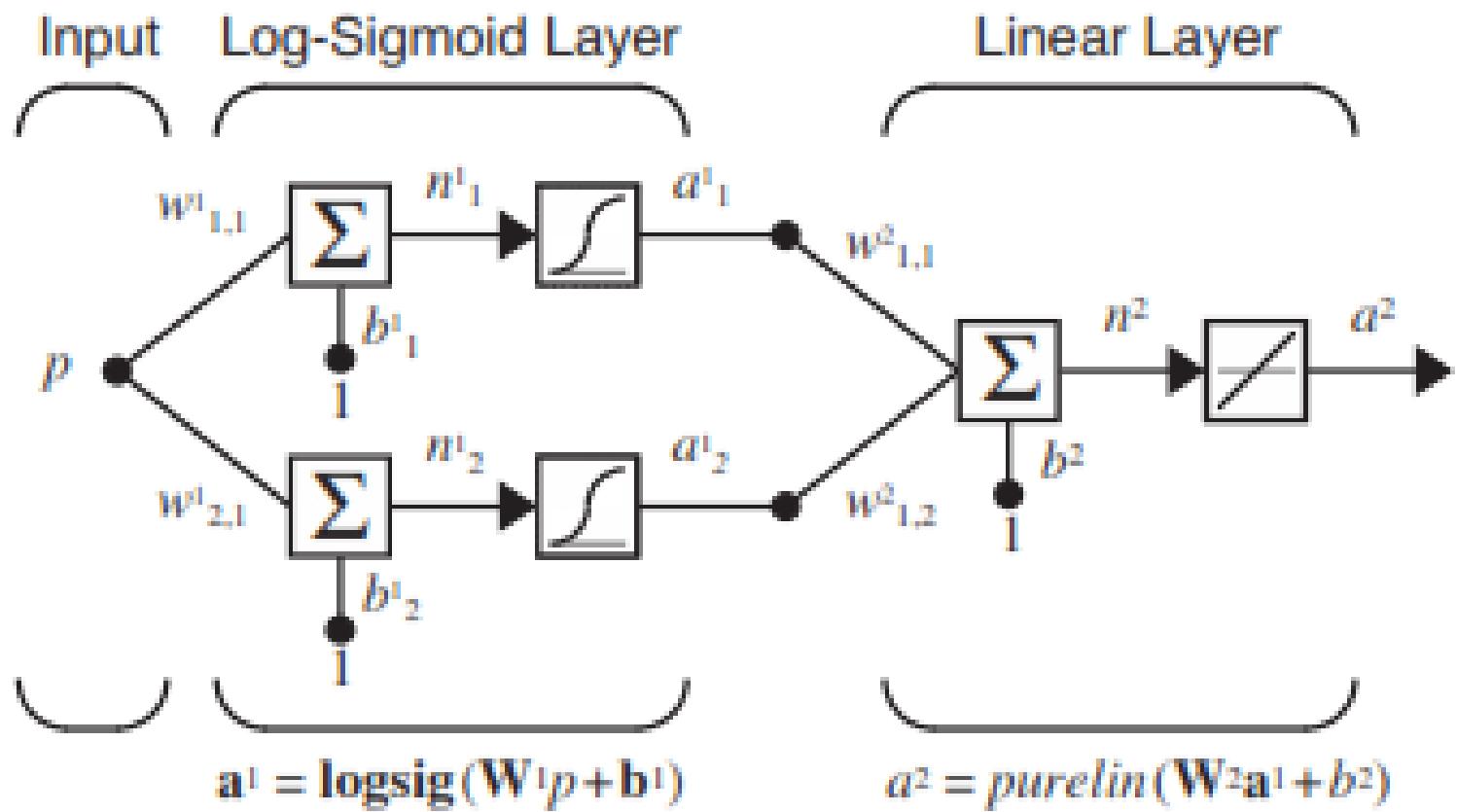
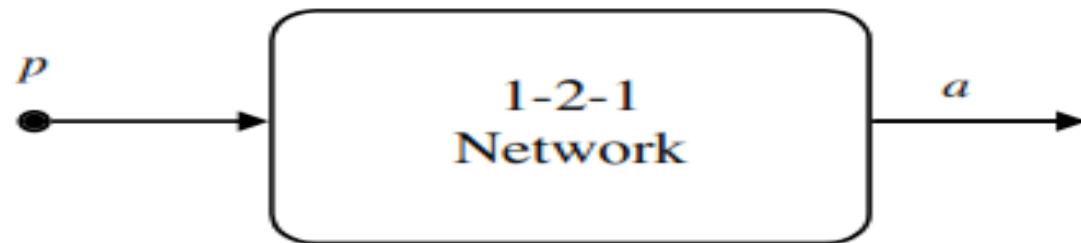
$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T \quad \mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha \mathbf{s}^m$$

Example: Function Approximation

- To illustrate the backpropagation algorithm, let's choose a 1-2-1 network to approximate the function

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





Initial Conditions

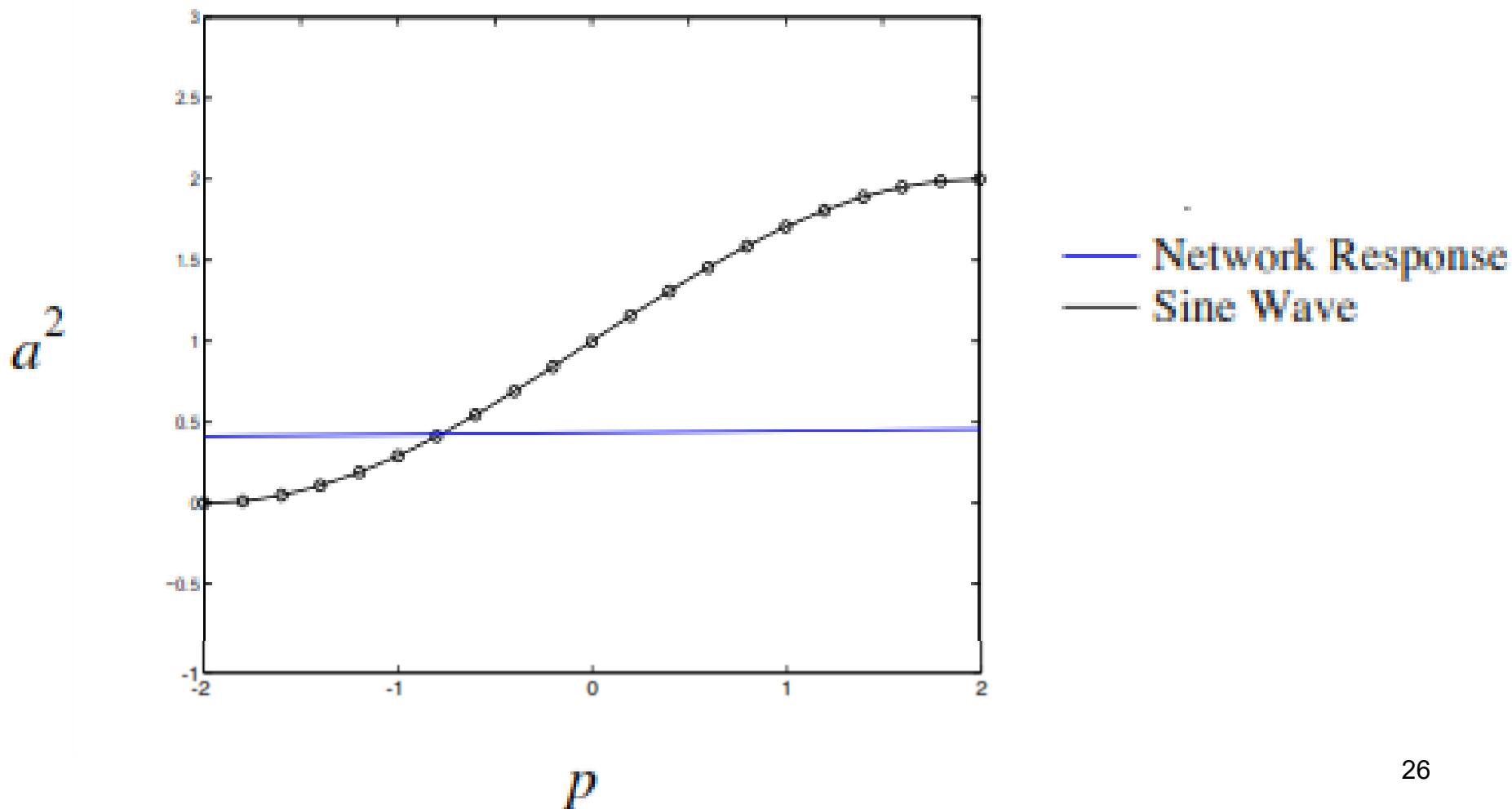
- Before we begin the backpropagation algorithm we need to choose some initial values for the network weights and biases. Generally these are chosen to be small random values.

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

- To obtain our training set we will evaluate this function at several values of p .
- Next, we need to select a training set:
$$\{p_1, t_1\}, \{p_2, t_2\}, \dots, \{p_q, t_q\} .$$
- In this case, we will sample the function at 21 points in the range [-2,2] at equally spaced intervals of 0.2.

Network Response

- The response of the network for these initial values is as shown, along with the sine function we wish to approximate.
- The training points are indicated by the circles



Forward Propagation

- The training points can be presented in any order, but they are often chosen randomly.
- For our initial input we will choose $p=1$, which is the 16th training point:

$$a^0 = p = 1$$

- The output of the first layer is then

$$\begin{aligned} a^1 &= f^1(W^1 a^0 + b^1) = \text{logsig}\left(\begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix} + \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix}\right) = \text{logsig}\left(\begin{bmatrix} -0.75 \\ -0.54 \end{bmatrix}\right) \\ &= \begin{bmatrix} \frac{1}{1 + e^{-0.75}} \\ \frac{1}{1 + e^{-0.54}} \end{bmatrix} = \begin{bmatrix} 0.321 \\ 0.368 \end{bmatrix}. \end{aligned}$$

- The second layer output is

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

- The error would then be

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

Transfer Function Derivatives

- The next stage of the algorithm is to backpropagate the sensitivities.
- Before we begin the backpropagation, we will need the derivatives of the transfer functions $f^1(n)$, and $f^2(n)$
- For the first layer

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

- For the second layer we have

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

Backpropagation

- We can now perform the backpropagation. The starting point is found at the second layer:

$$\mathbf{s}^2 = -2\mathbf{F}^2(\mathbf{n}^2)(\mathbf{t} - \mathbf{a}) = -2[f^2(n^2)](1.261) = -2[1](1.261) = -2.522$$

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

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

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

Weight Update

$$\alpha = 0.1$$

$$\mathbf{W}^2(1) = \mathbf{W}^2(0) - \alpha s^2 (\mathbf{a}^1)^T = \begin{bmatrix} 0.09 & -0.17 \end{bmatrix} - 0.1 \begin{bmatrix} -2.522 \end{bmatrix} \begin{bmatrix} 0.321 & 0.368 \end{bmatrix}$$

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

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

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

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

How to complete the training

- Up to this the first iteration of the back propagation algorithm is completed.
- We next proceed to randomly choose another input from the training set and perform another iteration of the algorithm.
- We continue to iterate until the difference between the network response and the target function reaches some acceptable level.
 - (Note that this will generally take many passes through the entire training set.)

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.
- For example, if each input occurs with equal probability, the mean square error performance index can be written

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

The total gradient of this performance index is

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

- Therefore, to implement a batch version of the backpropagation algorithm, we would step from the forward Eqs. through the sensitive calculations for all of the inputs in the training set.
- Then, the individual gradients would be **averaged** to get the total gradient.
- The update equations for the batch steepest descent algorithm would then be

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

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

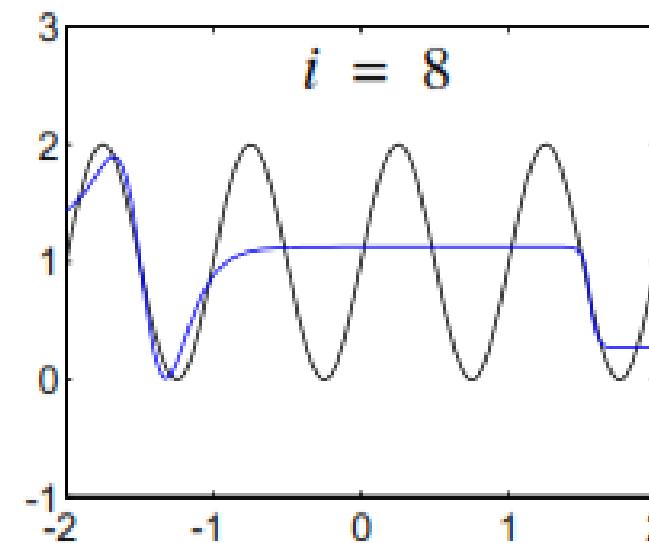
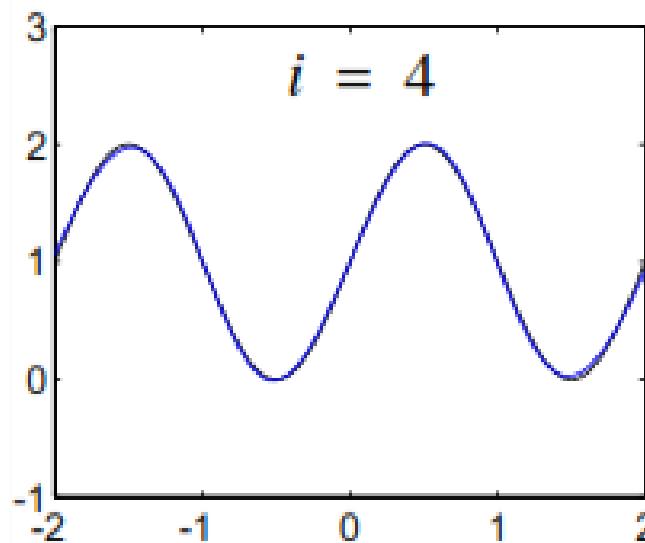
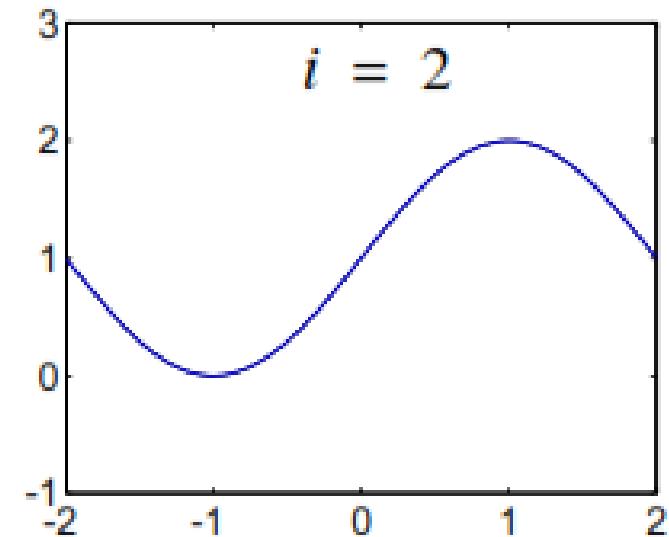
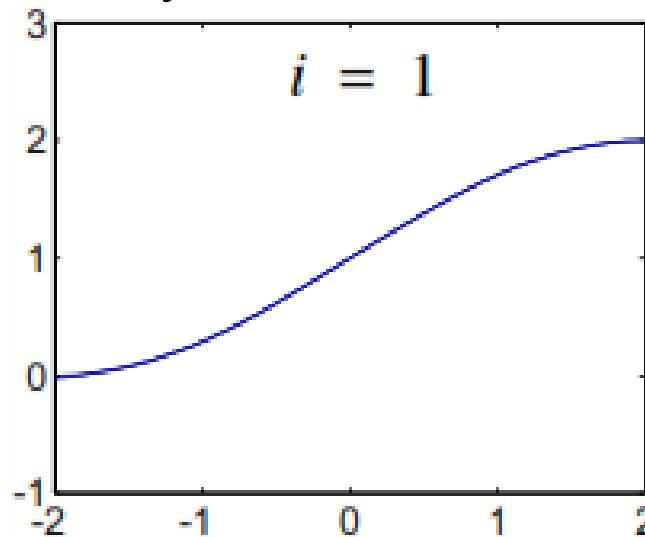
Choice of Network Architecture

- For our first example let's assume that we want to approximate the following functions:

$$g(p) = 1 + \sin\left(\frac{i\pi}{4}p\right) \text{ for } -2 \leq p \leq 2,$$

- where i takes on the values 1, 2, 4 and 8.
- As i is increased, the function becomes more complex, because we will have more periods of the sine wave over the interval $-2 \leq p \leq 2$.
- It will be more difficult for a neural network with a fixed number of neurons in the hidden layers to approximate $g(p)$ as i is increased.
- For this first example we will use a 1-3-1 network,
- The transfer function for the first layer is log-sigmoid and the transfer function for the second layer is linear.
- This type of two-layer network can produce a response that is a sum of three log-sigmoid functions (or as many log-sigmoids as there are neurons in the hidden layer).
- Clearly there is a limit to how complex a function this network can implement.

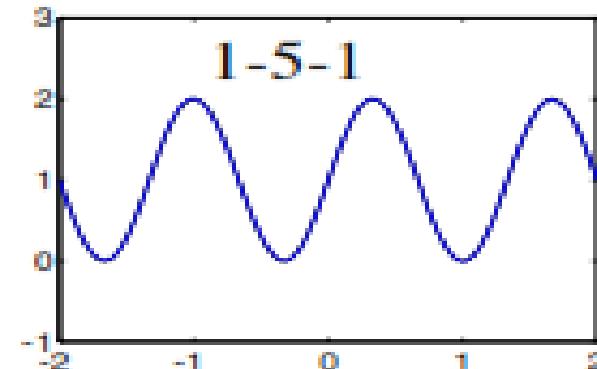
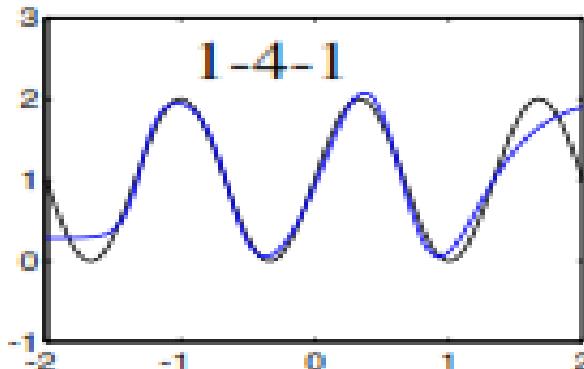
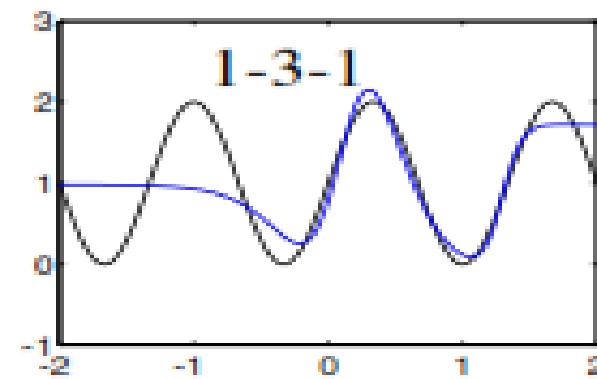
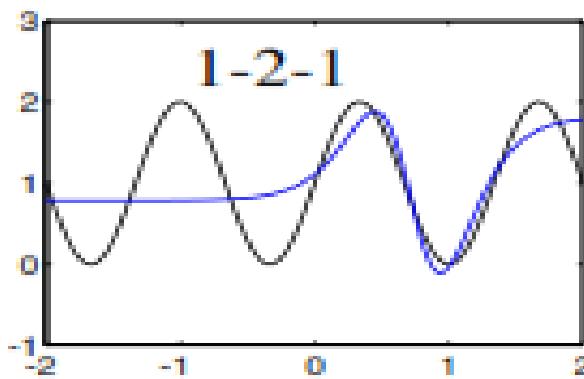
- The final network responses after it has been trained are shown by the blue lines



- We can see that for $i=4$ the 1-3-1 network reaches its maximum capability.
- When $i > 4$ the network is not capable of producing an accurate approximation

Choice of Network Architecture (1-S¹-1), the response of this network is a superposition of S¹ sigmoid functions

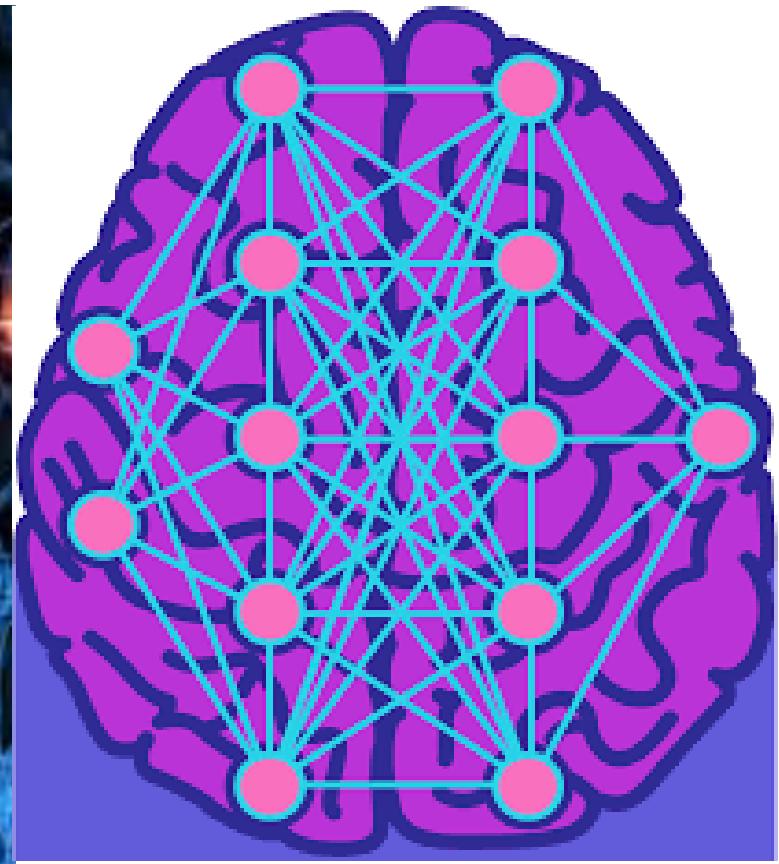
$$g(p) = 1 + \sin\left(\frac{6\pi}{4}p\right)$$



- To summarize these results:
- a 1- S^1 -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^1 sigmoid functions**.
- If we want to approximate a function that has a large number of inflection points, we will need to have a large number of neurons in the hidden layer.

ITF309

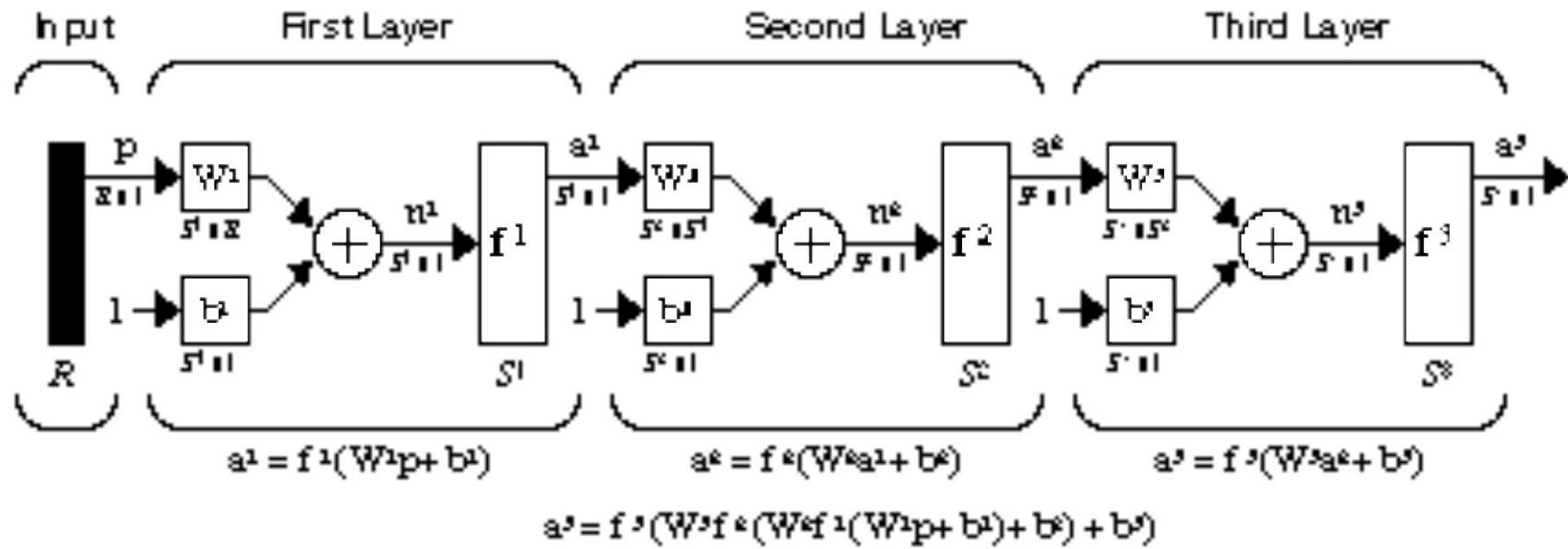
Artificial Neural Networks



CHAPTER 12

Variations of back-propagation

Multilayer Network



$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1} \mathbf{a}^m + \mathbf{b}^{m+1}) \quad m = 1, 2, \dots, M-1$$

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

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

Summary of BP Algorithm

- The *first step* is to propagate the **input forward** through the network:

$$\mathbf{a}_0 = \mathbf{p}$$

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}), m = 0, 1, 2, \dots, M - 1$$

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

- The *second step* is to propagate the **sensitivities backward** through the network:

- Output layer:

- Hidden layer: $\mathbf{s}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t} - \mathbf{a})$

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \mathbf{s}^{m+1}, m = M - 1, \dots, 2, 1$$

- The *final step* is to **update the weights and biases**:

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

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

Practical Issues

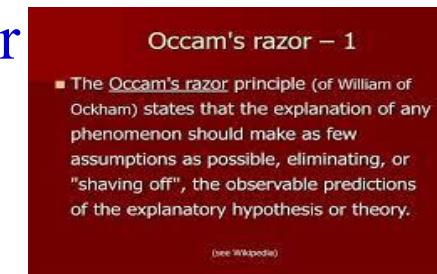
- While in theory a multiple layer neural network with nonlinear transfer function trained using backpropagation is sufficient to approximate any function or solve any recognition problem, there are practical issues
 - *What is the optimal architecture for a particular problem/application?*
 - *What is the performance on unknown test data?*
 - *Will the network converge to a good solution?*
 - *How long does it take to train a network?*

The Issue of Generalization

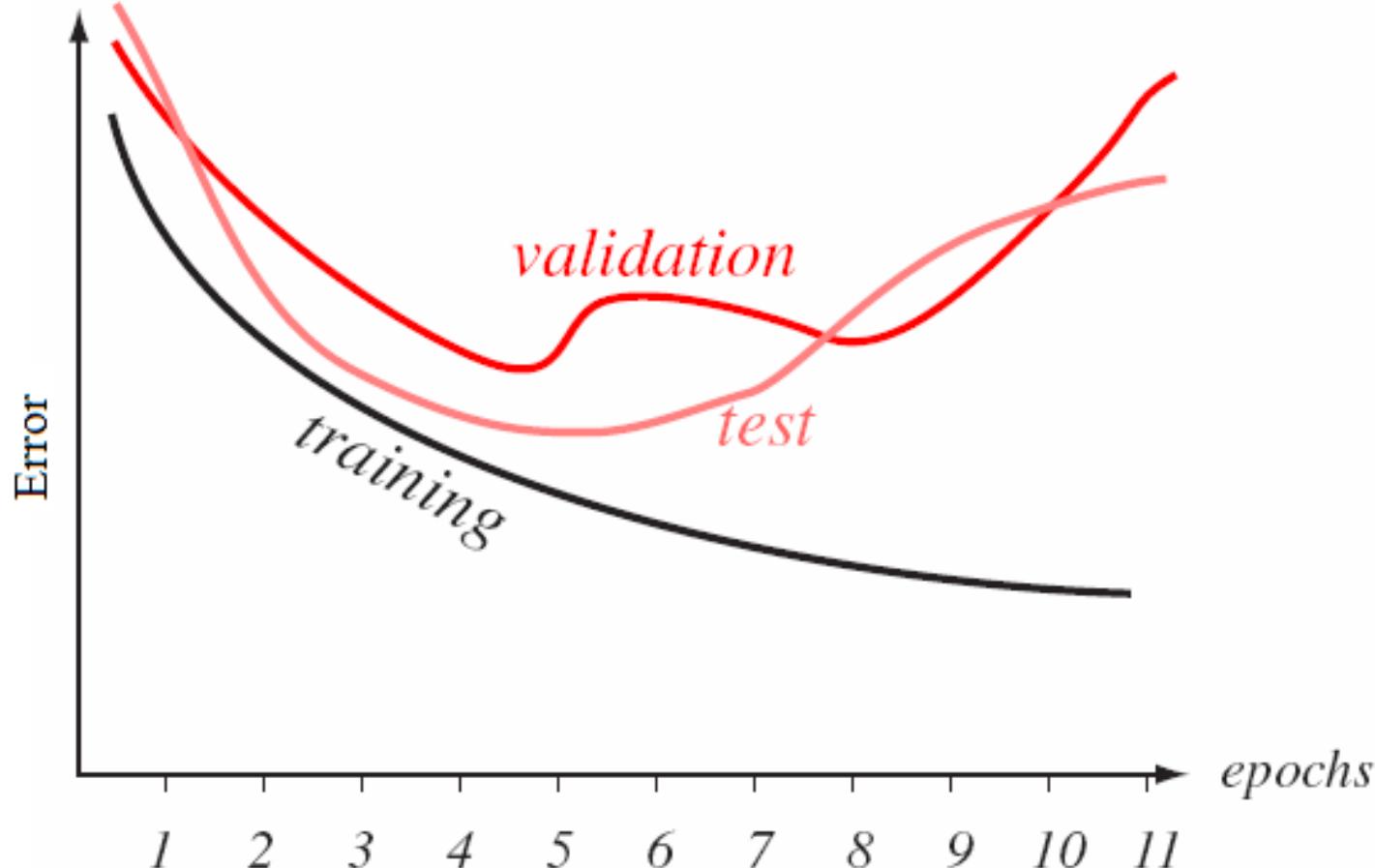
- We are interested in a neural network trained only on a training set will work well for novel and unseen test data
 - For example, for face recognition, a neural network can only recognize those in the training set is not very useful
 - Generalization is one of the most fundamental problems in neural networks and many other recognition techniques

Improving Generalization

- Heuristics
 - A neural network should have fewer parameters than the number of data points in the training set
 - Simpler neural networks should be preferred over complicated ones, known as Occam's razor
 - More domain specific knowledge
- Cross validation
 - Divide the labeled examples into training and validation sets
 - Stop training when the error on the validation set increases



Cross Validation



Convergence Issues

- A neural network may converge to a bad solution
 - Train several neural networks from different initial conditions
- The convergence is slow
 - Practical techniques
 - Variations of basic backpropagation algorithms

Practical Techniques for Improving Backpropagation

- Transfer functions
 - Prior information to choose appropriate transfer functions
 - Parameters for the sigmoid function

Practical Techniques for Improving Backpropagation

- Scaling input
 - We can standardize each feature component to have zero mean and the same variance
- Target values
 - For pattern recognition applications, use 1 for the target category and -1 for non-target category
- Training with noise

Practical Techniques for Improving Backpropagation

- Manufacturing data
 - If we have knowledge about the sources of variation among the inputs, we can manufacture training data
 - For face detection, we can rotate and enlarge / shrink the training images
- Initializing weights
 - If we use standardized data, we want positive and negative weights as well from a uniform distribution
 - Uniform learning

Practical Techniques for Improving Backpropagation

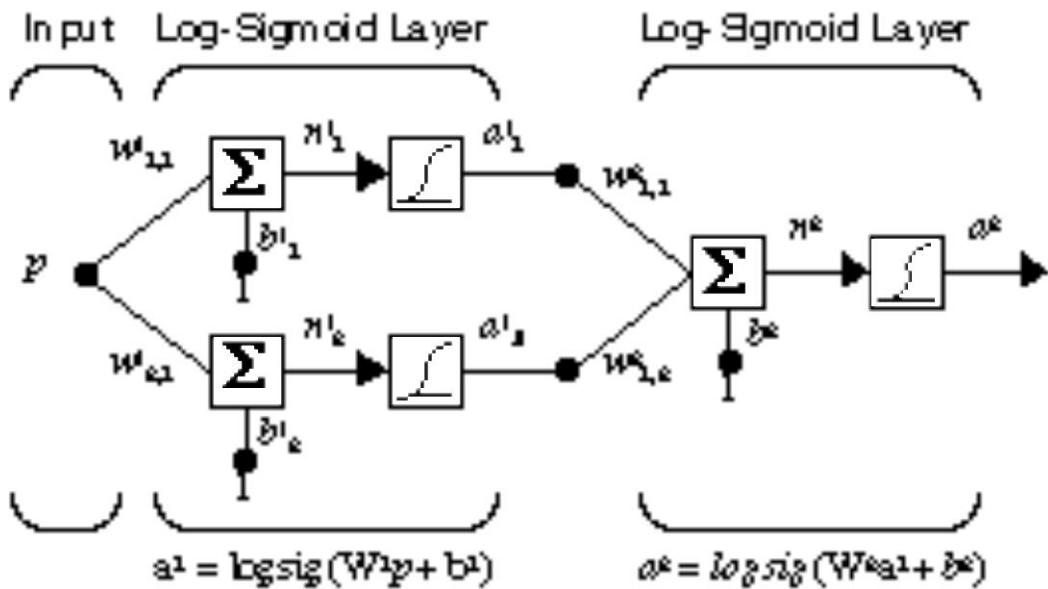
- Training protocols
 - Epoch corresponds to a single presentation of all the patterns in the training set
 - Stochastic training
 - Training samples are chosen randomly from the set and the weights are updated after each sample
 - Batch training
 - All the training samples are presented to the network before weights are updated
 - On-line training
 - Each training sample is presented once and only once
 - There is no memory for storing training samples

Speeding up Convergence

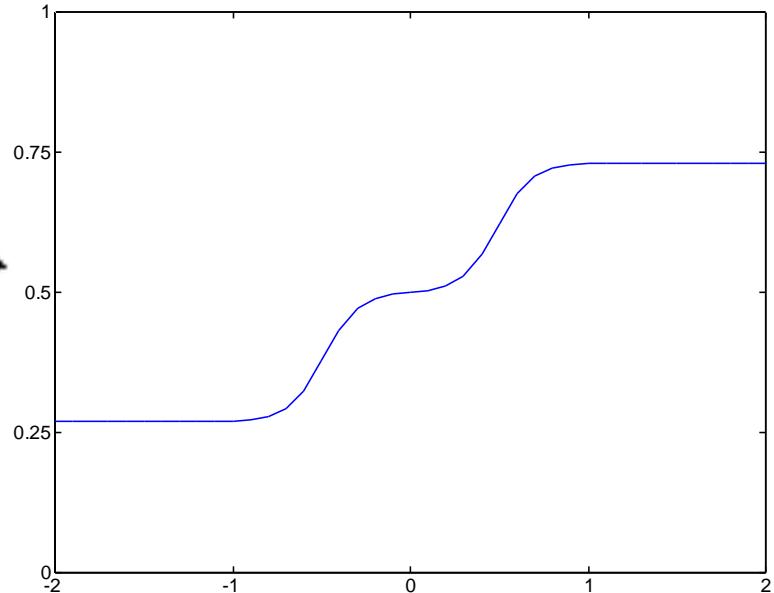
- Heuristics
 - **Momentum**
 - Variable learning rate
- Conjugate gradient
- Second-order methods
 - Newton's method
 - **Levenberg-Marquardt algorithm**

Performance Surface Example

Network Architecture



Nominal Function

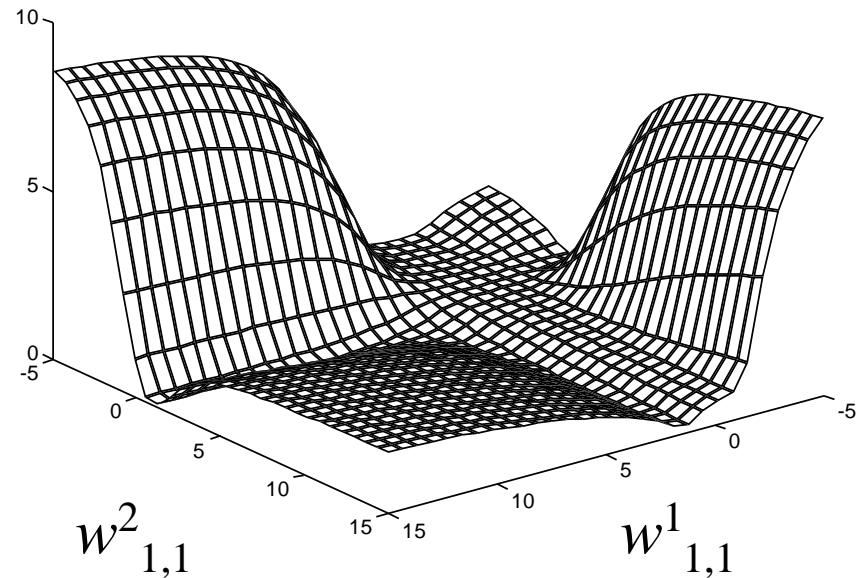
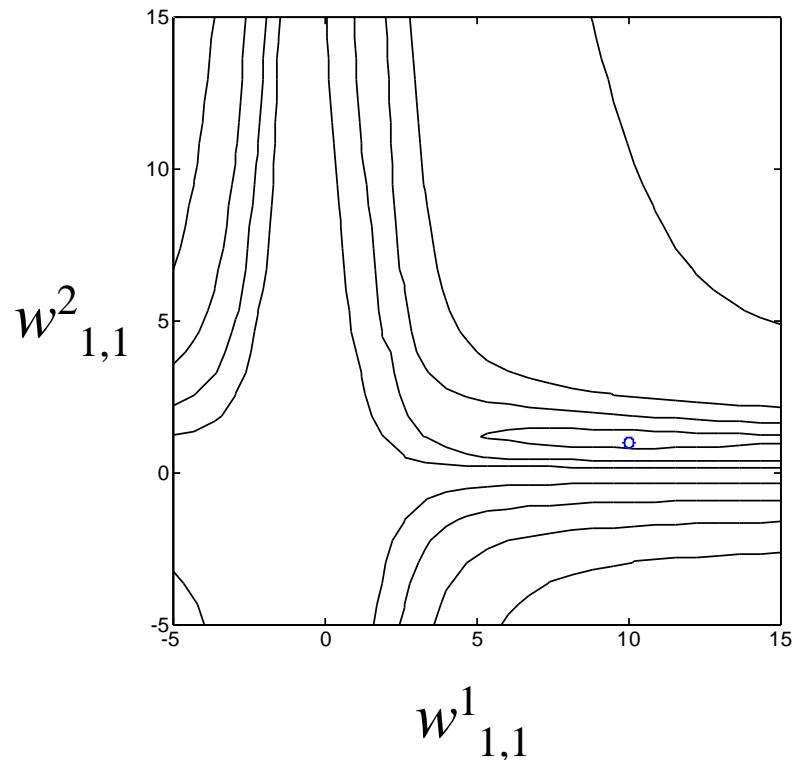


Parameter Values

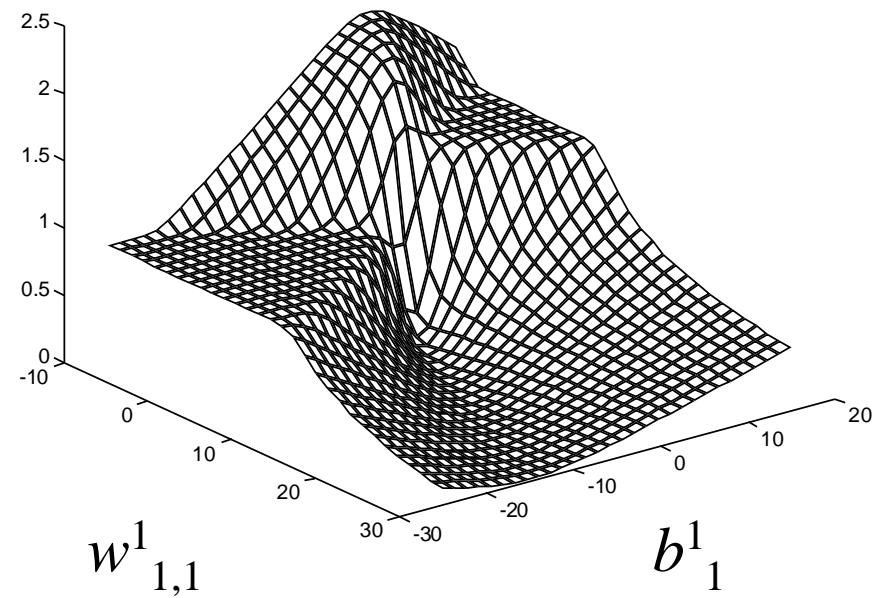
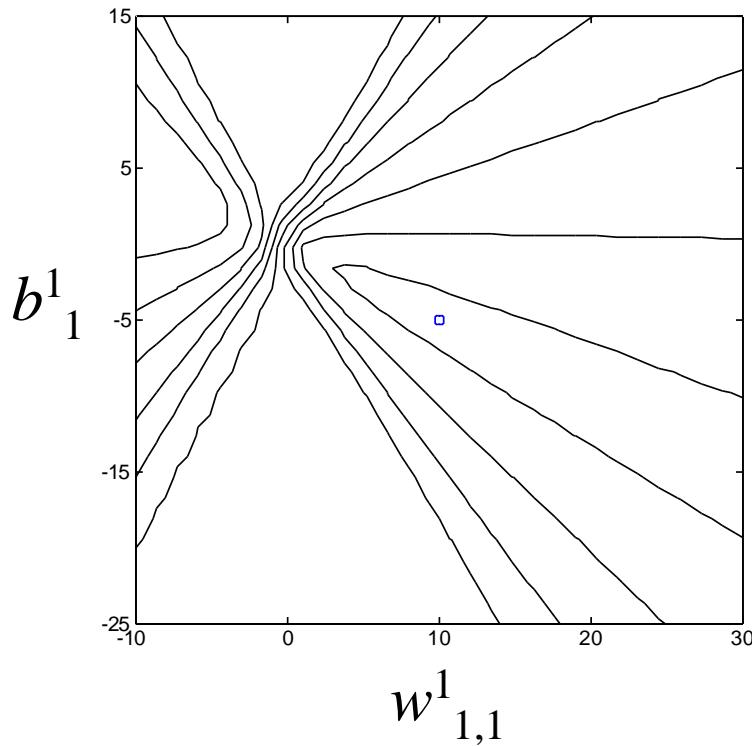
$$w_{1,1}^1 = 10 \quad w_{2,1}^1 = 10 \quad b_1^1 = -5 \quad b_2^1 = 5$$

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

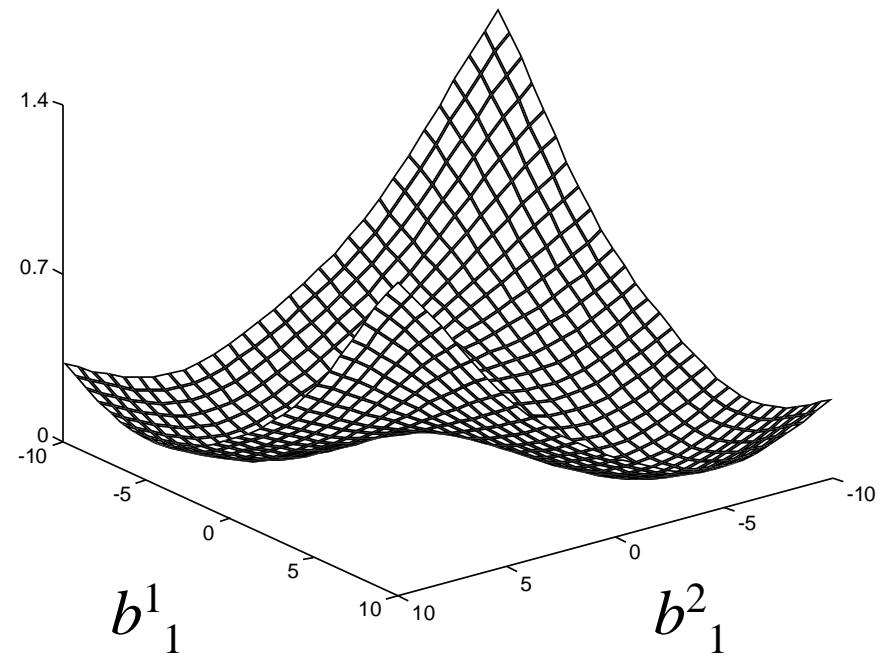
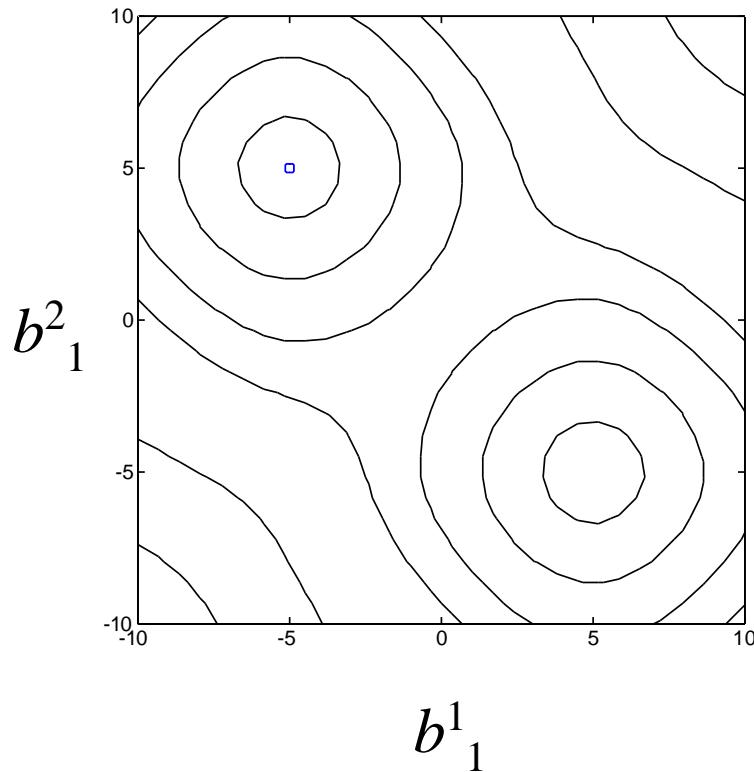
Squared Error vs. $w_{1,1}^1$ and $w_{1,1}^2$



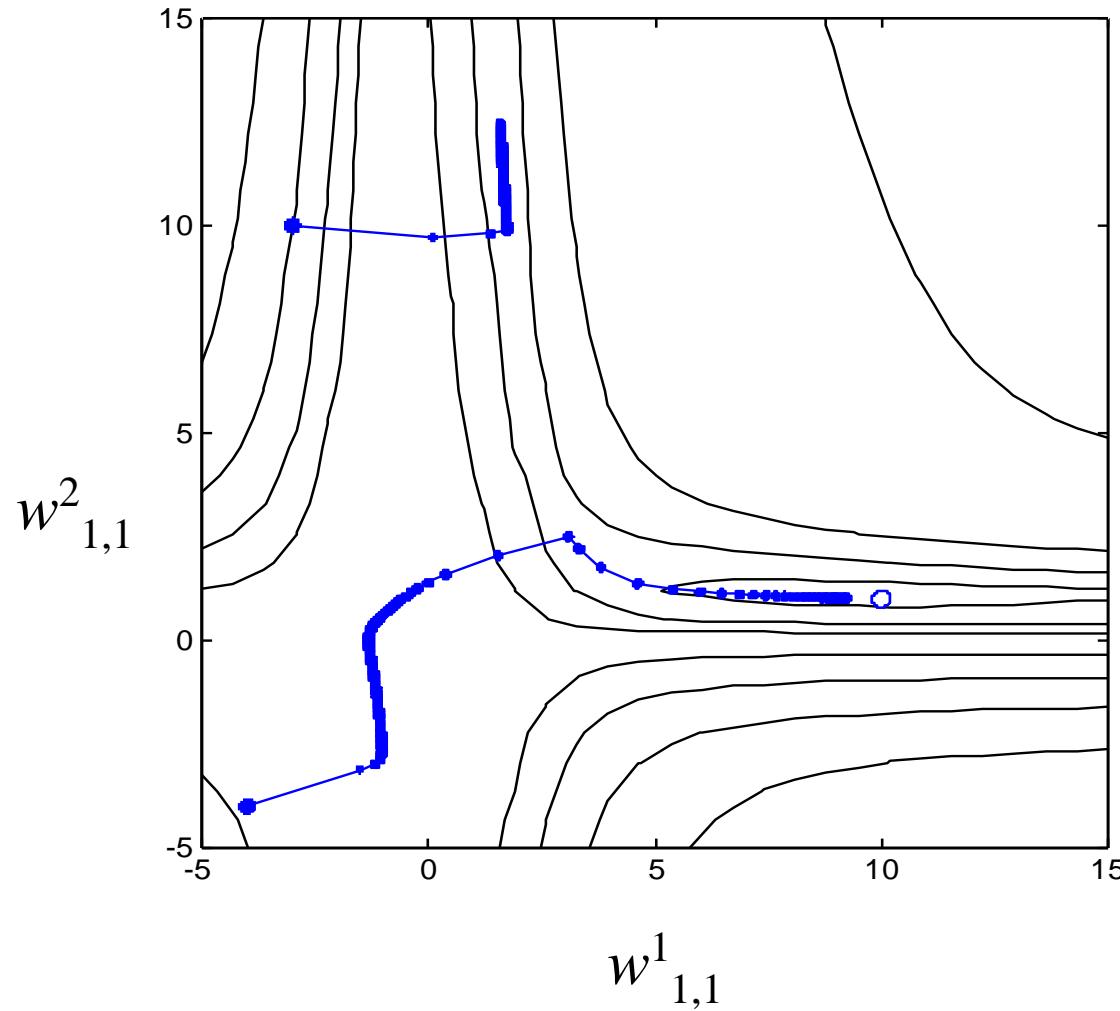
Squared Error vs. $w^1_{1,1}$ and b^1_1



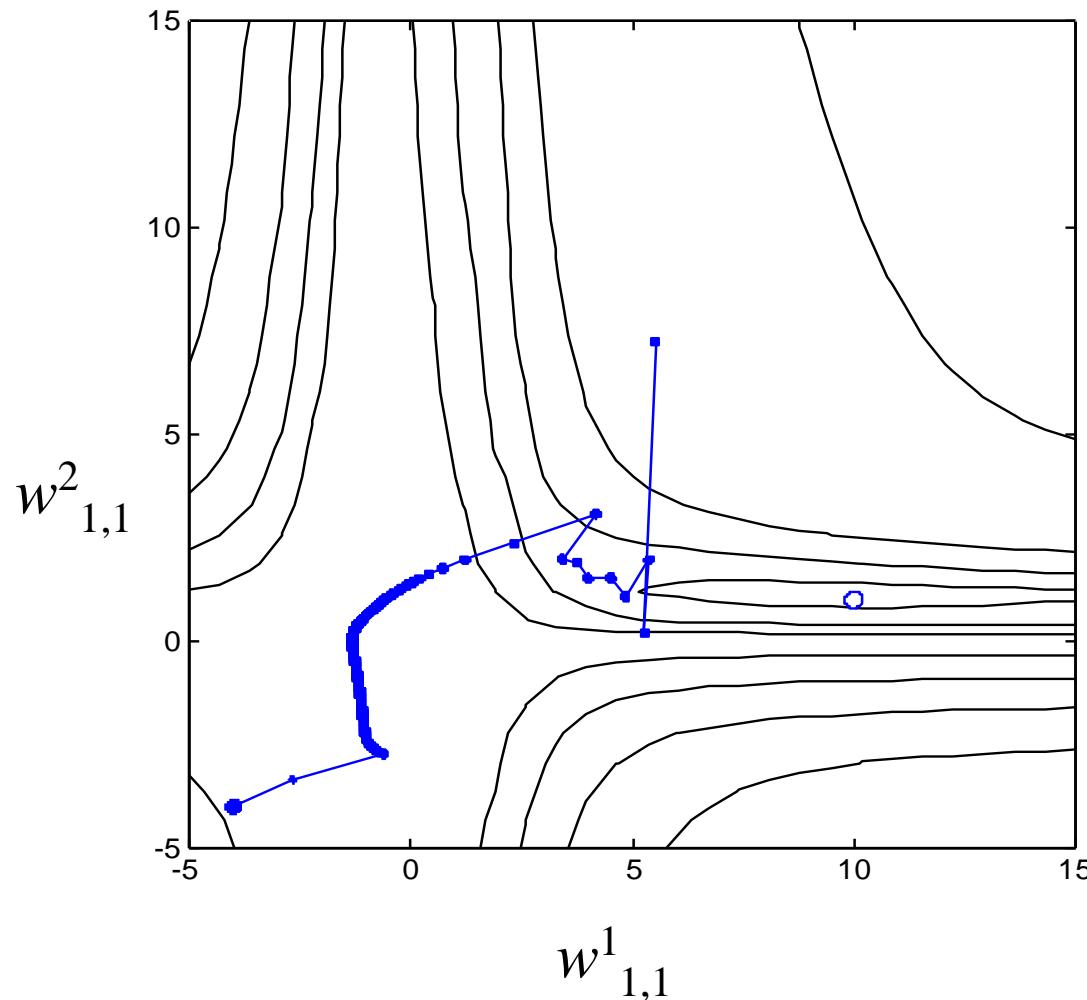
Squared Error vs. b^1_1 and b^2_1



Convergence Example



Learning Rate Too Large



Momentum

Filter

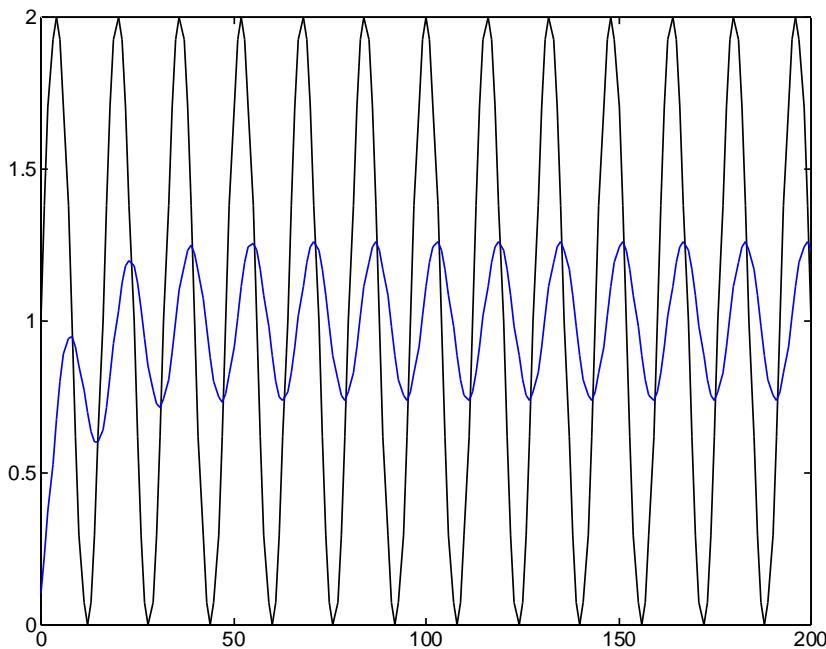
$$y(k) = \gamma y(k-1) + (1-\gamma)w(k)$$

$$0 \leq \gamma < 1$$

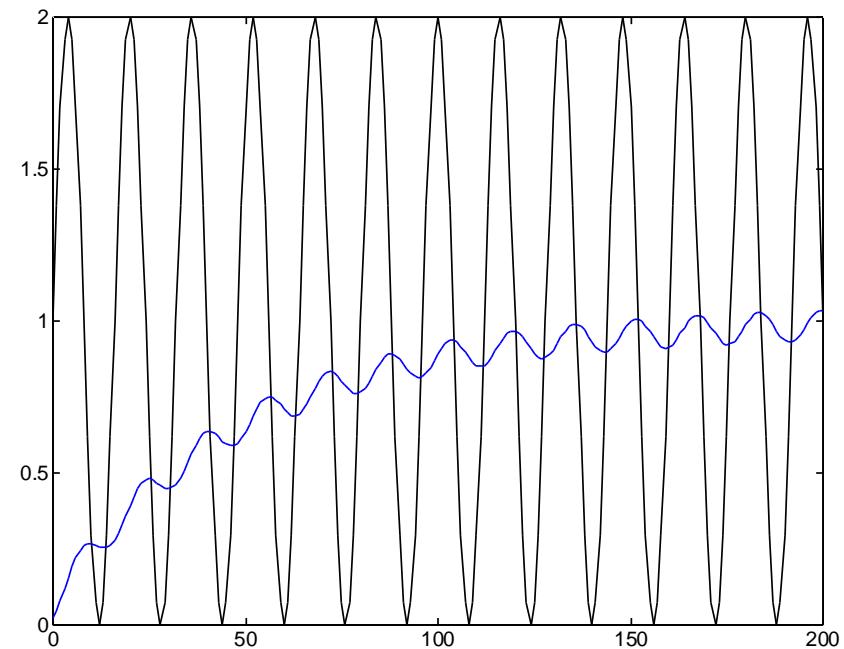
Example

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

$$\gamma = 0.9$$



$$\gamma = 0.98$$



Momentum Backpropagation

Steepest Descent Backpropagation (SDBP)

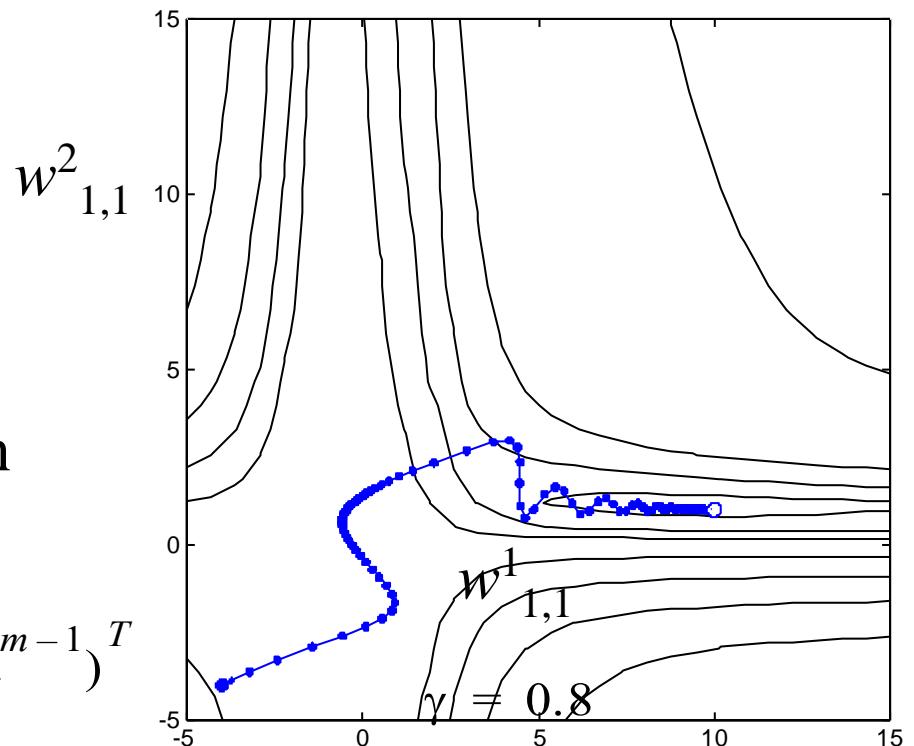
$$\Delta \mathbf{W}^m(k) = -\alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T$$

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

Momentum Backpropagation (MOBP)

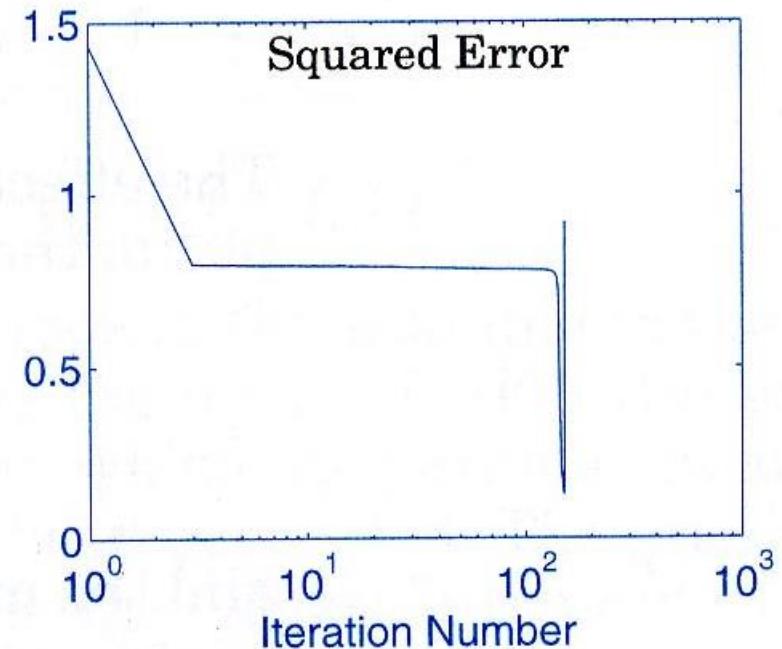
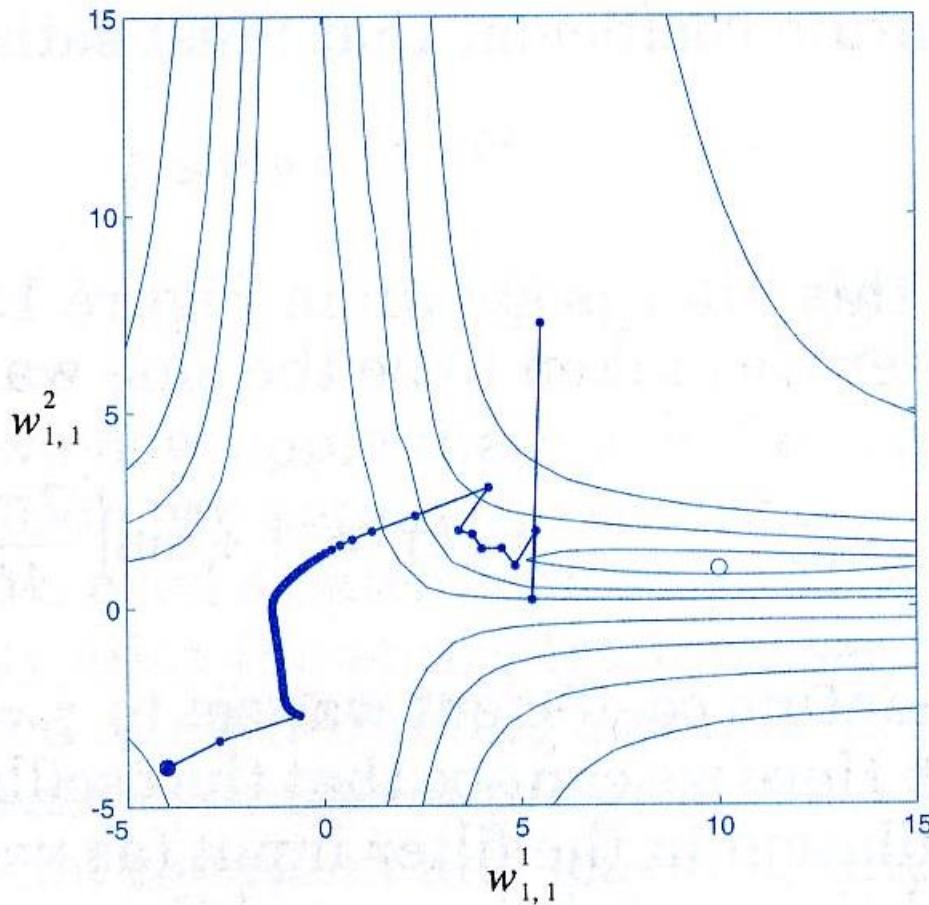
$$\Delta \mathbf{W}^m(k) = \gamma \Delta \mathbf{W}^m(k-1) - (1-\gamma)\alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T$$

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

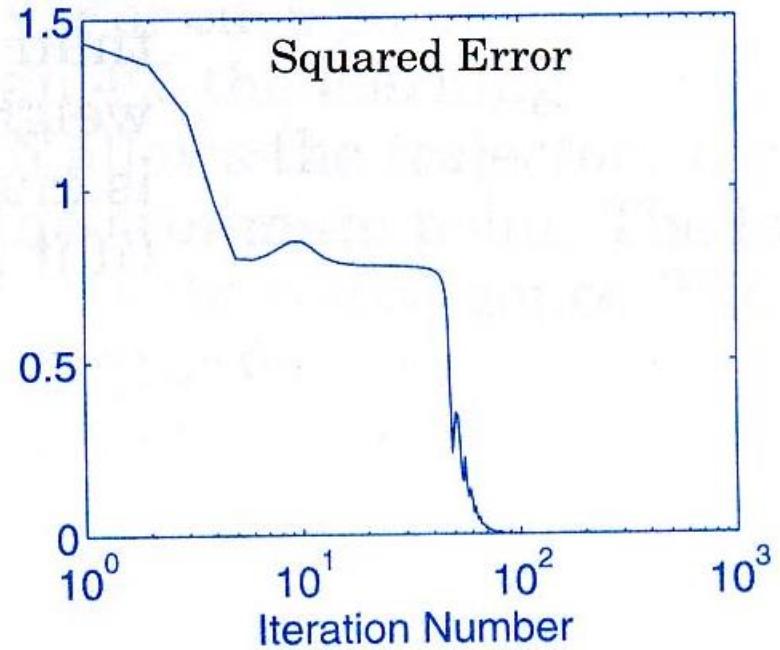
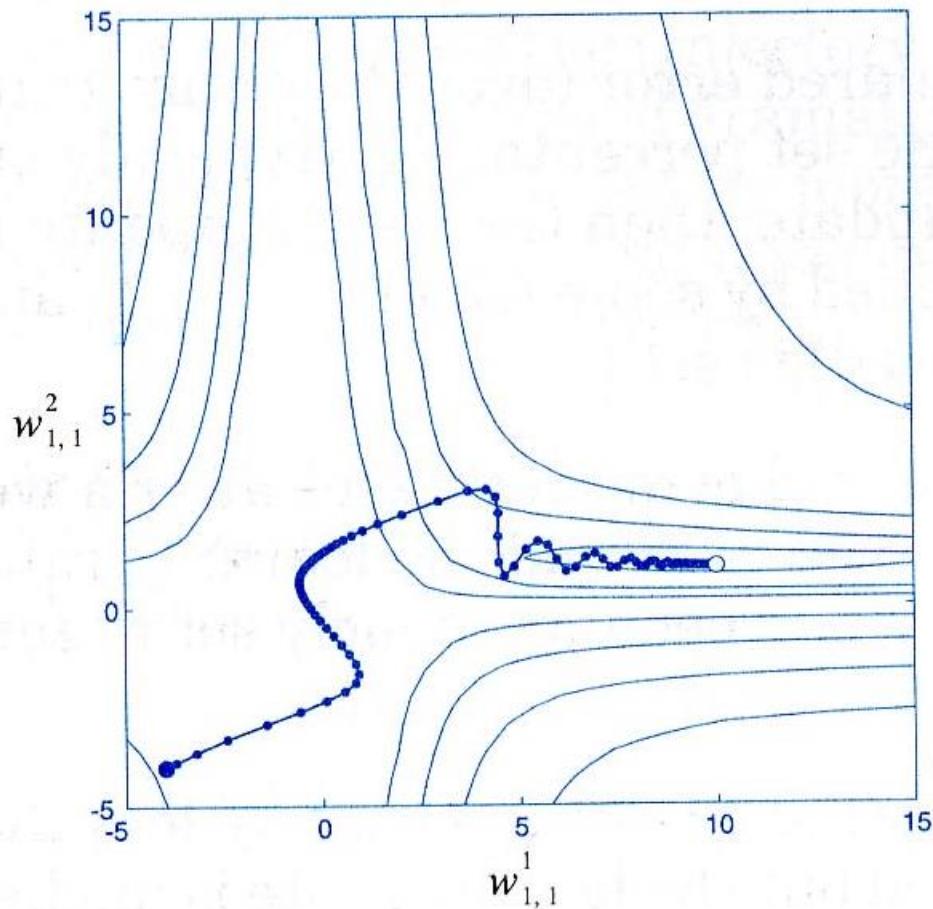


Momentum Backpropagation

Using standard BP



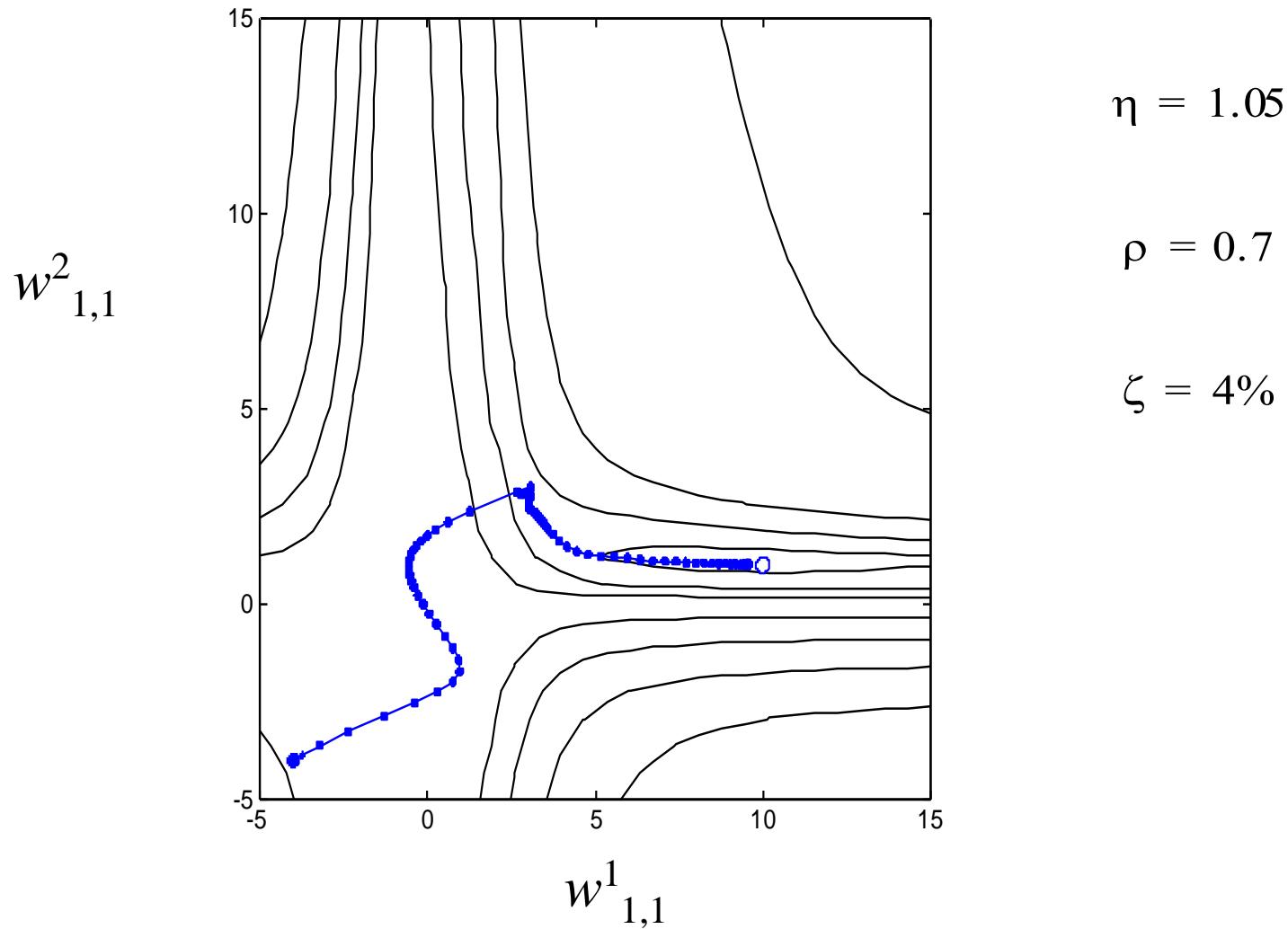
Momentum Backpropagation



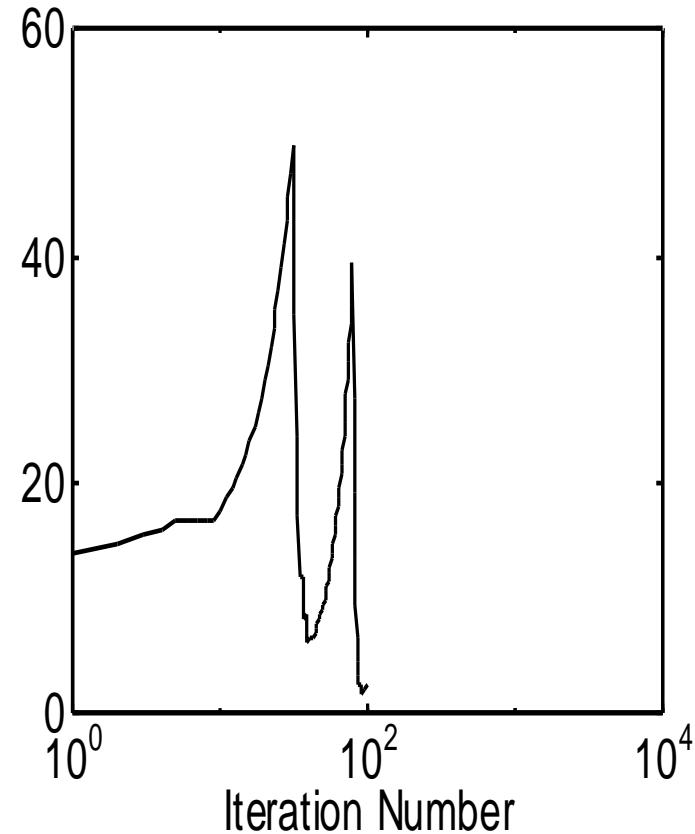
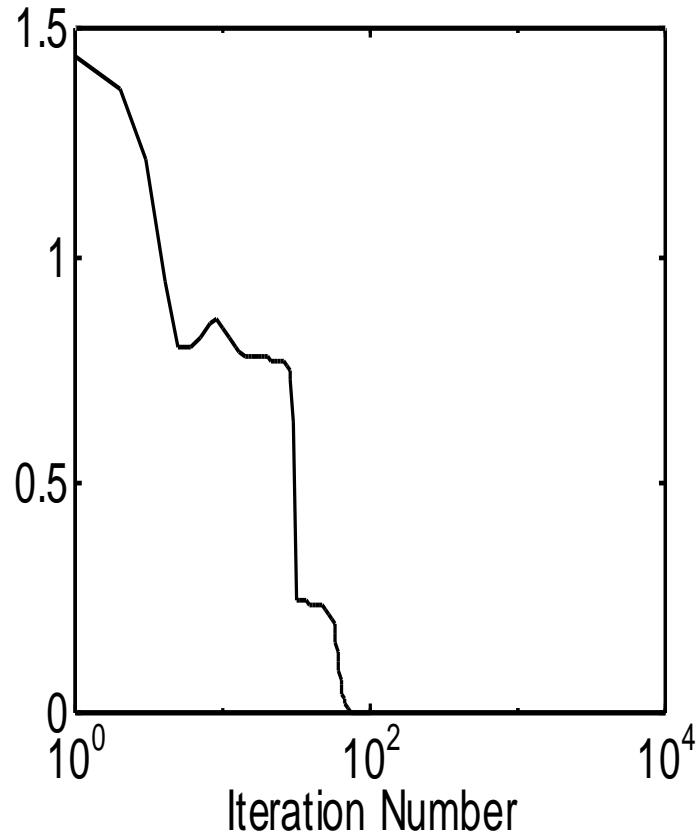
Variable Learning Rate (VLBP)

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

VLBP Example



VLBP Example



Newton's Method

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

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

$$\mathbf{g}_k + \mathbf{A}_k \Delta\mathbf{x}_k = \mathbf{0}$$

$$\Delta\mathbf{x}_k = -\mathbf{A}_k^{-1} \mathbf{g}_k$$

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

Example

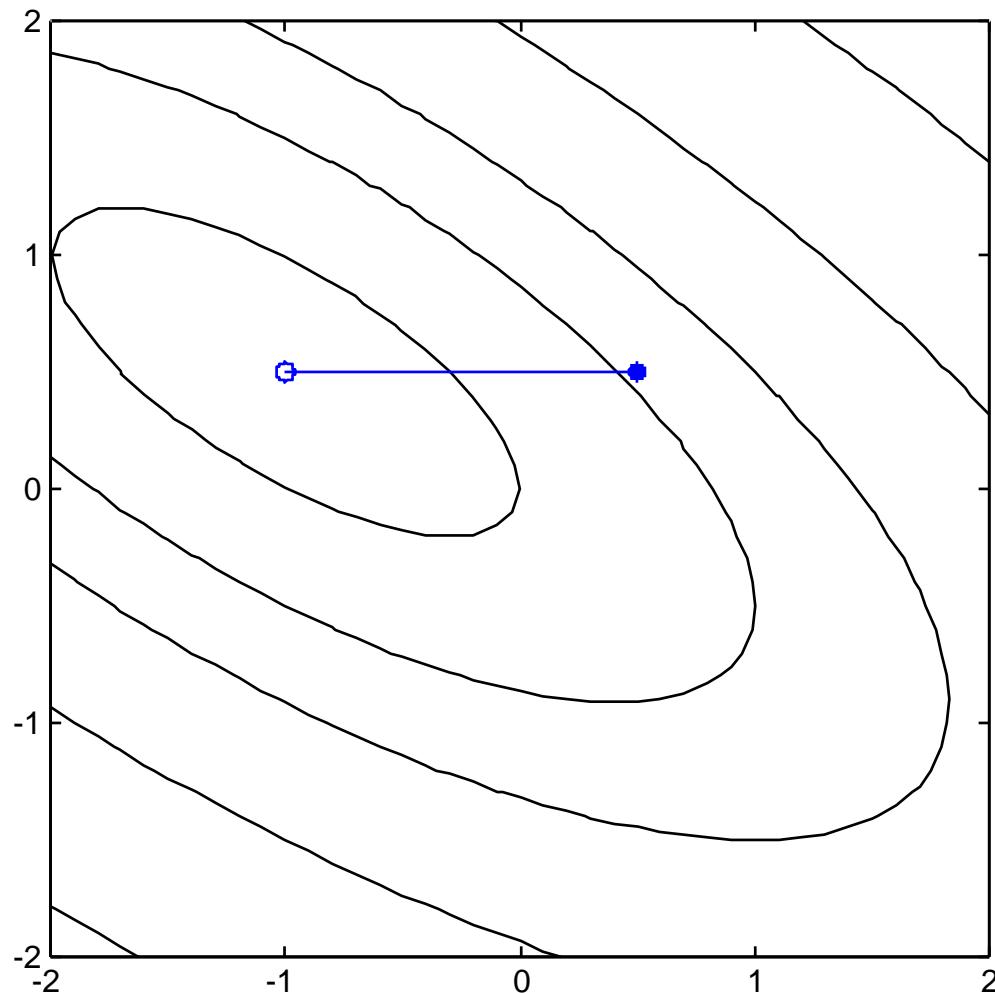
$$F(\mathbf{x}) = x_1^2 + 2x_1x_2 + 2x_2^2 + x_1$$

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

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

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

Plot



BP Based on 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:

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

Hessian

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

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

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

Gauss-Newton Method

Approximate the Hessian matrix as:

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

Newton's method becomes:

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

$$= \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k)]^{-1} \mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k)$$

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}

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

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \mu_k \mathbf{I}]^{-1} \mathbf{J}^T(\mathbf{x}_k) \mathbf{v}(\mathbf{x}_k)$$

Adjustment of μ_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 μ_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 μ_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:

$$F(\mathbf{x}) = \sum_{q=1}^Q (\mathbf{t}_q - \mathbf{a}_q)^T (\mathbf{t}_q - \mathbf{a}_q) = \sum_{q=1}^Q \mathbf{e}_q^T \mathbf{e}_q = \sum_{q=1}^Q \sum_{j=1}^{S^M} (e_{j,q})^2 = \sum_{i=1}^N (v_i)^2$$

The error vector is:

$$\mathbf{v}^T = [v_1 \ v_2 \ \dots \ v_N] = [e_{1,1} \ e_{2,1} \ \dots \ e_{S^M,1} \ e_{1,2} \ \dots \ e_{S^M,Q}]$$

The parameter vector is:

$$\mathbf{x}^T = [x_1 \ x_2 \ \dots \ x_n] = [w_{1,1}^1 \ w_{1,2}^1 \ \dots \ w_{S^1,R}^1 \ b_1^1 \ \dots \ b_{S^1}^1 \ w_{1,1}^2 \ \dots \ b_{S^M}^M]$$

The dimensions of the two vectors are:

$$N = Q \times S^M \quad n = S^1(R+1) + S^2(S^1+1) + \dots + S^M(S^{M-1}+1)$$

Jacobian Matrix

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial e_{1,1}}{\partial w_{1,1}^1} & \frac{\partial e_{1,1}}{\partial w_{1,2}^1} & \cdots & \frac{\partial e_{1,1}}{\partial w_{S^1,R}^1} & \frac{\partial e_{1,1}}{\partial b_1^1} & \cdots \\ \vdots & \vdots & & \vdots & \vdots & \\ \frac{\partial e_{S^M,1}}{\partial w_{1,1}^1} & \frac{\partial e_{S^M,1}}{\partial w_{1,2}^1} & \cdots & \frac{\partial e_{e_{S^M,1}}}{\partial w_{S^1,R}^1} & \frac{\partial e_{e_{S^M,1}}}{\partial b_1^1} & \cdots \\ \frac{\partial e_{1,2}}{\partial w_{1,1}^1} & \frac{\partial e_{1,2}}{\partial w_{1,2}^1} & \cdots & \frac{\partial e_{1,2}}{\partial w_{S^1,R}^1} & \frac{\partial e_{1,2}}{\partial b_1^1} & \cdots \\ \vdots & \vdots & & \vdots & \vdots & \end{bmatrix}$$

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:

Marquardt Sensitivity

If we define a Marquardt sensitivity:

$$\tilde{s}_{i,h}^m \equiv \frac{\partial v_h}{\partial n_{i,q}^m} = \frac{\partial e_{k,q}}{\partial n_{i,q}^m} \quad h = (q-1)S^M + k$$

We can compute the Jacobian as follows:

weight

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial w_{i,j}^m} = \frac{\partial e_{k,q}}{\partial n_{i,q}^m} \times \frac{\partial n_{i,q}^m}{\partial w_{i,j}^m} = \tilde{s}_{i,h}^m \times \frac{\partial n_{i,q}^m}{\partial w_{i,j}^m} = \tilde{s}_{i,h}^m \times a_{j,q}^{m-1}$$

bias

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial b_i^m} = \frac{\partial e_{k,q}}{\partial n_{i,q}^m} \times \frac{\partial n_{i,q}^m}{\partial b_i^m} = \tilde{s}_{i,h}^m \times \frac{\partial n_{i,q}^m}{\partial b_i^m} = \tilde{s}_{i,h}^m$$

Computing the Sensitivities

Initialization

$$\tilde{s}_{i,h}^M = \frac{\partial v_h}{\partial n_{i,q}^M} = \frac{\partial e_{k,q}}{\partial n_{i,q}^M} = \frac{\partial(t_{k,q} - a_{k,q}^M)}{\partial n_{i,q}^M} = -\frac{\partial a_{k,q}^M}{\partial n_{i,q}^M}$$

$$\tilde{s}_{i,h}^M = \begin{cases} -\dot{f}^M(n_{i,q}^M) & \text{for } i = k \\ 0 & \text{for } i \neq k \end{cases}$$

$$\tilde{\mathbf{S}}_q^M = -\dot{\mathbf{F}}^M(\mathbf{n}_q^M)$$

Backpropagation

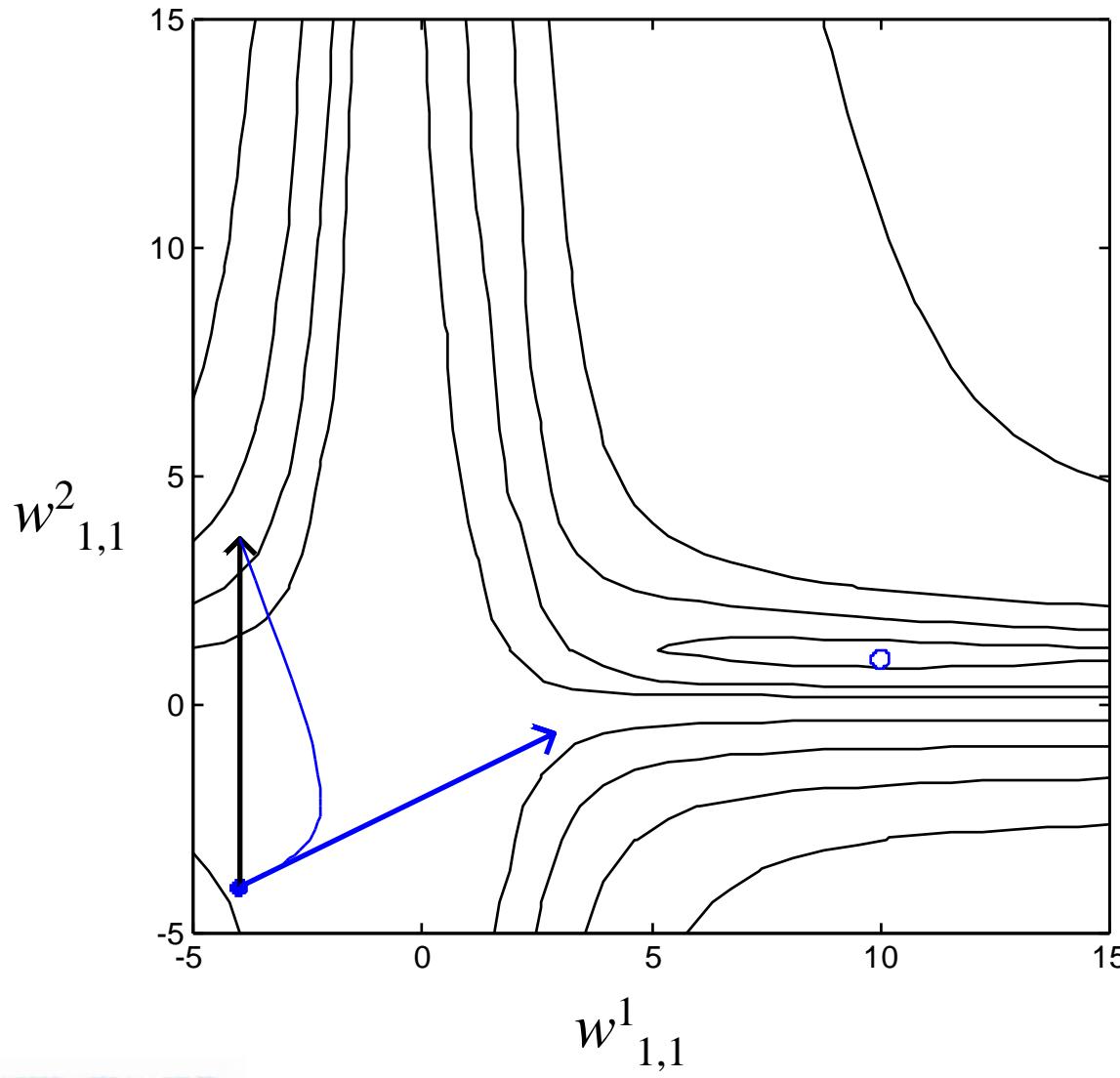
$$\tilde{\mathbf{S}}_q^m = \dot{\mathbf{F}}^m(\mathbf{n}_q^m)(\mathbf{W}^{m+1})^T \tilde{\mathbf{S}}_q^{m+1}$$

$$\tilde{\mathbf{S}}^m = [\tilde{\mathbf{S}}_1^m | \tilde{\mathbf{S}}_2^m | \dots | \tilde{\mathbf{S}}_Q^m]$$

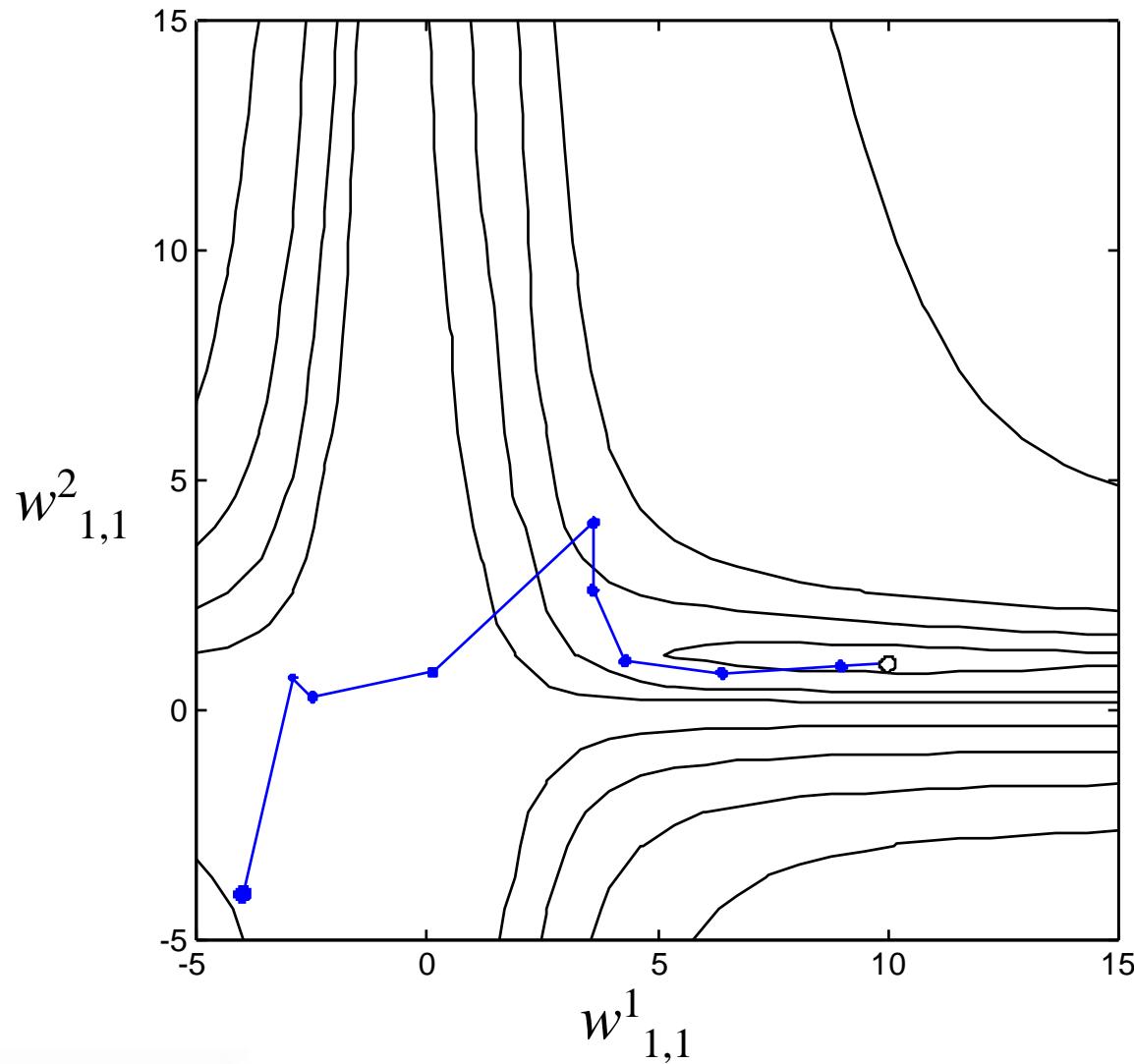
LMBP

- Present all inputs to the network and compute the corresponding network outputs and the errors. Compute the sum of squared errors over all inputs.
- 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.
- Solve to obtain the change in the weights.
- Re-compute 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 μ_k by v , update the weights and go back to step 1. If the sum of squares is not reduced, then multiply μ_k by v and go back to step 3.

Example LMBP Step



LMBP Trajectory



LMBP



$$((p_1 = [1]), (t_1 = [1])) , ((p_2 = [2]), (t_2 = [2])) ,$$

and that the parameters are initialized to

$$W^1 = [1], b^1 = [0], W^2 = [2], b^2 = [1].$$

Find the Jacobian matrix for the first step of the Levenberg-Marquardt method.

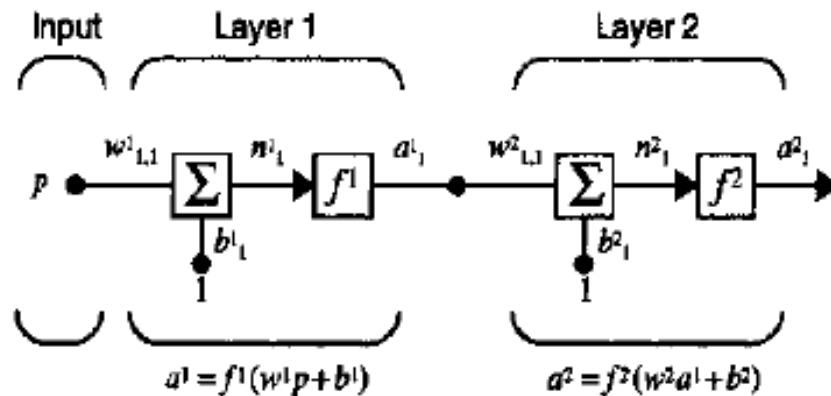


Figure P12.5 Two-Layer Network for LMBP Demonstration

The first step is to propagate the inputs through the network and compute the errors.

LMBP

$$\mathbf{a}_1^0 = \mathbf{p}_1 = [1]$$

$$\mathbf{n}_1^1 = \mathbf{W}^1 \mathbf{a}_1^0 + \mathbf{b}^1 = [1][1] + [0] = [1], \mathbf{a}_1^1 = \mathbf{f}^1(\mathbf{n}_1^1) = ([1])^2 = [1]$$

$$\mathbf{n}_1^2 = \mathbf{W}^2 \mathbf{a}_1^1 + \mathbf{b}^2 = ([2][1] + [1]) = [3], \mathbf{a}_1^2 = \mathbf{f}^2(\mathbf{n}_1^2) = ([3]) = [3]$$

$$\mathbf{e}_1 = (\mathbf{t}_1 - \mathbf{a}_1^2) = ([1] - [3]) = [-2]$$

$$\mathbf{a}_2^0 = \mathbf{p}_2 = [2]$$

$$\mathbf{n}_2^1 = \mathbf{W}^1 \mathbf{a}_2^0 + \mathbf{b}^1 = [1][2] + [0] = [2], \mathbf{a}_2^1 = \mathbf{f}^1(\mathbf{n}_2^1) = ([2])^2 = [4]$$

$$\mathbf{n}_2^2 = \mathbf{W}^2 \mathbf{a}_2^1 + \mathbf{b}^2 = ([2][4] + [1]) = [9], \mathbf{a}_2^2 = \mathbf{f}^2(\mathbf{n}_2^2) = ([9]) = [9]$$

$$\mathbf{e}_2 = (\mathbf{t}_2 - \mathbf{a}_2^2) = ([2] - [9]) = [-7]$$

12-43

LMBP

The next step is to initialize and backpropagate the Marquardt sensitivities using Eq. (12.46) and Eq. (12.47).

$$\tilde{\mathbf{S}}_1^2 = -\mathbf{F}^2(\mathbf{n}_1^2) = -[1]$$

$$\tilde{\mathbf{S}}_1^1 = \mathbf{F}^1(\mathbf{n}_1^1)(\mathbf{W}^2)^T \tilde{\mathbf{S}}_1^2 = \begin{bmatrix} 2n_{1,1}^1 \end{bmatrix} [2] [-1] = [2(1)] [2] [-1] = [-4]$$

$$\tilde{\mathbf{S}}_2^2 = -\mathbf{F}^2(\mathbf{n}_2^2) = -[1]$$

$$\tilde{\mathbf{S}}_2^1 = \mathbf{F}^1(\mathbf{n}_2^1)(\mathbf{W}^2)^T \tilde{\mathbf{S}}_2^2 = \begin{bmatrix} 2n_{1,2}^2 \end{bmatrix} [2] [-1] = [2(2)] [2] [-1] = [-8]$$

$$\tilde{\mathbf{S}}^1 = [\tilde{\mathbf{S}}_1^1 | \tilde{\mathbf{S}}_2^1] = [-4 \ -8], \quad \tilde{\mathbf{S}}^2 = [\tilde{\mathbf{S}}_1^2 | \tilde{\mathbf{S}}_2^2] = [-1 \ -1]$$

We can now compute the Jacobian matrix using Eq. (12.43), Eq. (12.44) and Eq. (12.37).

LMBP

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial v_1}{\partial x_1} & \frac{\partial v_1}{\partial x_2} & \frac{\partial v_1}{\partial x_3} & \frac{\partial v_1}{\partial x_4} \\ \frac{\partial v_2}{\partial x_1} & \frac{\partial v_2}{\partial x_2} & \frac{\partial v_2}{\partial x_3} & \frac{\partial v_2}{\partial x_4} \end{bmatrix} = \begin{bmatrix} \frac{\partial e_{1,1}}{\partial w_{1,1}^1} & \frac{\partial e_{1,1}}{\partial b_1^1} & \frac{\partial e_{1,1}}{\partial w_{1,1}^2} & \frac{\partial e_{1,1}}{\partial b_1^2} \\ \frac{\partial e_{1,2}}{\partial w_{1,1}^1} & \frac{\partial e_{1,2}}{\partial b_1^1} & \frac{\partial e_{1,2}}{\partial w_{1,1}^2} & \frac{\partial e_{1,2}}{\partial b_1^2} \end{bmatrix}$$

$$[\mathbf{J}]_{1,1} = \frac{\partial v_1}{\partial x_1} = \frac{\partial e_{1,1}}{\partial w_{1,1}^1} = \frac{\partial e_{1,1}}{\partial n_{1,1}^1} \times \frac{\partial n_{1,1}^1}{\partial w_{1,1}^1} = \tilde{s}_{1,1}^1 \times \frac{\partial n_{1,1}^1}{\partial w_{1,1}^1} = \tilde{s}_{1,1}^1 \times a_{1,1}^0 \\ = (-4)(1) = -4$$

$$[\mathbf{J}]_{1,2} = \frac{\partial v_1}{\partial x_2} = \frac{\partial e_{1,1}}{\partial b_1^1} = \frac{\partial e_{1,1}}{\partial n_{1,1}^1} \times \frac{\partial n_{1,1}^1}{\partial b_1^1} = \tilde{s}_{1,1}^1 \times \frac{\partial n_{1,1}^1}{\partial b_1^1} = \tilde{s}_{1,1}^1 = -4$$

$$[\mathbf{J}]_{1,3} = \frac{\partial v_1}{\partial x_3} = \frac{\partial e_{1,1}}{\partial n_{1,1}^2} \times \frac{\partial n_{1,1}^2}{\partial w_{1,1}^2} = \tilde{s}_{1,1}^2 \times \frac{\partial n_{1,1}^2}{\partial w_{1,1}^2} = \tilde{s}_{1,1}^2 \times a_{1,1}^1 = (-1)(1) = -1$$

$$[\mathbf{J}]_{1,4} = \frac{\partial v_1}{\partial x_4} = \frac{\partial e_{1,1}}{\partial n_{1,1}^2} \times \frac{\partial n_{1,1}^2}{\partial b_1^2} = \tilde{s}_{1,1}^2 \times \frac{\partial n_{1,1}^2}{\partial b_1^2} = \tilde{s}_{1,1}^2 = -1$$

LMBP

$$[\mathbf{J}]_{2,1} = \frac{\partial v_2}{\partial x_1} = \frac{\partial e_{1,2}}{\partial n_{1,2}^1} \times \frac{\partial n_{1,2}^1}{\partial w_{1,1}^1} = \tilde{s}_{1,2}^1 \times \frac{\partial n_{1,2}^1}{\partial w_{1,1}^1} = \tilde{s}_{1,2}^1 \times a_{1,2}^0 = (-8)(2) = -16$$

$$[\mathbf{J}]_{2,2} = \frac{\partial v_2}{\partial x_2} = \frac{\partial e_{1,2}}{\partial b_1^1} = \frac{\partial e_{1,2}}{\partial n_{1,2}^1} \times \frac{\partial n_{1,2}^1}{\partial b_1^1} = \tilde{s}_{1,2}^1 \times \frac{\partial n_{1,2}^1}{\partial b_1^1} = \tilde{s}_{1,2}^1 = -8$$

$$[\mathbf{J}]_{2,3} = \frac{\partial v_2}{\partial x_3} = \frac{\partial e_{1,2}}{\partial n_{1,2}^2} \times \frac{\partial n_{1,2}^2}{\partial w_{1,1}^2} = \tilde{s}_{1,2}^2 \times \frac{\partial n_{1,2}^2}{\partial w_{1,1}^2} = \tilde{s}_{1,2}^2 \times a_{1,2}^1 = (-1)(4) = -4$$

$$[\mathbf{J}]_{2,4} = \frac{\partial v_2}{\partial x_4} = \frac{\partial e_{1,2}}{\partial b_1^2} = \frac{\partial e_{1,2}}{\partial n_{1,2}^2} \times \frac{\partial n_{1,2}^2}{\partial b_1^2} = \tilde{s}_{1,2}^2 \times \frac{\partial n_{1,2}^2}{\partial b_1^2} = \tilde{s}_{1,2}^2 = -1$$

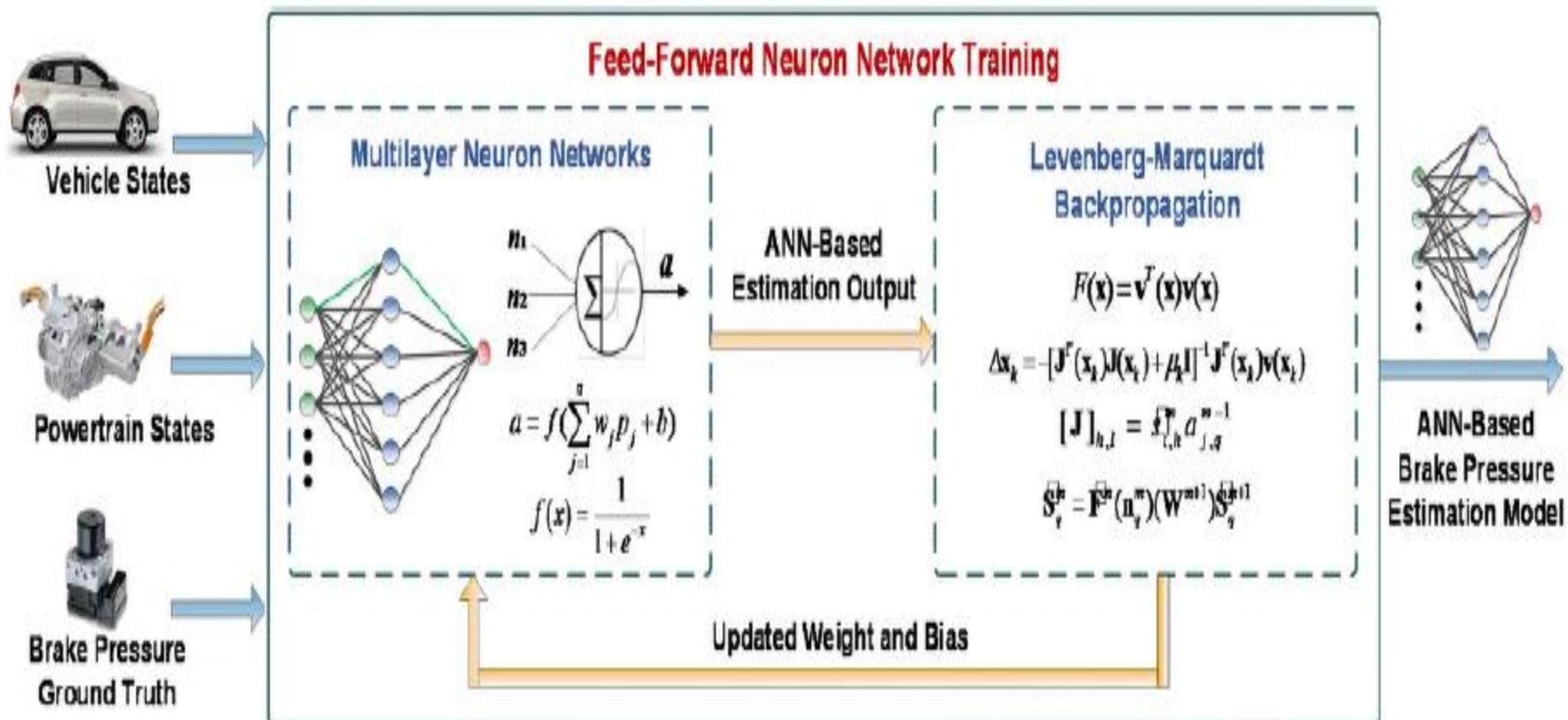
Therefore the Jacobian matrix is

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} -4 & -4 & -1 & -1 \\ -16 & -8 & -4 & -1 \end{bmatrix}$$

Back propagation ***BP*** vs Levenberg Marquardt ***LM***

- Those are two completely unrelated concepts.
- Levenberg-Marquardt (LM) is an optimization method, while backprop is just the recursive application of the chain rule for derivatives.
- What LM intuitively does is this: when it is far from a local minimum, it ignores the curvature of the loss and acts as gradient descent. However, as it gets closer to a local minimum it pays more and more attention to the curvature by switching from gradient descent to a Gauss-Newton like approach.
- The LM method needs both the gradient and the Hessian.
- For neural networks LM usually isn't really useful as you can't construct such a huge Hessian, and even if you do, it lacks the sparse structure needed to invert it efficiently.

Levenberg Marquardt



The Levenberg-Marquardt (LM) Algorithm

[26, 27]. The basic principle of the LM is that it alters its state to the steepest descent until the local curvature is appropriate to make a quadratic estimation with GN algorithm to speed up the convergence [28]. LM uses Hessian matrix for approximation of error surface. The error function is given as:

$$E(t) = \frac{1}{2} \sum_{i=1}^N e_i^2(t) \quad (1)$$

Here, $e(t)$ is the error and the number of vector elements are denoted with N then the gradient descent is calculated in Eq. (2):

$$\nabla E(t) = J^T(t)e(t) \quad (2)$$

$$\nabla^2 E(t) = J^T(t)J(t) \quad (3)$$

Meanwhile, the gradient descent is denoted with $\nabla E(t)$, Hessian Matrix with $\nabla^2 E(t)$ and Jacobain Matrix with $J(t)$. The Jacobain Matrix is calculated in Eq. (4), GN method in Eq. (5) and LM in Eq. (6):

$$J(t) = \begin{bmatrix} \frac{\partial v_1(t)}{\partial t_1} & \frac{\partial v_1(t)}{\partial t_2} & \cdots & \frac{\partial v_1(t)}{\partial t_n} \\ \frac{\partial v_2(t)}{\partial t_1} & \frac{\partial v_2(t)}{\partial t_2} & \cdots & \frac{\partial v_2(t)}{\partial t_n} \\ \vdots & & & \\ \frac{\partial v_n(t)}{\partial t_1} & \frac{\partial v_n(t)}{\partial t_2} & \cdots & \frac{\partial v_n(t)}{\partial t_n} \end{bmatrix} \quad (4)$$

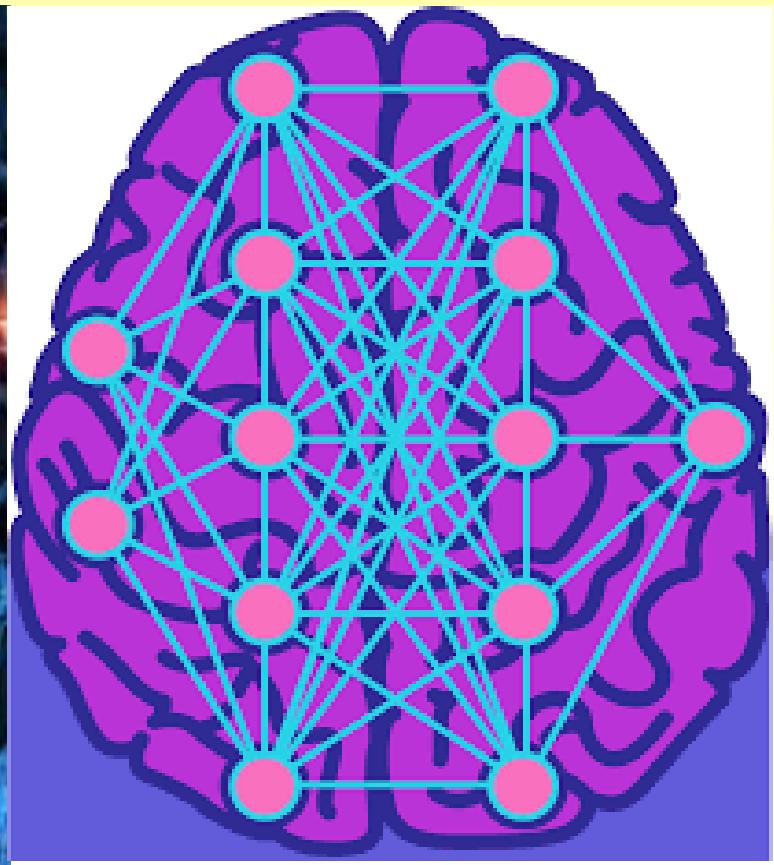
$$\nabla w = -[J^T(t)J(t)]^{-1} J(t)e(t) \quad (5)$$

$$w(k+1) = w(k) - [J^T(t)J(t) + \mu I]^{-1} J(t)e(t) \quad (6)$$

Here, μ is a constant and I is identity matrix. The algorithm will approach the GN, which ought to deliver rapid convergence to global minima. In addition, it should be noted that when the parameter λ is large, Eq. (6) uses gradient descent otherwise for a small λ , the algorithm acts like GN method.

ITF309

Artificial Neural Networks



CHAPTER 16

Self Organized Map (SOM) Neural Network

- Like most artificial neural networks, **SOMs** operate in two modes: **training and mapping**. "**Training**" builds the map using input examples (a competitive process, also called vector quantization), while "**mapping**" automatically classifies a new input vector.

Self-Organising Map (SOM)

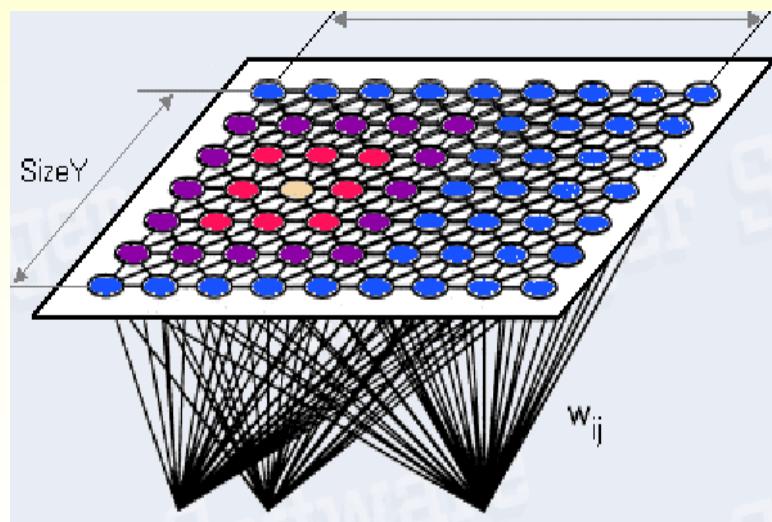
- The Self-Organising Map (SOM) is an unsupervised machine learning algorithm introduced by Teuvo Kohonen in the 1980s [1]. As the name suggests, the map organises itself without any instruction from others. It is a brain-inspired model. A different area of the cerebral cortex in our brain is responsible for specific activities. A sensory input like vision, hearing, smell, and taste is mapped to neurons of a corresponding cortex area via synapses in a self-organising way. It is also known that the neurons with similar output are in proximity. SOM is trained through a competitive neural network, a single-layer feed-forward network that resembles these brain mechanisms.

- A **self-organizing map** consists of components called nodes or neurons. Associated with each node are a weight vector of the same dimension as the input data vectors, and a position in the map space.
- The usual arrangement of nodes is a two-dimensional regular spacing in a hexagonal or rectangular grid. The self-organizing map describes a mapping from a higher-dimensional input space to a lower-dimensional map space.
- The procedure for placing a vector from data space onto the map is to find the node with the closest (smallest distance metric) weight vector to the data space vector.

- The training utilizes **competitive learning**. When a training example is fed to the network, its **Euclidean distance** to all weight vectors is computed. The neuron whose weight vector is most similar to the input is called the **best matching unit** (BMU).
- The weights of the BMU and neurons close to it in the SOM lattice are adjusted towards the input vector.
- The magnitude of the change decreases with time and with distance (within the lattice) from the BMU.

- The Self-Organizing Map is one of the most popular neural network models. It belongs to the category of **competitive learning networks**. The Self-Organizing Map is based on unsupervised learning, which means that no human intervention is needed during the learning and that little needs to be known about the characteristics of the input data.
- We could, for example, use the **SOM** for clustering data without knowing the class memberships of the input data. The SOM can be used to detect features inherent to the problem and thus has also been called **SOFM**, the **Self-Organized Feature Map**.

- **The Self-Organizing Map is a two-dimensional array of neurons:**
- This has the same dimension as the input vectors (n -dimensional). The neurons are connected to adjacent neurons by a neighborhood relation. This dictates the topology, or the structure, of the map. Usually, the neurons are connected to each other via rectangular or hexagonal topology.



Self Organized Map (SOM)

- The **self-organizing map (SOM)** is a method for unsupervised learning, based on a grid of artificial neurons whose weights are adapted to match input vectors in a training set.
- It was first described by the Finnish professor **Teuvo Kohonen** and is thus sometimes referred to as a **Kohonen map**.

Teuvo Kohonen



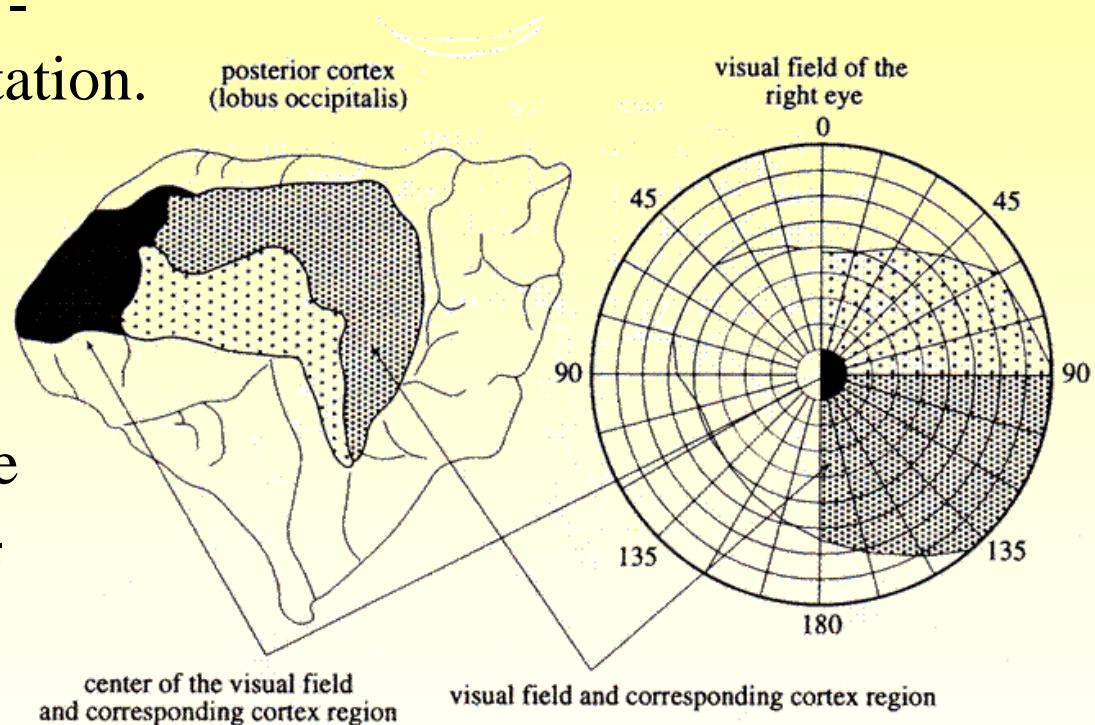
Dr. Eng., Professor of the Academy of Finland;
Head, Neural Networks Research Centre,
Helsinki University of Technology, Finland

Brain's self-organization

The brain maps the external multidimensional representation of the world into a similar 1 or 2 - dimensional internal representation.

That is, the brain processes the external signals in a topology-preserving way

Mimicking the way the brain learns, our system should be able to do the same thing.



Why SOM ?

- **Unsupervised Learning**
- **Clustering**

Self Organizing Networks

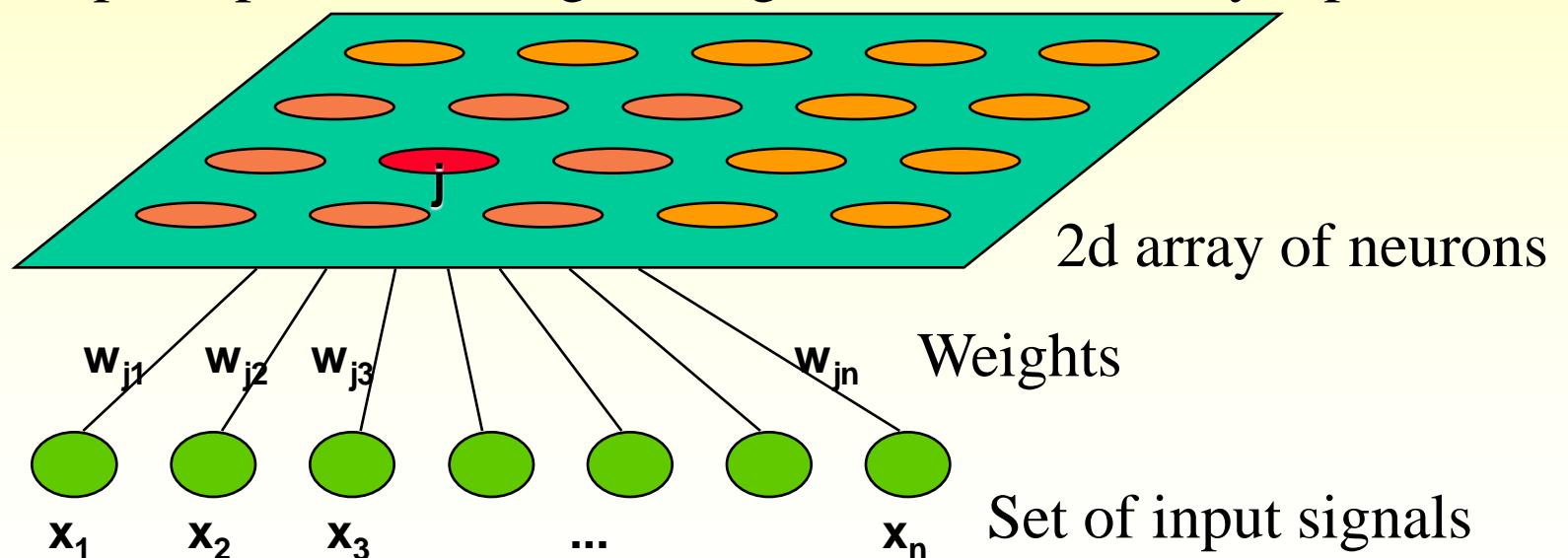
- Discover **significant patterns or features** in the input data
- Discovery is done **without a teacher**
- Synaptic weights are changed according to **local rules**
- The changes affect a neuron's immediate environment until **a final configuration** develops

Network Architecture

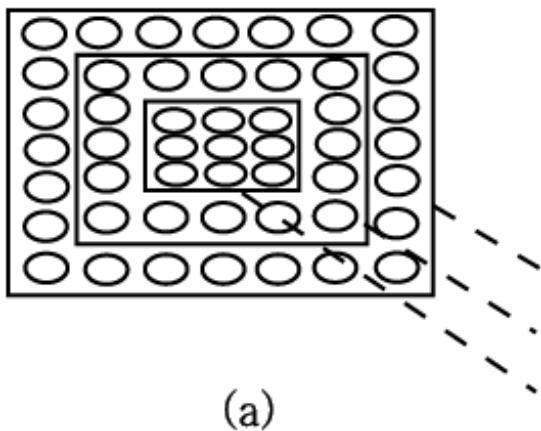
- Two layers of units
 - Input: n units (length of training vectors)
 - Output: m units (number of categories)
- Input units fully connected with weights to output units

SOM - Architecture

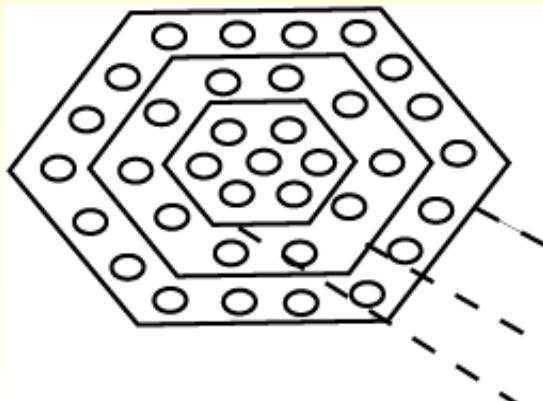
- Lattice of neurons ('nodes') accepts and responds to set of input signals
- Responses compared; 'winning' neuron selected from lattice
- Selected neuron activated ***together with 'neighbourhood' neurons***
- Adaptive process changes weights to more closely inputs



Measuring distances between nodes



(a)



(b)

- Distances between output neurons will be used in the learning process.
- It may be based upon:
 - a) Rectangular lattice
 - b) Hexagonal lattice

- Let $d(i,j)$ be the distance between the output nodes i,j
- $d(i,j) = 1$ if node j is in the first outer rectangle/hexagon of node i
- $d(i,j) = 2$ if node j is in the second outer rectangle/hexagon of node i
- And so on..

- Each neuron is a node containing a template against which input patterns are matched.
- All Nodes are presented with the same input pattern in parallel and compute the distance between their template and the input in parallel.
- Only the node with the closest match between the input and its template produces an **active output**.
- Each Node therefore acts like a separate decoder (or pattern detector, **feature detector**) for the same input and the interpretation of the input derives from the presence or absence of an active response at each location
(rather than the magnitude of response or an input-output transformation as in feedforward or feedback networks).

SOM: interpretation

- Each SOM **neuron** can be seen as representing a **cluster** containing all the input examples which are mapped to that neuron.
- For a given input, the **output** of SOM is the neuron with weight vector most similar (with respect to Euclidean distance) to that input.

Simple Models

- Network has inputs and outputs
- There is **no feedback** from the environment
 → **no supervision**
- The network updates the weights following some learning rule, and finds patterns, features or categories within the inputs presented to the network

More about SOM learning

- Upon repeated presentations of the training examples, the weight vectors of the neurons tend to follow the distribution of the examples.
- This results in a topological ordering of the neurons, where neurons adjacent to each other tend to have similar weight vectors.
- The input space of patterns is mapped onto a discrete output space of neurons.

SOM – Learning Algorithm

1. Randomly initialise all weights
2. Select input vector $\mathbf{x} = [x_1, x_2, x_3, \dots, x_n]$ from training set
3. Compare \mathbf{x} with weights \mathbf{w}_j for each neuron j to

$$d_j = \sum_i (w_{ij} - x_i)^2$$

4. determine winner
find unit j with the minimum distance
5. Update winner so that it becomes more like \mathbf{x} , together with the winner's *neighbours* for units within the radius according to $w_{ij}(n+1) = w_{ij}(n) + \eta(n)[x_i - w_{ij}(n)]$
6. Adjust parameters: learning rate & 'neighbourhood function'
7. Repeat from (2) until ... ?

Note that: Learning rate generally decreases
with time:

$$0 < \eta(n) \leq \eta(n-1) \leq 1$$

SOM Algorithm

- Select output layer network topology
 - Initialize current neighborhood distance, $D(0)$, to a positive value
- Initialize weights from inputs to outputs to small random values
- Let $t = 1$
- While computational bounds are not exceeded do
 - 1) Select an input sample i_l
 - 2) Compute the square of the Euclidean distance of i_l from weight vectors (w_j) associated with each output node
$$\sum_{k=1}^n (i_{l,k} - w_{j,k}(t))^2$$
 - 3) Select output node j^* that has weight vector with minimum value from step 2)
 - 4) Update weights to all nodes within a topological distance given by $D(t)$ from j^* , using the weight update rule:
$$w_j(t+1) = w_j(t) + \eta(t)(i_l - w_j(t))$$

- 5) Increment t
- Endwhile

Learning rate generally decreases with time:

$$0 < \eta(t) \leq \eta(t-1) \leq 15$$

From Mehotra et al. (1997), p. 189

Algorithm

1. Randomize the map's nodes' weight vectors
2. Grab an input vector $D(t)$
3. Traverse each node in the map
 1. Use the Euclidean distance formula to find the similarity between the input vector and the map's node's weight vector
 2. Track the node that produces the smallest distance (this node is the best matching unit, BMU)
4. Update the nodes in the neighborhood of the BMU (including the BMU itself) by pulling them closer to the input vector
 1.
$$W_v(s + 1) = W_v(s) + \Theta(u, v, s) \alpha(s)(D(t) - W_v(s))$$
5. Increase s and repeat from step 2 while $s < \lambda$

Variables

These are the variables needed, with vectors in bold,

- s is the current iteration
- λ is the iteration limit
- t is the index of the target input data vector in the input data set \mathbf{D}
- $\mathbf{D}(t)$ is a target input data vector
- v is the index of the node in the map
- \mathbf{w}_v is the current weight vector of node v
- u is the index of the best matching unit (BMU) in the map
- $\Theta(u, v, s)$ is a restraint due to distance from BMU, usually called the neighborhood function, and
- $\alpha(s)$ is a learning restraint due to iteration progress.

Neighborhood function

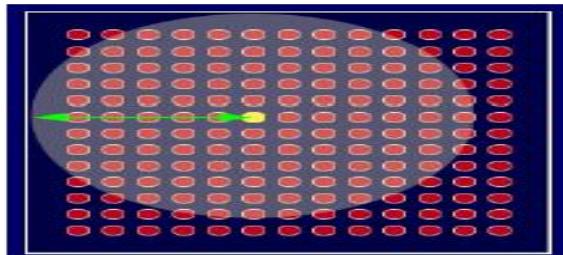
The neighborhood function $\Theta(u, v, s)$ depends on the lattice distance between the BMU (neuron u) and neuron v . In the simplest form it is 1 for all neurons close enough to BMU and 0 for others, but a Gaussian function is a common choice, too. Regardless of the functional form, the neighborhood function shrinks with time.

Neighborhood Function

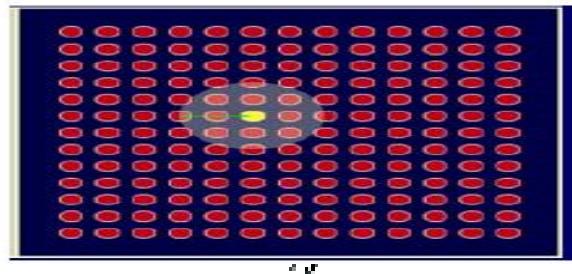
- Gaussian neighborhood function:

$$h_i(d_{ij}) = \exp\left(-\frac{d_{ij}^2}{2\sigma^2}\right)$$

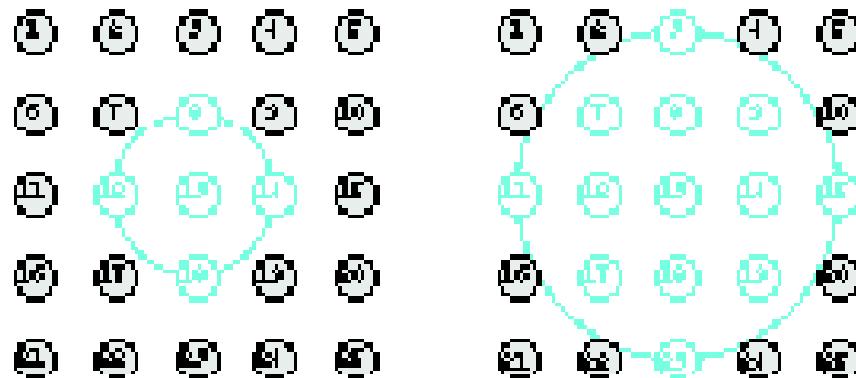
- d_{ji} : lateral distance of neurons i and j
 - in a 1-dimensional lattice $|j - i|$
 - in a 2-dimensional lattice $\|r_j - r_i\|$
where r_j is the position of neuron j in the lattice.



Neighborhood of a
best matching unit
...decreases over time



$N_{13}(1) = \{8, 12, 13, 14, 18\}$ ← Two possible neighborhoods of neuron 13
 $N_{13}(2) = \{3, 7, 8, 9, 11, 12, 13, 14, 15, 17, 18, 19, 23\}$ ←



$N_{13}(1)$

EELU ITF309 Neural Network
Lecture 13

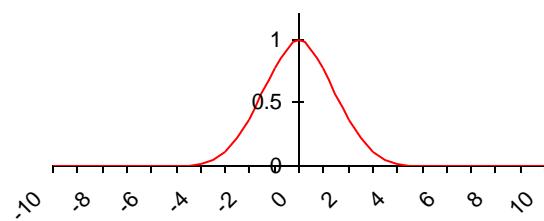
Neighborhood Function

- σ measures the degree to which excited neurons in the vicinity of the winning neuron cooperate in the learning process.
- In the learning algorithm σ is updated at each iteration during the ordering phase using the following exponential decay update rule, with parameters

$$\sigma(n) = \sigma_0 \exp\left(-\frac{n}{T_1}\right)$$

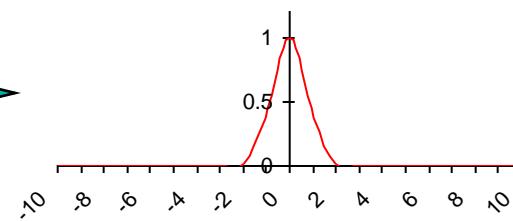
Neighbourhood function

Degree of neighbourhood

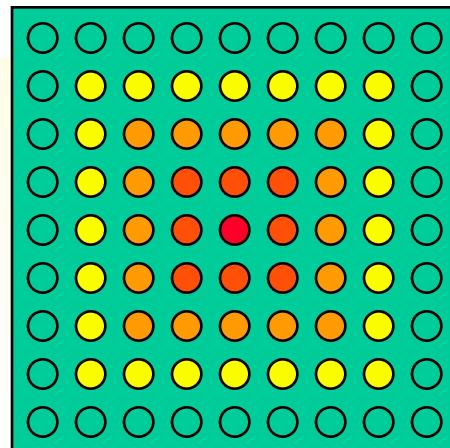


Time →

Degree of neighbourhood

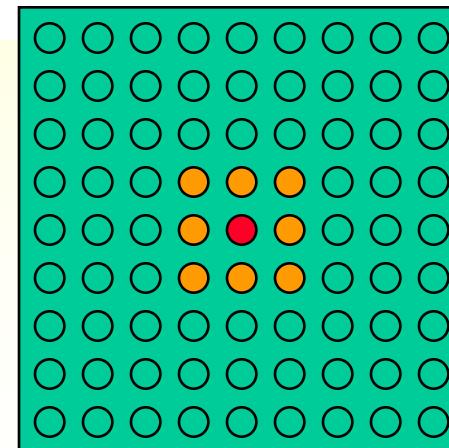


Distance from winner



Time →

Distance from winner



UPDATE RULE

$$w_j(n+1) = w_j(n) + \eta(n) h_{ij(x)}(n) (\mathbf{x} - w_j(n))$$

exponential decay update of the learning rate:

$$\eta(n) = \eta_0 \exp\left(-\frac{n}{T_2}\right)$$

Two-phases learning approach

- Self-organizing or ordering phase. The learning rate and spread of the Gaussian neighborhood function are adapted during the execution of SOM, using for instance the exponential decay update rule.
- Convergence phase. The learning rate and Gaussian spread have small fixed values during the execution of SOM.

Ordering Phase

- Self organizing or ordering phase:
 - Topological ordering of weight vectors.
 - May take 1000 or more iterations of SOM algorithm.
- Important choice of the parameter values. For instance
 - $\eta(n)$: $\eta_0 = 0.1$ $T_2 = 1000$
 \Rightarrow decrease gradually $\eta(n) \geq 0.01$
 - $h_{ji(x)}(n)$: σ_0 big enough $T_1 = \frac{1000}{\log(\sigma_0)}$
- With this parameter setting initially the neighborhood of the winning neuron includes almost all neurons in the network, then it shrinks slowly with time.

Convergence Phase

- Convergence phase:
 - Fine tune the weight vectors.
 - Must be at least 500 times the number of neurons in the network \Rightarrow thousands or tens of thousands of iterations.
- Choice of parameter values:
 - $\eta(n)$ maintained on the order of 0.01.
 - Neighborhood function such that the neighbor of the winning neuron contains only the nearest neighbors. It eventually reduces to one or zero neighboring neurons.

Example

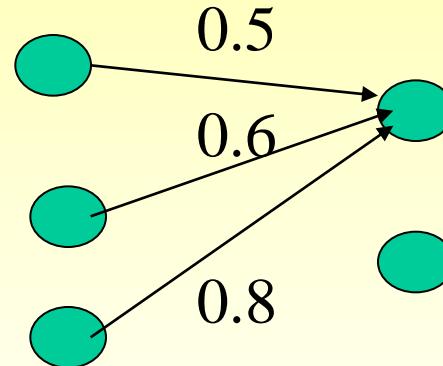
https://www.youtube.com/watch?v=_IRcxgG0FL4

An SOFM network with three inputs and two cluster units is to be trained using the four training vectors:

[0.8 0.7 0.4], [0.6 0.9 0.9], [0.3 0.4 0.1], [0.1 0.1 0.2] and initial weights

$$\begin{bmatrix} 0.5 & 0.4 \\ 0.6 & 0.2 \\ 0.8 & 0.5 \end{bmatrix}$$

weights to the first cluster unit



The initial radius is 0 and the learning rate η is 0.5 . Calculate the weight changes during the first cycle through the data, taking the training vectors in the given order.

Solution

The Euclidian distance of the input vector 1 to cluster unit 1 is:

$$d_1 = (0.5 - 0.8)^2 + (0.6 - 0.7)^2 + (0.8 - 0.4)^2 = 0.26$$

The Euclidian distance of the input vector 1 to cluster unit 2 is:

$$d_2 = (0.4 - 0.8)^2 + (0.2 - 0.7)^2 + (0.5 - 0.4)^2 = 0.42$$

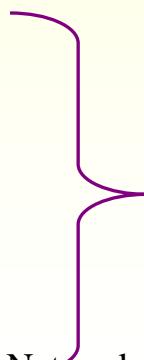
Input vector 1 is closest to cluster unit 1 so update weights to cluster unit 1:

$$w_{ij}(n+1) = w_{ij}(n) + 0.5[x_i - w_{ij}(n)]$$

$$0.65 = 0.5 + 0.5(0.8 - 0.5)$$

$$0.65 = 0.6 + 0.5(0.7 - 0.6)$$

$$0.6 = 0.8 + 0.5(0.4 - 0.8)$$



$$\begin{bmatrix} 0.65 & 0.4 \\ 0.65 & 0.2 \\ 0.60 & 0.5 \end{bmatrix}$$

Solution

The Euclidian distance of the input vector 2 to cluster unit 1 is:

$$d_1 = (0.65 - 0.6)^2 + (0.65 - 0.9)^2 + (0.6 - 0.9)^2 = 0.155$$

The Euclidian distance of the input vector 2 to cluster unit 2 is:

$$d_2 = (0.4 - 0.6)^2 + (0.2 - 0.9)^2 + (0.5 - 0.9)^2 = 0.69$$

Input vector 2 is closest to cluster unit 1 so update weights to cluster unit 1 again:

$$\begin{aligned} w_{ij}(n+1) &= w_{ij}(n) + 0.5[x_i - w_{ij}(n)] \\ 0.625 &= 0.65 + 0.5(0.6 - 0.65) \\ 0.775 &= 0.65 + 0.5(0.9 - 0.65) \\ 0.750 &= 0.60 + 0.5(0.9 - 0.60) \end{aligned} \quad \left. \begin{array}{l} \\ \\ \end{array} \right\} \begin{bmatrix} 0.625 & 0.4 \\ 0.775 & 0.2 \\ 0.750 & 0.5 \end{bmatrix}$$

Repeat the same update procedure for input vector 3 and 4 also.

Another Self-Organizing Map (SOM) Example

- From Fausett (1994)

- $n = 4, m = 2$

- More typical of SOM application

- Smaller number of units in output than in input;
dimensionality reduction

- Training samples

- i1: (1, 1, 0, 0)

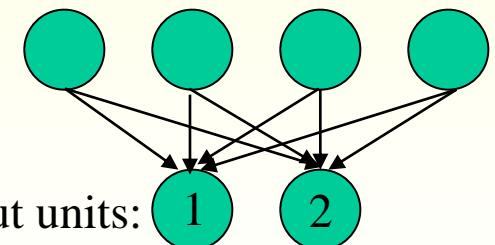
- i2: (0, 0, 0, 1)

- i3: (1, 0, 0, 0)

- i4: (0, 0, 1, 1)

Network Architecture

Input units:



Output units:

What should we expect as outputs?

What are the Euclidean Distances Between the Data Samples?

- Training samples

i1: (1, 1, 0, 0)

i2: (0, 0, 0, 1)

i3: (1, 0, 0, 0)

i4: (0, 0, 1, 1)

	i1	i2	i3	i4
i1	0			
i2		0		
i3			0	
i4				0

Euclidean Distances Between Data Samples

- Training samples

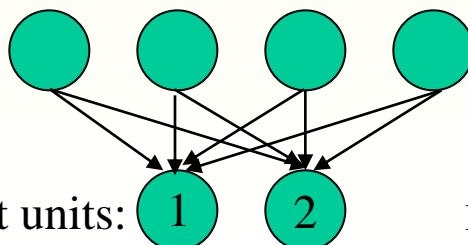
i1: (1, 1, 0, 0)

i2: (0, 0, 0, 1)

i3: (1, 0, 0, 0)

i4: (0, 0, 1, 1)

Input units:



Output units:

	i1	i2	i3	i4
i1	0			
i2	3	0		
i3	1	2	0	
i4	4	1	3	0

EELU What might we expect from the SOM?
Lecture 11

Example Details

- Training samples

$$i1: (1, 1, 0, 0)$$

$$i2: (0, 0, 0, 1)$$

$$i3: (1, 0, 0, 0)$$

$$i4: (0, 0, 1, 1)$$

- With only 2 outputs, neighborhood = 0

– Only update weights associated with winning output unit (cluster) at each iteration

- Learning rate

$$\eta(t) = 0.6; 1 \leq t \leq 4$$

$$\eta(t) = 0.5 \eta(1); 5 \leq t \leq 8$$

$$\eta(t) = 0.5 \eta(5); 9 \leq t \leq 12$$

etc.

- Initial weight matrix

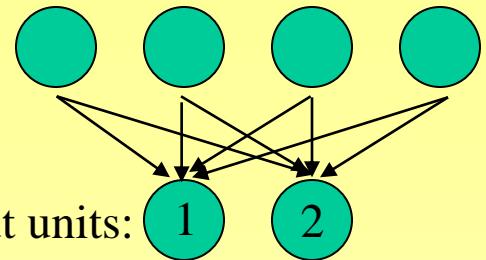
(random values between 0 and 1)

$$d^2 = (\text{Euclidean distance})^2 = \sum_{k=1}^n (i_{l,k} - w_{j,k}(t))^2$$

$$\text{Weight update: } w_j(t+1) = w_j(t) + \eta(t)(i_l - w_j(t))$$

Problem: Calculate the weight updates for the first four steps

Input units:



Output units:

First Weight Update

- Training sample: i1

- Unit 1 weights

- $d^2 = (.2-1)^2 + (.6-1)^2 + (.5-0)^2 + (.9-0)^2 = 1.86$

- Unit 2 weights

- $d^2 = (.8-1)^2 + (.4-1)^2 + (.7-0)^2 + (.3-0)^2 = .98$

- Unit 2 wins

- Weights on winning unit are updated

$$\text{new-unit2-weights} = [.8 \quad .4 \quad .7 \quad .3] + 0.6([1 \ 1 \ 0 \ 0] - [.8 \quad .4 \quad .7 \quad .3]) = [.92 \quad .76 \quad .28 \quad .12]$$

- Giving an updated weight matrix:

$$\text{Unit 1: } [.2 \quad .6 \quad .5 \quad .9]$$

$$\text{Unit 2: } [.92 \quad .76 \quad .28 \quad .12]$$

i1: (1, 1, 0, 0)

i2: (0, 0, 0, 1)

i3: (1, 0, 0, 0)

i4: (0, 0, 1, 1)

Second Weight Update

i1: (1, 1, 0, 0)
i2: (0, 0, 0, 1)
i3: (1, 0, 0, 0)
i4: (0, 0, 1, 1)

- Training sample: i2

- Unit 1 weights

- $d^2 = (.2-0)^2 + (.6-0)^2 + (.5-0)^2 + (.9-1)^2 = .66$

- Unit 2 weights

- $d^2 = (.92-0)^2 + (.76-0)^2 + (.28-0)^2 + (.12-1)^2 = 2.28$

- Unit 1 wins

- Weights on winning unit are updated

$$\text{new-unit1-weights} = [.2 \quad .6 \quad .5 \quad .9] + 0.6([0 \quad 0 \quad 0 \quad 1] - [.2 \quad .6 \quad .5 \quad .9]) = [.08 \quad .24 \quad .20 \quad .96]$$

- Giving an updated weight matrix:

$$\begin{aligned}\text{Unit 1: } & [.08 \quad .24 \quad .20 \quad .96] \\ \text{Unit 2: } & [.92 \quad .76 \quad .28 \quad .12]\end{aligned}$$

Third Weight Update

- Training sample: i3

- Unit 1 weights

- $d^2 = (.08-1)^2 + (.24-0)^2 + (.2-0)^2 + (.96-0)^2 = 1.87$

- Unit 2 weights

- $d^2 = (.92-1)^2 + (.76-0)^2 + (.28-0)^2 + (.12-0)^2 = 0.68$

- Unit 2 wins

- Weights on winning unit are updated

$$\text{new-unit2-weights} = [.92 \quad .76 \quad .28 \quad .12] + 0.6([1 \quad 0 \quad 0 \quad 0] - [.92 \quad .76 \quad .28 \quad .12]) = \\ [.97 \quad .30 \quad .11 \quad .05]$$

- Giving an updated weight matrix:

- Unit 1: $[\begin{array}{cccc} .08 & .24 & .20 & .96 \end{array}]$

- Unit 2: $[\begin{array}{cccc} .97 & .30 & .11 & .05 \end{array}]$

i1: (1, 1, 0, 0)

i2: (0, 0, 0, 1)

i3: (1, 0, 0, 0)

i4: (0, 0, 1, 1)

Fourth Weight Update

i1: (1, 1, 0, 0)
i2: (0, 0, 0, 1)
i3: (1, 0, 0, 0)
i4: (0, 0, 1, 1)

- Training sample: i4
 - Unit 1 weights
 - $d^2 = (.08-0)^2 + (.24-0)^2 + (.2-1)^2 + (.96-1)^2 = .71$
 - Unit 2 weights
 - $d^2 = (.97-0)^2 + (.30-0)^2 + (.11-1)^2 + (.05-1)^2 = 2.74$
 - Unit 1 wins
 - Weights on winning unit are updated

$$\text{new-unit1-weights} = [.08 \quad .24 \quad .20 \quad .96] + 0.6([0 \ 0 \ 1 \ 1] - [.08 \quad .24 \quad .20 \quad .96]) = [.03 \quad .10 \quad .68 \quad .98]$$

- Giving an updated weight matrix:

$$\begin{aligned}\text{Unit 1: } & [.03 \quad .10 \quad .68 \quad .98] \\ \text{Unit 2: } & [.97 \quad .30 \quad .11 \quad .05]\end{aligned}$$

Applying the SOM Algorithm

Data sample utilized

time (t)	1	2	3	4	D(t)	$\eta(t)$
1	Unit 2				0	0.6
2		Unit 1			0	0.6
3			Unit 2		0	0.6
4				Unit 1	0	0.6

‘winning’ output unit

After many iterations (epochs)
through the data set:

$$\begin{array}{ll} \text{Unit 1: } & \begin{bmatrix} 0 & 0 & .5 & 1.0 \end{bmatrix} \\ \text{Unit 2: } & \begin{bmatrix} 1.0 & .5 & 0 & 0 \end{bmatrix} \end{array}$$

Did we get the clustering that we expected?

Training samples

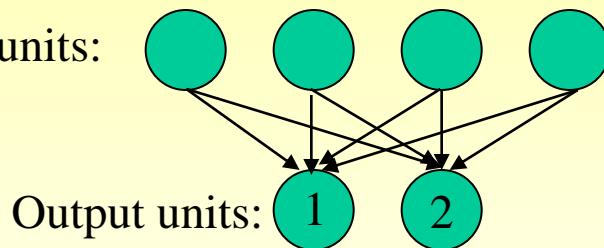
i1: (1, 1, 0, 0)

i2: (0, 0, 0, 1)

i3: (1, 0, 0, 0)

i4: (0, 0, 1, 1)

Input units:



Output units:

Weights

Unit 1:

$$\begin{bmatrix} 0 & 0 & .5 & 1.0 \end{bmatrix}$$

Unit 2:

$$\begin{bmatrix} 1.0 & .5 & 0 & 0 \end{bmatrix}$$

What clusters do the data samples fall into?

Training samples

i1: (1, 1, 0, 0)

i2: (0, 0, 0, 1)

i3: (1, 0, 0, 0)

i4: (0, 0, 1, 1)

Solution

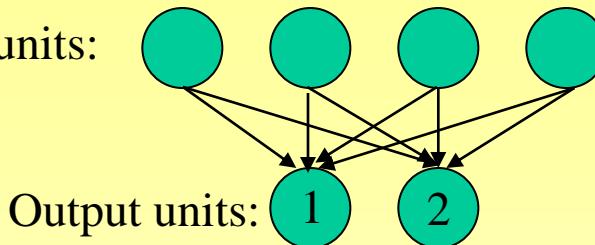
Weights

Unit 1:

$$\begin{bmatrix} 0 & 0 & .5 & 1.0 \\ 1.0 & .5 & 0 & 0 \end{bmatrix}$$

Unit 2:

Input units:



Output units:

- Sample: i1
 - Distance from unit1 weights
 - $(1-0)^2 + (1-0)^2 + (0-.5)^2 + (0-1.0)^2 = 1+1+.25+1=3.25$
 - Distance from unit2 weights
 - $(1-1)^2 + (1-.5)^2 + (0-0)^2 + (0-0)^2 = 0+.25+0+0=.25$ (winner)

- Sample: i2
 - Distance from unit1 weights
 - $(0-0)^2 + (0-0)^2 + (0-.5)^2 + (1-1.0)^2 = 0+0+.25+0$ (winner)
 - Distance from unit2 weights
 - $(0-1)^2 + (0-.5)^2 + (0-0)^2 + (1-0)^2 = 1+.25+0+1=2.25$

$$\sum_{k=1}^n (i_{l,k} - w_{j,k}(t))^2$$

$$d^2 = (\text{Euclidean distance})^2 = \frac{\text{EELU ITF309 Neural Network}}{\text{Lecture 11}}$$

Training samples

i1: (1, 1, 0, 0)

i2: (0, 0, 0, 1)

i3: (1, 0, 0, 0)

i4: (0, 0, 1, 1)

Solution

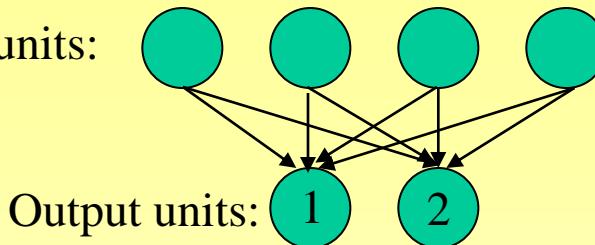
Weights

Unit 1:

$$\begin{bmatrix} 0 & 0 & .5 & 1.0 \\ 1.0 & .5 & 0 & 0 \end{bmatrix}$$

Unit 2:

Input units:



Output units:

- Sample: i3
 - Distance from unit1 weights
 - $(1-0)^2 + (0-0)^2 + (0-.5)^2 + (0-1.0)^2 = 1+0+.25+1=2.25$
 - Distance from unit2 weights
 - $(1-1)^2 + (0-.5)^2 + (0-0)^2 + (0-0)^2 = 0+.25+0+0=.25$ (winner)
- Sample: i4
 - Distance from unit1 weights
 - $(0-0)^2 + (0-0)^2 + (1-.5)^2 + (1-1.0)^2 = 0+0+.25+0$ (winner)
 - Distance from unit2 weights
 - $(0-1)^2 + (0-.5)^2 + (1-0)^2 + (1-0)^2 = 1+.25+1+1=3.25$

$$\sum_{k=1}^n (i_{l,k} - w_{j,k}(t))^2$$

$$d^2 = (\text{Euclidean distance})^2 = \frac{\text{EELU ITF309 Neural Network}}{\text{Lecture 11}}$$

Examples of Applications

- Kohonen (1984). Speech recognition - a map of phonemes in the Finish language
- Optical character recognition - clustering of letters of different fonts
- Angeliol *etal* (1988) – travelling salesman problem (an optimization problem)
- Kohonen (1990) – learning vector quantization (pattern classification problem)
- Ritter & Kohonen (1989) – semantic maps

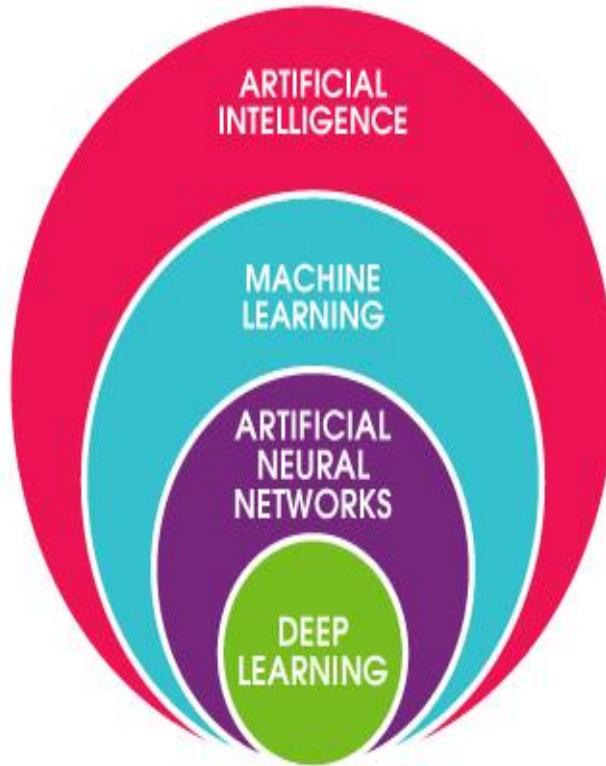
Summary

- Unsupervised learning is very common
- US learning requires redundancy in the stimuli
- Self organization is a basic property of the brain's computational structure
- SOMs are based on
 - competition (wta units)
 - cooperation
 - synaptic adaptation
- SOMs conserve topological relationships between the stimuli
- Artificial SOMs have many applications in computational neuroscience

ANN Architecture Taxonomy

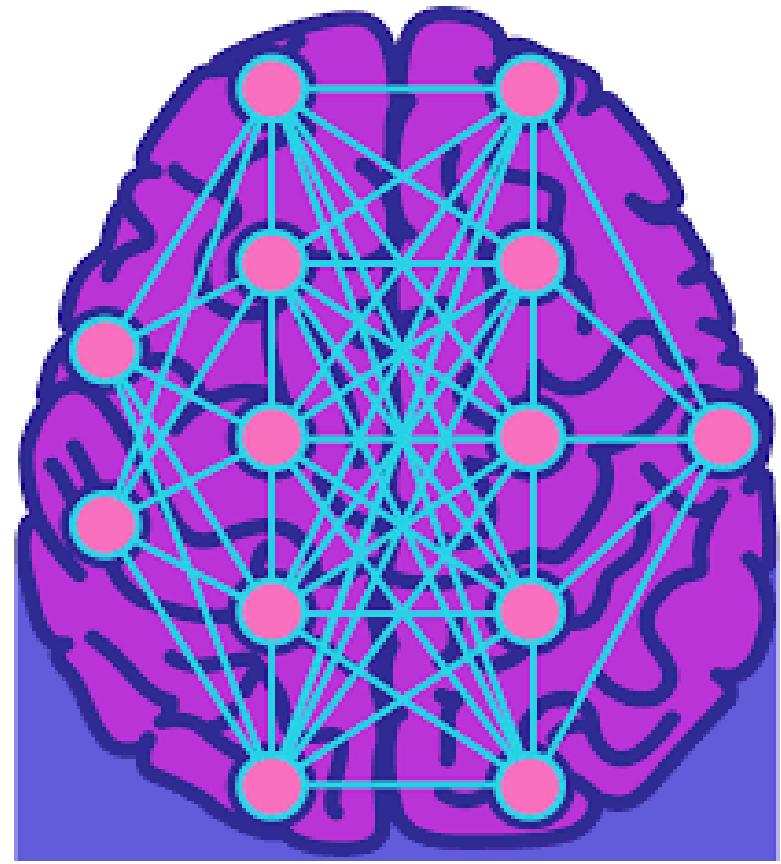


What is Deep Learning



ITF309

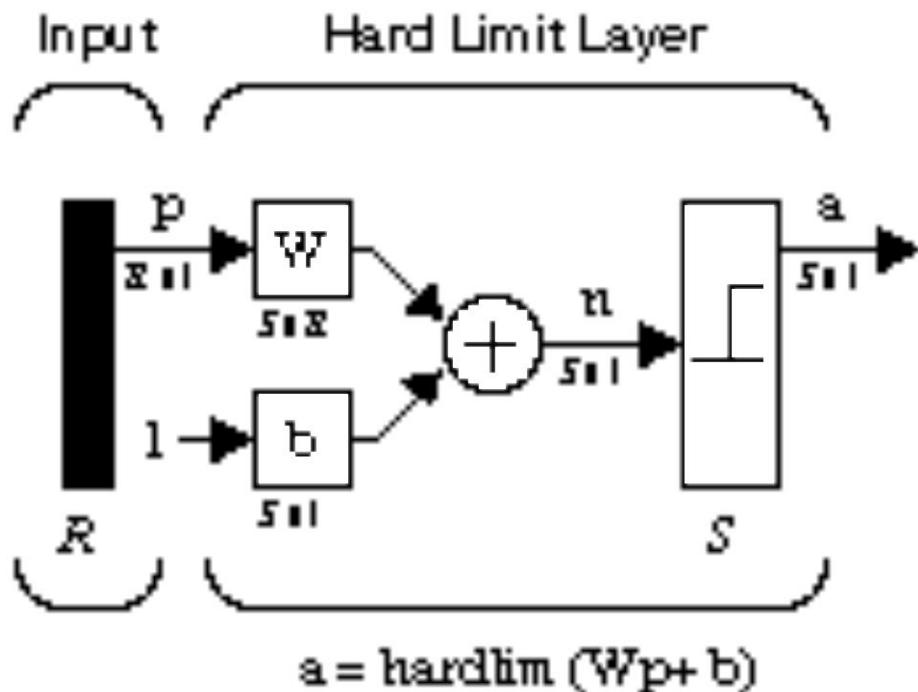
Artificial Neural Networks



Outline

- Supervised Hebbian Learning

Review: Perceptron Architecture



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

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

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

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

Review: Perceptron Learning Rule

- Perceptron learning rule

$$W^{new} = W^{old} + ep^T$$

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

- How can we derive the perceptron learning rule or justify it?

Objectives

- The **Hebb rule**, proposed by Donald Hebb in 1949, was one of the first neural network **learning laws**.
- A possible mechanism for **synaptic modification** in the brain.
- Use the **linear algebra concepts** to explain why Hebbian learning works.
- The **Hebb rule** can be used to train neural networks for **pattern recognition**.

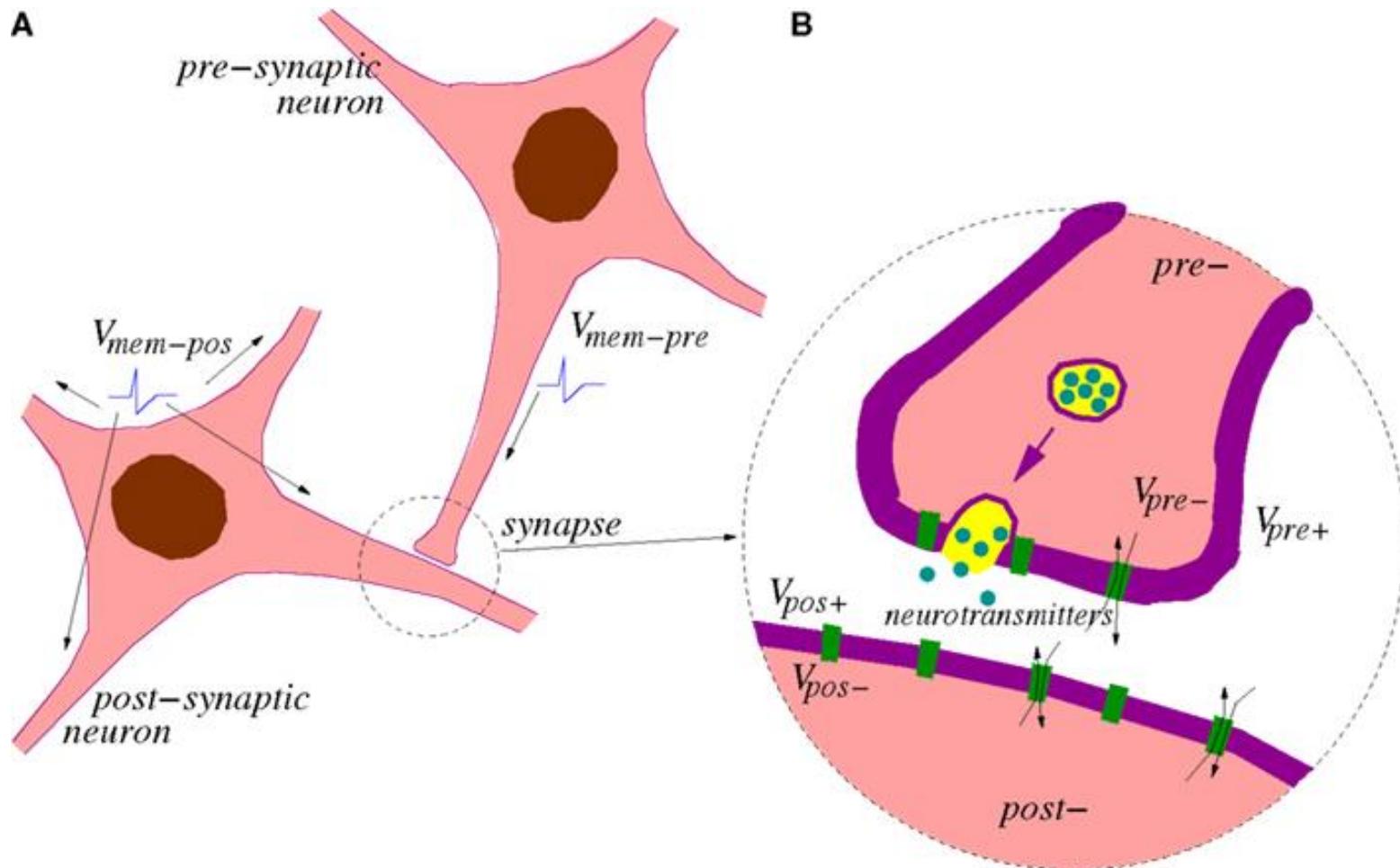
Hebb's Postulate

- **Hebbian learning**

(The Organization of Behavior)

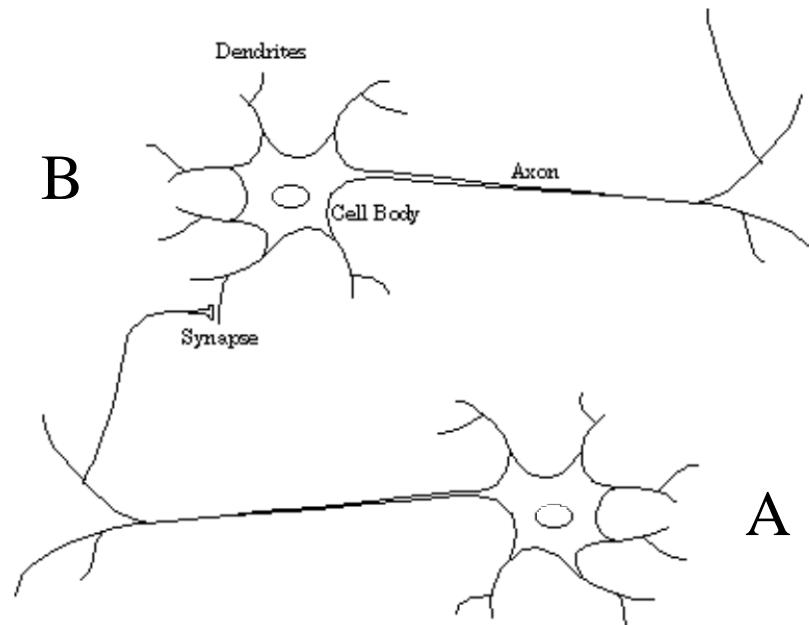
When **an axon of cell A** is **near** enough to **excite a cell B** and repeatedly or persistently takes part in firing it; some **growth process** or **metabolic change** takes place in **one or both cells** such that A's **efficiency**, as one of the cells firing B, is **increased**.

Hebb's Postulate



Hebb's Postulate

- The 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



Hebb (1949)

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

Hebb's rule

- Each time that a particular synaptic connection is active, see if the receiving cell also becomes active. If so, the connection contributed to the success (firing) of the receiving cell and should be strengthened. If the receiving cell was not active in this time period, our synapse did not contribute to the success the trend and should be weakened.

Hebb Rule

- If two neurons on either side of a synapse are activated simultaneously, the strength of the synapse will increase

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

The diagram illustrates the Hebb rule update equation. It shows the formula $w_{ij}^{new} = w_{ij}^{old} + \alpha f_i(a_{iq})g_j(p_{jq})$. Two arrows point upwards from the right side of the equation towards the terms $f_i(a_{iq})$ and $g_j(p_{jq})$. The arrow pointing to $f_i(a_{iq})$ is labeled "Presynaptic Signal". The arrow pointing to $g_j(p_{jq})$ is labeled "Postsynaptic Signal".

Hebb Rule – cont.

Simplified Form:

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

Supervised Form:

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

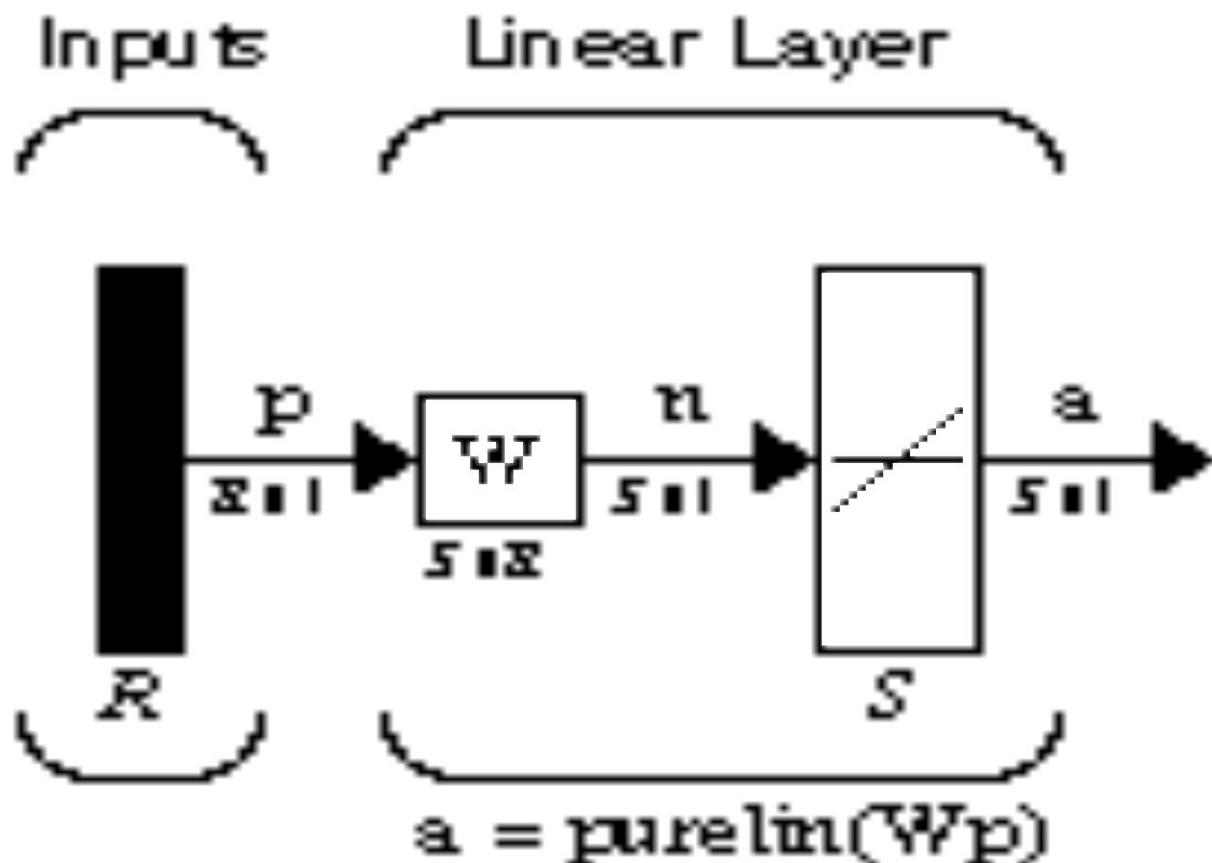
Matrix Form:

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

Linear Associator

- **Associative memory**
 - The task is to learn Q pairs of prototype input/output vectors
$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \dots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$
 - In other words, if the network receives an input $\mathbf{p} = \mathbf{p}_q$, then it should produce an output $\mathbf{a} = \mathbf{t}_q$
 - In addition, for an input similar to a prototype the network should produce an output similar to the corresponding output

Linear Associator – cont.



Linear Associator – cont.

- Hebb rule for the linear associator

$$W^{new} = W^{old} + t_q p_q^T$$

Batch Operation

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

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

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

Case I, input patterns are orthogonal.

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

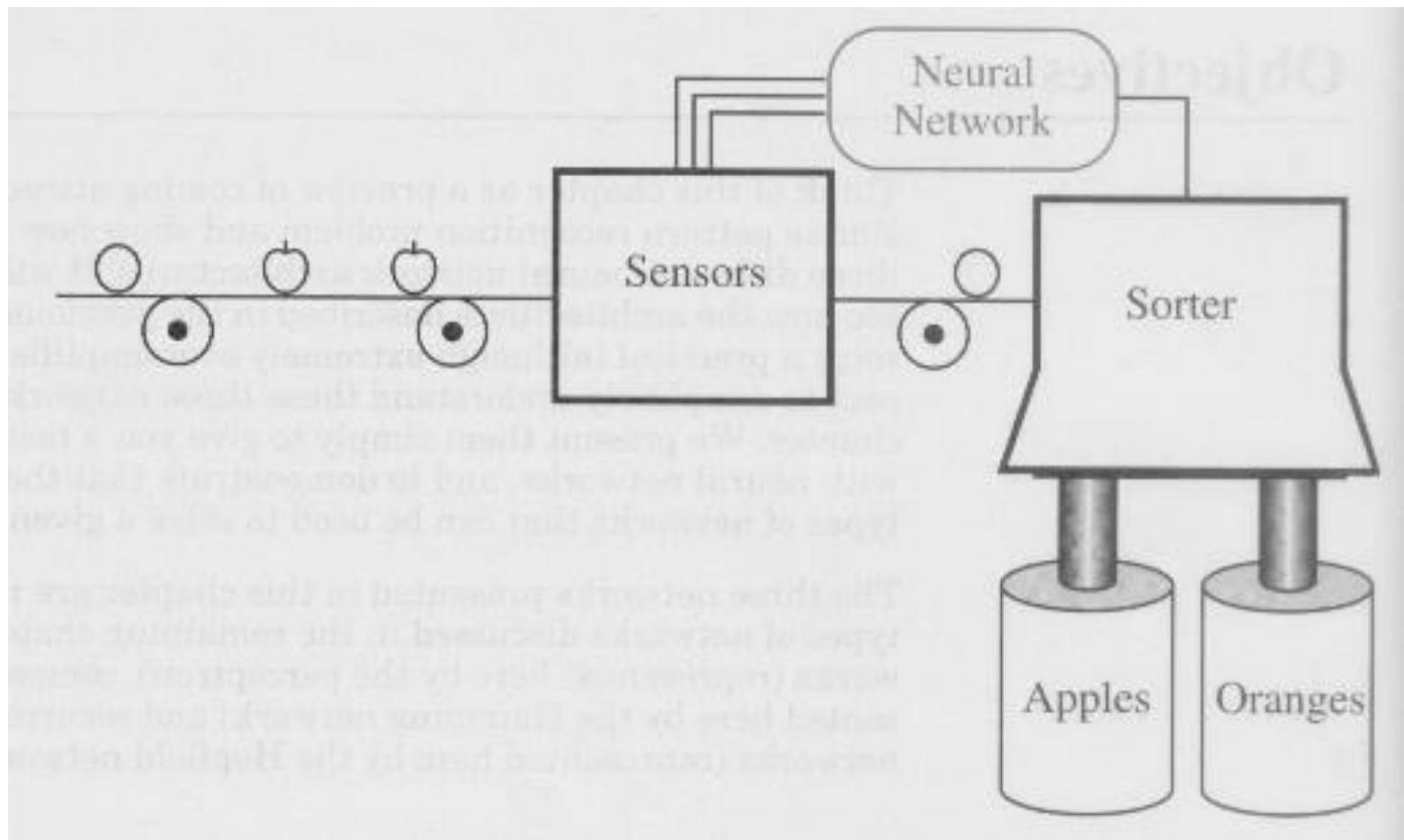
Therefore the network output equals the target: $\mathbf{a} = \mathbf{W}\mathbf{p}_k = \mathbf{t}_k$

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

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

Error

Linear Associator – cont.



Example: Orthonormal Case

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

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

$$\Rightarrow \mathbf{Wp}_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \mathbf{Wp}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}. \text{ Success!}$$

Example: Not Orthogonal Case

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

Weight Matrix (Hebb Rule):

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

Tests:

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

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

Not Orthogonal Case

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} 0.5774 \\ -0.5774 \\ -0.5774 \end{bmatrix}, \mathbf{t}_1 = [-1] \right\}, \left\{ \mathbf{p}_2 = \begin{bmatrix} 0.5774 \\ 0.5774 \\ -0.5774 \end{bmatrix}, \mathbf{t}_2 = [1] \right\}$$
$$\Rightarrow \mathbf{W} = \mathbf{T}\mathbf{P}^T = [-1 \quad 1] \begin{bmatrix} 0.5774 & -0.5774 & -0.5774 \\ 0.5774 & 0.5774 & -0.5774 \end{bmatrix}$$
$$= [0 \quad 1.547 \quad 0]$$

$$\Rightarrow \mathbf{W}\mathbf{p}_1 = [-0.8932], \quad \mathbf{W}\mathbf{p}_2 = [0.8932]$$

The outputs are **close**, but do not quite match the target outputs.

Pseudoinverse Rule

Performance Index: $\mathbf{Wp}_q = \mathbf{t}_q \quad q = 1, 2, \dots, Q$

$$F(\mathbf{W}) = \sum_{q=1}^Q \|\mathbf{t}_q - \mathbf{Wp}_q\|^2$$

Matrix Form: $\mathbf{WP} = \mathbf{T}$

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

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

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

Pseudoinverse Rule - (2)

$$\mathbf{WP} = \mathbf{T}$$

Minimize:

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

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

$$\mathbf{W} = \mathbf{TP}^{-1}$$

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

$$\mathbf{W} = \mathbf{TP}^+$$

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

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

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

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

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

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

Another Pattern Recognition Example

- P7.4 on pp. 7-22 – 7-23
- Given the following three patterns

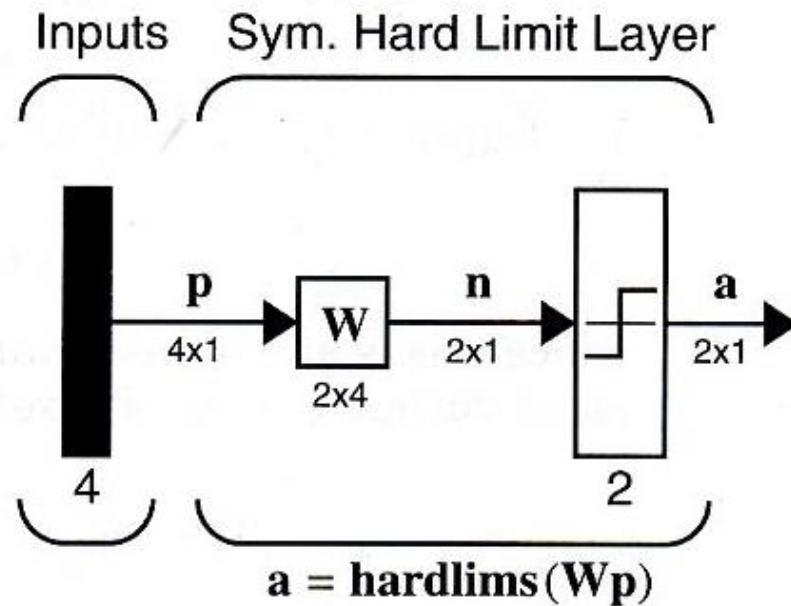


$\mathbf{p}_1 \quad \mathbf{p}_2 \quad \mathbf{p}_3$

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ -1 \\ 1 \\ 1 \end{bmatrix} \quad \mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix} \quad \mathbf{p}_3 = \begin{bmatrix} -1 \\ -1 \\ -1 \\ 1 \end{bmatrix} \quad \mathbf{p}_t = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}.$$

$$\mathbf{t}_1 = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \quad \mathbf{t}_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \quad \mathbf{t}_3 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

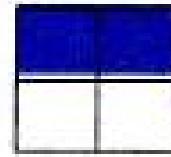
Another Pattern Recognition Example – cont.



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

Another Pattern Recognition Example – cont.

- For the following test pattern



$$\begin{aligned} \mathbf{p}_t &= \mathbf{hardlims}(\mathbf{W}\mathbf{p}_t) = \mathbf{hardlims} \left(\begin{bmatrix} -3 & -1 & -1 & -1 \\ 1 & 3 & -1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \right) \\ &= \mathbf{hardlims} \left(\begin{bmatrix} -2 \\ -2 \end{bmatrix} \right) = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \rightarrow \mathbf{p}_1. \end{aligned}$$

Associative memory

- Auto-associative memory, often misunderstood to be only a form of back-propagation or other neural networks. It is actually a more generic term that refers to all memories that enable one to retrieve a piece of data from only a tiny sample of itself.
- **Associative memory is defined as the ability to learn and remember the relationship between unrelated items such as the name of someone we have just met or the aroma of a particular perfume.**

Autoassociative & heteroassociative

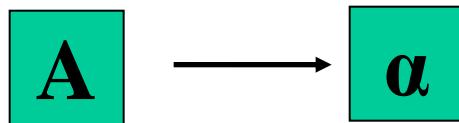
- That is to say, a system that "associates" two patterns is one that, when presented only of these patterns later, the other can be reliably recalled.
- A memory that reproduces its input pattern as output is referred to as *autoassociative* (i.e. associating patterns with themselves). One that produces output patterns dissimilar to its inputs is termed *heteroassociative* (i.e. associating patterns with other patterns).

With Hebbian learning (1949) , two learning methods are possible

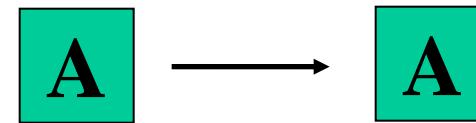
- With *supervised learning* an input is associated with an output
 - If the input and output are the same, we speak of *auto-associative* learning
 - If they are different it is called *hetero-associative* learning
- With *unsupervised learning* there is no teacher: the network tries to discern regularities in the input patterns

Associative memory

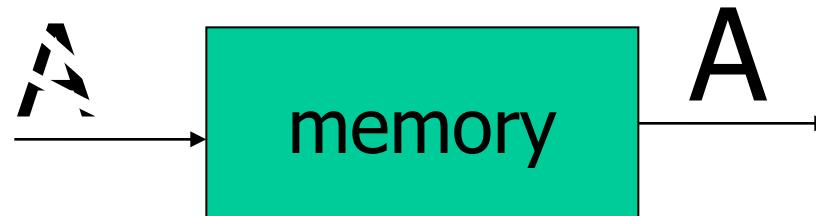
Hetero associative



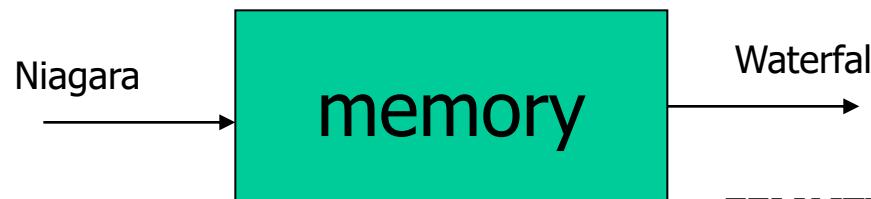
Auto associative



Auto-association (Same Patterns)

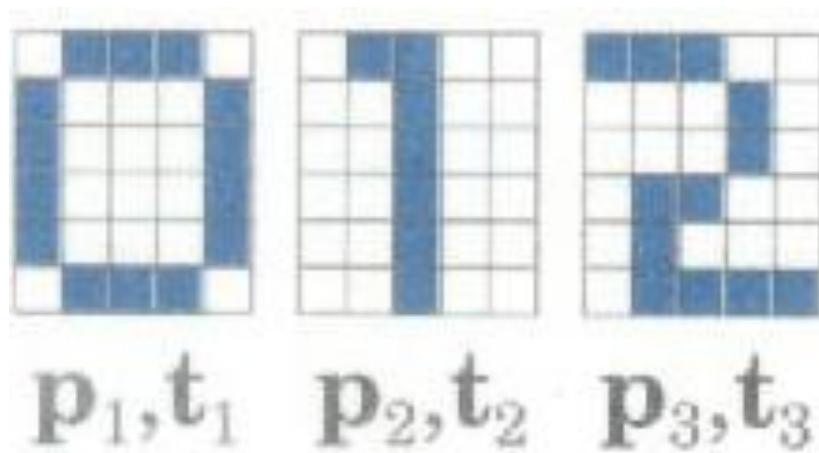


Hetero-association (Different Patterns)



Autoassociative Memory

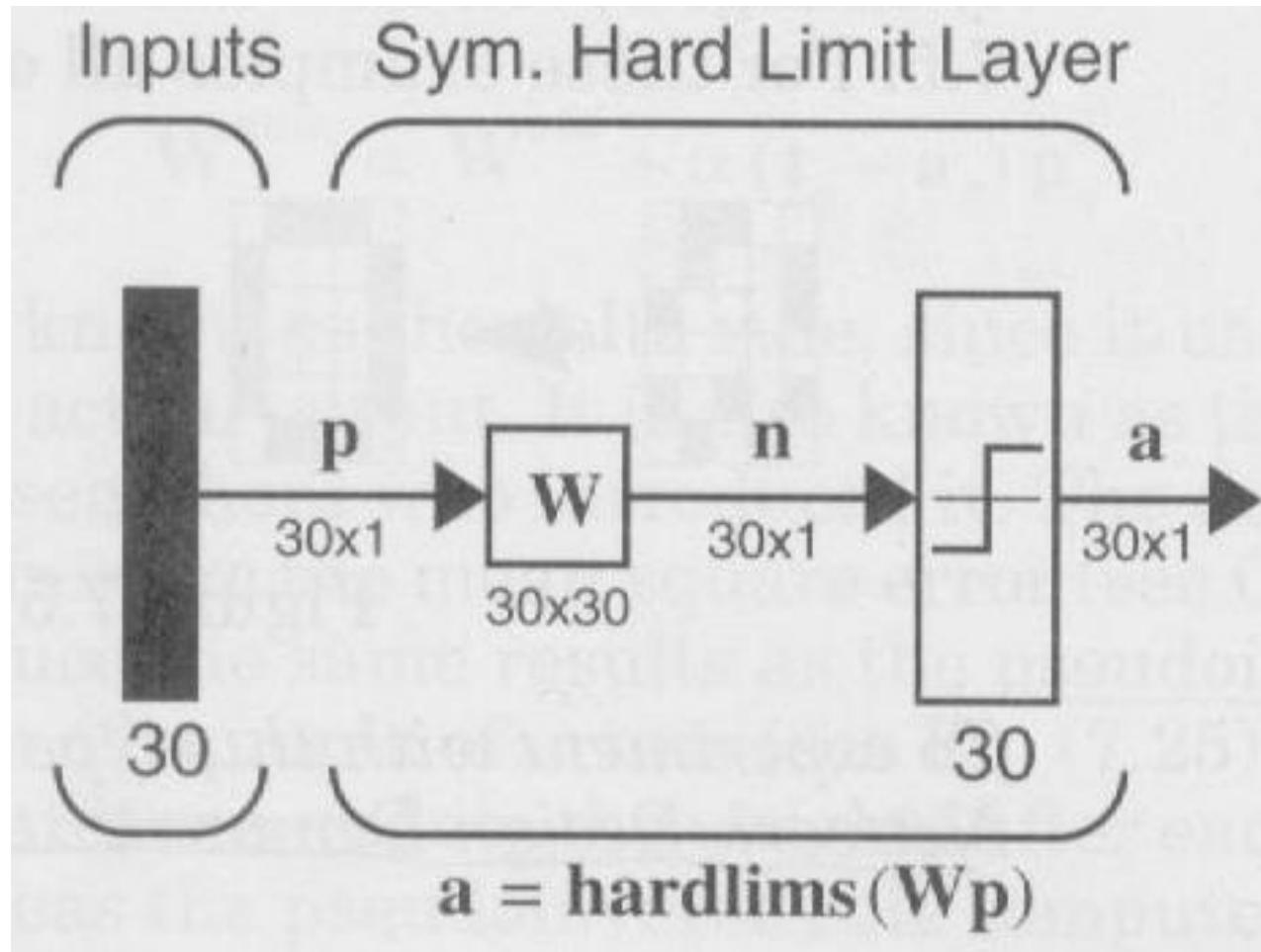
- Autoassociative memory tries to memorize the input patterns
 - The desired output is the input vector



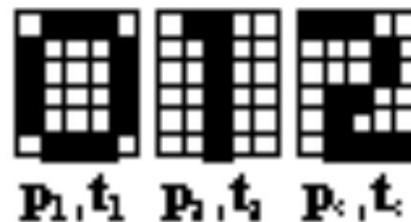
Autoassociative Memory – cont.

- Linear autoassociator
 - For orthogonal patterns with elements to be either 1 or -1, the prototypes are the eigenvectors of the weight matrix given by the Hebb rule
 - The given pattern is not recalled correctly but is amplified by a constant factor
 - See P7.5 (pp. 7-23 – 7-25)

Autoassociative Memory – cont.

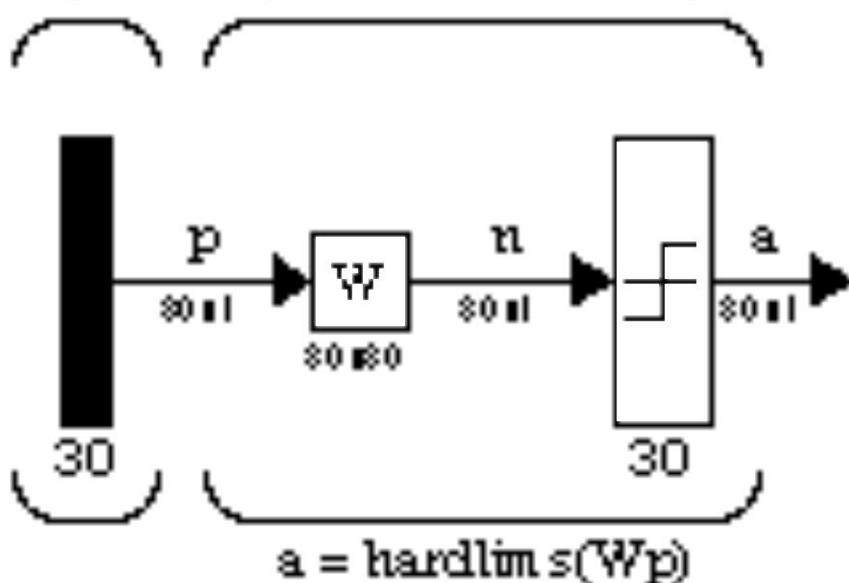


Autoassociative Memory



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

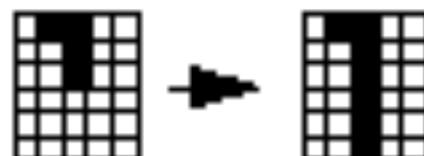
Inputs Sym. Hard Limit Layer



$$\mathbf{W} = \mathbf{p}_1\mathbf{p}_1^T + \mathbf{p}_2\mathbf{p}_2^T + \mathbf{p}_3\mathbf{p}_3^T$$

Tests

50% Occluded



67% Occluded



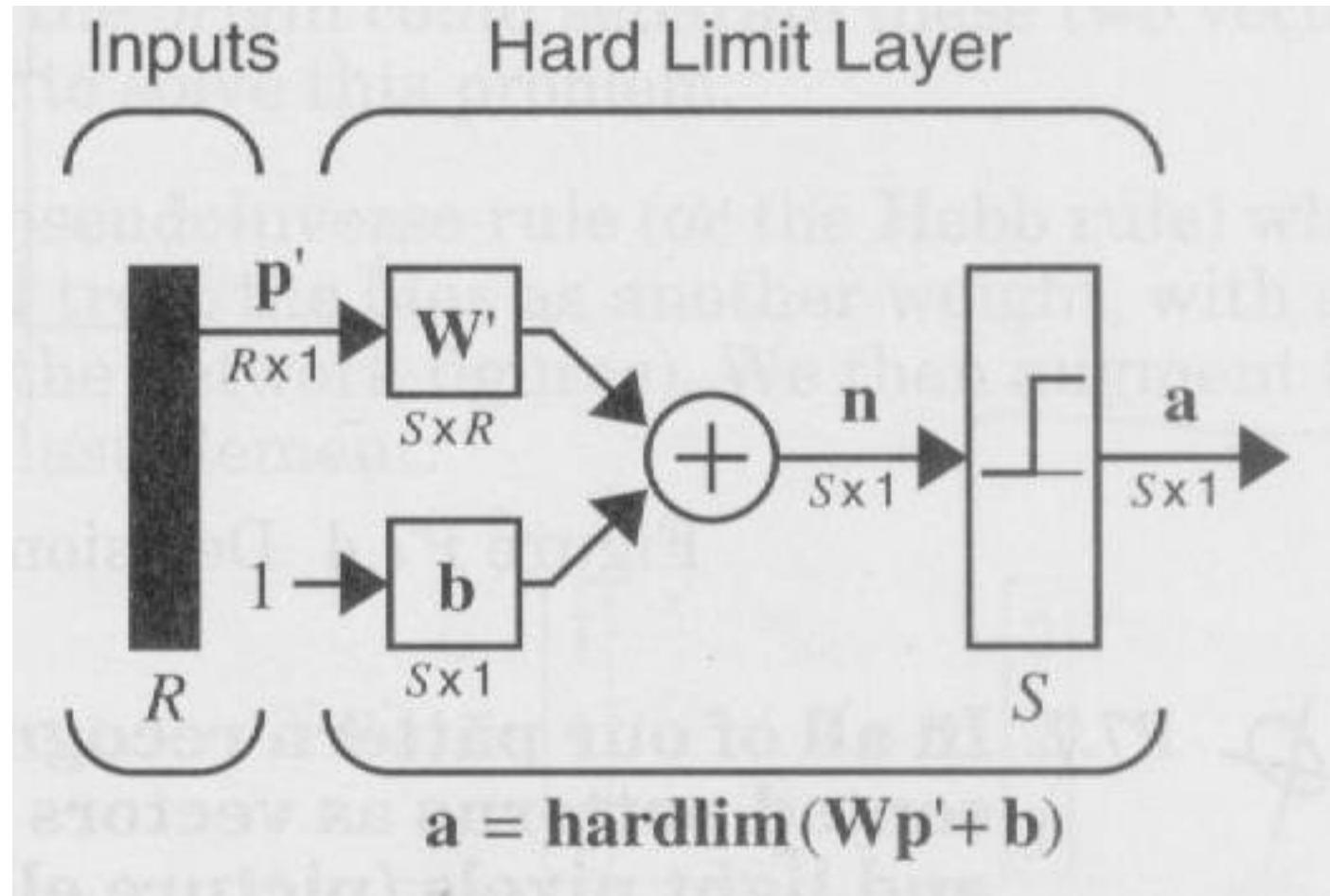
Noisy Patterns (7 pixels)



Autoassociative Memory – cont.

- How can we determine weights and bias for a network with bias?
 - P7.6 (pp. 7-25 --- 7-27)
- How can we determine weight and bias for a network with desired outputs are 0 and 1's instead of -1 and 1?
 - P7.7 (pp. 7-27 – 7-28)

Autoassociative Memory – cont.



Variations of Hebbian Learning

- Hebb rule is the most fundamental learning rule and many learning rules are related to it

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

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

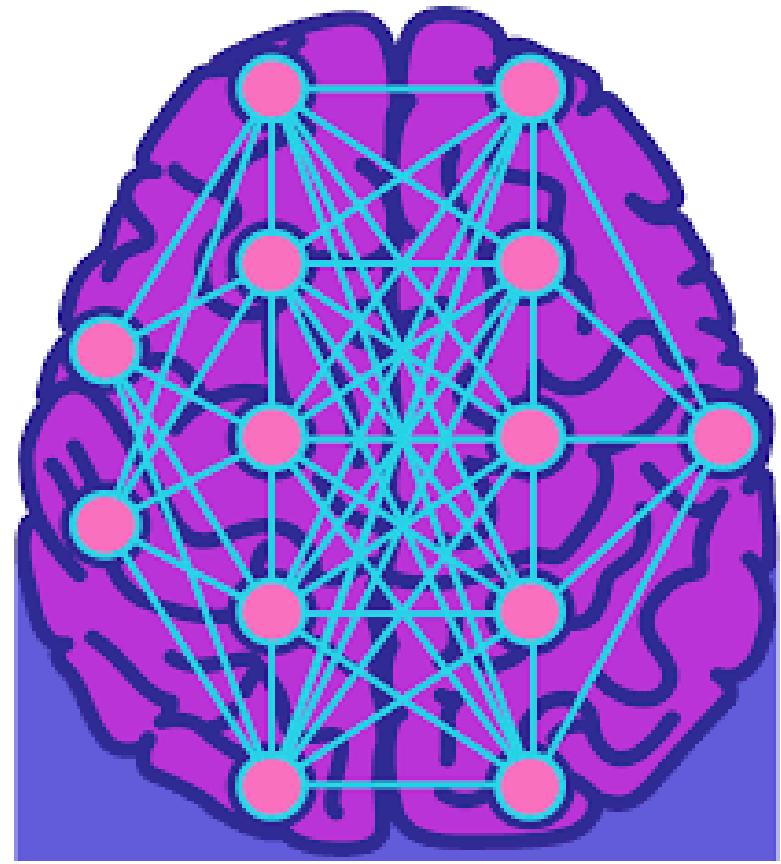
Smoothing: $\mathbf{W}^{new} = \mathbf{W}^{old} + \alpha \mathbf{t}_q \mathbf{p}_q^T - \gamma \mathbf{W}^{old} = (1 - \gamma) \mathbf{W}^{old} + \alpha \mathbf{t}_q \mathbf{p}_q^T$

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

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

ITF309

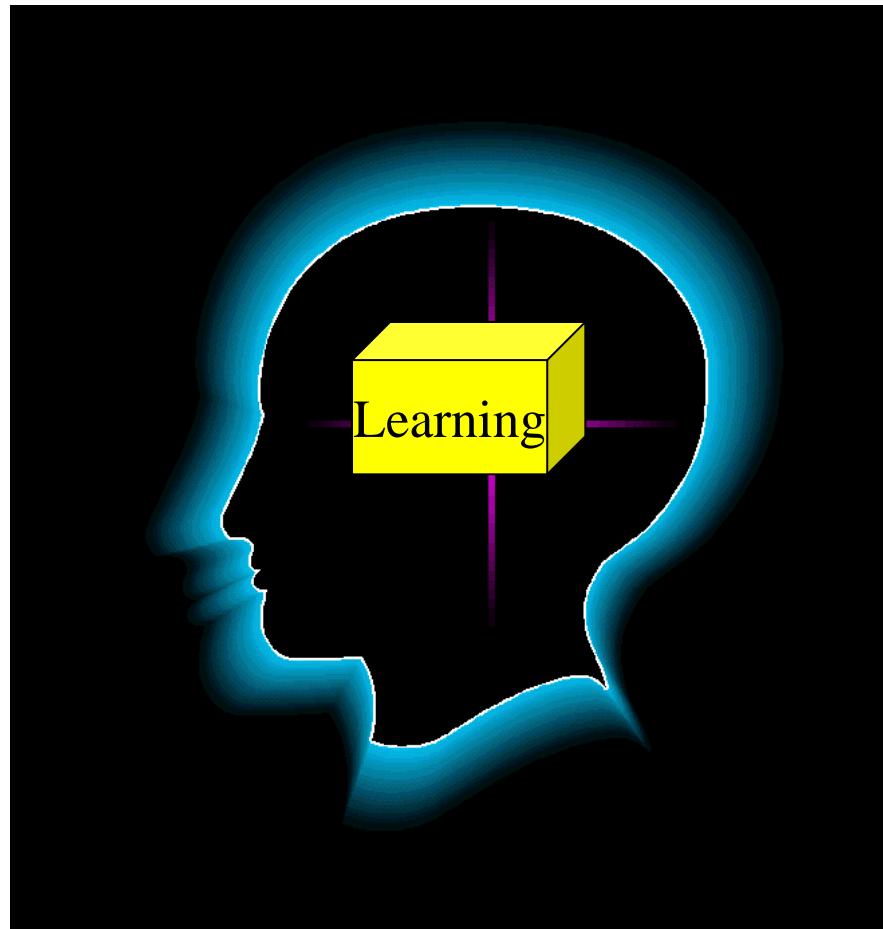
Artificial Neural Networks



Outline

- Neural models and neural network architectures
 - Learning Processes
 - Neural model
 - Single-input neuron
 - Transfer functions
 - Multiple-input neurons
 - Network architecture

Learning Processes



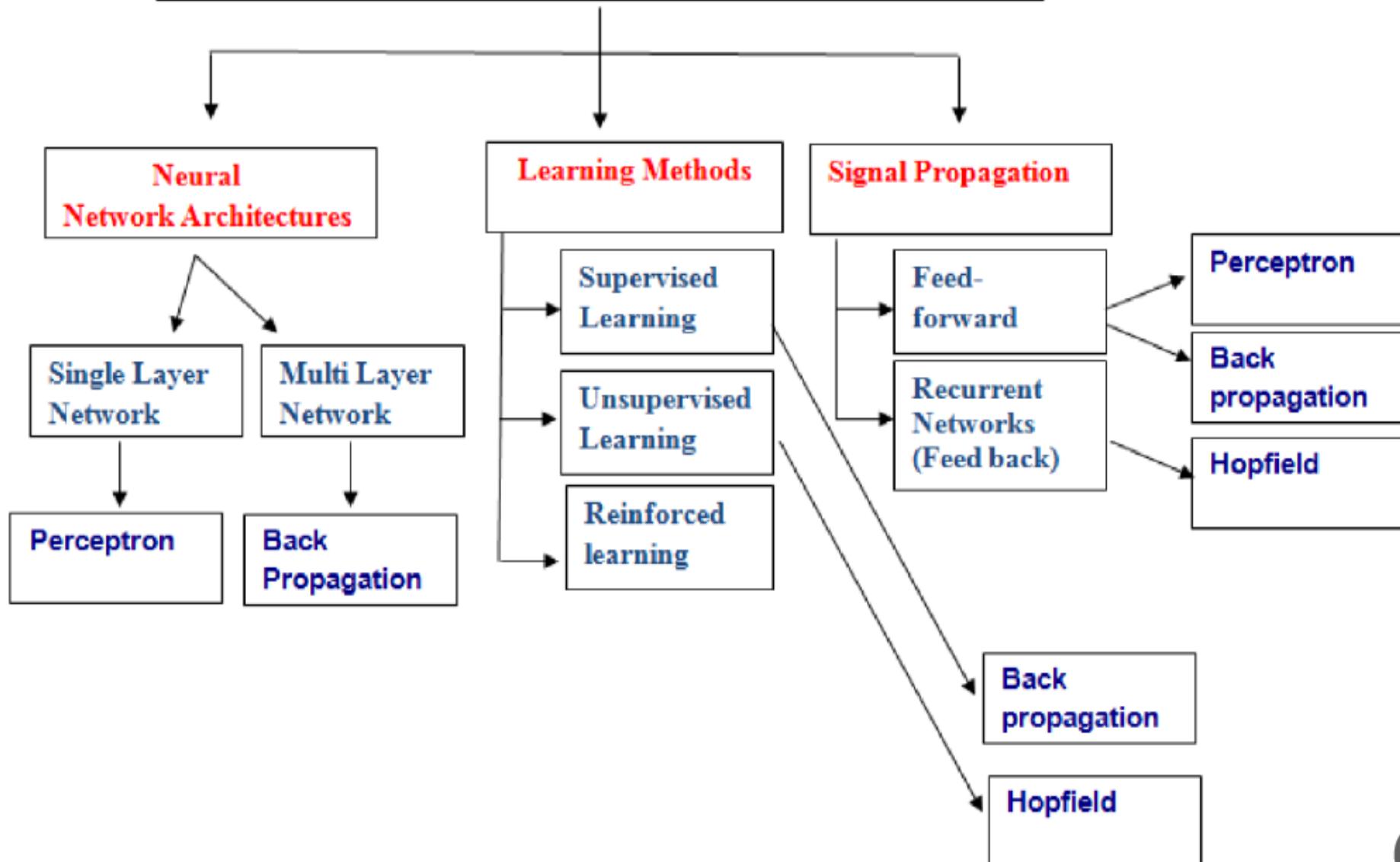
What is Learning?

- Learning is a process by which a system modifies its behavior by adjusting its parameters in response to the stimulation by the environment.
- Elements of Learning
 - Stimulation from the environment
 - Parameter adaptation rule
 - Change of behavior of the system

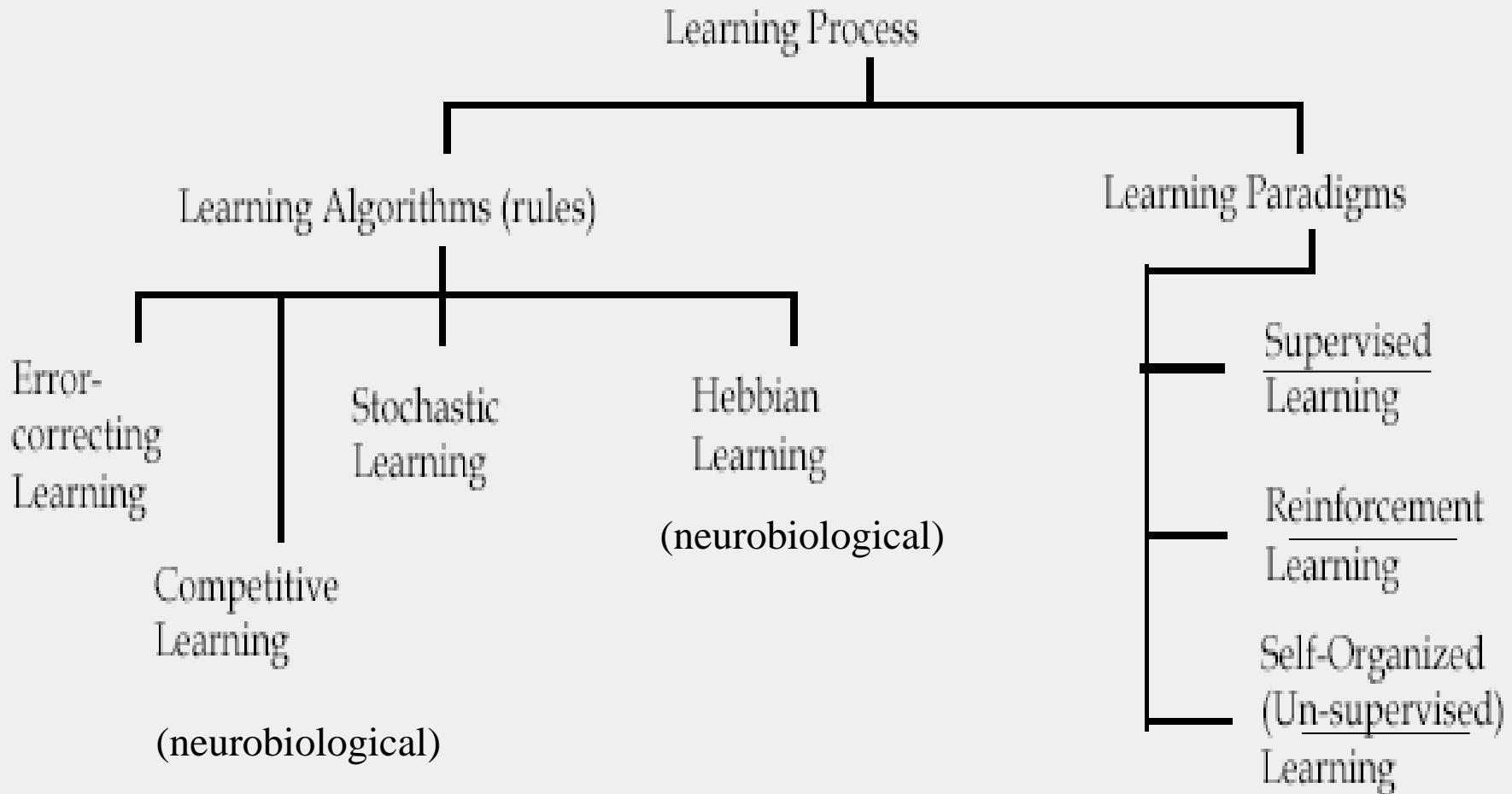
Learning of NN

- Learning of NN is a process by which the free parameters of a neural network are adapted through a process of stimulation by the environment in which the network is embedded.
- The **type of the learning** is determined by the manner in which the parameter changes take place. (Mendel & McLaren 1970)

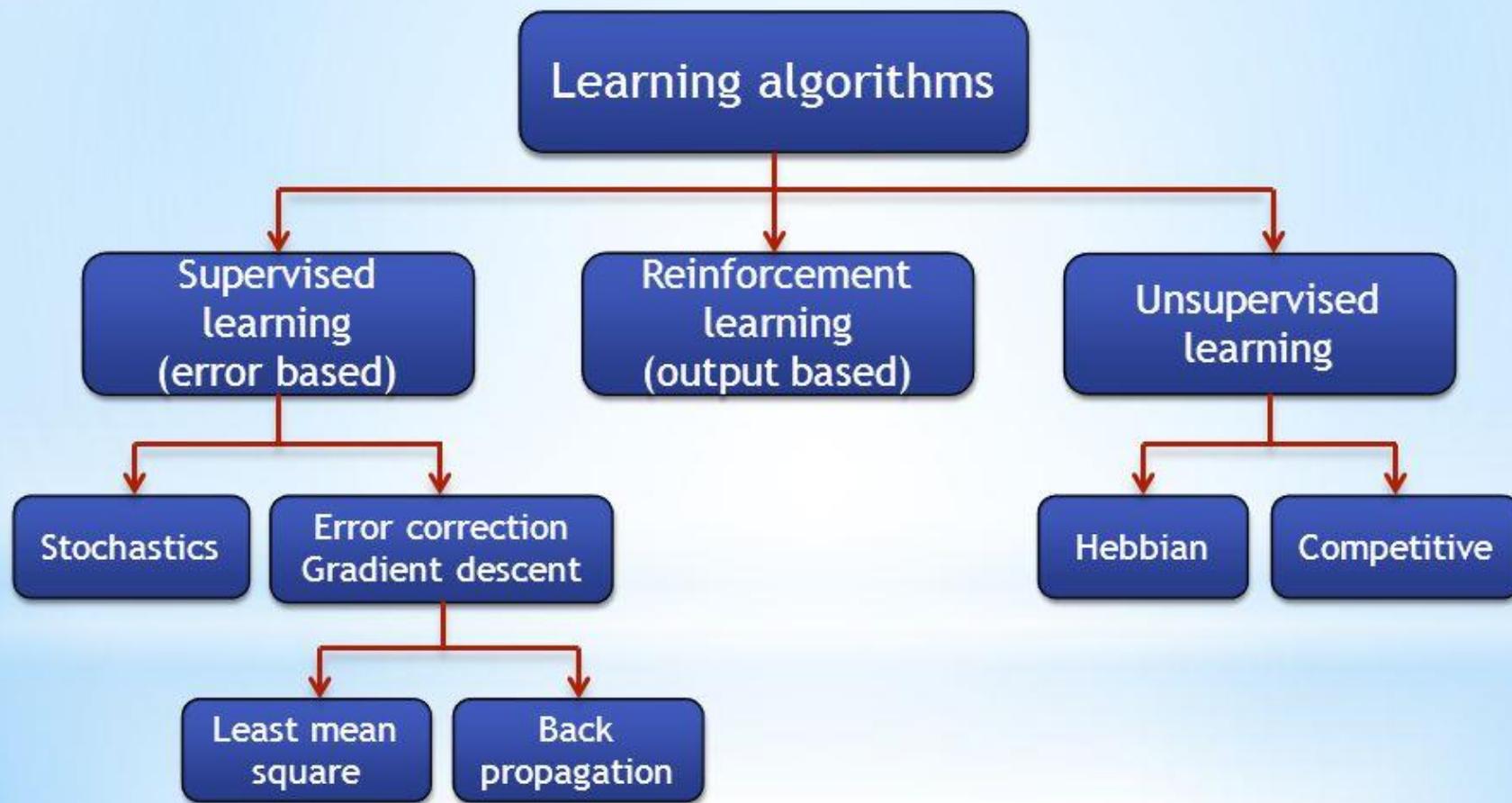
Classification of Artificial Neural Networks



Taxonomy of Learning



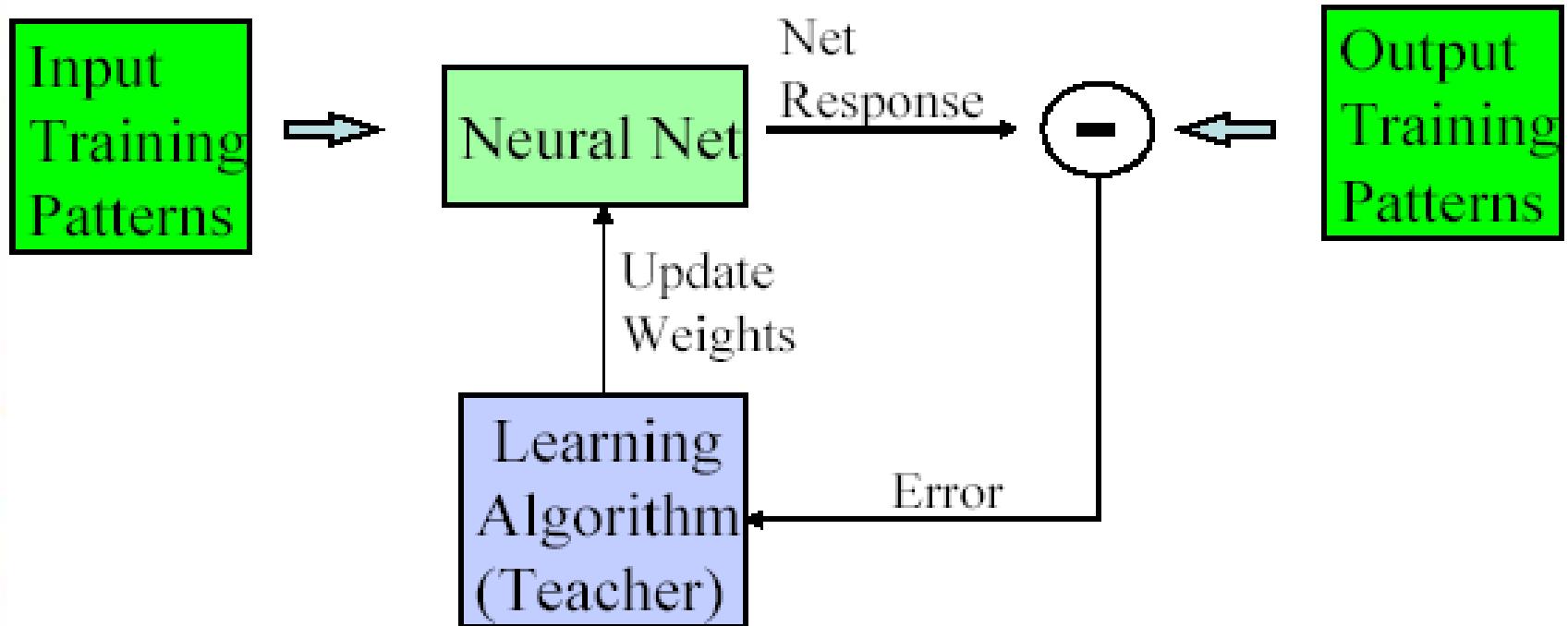
ANN - Training or Learning



Supervised Learning

- In supervised learning, the network is presented with inputs together with the target (teacher signal) outputs.
- Then, the neural network tries to produce an output as close as possible to the target signal by adjusting the values of internal weights.
- The most common supervised learning method is the **“error correction method”**, using methods such as
 - Least Mean Square (LMS)
 - Back Propagation

NN - Supervised Learning



Learning Algorithm trains weights to
minimize Error cost function

Unsupervised Learning

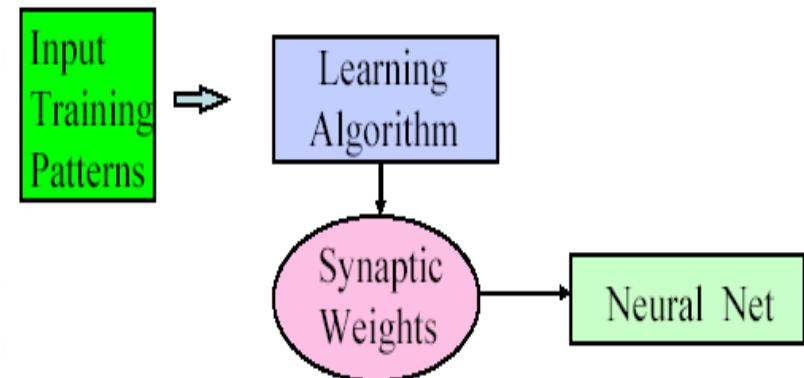
- In unsupervised learning, there is no teacher (target signal) from outside and the network adjusts its weights in response to only the input patterns.
- A typical example of unsupervised learning is
-

Unsupervised Hebbian learning

- Ex. Principal component analysis

Unsupervised competitive learning

- Ex. Clustering,
- Data compression



Learning Algorithm trains weights to reach some internal cost function

Unsupervised Competitive Learning

- In *unsupervised competitive learning* the neurons take part in some competition for each input. The winner of the competition and sometimes some other neurons are allowed to change their weights
- In *simple competitive learning* only the winner is allowed to learn (change its weight).
- In **self-organizing maps** other neurons in the neighborhood of the winner may also learn.

Learning Tasks

Supervised

Data:
Labeled examples
(input , desired output)

Tasks:
classification
pattern recognition
regression

NN models:
perceptron
adaline
feed-forward NN
radial basis function
support vector machines

Unsupervised

Data:
Unlabeled examples
(different realizations of the input)

Tasks:
clustering

NN models:
self-organizing maps (SOM)
Hopfield networks

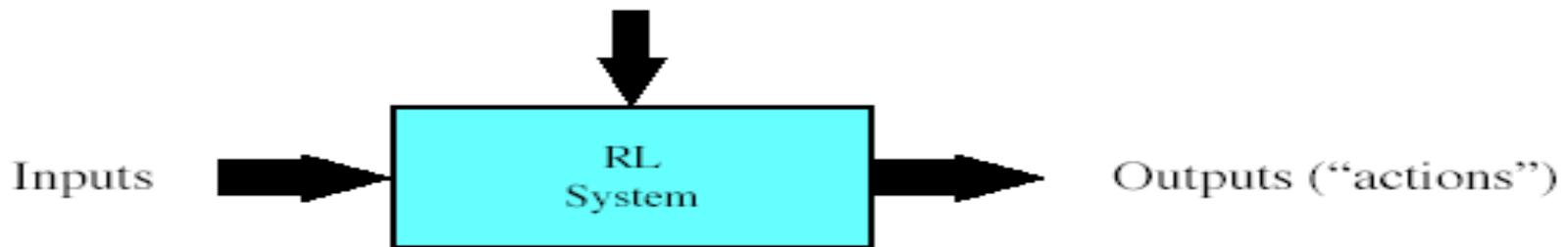
Reinforcement Learning

University of Guelph

Computer Science Department



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



Objective: get as much reward as possible

Reinforcement learning: Generalization of Supervised Learning;
Reward given later, not necessarily tied to specific action

Learning Laws

- Perceptron Rule
- Hebb's Rule
- Hopfield Law
- Delta Rule (Least Mean Square Rule)
- The Gradient Descent Rule
- Kohonen's Learning Law

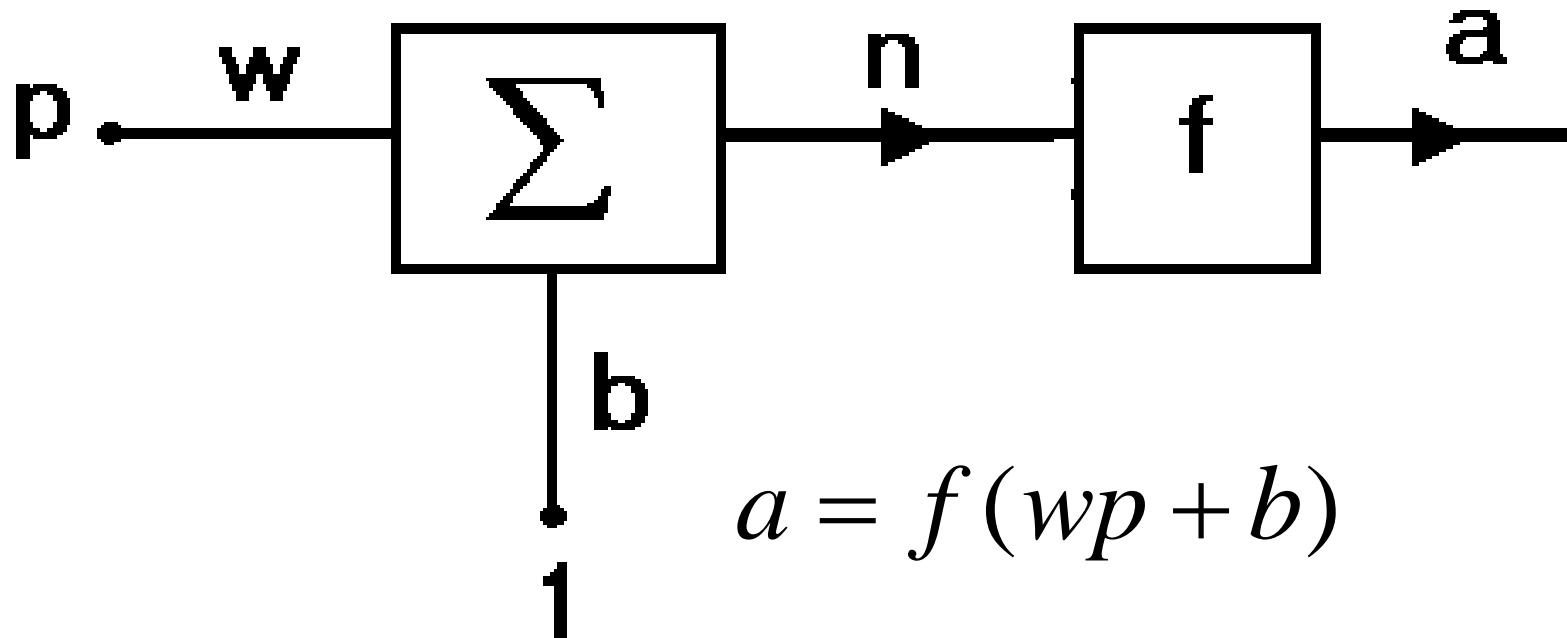
Neuron Models

- McCulloch-Pitts neuron is one example
 - Different functions can be used
 - Different connectivity patterns can be used

McCulloch and Pitts Model

- McCulloch and Pitts Neuron
 - It is a simple model as a binary threshold unit
 - The neuron first computes a weighted sum of its inputs
 - It outputs one if the weighted sum is above a threshold and zero otherwise

A Single-Input Neuron



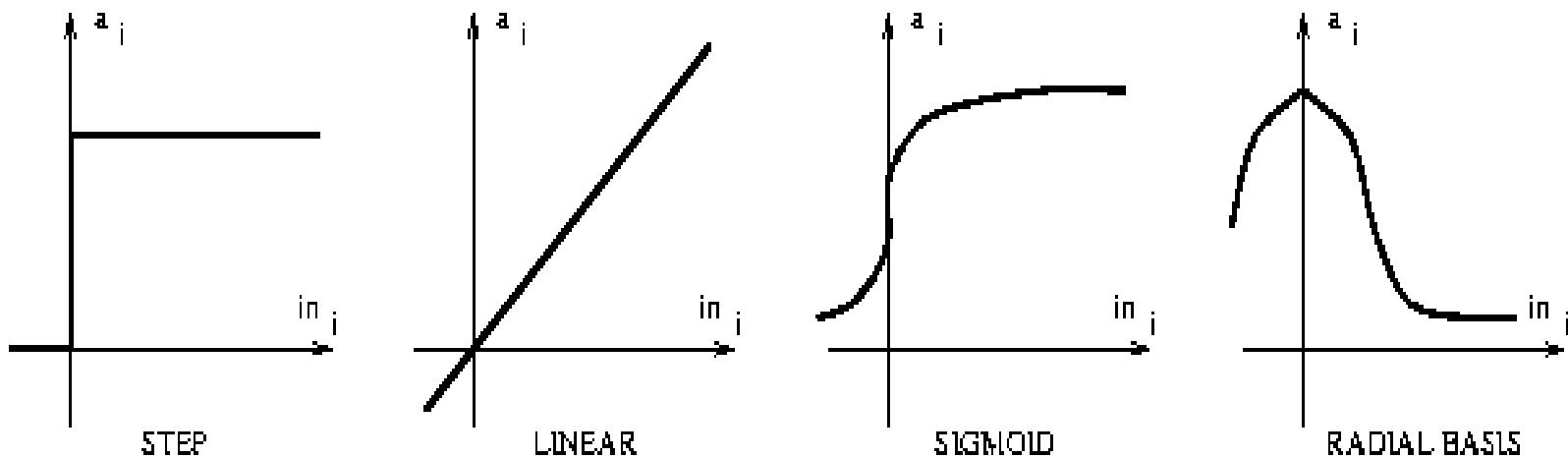
Note that w and b are adjustable parameters of the neurons: w is called weight and b is called bias

A Single-Input Neuron – cont.

- Transfer functions
 - They can be linear or nonlinear
 - What is a linear function?
 - What is a nonlinear function?
 - Commonly used transfer functions
 - Hard limit transfer function
 - Linear transfer function
 - Sigmoid function

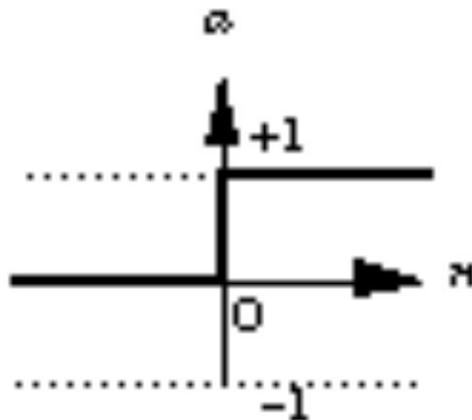
A Single-Input Neuron – cont.

- Transfer functions
 - $a = f(n)$



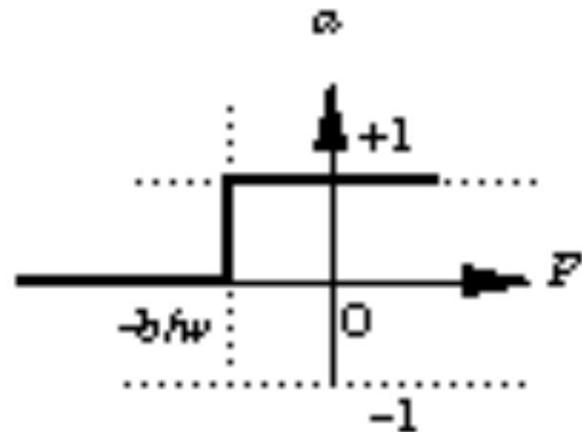
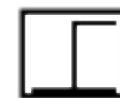
A Single-Input Neuron – cont.

- Hard limit transfer function
 - Also known as the step function



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

Hard Limit Transfer Function



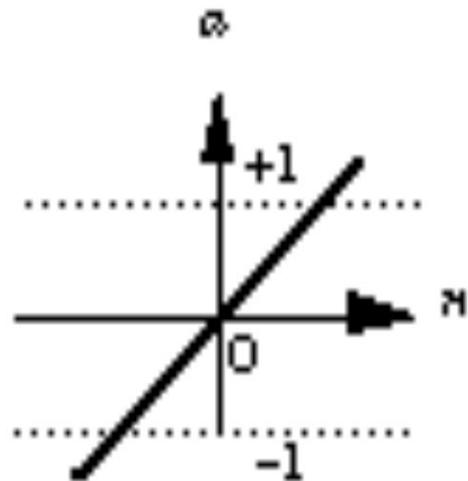
$$a = \text{hardlim}(wp+b)$$

Single-Input Hardlim Neuron

- Symmetrical hard limit transfer function

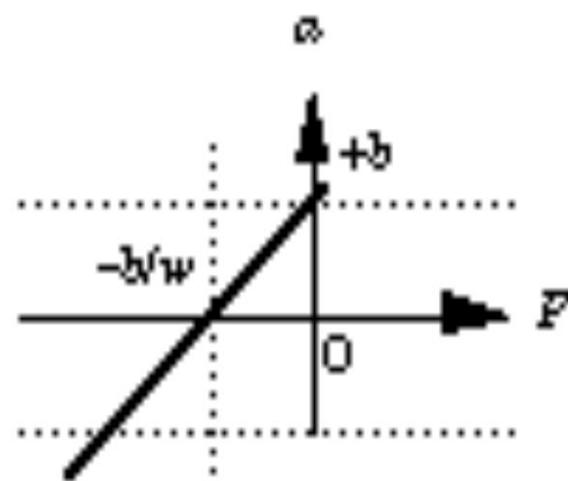
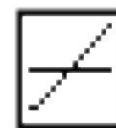
A Single-Input Neuron – cont.

- Linear transfer function



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

Linear Transfer Function

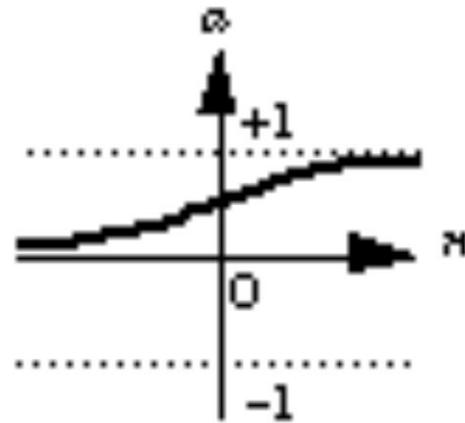


$$a = \text{purelin}(wp + b)$$

Single-Input purelin Neuron

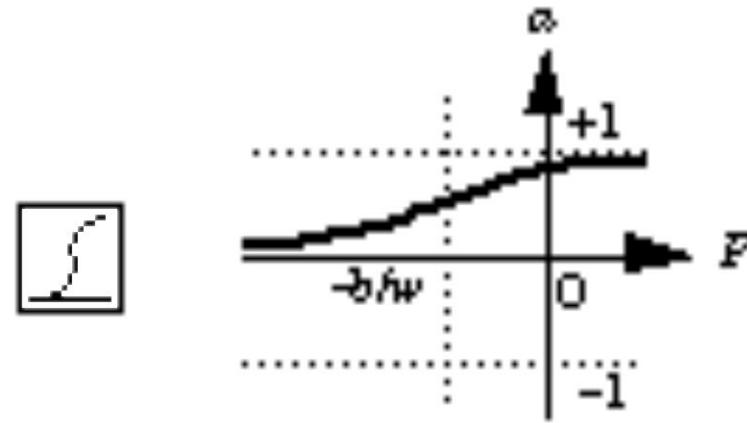
A Single-Input Neuron – cont.

- Log-sigmoid transfer function



$$a = \log \frac{1}{1 + e^{-x}}$$

Log-Sigmoid Transfer Function



$$a = \log \frac{1}{1 + e^{-(wx + b)}}$$

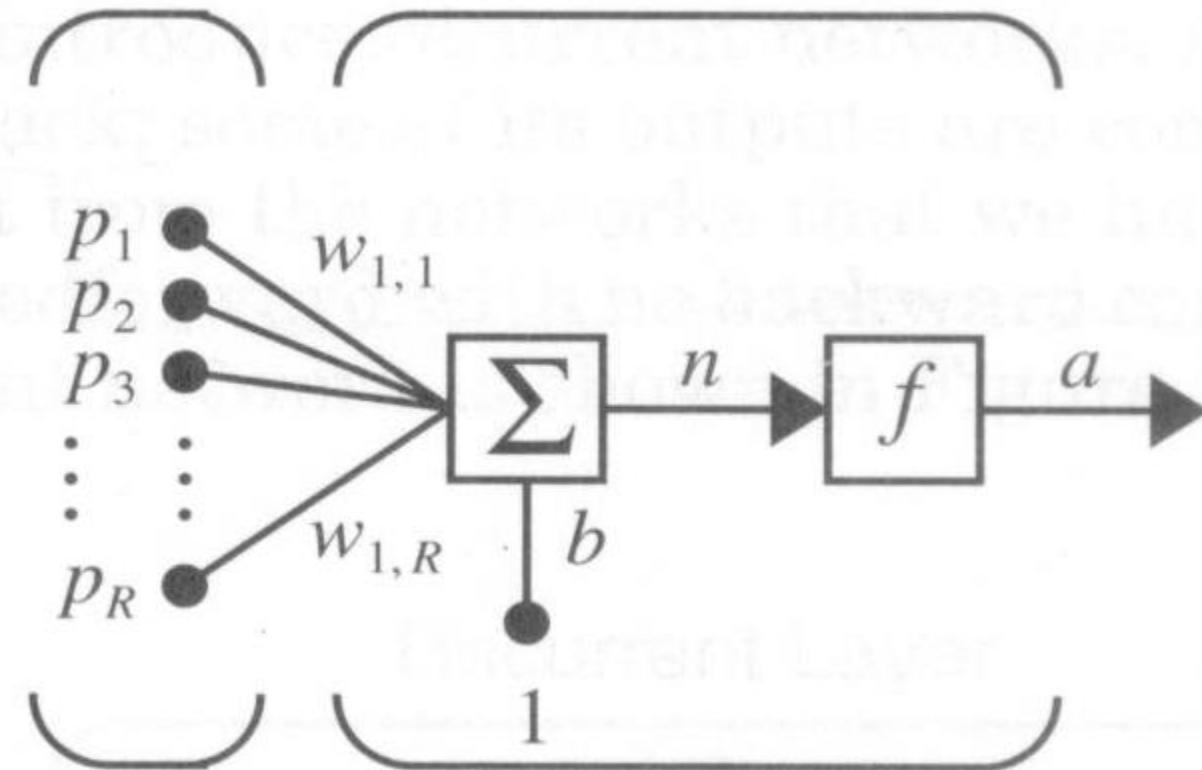
Single-Input log-sigmoid Neuron

A Single-Input Neuron – cont.

- Other transfer functions
 - See Table 2.1 on p. 2-6
- How to choose transfer functions
 - Transfer functions are determined by the input and desired output range

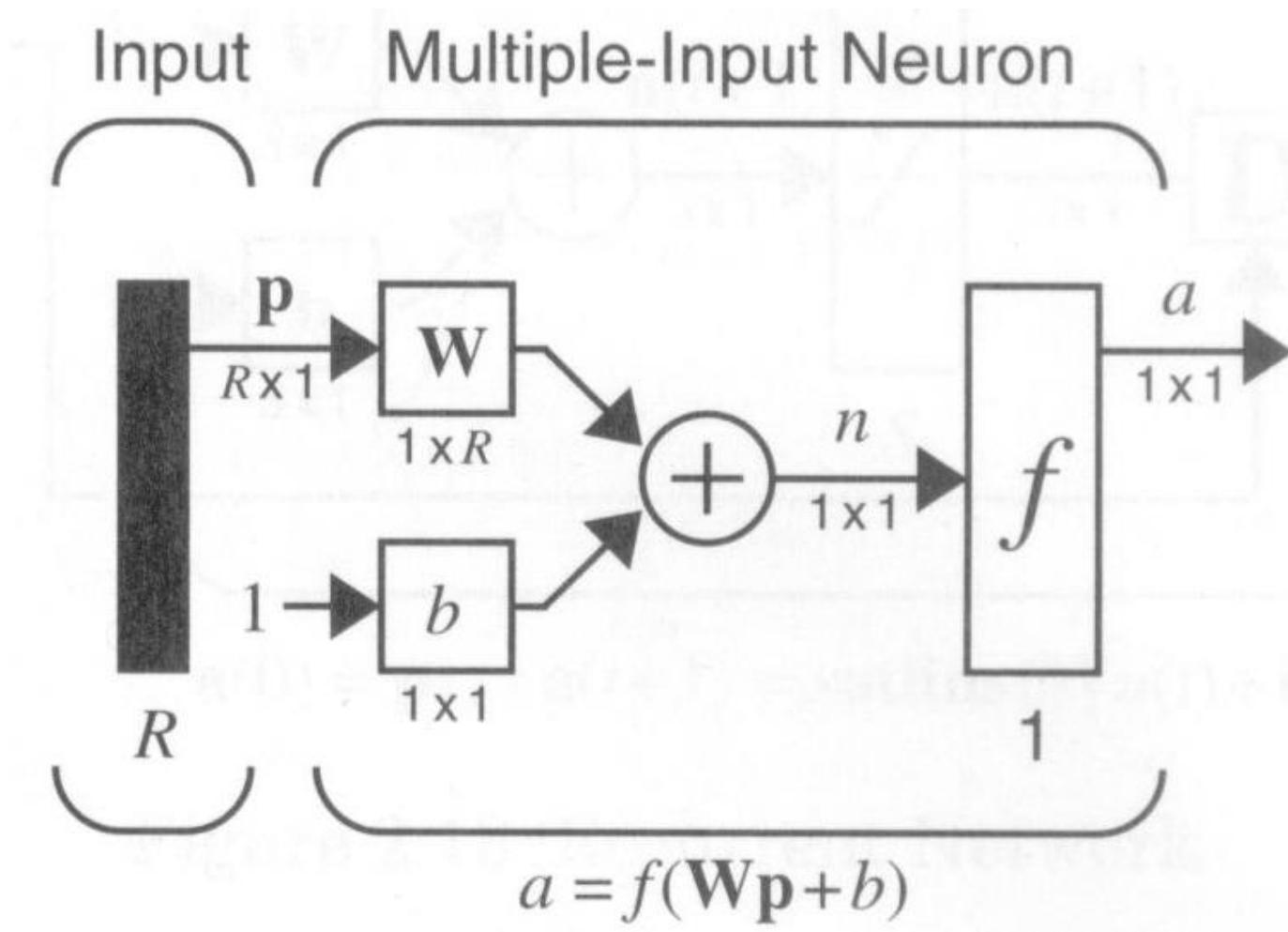
Multiple-Input Neuron

Inputs Multiple-Input Neuron



$$a = f(\mathbf{W}\mathbf{p} + b)$$

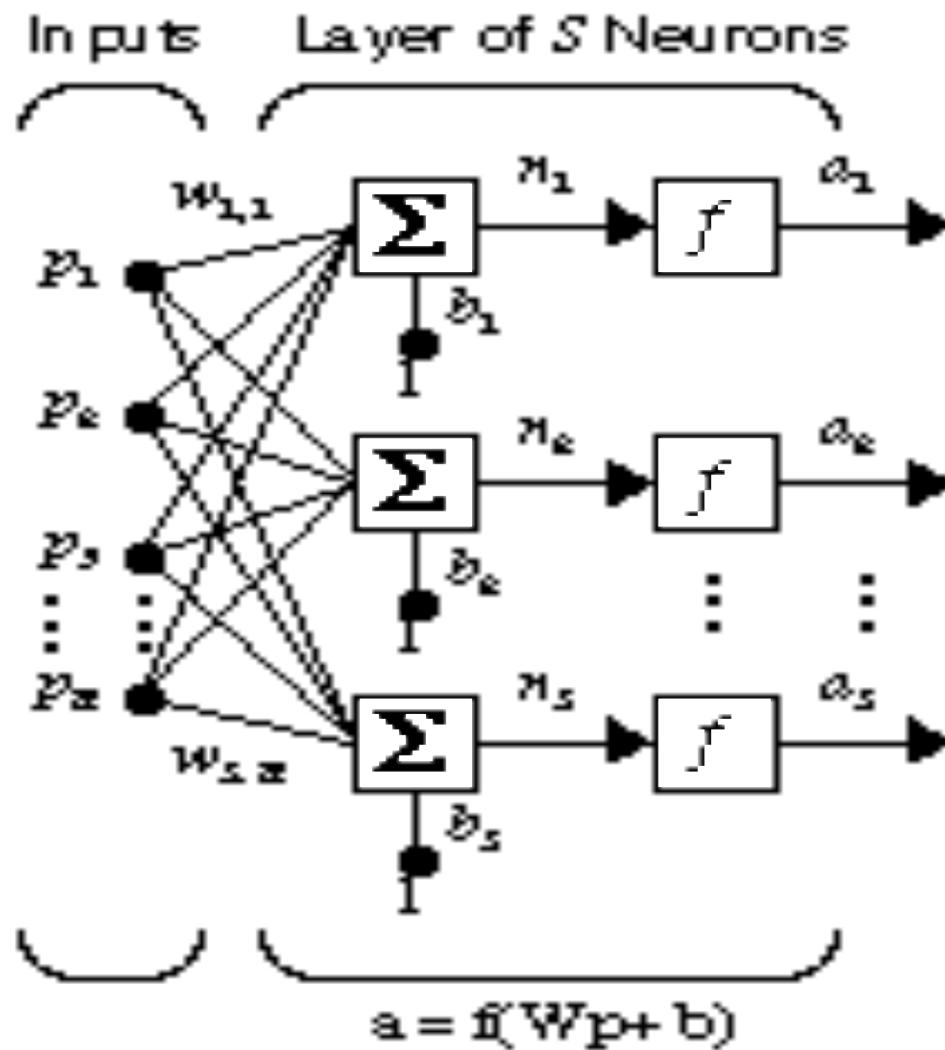
Multiple-Input Neuron – cont.



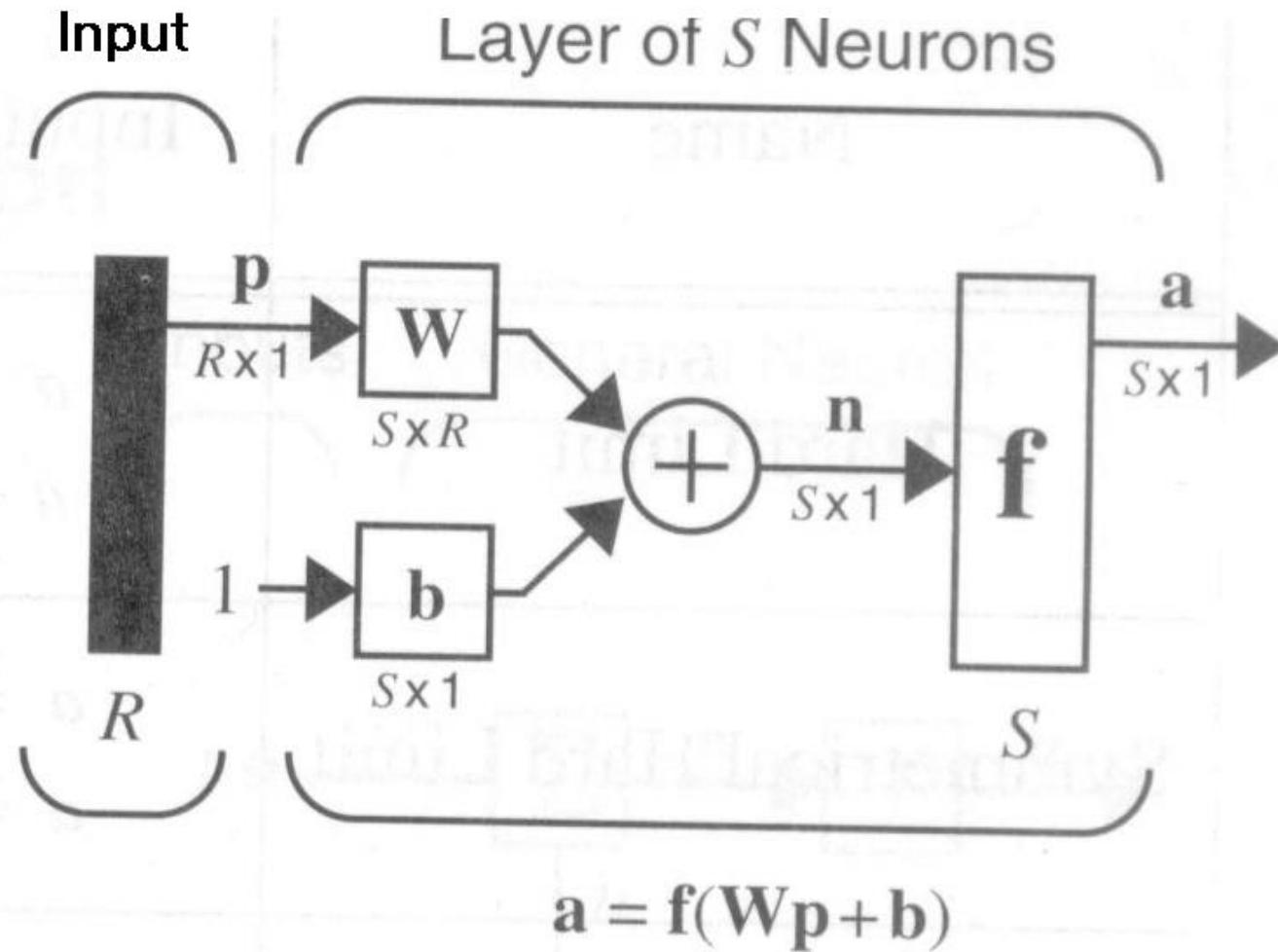
Neural Network Architecture

- A layer of neurons
 - Consists of a set of neurons
 - The inputs are connected to each of the neurons

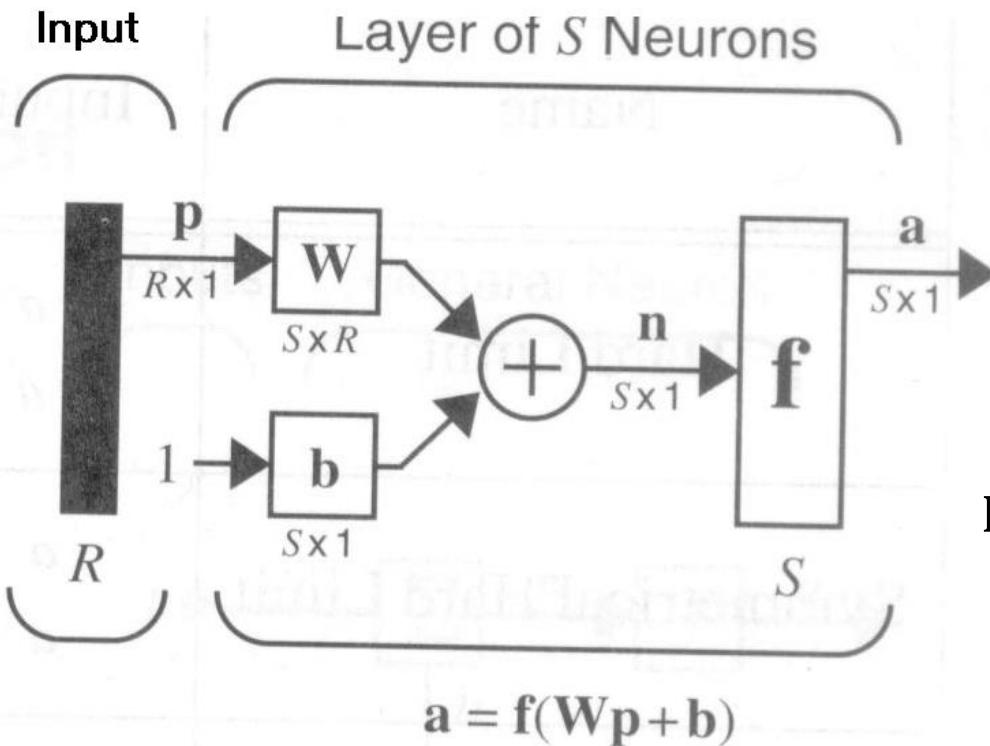
A Layer of Neurons



A Layer of Neurons

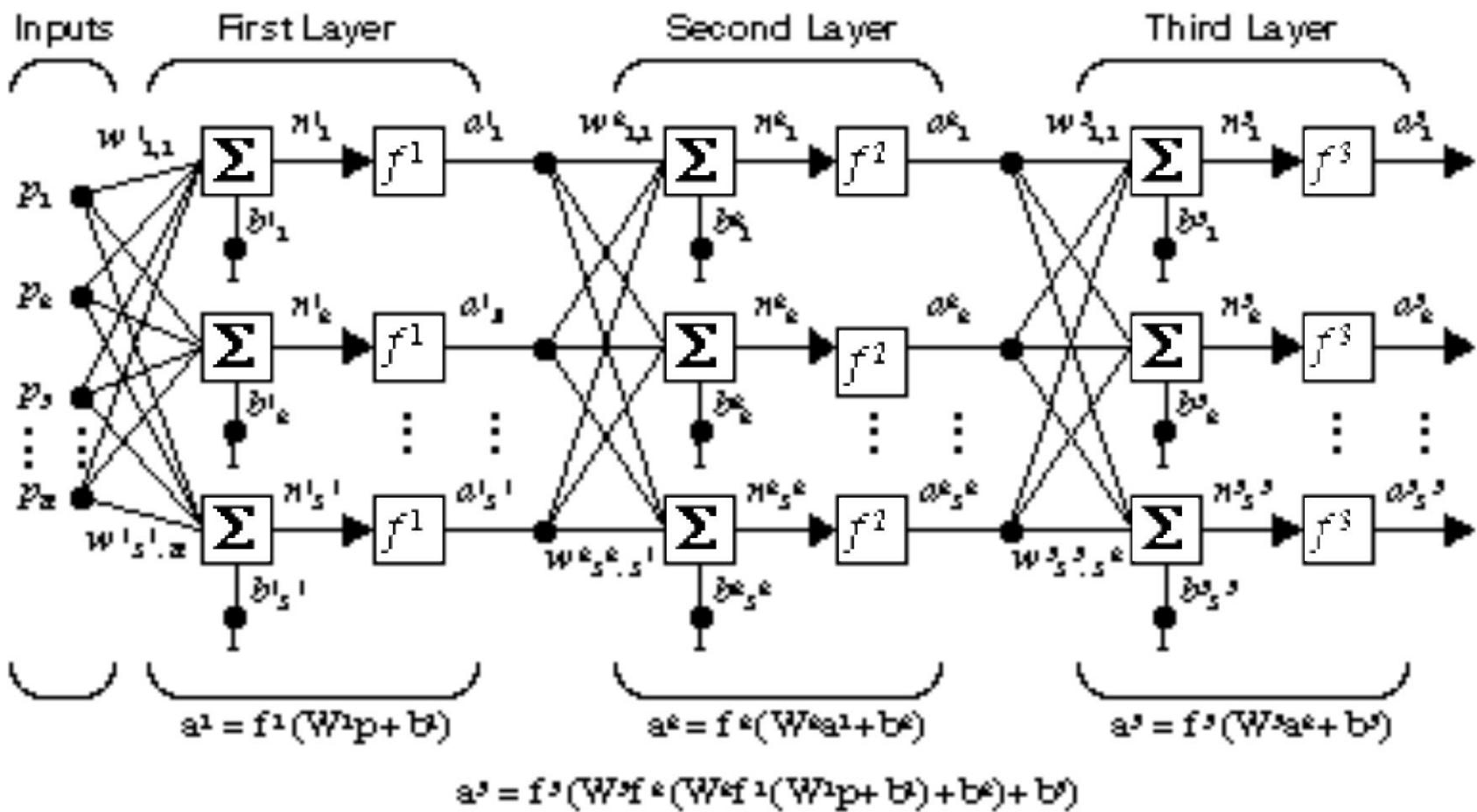


A Layer of Neurons

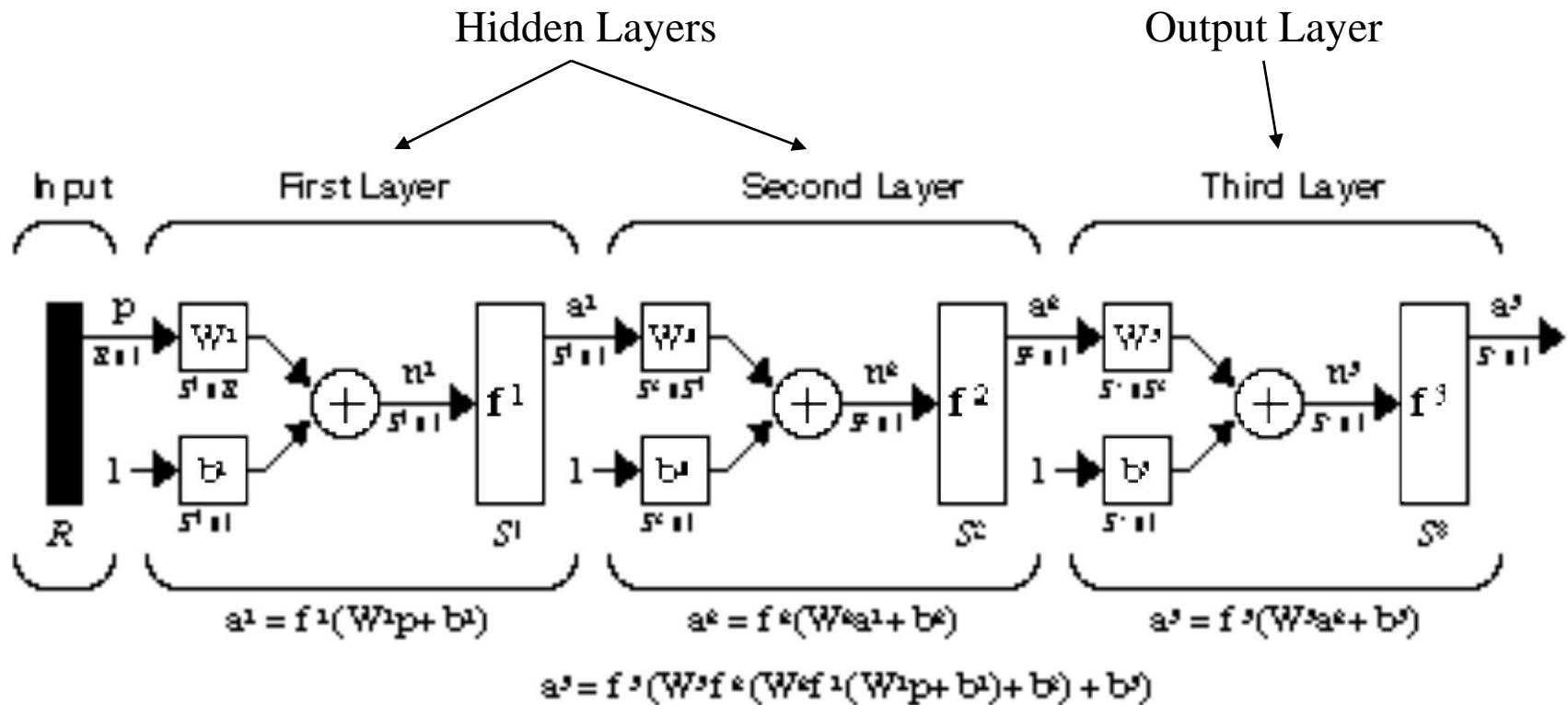


$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$
$$\mathbf{p} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_R \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_S \end{bmatrix} \quad \mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_S \end{bmatrix}$$

Multiple Layers of Neurons



Multiple Layers of Neurons

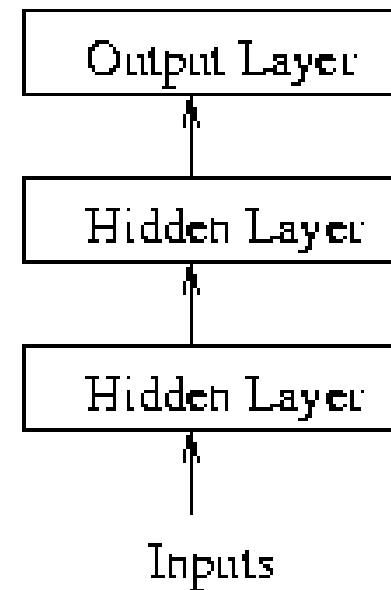
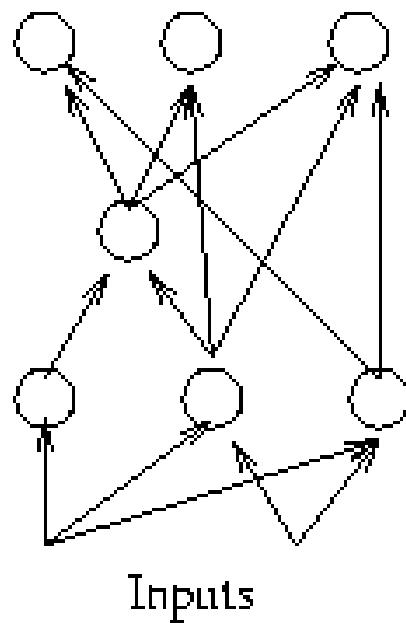


Multiple Layers of Neurons – cont.

- Multiple layer neural networks
 - A network with several layers
 - Multi-layer networks are more powerful than single-layer networks with nonlinear transfer functions

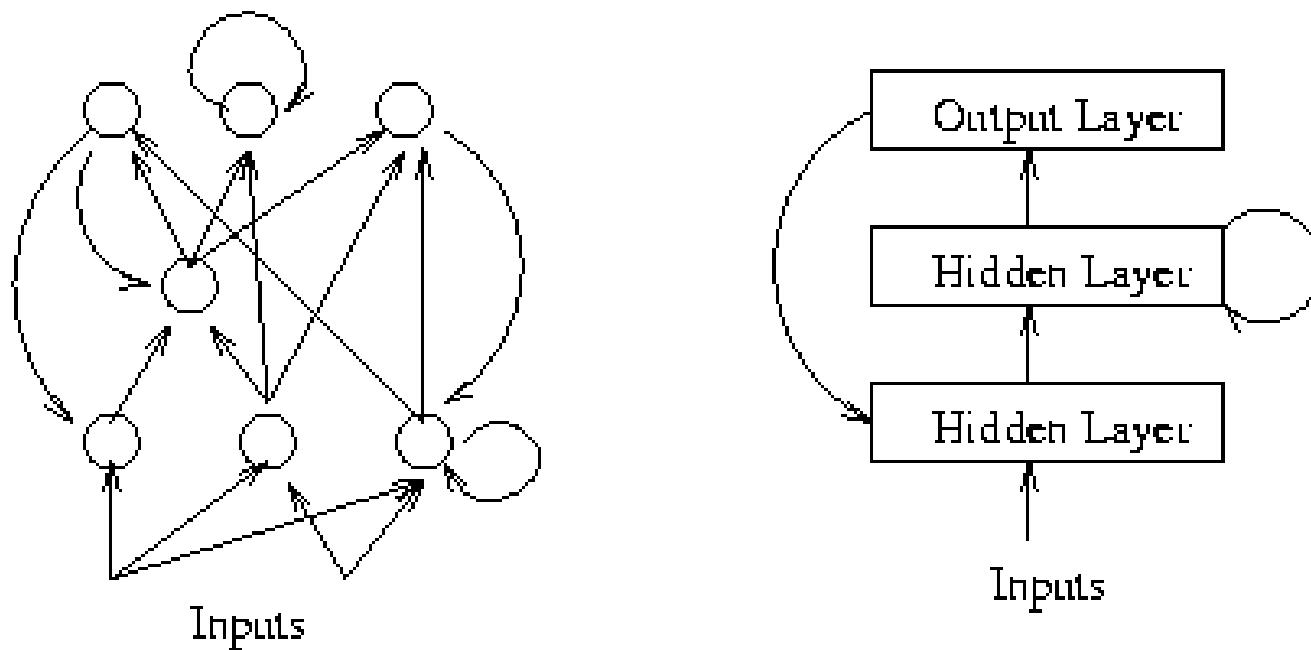
Neural Network Architecture – cont.

- Feed-forward networks



Neural Network Architecture – cont.

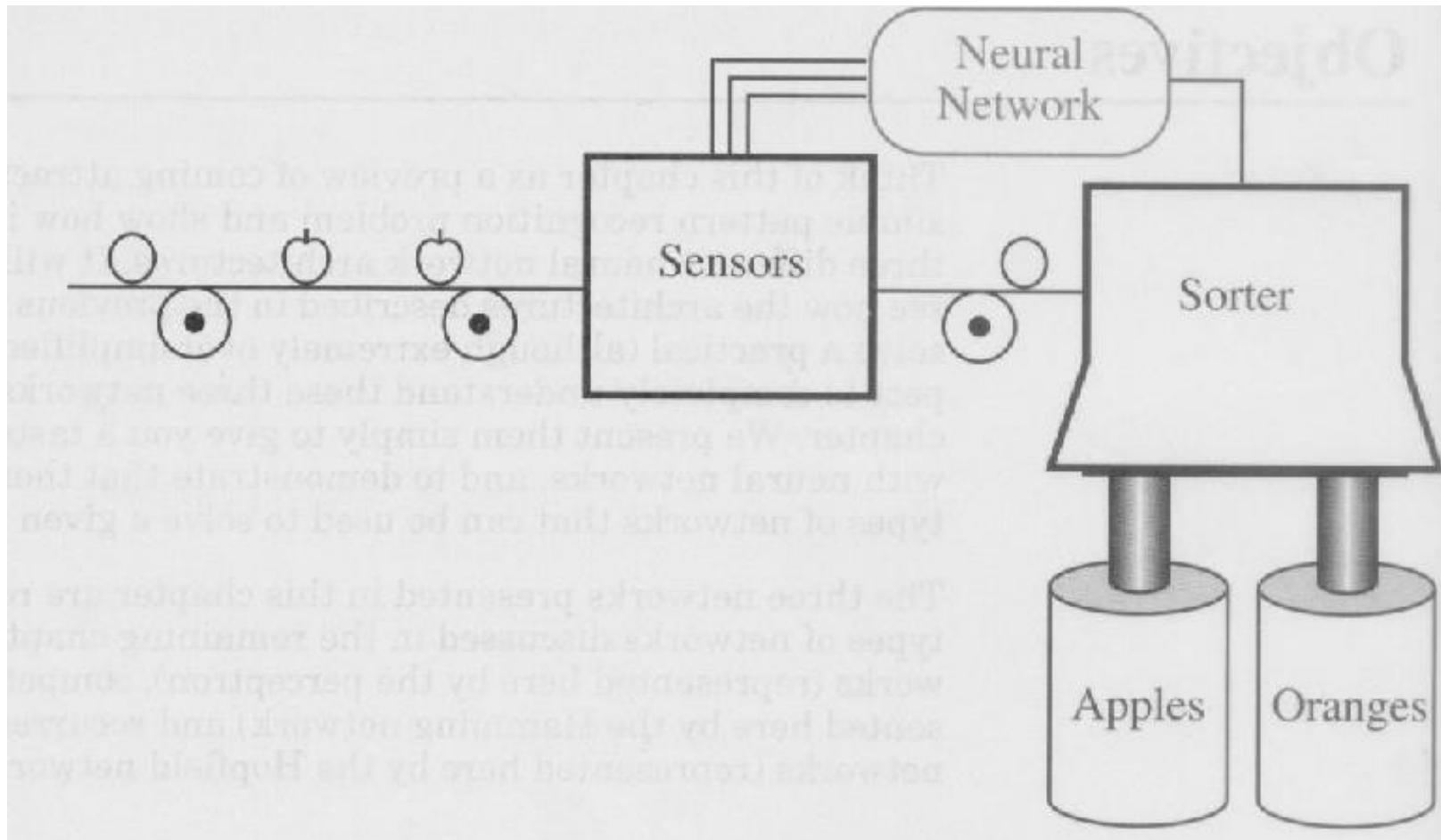
- Recurrent neural networks



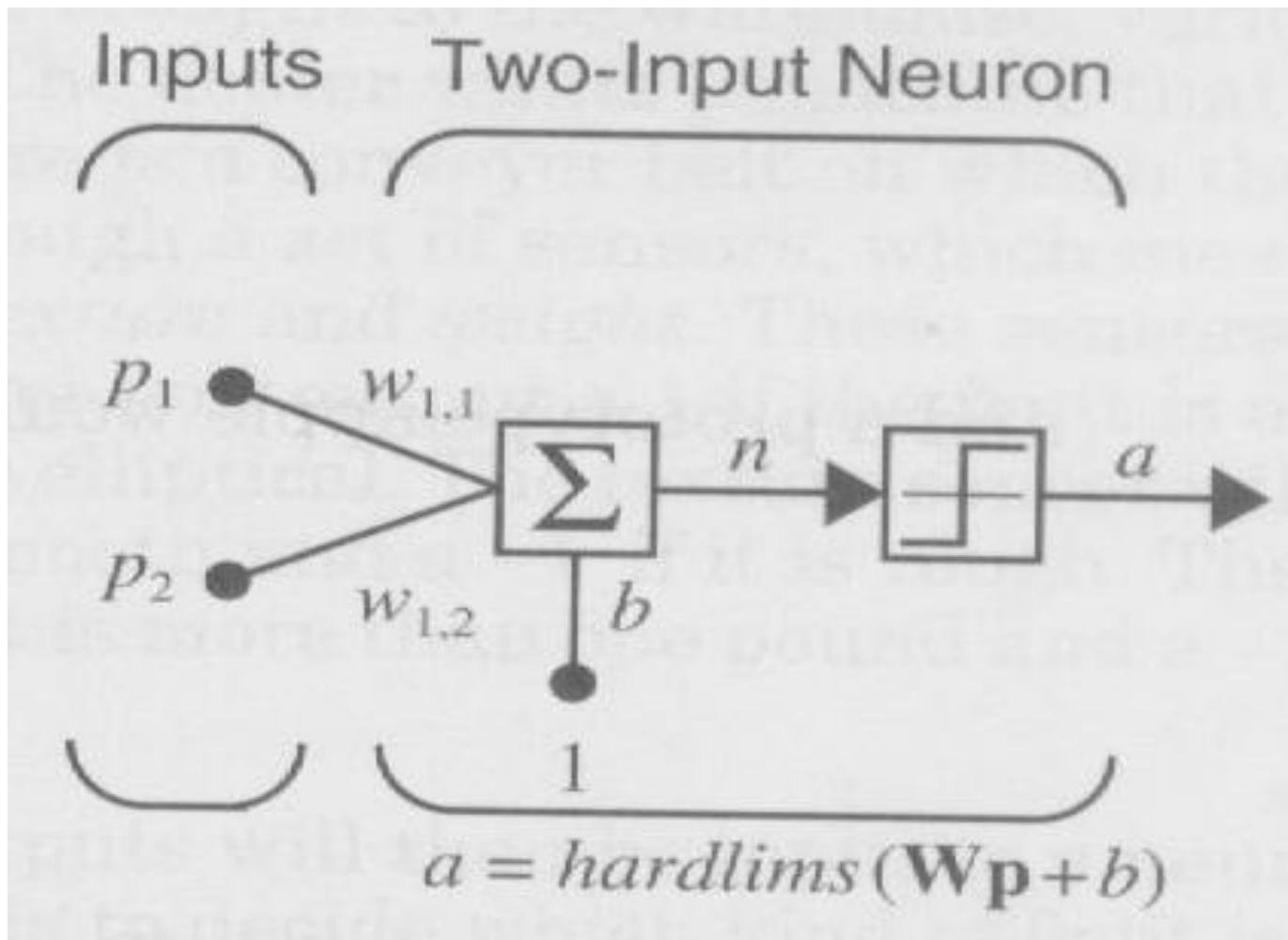
Network Architecture – cont.

- How to determine a network architecture
 - Number of network inputs is given by the number of problem inputs
 - Number of neurons in output layer is given by the number of problem outputs
 - The transfer function is partly determined by the output specifications

An Illustrative Example



Perceptron



Outline

- Perceptron Neural Network
- Perceptron Learning Rule
 - Perceptron network
 - Decision boundary

Using Neural Networks

- Choose a neural network architecture
 - Feed-forward network
 - Recurrent network
 - Each network has its own characteristics
 - Simple networks may not be able to solve a complex problem
 - For example, perceptron can only solve linearly separable problems

Using Neural Networks – cont.

- Specify the network architecture
 - How many layers
 - How many neurons at each layer
 - A specific connection pattern
- Choose a learning algorithm
 - Specify the parameters of the learning algorithm
 - Decide how to train the network

Using Neural Networks – cont.

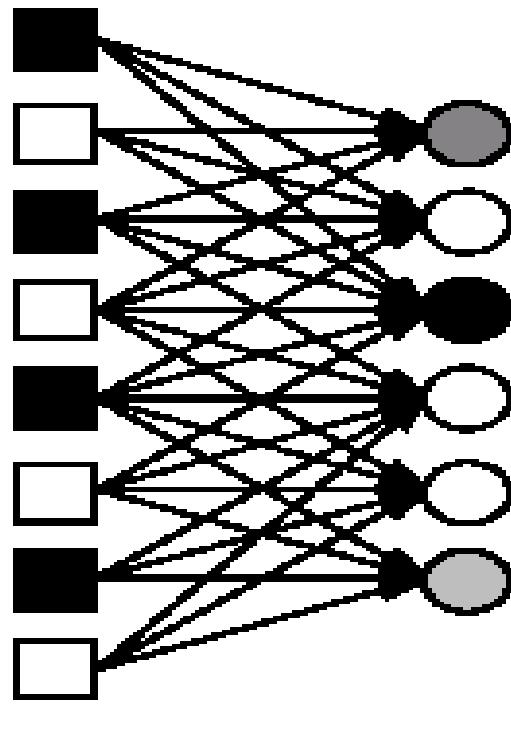
- Testing
 - A trained network will be normally tested on data that are not used during training
 - A network's ability to process outside training is called generalization
 - Generalization is critical for practical applications

The Perceptron

by Frank Rosenblatt (1958, 1962)

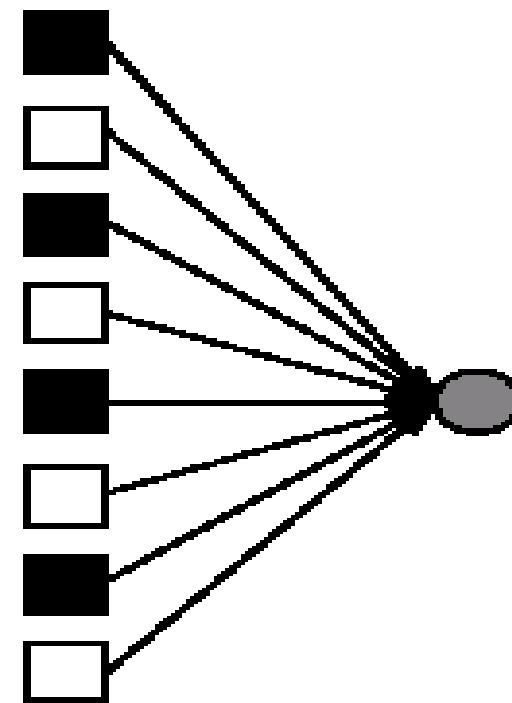
- Two-layers
- binary nodes (McCulloch-Pitts nodes) that take values 0 or 1
- continuous weights, initially chosen randomly
- Today, used as a synonym for a single-layer, feed-forward network

Perceptrons



I_j $W_{j,t}$ O_t
Input Units Output Units

Perceptron Network

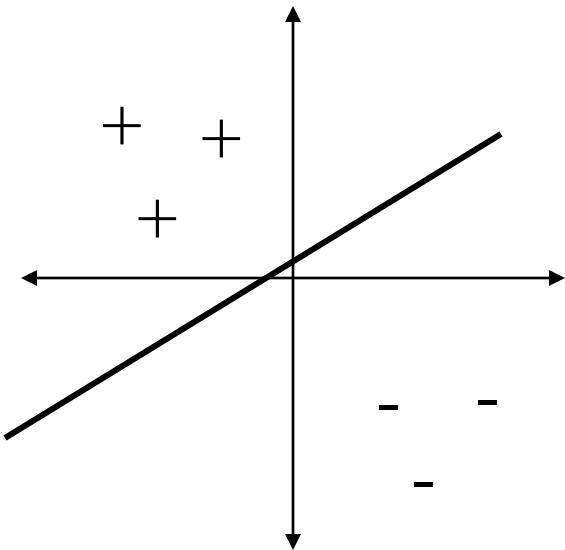


I_j W_j O
Input Units Output Unit

Single Perceptron

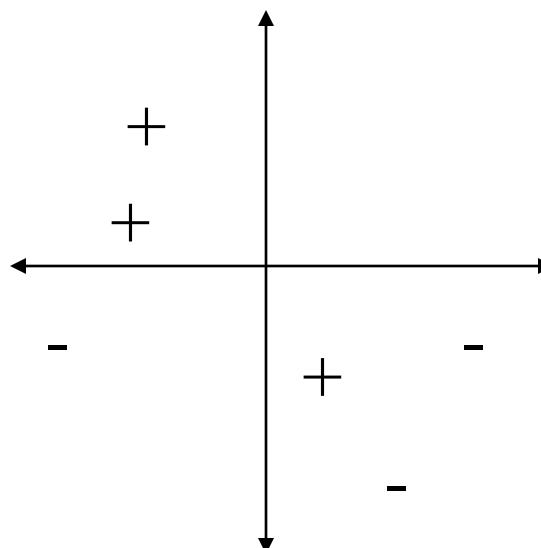
Hyperplane, again

A perceptron can learn only examples that are called “linearly separable”. These are examples that can be perfectly separated by a hyperplane.



Linearily separable
3/4/2024

EELU ITF309 Neural Network Lecture 2



Non-linearly separable

45

How does the perceptron learn its classification tasks?

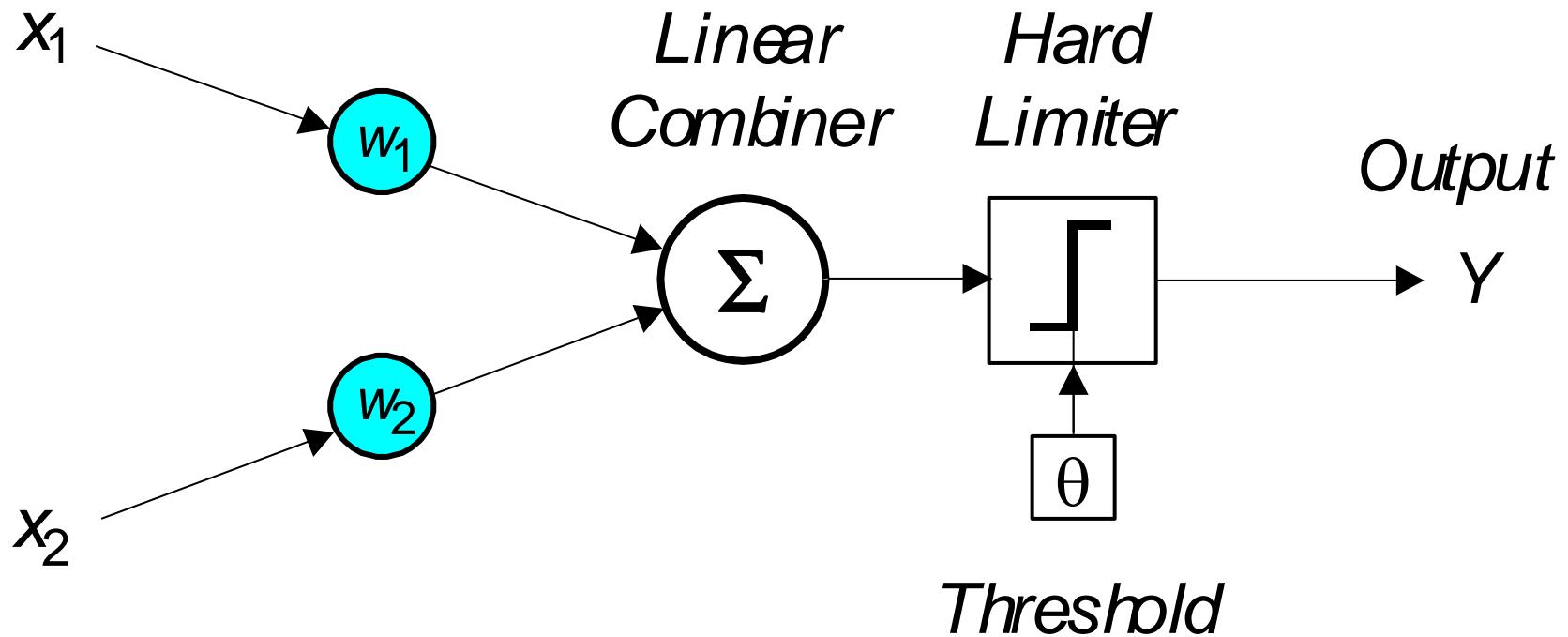
This is done by making small adjustments in the weights to reduce the difference between the actual and desired outputs of the perceptron. The initial weights are randomly assigned, usually in the range $[-0.5, 0.5]$, and then updated to obtain the output consistent with the training examples.

Can a single neuron learn a task?

- In 1958, Frank Rosenblatt introduced a training algorithm that provided the first procedure for training a simple ANN: a **perceptron**.
- The perceptron is the simplest form of a neural network. It consists of a single neuron with *adjustable* synaptic weights and a *hard limiter*.

Single-layer two-input perceptron

Inputs



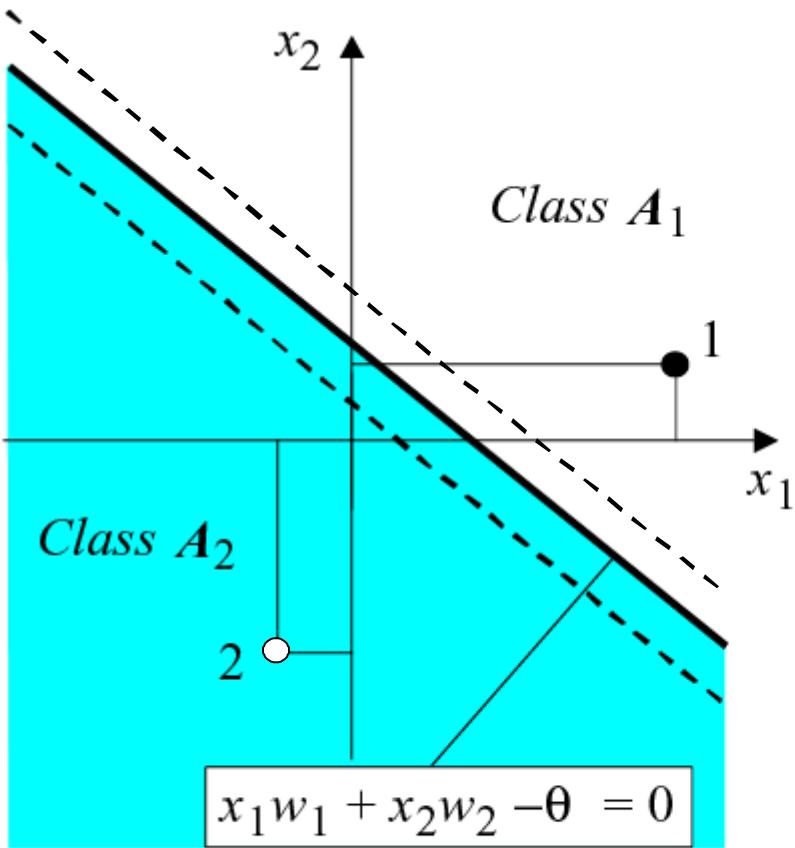
The Perceptron

- The operation of Rosenblatt's perceptron is based on the **McCulloch and Pitts neuron model**. The model consists of a linear combiner followed by a hard limiter.
- The weighted sum of the inputs is applied to the hard limiter, which produces an output equal to $+1$ if its input is positive and -1 if it is negative.

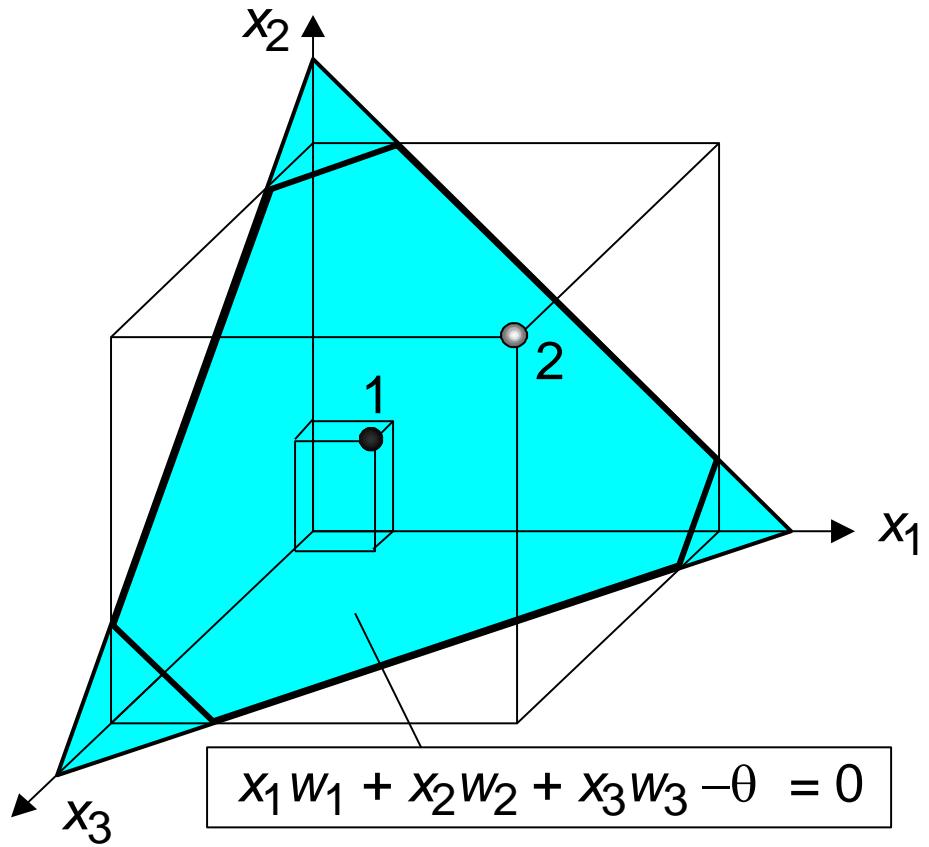
-
- The aim of the perceptron is to classify inputs, x_1, x_2, \dots, x_n , into one of two classes, say A_1 and A_2 .
 - In the case of an elementary perceptron, the n -dimensional space is divided by a *hyperplane* into two decision regions. The hyperplane is defined by the *linearly sep*

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

Linear separability in the perceptrons



(a) Two-input perceptron.



(b) Three-input perceptron.

How does the perceptron learn its classification tasks?

This is done by making small adjustments in the weights to reduce the difference between the actual and desired outputs of the perceptron. The initial weights are randomly assigned, usually in the range $[-0.5, 0.5]$, and then updated to obtain the output consistent with the training examples.

-
- If at iteration p , the actual output is $Y_d(p)$ and the perceptron output is $Y(p)$, then the error is given by:
$$e(p) = Y_d(p) - Y(p) \quad \text{where } p = 1, 2, 3, \dots$$

Iteration p here refers to the p th training example presented to the perceptron.

- If the error, $e(p)$, is positive, we need to increase perceptron output $Y(p)$, but if it is negative, we need to decrease $Y(p)$.

The perceptron learning rule

$$w_i(p+1) = w_i(p) + \alpha \cdot x_i(p) \cdot e(p)$$

where $p = 1, 2, 3, \dots$

α is the **learning rate**, a positive constant less than unity.

The perceptron learning rule was first proposed by **Rosenblatt** in 1960. Using this rule we can derive the perceptron training algorithm for classification tasks.

Perceptron's training algorithm

Step 1: Initialisation

Set initial weights w_1, w_2, \dots, w_n and threshold θ to random numbers in the range $[-0.5, 0.5]$.

If the error, $e(p)$, is positive, we need to increase perceptron output $Y(p)$, but if it is negative, we need to decrease $Y(p)$.

Perceptron's training algorithm (continued)

Step 2: Activation

Activate the perceptron by applying inputs $x_1(p), x_2(p), \dots, x_n(p)$ and desired output $Y_d(p)$.

Calculate the actual output at iteration $p = 1$

$$Y(p) = \text{step} \left[\sum_{i=1}^n x_i(p) w_i(p) - \theta \right]$$

where n is the number of the perceptron inputs,
and step is a step activation function.

Perceptron's training algorithm (continued)

Step 3: Weight training

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

Update the weights

where $\Delta w_i(p)$ is the weight correction at iteration p.

The weight correction is computed by the delta rule:

$$\Delta w_i(p) = \alpha \cdot x_i(p) \cdot \epsilon(p)$$

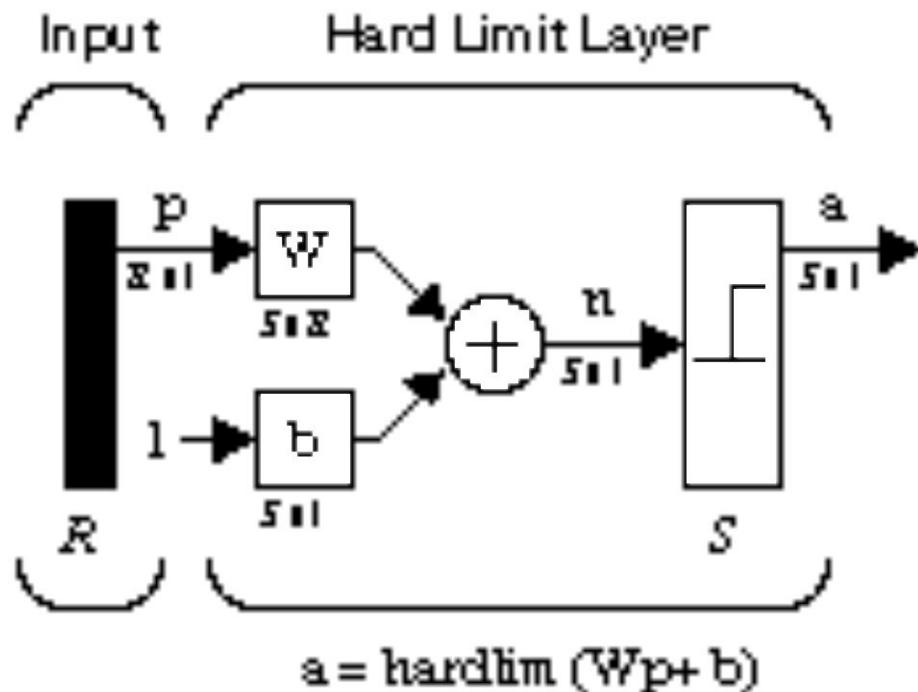
Step 4: Iteration

Increase iteration p by one, go back to Step 2 and repeat the process until convergence.

Perceptron Network

- A perceptron network is a single-layer neural network
 - With hardlim as the transfer function
 - Or hardlims, which does not affect the capabilities of the network
 - It in general can have more than one neuron
 - We can, however, determine the weights and bias of each neuron individually

Perceptron Architecture



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

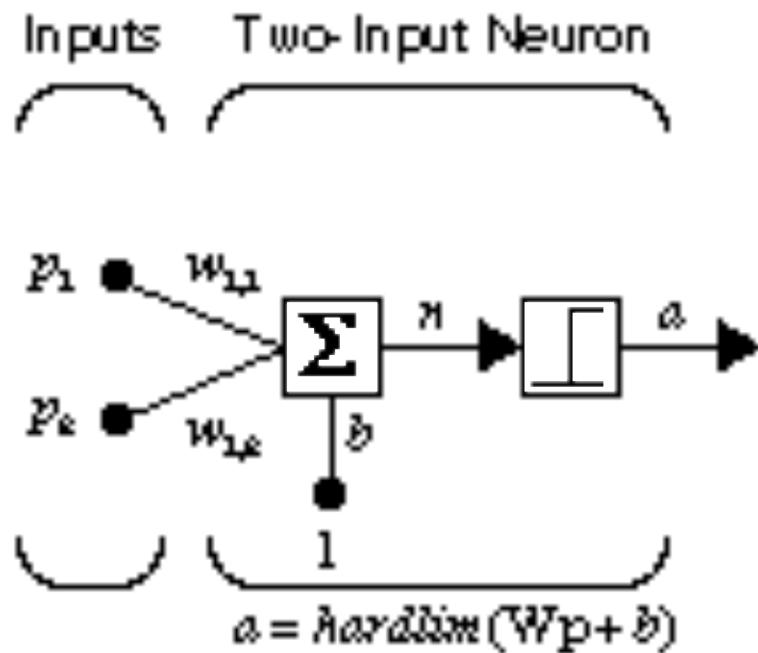
$$_i W = \begin{bmatrix} w_{i,1} \\ w_{i,2} \\ \vdots \\ w_{i,R} \end{bmatrix}$$

$$W = \begin{bmatrix} {}^T_1 W \\ {}^T_2 W \\ \vdots \\ {}^T_S W \end{bmatrix}$$

$$a_i = \text{hardlim}(n_i) = \text{hardlim}({}_i W^T p + b_i)$$

Single-Neuron Perceptron

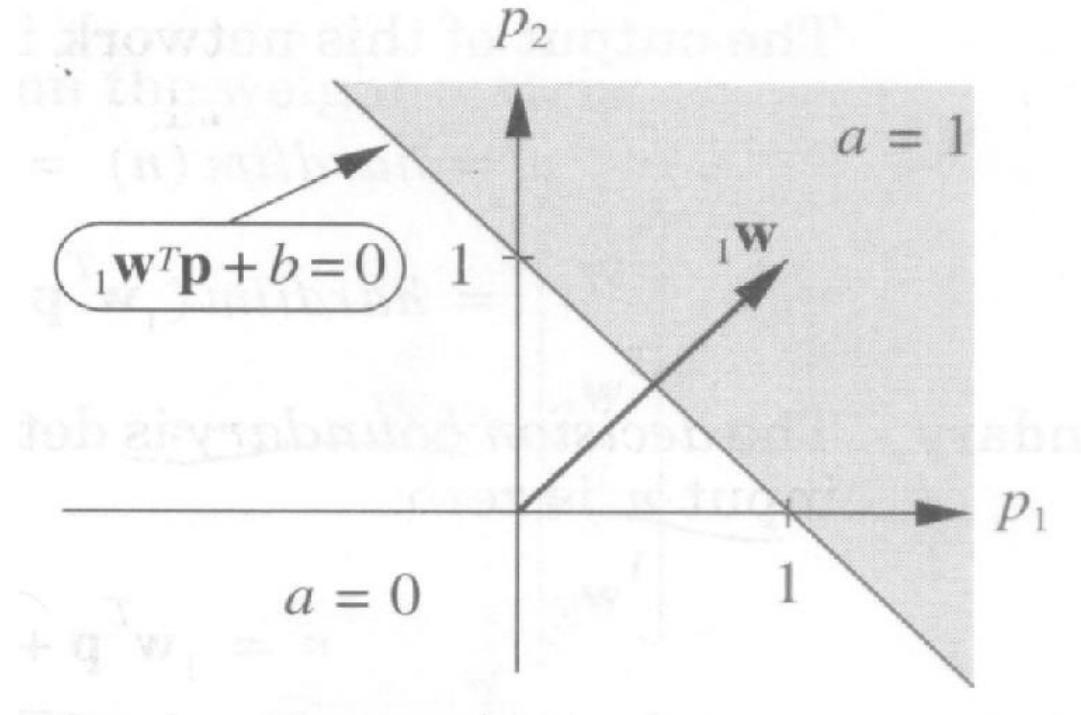
- The output of a single-neuron perceptron is given by $a = \text{hardlim}(w_{1,1}p_1 + w_{1,2}p_2 + b)$



Single-Neuron Perceptron – cont.

- Decision boundary

$$n = w_{1,1}p_1 + w_{1,2}p_2 + b = 0$$

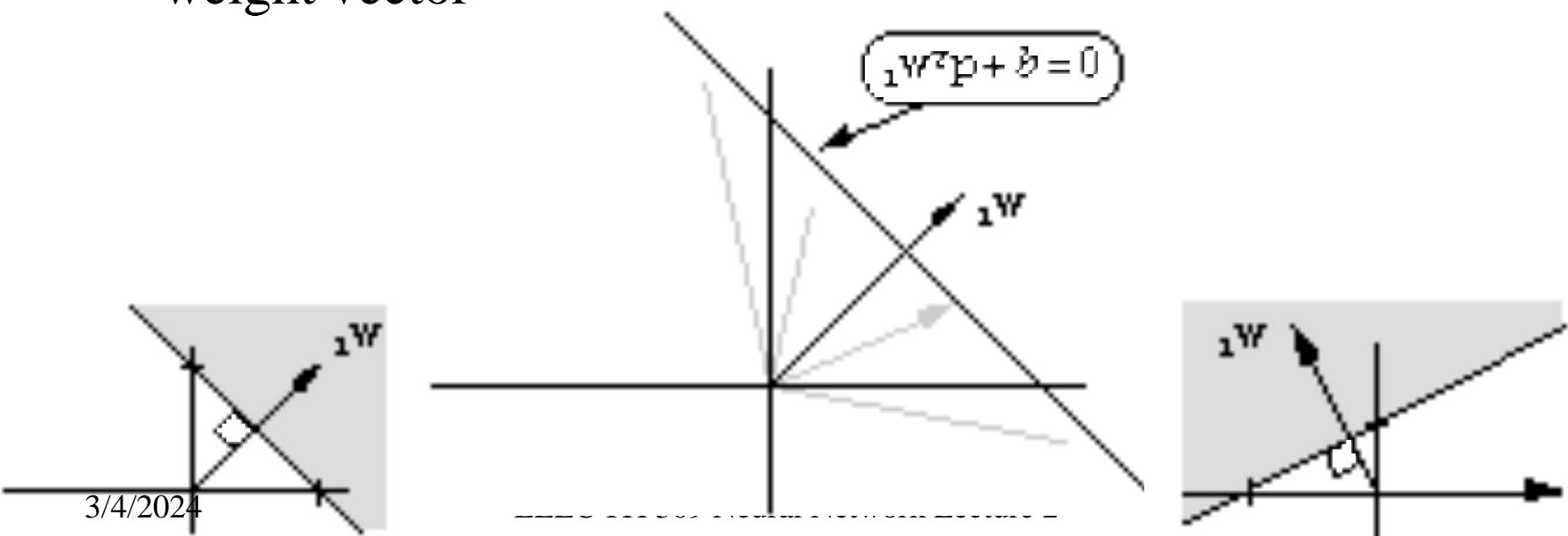


Decision Boundary

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

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

- 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

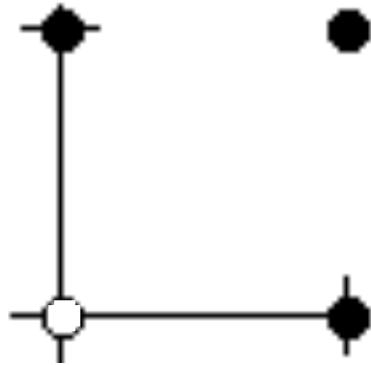


Single-Neuron Perceptron – cont.

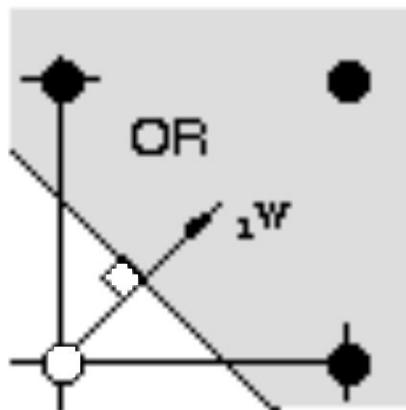
- Design a perceptron graphically
 - Select a decision boundary
 - Choose a weight vector that is orthogonal to the decision boundary
 - Find the bias b
 - When there are multiple decision boundaries, which one is the “best”?

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.

$$_1\mathbf{w} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

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

$$_1\mathbf{w}^T \mathbf{p} + b = \begin{bmatrix} 0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 0 \\ 0.5 \end{bmatrix} + b = 0.25 + b = 0 \quad \Rightarrow \quad b = -0.25$$

Multiple-Neuron Perceptron

- A decision boundary for each neuron
 - For neuron i the boundary is defined as

$$_i w^T p + b_i = 0$$

- A multiple-neuron perceptron can classify into many categories

Learning Rules

- A learning rule is a procedure for modifying the weights and biases of a neural network
 - It is often called a training algorithm
 - The purpose of the learning rule is to train the network to perform some task

Learning Rules – cont.

- Three categories of learning algorithms
 - Supervised learning
 - Training set $\{p_1, t_1\}, \{p_2, t_2\}, \dots, \{p_Q, t_Q\}$
 - Here t_q is called the target for input p_q
 - Reinforcement learning
 - No correct output is provided for each network input but a measure of the performance is provided
 - It is not very common
 - Unsupervised learning
 - There are no target outputs available

Perceptron Learning Rule

- Perceptron learning rule

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

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

Perceptron Learning Rule – cont.

- Proof of convergence
 - We need to show the learning rule will always converge to weights that accomplish the desired classification assuming that such weights exist
 - We need to show that we only need to update the weights finite number of times

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

Training Set

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

Initial Weights

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

First Iteration

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

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

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

Second Iteration

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

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

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

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

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

Check

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

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

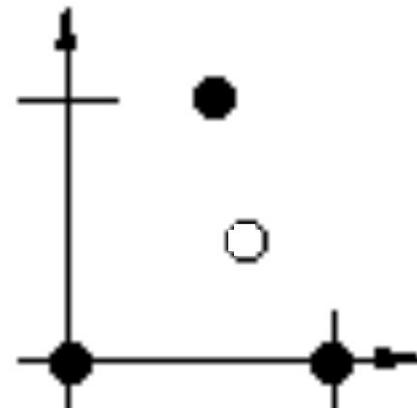
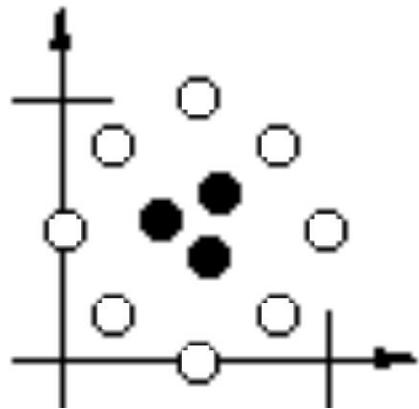
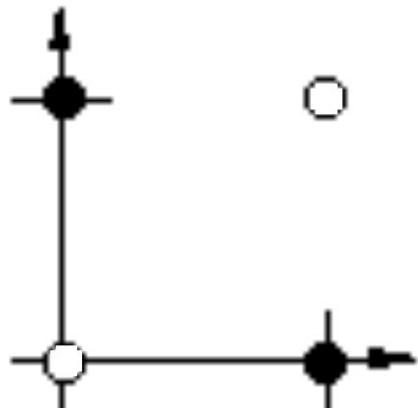
$$a = \text{hardlim}(\mathbf{W}\mathbf{p}_2 + b) = \text{hardlim}(\begin{bmatrix} -1.5 \\ -1 \\ -0.5 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + 0.5)$$

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

Limitations of Perceptrons

- Linear separability
 - Perceptron networks can only solve problems that are linearly separable, which means the decision boundary is linear
- Many problems that are not linear separable
 - XOR is a well-known example

Linearly Inseparable Problems



Limitations of Perceptrons – cont.

- Possible solutions
 - Have a more complex network architecture
 - Have several layers instead of one-layer
 - How to train the network then?
 - Project the data into a high dimensional feature space
 - Called kernel representation
 - The resulting network is called support vector machines, which are widely used for real-world applications

Deep Learning Overview

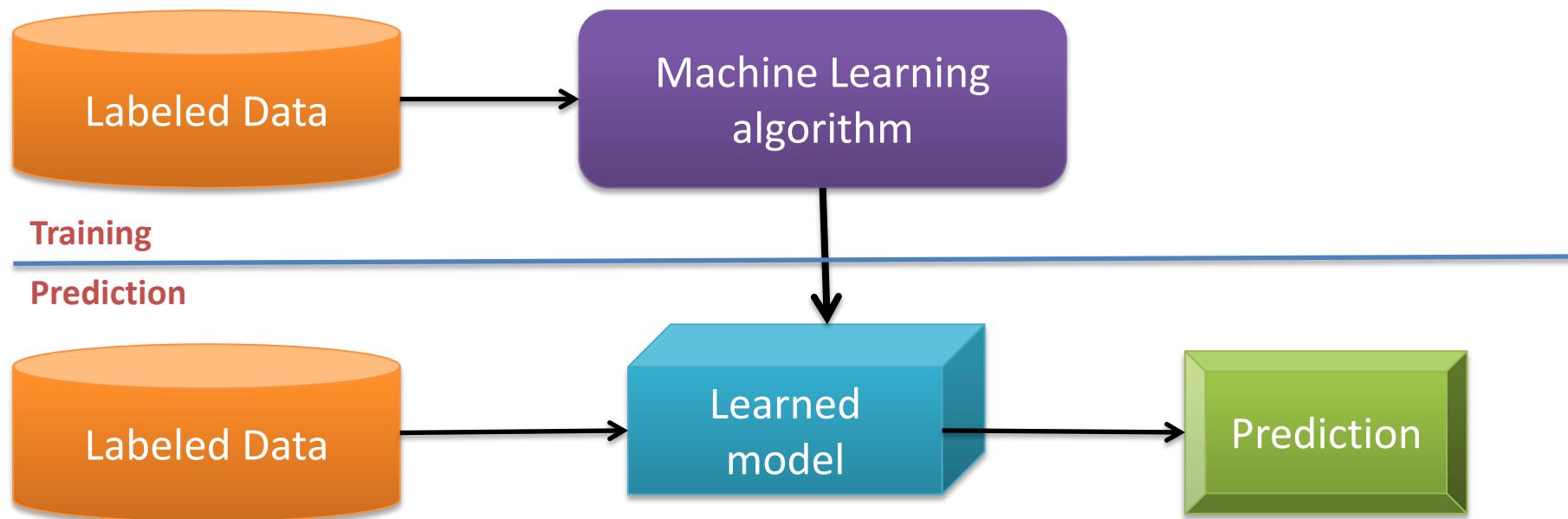
Lecture Outline

- Introduction to deep learning
- Elements of neural networks (NNs)
 - Activation functions
- Training NNs
 - Gradient descent
 - Regularization methods
- NN architectures
 - Convolutional NNs
 - Recurrent NNs

Machine Learning Basics

Machine Learning Basics

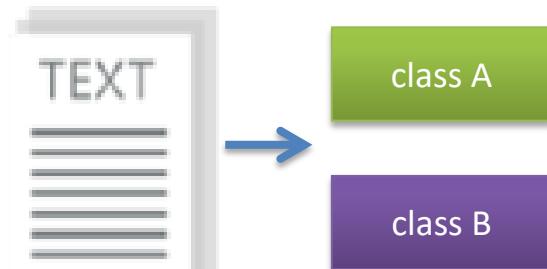
- **Artificial Intelligence** is a scientific field concerned with the development of algorithms that allow computers to learn without being explicitly programmed
- **Machine Learning** is a branch of Artificial Intelligence, which focuses on methods that learn from data and make predictions on unseen data



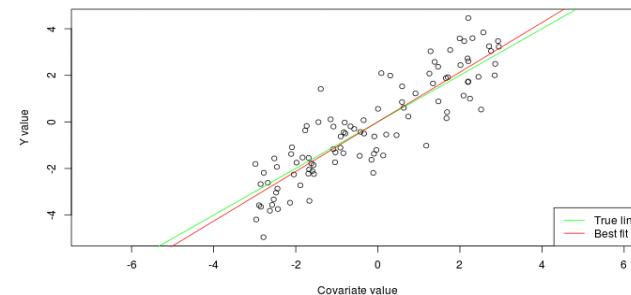
Machine Learning Types

Machine Learning Basics

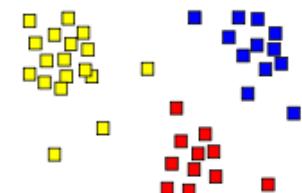
- **Supervised**: learning with **labeled data**
 - Example: email classification, image classification
 - Example: regression for predicting real-valued outputs
- **Unsupervised**: discover patterns in **unlabeled data**
 - Example: cluster similar data points
- **Reinforcement learning**: learn to act based on **feedback/reward**
 - Example: learn to play Go



Classification



Regression



Clustering

Supervised Learning

Machine Learning Basics

- *Supervised learning* categories and techniques
 - Numerical classifier functions
 - Linear classifier, perceptron, logistic regression, support vector machines (SVM), neural networks
 - Parametric (probabilistic) functions
 - Naïve Bayes, Gaussian discriminant analysis (GDA), hidden Markov models (HMM), probabilistic graphical models
 - Non-parametric (instance-based) functions
 - k -nearest neighbors, kernel regression, kernel density estimation, local regression
 - Symbolic functions
 - Decision trees, classification and regression trees (CART)
 - Aggregation (ensemble) learning
 - Bagging, boosting (Adaboost), random forest

Unsupervised Learning

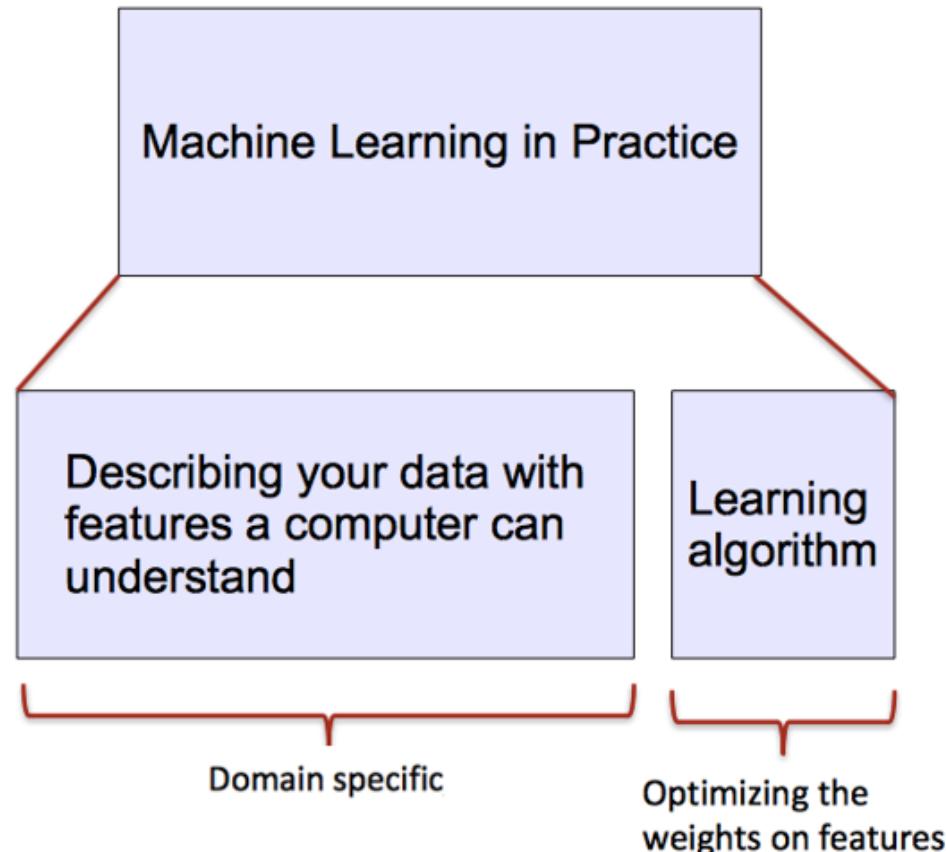
Machine Learning Basics

- ***Unsupervised learning*** categories and techniques
 - **Clustering**
 - k -means clustering
 - Mean-shift clustering
 - Spectral clustering
 - **Density estimation**
 - Gaussian mixture model (GMM)
 - Graphical models
 - **Dimensionality reduction**
 - Principal component analysis (PCA)
 - Factor analysis

ML vs. Deep Learning

Introduction to Deep Learning

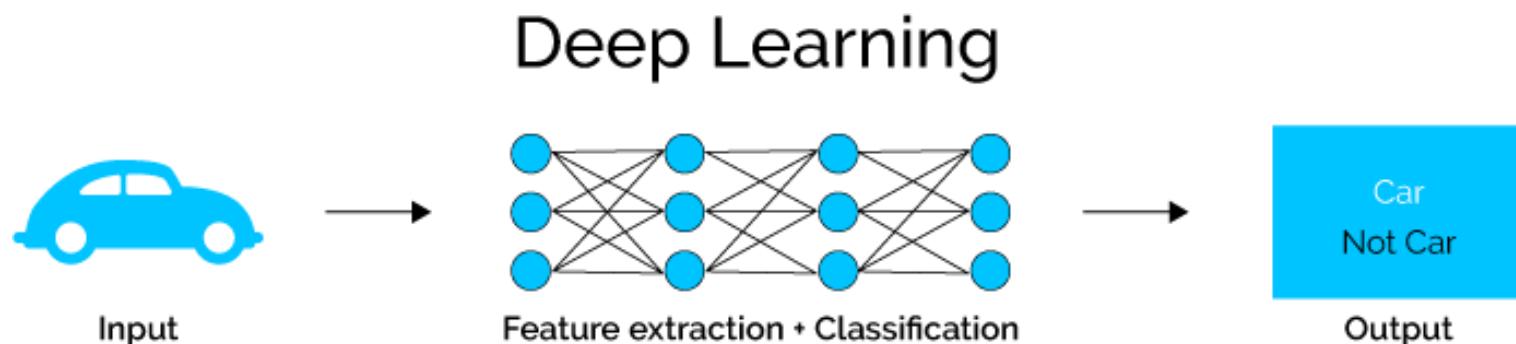
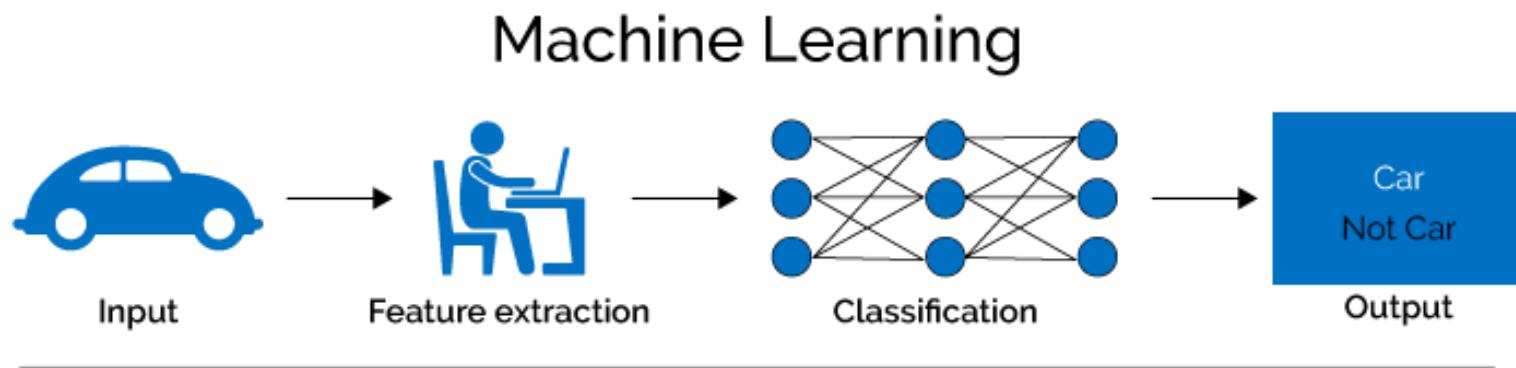
- Conventional machine learning methods rely on **human-designed feature representations**
 - ML becomes just optimizing weights to best make a final prediction



ML vs. Deep Learning

Introduction to Deep Learning

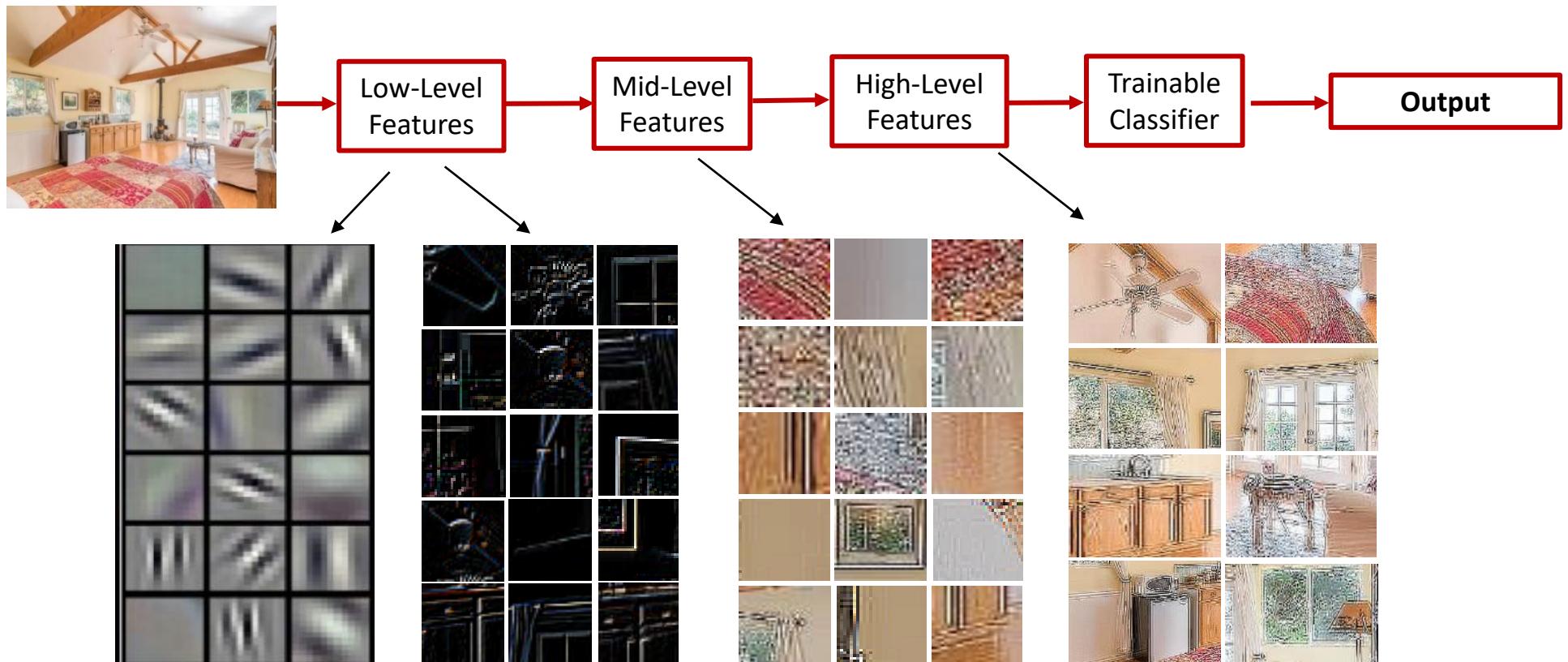
- **Deep learning** (DL) is a machine learning subfield that uses multiple layers for learning data representations
 - DL is exceptionally effective at learning patterns



ML vs. Deep Learning

Introduction to Deep Learning

- DL applies a multi-layer process for learning rich hierarchical features (i.e., data representations)
 - Input image pixels → Edges → Textures → Parts → Objects



Why is DL Useful?

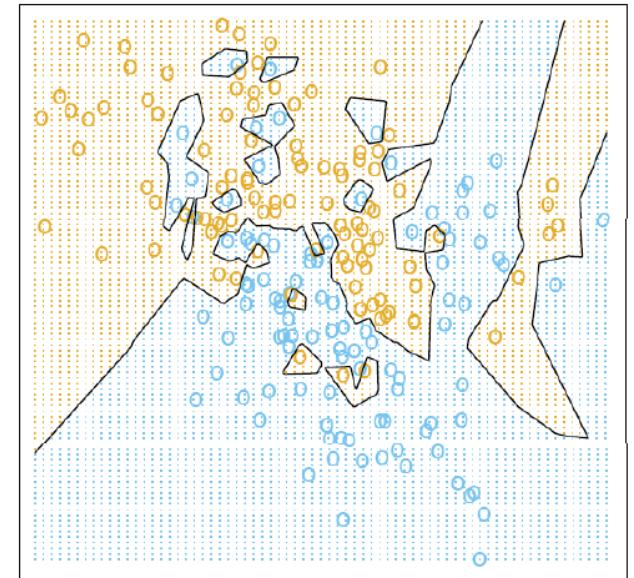
Introduction to Deep Learning

- DL provides a flexible, learnable framework for representing visual, text, linguistic information
 - Can learn in supervised and unsupervised manner
- DL represents an effective end-to-end learning system
- Requires large amounts of training data
- Since about 2010, DL has outperformed other ML techniques
 - First in vision and speech, then NLP, and other applications

Representational Power

Introduction to Deep Learning

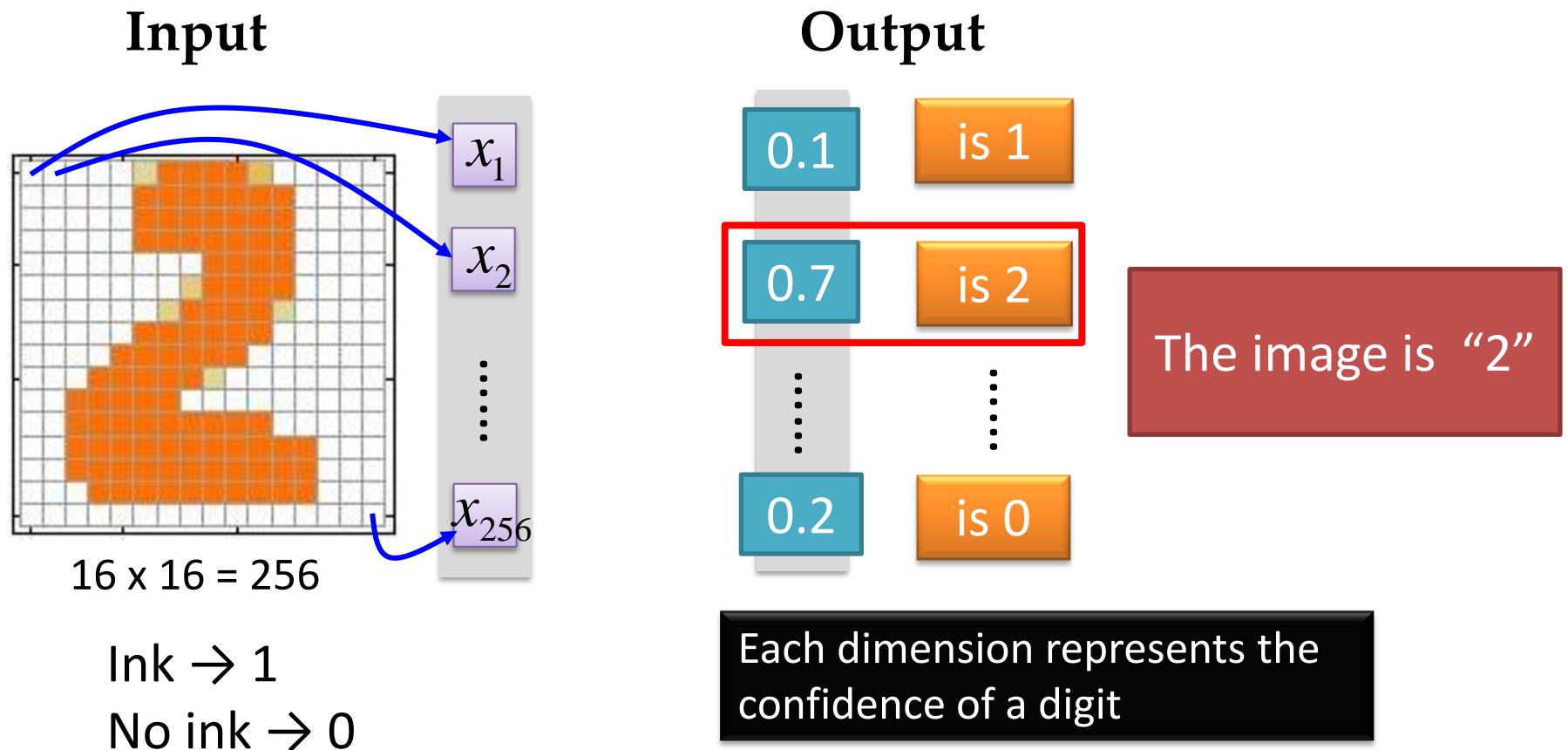
- NNs with at least one hidden layer are **universal approximators**
 - Given any continuous function $h(x)$ and some $\epsilon > 0$, there exists a NN with one hidden layer (and with a reasonable choice of non-linearity) described with the function $f(x)$, such that $\forall x, |h(x) - f(x)| < \epsilon$
 - I.e., NN can approximate any arbitrary complex continuous function
- NNs use nonlinear mapping of the inputs x to the outputs $f(x)$ to compute complex decision boundaries
- But then, why use deeper NNs?
 - The fact that deep NNs work better is an empirical observation
 - Mathematically, deep NNs have the same representational power as a one-layer NN



Introduction to Neural Networks

Introduction to Neural Networks

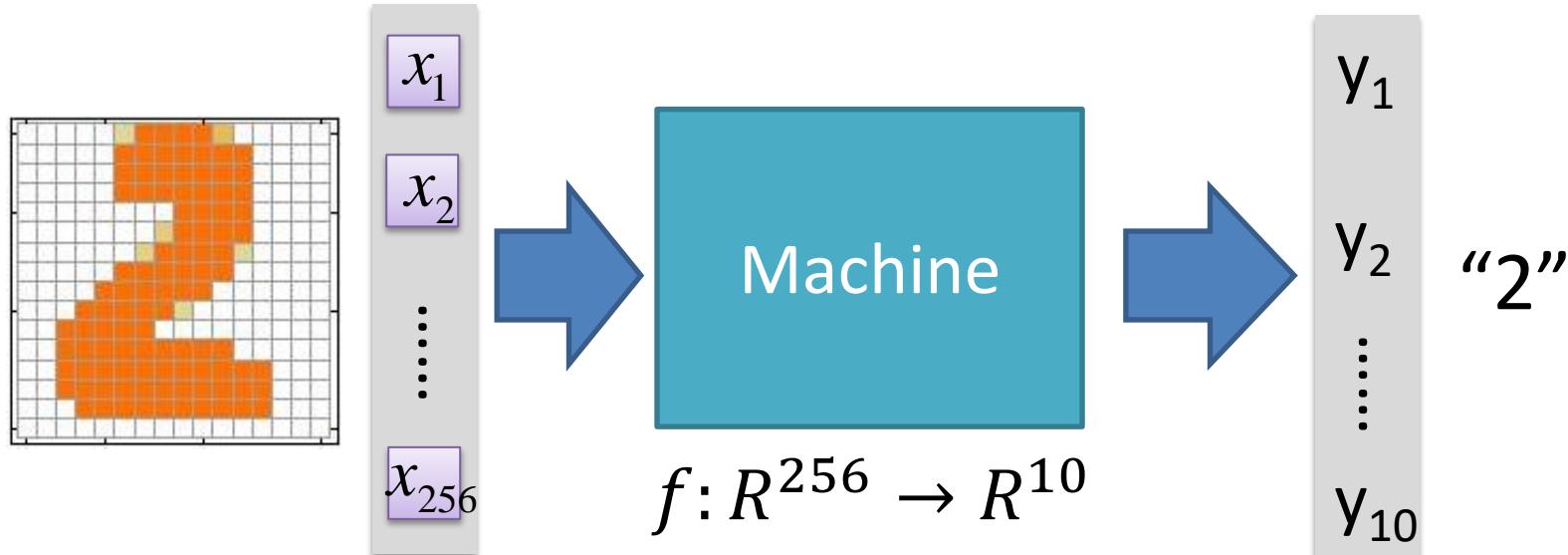
- Handwritten digit recognition (**MNIST dataset**)
 - The intensity of each pixel is considered an **input** element
 - **Output** is the class of the digit



Introduction to Neural Networks

Introduction to Neural Networks

- Handwritten digit recognition

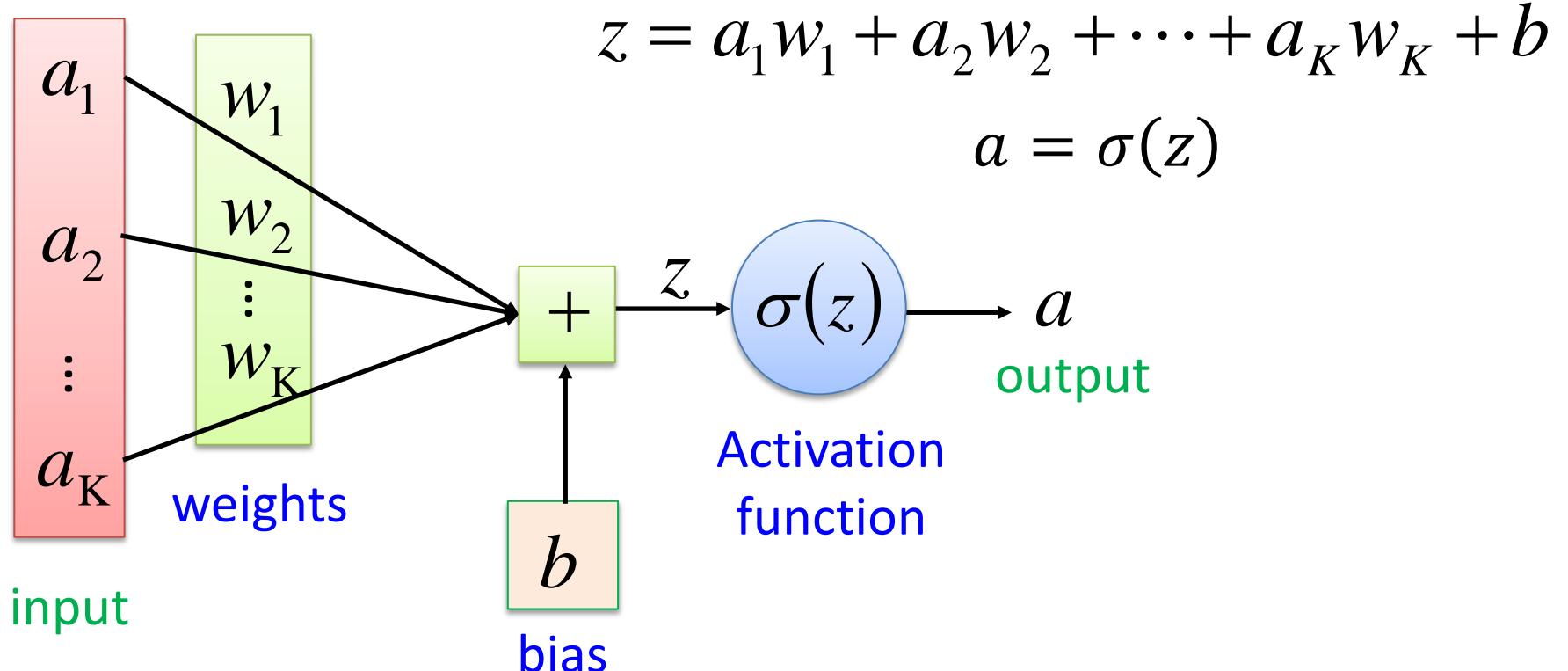


The function f is represented by a neural network

Elements of Neural Networks

Introduction to Neural Networks

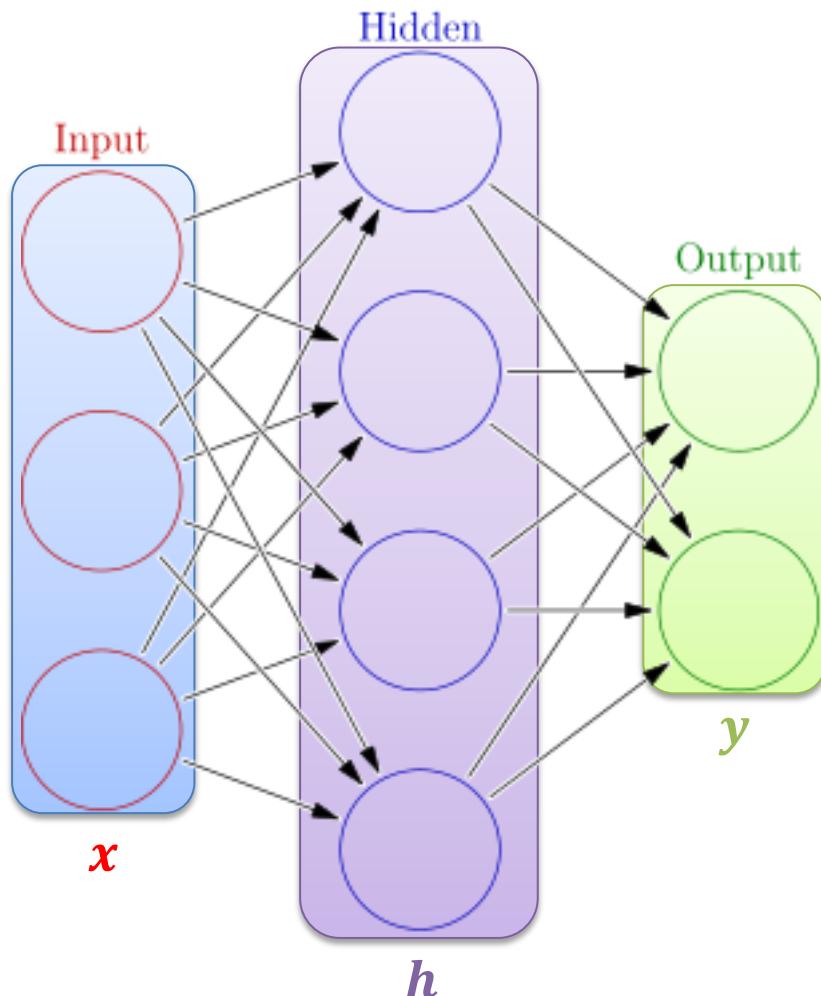
- NNs consist of hidden layers with neurons (i.e., computational units)
- A single **neuron** maps a set of inputs into an output number, or $f: R^K \rightarrow R$



Elements of Neural Networks

Introduction to Neural Networks

- A NN with one hidden layer and one output layer



Weights Biases

hidden layer $h = \sigma(W_1x + b_1)$

output layer $y = \sigma(W_2h + b_2)$

Activation functions

$4 + 2 = 6$ neurons (not counting inputs)

$[3 \times 4] + [4 \times 2] = 20$ weights

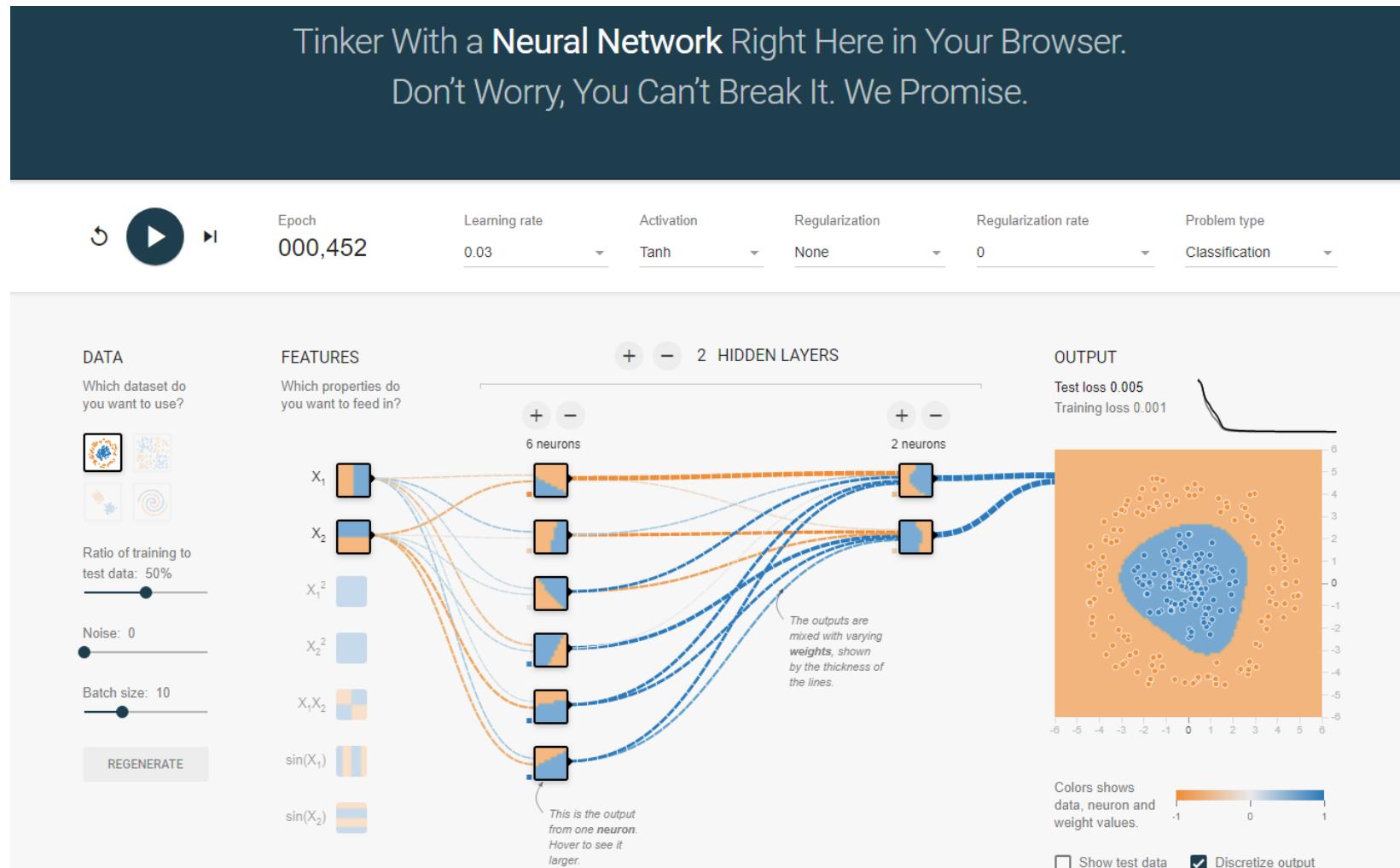
$4 + 2 = 6$ biases

26 learnable parameters

Elements of Neural Networks

Introduction to Neural Networks

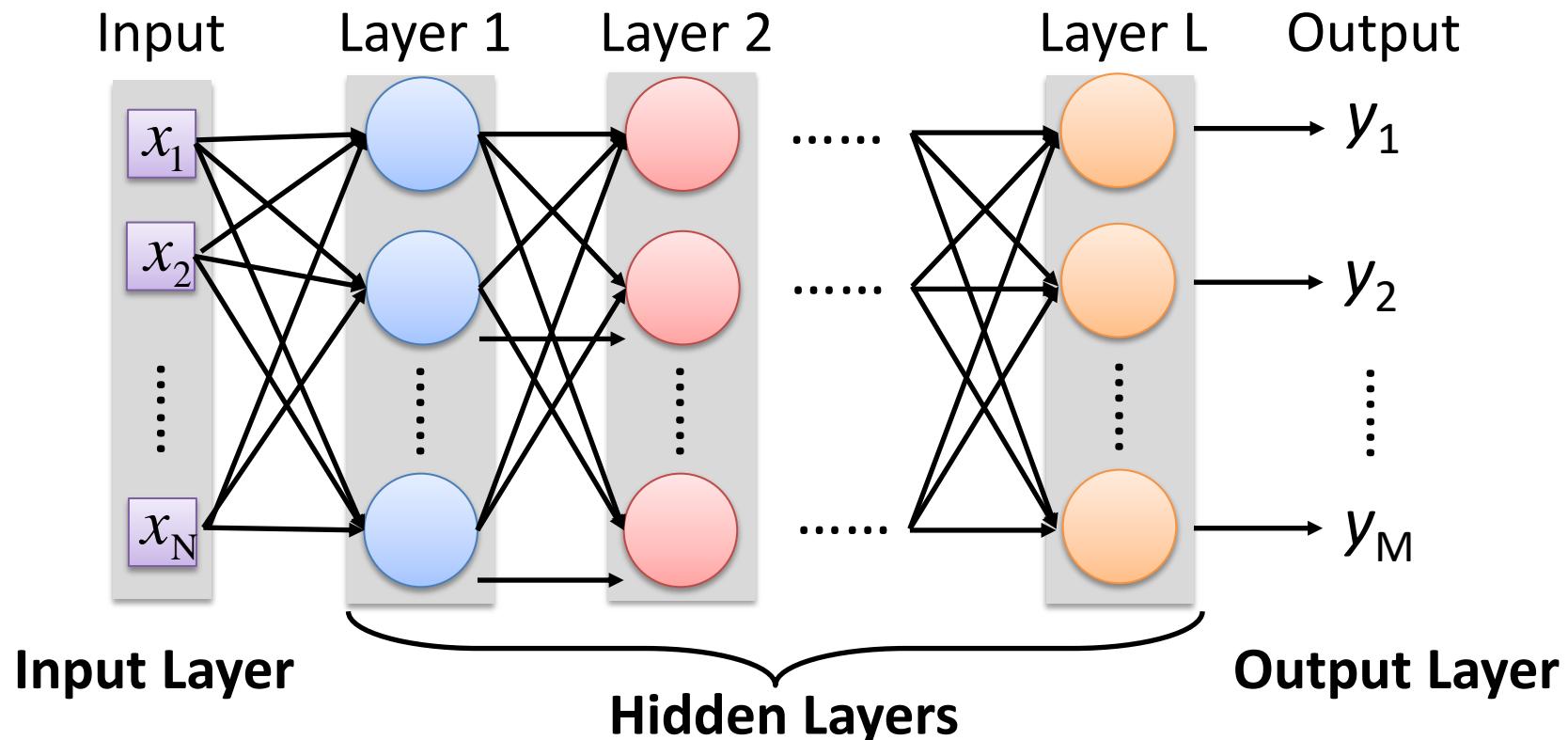
- A neural network playground [link](#)



Elements of Neural Networks

Introduction to Neural Networks

- Deep NNs have many hidden layers
 - Fully-connected (dense) layers (a.k.a. Multi-Layer Perceptron or MLP)
 - Each neuron is connected to all neurons in the succeeding layer

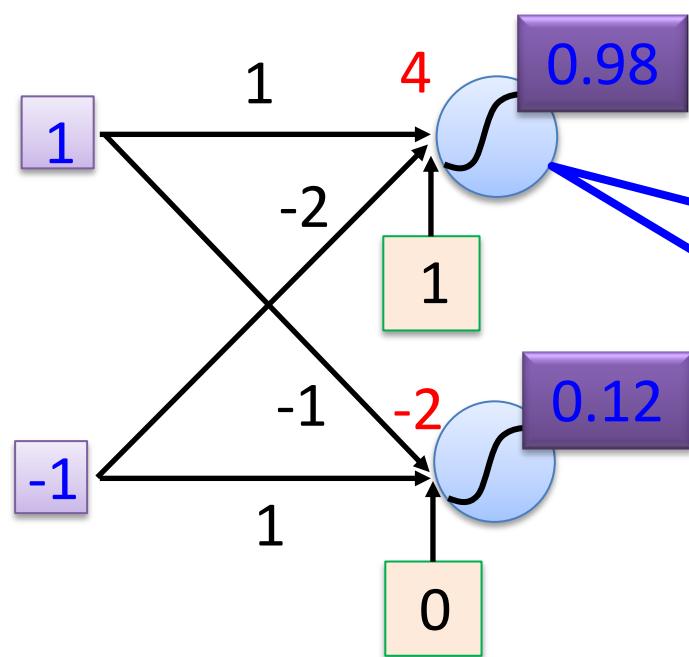


Elements of Neural Networks

Introduction to Neural Networks

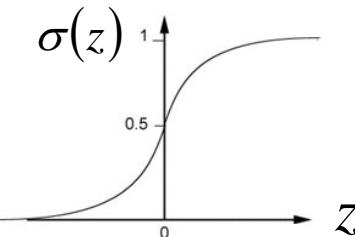
- A simple network, toy example

$$(1 \cdot 1) + (-1) \cdot (-2) + 1 = 4$$



Sigmoid Function

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

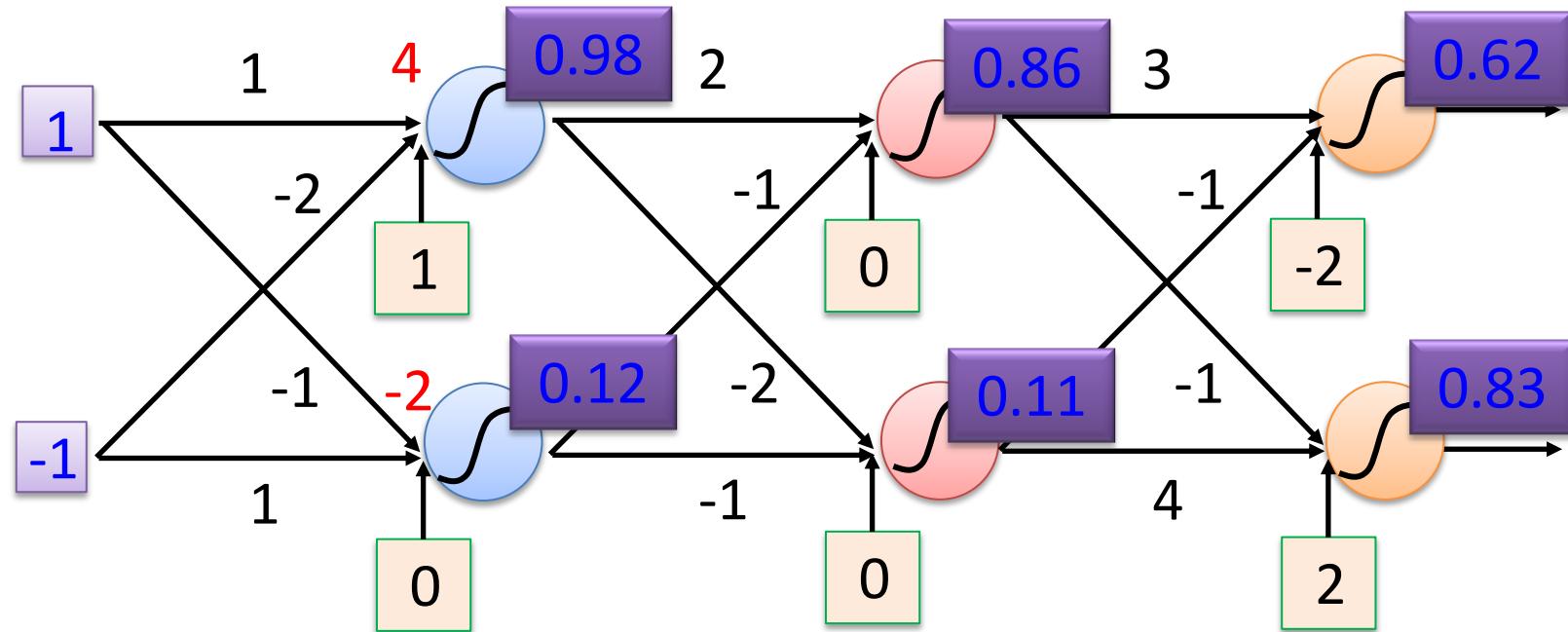


$$1 \cdot (-1) + (-1) \cdot 1 + 0 = -2$$

Elements of Neural Networks

Introduction to Neural Networks

- A simple network, toy example (cont'd)
 - For an input vector $[1 \ -1]^T$, the output is $[0.62 \ 0.83]^T$

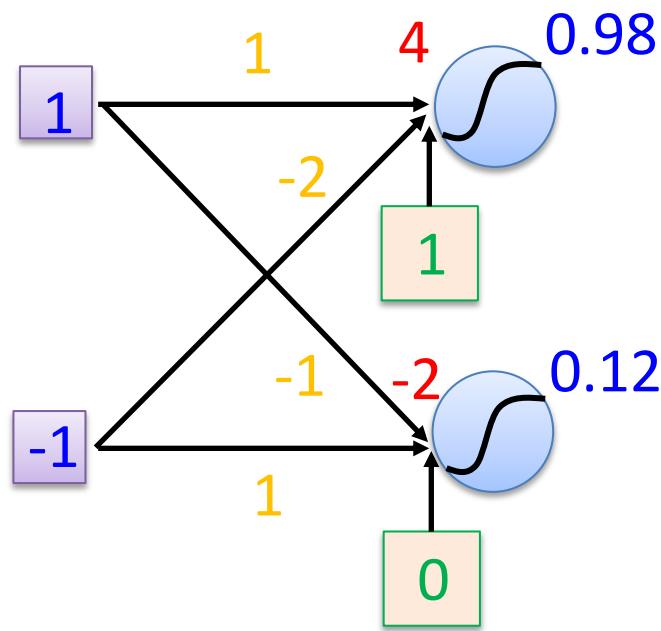


$$f: R^2 \rightarrow R^2 \quad f \left(\begin{bmatrix} 1 \\ -1 \end{bmatrix} \right) = \begin{bmatrix} 0.62 \\ 0.83 \end{bmatrix}$$

Matrix Operation

Introduction to Neural Networks

- Matrix operations are helpful when working with multidimensional inputs and outputs



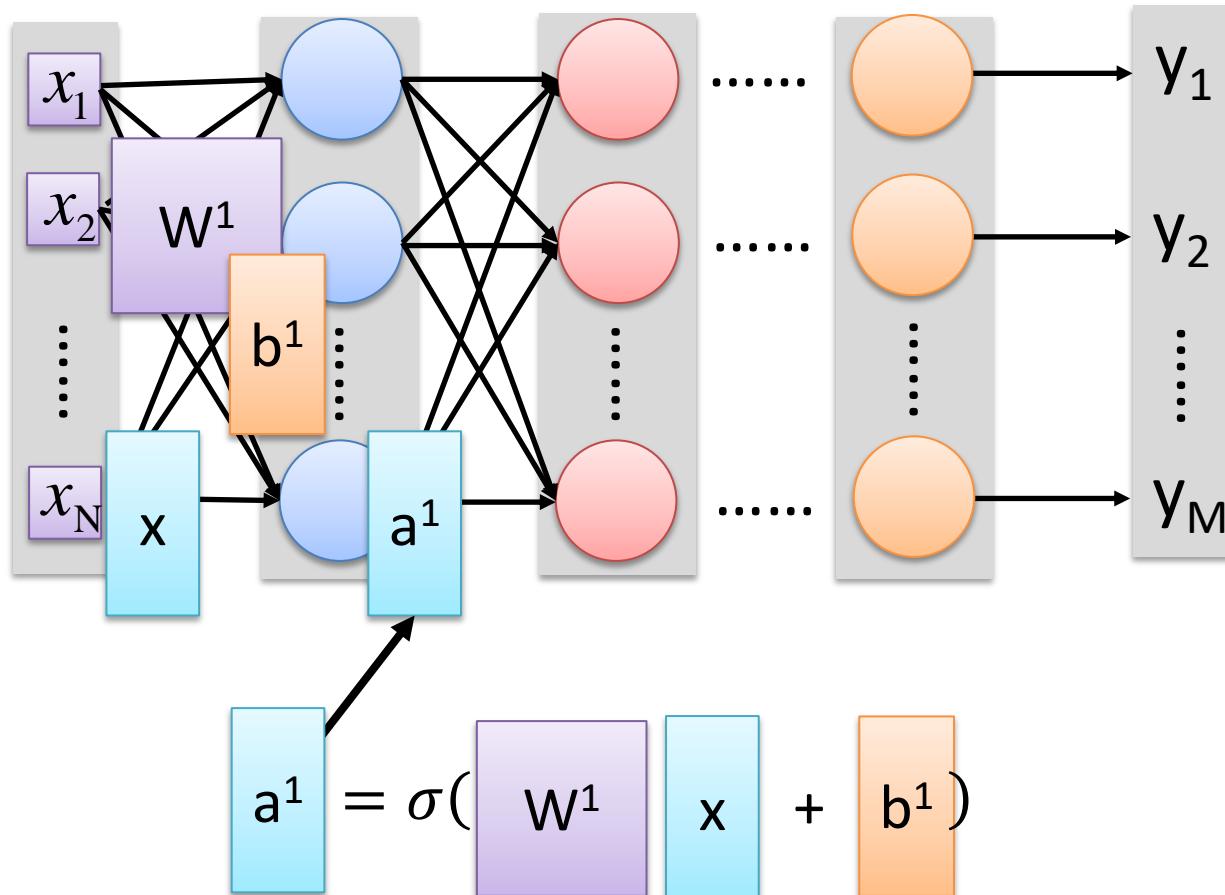
$$\sigma(\underbrace{Wx + b}_{a}) = a$$

$$\sigma(\underbrace{\begin{bmatrix} 1 & -2 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}}_{\begin{bmatrix} 4 \\ -2 \end{bmatrix}}) = \begin{bmatrix} 0.98 \\ 0.12 \end{bmatrix}$$

Matrix Operation

Introduction to Neural Networks

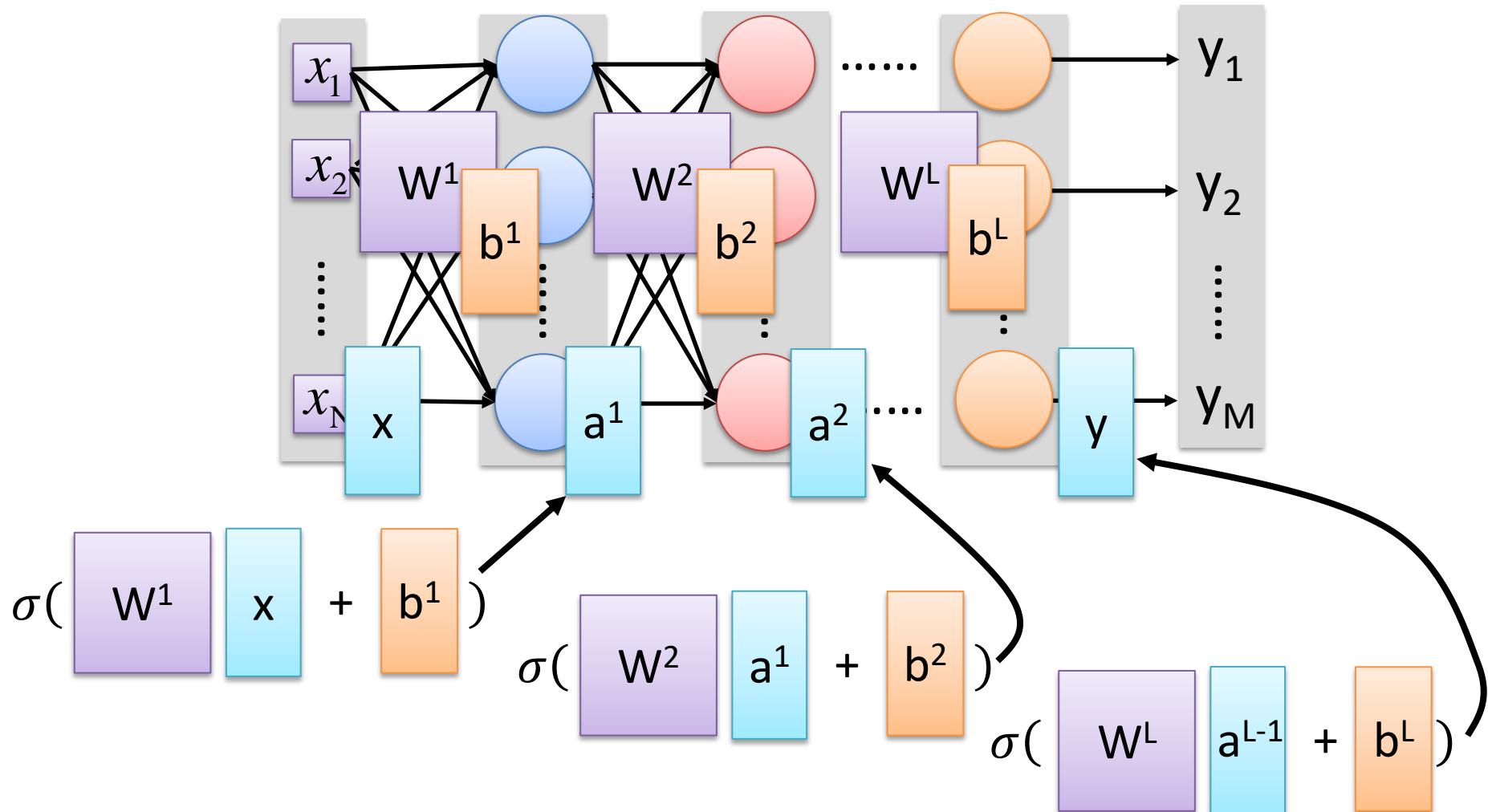
- Multilayer NN, matrix calculations for the first layer
 - Input vector x , weights matrix W^1 , bias vector b^1 , output vector a^1



Matrix Operation

Introduction to Neural Networks

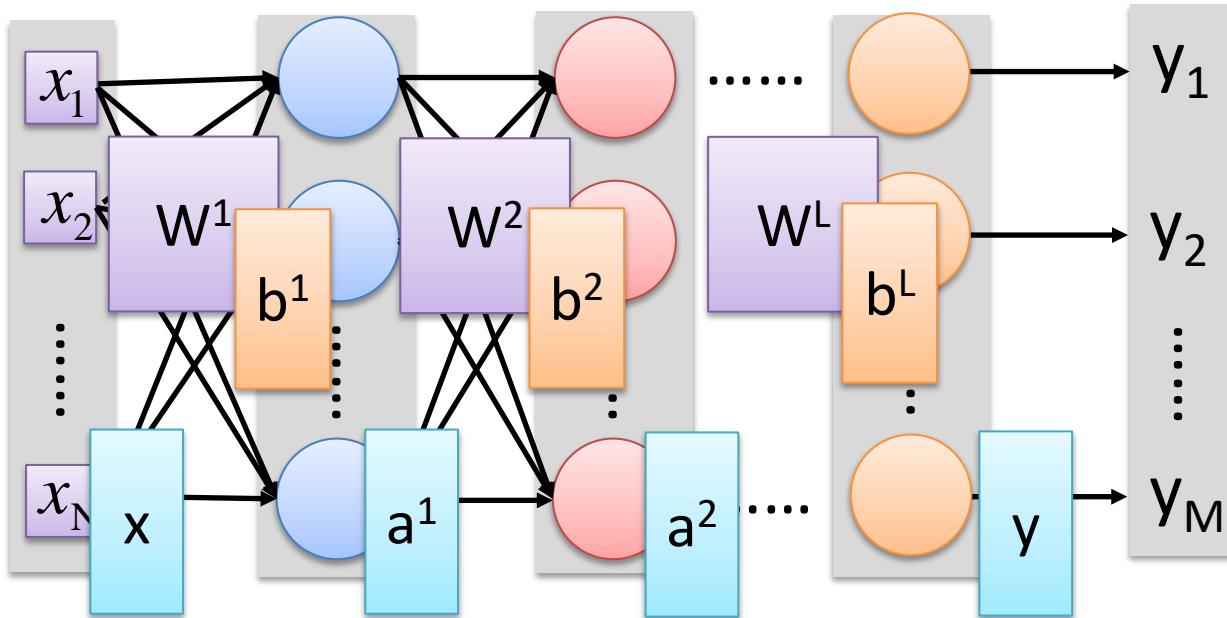
- Multilayer NN, matrix calculations for all layers



Matrix Operation

Introduction to Neural Networks

- Multilayer NN, function f maps inputs x to outputs y , i.e., $y = f(x)$



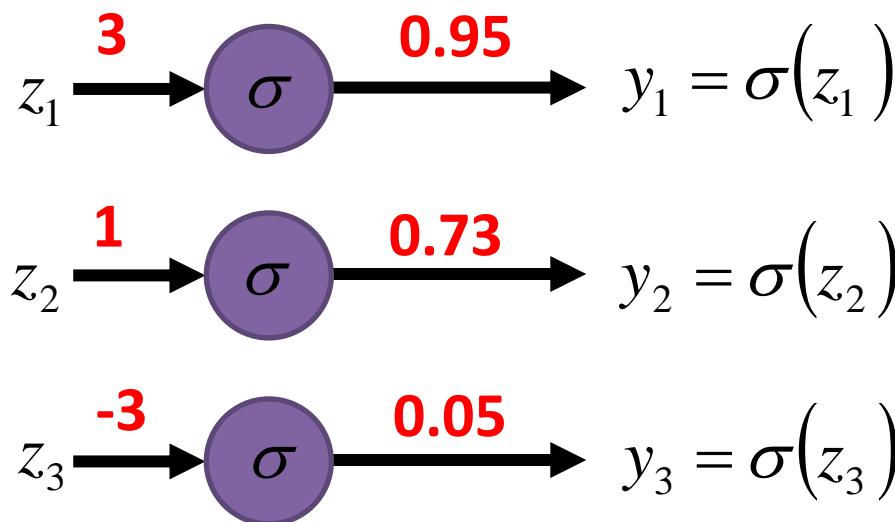
$$y = f(x) = \sigma(W^L \cdots \sigma(W^2 \sigma(W^1 x + b^1) + b^2) \cdots + b^L)$$

Softmax Layer

Introduction to Neural Networks

- In **multi-class classification** tasks, the output layer is typically a **softmax layer**
 - I.e., it employs a **softmax activation function**
 - If a layer with a sigmoid activation function is used as the output layer instead, the predictions by the NN may not be easy to interpret
 - Note that an output layer with sigmoid activations can still be used for binary classification

A Layer with Sigmoid Activations



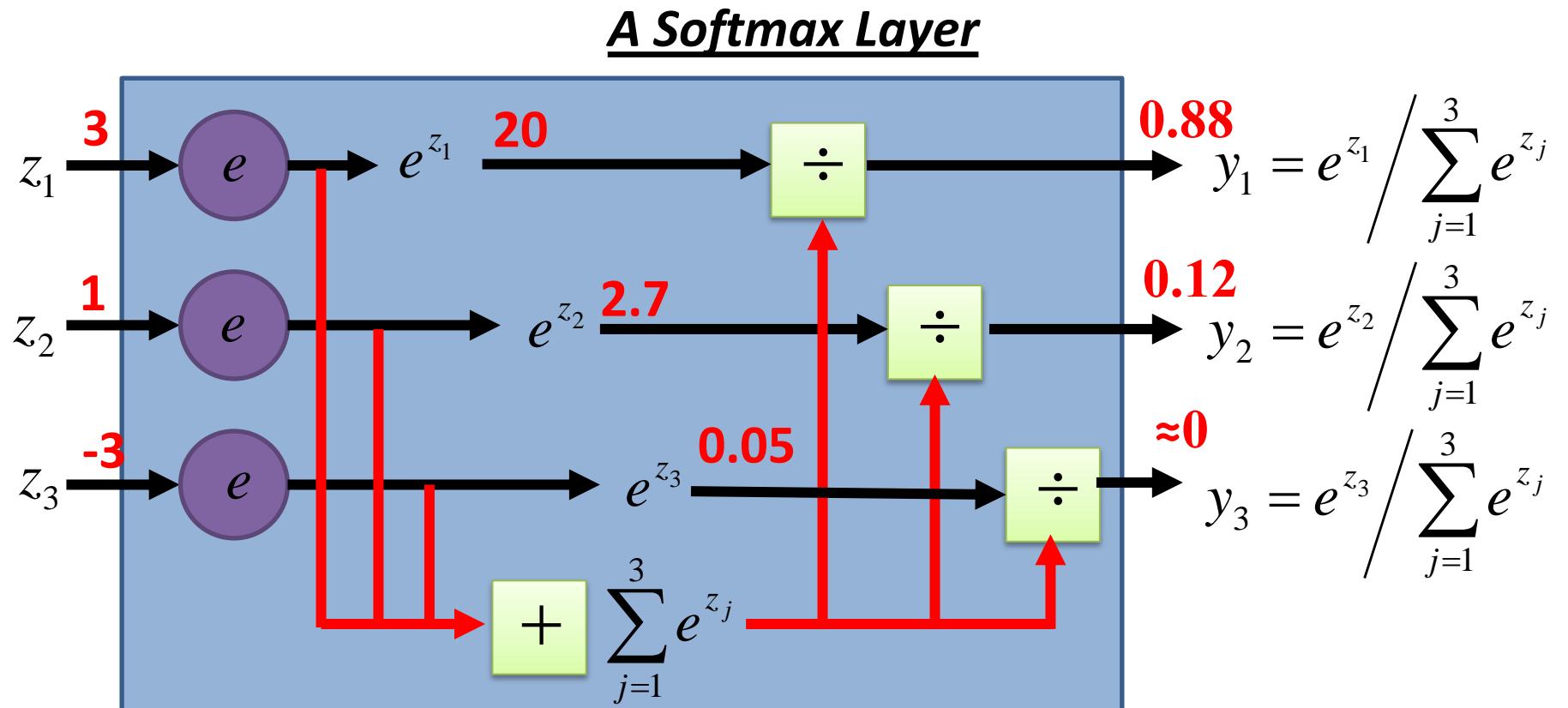
Softmax Layer

Introduction to Neural Networks

- The **softmax layer** applies softmax activations to output a probability value in the range [0, 1]
 - The values z inputted to the softmax layer are referred to as *logits*

Probability:

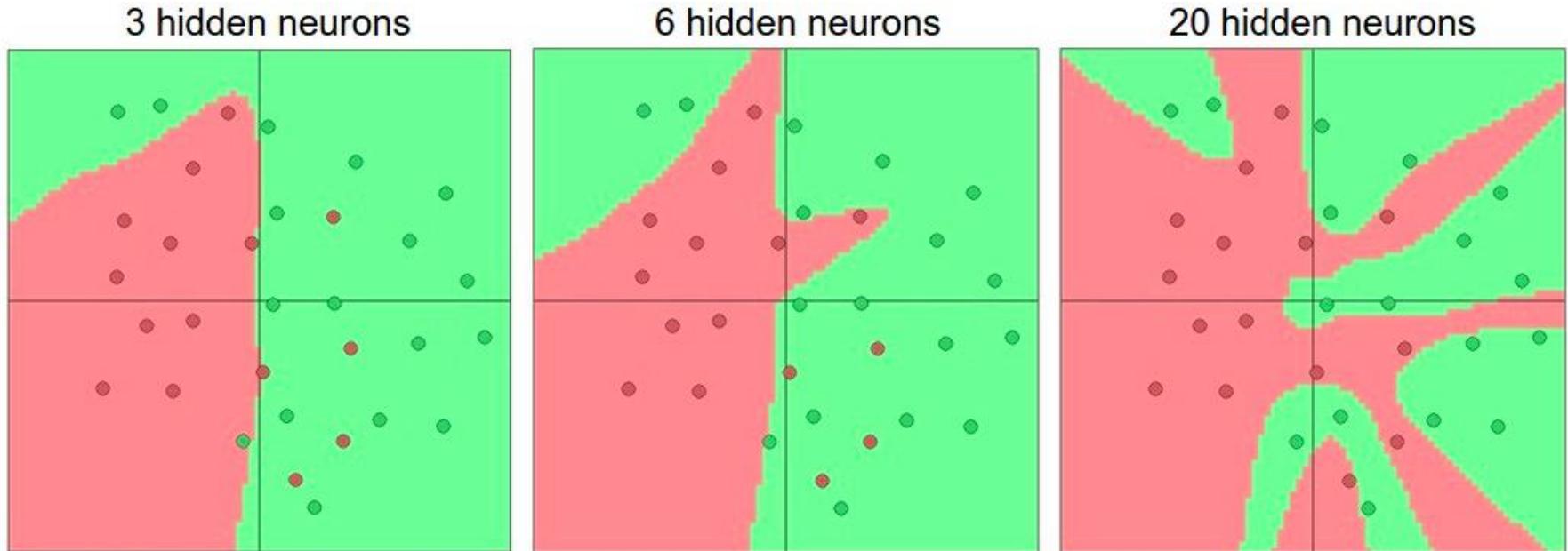
- $0 < y_i < 1$
- $\sum_i y_i = 1$



Activation Functions

Introduction to Neural Networks

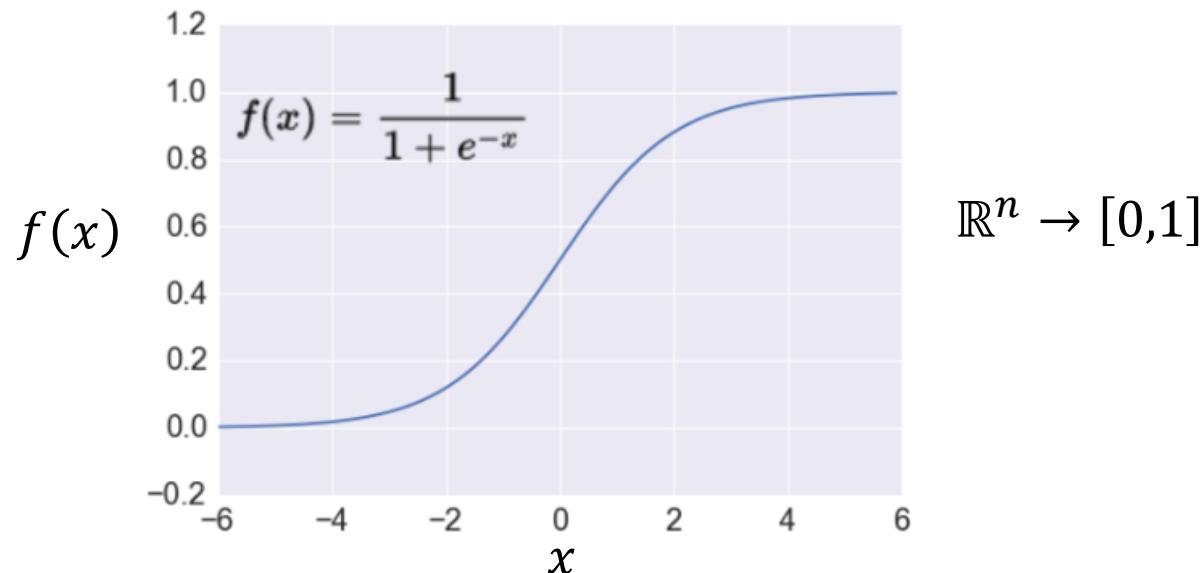
- **Non-linear activations** are needed to learn complex (non-linear) data representations
 - Otherwise, NNs would be just a linear function (such as $W_1 W_2 x = Wx$)
 - NNs with large number of layers (and neurons) can approximate more complex functions
 - Figure: more neurons improve representation (but, may overfit)



Activation: Sigmoid

Introduction to Neural Networks

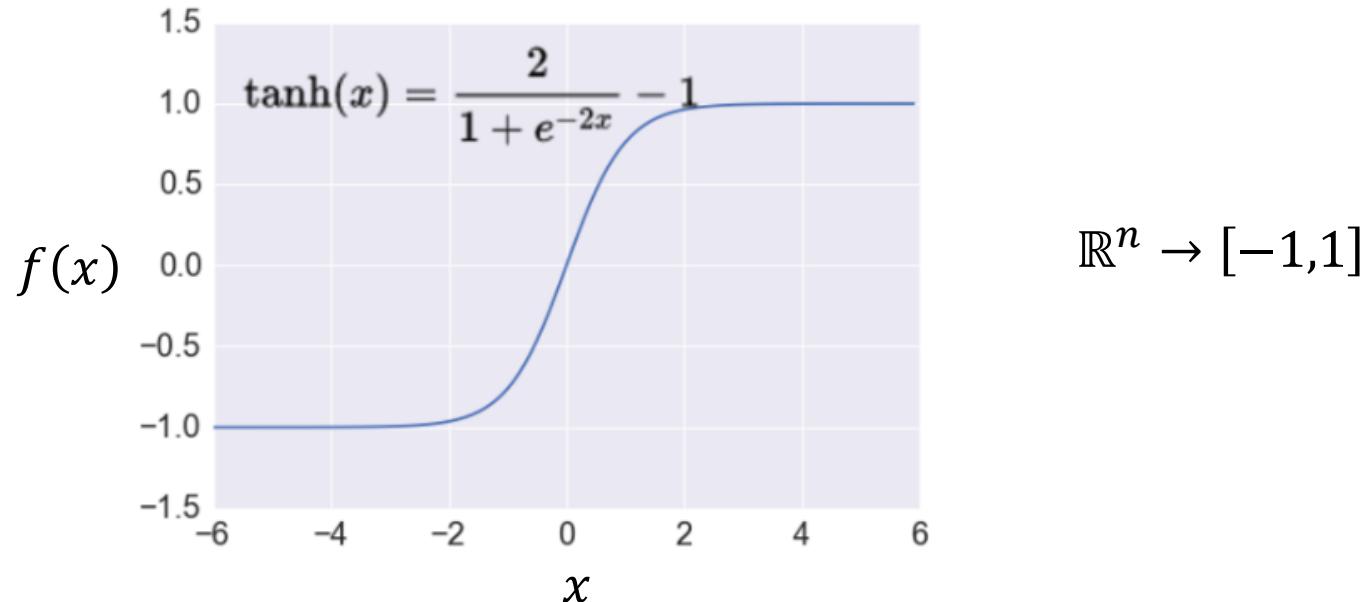
- **Sigmoid function** σ : takes a real-valued number and “squashes” it into the range between 0 and 1
 - The output can be interpreted as the firing rate of a biological neuron
 - Not firing = 0; Fully firing = 1
 - When the neuron’s activation are 0 or 1, sigmoid neurons saturate
 - Gradients at these regions are almost zero (almost no signal will flow)
 - Sigmoid activations are less common in modern NNs



Activation: Tanh

Introduction to Neural Networks

- **Tanh function:** takes a real-valued number and “squashes” it into range between -1 and 1
 - Like sigmoid, tanh neurons saturate
 - Unlike sigmoid, the output is zero-centered
 - It is therefore preferred than sigmoid
 - Tanh is a scaled sigmoid: $\tanh(x) = 2 \cdot \sigma(2x) - 1$



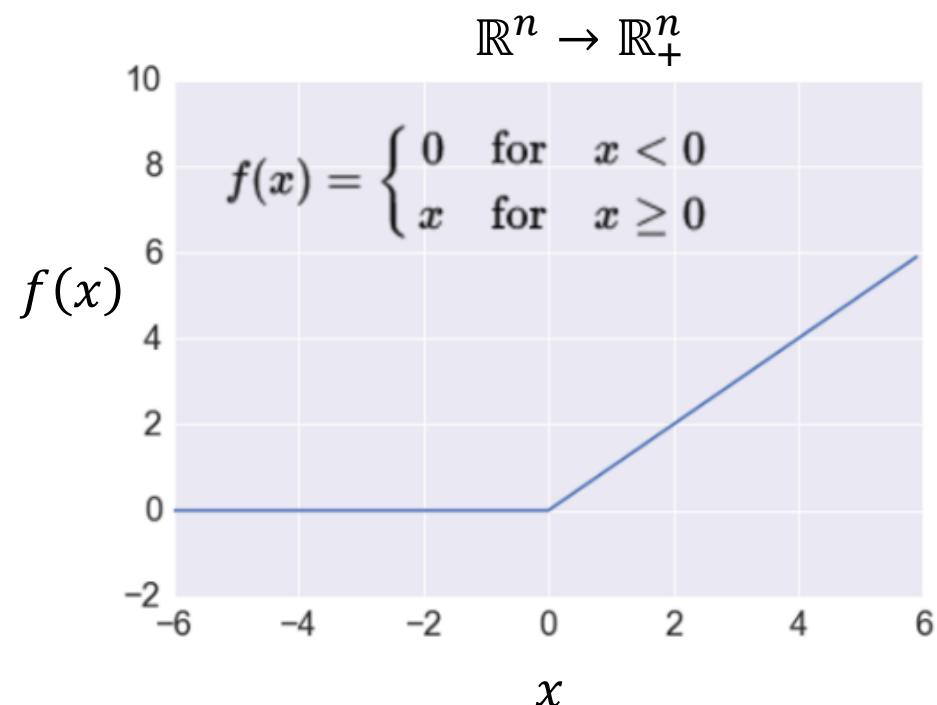
Activation: ReLU

Introduction to Neural Networks

- **ReLU** (Rectified Linear Unit): takes a real-valued number and thresholds it at zero

$$f(x) = \max(0, x)$$

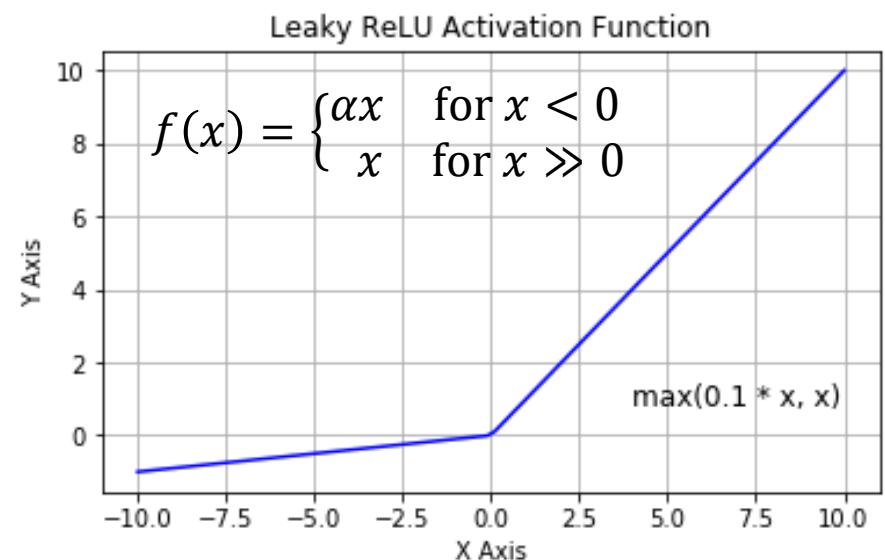
- Most modern deep NNs use ReLU activations
- ReLU is fast to compute
 - Compared to sigmoid, tanh
 - Simply threshold a matrix at zero
- Accelerates the convergence of gradient descent
 - Due to linear, non-saturating form
- Prevents the gradient vanishing problem



Activation: Leaky ReLU

Introduction to Neural Networks

- The problem of ReLU activations: they can “die”
 - ReLU could cause weights to update in a way that the gradients can become zero and the neuron will not activate again on any data
 - E.g., when a large learning rate is used
- **Leaky ReLU** activation function is a variant of ReLU
 - Instead of the function being 0 when $x < 0$, a leaky ReLU has a small negative slope (e.g., $\alpha = 0.01$, or similar)
 - This resolves the dying ReLU problem
 - Most current works still use ReLU
 - With a proper setting of the learning rate, the problem of dying ReLU can be avoided

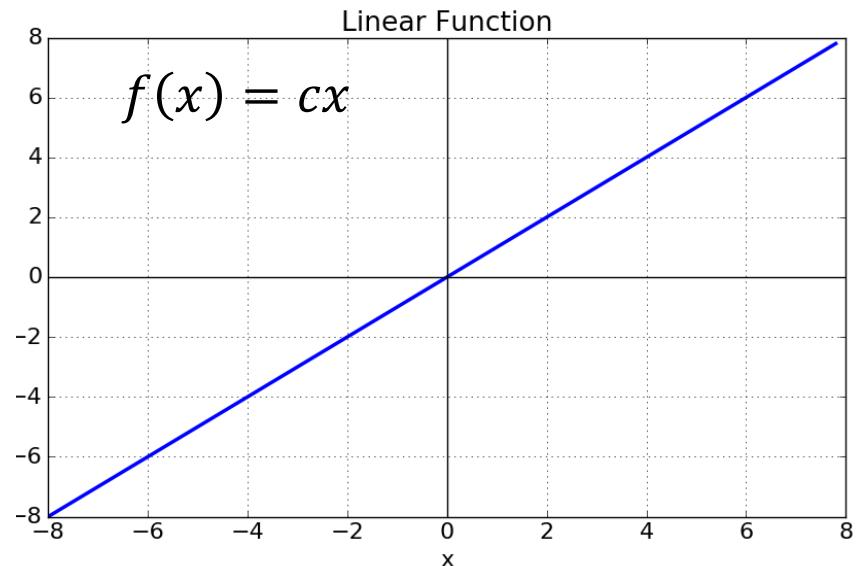


Activation: Linear Function

Introduction to Neural Networks

- **Linear function** means that the output signal is proportional to the input signal to the neuron
 - If the value of the constant c is 1, it is also called **identity activation function**
 - This activation type is used in regression problems
 - E.g., the last layer can have linear activation function, in order to output a real number (and not a class membership)

$$\mathbb{R}^n \rightarrow \mathbb{R}^n$$



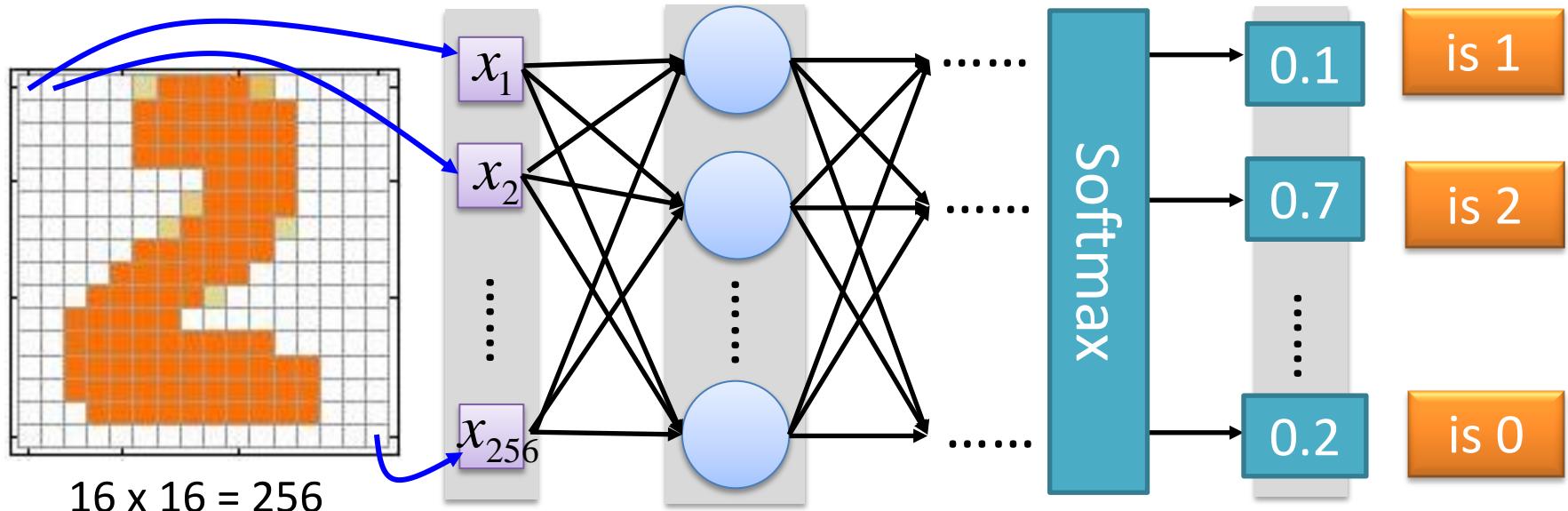
Training NNs

Training Neural Networks

- The network *parameters* θ include the **weight matrices** and **bias vectors** from all layers

$$\theta = \{W^1, b^1, W^2, b^2, \dots, W^L, b^L\}$$

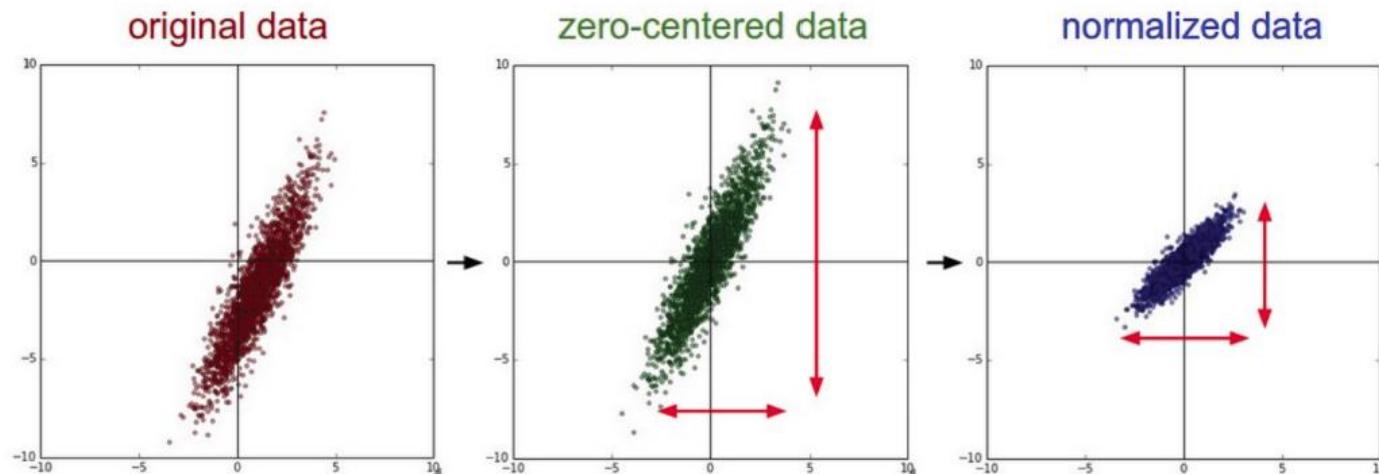
- Often, the model parameters θ are referred to as **weights**
- Training a model to learn a set of parameters θ that are optimal (according to a criterion) is one of the greatest challenges in ML



Training NNs

Training Neural Networks

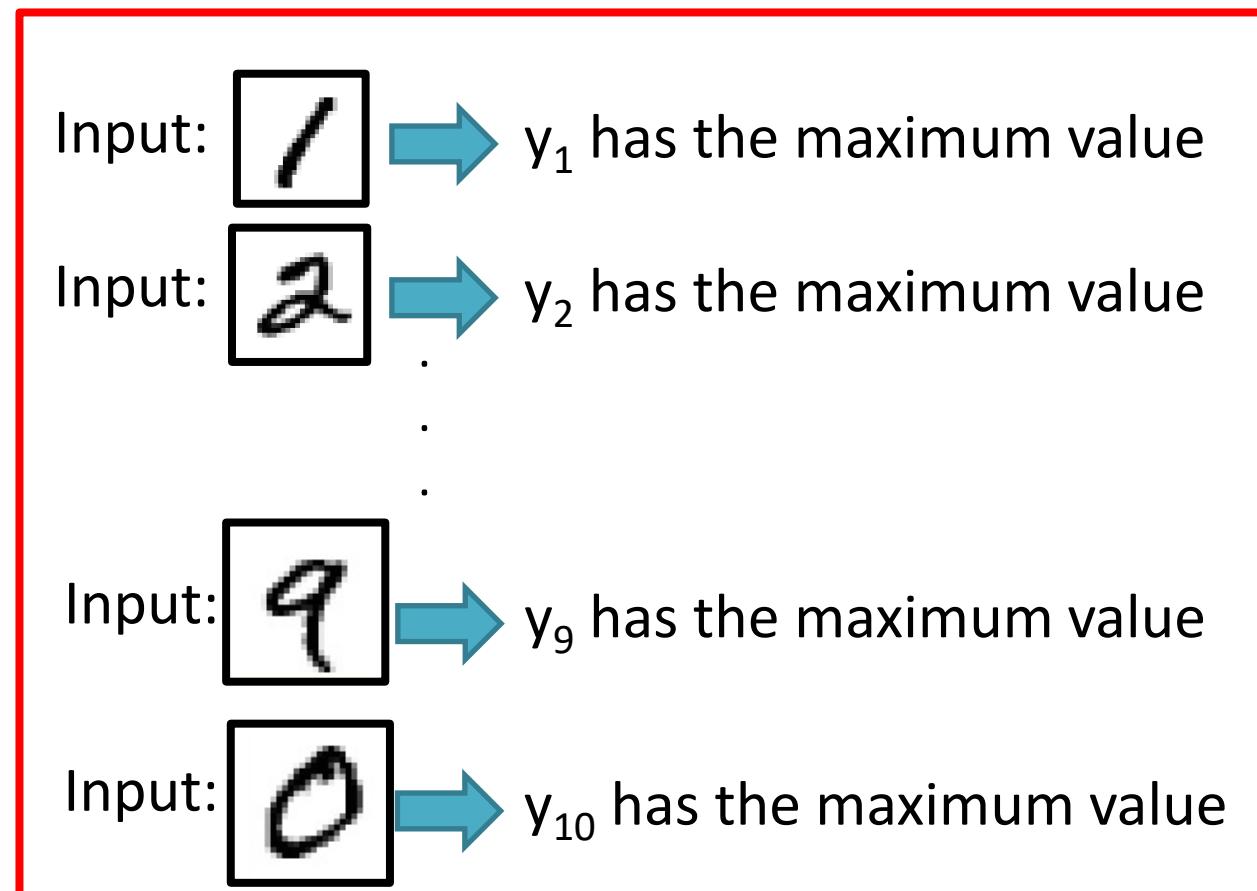
- **Data preprocessing** - helps convergence during training
 - **Mean subtraction**, to obtain zero-centered data
 - Subtract the mean for each individual data dimension (feature)
 - **Normalization**
 - Divide each feature by its standard deviation
 - To obtain standard deviation of 1 for each data dimension (feature)
 - Or, scale the data within the range [0,1] or [-1, 1]
 - E.g., image pixel intensities are divided by 255 to be scaled in the [0,1] range



Training NNs

Training Neural Networks

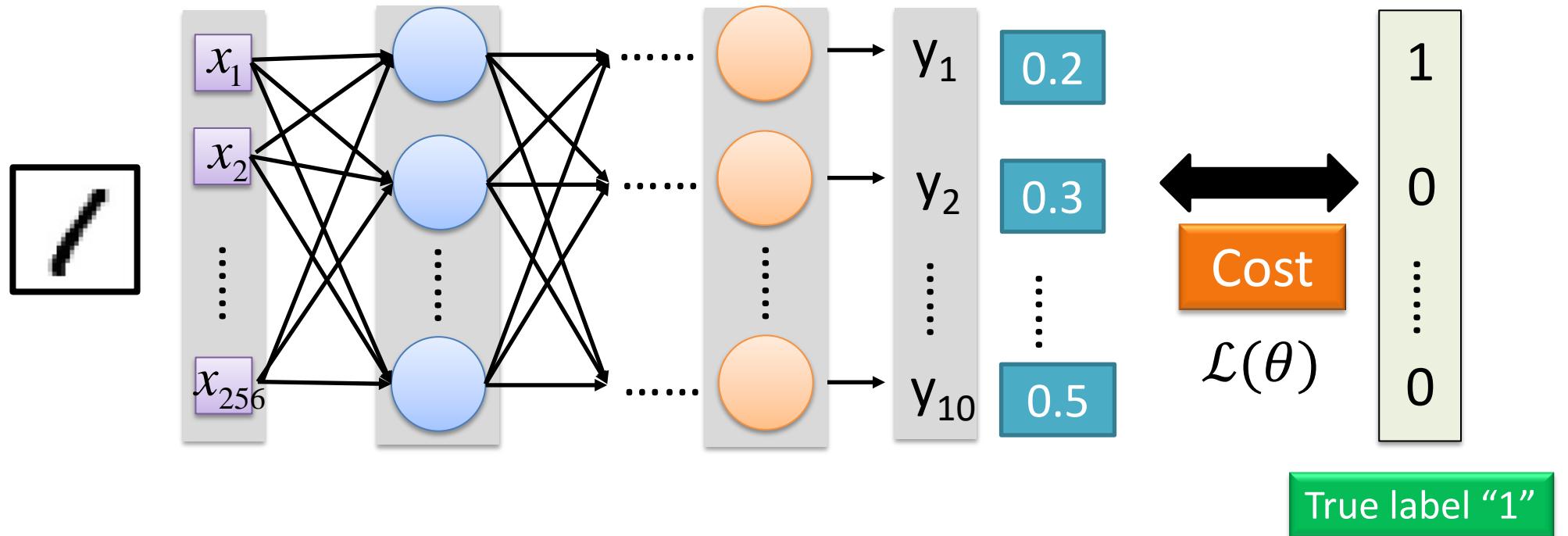
- To train a NN, set the parameters θ such that for a training subset of images, the corresponding elements in the predicted output have maximum values



Training NNs

Training Neural Networks

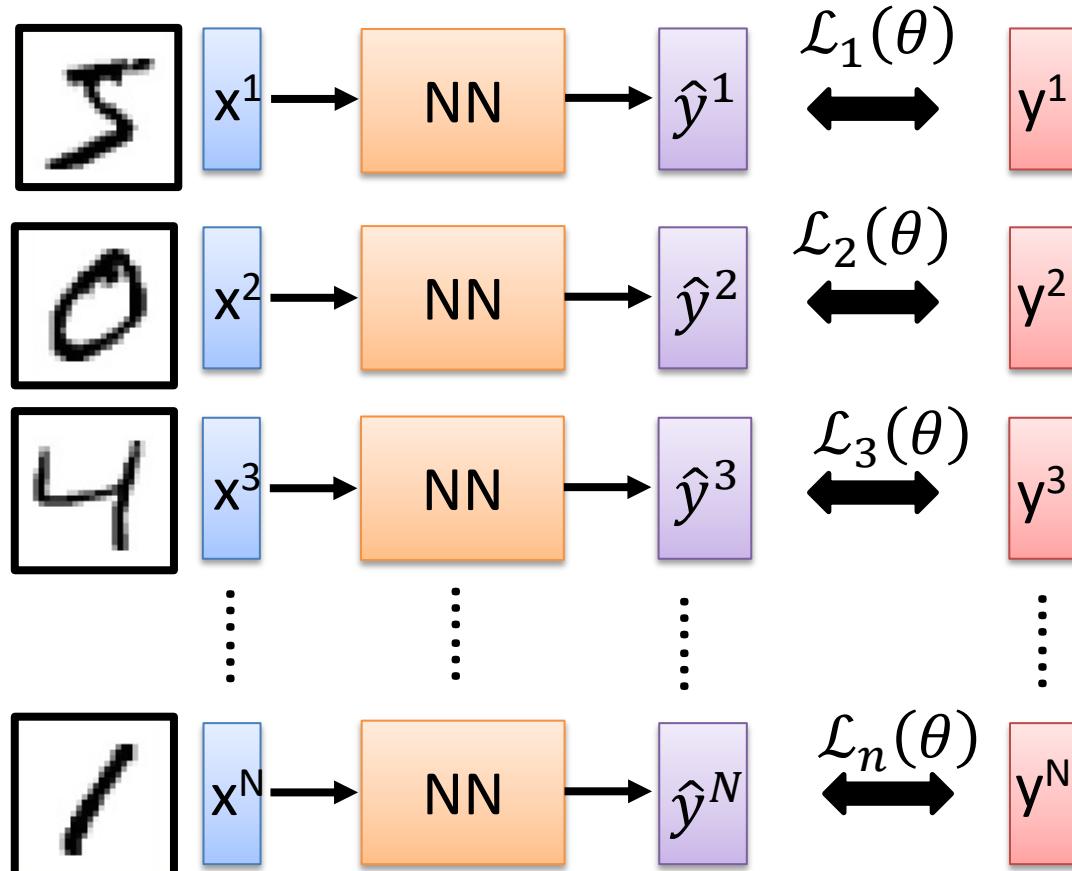
- Define a **loss function**/objective function/cost function $\mathcal{L}(\theta)$ that calculates the difference (error) between the model prediction and the true label
 - E.g., $\mathcal{L}(\theta)$ can be mean-squared error, cross-entropy, etc.



Training NNs

Training Neural Networks

- For a training set of N images, calculate the total loss overall all images:
$$\mathcal{L}(\theta) = \sum_{n=1}^N \mathcal{L}_n(\theta)$$
- Find the optimal parameters θ^* that minimize the total loss $\mathcal{L}(\theta)$



What is the use of Loss Functions?

- The loss function is used in any Machine Learning method to give feedback on how good or bad is our model doing for a given input.
- It allows us to find out if our training has been successful. In some cases, it can be minimized to minimize prediction errors.
- Many loss functions have been introduced over time, from very basic ones such as Mean Squared Error (MSE) to more complex ones such as Cross-Entropy Loss (CEL).

What do you mean by Cost Function?

- Before going into Artificial Neural networks, we need to learn what is the cost function. The general term of the cost function is an evaluation criterion for optimization.
- For example: In the case of Linear Regression, Mean Squared Error is used as a cost function for finding optimal weights of the linear regression model by minimizing its value.
- And in the case of Logistic Regression, Cross-Entropy is used as a cost function for finding optimal weights of the logistic regression model by minimizing its value.
- And in the case of Neural Networks or Deep Learning models, it's a loss function that can be defined using different metrics like MSE (Mean Squared Error), Accuracy (Accuracy), etc.

Loss Functions

Training Neural Networks

- *Regression tasks*

Training examples

Pairs of N inputs x_i and ground-truth output values y_i

Output Layer

Linear (Identity) or Sigmoid Activation

Loss function

Mean Squared Error

$$\mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2$$

Mean Absolute Error

$$\mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^n |y^{(i)} - \hat{y}^{(i)}|$$

Loss Functions

Training Neural Networks

- *Classification tasks*

Training examples

Pairs of N inputs x_i and ground-truth class labels y_i

Output Layer

Softmax Activations
[maps to a probability distribution]

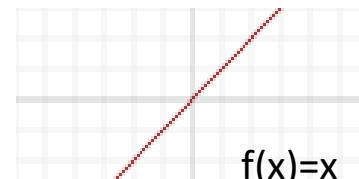
$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\top \mathbf{w}_j}}{\sum_{k=1}^K e^{\mathbf{x}^\top \mathbf{w}_k}}$$

Loss function

Cross-entropy $\mathcal{L}(\theta) = -\frac{1}{N} \sum_{i=1}^N \sum_{k=1}^K [y_k^{(i)} \log \hat{y}_k^{(i)} + (1 - y_k^{(i)}) \log (1 - \hat{y}_k^{(i)})]$

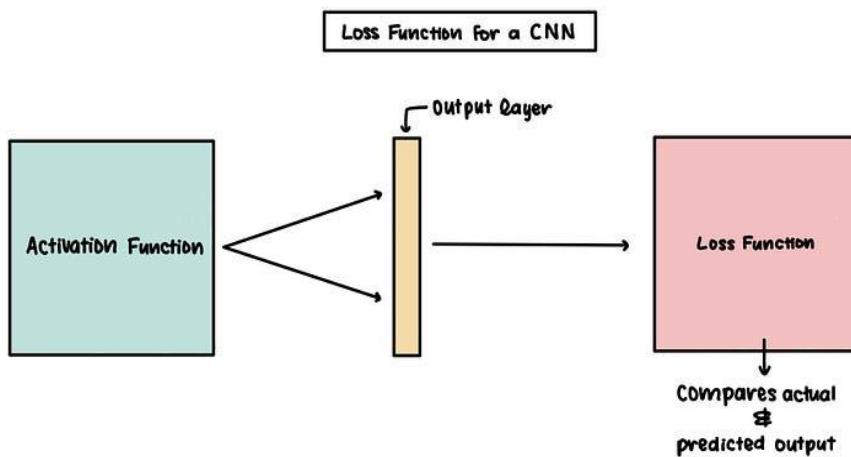
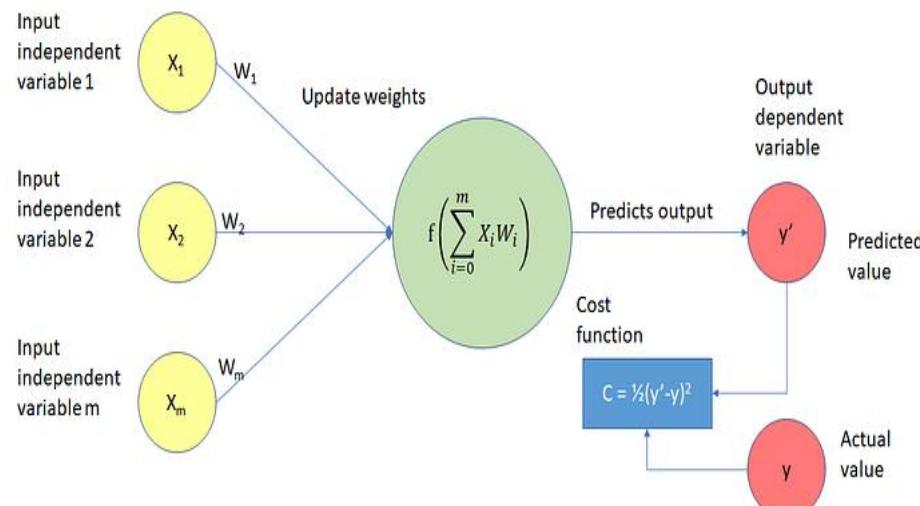
Ground-truth class labels y_i and model predicted class labels \hat{y}_i

Loss functions and output

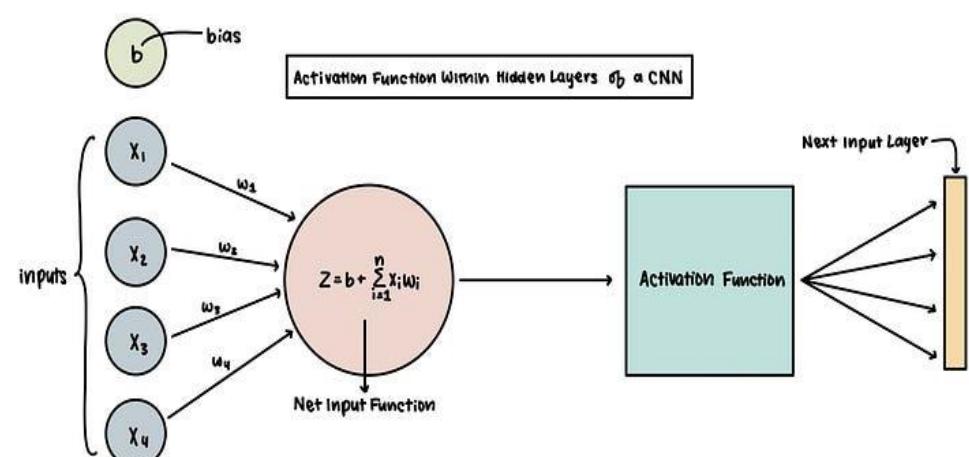
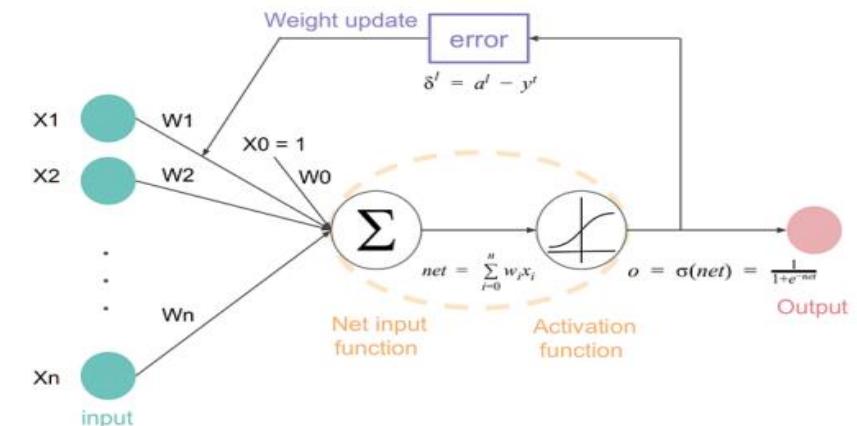
	Classification	Regression
Training examples	$R^n \times \{\text{class_1, ..., class_n}\}$ (one-hot encoding)	$R^n \times R^m$
Output Layer	Soft-max [map R^n to a probability distribution] $P(y = j \mathbf{x}) = \frac{e^{\mathbf{x}^\top \mathbf{w}_j}}{\sum_{k=1}^K e^{\mathbf{x}^\top \mathbf{w}_k}}$	Linear (Identity) or Sigmoid 
Cost (loss) function	Cross-entropy $J(\theta) = -\frac{1}{n} \sum_{i=1}^n \sum_{k=1}^K \left[y_k^{(i)} \log \hat{y}_k^{(i)} + (1 - y_k^{(i)}) \log (1 - \hat{y}_k^{(i)}) \right]$	Mean Squared Error $J(\theta) = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2$ Mean Absolute Error $J(\theta) = \frac{1}{n} \sum_{i=1}^n y^{(i)} - \hat{y}^{(i)} $
List of loss functions		

Activation Function vs Loss Function

Loss function is a method of evaluating “how well your algorithm models your dataset”.



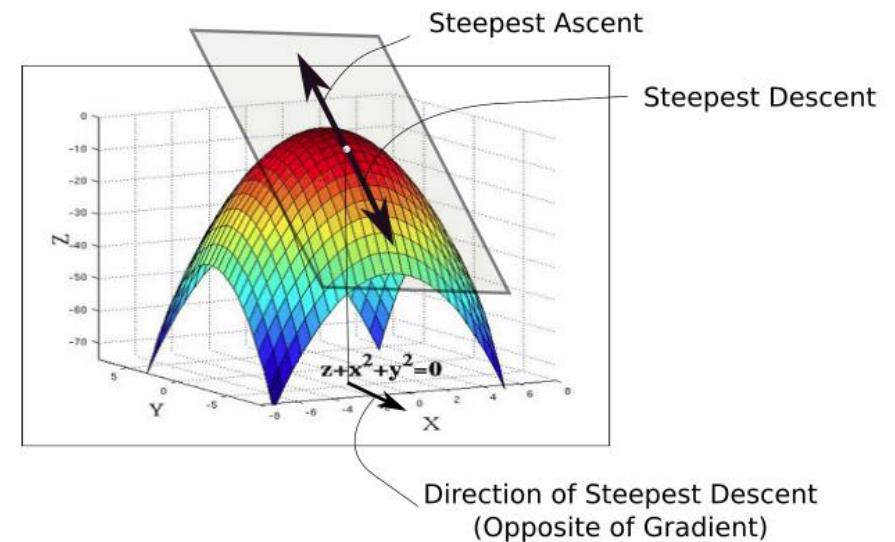
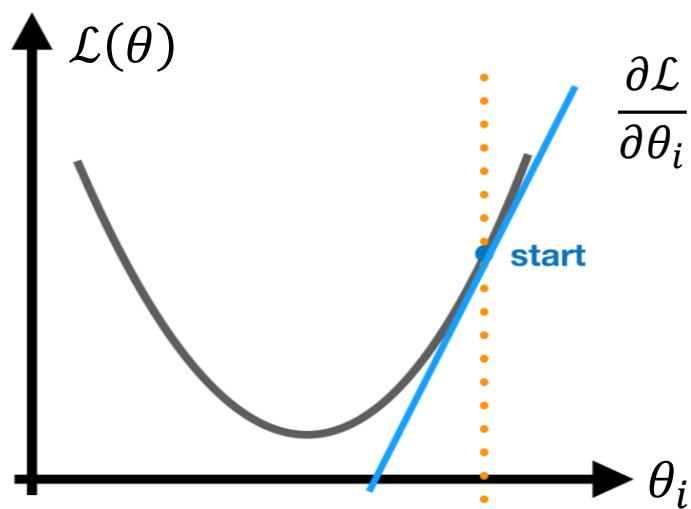
An activation function transforms the shape/representation of the data going into it.



Training NNs

Training Neural Networks

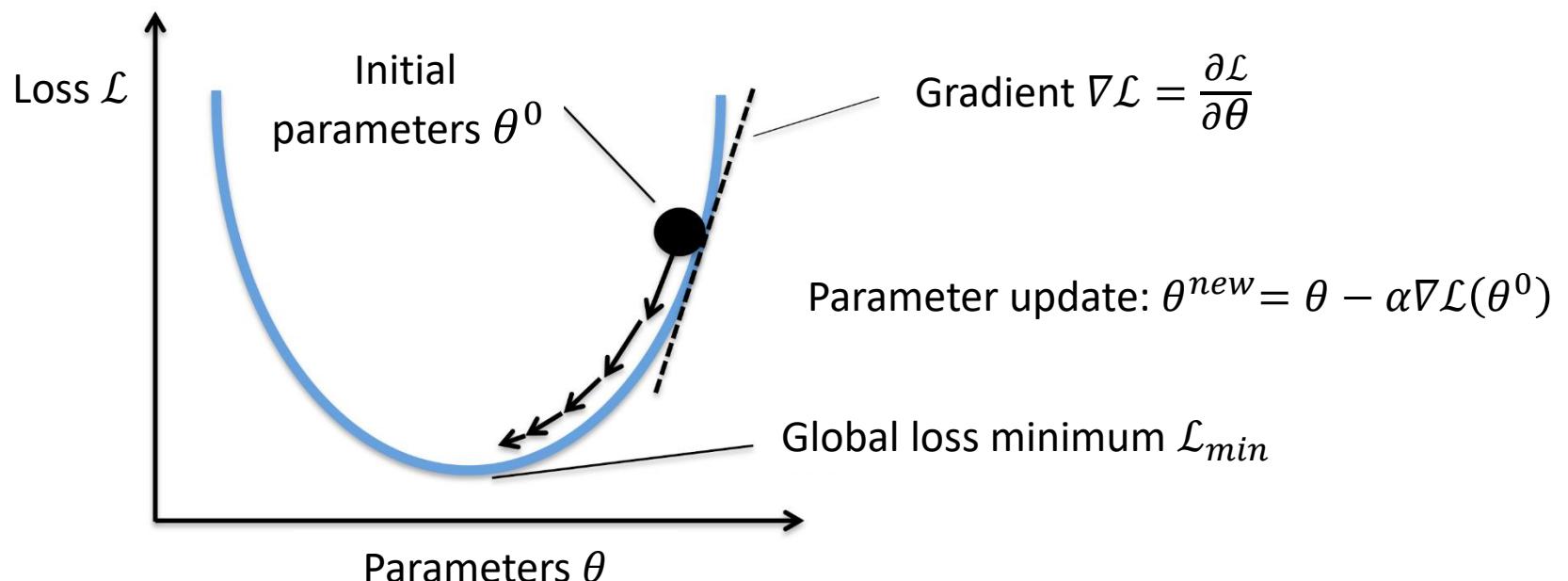
- Optimizing the loss function $\mathcal{L}(\theta)$
 - Almost all DL models these days are trained with a variant of the *gradient descent* (GD) algorithm
 - GD applies iterative refinement of the network **parameters θ**
 - GD uses the opposite direction of the **gradient** of the loss with respect to the NN parameters (i.e., $\nabla \mathcal{L}(\theta) = [\partial \mathcal{L} / \partial \theta_i]$) for updating θ
 - The gradient of the loss function $\nabla \mathcal{L}(\theta)$ gives the direction of fastest increase of the loss function $\mathcal{L}(\theta)$ when the parameters θ are changed



Gradient Descent Algorithm

Training Neural Networks

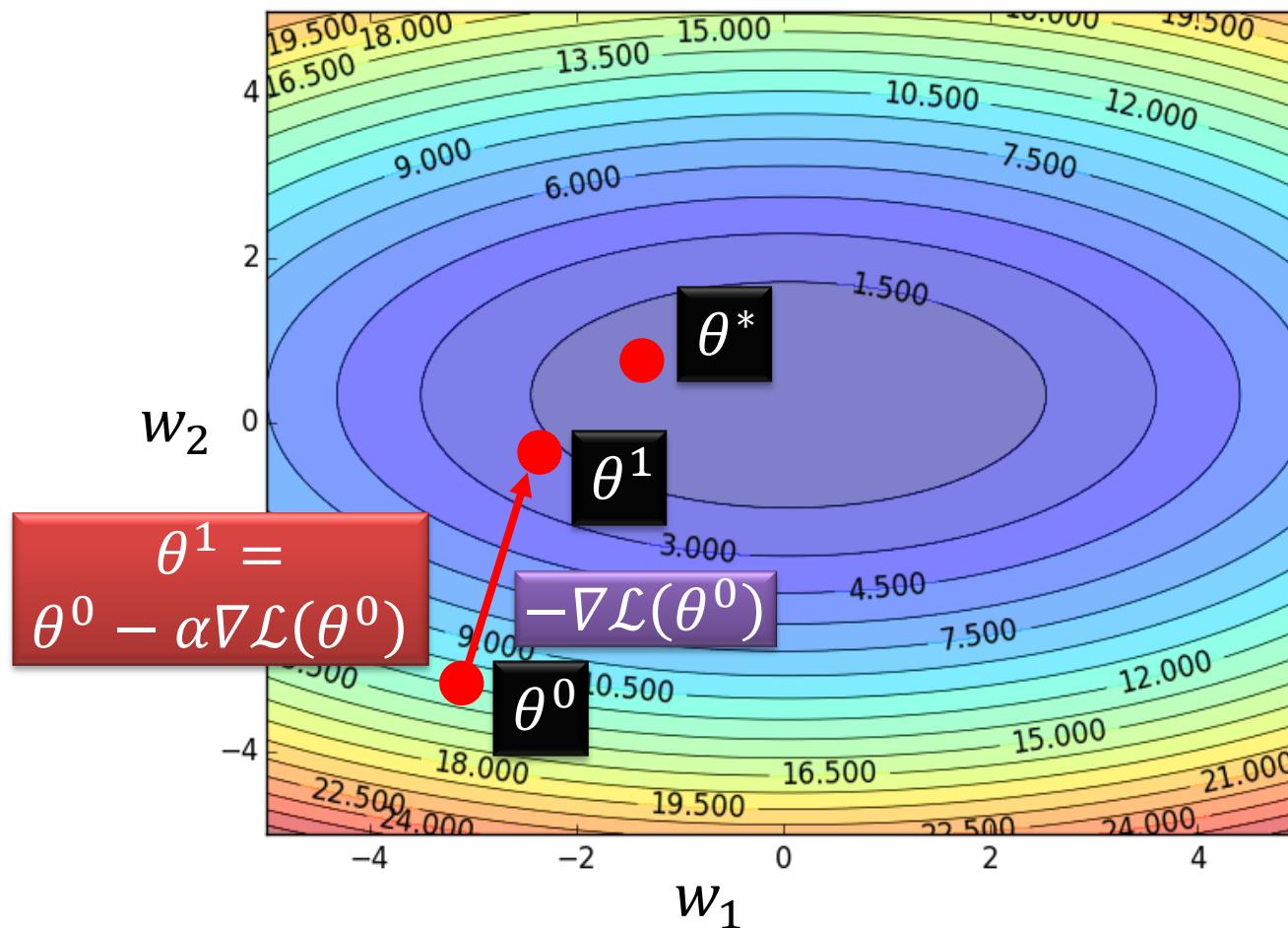
- Steps in the *gradient descent algorithm*:
 1. Randomly initialize the model parameters, θ^0
 2. Compute the gradient of the loss function at the initial parameters θ^0 : $\nabla \mathcal{L}(\theta^0)$
 3. Update the parameters as: $\theta^{new} = \theta^0 - \alpha \nabla \mathcal{L}(\theta^0)$
 - Where α is the learning rate
 4. Go to step 2 and repeat (until a terminating criterion is reached)



Gradient Descent Algorithm

Training Neural Networks

- Example: a NN with only 2 parameters w_1 and w_2 , i.e., $\theta = \{w_1, w_2\}$
 - The different colors represent the values of the loss (minimum loss θ^* is ≈ 1.3)



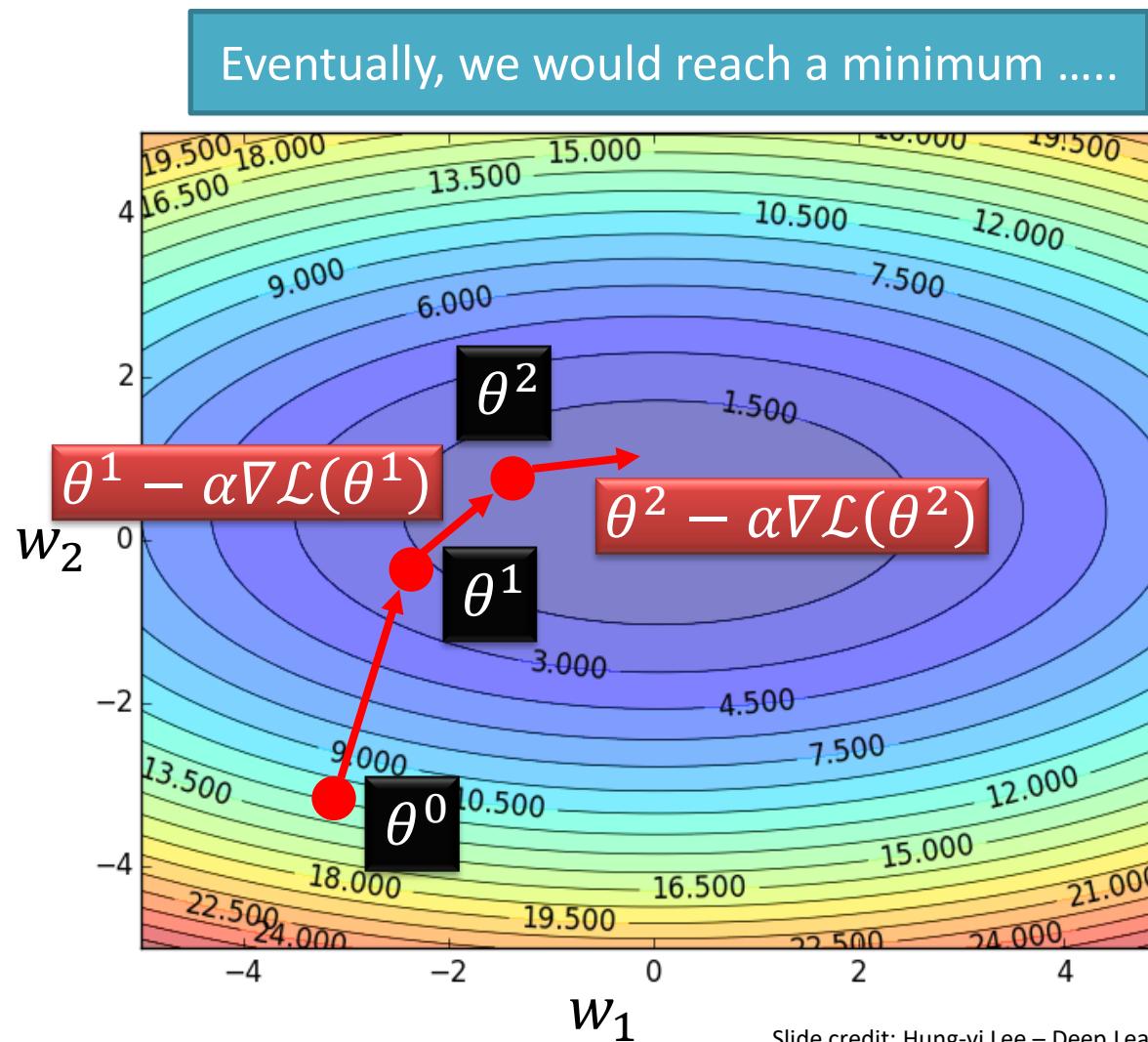
1. Randomly pick a starting point θ^0
2. Compute the gradient at θ^0 , $\nabla \mathcal{L}(\theta^0)$
3. Times the learning rate η , and update θ ,
 $\theta^{new} = \theta^0 - \alpha \nabla \mathcal{L}(\theta^0)$
4. Go to step 2, repeat

$$\nabla \mathcal{L}(\theta^0) = \begin{bmatrix} \partial \mathcal{L}(\theta^0) / \partial w_1 \\ \partial \mathcal{L}(\theta^0) / \partial w_2 \end{bmatrix}$$

Gradient Descent Algorithm

Training Neural Networks

- Example (contd.)

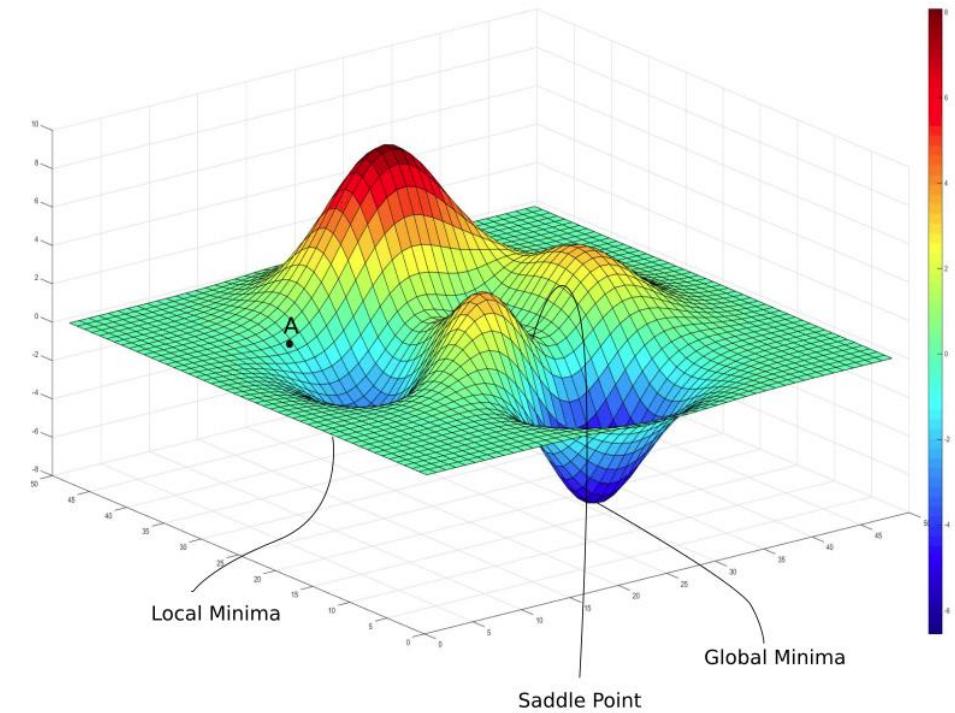
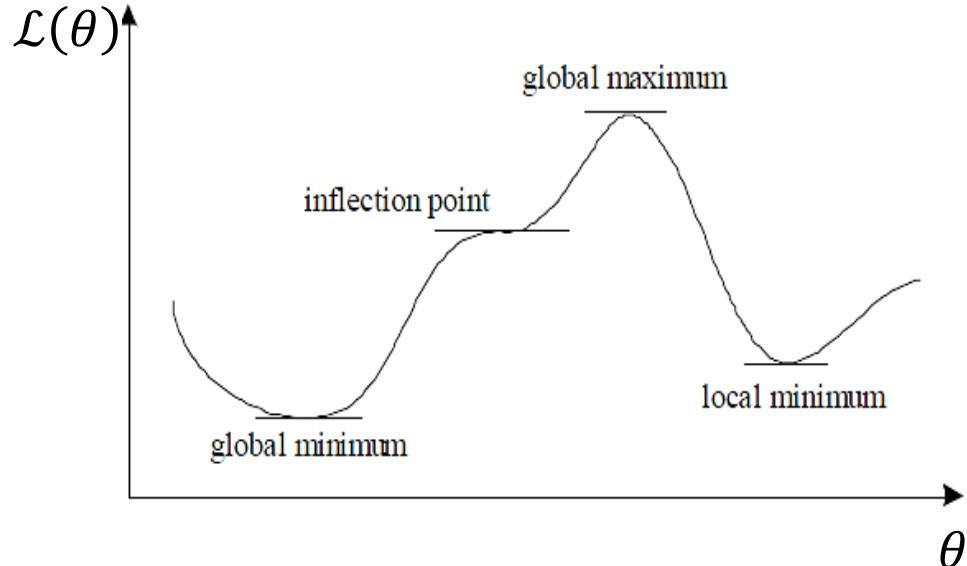


2. Compute the gradient at θ^{old} , $\nabla \mathcal{L}(\theta^{old})$
3. Times the learning rate η , and update θ ,
 $\theta^{new} = \theta^{old} - \alpha \nabla \mathcal{L}(\theta^{old})$
4. Go to step 2, repeat

Gradient Descent Algorithm

Training Neural Networks

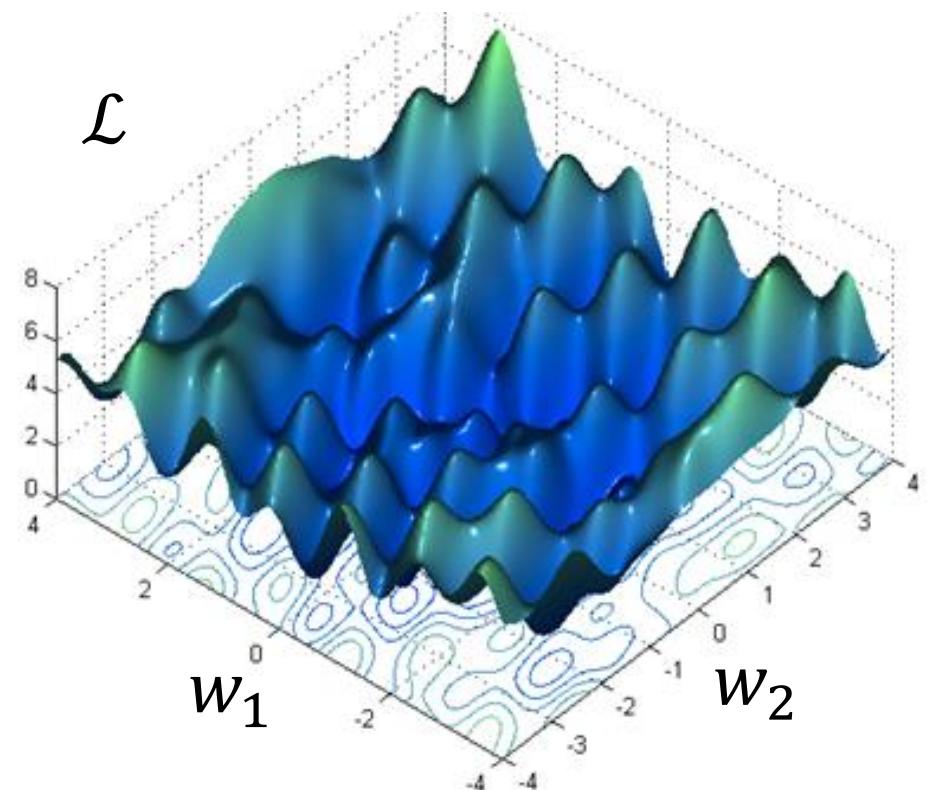
- Gradient descent algorithm stops when a **local minimum** of the loss surface is reached
 - GD does not guarantee reaching a **global minimum**
 - However, empirical evidence suggests that GD works well for NNs



Gradient Descent Algorithm

Training Neural Networks

- For most tasks, the **loss surface** $\mathcal{L}(\theta)$ is highly complex (and non-convex)
- Random initialization in NNs results in different initial parameters θ^0 every time the NN is trained
 - Gradient descent may reach different minima at every run
 - Therefore, NN will produce different predicted outputs
- In addition, currently we don't have algorithms that guarantee reaching a **global minimum** for an arbitrary loss function



Backpropagation

Training Neural Networks

- Modern NNs employ the **backpropagation** method for calculating the gradients of the loss function $\nabla \mathcal{L}(\theta) = \partial \mathcal{L} / \partial \theta_i$
 - Backpropagation is short for “backward propagation”
- For training NNs, **forward propagation** (forward pass) refers to passing the inputs x through the hidden layers to obtain the model outputs (predictions) y
 - The loss $\mathcal{L}(y, \hat{y})$ function is then calculated
 - **Backpropagation** traverses the network in reverse order, from the outputs y backward toward the inputs x to calculate the gradients of the loss function $\nabla \mathcal{L}(\theta)$
 - The chain rule is used for calculating the partial derivatives of the loss function with respect to the parameters θ in the different layers in the network
- Each update of the model parameters θ during training takes one forward and one backward pass (e.g., of a batch of inputs)
- Automatic calculation of the gradients (**automatic differentiation**) is available in all current deep learning libraries
 - It significantly simplifies the implementation of deep learning algorithms, since it obviates deriving the partial derivatives of the loss function by hand

Mini-batch Gradient Descent

Training Neural Networks

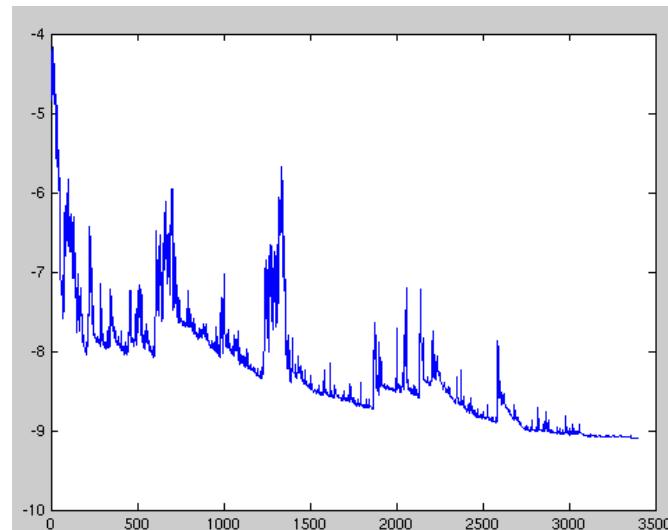
- It is wasteful to compute the loss over the **entire training dataset** to perform a single parameter update for large datasets
 - E.g., ImageNet has 14M images
 - Therefore, GD (a.k.a. vanilla GD) is almost always replaced with mini-batch GD
- ***Mini-batch gradient descent***
 - Approach:
 - Compute the loss $\mathcal{L}(\theta)$ on a mini-batch of images, update the parameters θ , and repeat until all images are used
 - At the next epoch, shuffle the training data, and repeat the above process
 - Mini-batch GD results in much faster training
 - Typical mini-batch size: 32 to 256 images
 - It works because the gradient from a mini-batch is a good approximation of the gradient from the entire training set

Stochastic Gradient Descent

Training Neural Networks

- ***Stochastic gradient descent***

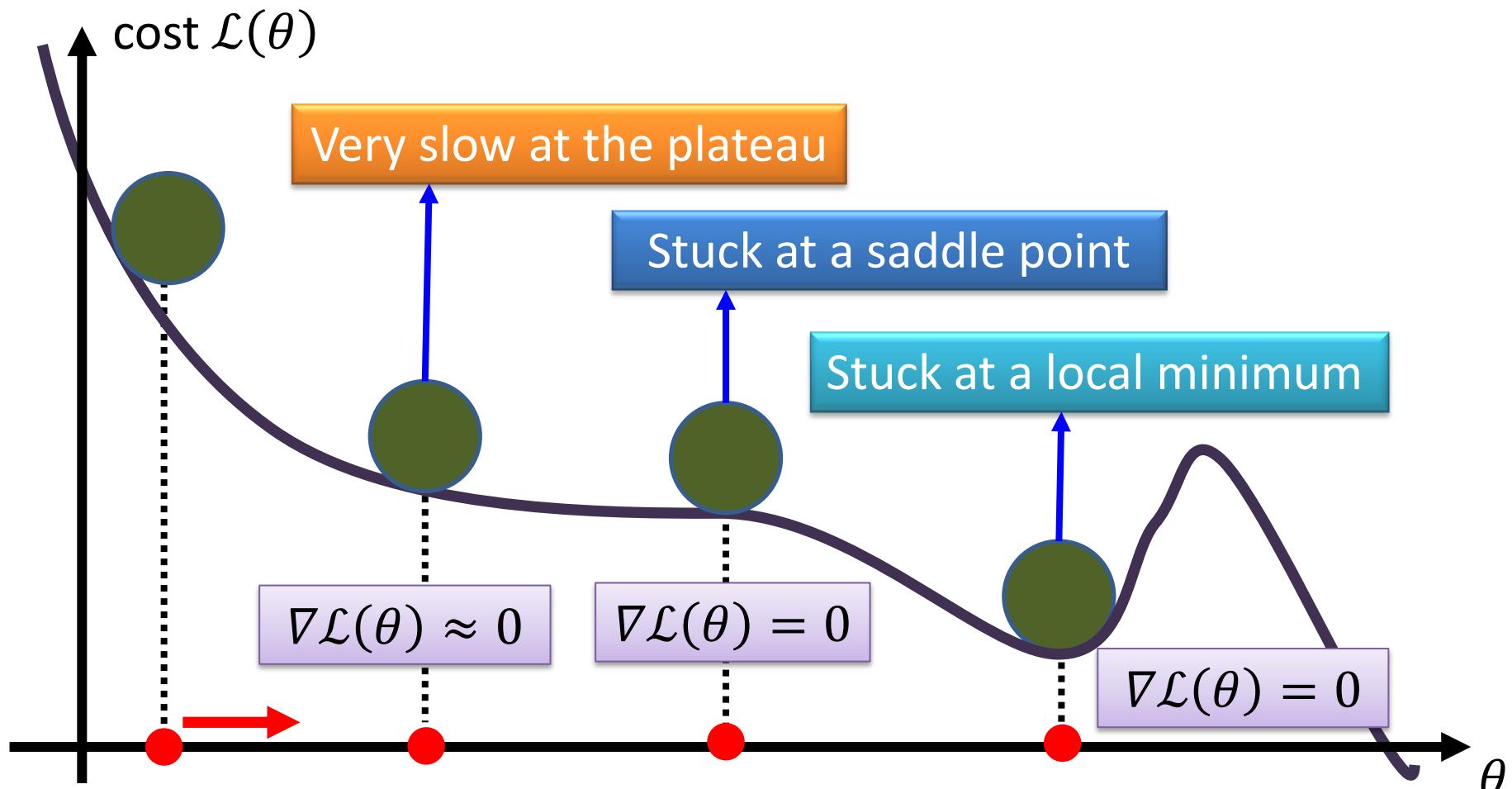
- SGD uses mini-batches that consist of a **single input example**
 - E.g., one image mini-batch
- Although this method is very fast, it may cause significant fluctuations in the loss function
 - Therefore, it is less commonly used, and mini-batch GD is preferred
- In most DL libraries, SGD typically means a mini-batch GD (with an option to add momentum)



Problems with Gradient Descent

Training Neural Networks

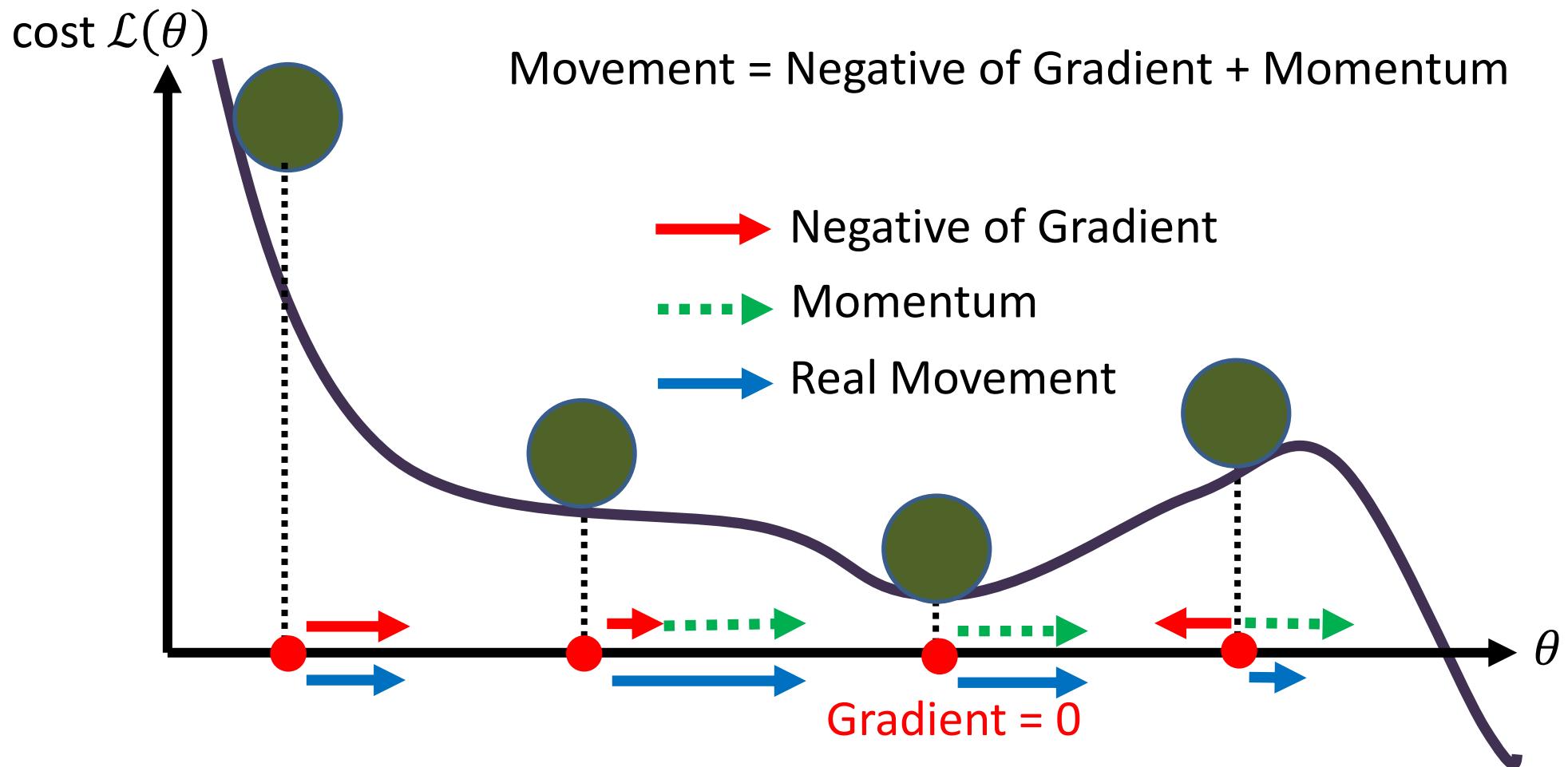
- Besides the local minima problem, the GD algorithm can be very slow at **plateaus**, and it can get stuck at **saddle points**



Gradient Descent with Momentum

Training Neural Networks

- *Gradient descent with momentum* uses the momentum of the gradient for parameter optimization



Gradient Descent with Momentum

Training Neural Networks

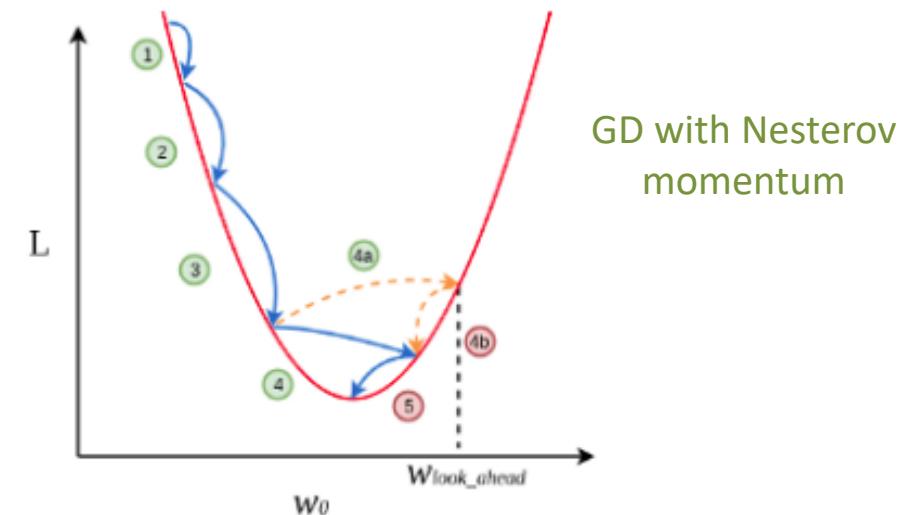
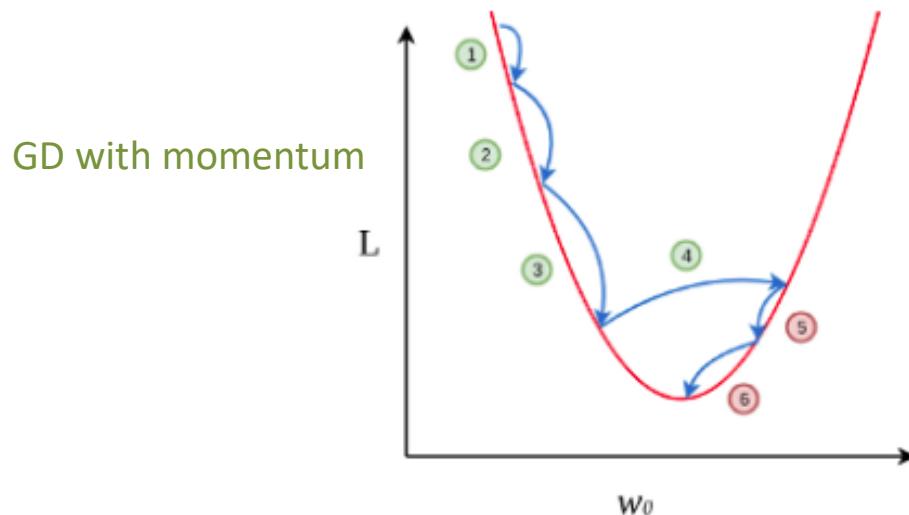
- Parameters update in **GD with momentum** at iteration t : $\theta^t = \theta^{t-1} - V^t$
 - Where: $V^t = \beta V^{t-1} + \alpha \nabla \mathcal{L}(\theta^{t-1})$
 - I.e., $\theta^t = \theta^{t-1} - \alpha \nabla \mathcal{L}(\theta^{t-1}) - \beta V^{t-1}$
- Compare to vanilla GD: $\theta^t = \theta^{t-1} - \alpha \nabla \mathcal{L}(\theta^{t-1})$
 - Where θ^{t-1} are the parameters from the previous iteration $t - 1$
- The term V^t is called **momentum**
 - This term accumulates the gradients from the past several steps, i.e.,
$$\begin{aligned}V^t &= \beta V^{t-1} + \alpha \nabla \mathcal{L}(\theta^{t-1}) \\&= \beta(\beta V^{t-2} + \alpha \nabla \mathcal{L}(\theta^{t-2})) + \alpha \nabla \mathcal{L}(\theta^{t-1}) \\&= \beta^2 V^{t-2} + \beta \alpha \nabla \mathcal{L}(\theta^{t-2}) + \alpha \nabla \mathcal{L}(\theta^{t-1}) \\&= \beta^3 V^{t-3} + \beta^2 \alpha \nabla \mathcal{L}(\theta^{t-3}) + \beta \alpha \nabla \mathcal{L}(\theta^{t-2}) + \alpha \nabla \mathcal{L}(\theta^{t-1})\end{aligned}$$
 - This term is analogous to a momentum of a heavy ball rolling down the hill
- The parameter β is referred to as a **coefficient of momentum**
 - A typical value of the parameter β is 0.9
- This method updates the parameters θ in the direction of the weighted average of the past gradients

Nesterov Accelerated Momentum

Training Neural Networks

- **Gradient descent with Nesterov accelerated momentum**

- Parameter update: $\theta^t = \theta^{t-1} - V^t$
 - Where: $V^t = \beta V^{t-1} + \alpha \nabla \mathcal{L}(\theta^{t-1} + \beta V^{t-1})$
- The term $\theta^{t-1} + \beta V^{t-1}$ allows to predict the position of the parameters in the next step (i.e., $\theta^t \approx \theta^{t-1} + \beta V^{t-1}$)
- The gradient is calculated with respect to the approximate future position of the parameters in the next iteration, θ^t , calculated at iteration $t - 1$



Adam

Training Neural Networks

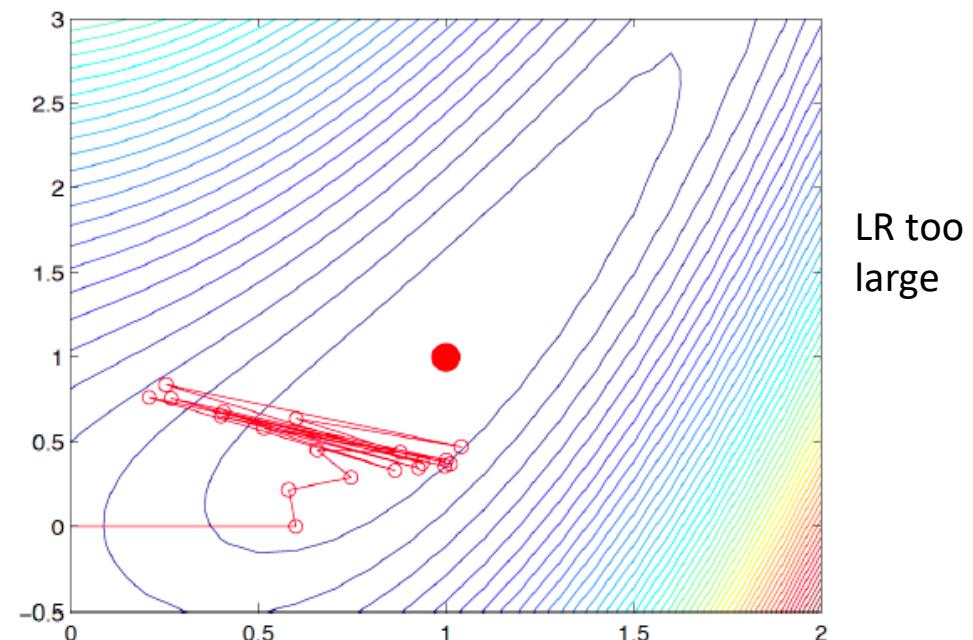
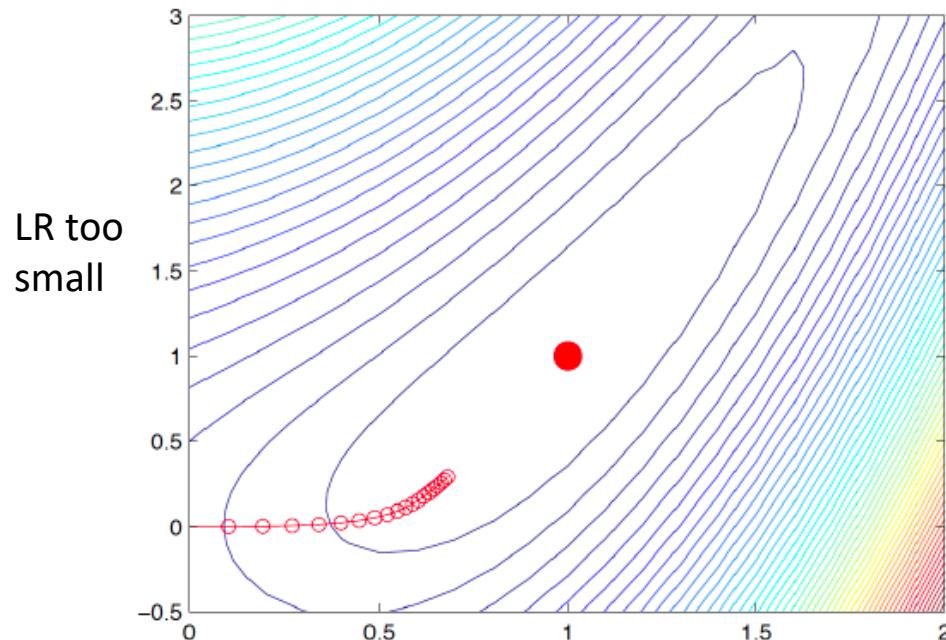
- **Adaptive Moment Estimation (Adam)**
 - Adam combines insights from the momentum optimizers that accumulate the values of past gradients, and it also introduces new terms based on the second moment of the gradient
 - Similar to GD with momentum, Adam computes a **weighted average of past gradients** (**first moment** of the gradient), i.e., $V^t = \beta_1 V^{t-1} + (1 - \beta_1) \nabla \mathcal{L}(\theta^{t-1})$
 - Adam also computes a **weighted average of past squared gradients** (**second moment** of the gradient), , i.e., $U^t = \beta_2 U^{t-1} + (1 - \beta_2) (\nabla \mathcal{L}(\theta^{t-1}))^2$
 - The parameter update is: $\theta^t = \theta^{t-1} - \alpha \frac{\hat{V}^t}{\sqrt{\hat{U}^t} + \epsilon}$
 - Where: $\hat{V}^t = \frac{V^t}{1-\beta_1}$ and $\hat{U}^t = \frac{U^t}{1-\beta_2}$
 - The proposed default values are $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$
- Other commonly used optimization methods include:
 - Adagrad, Adadelta, RMSprop, Nadam, etc.
 - Most commonly used optimizers nowadays are Adam and SGD with momentum

Learning Rate

Training Neural Networks

- **Learning rate**

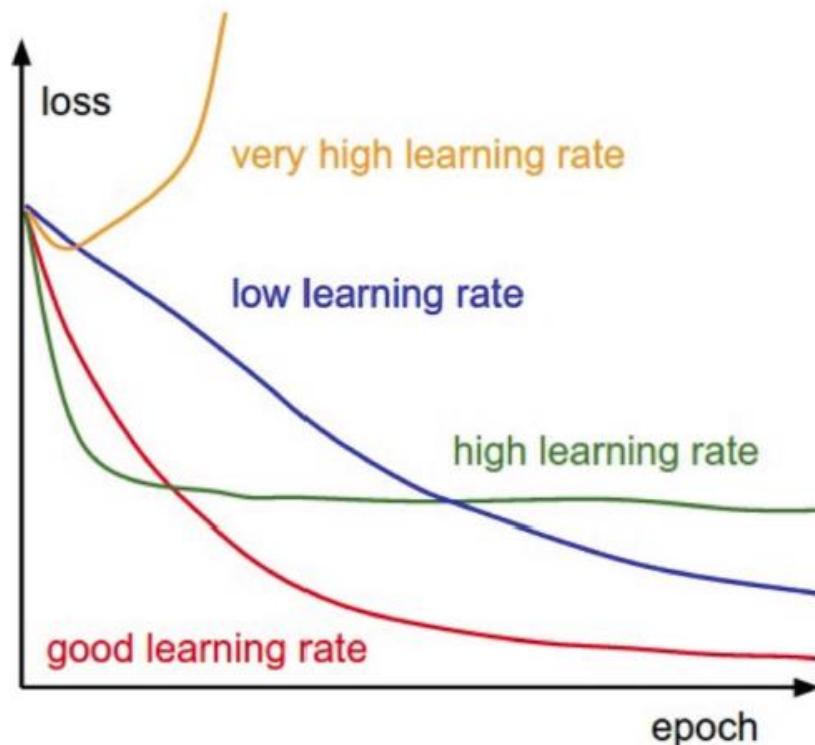
- The gradient tells us the direction in which the loss has the steepest rate of increase, but it does not tell us how far along the opposite direction we should step
- Choosing the learning rate (also called the **step size**) is one of the most important hyper-parameter settings for NN training



Learning Rate

Training Neural Networks

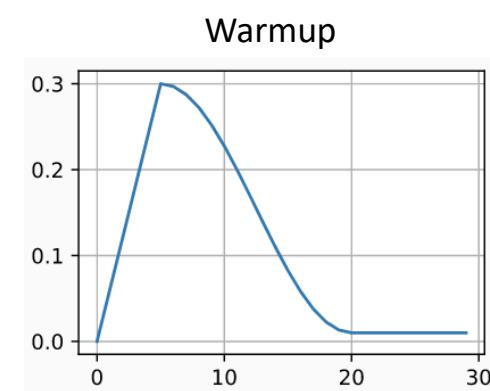
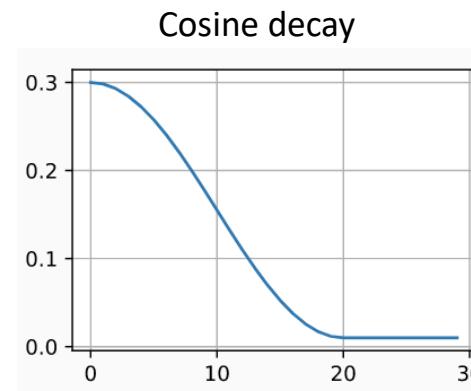
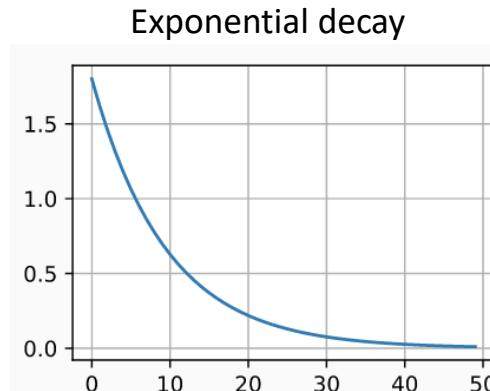
- Training loss for different learning rates
 - High learning rate: the loss increases or plateaus too quickly
 - Low learning rate: the loss decreases too slowly (takes many epochs to reach a solution)



Learning Rate Scheduling

Training Neural Networks

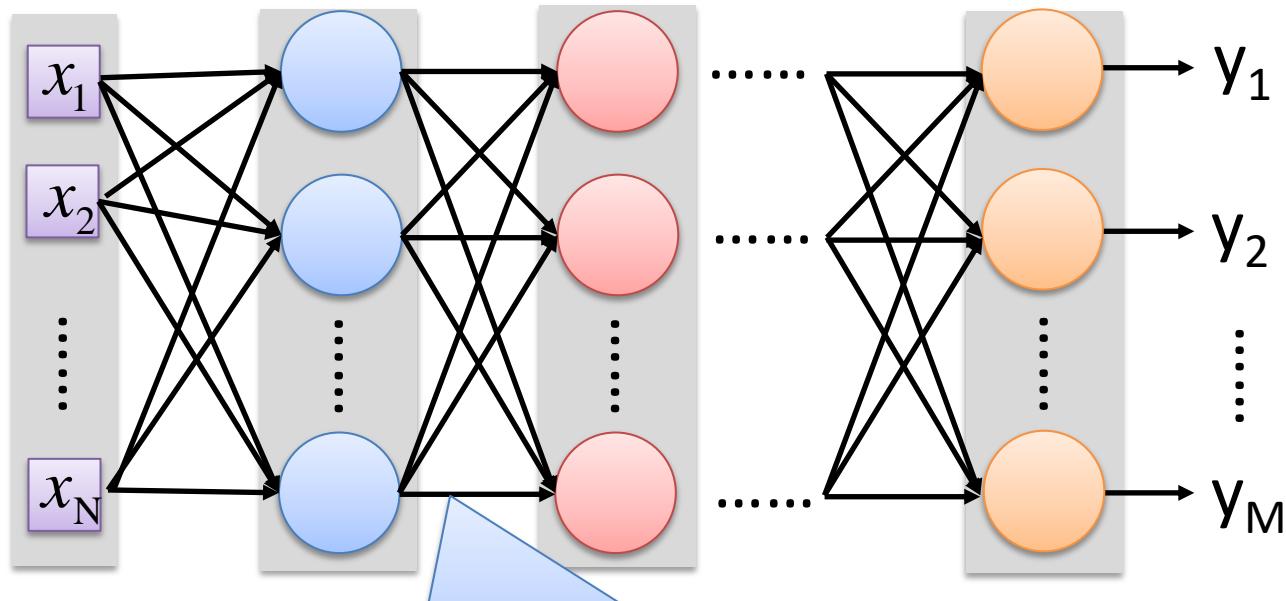
- **Learning rate scheduling** is applied to change the values of the learning rate during the training
 - **Annealing** is reducing the learning rate over time (a.k.a. learning rate decay)
 - Approach 1: reduce the learning rate by some factor **every few epochs**
 - Typical values: reduce the learning rate by a half every 5 epochs, or divide by 10 every 20 epochs
 - Approach 2: **exponential** or **cosine decay** gradually reduce the learning rate over time
 - Approach 3: reduce the learning rate by a constant (e.g., by half) whenever the **validation loss stops improving**
 - In TensorFlow: `tf.keras.callbacks.ReduceLROnPlateau()`
 - » Monitor: validation loss, factor: 0.1 (i.e., divide by 10), patience: 10 (how many epochs to wait before applying it), Minimum learning rate: 1e-6 (when to stop)
 - **Warmup** is gradually increasing the learning rate initially, and afterward let it cool down until the end of the training



Vanishing Gradient Problem

Training Neural Networks

- In some cases, during training, the gradients can become either very small (vanishing gradients) or very large (exploding gradients)
 - They result in very small or very large update of the parameters
 - Solutions: change learning rate, ReLU activations, regularization, LSTM units in RNNs

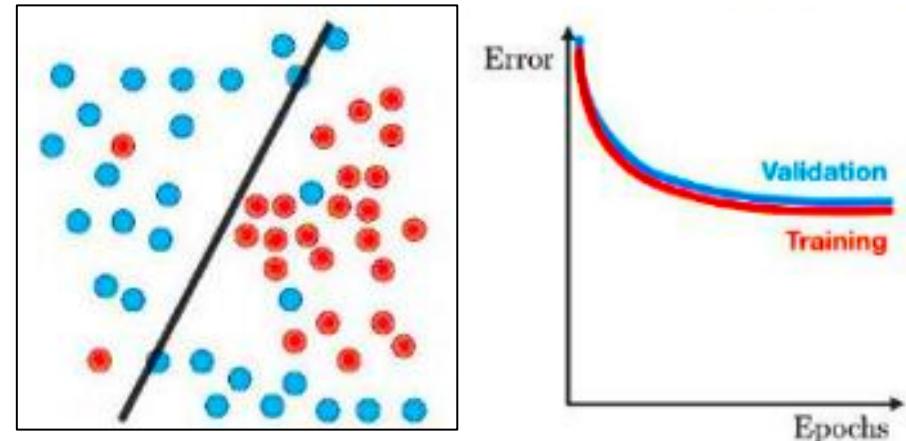


Generalization

Generalization

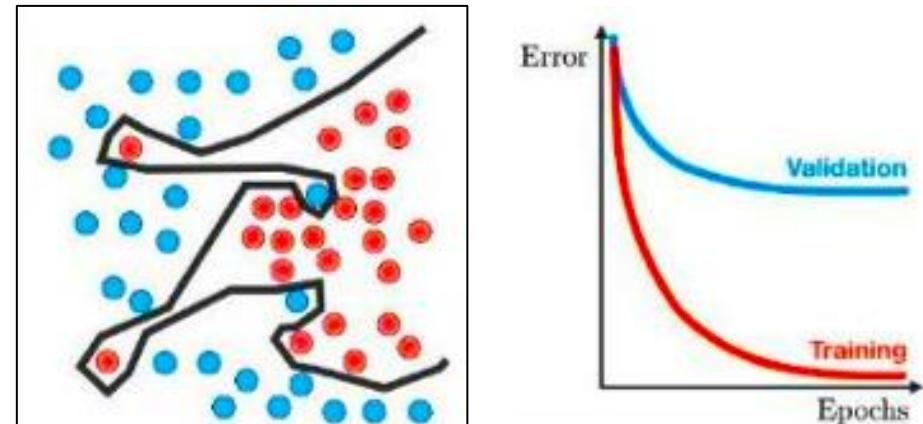
- ***Underfitting***

- The model is too “simple” to represent all the relevant class characteristics
- E.g., model with too few parameters
- Produces high error on the training set and high error on the validation set



- ***Overfitting***

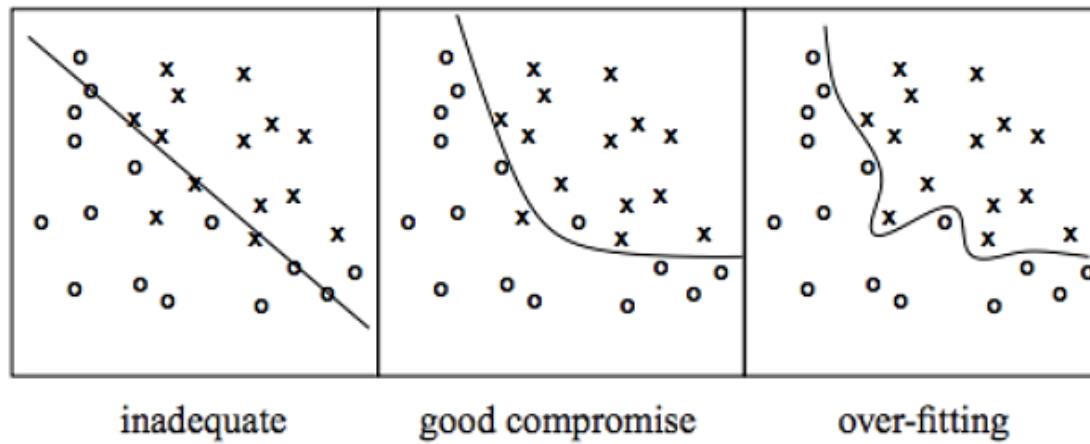
- The model is too “complex” and fits irrelevant characteristics (noise) in the data
- E.g., model with too many parameters
- Produces low error on the training error and high error on the validation set



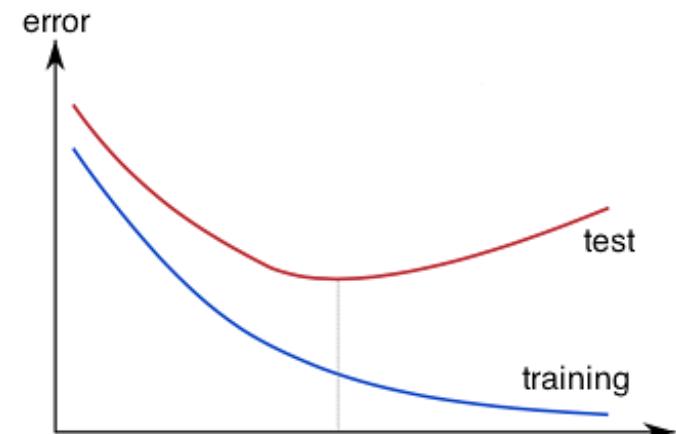
Overfitting

Generalization

- Overfitting – a model with high capacity fits the noise in the data instead of the underlying relationship



- The model may fit the training data very well, but fails to **generalize** to new examples (test or validation data)



Regularization: Weight Decay

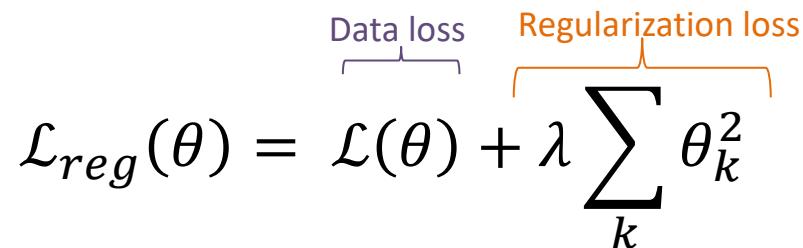
Regularization

- *ℓ_2 weight decay*

- A regularization term that penalizes large weights is added to the loss function

$$\mathcal{L}_{reg}(\theta) = \mathcal{L}(\theta) + \lambda \sum_k \theta_k^2$$

Data loss Regularization loss

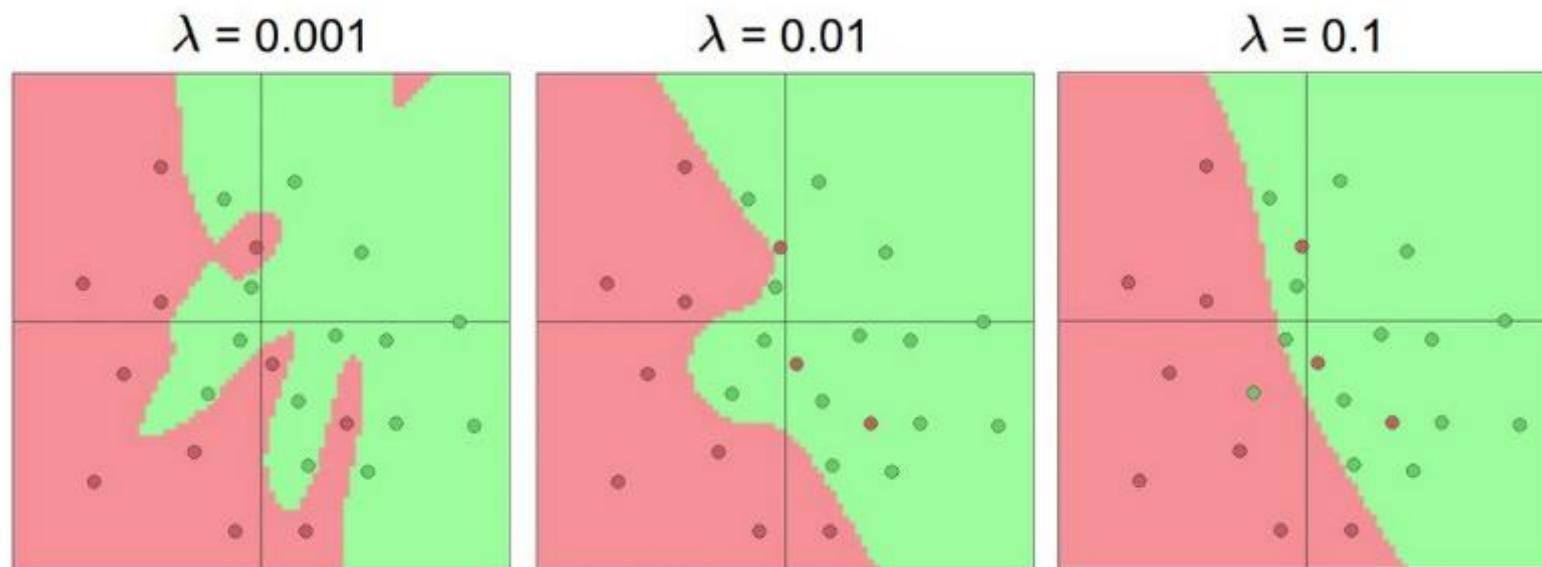


- For every weight in the network, we add the regularization term to the loss value
 - During gradient descent parameter update, every weight is decayed linearly toward zero
 - The **weight decay coefficient λ** determines how dominant the regularization is during the gradient computation

Regularization: Weight Decay

Regularization

- Effect of the decay coefficient λ
 - Large weight decay coefficient \rightarrow penalty for weights with large values



Regularization: Weight Decay

Regularization

- **ℓ_1 weight decay**

- The regularization term is based on the ℓ_1 norm of the weights

$$\mathcal{L}_{reg}(\theta) = \mathcal{L}(\theta) + \lambda \sum_k |\theta_k|$$

- ℓ_1 weight decay is less common with NN
 - Often performs worse than ℓ_2 weight decay
 - It is also possible to combine ℓ_1 and ℓ_2 regularization
 - Called **elastic net regularization**

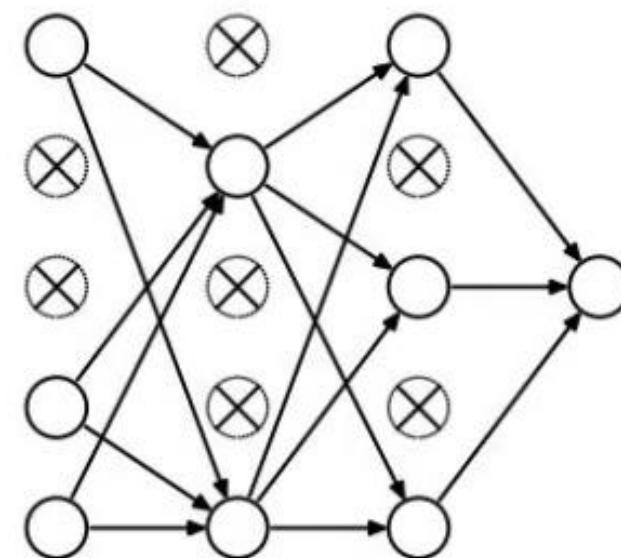
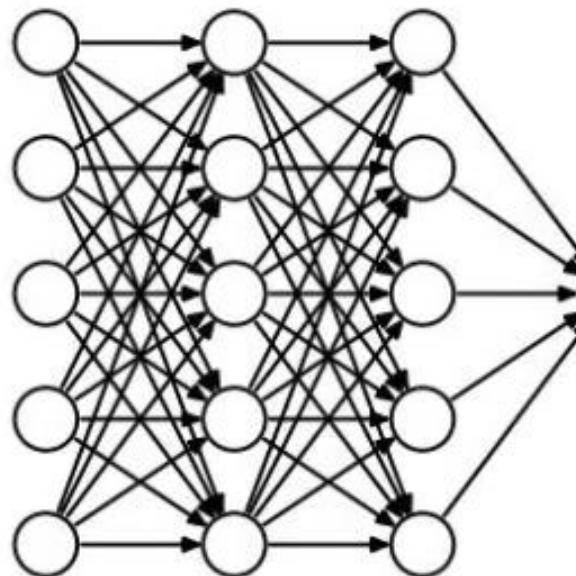
$$\mathcal{L}_{reg}(\theta) = \mathcal{L}(\theta) + \lambda_1 \sum_k |\theta_k| + \lambda_2 \sum_k \theta_k^2$$

Regularization: Dropout

Regularization

- **Dropout**

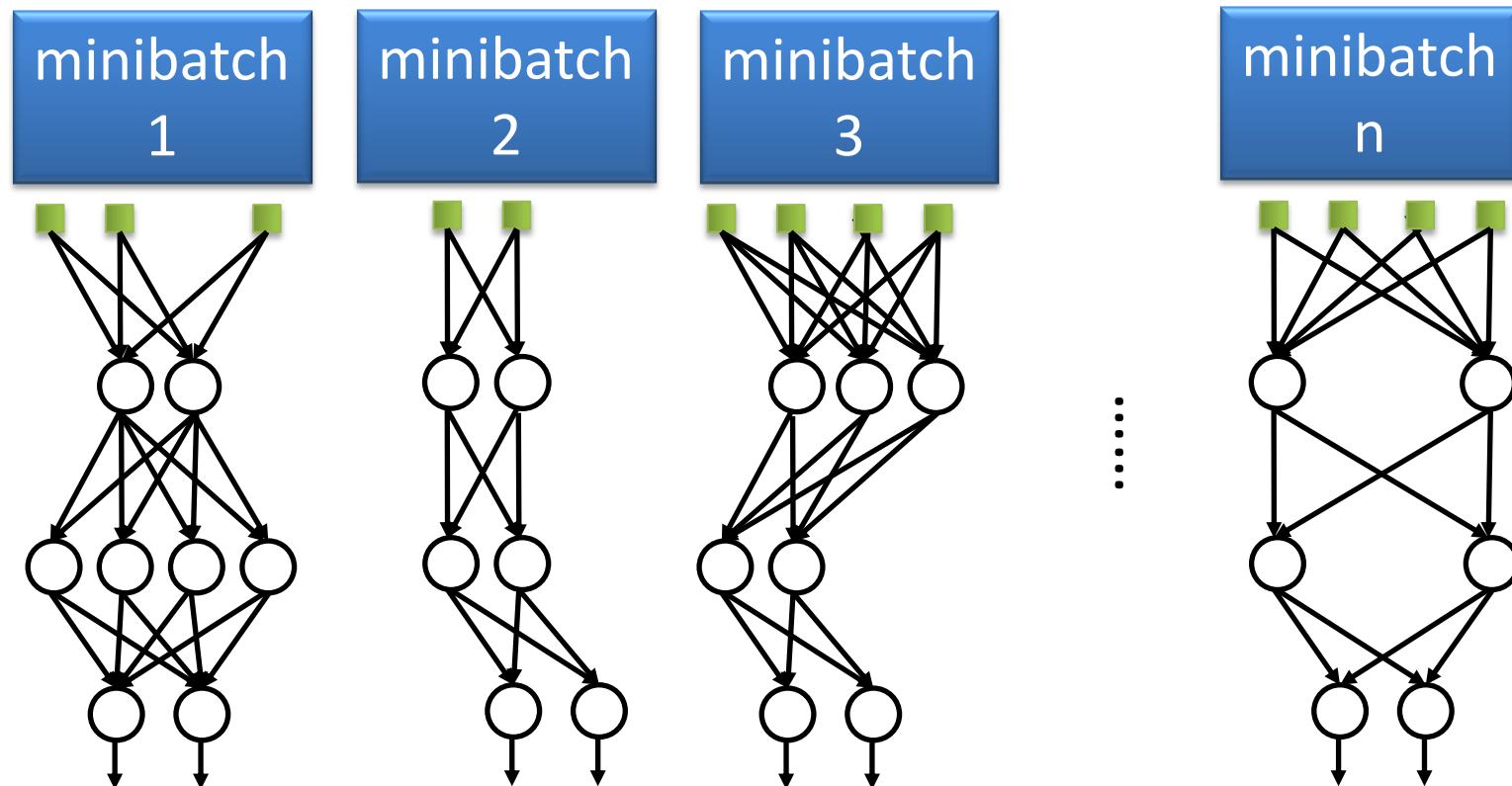
- Randomly drop units (along with their connections) during training
- Each unit is retained with a fixed **dropout rate p** , independent of other units
- The hyper-parameter p needs to be chosen (tuned)
 - Often, between 20% and 50% of the units are dropped



Regularization: Dropout

Regularization

- Dropout is a kind of ensemble learning
 - Using one mini-batch to train one network with a slightly different architecture

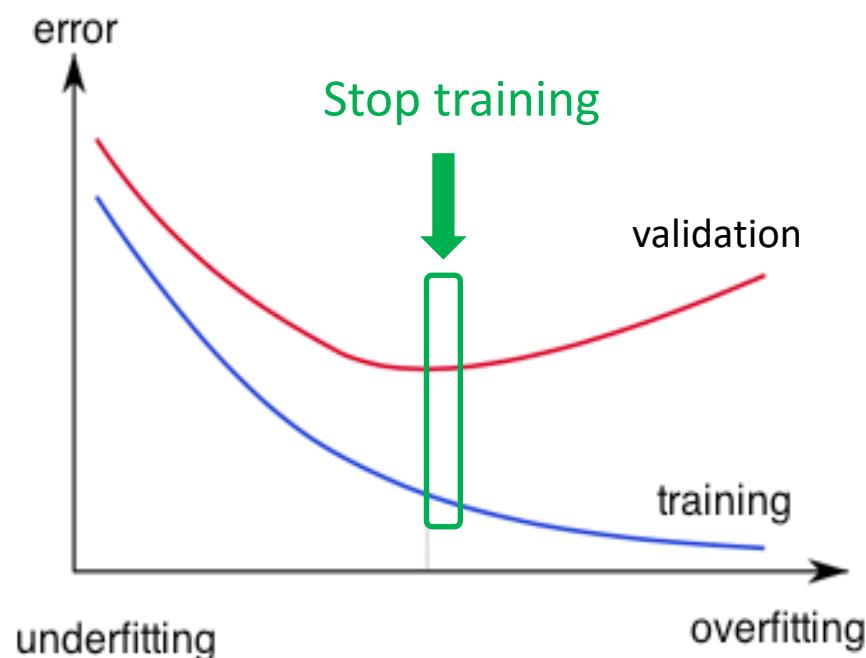


Regularization: Early Stopping

Regularization

- *Early-stopping*

- During model training, use a **validation set**
 - E.g., validation/train ratio of about 25% to 75%
- Stop when the validation accuracy (or loss) has not improved after n epochs
 - The parameter n is called **patience**



Batch Normalization

Regularization

- **Batch normalization layers** act similar to the data preprocessing steps mentioned earlier
 - They calculate the mean μ and variance σ of a batch of input data, and normalize the data x to a zero mean and unit variance
 - I.e., $\hat{x} = \frac{x-\mu}{\sigma}$
- **BatchNorm layers** alleviate the problems of proper initialization of the parameters and hyper-parameters
 - Result in faster convergence training, allow larger learning rates
 - Reduce the internal covariate shift
- BatchNorm layers are inserted immediately after convolutional layers or fully-connected layers, and before activation layers
 - They are very common with convolutional NNs

Hyper-parameter Tuning

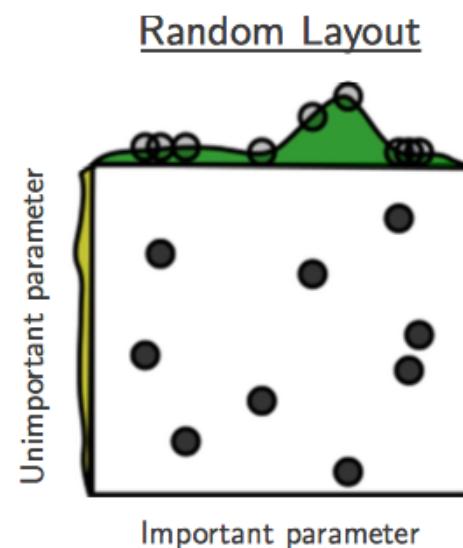
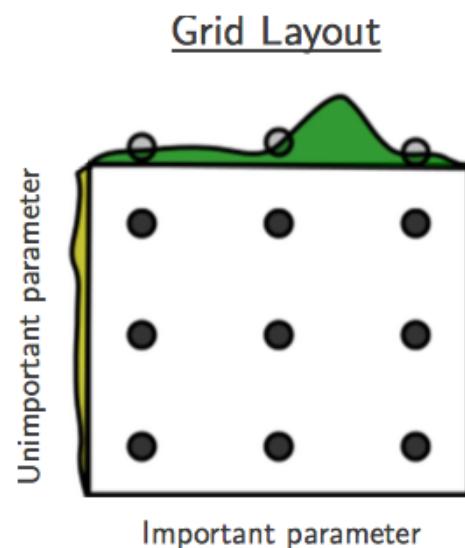
Hyper-parameter Tuning

- Training NNs can involve setting many *hyper-parameters*
- The most common hyper-parameters include:
 - Number of layers, and number of neurons per layer
 - Initial learning rate
 - Learning rate decay schedule (e.g., decay constant)
 - Optimizer type
- Other hyper-parameters may include:
 - Regularization parameters (ℓ_2 penalty, dropout rate)
 - Batch size
 - Activation functions
 - Loss function
- Hyper-parameter tuning can be time-consuming for larger NNs

Hyper-parameter Tuning

Hyper-parameter Tuning

- Grid search
 - Check all values in a range with a step value
- Random search
 - Randomly sample values for the parameter
 - Often preferred to grid search
- Bayesian hyper-parameter optimization
 - Is an active area of research



k-Fold Cross-Validation

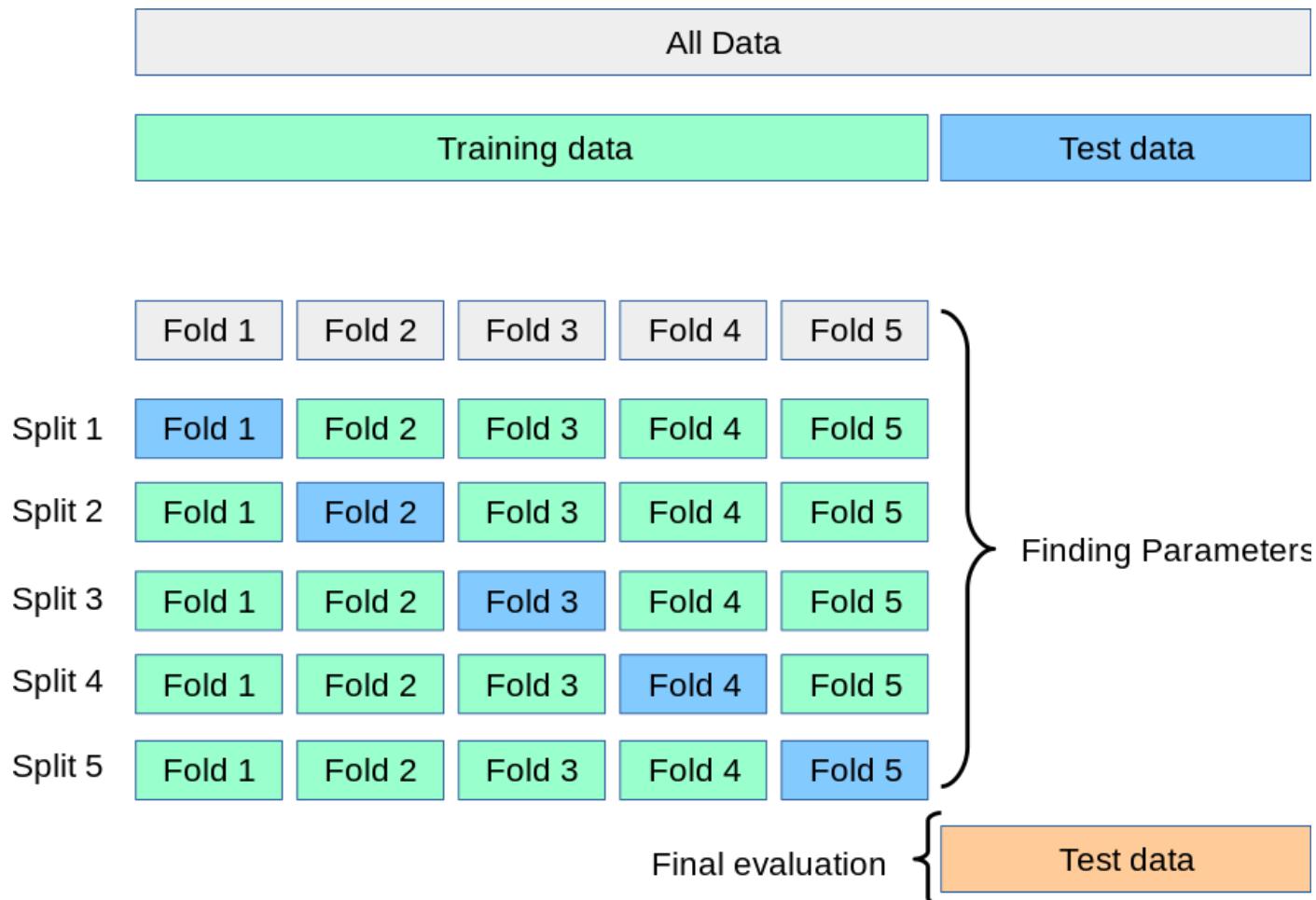
k-Fold Cross-Validation

- Using ***k-fold cross-validation*** for hyper-parameter tuning is common when the size of the training data is small
 - It also leads to a better and less noisy estimate of the model performance by averaging the results across several folds
- E.g., 5-fold cross-validation (see the figure on the next slide)
 1. Split the train data into 5 equal folds
 2. First use folds 2-5 for training and fold 1 for validation
 3. Repeat by using fold 2 for validation, then fold 3, fold 4, and fold 5
 4. Average the results over the 5 runs (for reporting purposes)
 5. Once the best hyper-parameters are determined, evaluate the model on the test data

k -Fold Cross-Validation

k-Fold Cross-Validation

- Illustration of a 5-fold cross-validation



Ensemble Learning

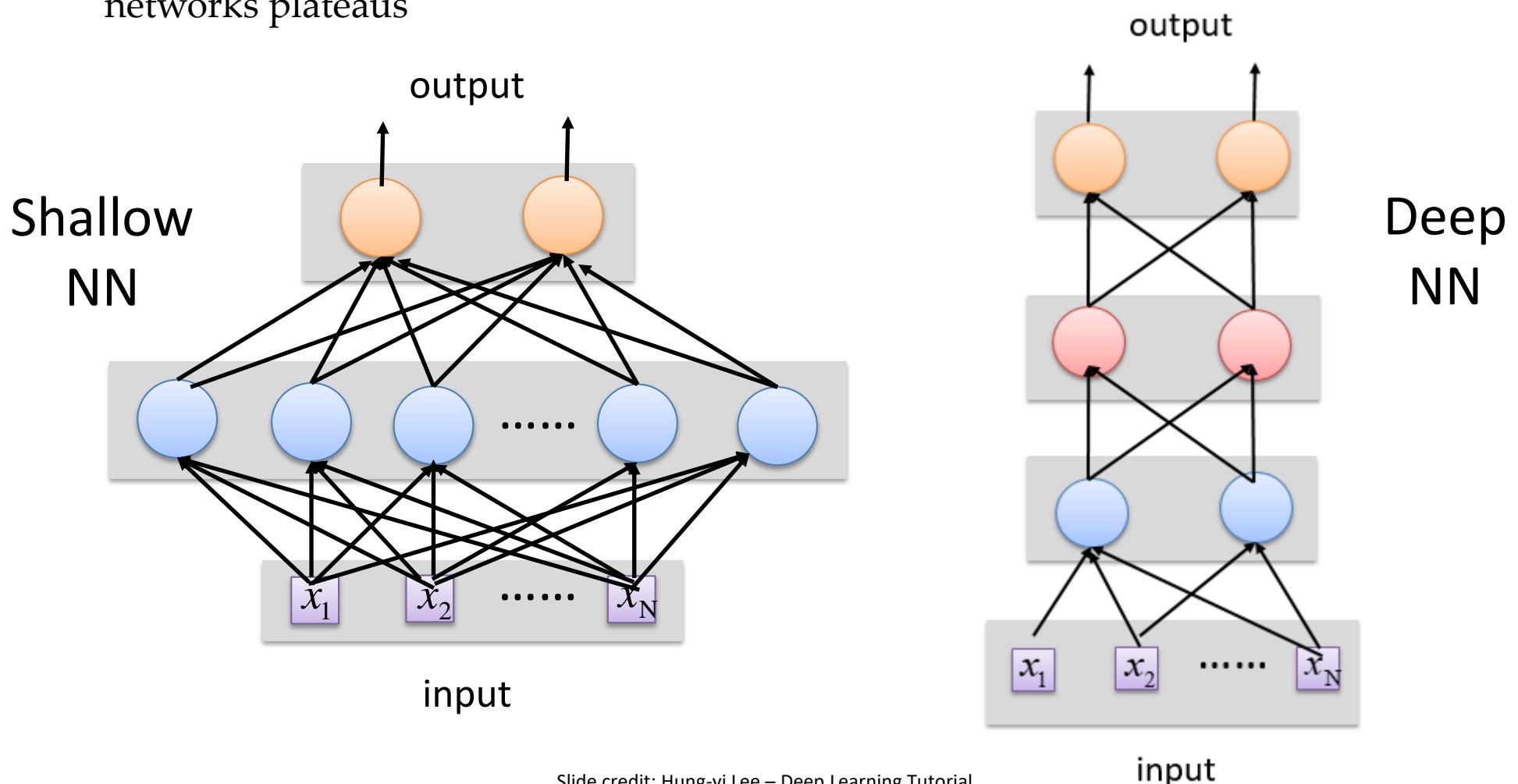
Ensemble Learning

- **Ensemble learning** is training multiple classifiers separately and combining their predictions
 - Ensemble learning often outperforms individual classifiers
 - Better results obtained with higher model variety in the ensemble
 - **Bagging** (**b**ootstrap **a**ggregating)
 - Randomly draw subsets from the training set (i.e., bootstrap samples)
 - Train separate classifiers on each subset of the training set
 - Perform classification based on the average vote of all classifiers
 - **Boosting**
 - Train a classifier, and apply weights on the training set (apply **higher weights on misclassified examples**, focus on “hard examples”)
 - Train new classifier, reweight training set according to prediction error
 - Repeat
 - Perform classification based on weighted vote of the classifiers

Deep vs Shallow Networks

Deep vs Shallow Networks

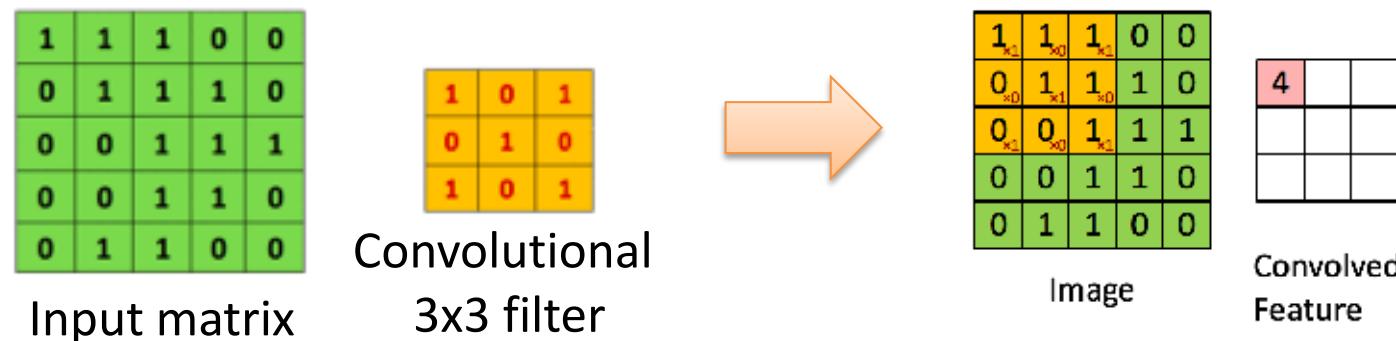
- **Deeper networks** perform better than shallow networks
 - But only up to some limit: after a certain number of layers, the performance of deeper networks plateaus



Convolutional Neural Networks (CNNs)

Convolutional Neural Networks

- *Convolutional neural networks* (CNNs) were primarily designed for image data
- CNNs use a **convolutional operator** for extracting data features
 - Allows **parameter sharing**
 - Efficient to train
 - Have **less parameters** than NNs with fully-connected layers
- CNNs are **robust to spatial translations** of objects in images
- A convolutional filter slides (i.e., convolves) across the image



Convolutional Neural Networks (CNNs)

Convolutional Neural Networks

- When the convolutional filters are scanned over the image, they capture useful features
 - E.g., edge detection by convolutions



$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$



Input Image

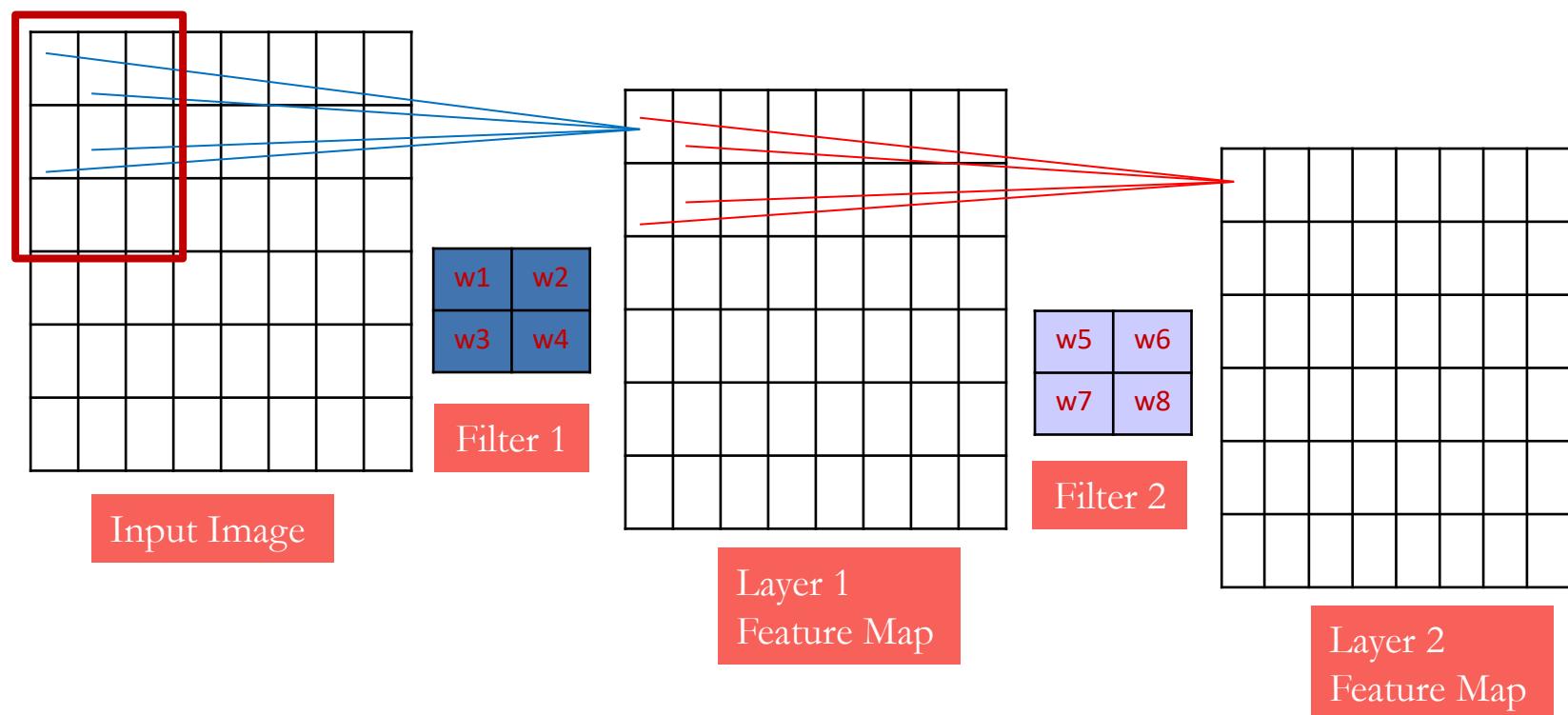


Convolved Image

Convolutional Neural Networks (CNNs)

Convolutional Neural Networks

- In CNNs, hidden units in a layer are only connected to a small region of the layer before it (called local **receptive field**)
 - The depth of each **feature map** corresponds to the number of convolutional filters used at each layer



Convolutional Neural Networks (CNNs)

Convolutional Neural Networks

- **Max pooling**: reports the maximum output within a rectangular neighborhood
- **Average pooling**: reports the average output of a rectangular neighborhood
- Pooling layers reduce the spatial size of the feature maps
 - Reduce the number of parameters, prevent overfitting

1	3	5	3
4	2	3	1
3	1	1	3
0	1	0	4

Input Matrix

MaxPool with a 2×2 filter with stride of 2

4	5
3	4

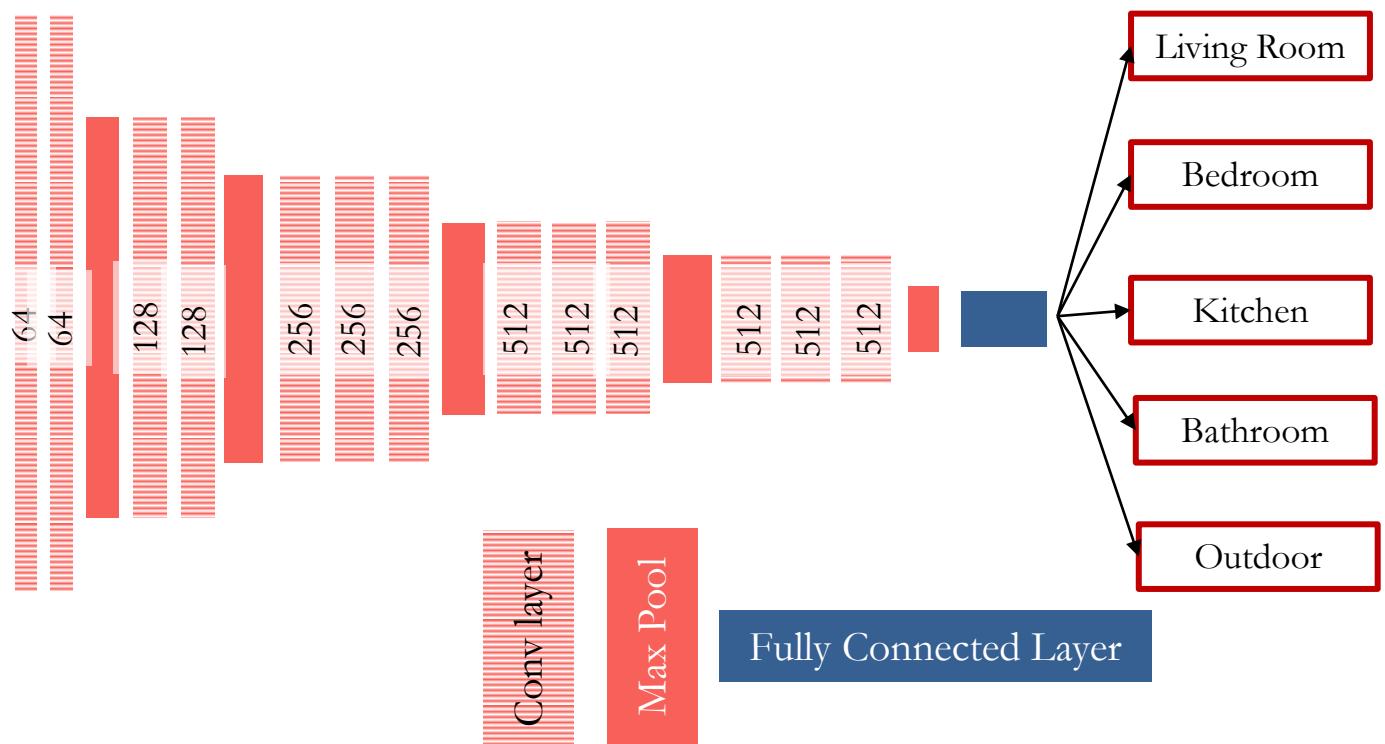
Output Matrix

Convolutional Neural Networks (CNNs)

Convolutional Neural Networks

- Feature extraction architecture

- After 2 convolutional layers, a max-pooling layer reduces the size of the feature maps (typically by 2)
- A fully convolutional and a softmax layers are added last to perform classification

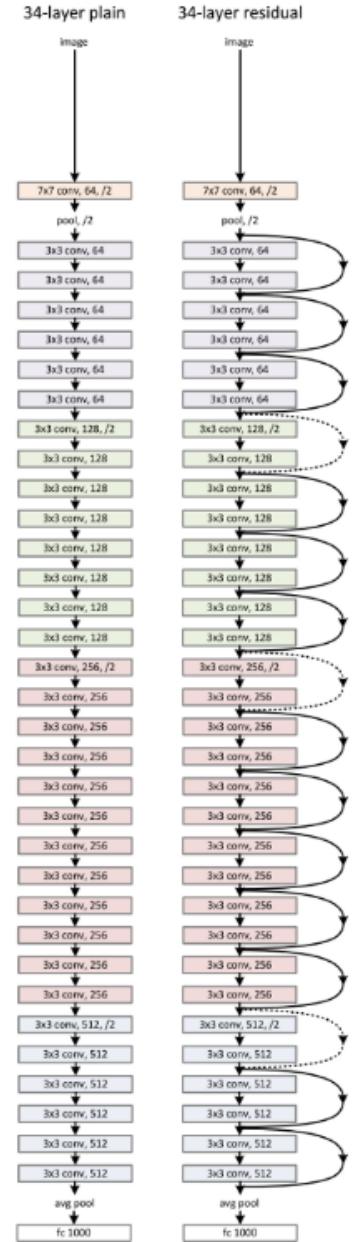
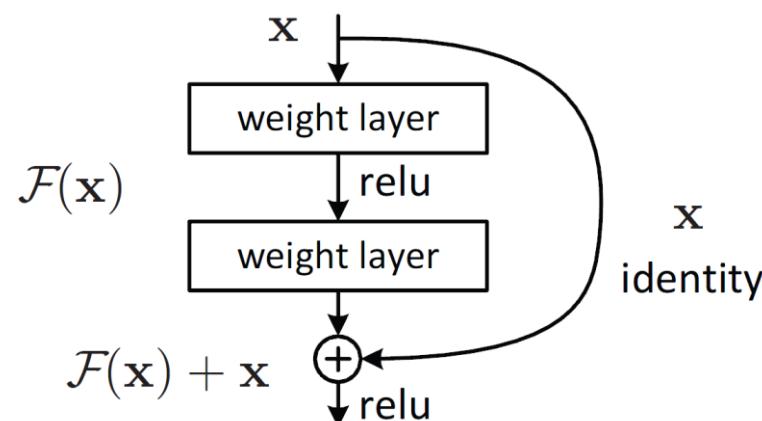


Residual CNNs

Convolutional Neural Networks

- **Residual networks** (ResNets)

- Introduce “identity” **skip connections**
 - Layer inputs are propagated and added to the layer output
 - Mitigate the problem of vanishing gradients during training
 - Allow training very deep NN (with over 1,000 layers)
- Several ResNet variants exist: 18, 34, 50, 101, 152, and 200 layers
- Are used as base models of other state-of-the-art NNs
 - Other similar models: ResNeXT, DenseNet



Recurrent Neural Networks (RNNs)

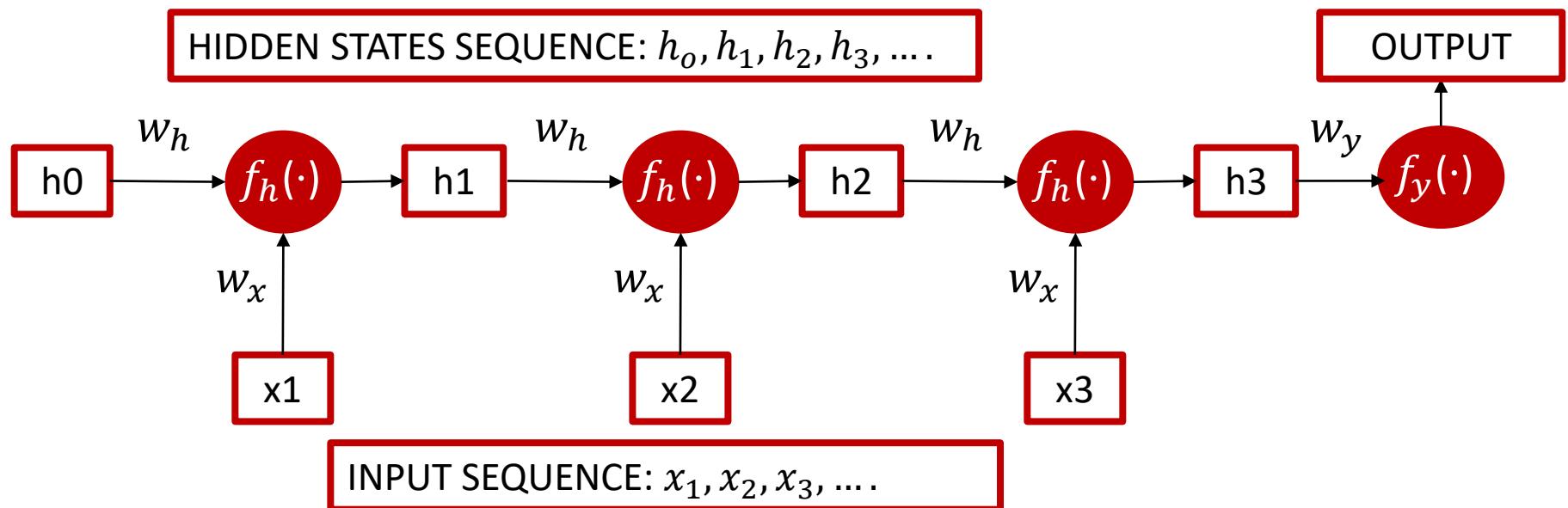
Recurrent Neural Networks

- **Recurrent NNs** are used for modeling **sequential data** and data with varying length of inputs and outputs
 - Videos, text, speech, DNA sequences, human skeletal data
- RNNs introduce recurrent connections between the neurons
 - This allows processing sequential data one element at a time by selectively passing information across a sequence
 - Memory of the previous inputs is stored in the model's internal state and affect the model predictions
 - Can capture correlations in sequential data
- RNNs use **backpropagation-through-time** for training
- RNNs are more sensitive to the vanishing gradient problem than CNNs

Recurrent Neural Networks (RNNs)

Recurrent Neural Networks

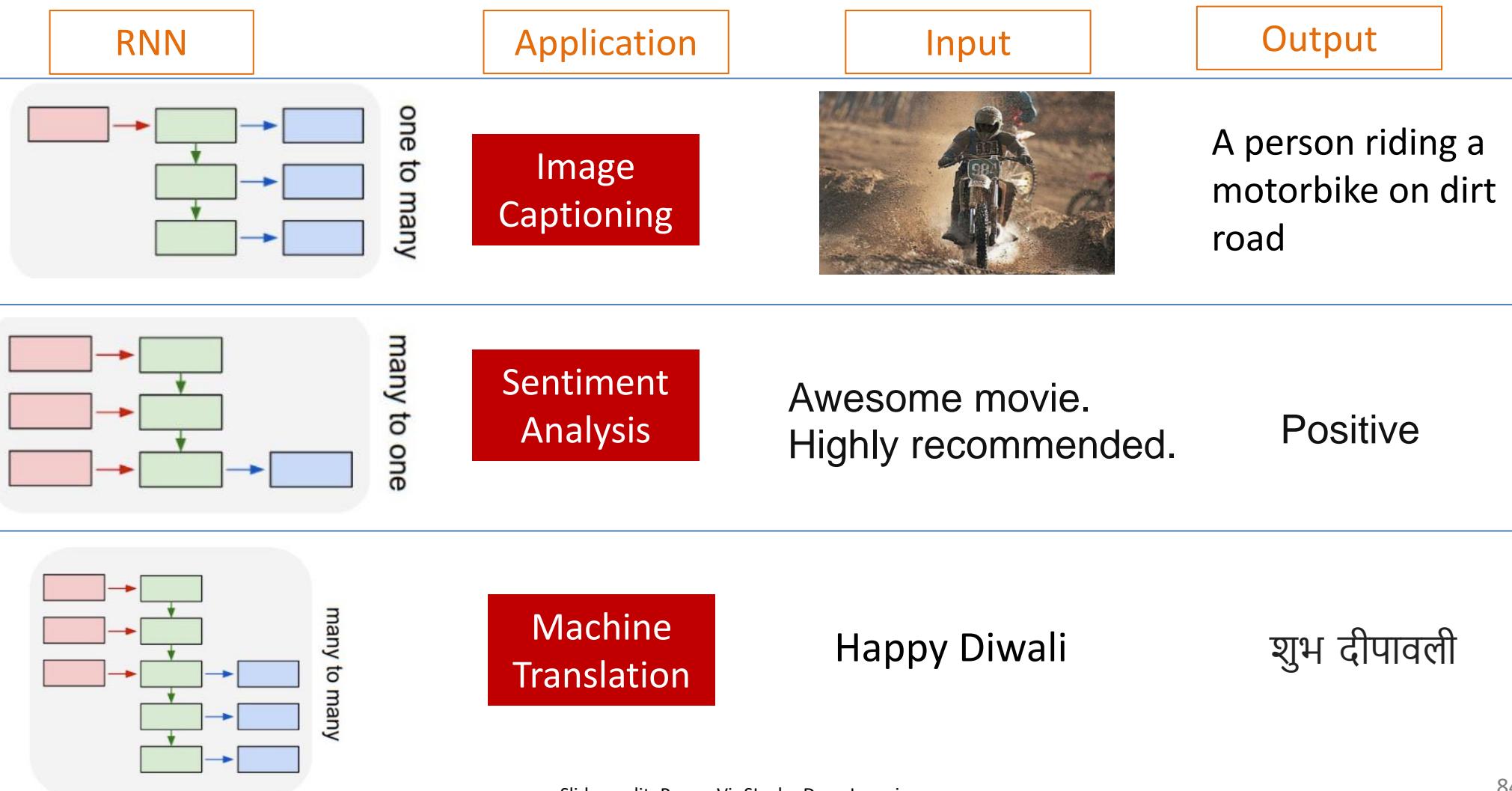
- RNN use same set of weights w_h and w_x **across all time steps**
 - A sequence of **hidden states** $\{h_0, h_1, h_2, h_3, \dots\}$ is learned, which represents the memory of the network
 - The hidden state at step t , $h(t)$, is calculated based on the previous hidden state $h(t - 1)$ and the input at the current step $x(t)$, i.e., $h(t) = f_h(w_h * h(t - 1) + w_x * x(t))$
 - The function $f_h(\cdot)$ is a nonlinear activation function, e.g., ReLU or tanh
- RNN shown rolled over time



Recurrent Neural Networks (RNNs)

Recurrent Neural Networks

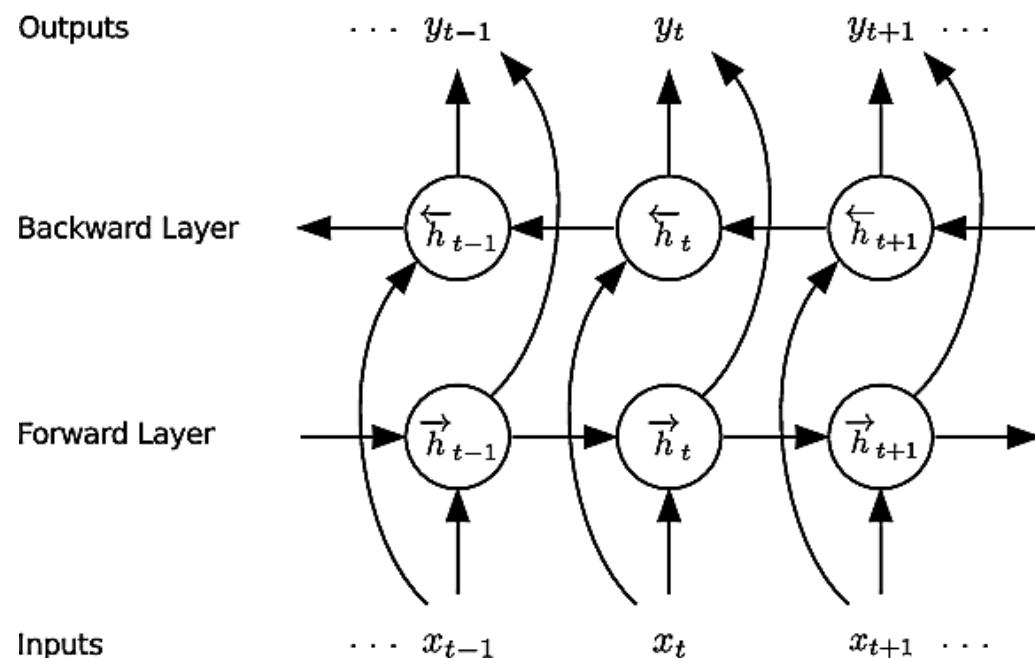
- RNNs can have one of many inputs and one of many outputs



Bidirectional RNNs

Recurrent Neural Networks

- **Bidirectional RNNs** incorporate both forward and backward passes through sequential data
 - The output may not only depend on the previous elements in the sequence, but also on future elements in the sequence
 - It resembles two RNNs stacked on top of each other



$$\vec{h}_t = \sigma(\overrightarrow{W}^{(hh)}\overleftarrow{h}_{t-1} + \overrightarrow{W}^{(hx)}x_t)$$

$$\overleftarrow{h}_t = \sigma(\overleftarrow{W}^{(hh)}\overleftarrow{h}_{t+1} + \overleftarrow{W}^{(hx)}x_t)$$

$$y_t = f([\vec{h}_t; \overleftarrow{h}_t])$$

Outputs both past and future elements

LSTM Networks

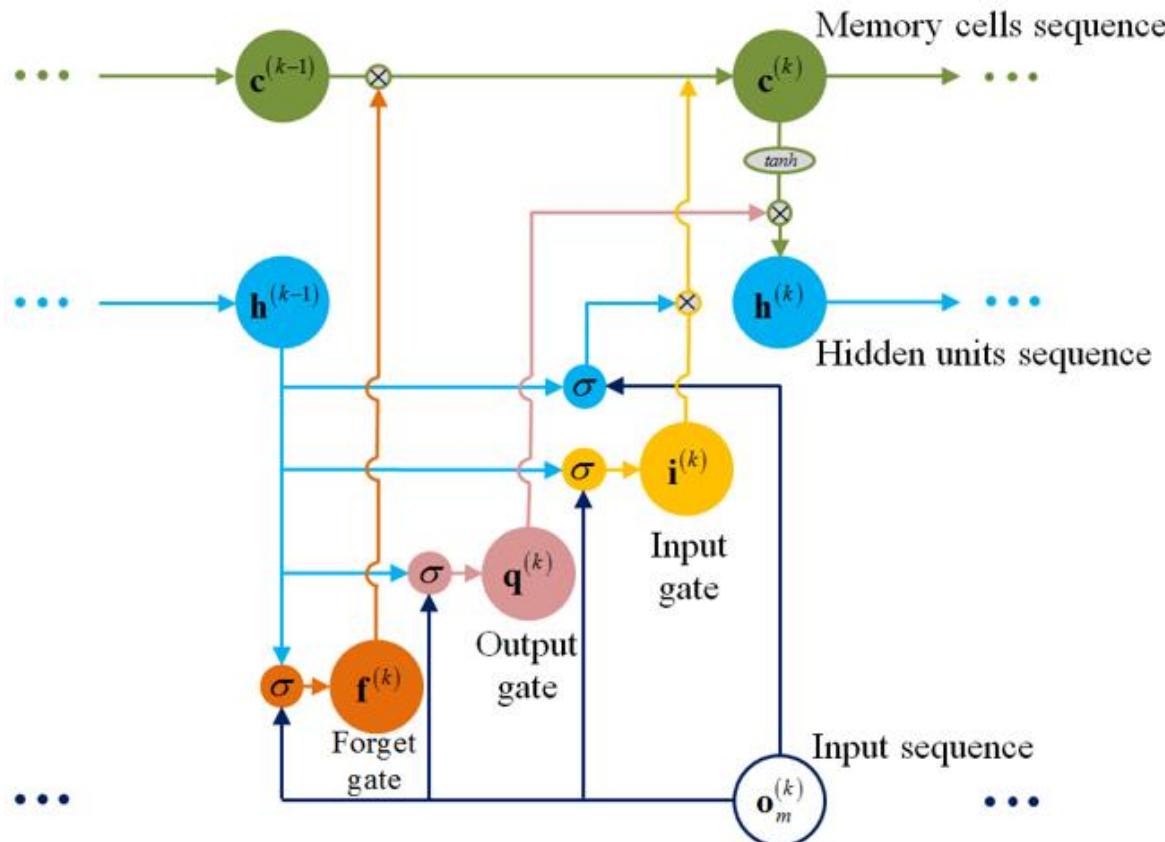
Recurrent Neural Networks

- **Long Short-Term Memory (LSTM)** networks are a variant of RNNs
- LSTM mitigates the vanishing/exploding gradient problem
 - Solution: a **Memory Cell**, updated at each step in the sequence
- Three gates control the flow of information to and from the Memory Cell
 - **Input Gate**: protects the current step from irrelevant inputs
 - **Output Gate**: prevents current step from passing irrelevant information to later steps
 - **Forget Gate**: limits information passed from one cell to the next
- Most modern RNN models use either LSTM units or other more advanced types of recurrent units (e.g., GRU units)

LSTM Networks

Recurrent Neural Networks

- LSTM cell
 - Input gate, output gate, forget gate, memory cell
 - LSTM can learn long-term correlations within data sequences



$$\begin{aligned}
 i^{(k)} &= \sigma(W_{oi} o_m^{(k)} + W_{hi} h^{(k-1)} + b_i) \\
 f^{(k)} &= \sigma(W_{of} o_m^{(k)} + W_{hf} h^{(k-1)} + b_f) \\
 q^{(k)} &= \sigma(W_{oq} o_m^{(k)} + W_{hq} h^{(k-1)} + b_q) \\
 c^{(k)} &= f^{(k)} c^{(k-1)} + i^{(k)} \sigma(W_{oc} o_m^{(k)} + W_{hc} h^{(k-1)} + b_c) \\
 h^{(k)} &= q^{(k)} \tanh(c^{(k)})
 \end{aligned}$$

References

1. Hung-yi Lee – Deep Learning Tutorial
2. Ismini Lourentzou – Introduction to Deep Learning
3. CS231n Convolutional Neural Networks for Visual Recognition (Stanford CS course) ([link](#))
4. James Hays, Brown – Machine Learning Overview
5. Param Vir Singh, Shunyuan Zhang, Nikhil Malik – Deep Learning
6. Sebastian Ruder – An Overview of Gradient Descent Optimization Algorithms ([link](#))

ANSWER THE FOLLOWING QUESTIONS

Every question has weight equals 7 marks (else question No.7 has weight equals 8 marks)

Solve the following classification problem with the Perceptron rule. Apply each input vector in order, for as many repetitions as it takes to ensure that the problem is solved.

$$\left\{ p_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, t_1 = 0 \right\} \left\{ p_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, t_2 = 1 \right\} \left\{ p_3 = \begin{bmatrix} -2 \\ 2 \end{bmatrix}, t_3 = 0 \right\} \left\{ p_4 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, t_4 = 1 \right\}$$

Use the initial weights and bias: $W(0) = [0 \ 0]$, $b(0) = 0$, Learning rate $\alpha = 1$,

2 Consider the following linear associator Figure (1):

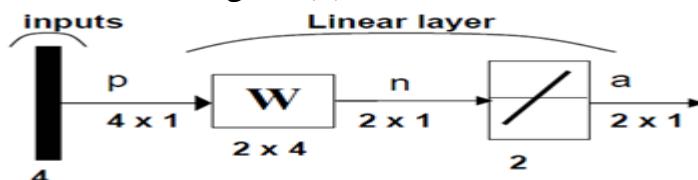


Figure (1)

Suppose that the prototype input/output vectors are

$$\left\{ p_1 = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}, t_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \right\} \left\{ p_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix}, t_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}$$

- I- Use the Hebb rule to find the appropriate weight matrix and design a linear associator network for these patterns.
- II- Use the Pseudo-inverse rule to find the appropriate weight matrix for this linear associator.

3 Suppose that you have the following input/target pairs:

$$\left\{ p_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, t_1 = 1 \right\} \left\{ p_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, t_2 = -1 \right\}.$$

These patterns occur with equal probability, and they are used to train an ADALINE network with no bias. Train the network using the LMS algorithm, with the initial guess set to zero and a learning rate $\alpha = 0.25$.

4 Consider the instar network shown in Figure (2). The training sequence for this network will consist of the following inputs:

$$\left\{ p^0(1) = 0, p(1) = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right\}, \left\{ p^0(2) = 0, p(2) = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right\},$$

These two sets of inputs are repeatedly presented to the network until the weight matrix converges. Perform the first four iterations of the instar rule, with learning rate $\alpha = 0.5$. Assume that the initial matrix is set to all zeros.

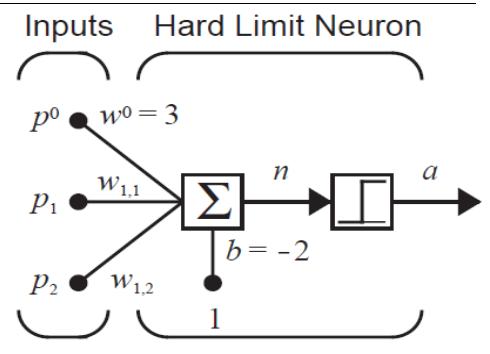


Figure (2)

- 5- Consider the two-layer network in Figure (3). with the following input and target: $\{p_1 = 1, t_1 = 2\}$. The initial weights and biases are given by
 $W^1(0) = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, W^2(0) = \begin{bmatrix} -1 & 1 \end{bmatrix}, b^1(0) = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, b^2(0) = [3]$

- i. Apply the input to the network and make one pass forward through the network to compute the output and the error.
ii. Compute the sensitivities by backpropagating through the network.

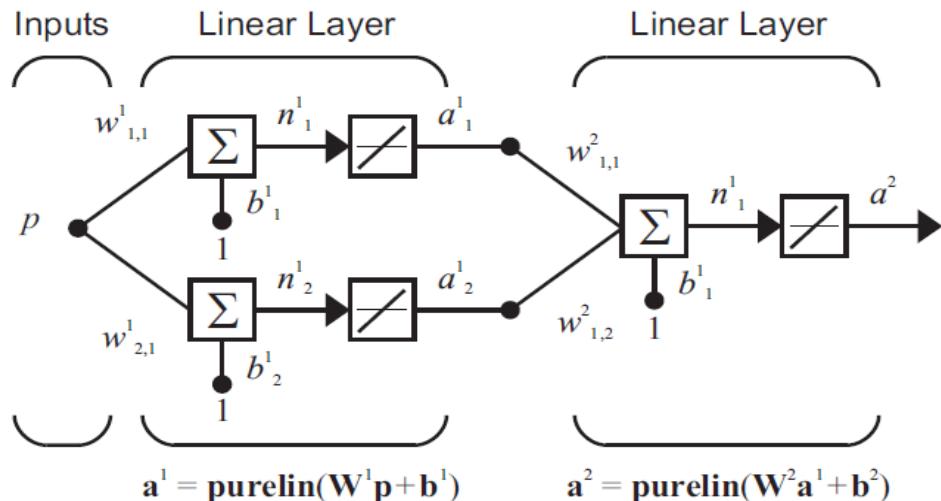


Figure (3)

- 6- An SOFM network with three inputs and two cluster units is to be trained using the four training vectors: $[0.8 \ 0.7 \ 0.4]$, $[0.6 \ 0.9 \ 0.9]$, $[0.3 \ 0.4 \ 0.1]$, $[0.1 \ 0.1 \ 0.2]$ and initial weights Unit 1= $[.5 \ .6 \ .8]$ and Unit 2= $[.4 \ .2 \ .5]$. Learning rate = 0.5. Calculate the weight changes during the first cycle through the data, taking the training vectors in the given order.

- 7- Consider the network of Figure (4) using for function approximation. The network transfer functions are chosen to be $f^1(n) = n^2$, $f^2(n) = n$. Assume that the training set consisting set consists of $\{(p_1 = [1], t_1 = [1])\}, \{(p_2 = [2], t_2 = [2])\}$ and that the parameters are initialized to $W^1 = [1]$, $b^1 = [0]$, $W^2 = [2]$, $b^2 = [1]$. Find the Jacobian matrix for the first step of the **Levenberg-Marquardt** method.

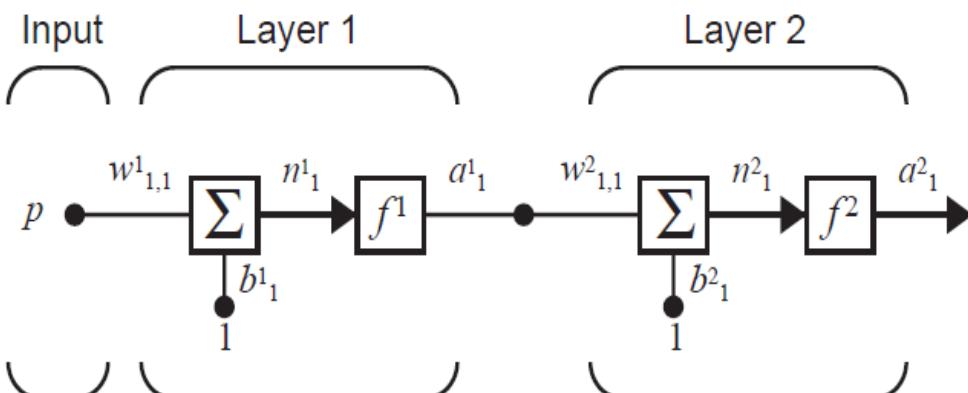


Figure (4)

GOOD LUCK!

ANSWER THE FOLLOWING QUESTIONS

**Q1-
4 marks** Consider the apple and orange problem, Train an ADALINE network using the LMS learning algorithm to distinguish between them, the input/output prototype vectors are

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}, \mathbf{t}_1 = [-1] \right\}, \left\{ \mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, \mathbf{t}_2 = [1] \right\}$$

Use $\alpha = 0.2$ and start with all the weights set to zero. *Perform at least two complete iterations.* .0.2

Answer:

**Q2-
4 marks**

Consider the classification problem

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

I- Design a single-neuron perceptron to solve this problem.

II- Test your solution with all your input vectors

III- Classify the following input vectors with your solution

$$p_5 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

Use the initial weights and bias:

$$W(0) = \begin{bmatrix} 0 & 0 \end{bmatrix} \quad b(0) = 0.$$

Q3- 4 marks

Consider the following linear associator Figure (1):

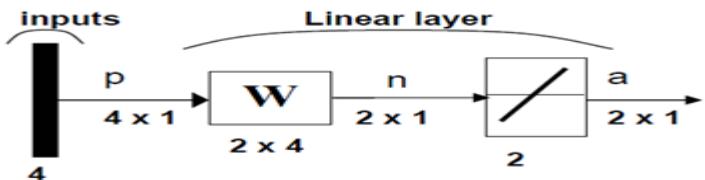


Figure (1)

Suppose that the prototype input/output vectors are

$$\left\{ p_1 = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}, t_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \right\} \left\{ p_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix}, t_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}$$

Use the Hebb rule to find the appropriate weight matrix and design a linear associator network for these patterns. Use the Pseudo-inverse rule to find the appropriate weight matrix for this linear associator.

Answer:

**Q4-
4 marks**

Can you represent the following boolean function with a single logistic threshold unit (i.e., a single unit from a neural network)? If yes, show the weights. If not, explain why not.

A	B	$f(A,B)$
1	1	0
0	0	0
1	0	1
0	1	0

Answer:

A two-layer neural network is to have four inputs and six outputs. The range of the outputs is to be continuous between 0 and 1. What can you tell about the network architecture? Specifically:

- a- How many neurons are required in each layer?
 - b- What are the dimensions of the first-layer and second-layer weight matrices?
 - c- What kinds of transfer functions can be used in each layer?
 - d- Are biases required in either layer?

Q5- MCQ**4 marks**

- Choose the correct answer:**
- I- A perceptron is guaranteed to perfectly learn a given linearly separable function within a finite number of training steps. **(True or False)**
- II- A single perceptron can compute the XOR function.. **(True or False)**
- III- Typically, Adalines produce better results for new (untrained) inputs than do perceptrons. **(True or False)**
- IV- A 4-input neuron has weights 1, 2, 3 and 4. The transfer function is linear with the constant of proportionality being equal to 2. The inputs are 4, 10, 5 and 20 respectively.
The output will be:
A. 238. B. 76. C. 119
- V- Which of the following is true?
A. On average, neural networks have higher computational rates than conventional computers.
B. Neural networks learn by example.
C. Neural networks mimic the way the human brain works.
- VI- What are the advantages of neural networks over conventional computers?
i- They have the ability to learn by example.
ii- They are more fault tolerant.
iii- They are more suited for real time operation due to their high 'computational' rates
A. (i) and (ii) are true
B. (i) and (iii) are true
C. all of them are true
- VII- What is classification?
A. Deciding which features to use in a pattern recognition problem.
B. Deciding which class an input pattern belongs to.
C. Deciding which type of neural network to use.
- VIII- What is generalization?
A. The ability of a pattern recognition system to approximate the desired output values for pattern vectors which are not in the test set.
B. The ability of a pattern recognition system to approximate the desired output values for pattern vectors which are not in the training set.
C. The ability of a pattern recognition system to extrapolate on pattern vectors which are not in the training set.
D. The ability of a pattern recognition system to interpolate on pattern vectors which are not in the test set.
- IX- Which of the following types of learning can used for training artificial neural networks?
A. Supervised learning.
B. Unsupervised learning.
C. Reinforcement learning.
D. All of the above answers.
E. None of the above answers.
- X- Which of the following statements is the best description of Hebb's learning rule?
A. "If a particular input stimulus is always active when a neuron fires then its weight should be increased."
B. "If a stimulus acts repeatedly at the same time as a response then a connection will form between the neurons involved. Later, the stimulus alone is sufficient to activate the response."
C. "The connection strengths of the neurons involved are modified to reduce the error between the desired and actual outputs of the system."

GOOD LUCK!