Artificial Neural Nets (ANNs)
[Connectionist]

Wang Houfeng
Institute of Computational Linguistics
Peking University

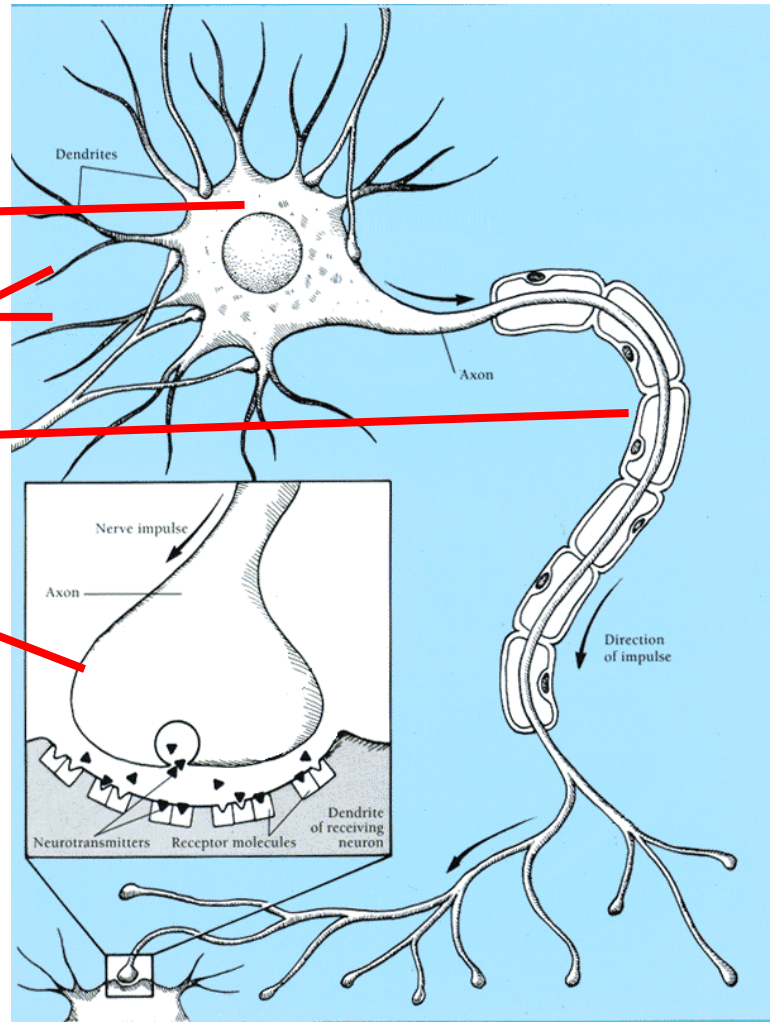
Outline

- **Perceptron**
- Perceptron Learning
- Sigmoid Unit
- MultiClass
- Multi-layer networks
- Backpropagation learning
- Others

Real Neuron

- Cell structures
  - Cell body
  - Dendrites(树突)
  - Axon (轴突)
  - Synaptic terminals
    （突触）

Synapse:the connective point between Neurons



Biological Neural Systems

- Neuron switching time : 10^{-3} secs
- Number of neurons in the human brain: ~10^{10}
- Connections (synapses) per neuron : ~10^{4}–10^{5}
- High degree of parallel computation
- Distributed representations

Definitions of Artificial Neural Networks (ANNs)

- Definition: "… a system composed of many simple processing elements operating in parallel whose function is determined by network structure, connection strengths, and the processing performed at computing elements or nodes." - DARPA (1988)
- Properties of ANNs
  - Many neuron-like threshold switching units
  - Many weighted interconnections among units
  - Highly parallel, distributed process
  - Learning by adaptation of the connection weights

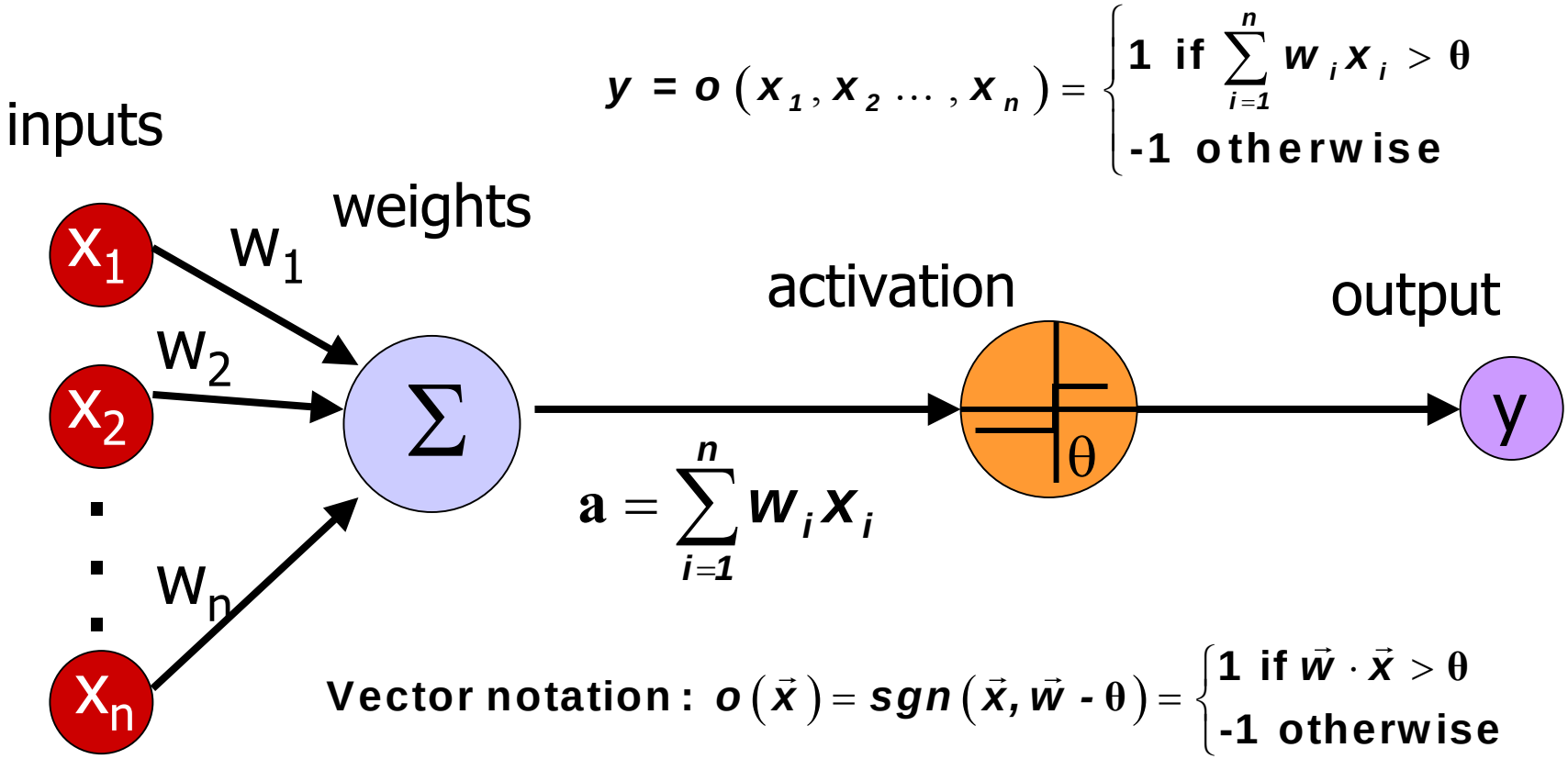
When to Consider Neural Networks

- Input: High-Dimensional and Discrete or Real-Valued
  - Conversion of symbolic data to quantitative (numerical) representations possible
- Output: Discrete or Real Vector-Valued
- Data: Possibly Noisy
- Target Function: Unknown Form
- Result: Humans do not need to interpret the results (black box model)

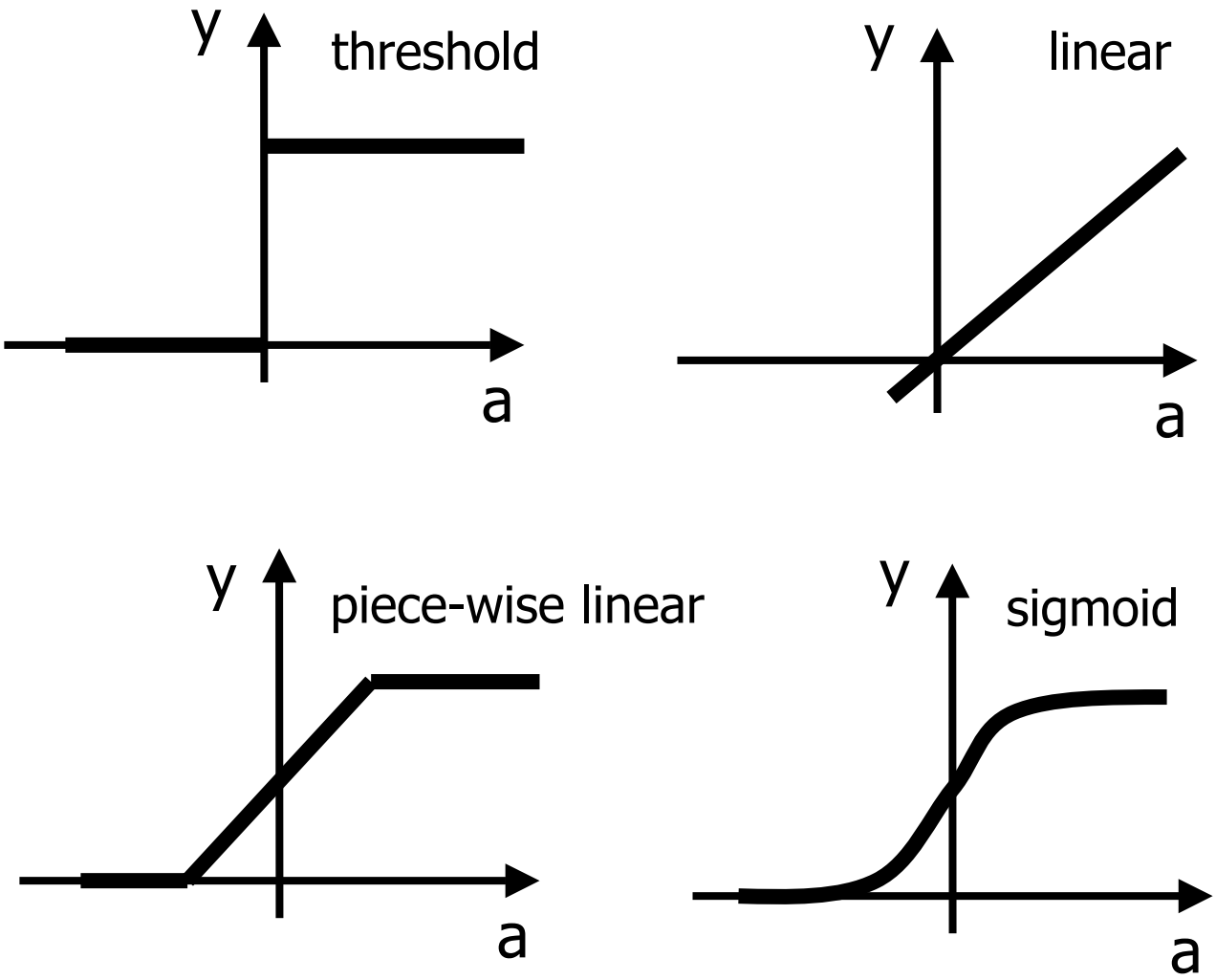
The Perceptron

- Perceptron: Single Neuron Model
  - Linear Threshold Unit (LTU) or Linear Threshold Gate (LTG)
  - Net input to unit: defined as linear combination $\mathbf{a} = \sum_{i=1}^{n} \boldsymbol{w}_i \boldsymbol{x}_i$
  - Output of unit: threshold (activation) function on net input (threshold θ)
- Perceptron Networks
  - Neuron is modeled using a unit connected by weighted links w_i to other units
  - Multi-Layer Perceptron (MLP)

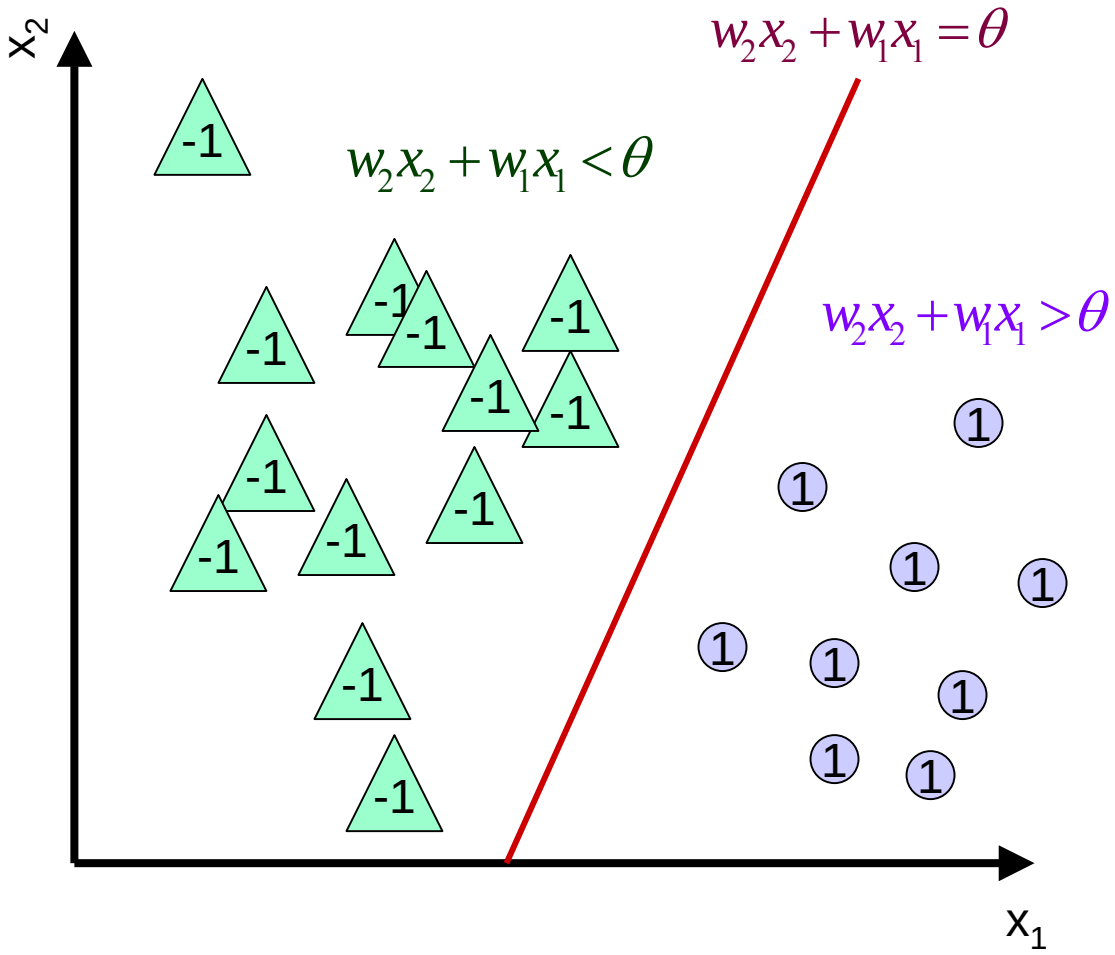
Threshold Logic Unit (TLU)



Activation Functions



Decision Surface of a Perceptron



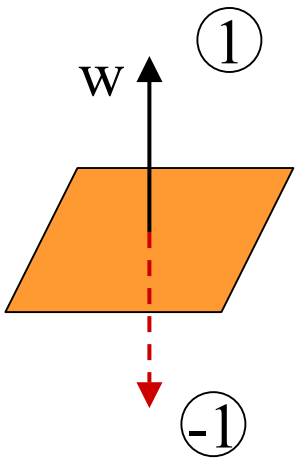
- **Line** equation:

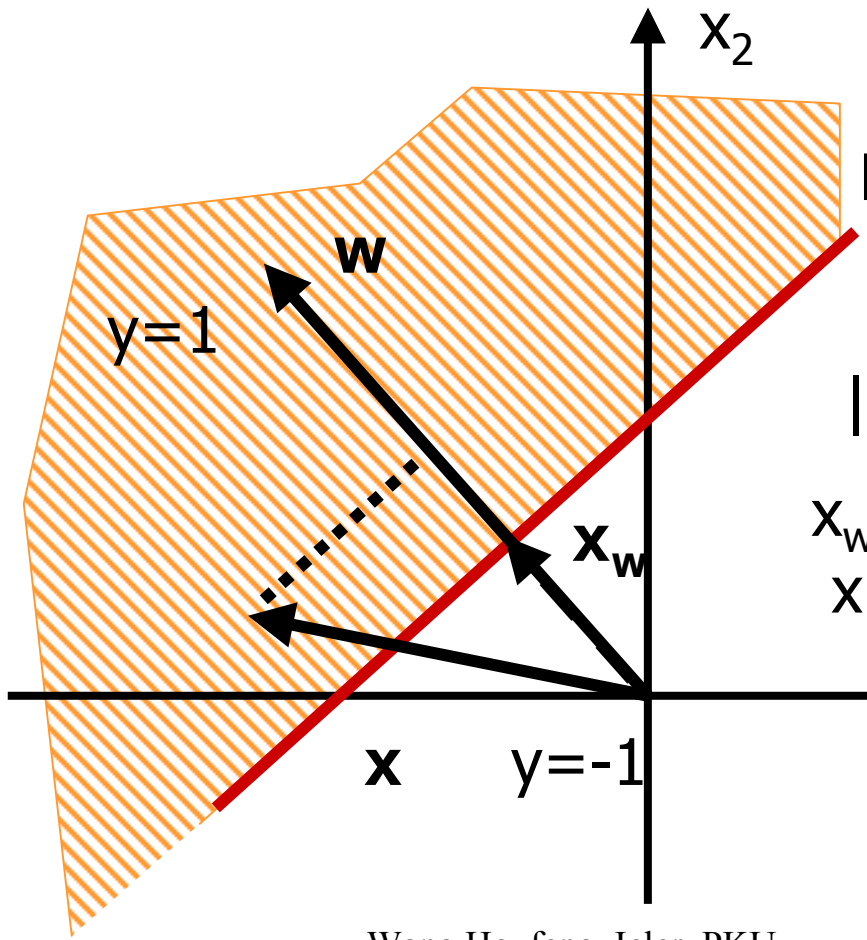
  $$w_2x_2 + w_1x_1 = \theta$$

- Classifier:
- If $w_2x_2 + w_1x_1 \geq \theta$
  - Output 1
- Else
  - Output -1

Geometric Interpretation

The relation $\mathbf{w} \bullet \mathbf{x} = \theta$ defines the decision line (plane)





Decision line

$\mathbf{w} \bullet \mathbf{x} = \theta$

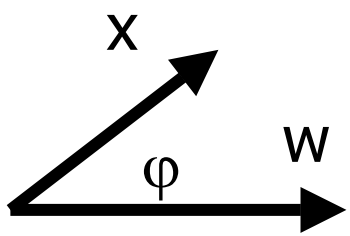
$|x_w| = \theta / |w|$

x_w: the projection from x to w

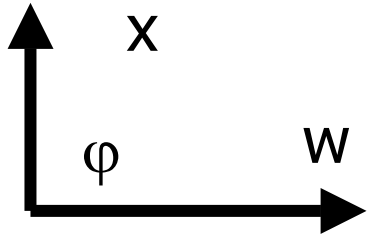
Geometric Interpretation

- In n dimensions the relation $\mathbf{w} \bullet \mathbf{x} = \theta$ defines a n-1 dimensional hyper-plane, which is orthogonal to the weight vector **w**.
- On one side of the hyper-plane ($\mathbf{w} \bullet \mathbf{x} > \theta$) all patterns are classified by the TLU as "1", while those that get classified as "0(-1)" lie on the other side of the hyper-plane.
- If patterns can be not separated by a hyper-plane then they cannot be correctly classified with a TLU.

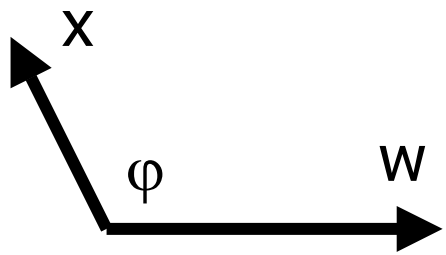
Inner Products & Projections



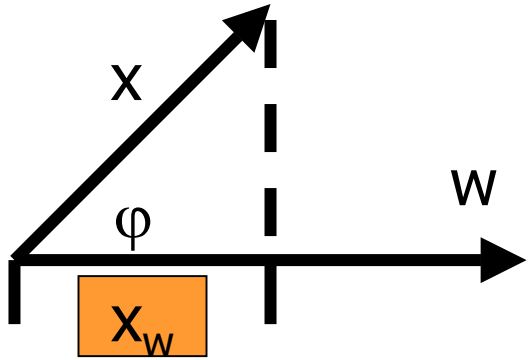
$w \bullet x > \theta$



$w \bullet x = \theta$

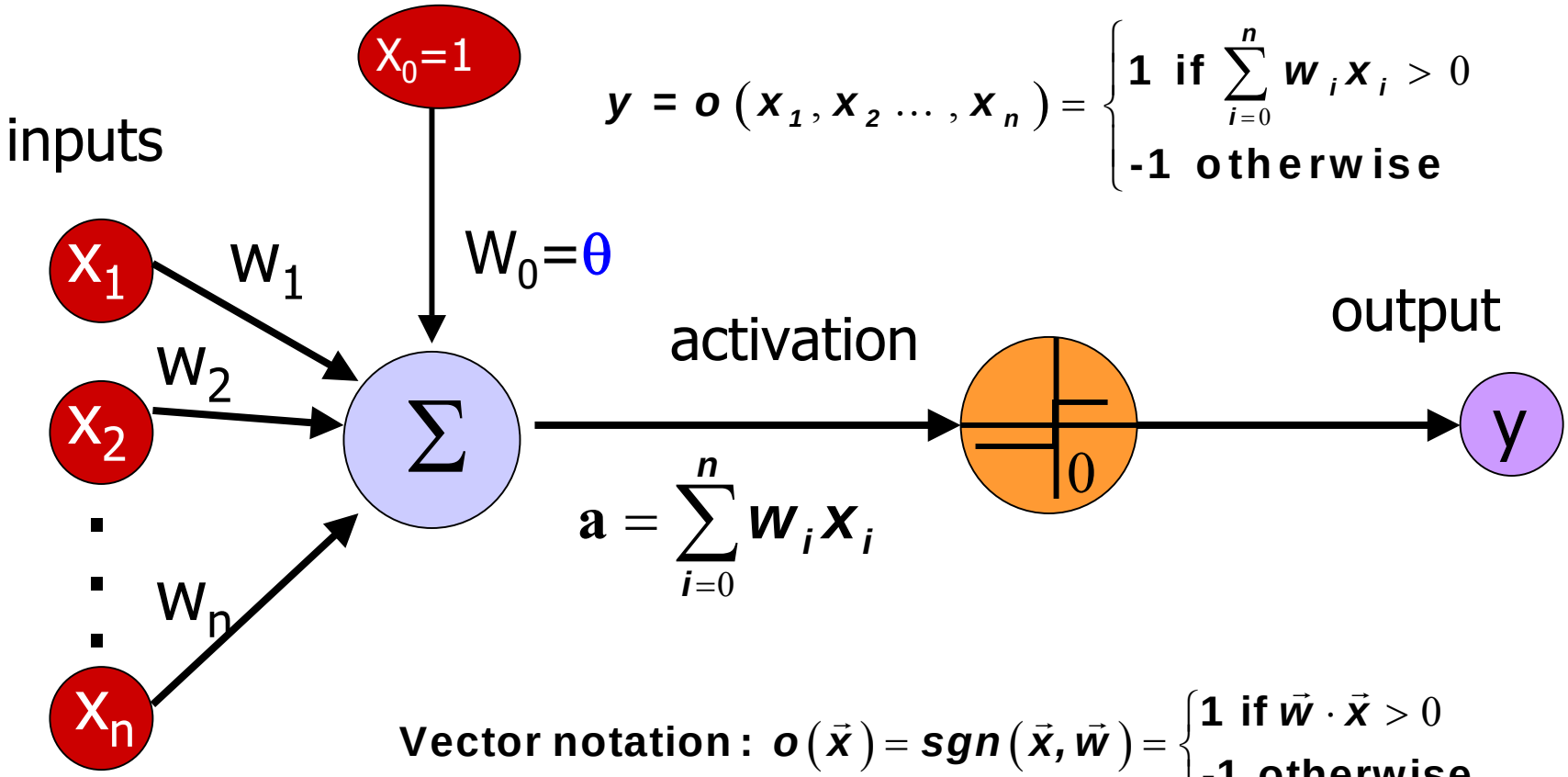


$w \bullet x < \theta$

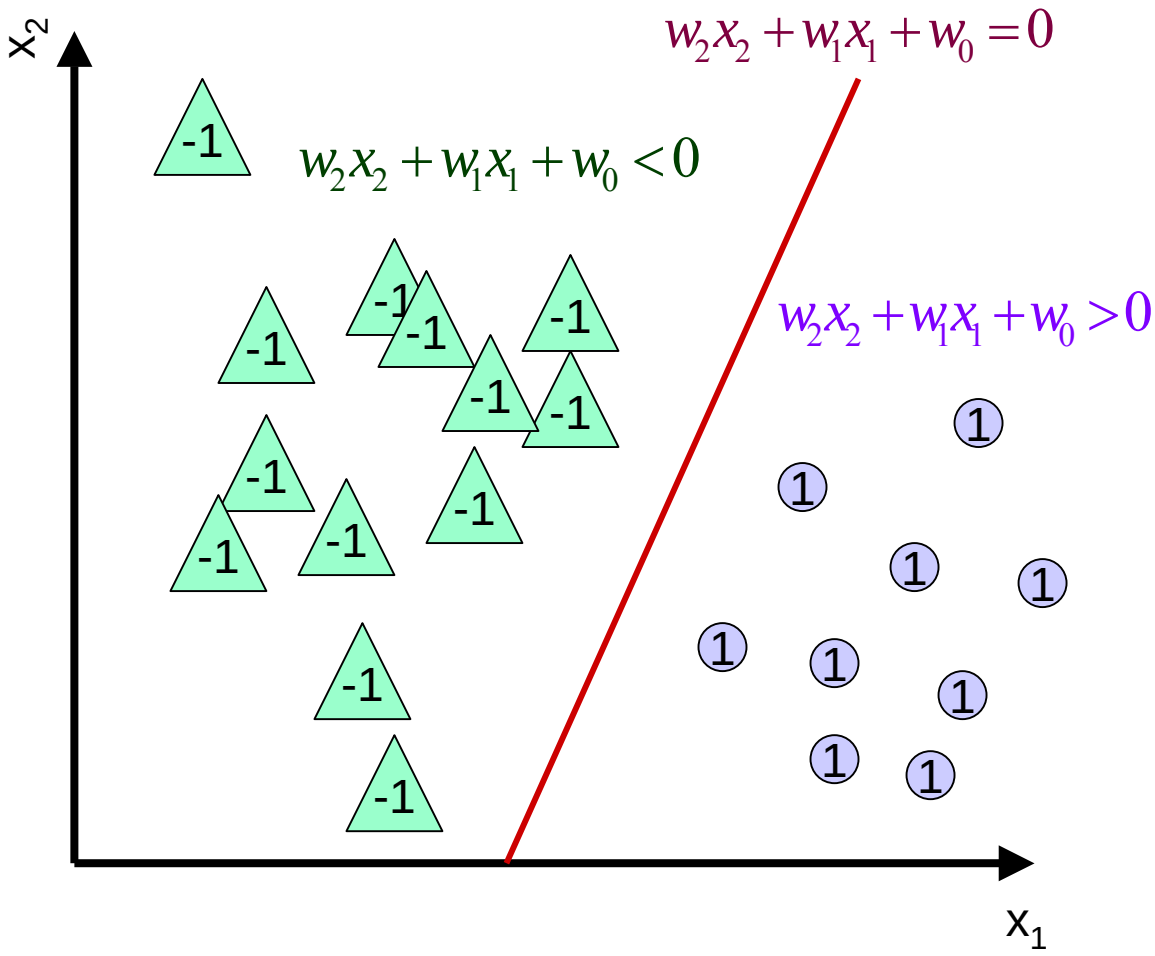


$w \bullet x = |w||x| \cos \varphi$

Bias: Θ to weight



Decision Surface of a Perceptron



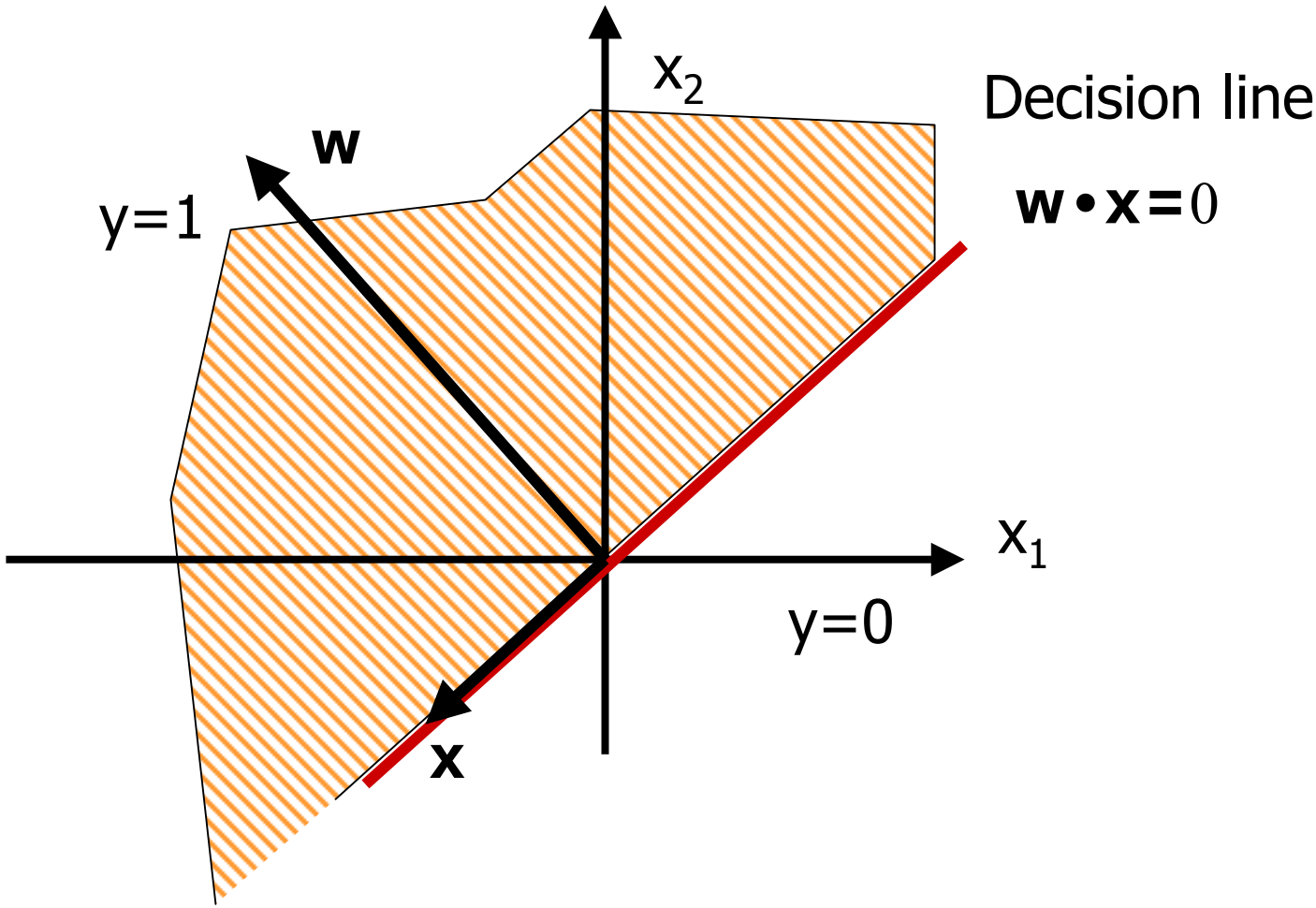
- Line equation:

  $w_2x_2 + w_1x_1 + w_0 = 0$

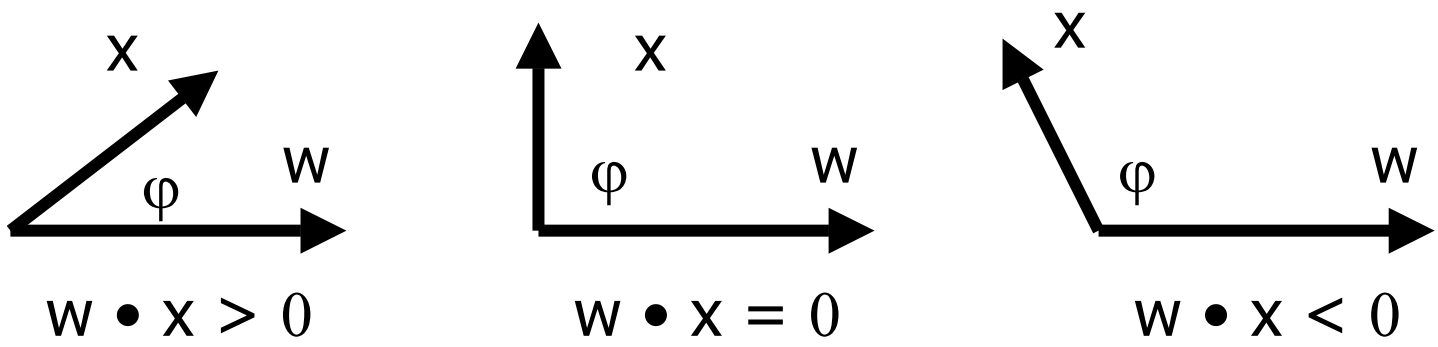
- Classifier:
- If $w_2x_2 + w_1x_1 + w_0 \geq 0$
  - Output 1
- Else
  - Output –1(or 0)

Geometric Interpretation

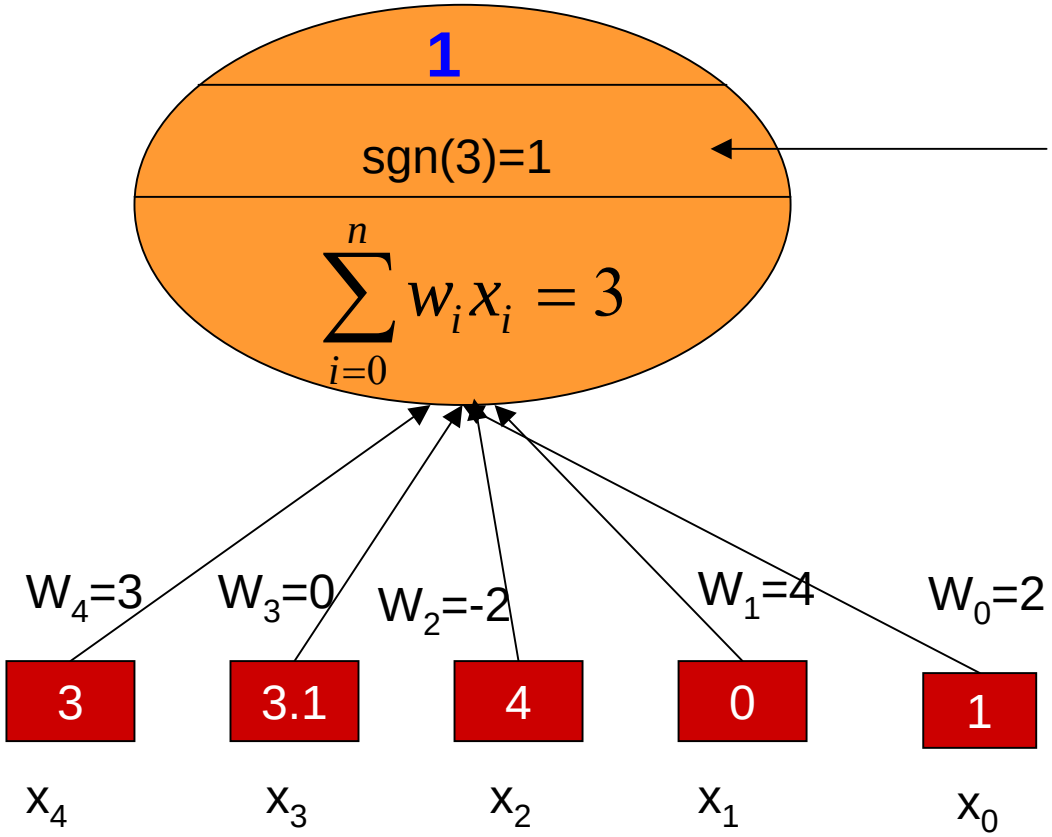
The relation $\mathbf{w} \bullet \mathbf{x} = 0$ defines the decision line(plane)



Inner Products & Projections



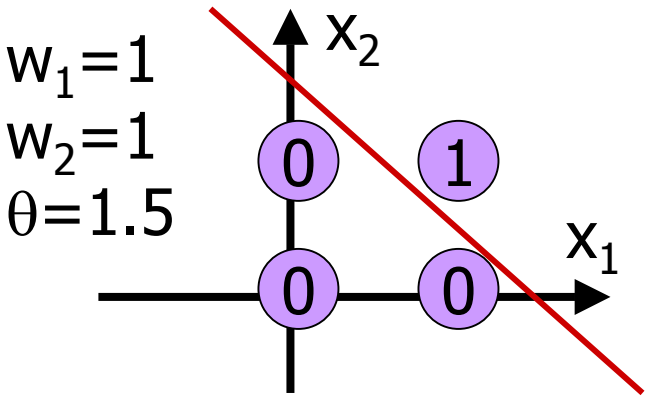
Structure

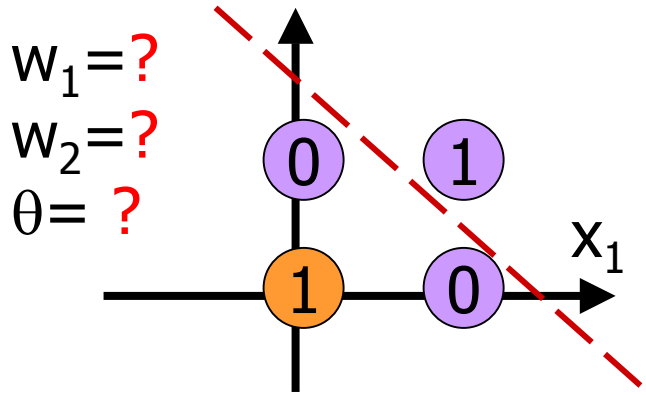

1
sgn(3)=1
$\sum_{i=0}^{n} w_i x_i = 3$
Transfer function: sgn
Output:
classification
W_4=3
W_3=0
W_2=-2
W_1=4
W_0=2
3
3.1
4
0
1
x_4
x_3
x_2
x_1
x_0
Input:
(attributes to classify)

Representability of a Perceptron

- Perceptron: Can Represent *Some* Useful Functions
  - **Linear Separable**
  - LTU emulation of logic gates (McCulloch and Pitts, 1943)
  - e.g., What weights represent $g(x_1, x_2) = AND(x_1, x_2)$?
    $OR(x_1, x_2)$? $NOT(x)$?
- Some Functions *Not* Representable
  - Not linearly separable
  - Solution: use networks of perceptrons (LTUs)

Linear Separability





Logical AND

x_1	x_2	a	y
0	0	0	0
0	1	1	0
1	0	1	0
1	1	2	1

Logical XOR

x_1	x_2	y	
0	0	0	
0	1	1	
1	0	1	
1	1	0	

Training Perceptron

- Use the Training Data to learn a Perceprton
- Hypotheses Space: $H = \{\vec{w} \mid \vec{w} \in \Re^{n+1}\}$
- Two Search method:
  - perceptron training rule;
  - gradient descent.
- Remember: the problem is to find "good" weights(minimize the error)

Outline

- Perceptrons
- **Percetron Learning**
- Sigmoid Unit
- MultiClass
- Multi-layer networks
- Backpropagation learning
- Others

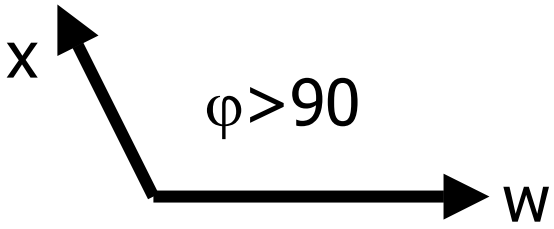
Training Perceptron

- Training set S of examples {**x**,**t**}
  - **x** is an input vector and
  - **t** the desired target vector
  - Example: **Logical And(OR)**

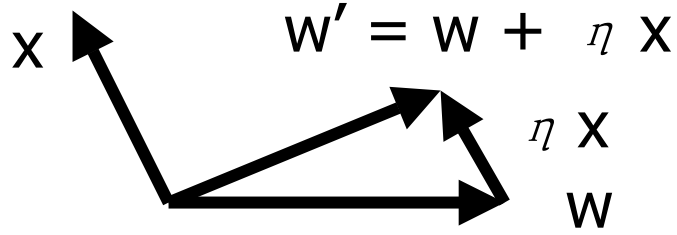
  S = {(0,0),0}, {(0,1),0}, {(1,0),0}, {(1,1),1}
- Iterative process
  - Present a training example x , compute network output y , compare output y with target t, adjust weights and thresholds
- Learning rule
  - Specifies how to **adjust the weights w and thresholds** θ of the network as a function of the inputs x, output y and target t.

Training:

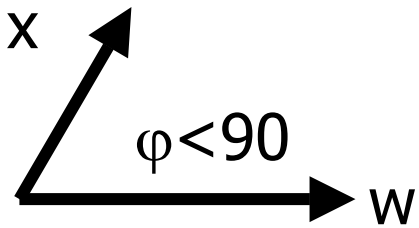
Adjusting the Weight Vector



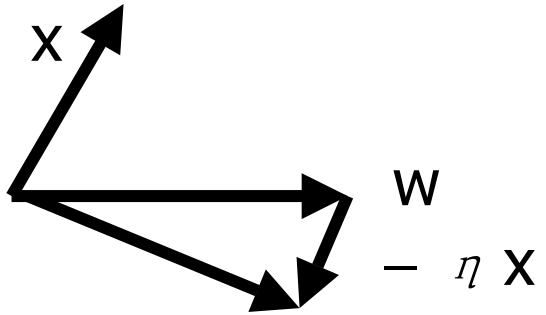
Target t=1
Output y=0



Move w in the direction of x



Target t=0
Output y=1



w′ = w - η x

Move w away from the direction of x

Training by Perceptron Training Rule

- Perceptron Learning Rule (Rosenblatt, 1959)

For each missclassified example $\vec{x}_d$ update weights:

$$\Delta w_i = \eta(t_d - o_d)x_{i,d}$$

$$w_i' \leftarrow w_i + \Delta w_i$$

In vector form:

$$\vec{w}' \leftarrow \vec{w} + \eta(t_d - o_d)\vec{x}_d$$

- η: arbitrary learning rate (e.g. 0.5)
- t_d : (true) label of the dth example
- o_d: output of the perceptron on the dth example
- $x_{i,d}$: value of predictor variable i of example d
- $t_d = o_d$: No change (for correctly classified examples)

Explaining the Perceptron Training Rule

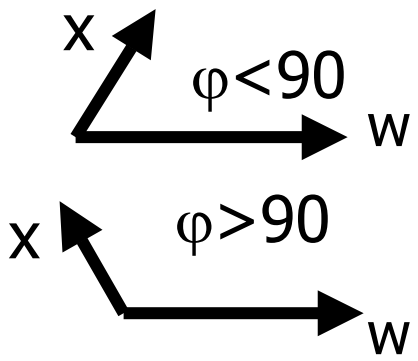
Rule : $\vec{w}' \leftarrow \vec{w} + \eta (t_d - o_d) \vec{x}_d$

Output : $\mathrm{sgn}(\vec{w} \cdot \vec{x}_d)$

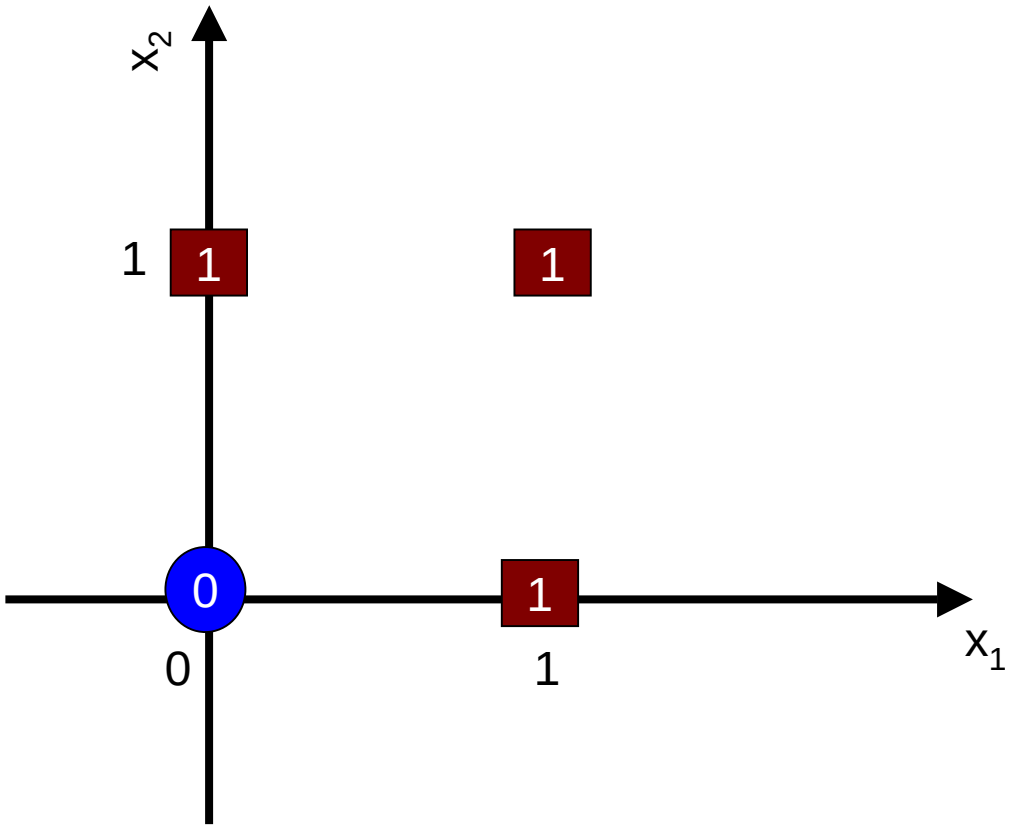
Effect on the output caused by a misclassified example x_d

$$\vec{w}' \cdot \vec{x}_d = \vec{w} \cdot \vec{x}_d + \Delta\vec{w}\, \vec{x}_d = \vec{w} \cdot \vec{x}_d + \eta(t_d - o_d)\|x_d\|^2$$

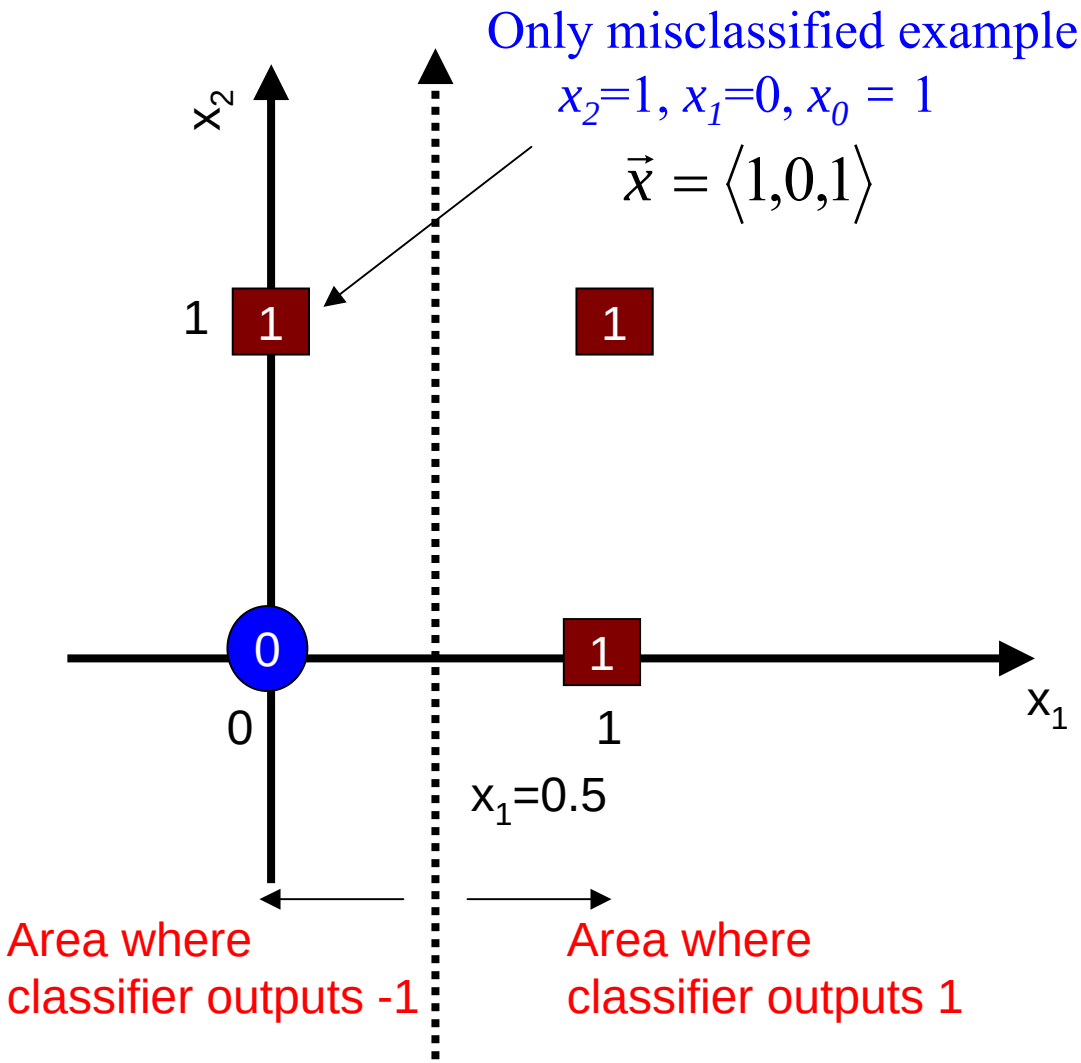
- $t_d = -1, o_d = 1$: $\vec{w} \cdot \vec{x}_d$ will decrease
-
- $t_d = 1, o_d = -1$: $\vec{w} \cdot \vec{x}_d$ will increase



Example of Training: The OR



Example of Training: The OR



- Initial random weights:

$$0x_2 + 1x_1 - 0.5 = 0$$

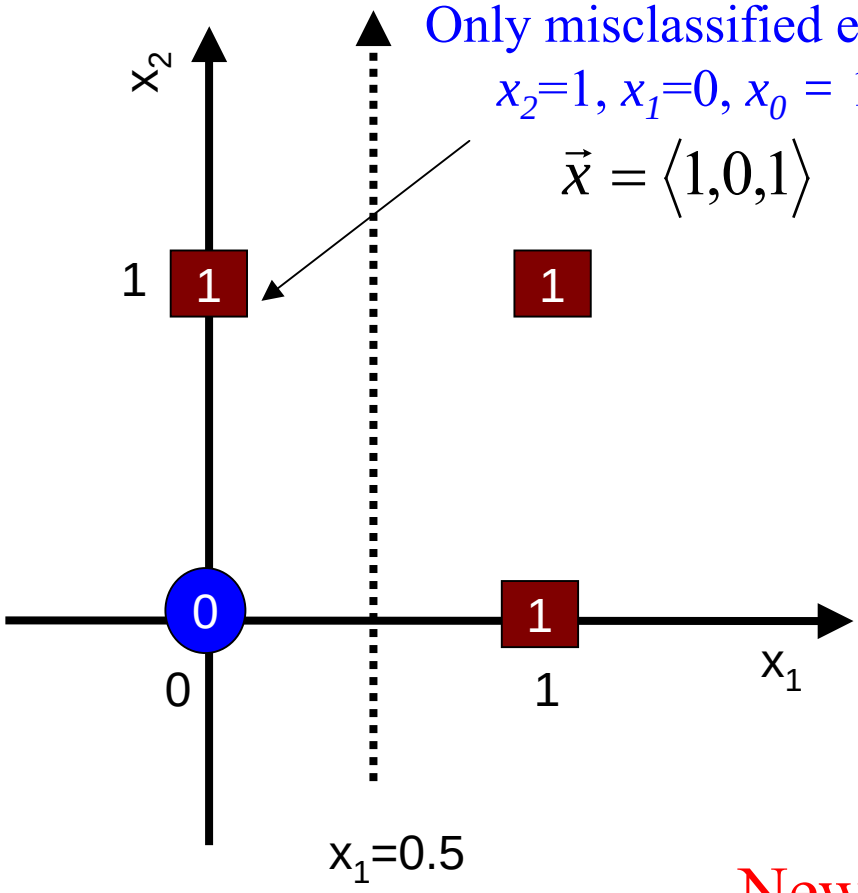
Define line:

x_1=0.5

Thus:

$$\vec{w} = \langle w_2, w_1, w_0 \rangle$$
$$= \langle 0,1,-0.5 \rangle$$

Example of Training: The OR



Old Line: $0x_2 + 1x_1 - 0.5 = 0$

Updateweights

$$\vec{w}' \leftarrow \vec{w} + \eta(t-o)\vec{x}$$

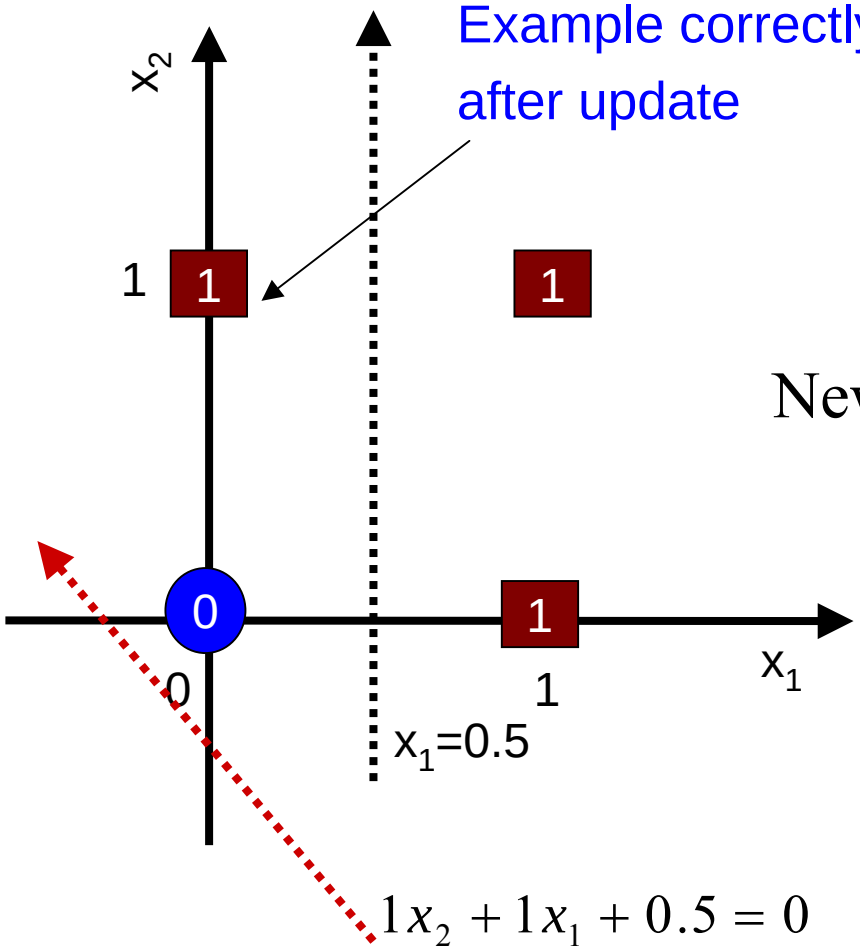
$$\vec{w}' \leftarrow \langle 0,1,-0.5 \rangle + 0.5 \cdot (1-(-1)) \cdot \langle 1,0,1 \rangle$$

$$\vec{w}' \leftarrow \langle 0,1,-0.5 \rangle + 1 \cdot \langle 1,0,1 \rangle$$

$$\vec{w}' \leftarrow \langle 1,1,0.5 \rangle$$

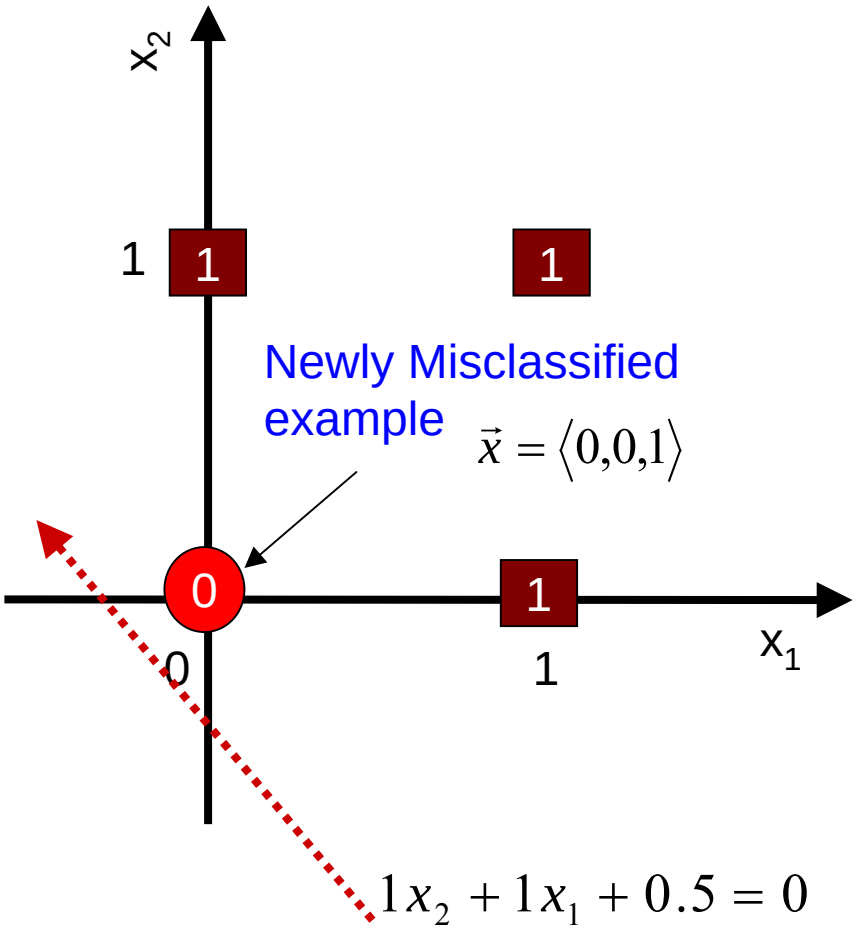
New line: $1x_2 + 1x_1 + 0.5 = 0$

Example of Training: The OR



New line : $1x_2 + 1x_1 + 0.5 = 0$

Example of Training: The OR



Next iteration:

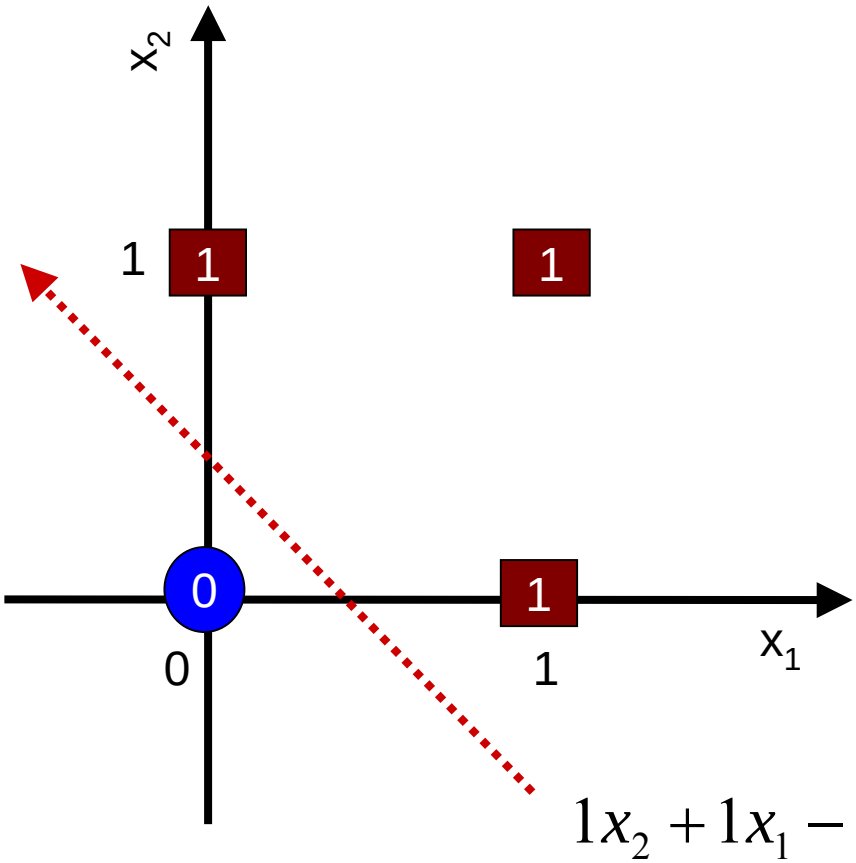
$$\vec{w}' \leftarrow \vec{w} + \eta(t-o)\vec{x}$$

$$\vec{w}' \leftarrow \langle 1,1,0.5 \rangle + 0.5 \cdot (-1-1) \cdot \langle 0,0,1 \rangle$$

$$\vec{w}' \leftarrow \langle 1,1,0.5 \rangle - 1\langle 0,0,1 \rangle$$

$$\vec{w}' \leftarrow \langle 1,1,-0.5 \rangle$$

Example of Training: The OR



New line:
$1x_2+1x_1-0.5=0$

Perfect classification

No change occurs next

Algorithm

Repeat

for each training vector pair (**x**,t)

evaluate the output o when **x** is the input

if o≠t then

form a new weight vector **w'** according to $\mathbf{w'}=\mathbf{w}+\eta\,(t-o)\,\mathbf{x}$

else

do nothing

end if

end for

Until o=t for all training vector pairs

Perceptron Convergence Theorem

- The algorithm converges to the correct classification
  - **if the training data is linearly separable**
  - and η is sufficiently small
- **If two classes of vectors X_1 and X_2 are linearly separable**, the application of the perceptron training algorithm will eventually result in a weight vector $\mathbf{w_0}$, such that $\mathbf{w_0}$ defines a TLU whose decision hyper-plane separates X_1 and X_2 (Rosenblatt 1962).
- Solution $\mathbf{w_0}$ is not unique, since if $\mathbf{w_0}\ \mathbf{x} = 0$ defines a hyper-plane, so does $\mathbf{w'_0} = k\ \mathbf{w_0}$.

Training by Gradient Descent

- Algorithm(Rosenblatt) will always converge within finite number of iterations if the data are linearly separable.
- Otherwise,
  - it may oscillate (no convergence) if the data are not separable linearly!
  - In this case, a approximately separable line could be found by using gradient descent method
- Solution:
  - Define an error function
  - Search for weights that minimize the error, i.e., find weights that zero the error gradient

Squared Error

- Consider linear unit **without threshold** and **continuous output** o (not just –1,1)
  - $o = w_0 + w_1 x_1 + \ldots + w_n x_n$
- Squared Error: t_d label of *d*-th example, o_d current output on *d-th* example:

$$E[w_0, w_1, \ldots w_n] = E(\vec{w}) = \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2$$

where, D is the set of training examples

Minimize Squared Error

- Reduce Error step by step by adjusting weight in inverted gradient direct and make it zero ideally:

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2$$

$$= \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d - o_d)^2$$

$$= \frac{1}{2}\sum_{d\in D}2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d)$$

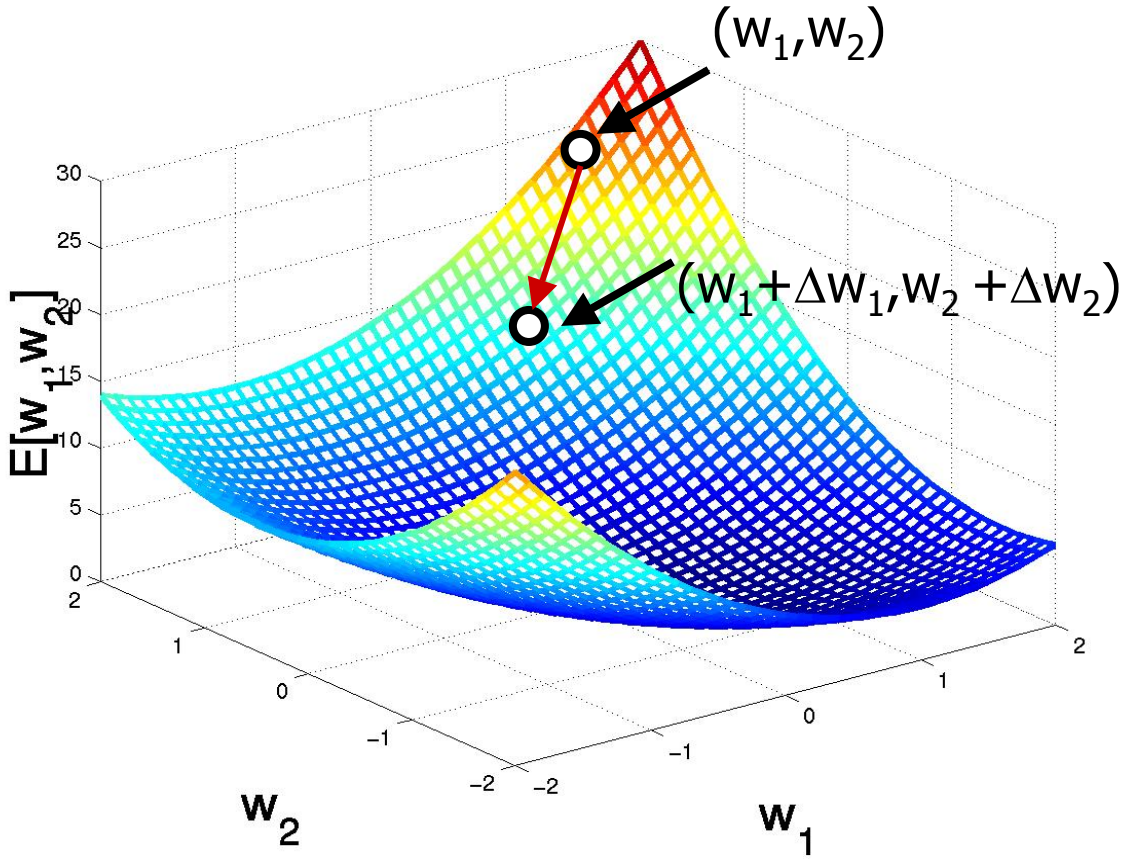
$$= \sum_{d\in D}(t_d - o_d)\frac{\partial}{\partial w_i}(-o_d)$$

$$= \sum_{d\in D}[(t_d - o_d)\frac{\partial}{\partial w_i}(-\sum_{i=0}^{n} w_i x_{id})]$$

$$= \sum_{d\in D}[(t_d - o_d)(-x_{id})]$$

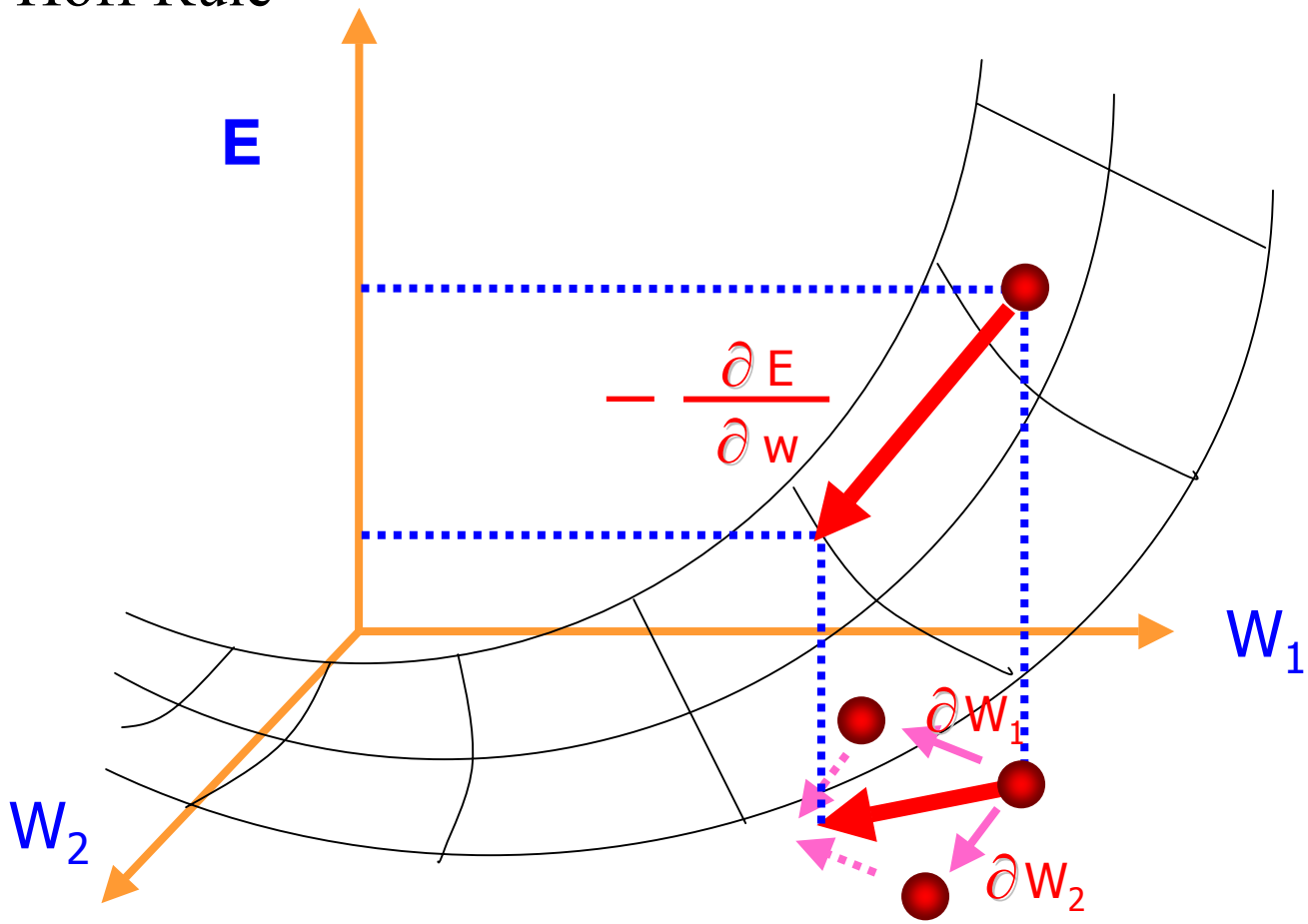
Gradient Descent

n+1 dimensional space with n weights



Gradient Descent

Derivation of Delta Rule/LMS(least-mean-square) Rule/ Windrow-Hoff Rule



The Gradient-Descent Rule

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_1}, \ldots \ldots \ldots, \frac{\partial E}{\partial w_N} \right]$$

The "gradient"

This is a N+1 dimensional vector (i.e., the 'slope' in weight space) Since we want to **reduce** errors, we want to go "down hill" We'll take a finite step in weight space:

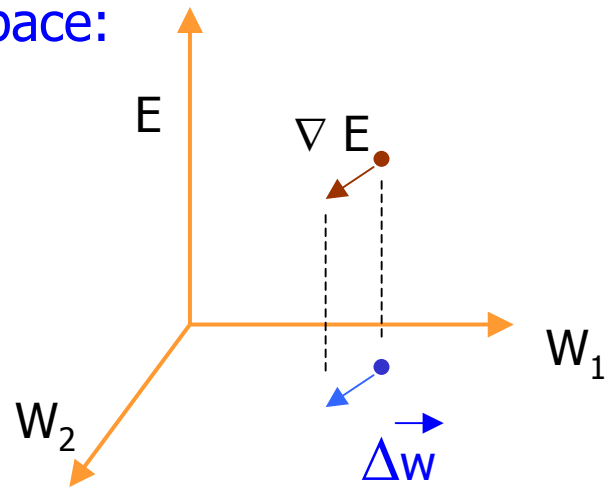
"delta" = change to $\vec{w}$

['i:te]

$$\Delta \vec{w} = - \eta \nabla E (\vec{w})$$

"Learning Rate"

$$\text{Or } \Delta w_i = - \eta \frac{\partial E}{\partial w_i}$$



Gradient Descent Training

Gradient-Descent-batch(*training_examples*, η)

Each training example is a pair of the form $<(x_1,\ldots x_n),t>$ where $(x_1,\ldots,x_n)$ is the vector of input values, and t is the target output value, η is the learning rate (e.g. 0.1)

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero
  - For each $<(x_1,\ldots x_n),t>$ in *training_examples* Do
    - Input the instance $(x_1,\ldots,x_n)$ to the linear unit and compute the output o
    - For each linear unit weight w_i Do
      - $\Delta w_i = \Delta w_i + \eta\ (t-o)\ x_i$
  - For each linear unit weight w_i Do
    - $w_i = w_i + \Delta w_i$

Gradient Descent Training

Gradient-Descent-Online(*training_examples*, η)

Each training example is a pair of the form $<(x_1,\ldots x_n),t>$ where $(x_1,\ldots,x_n)$ is the vector of input values, and t is the target output value, η is the learning rate (e.g. 0.1)

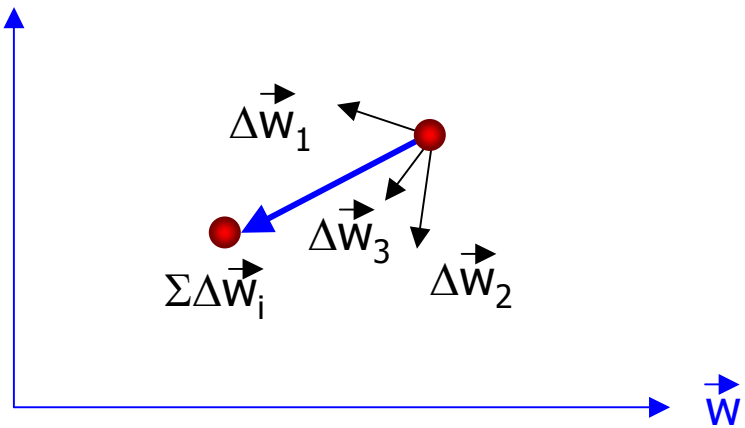
- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero
  - For each $<(x_1,\ldots x_n),t>$ in *training_examples* Do
    - Input the instance $(x_1,\ldots,x_n)$ to the linear unit and compute the output o
    - For each linear unit weight w_i Do
      - $w_i = w_i + \eta\,(t-o)\,x_i$ (ie. $\Delta w_i = \eta\,(t-o)\,x_i$)

Online vs. Batch

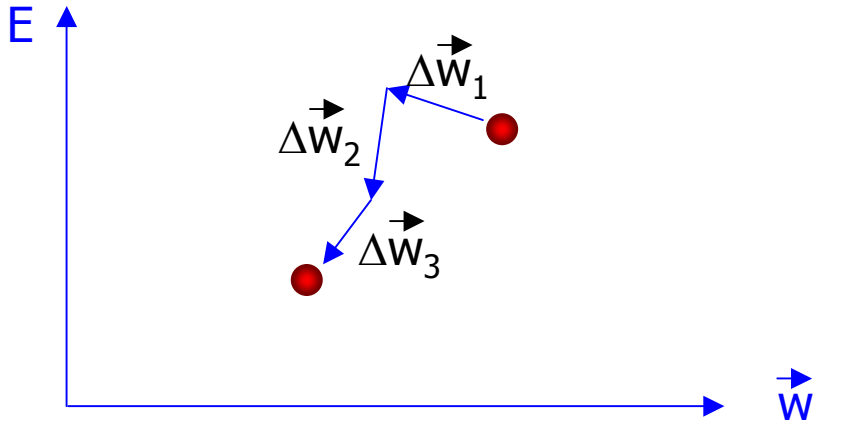
- Technically, we should look at the error gradient for the entire training set, before taking a step in weight space
- *However*, as presented, we take a step after each example ("on-line")
  - Much faster convergence
  - Can reduce local minimal value

Online vs. Batch

BATCH – add Δw vectors for every training example, then 'move' in weight space.



ON-LINE – "move" after each example (ie, *stochastic* gradient descent)



* Note $\Delta w_{i,BATCH} \neq \Delta w_{i,\ ON\text{-}LINE}$, for i > 1

* Final locations in $\vec{w}$ space need not be the same for BATCH and ON-LINE

Perceptron Rule vs. Gradient Descent Rule

- Perceptron rule

  $w'_i = w_i + \eta (t_d - o_d) x_{i,d}$

  - derived from manipulation of decision surface (threshold for output). Applied in TLU(linear Separable)

- gradient descent rule

  $w'_i = w_i + \eta (t_d - o_d) x_{i,d}$

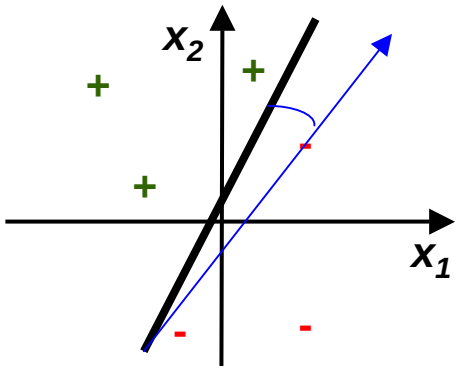
  - derived from minimization of error function

  $E[w_1,\ldots,w_n] = \frac{1}{2} \Sigma_d (t_d - o_d)^2$

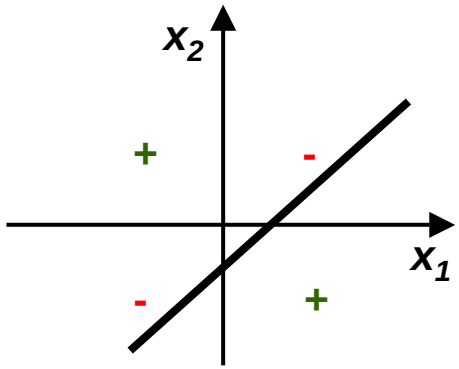
  by means of gradient descent (unthreshold for output).

  - Applied in all perceptron forms---continuous function

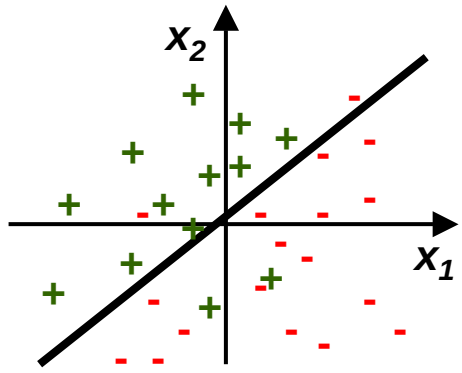
Perceptron Rule vs. Gradient Descent Rule



Example A



Example B



Example C

- LS(linear Separable) Concepts: Can Achieve Perfect Classification
  - Example A: perceptron training rule converges
- Non-LS Concepts: Can Only Approximate
  - Example B: not LS; delta rule converges
  - Example C: not LS; better results from delta rule

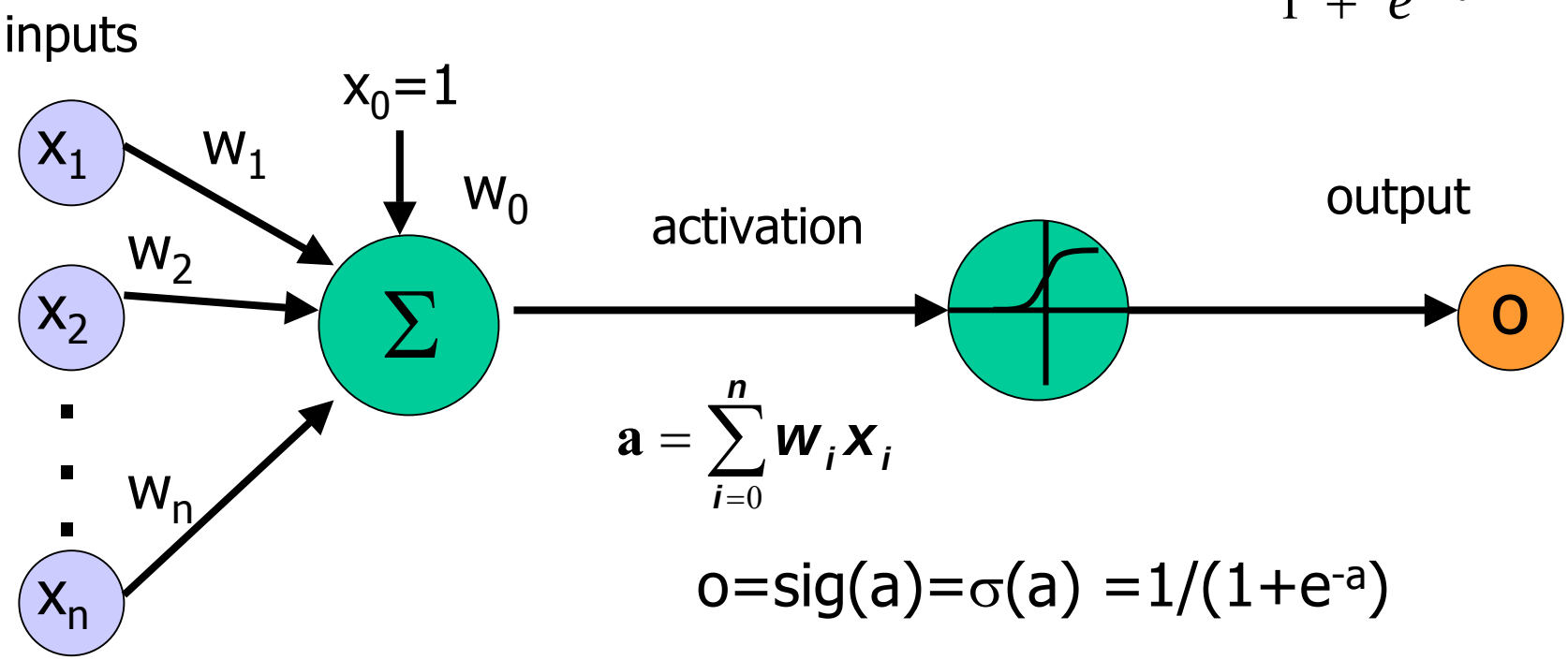
Outline

- Perceptrons
- Perceptrons learning
- ➢ **Sigmoid Unit**
- MultiClass
- Multi-layer networks
- Backpropagation Learning
- Others

Sigmoid (Logistic function)

- Replace $\mathrm{sgn}(\vec{w} \cdot \vec{x}_d)$ with the sigmoid $sig(\vec{w} \cdot \vec{x}_d)$

$$where, \quad sig(y) = \frac{1}{1 + e^{-y}}$$



Calculating the Gradient

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (-o_d)$$

$$= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (-sig(\vec{w} \cdot \vec{x}_d))$$

$$= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (-sig(\vec{w} \cdot \vec{x}_d))$$

$$= -\sum_{d \in D} (t_d - o_d) \frac{\partial sig(\vec{w} \cdot \vec{x}_d)}{\partial (\vec{w} \cdot \vec{x}_d)} \frac{\partial (\vec{w} \cdot \vec{x}_d)}{\partial w_i}$$

$$= -\sum_{d \in D} (t_d - o_d) sig(\vec{w} \cdot \vec{x}_d)(1 - sig(\vec{w} \cdot \vec{x}_d)) \frac{\partial (\vec{w} \cdot \vec{x}_d)}{\partial w_i}$$

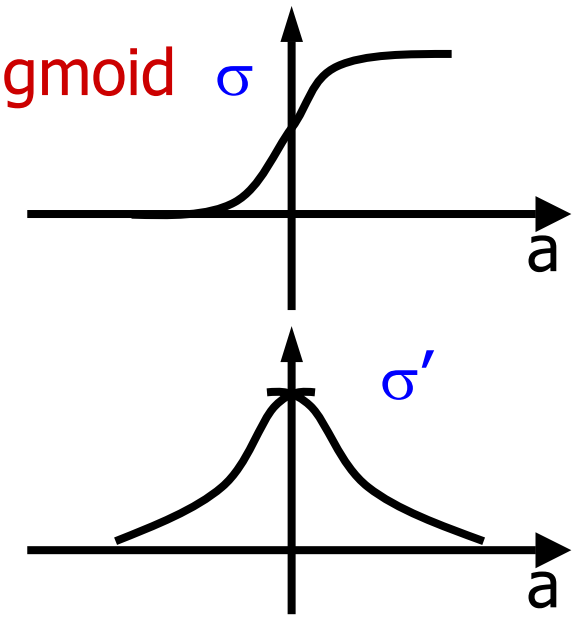
$$= -\sum_{d \in D} (t_d - o_d) sig(\vec{w} \cdot \vec{x}_d)(1 - sig(\vec{w} \cdot \vec{x}_d)) \cdot x_{i,d}$$

$$\nabla E(\vec{w}) = -\sum_{d \in D} (t_d - o_d) sig(\vec{w} \cdot \vec{x}_d)(1 - sig(\vec{w} \cdot \vec{x}_d)) \cdot \vec{x}_d$$

Note:

$$sig(x) = \frac{1}{1 + e^{-x}}$$

$$\frac{dsig(x)}{dx} = sig(x)(1 - sig(x))$$



Updating the Weights with Gradient Descent

$$\vec{w} \leftarrow \vec{w} - \eta \nabla E(\vec{w}) \qquad \text{where, } \Delta\vec{w} = -\eta \nabla E(\vec{w})$$

$$\vec{w} \leftarrow \vec{w} + \eta \sum_{d \in D} (t_d - o_d) sig(\vec{w} \cdot \vec{x}_d)(1 - sig(\vec{w} \cdot \vec{x}_d)) \cdot \vec{x}_d$$

- Each weight update goes through all training instances
- Each weight update more expensive but more accurate
- Always converges to a local minimum regardless of the data
- When using the sigmoid: output is a real number between 0 and 1
- Thus, labels (desired outputs) have to be represented with numbers from 0 to 1

Outline

- Perceptrons
- Perceptrons learning
- Sigmoid Unit
- **MultiClass**
- Multi-layer networks
- Backpropagation Learning
- Application: Classification

An Example

X_0	X_1	X_2	…	Class
1	0.4	-1		A
1	9	0.5		A
1	1	3		C
1	8.4	-.8		B
1	-3.4	.2		D

Encoding Multiclass Problems

- Use one perceptron (output unit) and encode the output as follows:
  - Use 0.125 to represent class A (middle point of [0,.25])
  - Use 0.375, to represent class B (middle point of [.25,.50])
  - Use 0.625, to represent class C (middle point of [.50,.75])
  - Use 0.875, to represent class D (middle point of [.75,1]
- The training data then becomes:

X_0	X_1	X_2	…	Class
1	0.4	-1		**0.125**
1	9	0.5		**0.125**
1	1	3		**0.625**
1	8.4	-.8		**0.365**
1	-3.4	.2		**0.875**

Encoding Multiclass Problems

- Use one perceptron (output unit) and encode the output as follows:
  - Use 0.125 to represent class A (middle point of [0,.25])
  - Use 0.375, to represent class B (middle point of [.25,.50])
  - Use 0.625, to represent class C (middle point of [.50,.75])
  - Use 0.875, to represent class D (middle point of [.75,1]
- To classify a new input vector *x*:

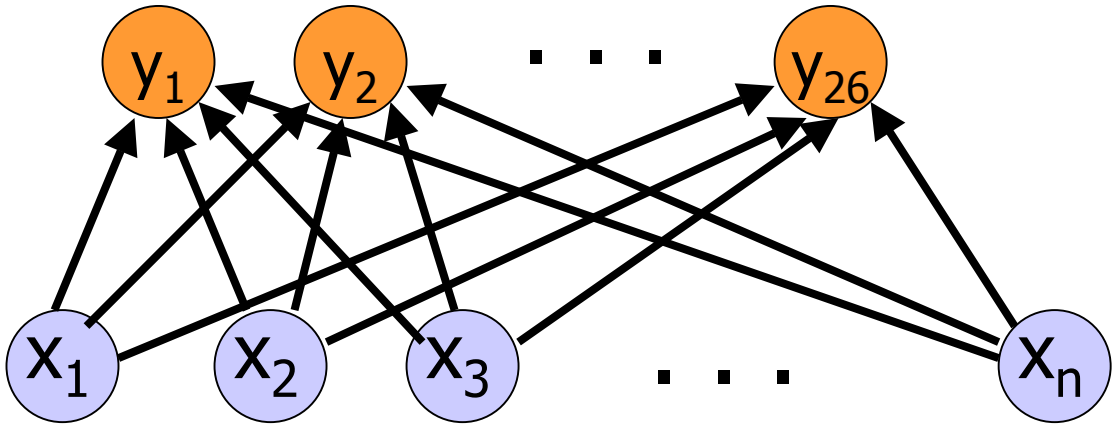
$$\text{If } sig(\vec{w}\cdot\vec{x}) \in [0,.25] \text{ classify as Class A}$$
$$\text{If } sig(\vec{w}\cdot\vec{x}) \in [.25,.5] \text{ classify as Class B}$$
$$\text{If } sig(\vec{w}\cdot\vec{x}) \in [.5,.75] \text{ classify as Class C}$$
$$\text{If } sig(\vec{w}\cdot\vec{x}) \in [.75,.1] \text{ classify as Class D}$$

- For two classes only and a sigmoid unit suggested values 0.1 and 0.9 (or 0.25 and 0.75)

Multiple TLUs for Multiclass

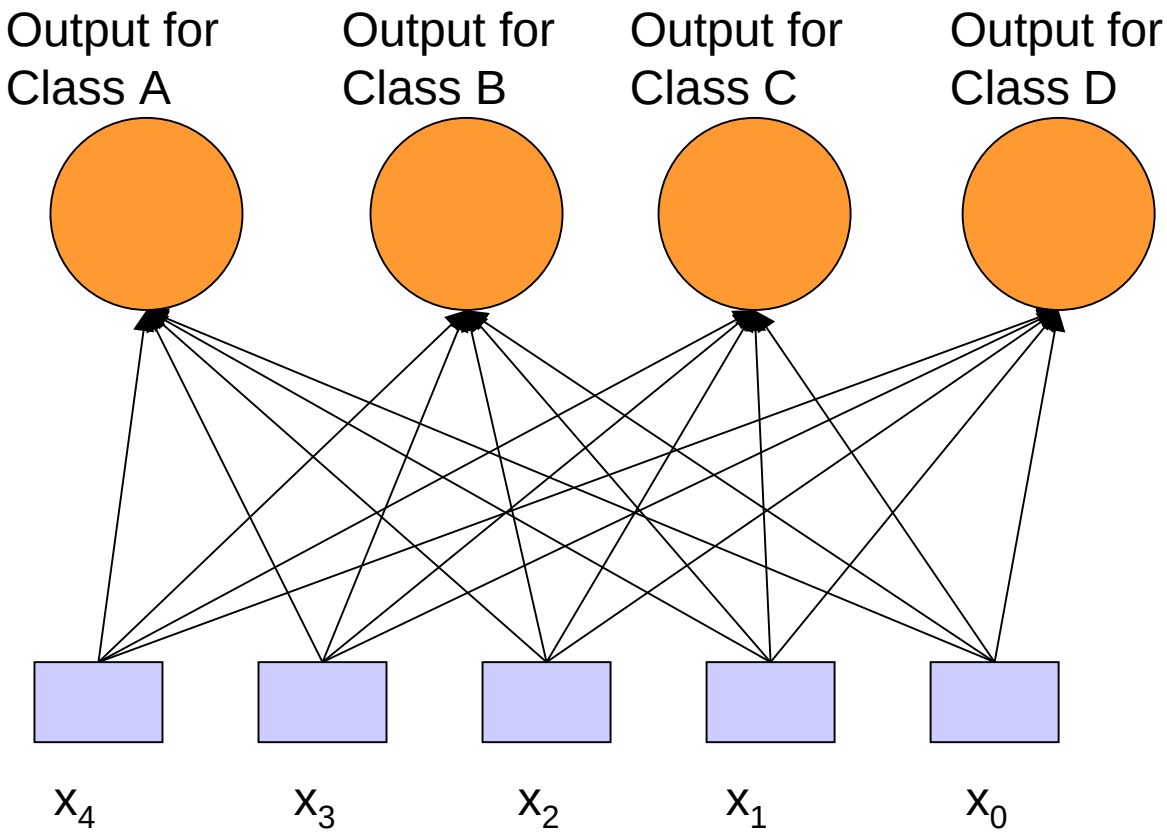
- Handwritten alphabetic character recognition
  - 26 classes : A,B,C...,Z
  - First TLU distinguishes between "A"s and "non-A"s, second TLU between "B"s and "non-B"s etc.

w_{ji} connects x_i with y_j



Multiclass with Largest Output

- Assign to class with largest output



1-of-M Encoding

- E.g., 4 nominal classes, A, B, C, D

X_0	X_1	X_2	…	Class A	Class B	Class C	Class D
1	0.4	-1		**0.9**	0.1	0.1	0.1
1	9	0.5		**0.9**	0.1	0.1	0.1
1	1	3		0.1	0.1	**0.9**	0.1
1	8.4	-.8		0.1	**0.9**	0.1	0.1
1	-3.4	.2		0.1	0.1	0.1	**0.9**

Outline

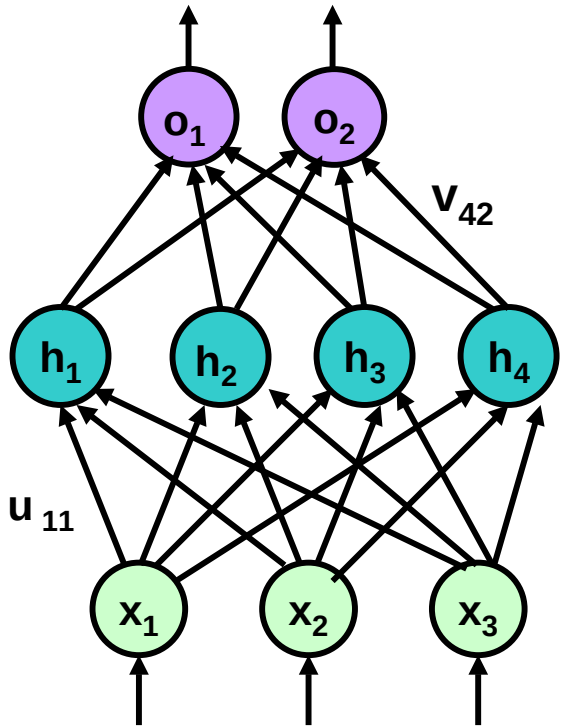
- Perceptrons
- Perceptrons learning
- Sigmoid Unit
- MultiClass
- ➢ **Multi-layer networks**
- Backpropagation Learning
- Others

Multi-Layer Networks

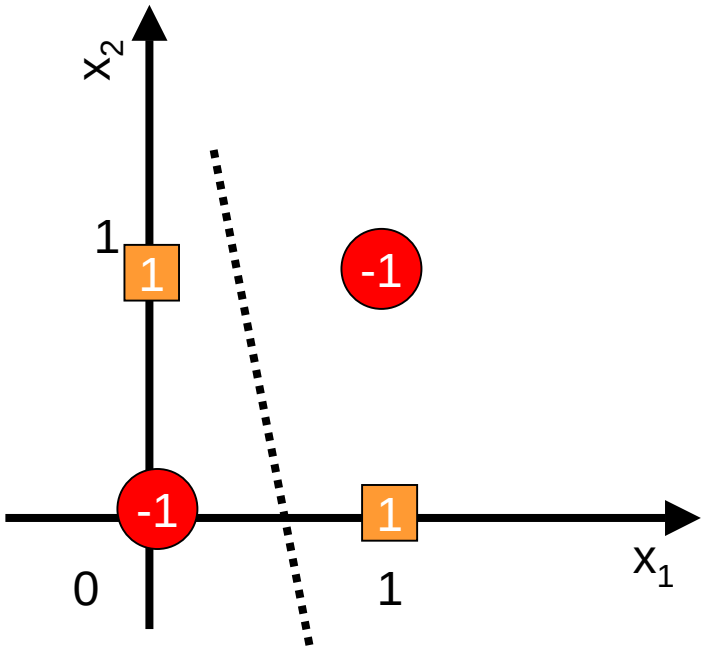
Output Layer

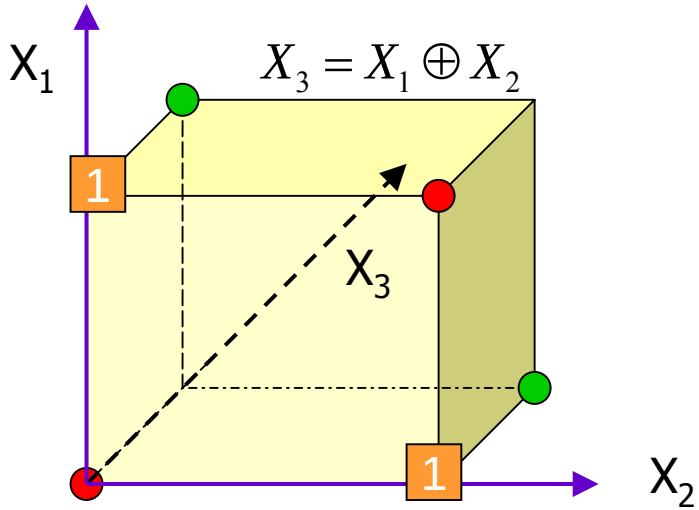
Hidden Layer

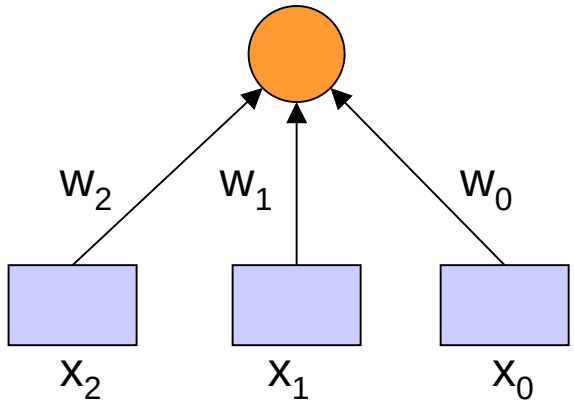
Input Layer



A problem: how to express XOR

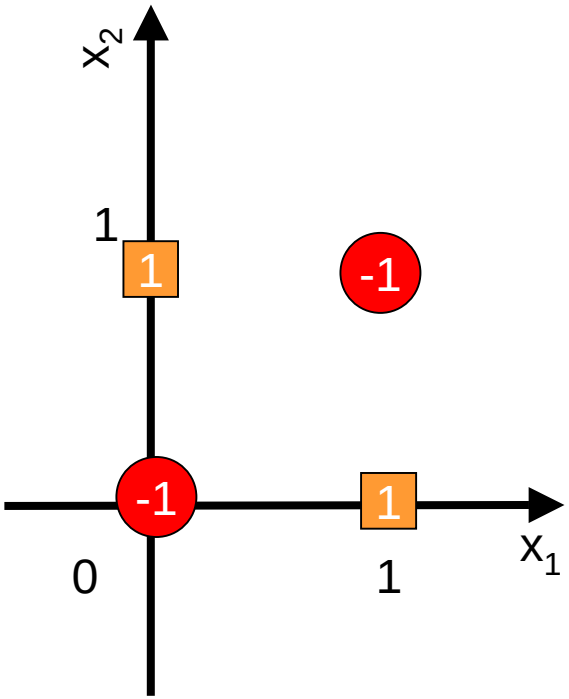


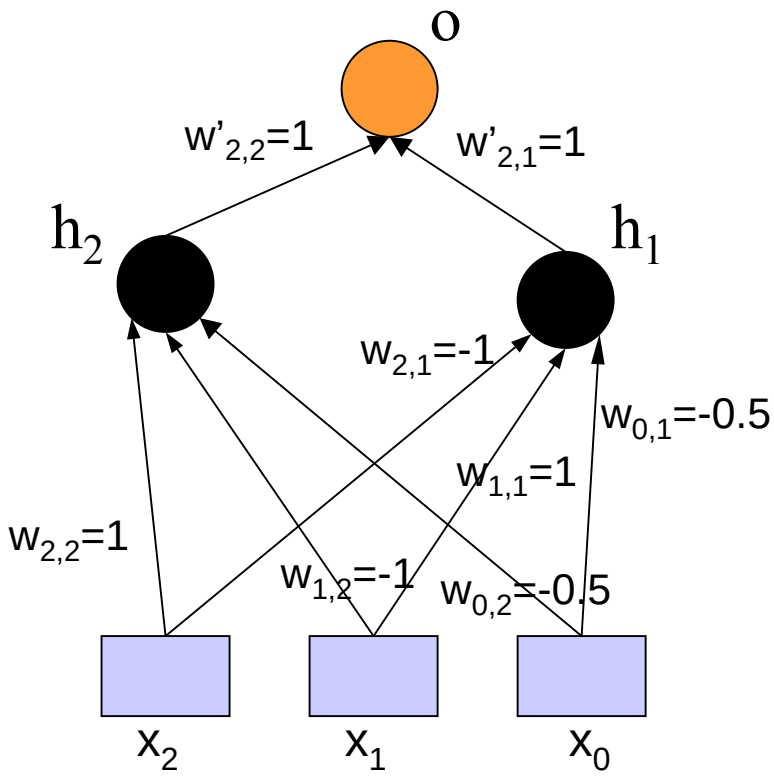




No line (no set of three weights) can separate the training examples (learn the true function).

Multilayer ANN



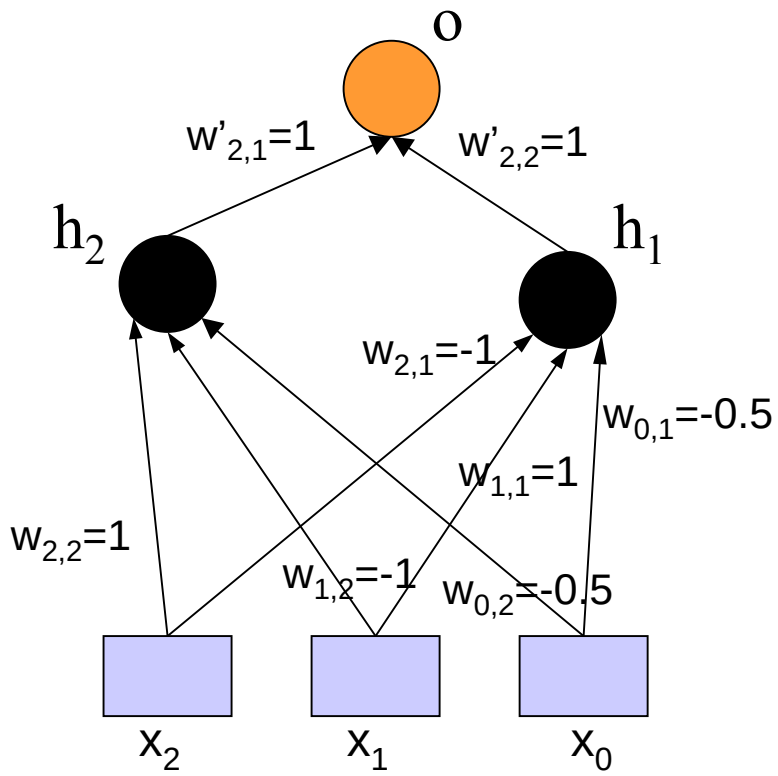


$$h_1 = \mathrm{sgn}(x_1 - x_2 - 0.5)$$
$$h_2 = \mathrm{sgn}(x_2 - x_1 - 0.5)$$

$$\mathbf{o} = \begin{cases} 1, & if \quad (w'_{2,2}h_2 + w'_{2,1}h_1) \geq 0 \\ -1, & otherwise \end{cases}$$

XOR: Multilayer ANN

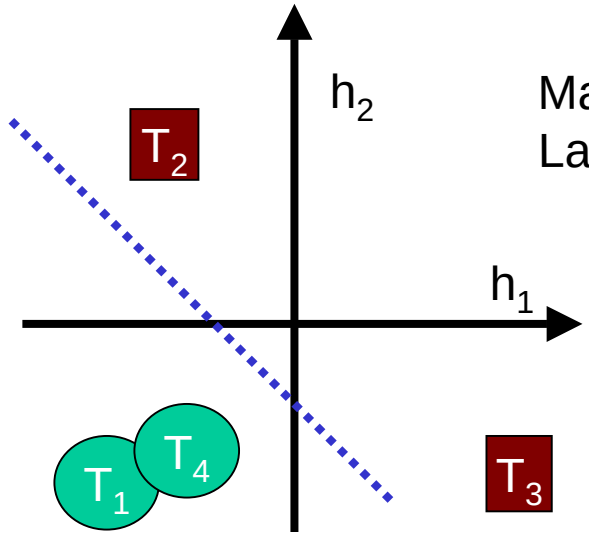
	x_1	x_2	h_1	h_2	**o**
T_1	0	0	-1	-1	-1
T_2	0	1	-1	1	1
T_3	1	0	1	-1	1
T_4	1	1	-1	-1	-1



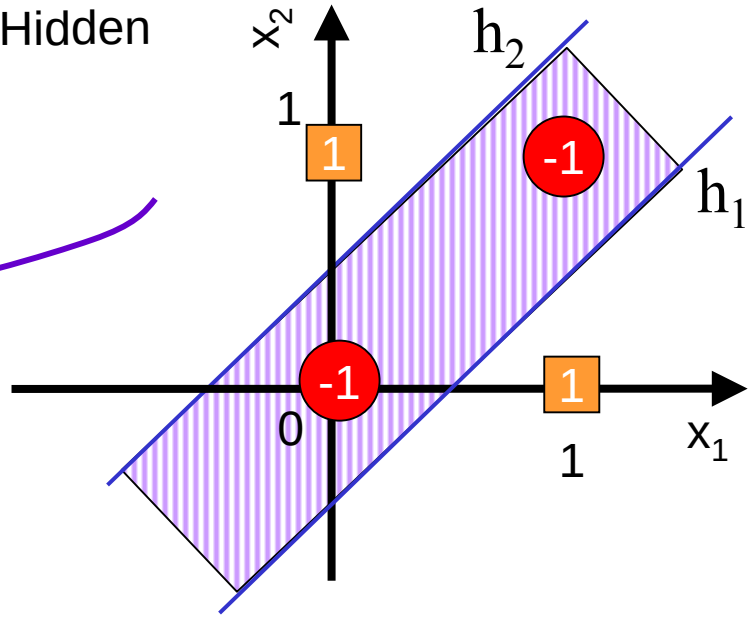
$$h_1 = \mathrm{sgn}(x_1 - x_2 - 0.5)$$
$$h_2 = \mathrm{sgn}(x_2 - x_1 - 0.5)$$

$$\mathbf{o} = \begin{cases} 1, & if \quad (w'_{2,2}h_2 + w'_{2,1}h_1) \geq 0 \\ -1, & otherwise \end{cases}$$

From the Viewpoint of the Output Layer



Mapped By Hidden Layer to:



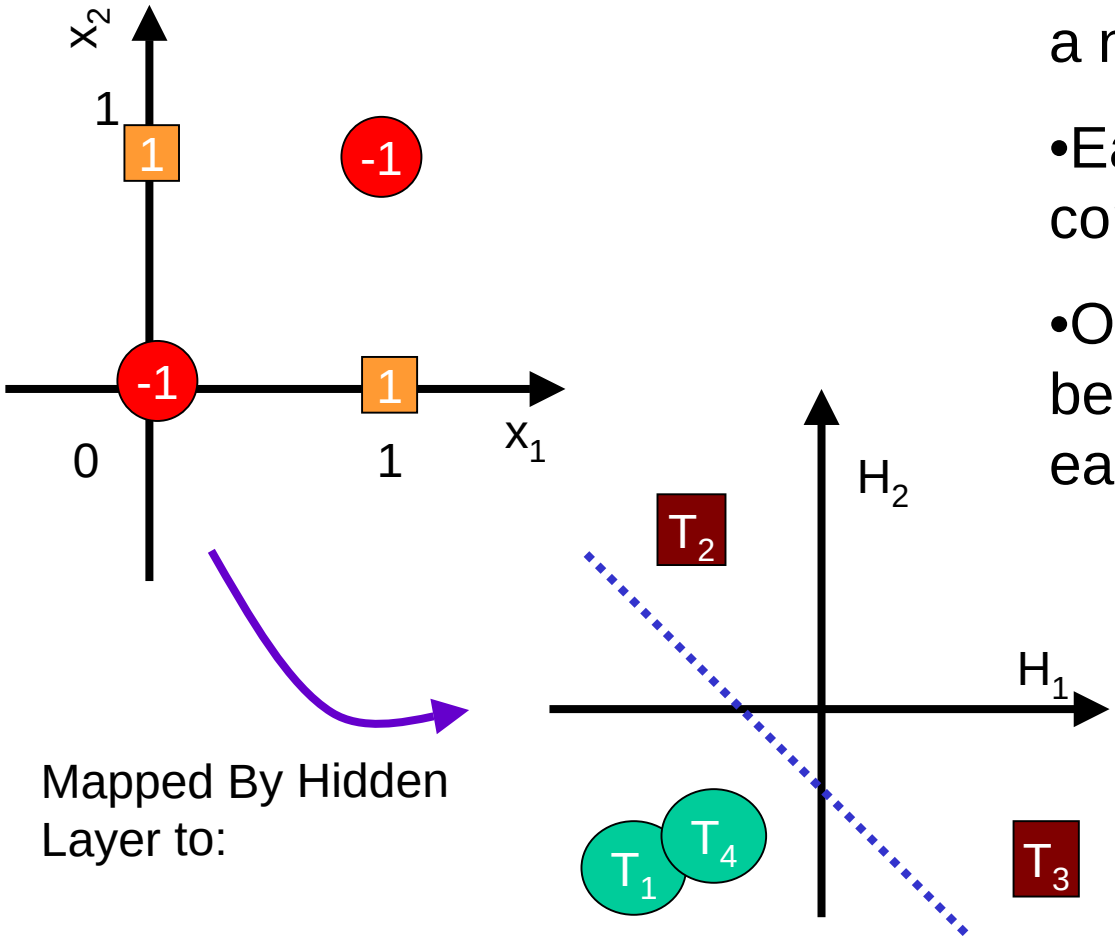
	x_1	x_2	h_1	h_2	O
T_1	0	0	-1	-1	-1
T_2	0	1	-1	1	1
T_3	1	0	1	-1	1
T_4	1	1	-1	-1	-1

$$h_1 = \mathrm{sgn}(x_1 - x_2 - 0.5)$$

$$h_2 = \mathrm{sgn}(x_2 - x_1 - 0.5)$$

$$\mathbf{o} = \begin{cases} 1, & if \quad (w'_{2,2}h_2 + w'_{2,1}h_1) \geq 0 \\ -1, & otherwise \end{cases}$$

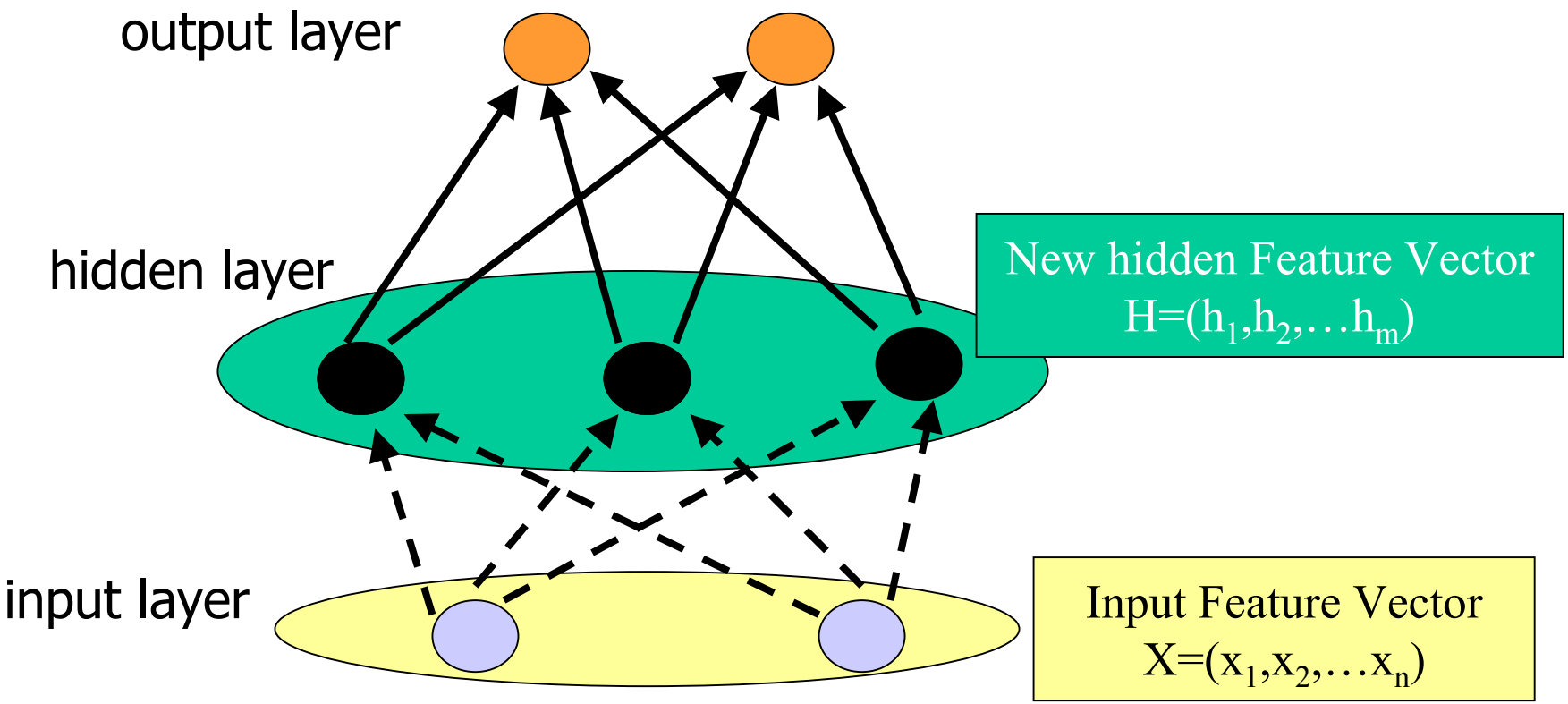
Hidden Layer



- Each hidden layer maps to a new feature space
- Each hidden node is a new constructed feature
- Original Problem may become separable (or easier)

MultiClass and Multilayer

(How to train the weight)



Mapping: Space to Space

- Input Space -> Hidden Space:

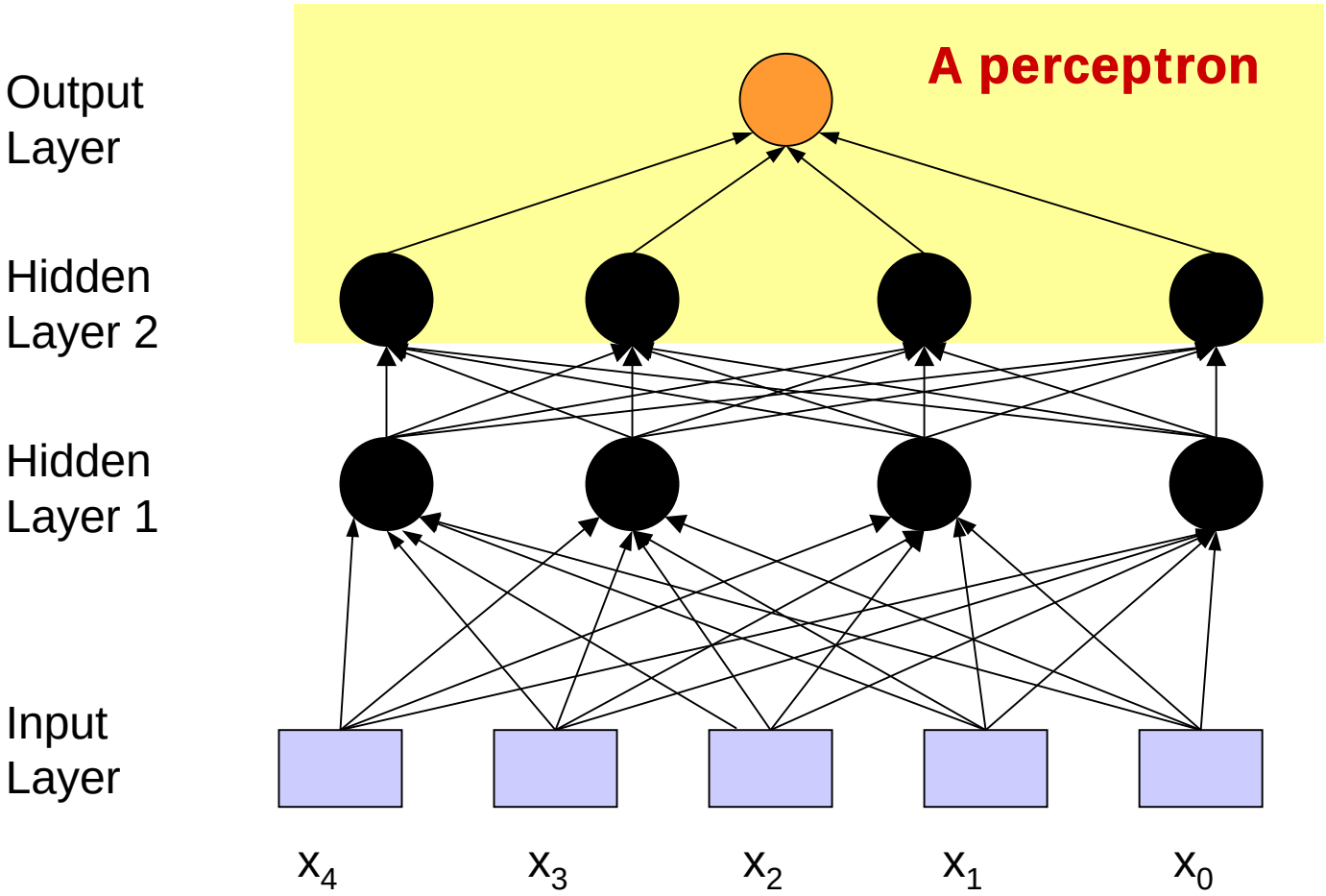
$$\vec{\mathbf{H}} \Leftarrow \vec{\mathbf{X}} \bullet \mathbf{W}_1 \quad (\text{Note: } \mathbf{W}_1, \text{ a matrix})$$

- Hidden Space -> Output Space:

$$\vec{\mathbf{O}} \Leftarrow \vec{\mathbf{H}} \bullet \mathbf{W}_2 \quad (\text{Note: } \mathbf{W}_2, \text{ a matrix})$$

Problem One: How many hidden layers?

$$\vec{O} \Leftarrow \vec{H}_m \Leftarrow \ldots \Leftarrow \vec{H}_2 \Leftarrow \vec{H}_1 \Leftarrow \vec{X}$$



If hidden layer is linear…

$$\vec{\mathbf{H}}_1 \Leftarrow \vec{\mathbf{X}} \bullet \mathbf{W}_1$$

$$\vec{\mathbf{H}}_2 \Leftarrow \vec{\mathbf{H}}_1 \bullet \mathbf{W}_2$$

…

$$\vec{\mathbf{H}}_m \Leftarrow \vec{\mathbf{H}}_{m-1} \bullet \mathbf{W}_m$$

$$\vec{\mathbf{O}} = \vec{\mathbf{H}}_m \bullet \mathbf{W}_{m+1}$$

$$= (\vec{\mathbf{H}}_{m-1} \bullet \mathbf{W}_m) \bullet \mathbf{W}_{m+1}$$

$$= \vec{\mathbf{H}}_{m-1} \bullet \mathbf{W}_m \bullet \mathbf{W}_{m+1}$$

…

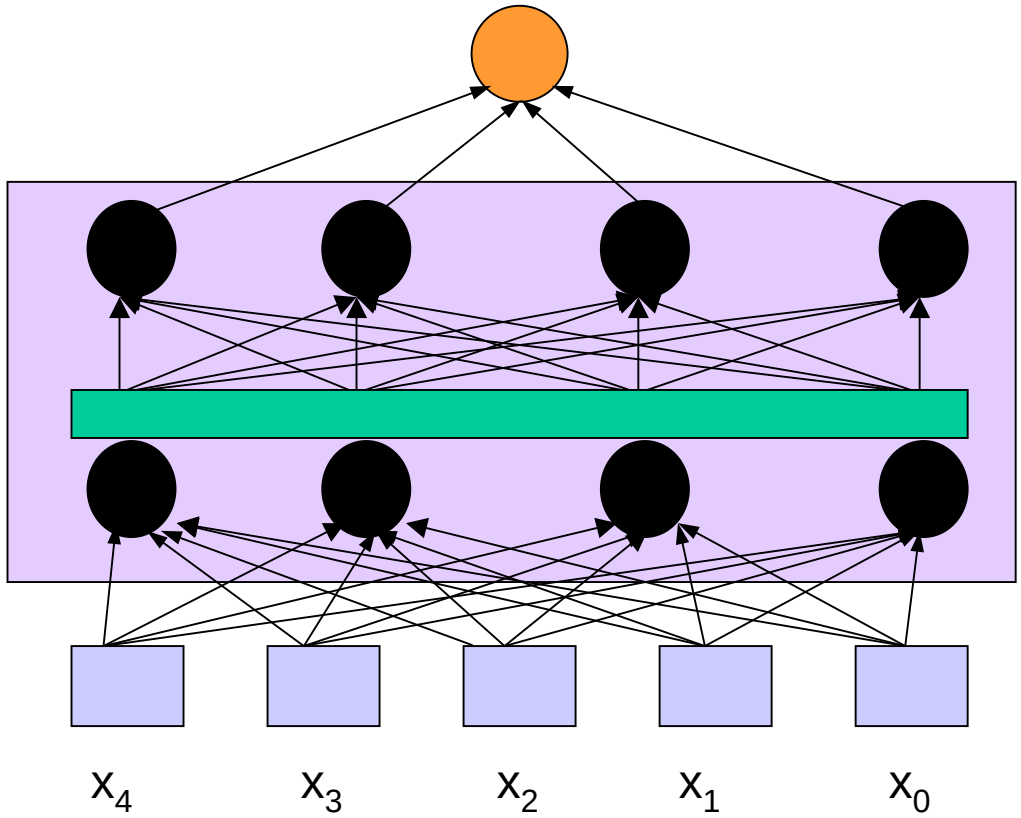
$$= \vec{\mathbf{X}} \bullet \mathbf{W}_1 \bullet \mathbf{W}_2 \ldots \bullet \mathbf{W}_m \bullet \mathbf{W}_{m+1}$$

$$= \vec{\mathbf{X}} \bullet [\mathbf{W}_1 \bullet \mathbf{W}_2 \ldots \bullet \mathbf{W}_m \bullet \mathbf{W}_{m+1}]$$

Let:

$$\vec{\mathbf{O}} \Leftarrow \vec{\mathbf{X}} \bullet \mathbf{W}$$

where, $\mathbf{W} = [\mathbf{W}_1 \bullet \mathbf{W}_2 \ldots \bullet \mathbf{W}_m \bullet \mathbf{W}_{m+1}]$



No hidden layer is needed !

Representational Power

- Perceptron: Can learn only linearly separable functions
- Boolean functions
  - Every boolean function can be represented by network with single hidden layer
- Continuous functions
  - Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989, Hornik 1989]
  - Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988]

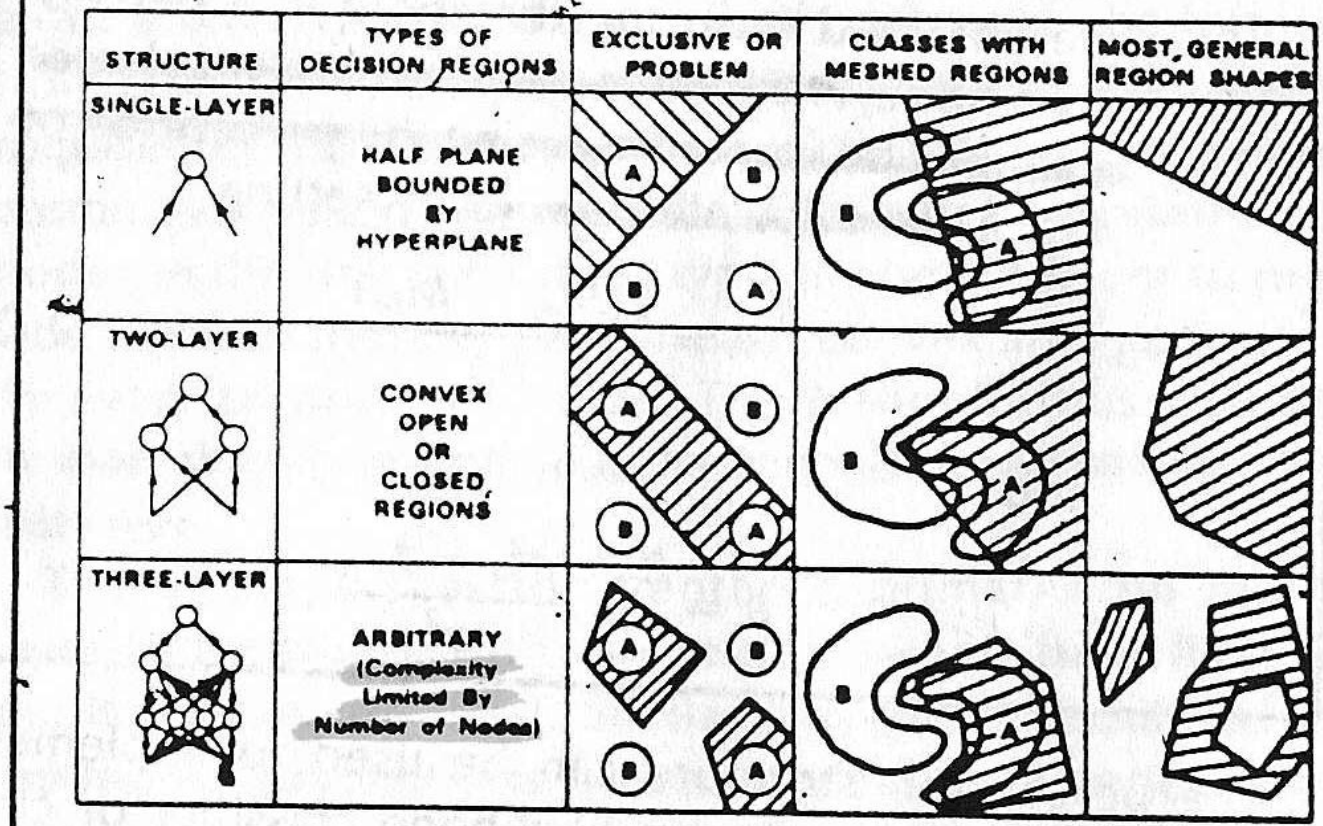
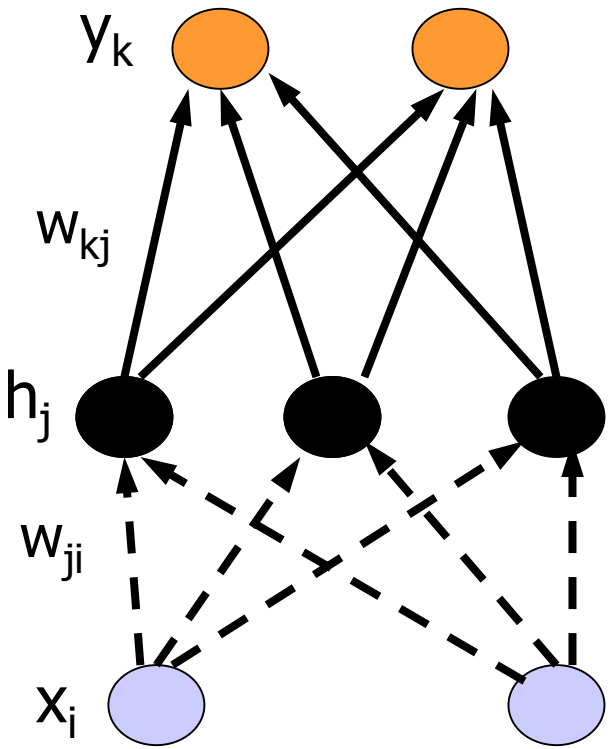


Figure 14. Types of decision regions that can be formed by single- and multi-layer perceptrons with one and two layers of hidden units and two inputs. Shading denotes decision regions for class A. Smooth closed contours bound input distributions for classes A and B. Nodes in all nets use hard limiting nonlinearities.

Outline

- Perceptrons
- Perceptrons learning
- Sigmoid Unit
- MultiClass
- Multi-layer networks
- **Backpropagation Learning**
- Others

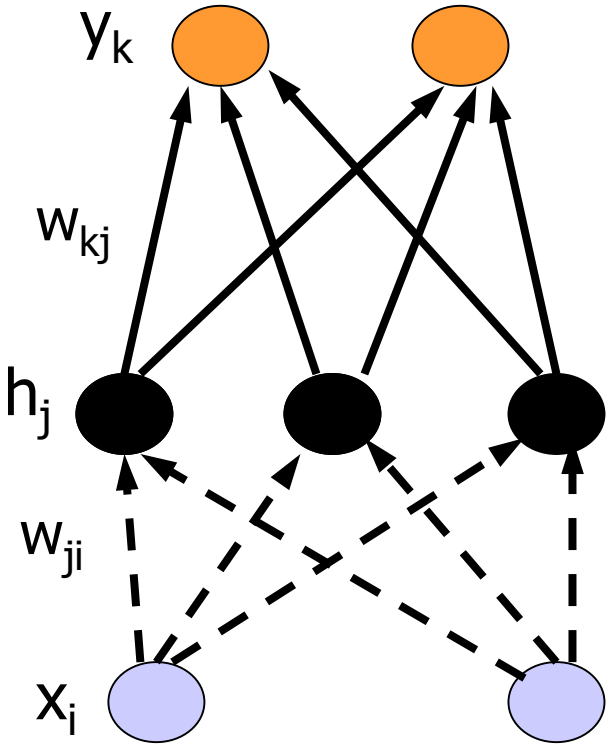
Feed-Forward & Backpropagation



Backward step:
propagate errors from
output to hidden layer

Forward step:
Propagate activation from
input to output layer

General Feed-forward

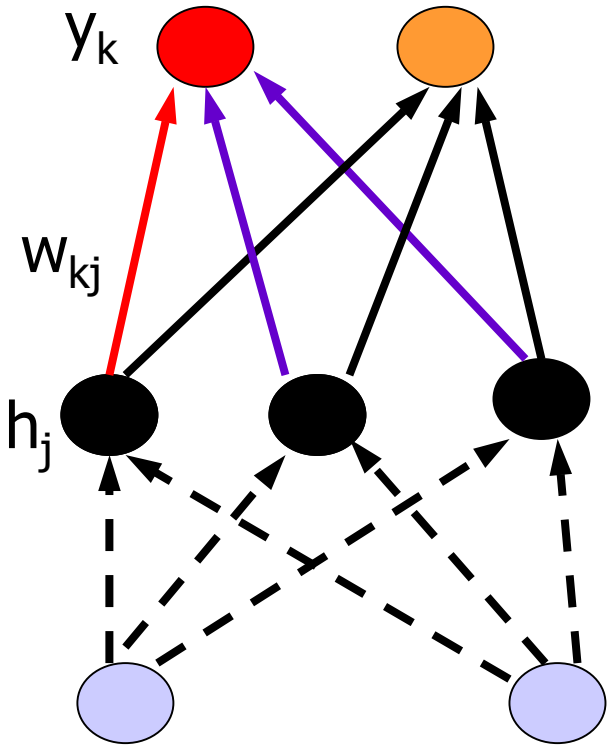


$$y_k = F_k(\mathbf{x}) = g\left(\sum_{j=1} w_{kj} \bullet h_j + w_{k0}\right)$$

$$= g\left(\sum_j w_{kj} \bullet g\left(\sum_i w_{ji} x_i + w_{j0}\right) + w_{k0}\right)$$

$$where, \quad h_j = g\left(\sum_i w_{ji} x_i + w_{j0}\right)$$

Training-Rule for Weights to the Output Layer: Feed-forward



$$E[w_k] = \frac{1}{2}\sum_k (t_k - y_k)^2$$

$$\frac{\partial E[w_k]}{\partial w_{kj}}$$

$$= -(t_k - y_k)\cdot\frac{\partial y_k}{\partial w_{kj}}$$

$$Let,\quad y_k = g(\sum_j w_{kj} h_j)$$

$$=>$$

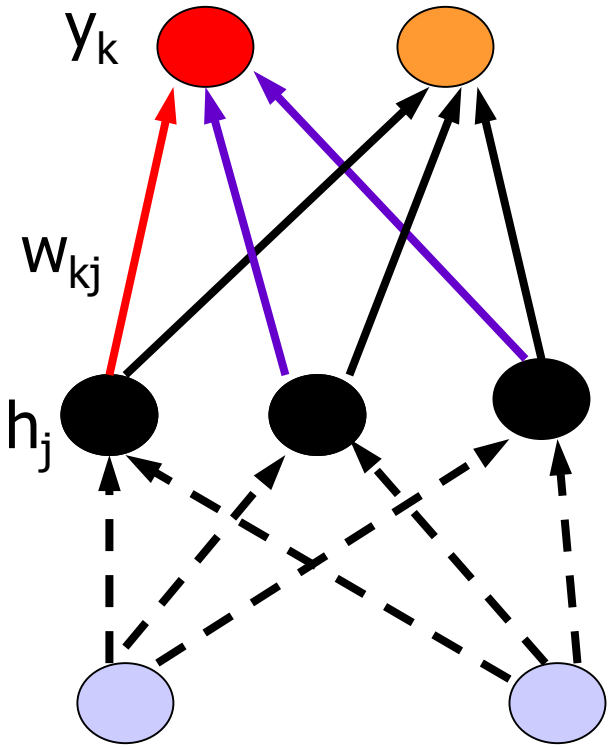
$$\frac{\partial y_k}{\partial w_{kj}} = \frac{\partial g(\sum_j w_{kj} h_j)}{\partial \sum_j w_{kj} h_j}\cdot\frac{\partial \sum_j w_{kj} h_j}{\partial w_{kj}}$$

$$\frac{\partial E[w_k]}{\partial w_{kj}} = -(t_k - y_k)\cdot g'(\sum_j w_{kj} h_j)\cdot h_j$$

$$=> \quad \Delta w_{kj} = \eta \delta_k h_j$$

$$where,\quad \delta_k = -(t_k - y_k)\cdot g'(\sum_j w_{kj} h_j)$$

For Top out(Sigmoid)



if $y_k = g(\sum_j w_{kj} h_j) = Sigmoid(\sum_j w_{kj} h_j)$

$$\frac{\partial E_d[w_k]}{\partial w_{kj}} = -y_k(1-y_k)(t_k - y_k)(h_j)$$

$$\Delta w_{kj} = \eta\, y_k(1-y_k)(t_k - y_k)(h_j)$$
$$= \eta\, \delta_k h_j$$

where,

$$\delta_k = y_k(1-y_k)(t_k - y_k)$$

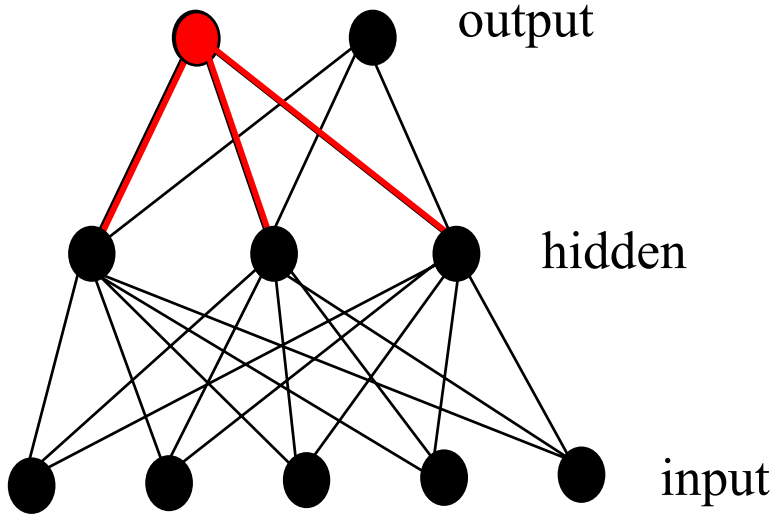
Error on Top

- First calculate error of output units and use this to change the top layer of weights.

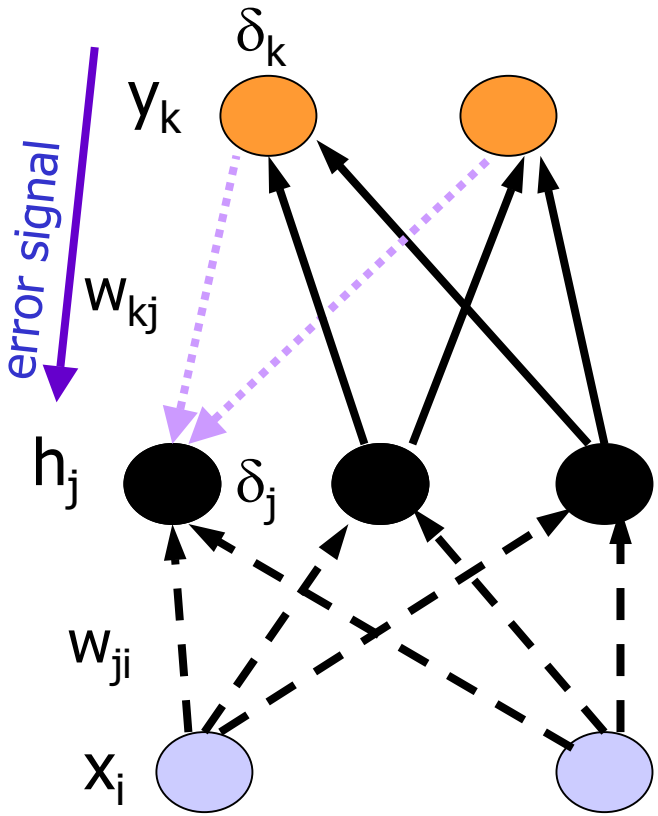
Current output: y_k=0.2
Correct output: t_k=1.0
Error $\delta_k = y_k(1–y_k)(t_k–y_k)$
0.2(1–0.2)(1–0.2)=0.128

Update weights into k

$$\Delta w_{kj} = \eta \delta_k h_j$$



Training-Rule for Weights to the Hidden Layer



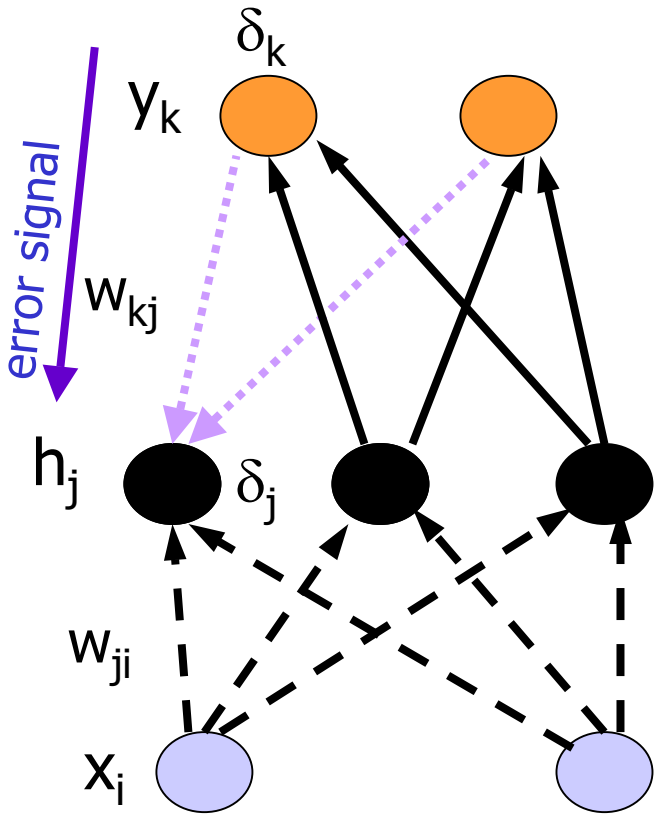
A problem:
No target values t for hidden layer units.

Error: Backpropagation

$$\delta_j = \Sigma_k w_{kj} \delta_k h_j (1-h_j)$$

$$\Delta w_{ji} = \eta \delta_j x_i$$

Training-Rule for Weights to the Hidden Layer: Backpropagation



$$E[w_{ji}] = \frac{1}{2}\sum_k (t_k - y_k)^2$$

$$= \frac{1}{2}\sum_k (t_k - \sigma(\sum_j w_{kj} h_j))^2$$

$$= \frac{1}{2}\sum_k (t_k - \sigma(\sum_j w_{kj}\sigma(\sum_i w_{ji} x_i))^2$$

$$\frac{\partial E[w_{ji}]}{\partial w_{ji}}$$

$$= -\sum_k (t_k - y_k)\sigma_k'(a) w_{kj} \sigma_j'(a) x_i$$

$$= -\sum_k \delta_k \; w_{kj} \; \sigma_j'(a) \; x_i$$

$$= -\sum_k \delta_k \; w_{kj} \; h_j(1-h_j) \; x_i$$

$$\Delta w_{ji} = \eta \delta_j x_i$$

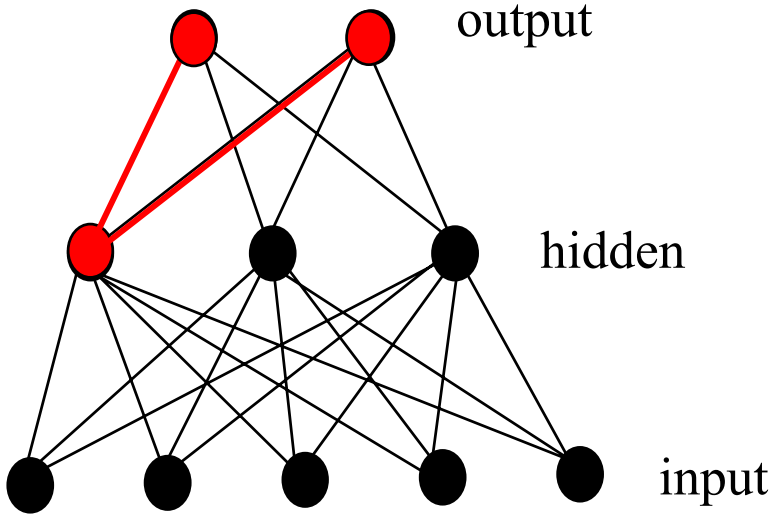
$where,$

$$\delta_j = \sum_k \delta_k w_{kj} h_j (1 - h_j)$$

Errors on Hidden

- Next calculate error for hidden units based on errors on the output units it feeds into.

$$\delta_j = h_j(1-h_j)\sum_k \delta_k w_{kj}$$



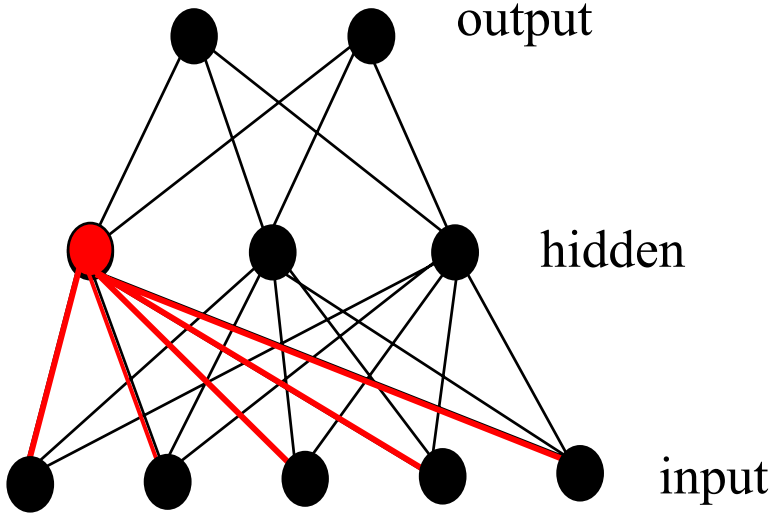
Errors on Hidden

- Finally update bottom layer of weights based on errors calculated for hidden units.

$$\delta_j = h_j(1-h_j)\sum_k \delta_k w_{kj}$$

Update weights into j

$$\Delta w_{ji} = \eta \delta_j x_i$$



Training Algorithm

- Three Protocols:
  - Batch Training: all examples are presented to the network before learning takes place, but usually, many passes will be made;
  - Online Training: each example will be presented once and only once. No memory to store any example.
  - Stochastic training: examples are chosen randomly from the training set and weight will be updated for each example presentation.

Stochastic Backpropagation

Begin (for three layers)

Initialize n_h (number of nodes in hidden layer)

Initialize ε (stop criterion), η (learning rate), **w** (weight);

m $\leftarrow$ 0;

do m $\leftarrow$ m+1

$\mathbf{x}^m \leftarrow$ randomly chosen example

$w_{ji} \leftarrow w_{ji} + \eta\delta_j x_i;\ \ w_{kj} \leftarrow w_{kj} + \eta\delta_k h_j$

until $\left\|\nabla \mathrm{E}[\mathbf{w}]\right\| < \varepsilon$ where, $\nabla \mathrm{E}[\mathbf{w}] = \dfrac{\partial \mathrm{E}[\mathbf{w}]}{\partial \mathbf{w}}$

return **w**

End

Batch Backpropagation

Begin (for three layers)

Initialize n_h (number of nodes in hidden layer)

Initialize ε (stop criterion), η (learning rate), $\mathbf{w}$ (weight);

r $\leftarrow$ 0;

do r $\leftarrow$ r+1 (increment epoch)

m $\leftarrow$ 0; $\Delta w_{ji} \leftarrow 0$; $\Delta w_{kj} \leftarrow 0$

do m $\leftarrow$ m+1

$\mathbf{x}^m \leftarrow$ chosen example

$\Delta w_{ji} \leftarrow \Delta w_{ji} + \eta \delta_j x_i$; $\Delta w_{kj} \leftarrow \Delta w_{kj} + \eta \delta_k h_j$

until (m=n)

$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$; $w_{kj} \leftarrow w_{kj} + \Delta w_{kj}$

until $\|\nabla E[\mathbf{w}]\| < \varepsilon$ where, $\nabla E[\mathbf{w}] = \dfrac{\partial E[\mathbf{w}]}{\partial \mathbf{w}}$

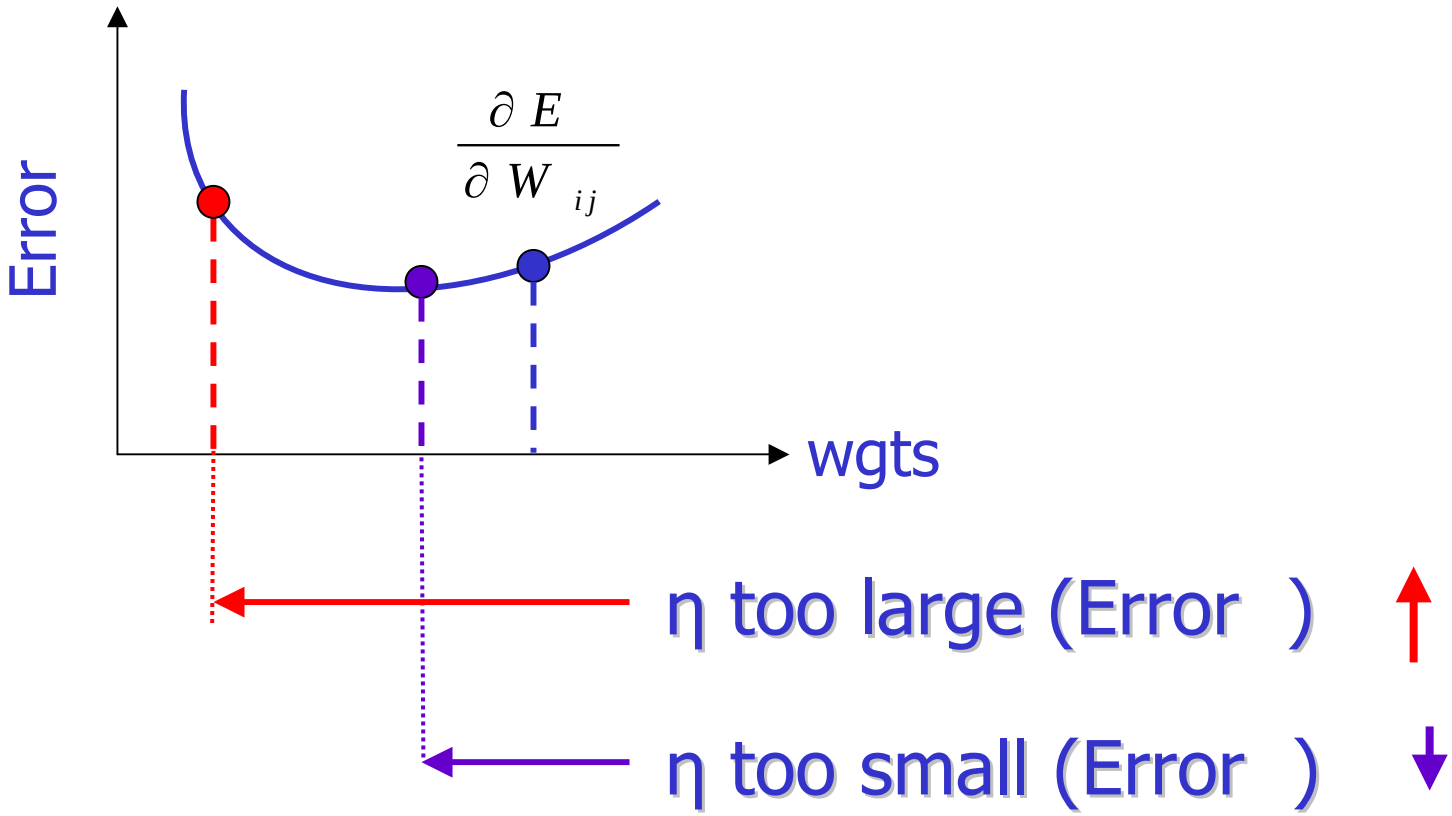
return $\mathbf{w}$

End

Outline

- Perceptrons
- Perceptrons learning
- Sigmoid Unit
- MultiClass
- Multi-layer networks
- Backpropagation Learning
- **Others**

Choosing η (the learning rate)



Adjusting η

0. Let $\eta = 0.25$
1. Measure ave. error over *k* examples

   - call this $\mathbf{E}_{\mathbf{before}}$
2. Adjust wgts according to neural-net learning algorithm being used
3. Measure ave error on same *k* examples

   - call this $\mathbf{E}_{\mathbf{after}}$
4. If $\mathbf{E}_{\mathbf{after}} > \mathbf{E}_{\mathbf{before}}$,

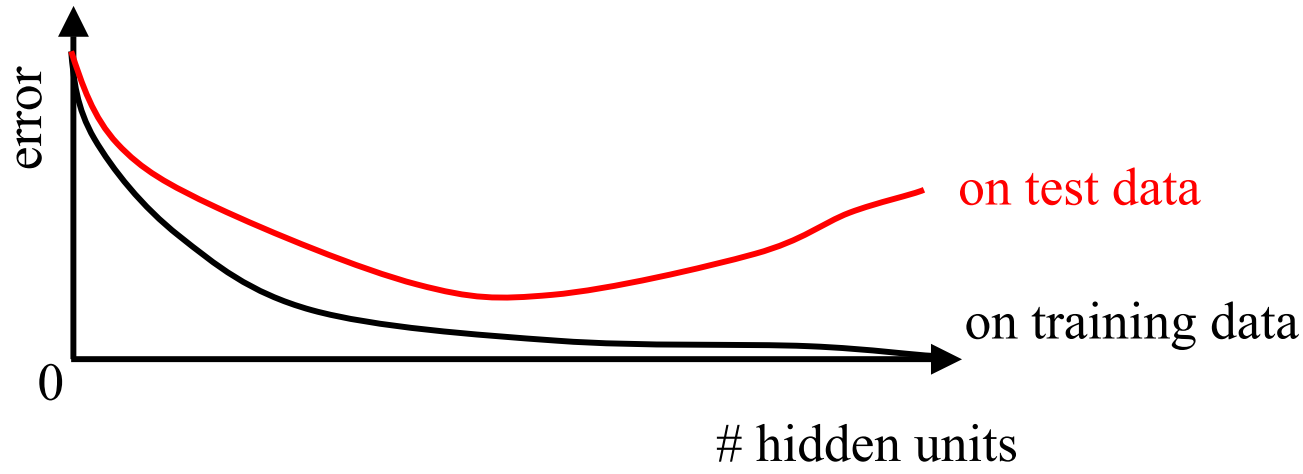
   then $\eta \leftarrow \eta * 0.99$

   else $\eta \leftarrow \eta * 1.01$
5. Go to 1

Note: *k* can be all training examples, also could be a subset

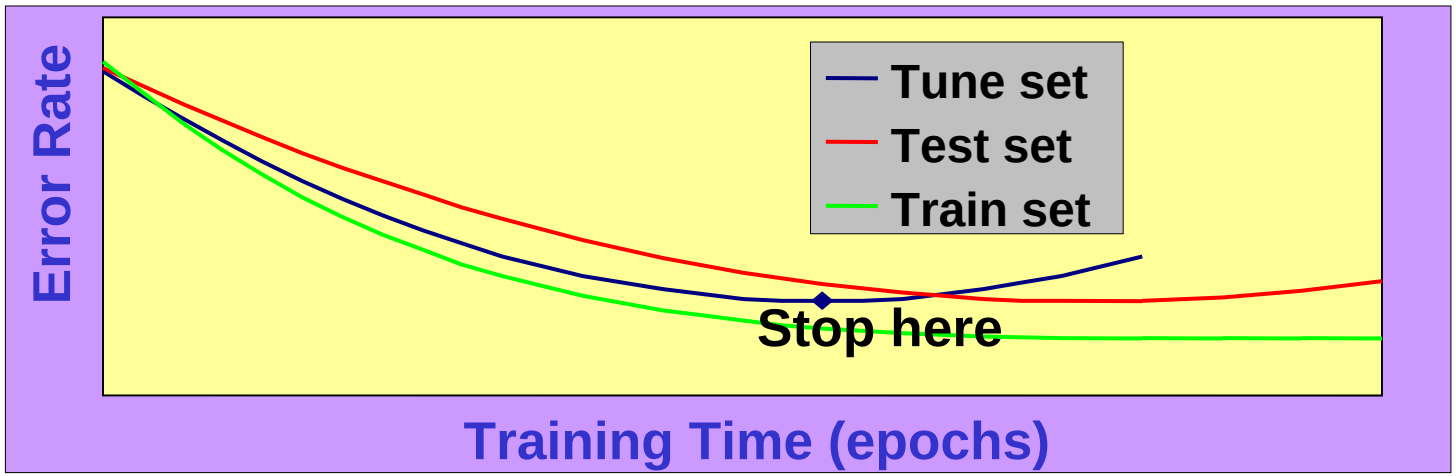
Number of Hidden Units

- Too few hidden units prevents the network from adequately fitting the data.
- Too many hidden units are easy to memorize the training set (or parts of it) and not generalize, thus can result in over-fitting.



Overfitting Avoidance

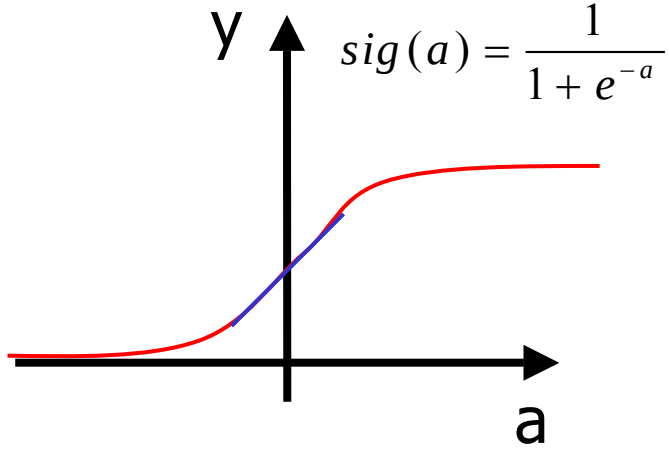
- Split data to:
  - Training set: used to update the weights
  - Tuning set: used in the stopping criterion
  - Test set: used in evaluating generalization error (performance)



Overfitting Avoidance: Method 2

- Sigmoid almost linear around zero
- Small weights imply decision surfaces that are almost linear
- Instead of trying to minimize only the error, minimize the error while penalizing for large weights

$$Cost(\vec{w}) = \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 + \frac{1}{2}\gamma\|w\|^2$$



i.e. Cost = Error Rate + Network Complexity

- This imposes a preference for smooth and simple (linear) surfaces

Overfitting Avoidance(Cont.)

$$\frac{\partial Cost(\vec{w})}{\partial w} = \frac{\partial E(\vec{w})}{\partial w} + \gamma w_{ij}$$

$$where, \quad E(\vec{w}) = \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2$$

$$So, \quad \Delta w_{ij} = -\eta \cdot \delta \cdot out_j - \eta \cdot \lambda \cdot w_{ij}$$

Weights decay toward zero

Empirically improves generalization

Classification with Neural Networks

- Select
  - Number of hidden layers
  - Number of hidden units
  - Connectivity
  - Typically: one hidden layer, hidden units is a small fraction of the input units, full connectivity
- Select error function
  - Typically: minimize mean squared error (with penalties for large weights), maximize log likelihood of the data

Classification with Neural Networks

- Select a training method:
  - Typically gradient descent (corresponds to vanilla Backpropagation)
  - Other optimization methods can be used:
    - For example, Line-Search Methods
      - Congugate Gradient methods
      - Newton and Quasi-Newton Methods
  - Select stopping criterion