# Fundamentals of Artificial Neural Network

林伯慎 Bor-shen Lin
bslin@cs.ntust.edu.tw
http:www.cs.ntust.edu.tw/~bslin

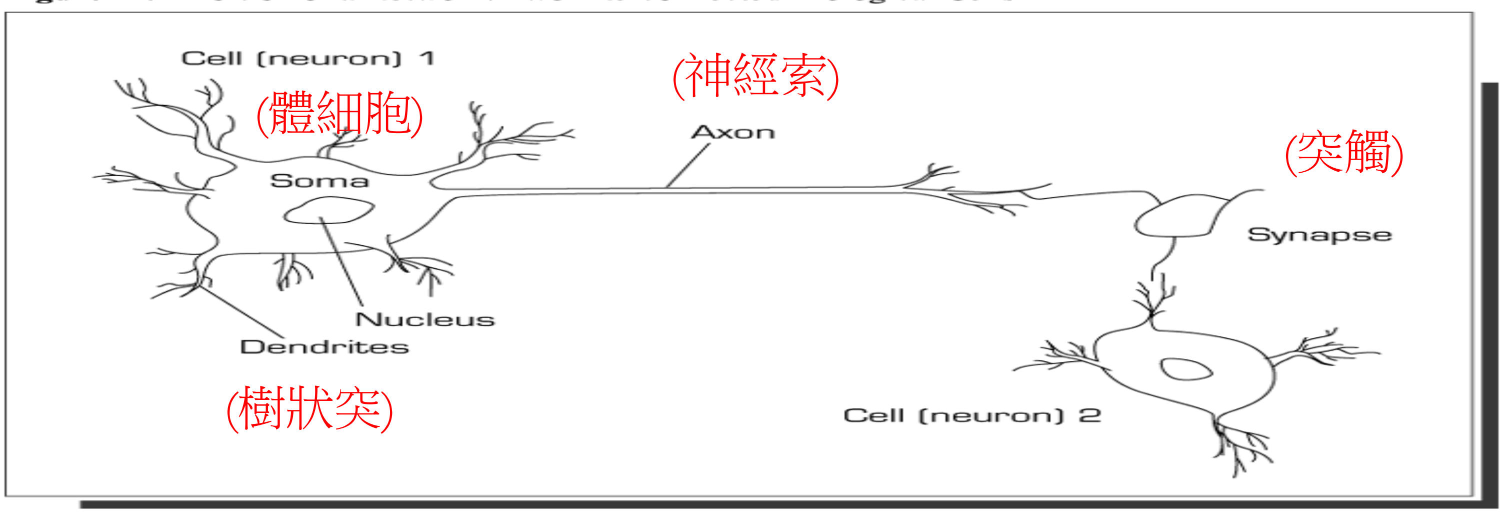
# Artificial Neural Network, Why?

- Mimic how human brains function
  - Human brain is superior to digital computers in many ways
    - A baby can recognize many objects
  - Computers are good at arithmetic
- Human brains are
  - Robust and fault-tolerant
  - Flexible and adaptive (learning)
  - Can deal with fuzzy, probabilistic, noisy or inconsistent information
  - Small, compact & consumes low power

# HUMAN BRAIN

- 50 to 150 billion neurons in brain
- Neurons grouped into networks
  - Axons send outputs to cells
  - Received by dendrites, across synapses

**Figure 12.4** Portion of a Network: Two Interconnected Biological Cells



# Functions of Neurons

- Transmission of Signal
  - Complicated chemical process
  - Transmitter substances are released from the sending site
  - Raise or lower electrical potential inside the body of receiving cell
  - If the potential reaches a threshold, a pulse of a fixed strength and duration is sent (fired)
  - After firing, the cell has to wait a time called refractory period before it can fire again

# Mathematical Model

- $y_i(t+\Delta t) = g(\Sigma_j\, w_{ji}x_j(t) - \mu_i)$.
- Activation function $g(h)$

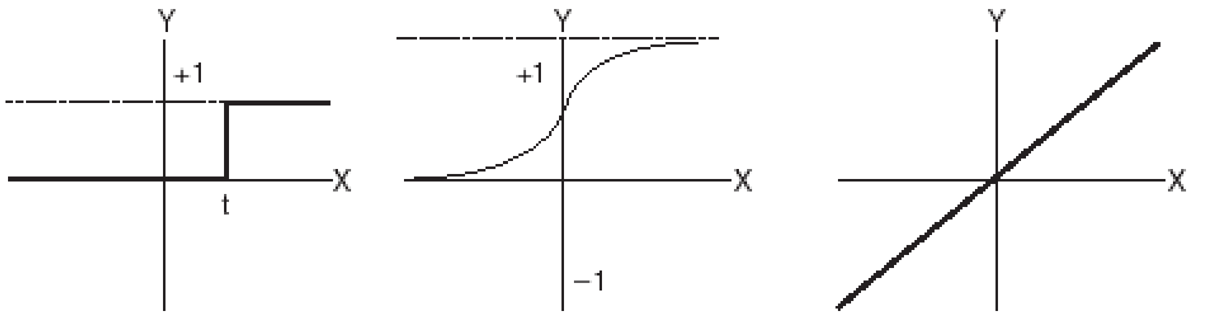
  $g(h) = 1$ if $h \geq 0$

  $0$ otherwise.
- $w_{ji}$: the strength of synapse connecting neuron $j$ to neuron $i$
- $\mu_i$ : threshold value for neuron $i$
- If the weighted sum of the inputs to the neuron $i$ exceeds the threshold $\mu_i$, the neuron fires and the output becomes $1$.

# Perceptron: Artificial Neuron

- Artificial neurons are based on biological neurons.
- Each neuron in the network receives one or more inputs.
- An activation function is applied to the inputs, which determines the output of the neuron – the activation level.

- The charts on the right show three typical activation functions g(·).




(a) Step funtion (b) Sigmoid funtion (c) Linear funtion

# Activation Function

- Characteristics
  - Monotonic (with positive slope)
  - Range: between 0 and 1
  - Differentiable s.t. easier for optimization
- Popular types
  - Sigmoid

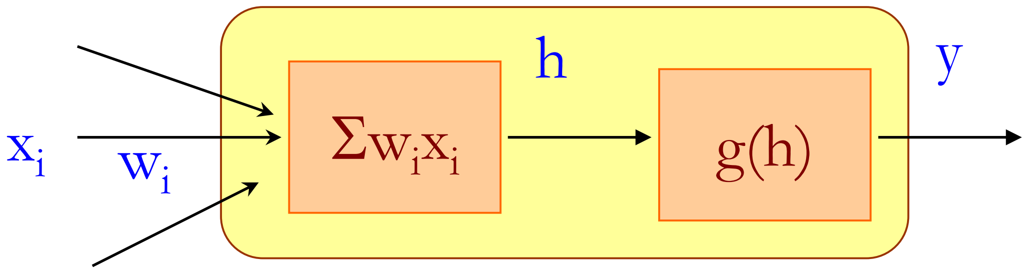
    $g(h) = \frac{1}{1+e^{-h}}$
  - Hyperbolic tangent (denoted as *tanh*)

    $g(h) = \frac{e^{h}-e^{-h}}{e^{h}+e^{-h}}.$
  - Rectified linear (denoted as RELU)

    $g(h) = max(0, h)$

# Perceptron

- A perceptron is a single neuron that classifies a set of inputs into one of two categories.
- If the inputs are in the form of a grid, a perceptron can be used to recognize visual images of shapes.
- The perceptron usually uses a step function, which returns 1 if the weighted sum of inputs exceeds a threshold, and –1 otherwise.



# Learning OR Function

- OR
  - 0 0 → 0
  - 0 1 → 1
  - 1 0 → 1
  - 1 1 → 1
- Initial setting
  - $w_1 = -0.2$
    $w_2 = 0.4$
    $a = 0.2$

| Epoch | X1 | X2 | Expected Y | Actual Y | Error | w1 | w2 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | −0.2 | 0.4 |
| 1 | 0 | 1 | 1 | 1 | 0 | −0.2 | 0.4 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0.4 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0.4 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0.4 |
| 2 | 0 | 1 | 1 | 1 | 0 | 0 | 0.4 |
| 2 | 1 | 0 | 1 | 0 | 1 | 0.2 | 0.4 |
| 2 | 1 | 1 | 1 | 1 | 0 | 0.2 | 0.4 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0.2 | 0.4 |
| 3 | 0 | 1 | 1 | 1 | 0 | 0.2 | 0.4 |
| 3 | 1 | 0 | 1 | 1 | 0 | 0.2 | 0.4 |
| 3 | 1 | 1 | 1 | 1 | 0 | 0.2 | 0.4 |

# Training a Perceptron

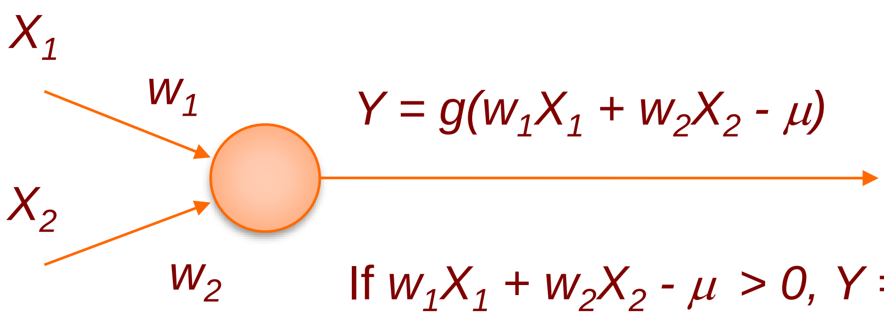
- A perceptron can be trained as follows:
  - First, inputs are given random weights (usually between –0.5 and 0.5).
  - An item of training data is presented. If the perceptron mis-classifies it, the weights are modified according to the following:

    $w_i' \leftarrow w_i + a \cdot \mathbf{x_i} \cdot \mathbf{e}$ ($e = d - y$, $d$ : desired output)
  - e is the size of the error
  - a is the learning rate, between 0 and 1.
- $h_i' = w_i'x_i = (w_i + ax_ie)x_i = w_ix_i + ax_i^2e = h_i + ax_i^2e$

  if $e < 0$ : $d < y$, $y$ too big → updated weight $w_i'$

  → decrease $h$ ($h_i' < h_i$) → decrease $y$ (since $g(\cdot)$ monotonic))

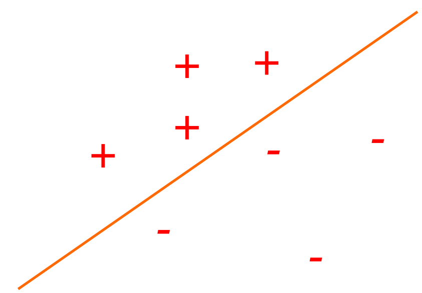
# Linear Decision Function of a Perceptron




If $w_1X_1 + w_2X_2 - \mu > 0$, $Y = 1$

Otherwise $Y = 0$

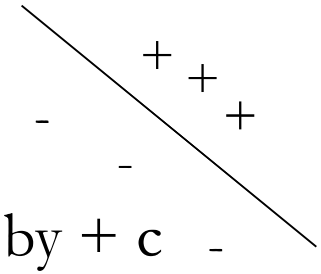
$f(X_1, X_2)$
$= w_1X_1 + w_2X_2 - \mu = 0$




- A family of linear functions with adjustable parameters $w_1$, $w_2$ and $\mu$.
- A learning procedure can be performed to update the parameters of the linear function so as to find out the decision boundary that separates the 2D points of the two classes.

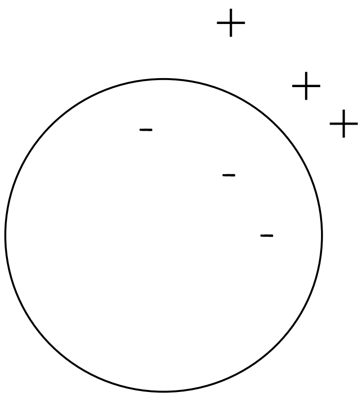
# Decision Functions for Classifiers

Decision boundary $f(x, y) = 0$ can separate the 2-D plane into two regions (+ or –).



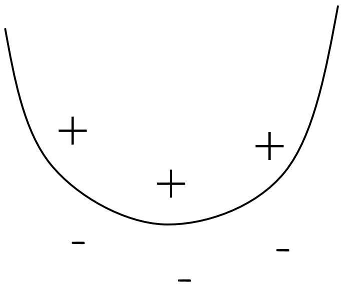
$f(x,y) = ax + by + c$

$(a, b > 0)$



$f(x,y) = (x-x_0)^2 + (y-y_0)^2 - r^2$

$f(x_0, y_0) = - r^2 < 0$

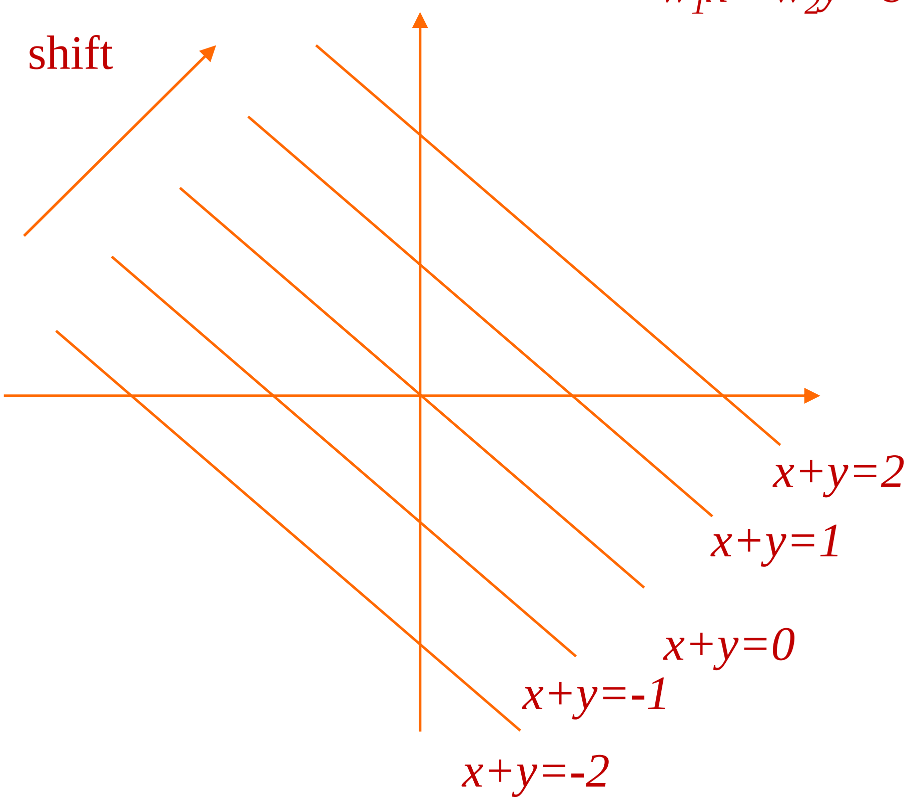


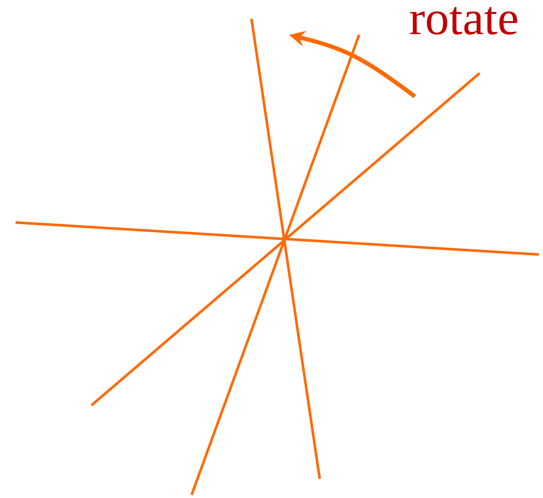
$f(x,y) = y - ax^2 - bx - c$

$(b^2-4ac > 0)$

# Family of Functions

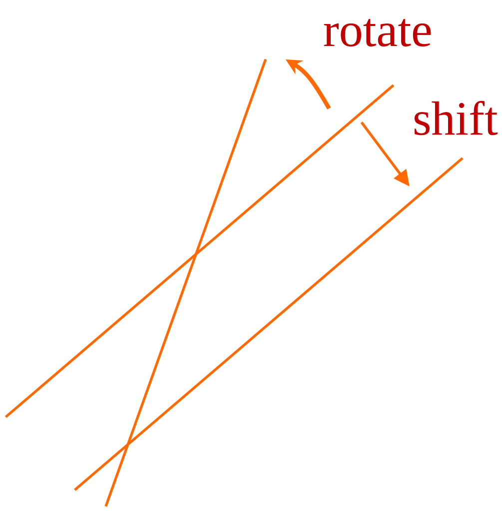
$w_1x + w_2y + b = 0$

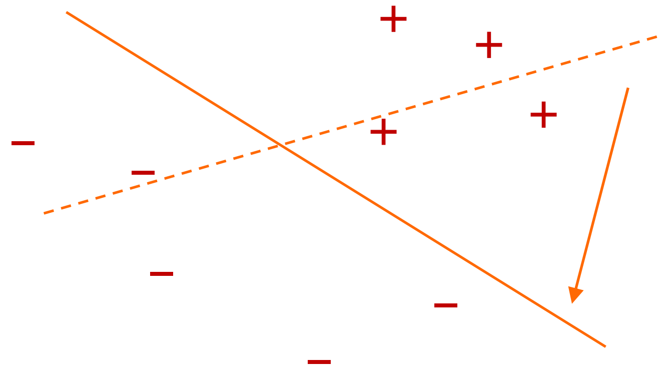



- Rotate the line by changing $w_1$ and $w_2$
- Shift the line by changing b

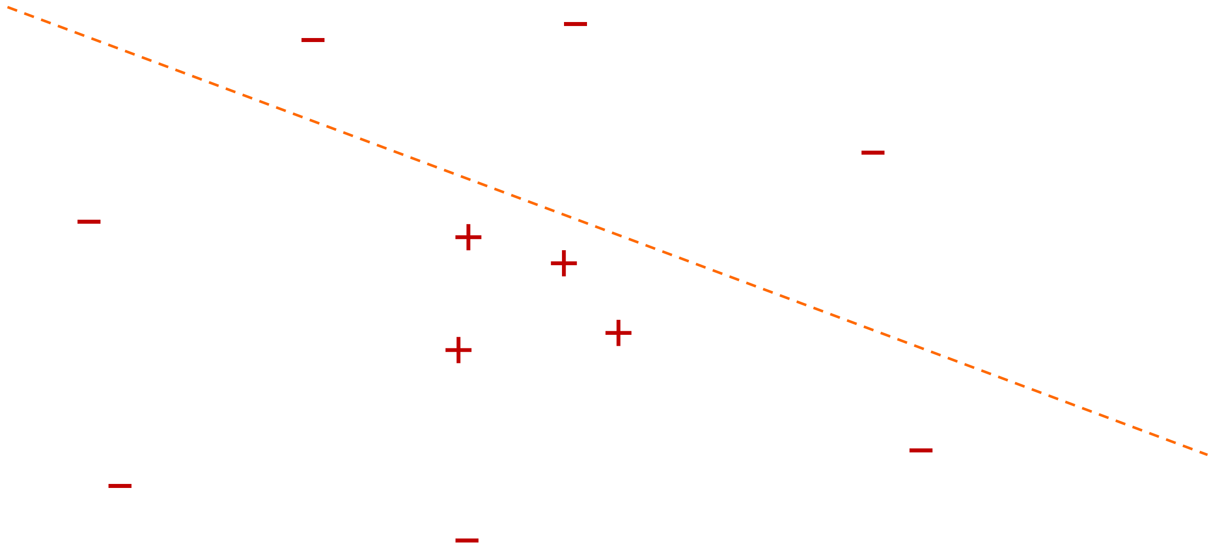
# How Does A Neuron Learn?



- $w_1x_1 + w_2x_2 + b = 0$
- Change orientation by learning $w_1$ and $w_2$
- Shift location by learning b

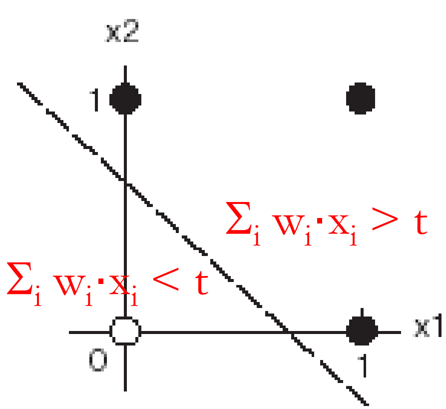
# Problem of Linear Classifier

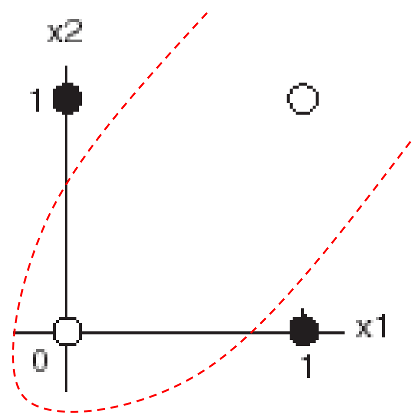



- NOT linearly separable (for single neuron)

# LIMITATION OF A PERCEPTRON

- A Single Perceptron can only classify linearly separable functions using step function.
- Can NOT model the problem that is not linearly separable such as simple Exclusive-OR.
  - No line can separate the black/white dots or EOR




$X = \Sigma_i\, w_i \cdot x_i$

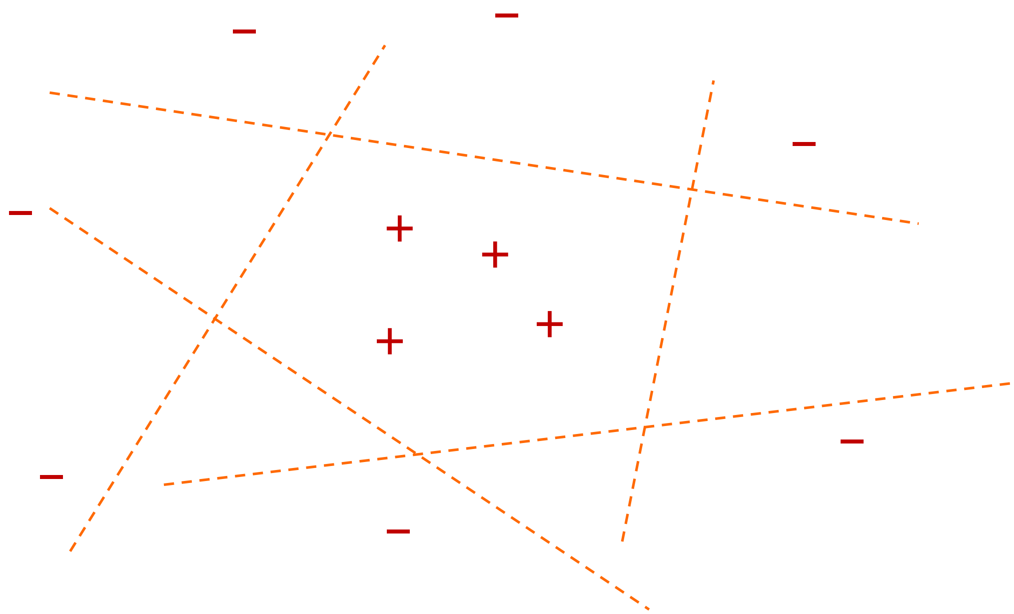
$Y = +1$ if $X > t$

$\quad -1$ if $X < t$

$\Sigma_i\, w_i \cdot x_i > t \qquad +1$

$\Sigma_i\, w_i \cdot x_i < t \qquad -1$

# Multiple Layers of Neurons

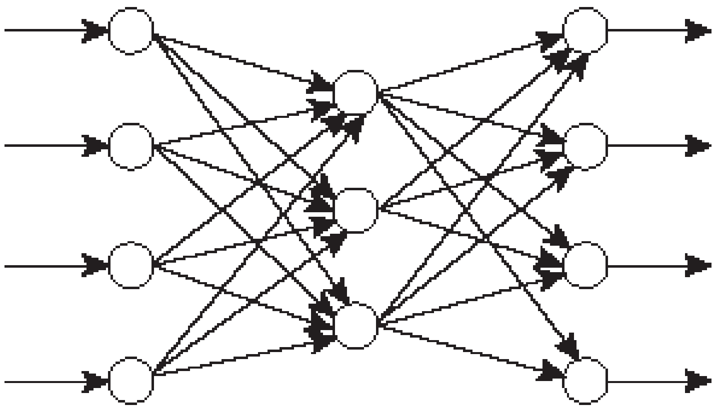



- Multiple neurons (every one is linear)
- Multiple layers to form decision boundary

# MULTILAYER PERCEPTRONS (MLP)

- Multilayer neural networks can classify a range of functions, including nonlinearly separable ones.

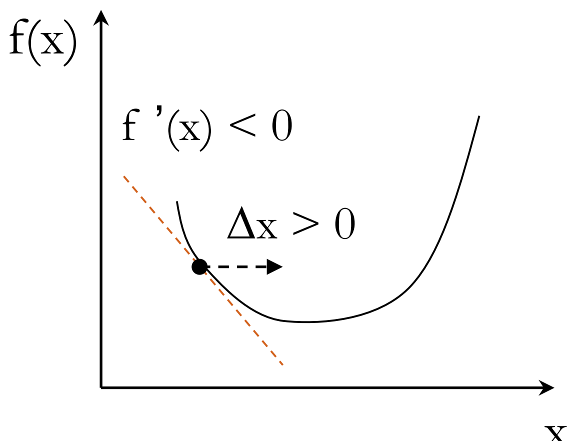
- Each input layer neuron connects to all neurons in the hidden layer.
- The neurons in the hidden layer connect to all neurons in the output layer.



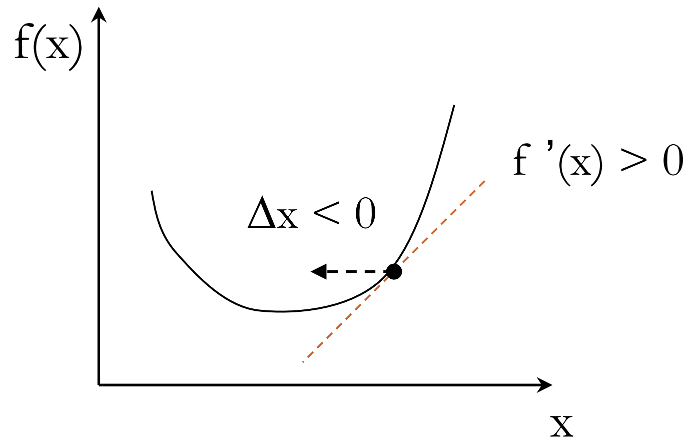
**A feed-forward network**

# Gradient Descent (for Minimization)

- To find minimum of f(x) from arbitrary point x
  - Local minimum can be obtained




If f '(x) < 0, increase x (dx > 0)

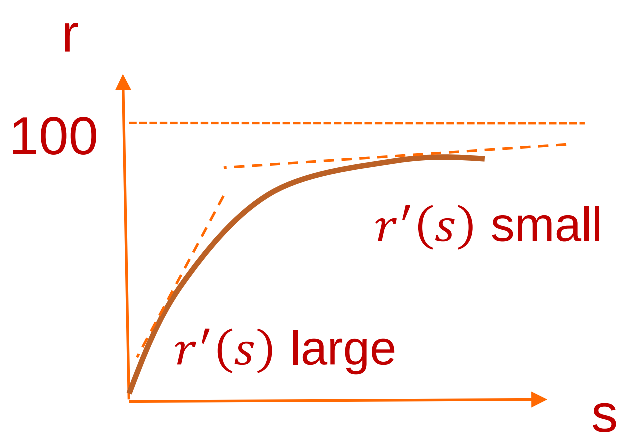



If f '(x) > 0, decrease x (dx < 0)

*$\Delta x = -\varepsilon(df/dx)$ to find minimal f(x)*

*$x_{new} = x_{old} + \Delta x$, $\Delta x = -\varepsilon(df/dx)$*

# Meaning of Derivative




- Try to increase scores by studying more
- Higher derivative: $r'(s) = \dfrac{dr}{ds}$ large
  - $dr = r'(s)ds \rightarrow ds$ leads to more increase of $r$ (higher $dr$)
- Lower derivative : $r'(s) = \dfrac{dr}{ds}$ small
  - $dr = r'(s)ds \rightarrow ds$ leads to less increase of $r$ (lower $dr$)

# Sign of Derivative

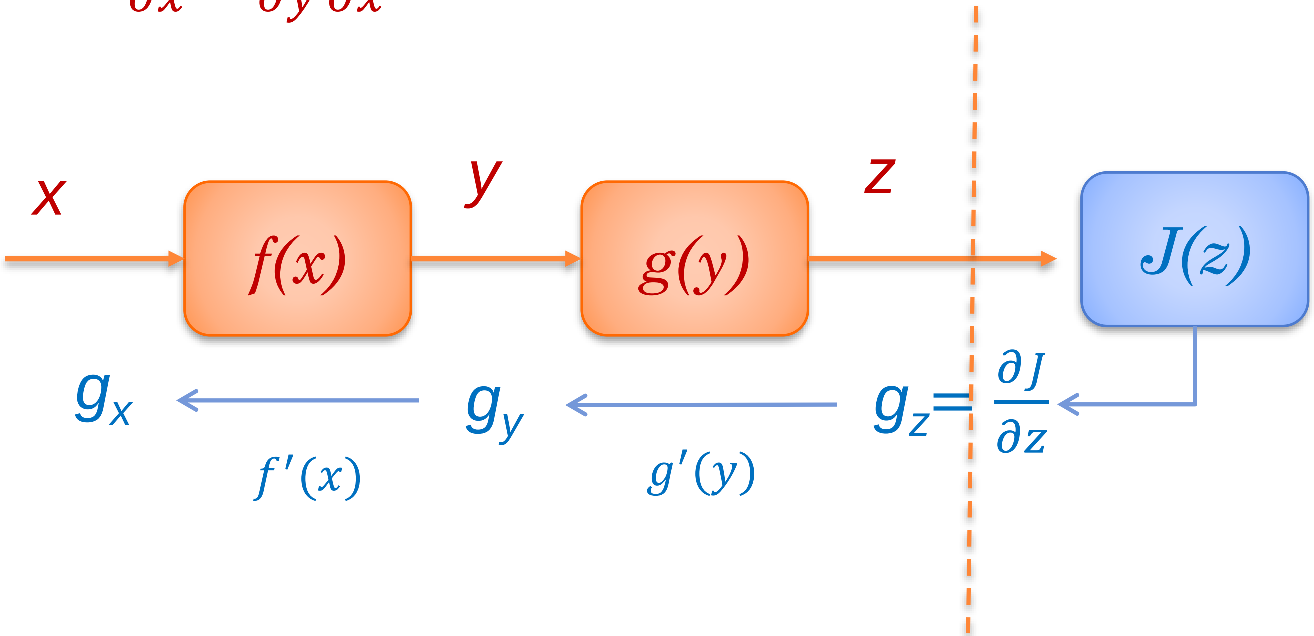
- Positive correlation
  - More study time leads to higher score
  - r(s) score as a function of study time

    $\frac{dr}{ds} > 0 \; \forall t$ ,  $r$: score, $s$: study time
  - One might get higher score by increasing the study time
- Negative correlation
  - Less party time leads to higher score
  - r(p) score as a function of party time

    $\frac{dr}{dp} < 0 \; \forall p$ ,  $r$: score, $p$: party time
  - One might get higher score by decreasing the party time.

# Computation: Chain Rule

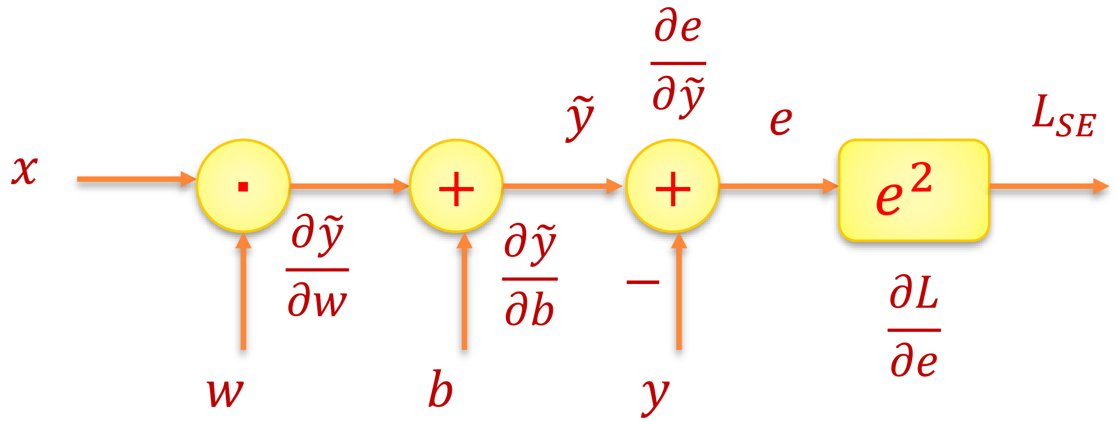
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x} = g'(y)f'(x)$$




- Propagate the gradient along a path

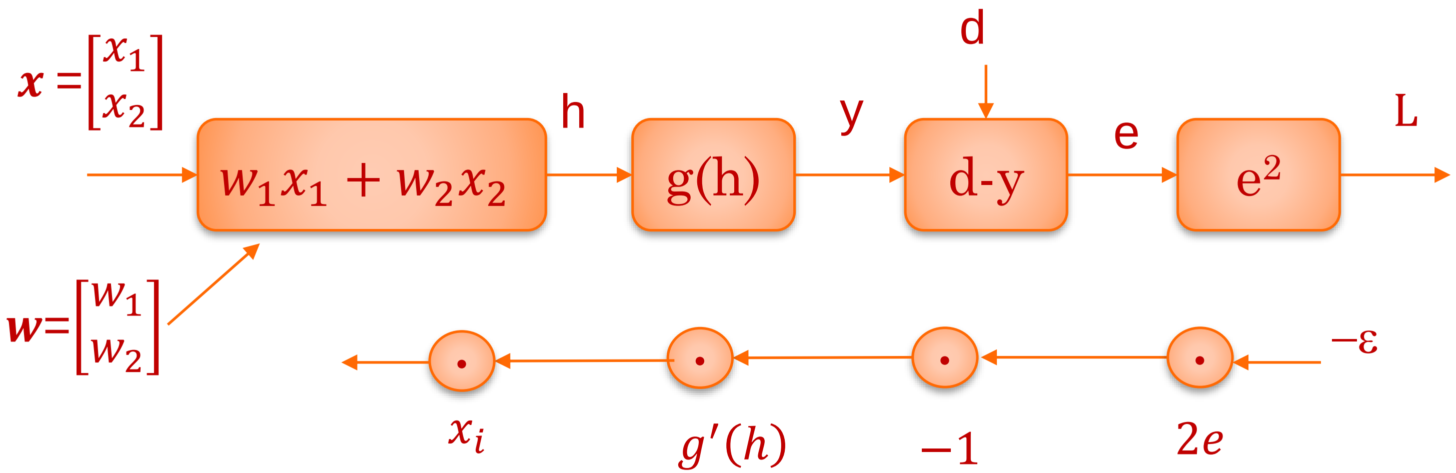
# Example: Linear Regression

- Solve it with gradient descent
- Linear regression: $\tilde{y} = w \cdot x + b$
- Loss: $L_{MSE} = E\left((y - \tilde{y})^2\right) \cong \sum_{i=1}^{n} (\tilde{y}_i - y_i)^2$



# Gradient Descent for a Neuron

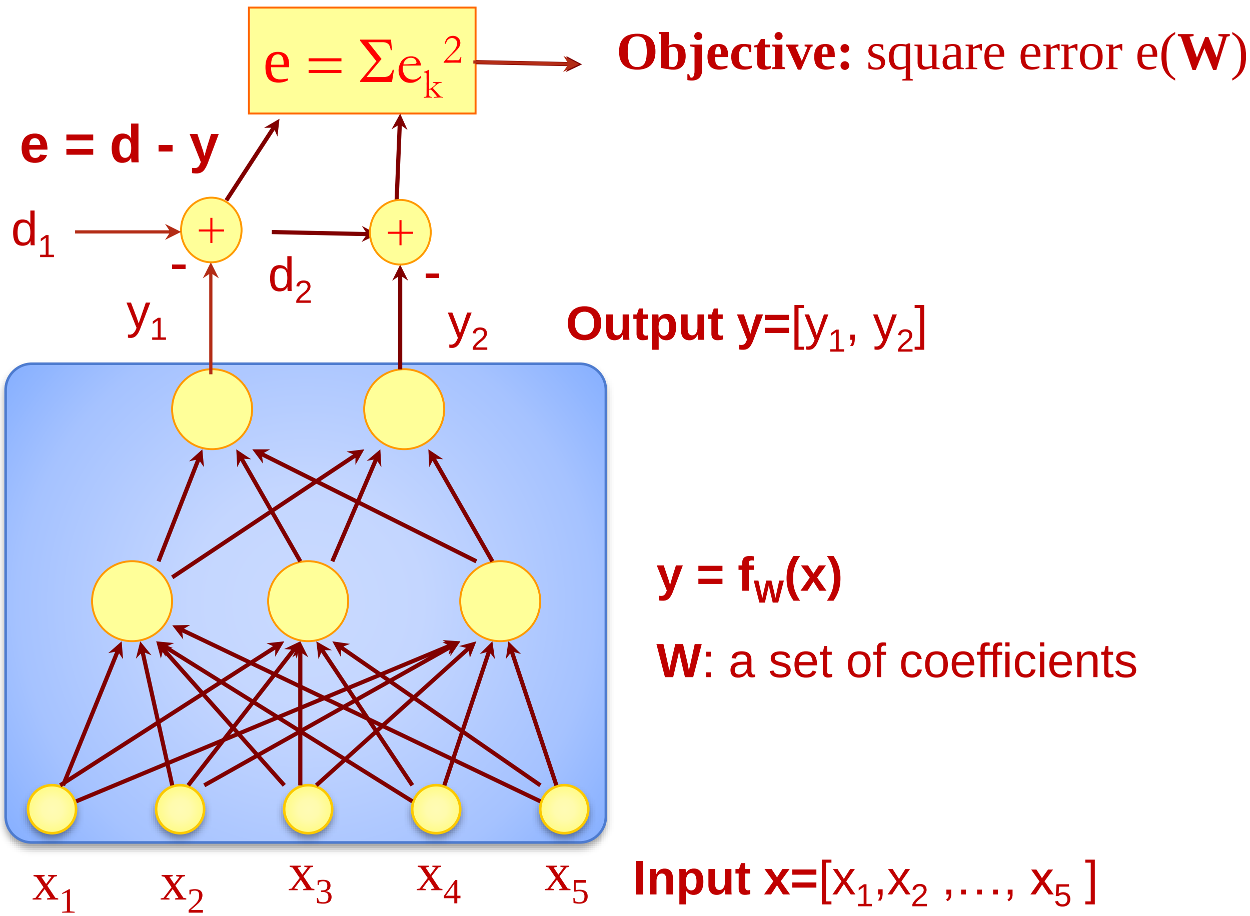
- $e(y) \equiv d - y$
- $y = g(h), h = w_1 x_1 + w_2 x_2$
- Objective: loss $\mathrm{L}_{SE} = e^2$




$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial y}\frac{\partial y}{\partial h}\frac{\partial h}{\partial w_i} = (2e)(-1)g'(h)x_i$$

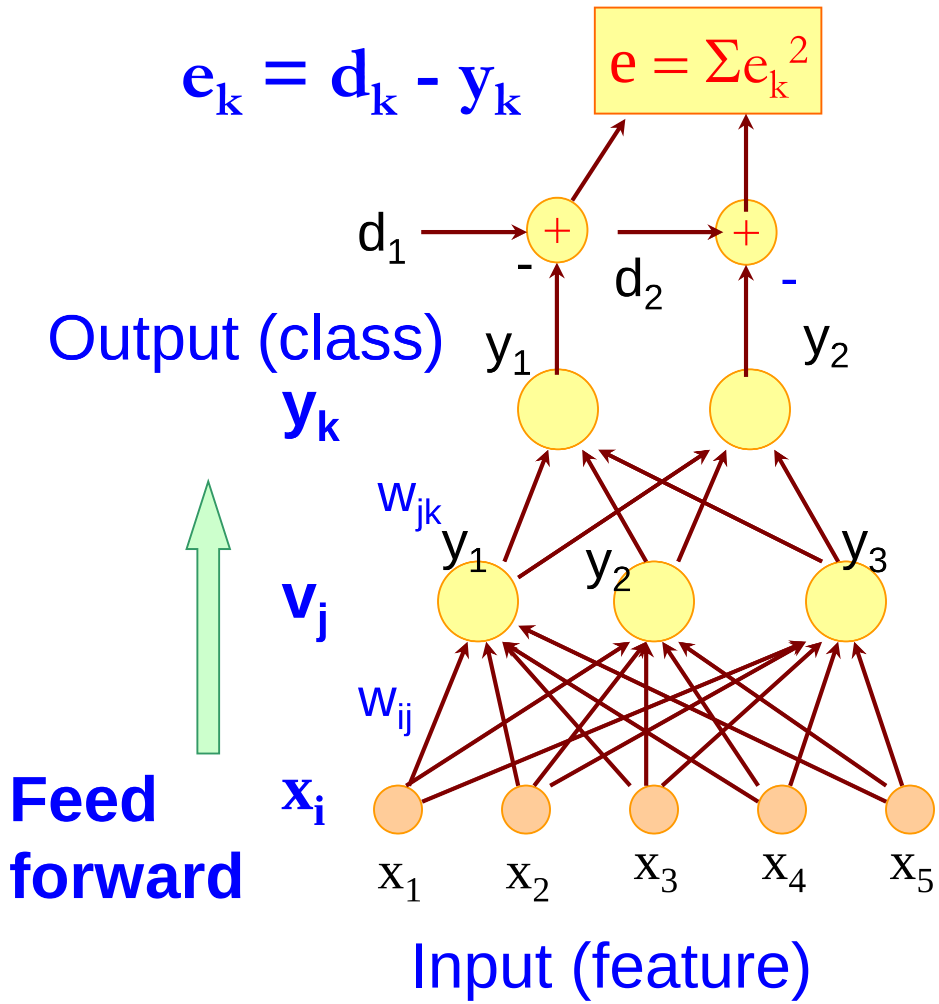
$$\mathrm{d}w_i = -\varepsilon\frac{\partial L}{\partial w_i} = 2 \cdot \varepsilon \cdot g'(h) \cdot e \cdot x_i$$

# Solving MLP



$e = \Sigma e_k^2$
**Objective:** square error e(**W**)
**e = d - y**
$d_1$
$d_2$
+
+
-
-
$y_1$
$y_2$
**Output y=**$[y_1, y_2]$
**y = f**$_W$**(x)**
**W**: a set of coefficients
$x_1$
$x_2$
$x_3$
$x_4$
$x_5$
**Input x=**$[x_1, x_2, \ldots, x_5]$

# Back Propagation




**Learn Backward**

$$\delta_k = g'(h_k) \cdot [d_k - y_k]$$
$$\Delta w_{jk} = -\eta \,(de/dw_{jk})$$
$$= \eta\, \delta_k\, y_j$$

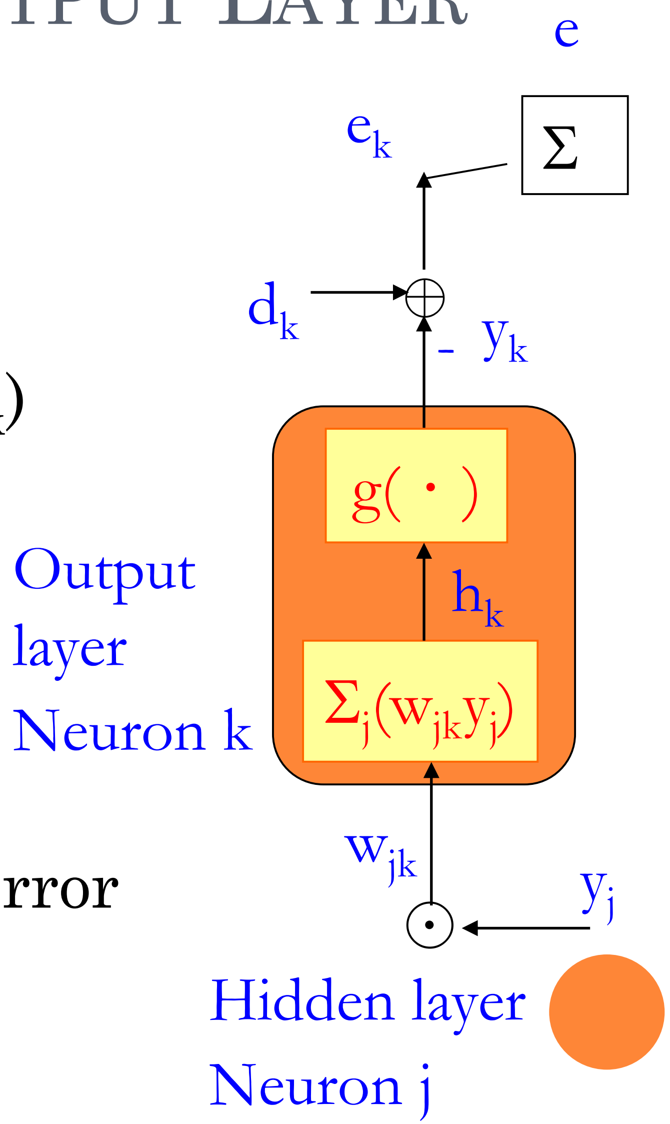
$$\delta_j = g'(h_j) \cdot \Sigma_k [w_{jk} \delta_k]$$
$$\Delta w_{ij} = \varepsilon\, \delta_j\, x_i$$

# Back Propagation

- i, j, k : neuron index for input layer (output $x_i$), hidden layer (output $y_j$), and output layer (output $y_k$) layers respectively.
- For node k in the output layer
  - the error $e_k$ is the difference between the desired output and the actual output ($e_k = d_k - y_k$).
  - The error gradient for k is: $\delta_k = y_k \cdot (1-y_k) \cdot e_k$
  - The weights are updated by: $w'_{jk} = w_{jk} + \alpha \cdot y_j \cdot \delta_k$
  - $\alpha$: learning rate (a positive number below 1)
- Similarly, for a node j in the hidden layer
  - The error gradient for k is: $\delta_j = y_j \cdot (1-y_j) \cdot \Sigma_k w_{jk} \delta_k$
  - The weights are updated by: $w'_{ij} = w_{ij} + \alpha \cdot x_i \cdot \delta_j$

# Learning Weights of Output Layer

- $e = \Sigma_k e_k^2$, $e_k = d_k - y_k$
  $y_k = g(h_k)$, $h_k = \Sigma_j\, w_{jk} y_j$
  by chain rule $de/dw_{jk}$ =
  $(de/de_k)(de_k/dy_k)(dy_k/dh_k)(dh_k/dw_{jk})$
  $= 2e_k\ (-1)\ \cdot g'(h_k)\ \cdot y_j$
  $\Delta w_{jk} = -\alpha\ (de/dw_{jk})$
  $= -\alpha\ \cdot 2(d_k - y_k)\ (-1)\ \cdot g'(h_k)\ \cdot y_j$
  $= \eta\ \cdot g'(h_k)\ \cdot (d_k - y_k)\ \cdot y_j$
  $= \eta\ \cdot \delta_k\ \cdot y_j$
  $\delta_k \equiv g'(h_k)\ \cdot (d_k - y_k)$ feedback of error

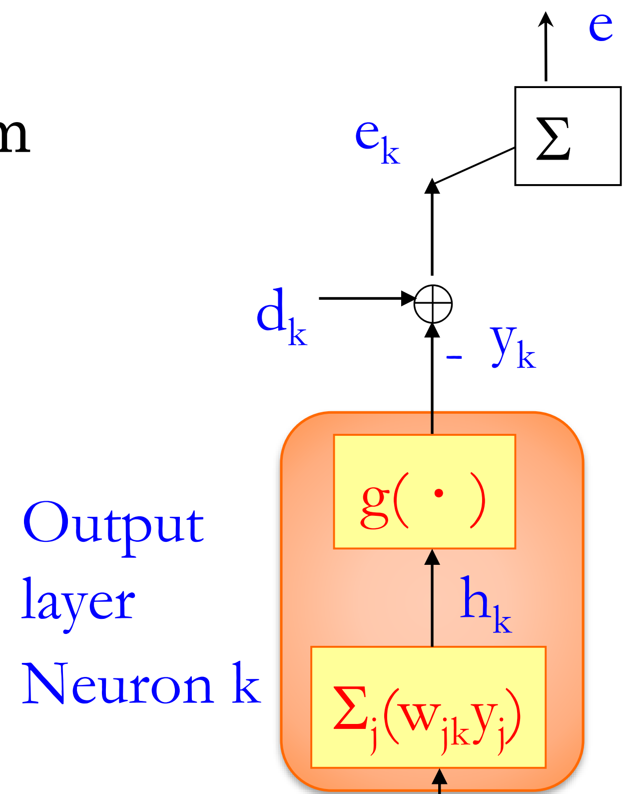


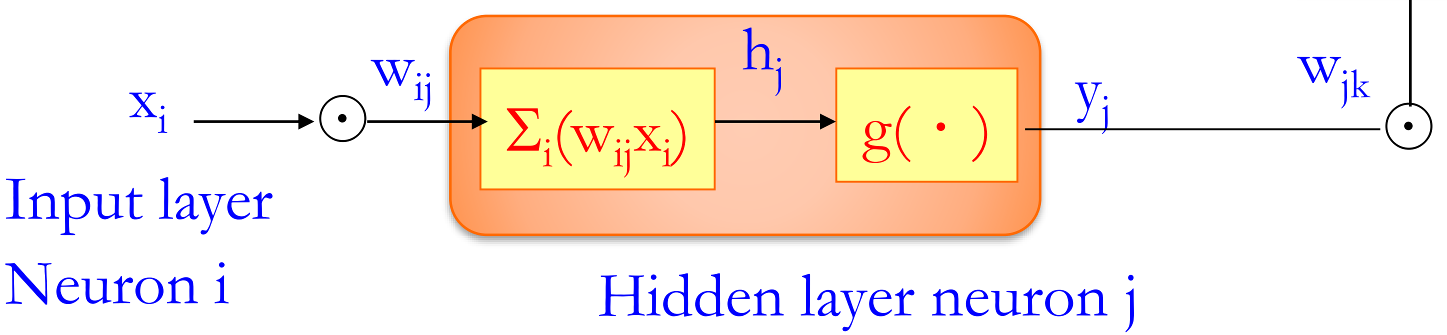
# Learning Weights of Hidden Layer

- All $e_k$'s are influenced by $w_{ij}$ from multiple paths, therefore,
- $\frac{\partial e}{\partial w_{ij}} = \left( \sum_k \frac{\partial e}{\partial e_k} \frac{\partial e_k}{\partial y_k} \frac{\partial y_k}{\partial h_k} \frac{\partial h_k}{\partial y_j} \right) \frac{\partial y_j}{\partial h_j} \frac{\partial h_j}{\partial w_{ij}}$

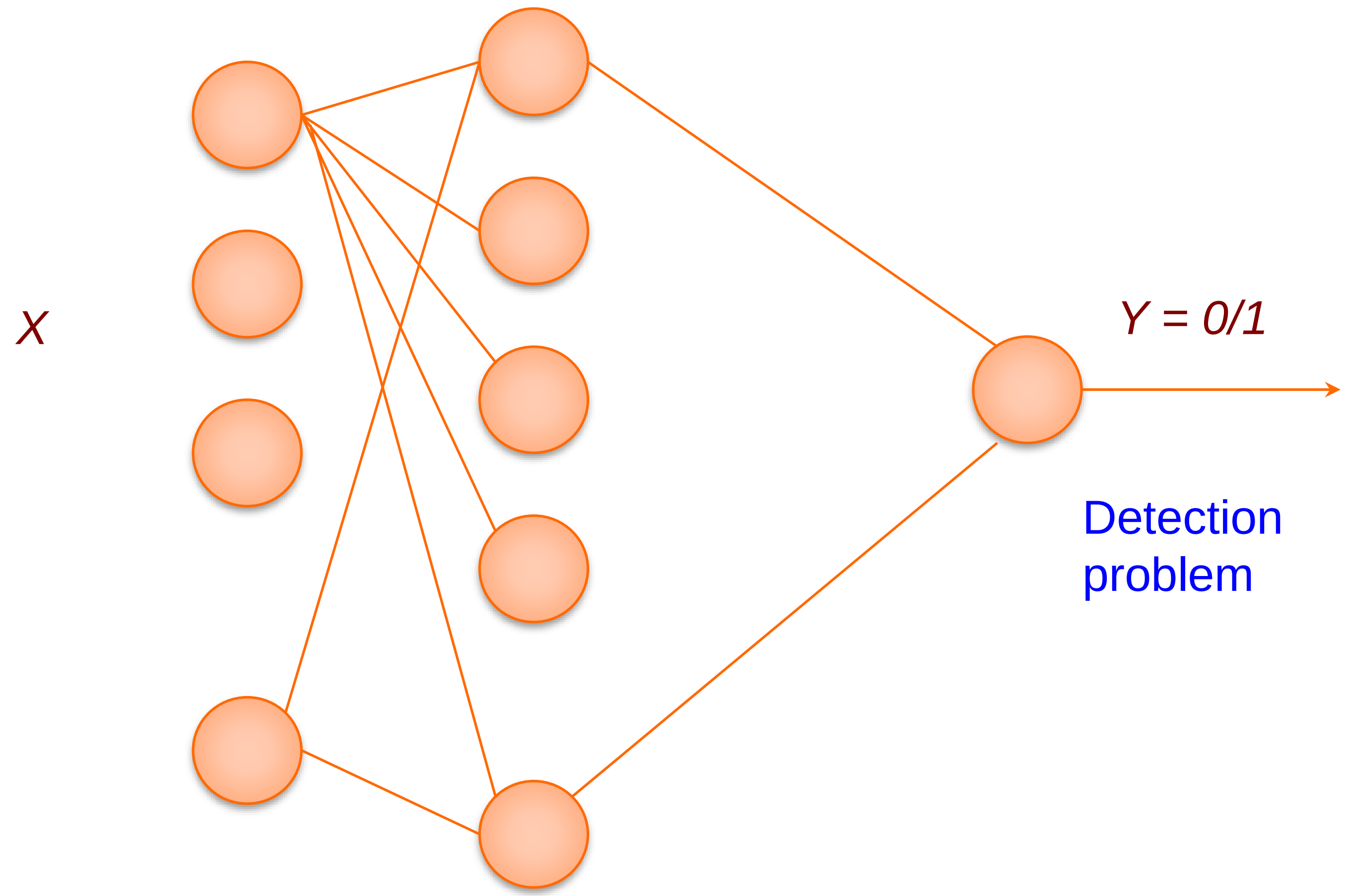
  $= \sum_k -2\delta_k w_{jk} g'(h_j) x_i$
- $dw_{ij} = -\alpha \frac{\partial e}{\partial w_{ij}} = \eta \cdot \delta_j . x_i$

  $\delta_j = g'(h_j) \sum_k \delta_k w_{jk}$

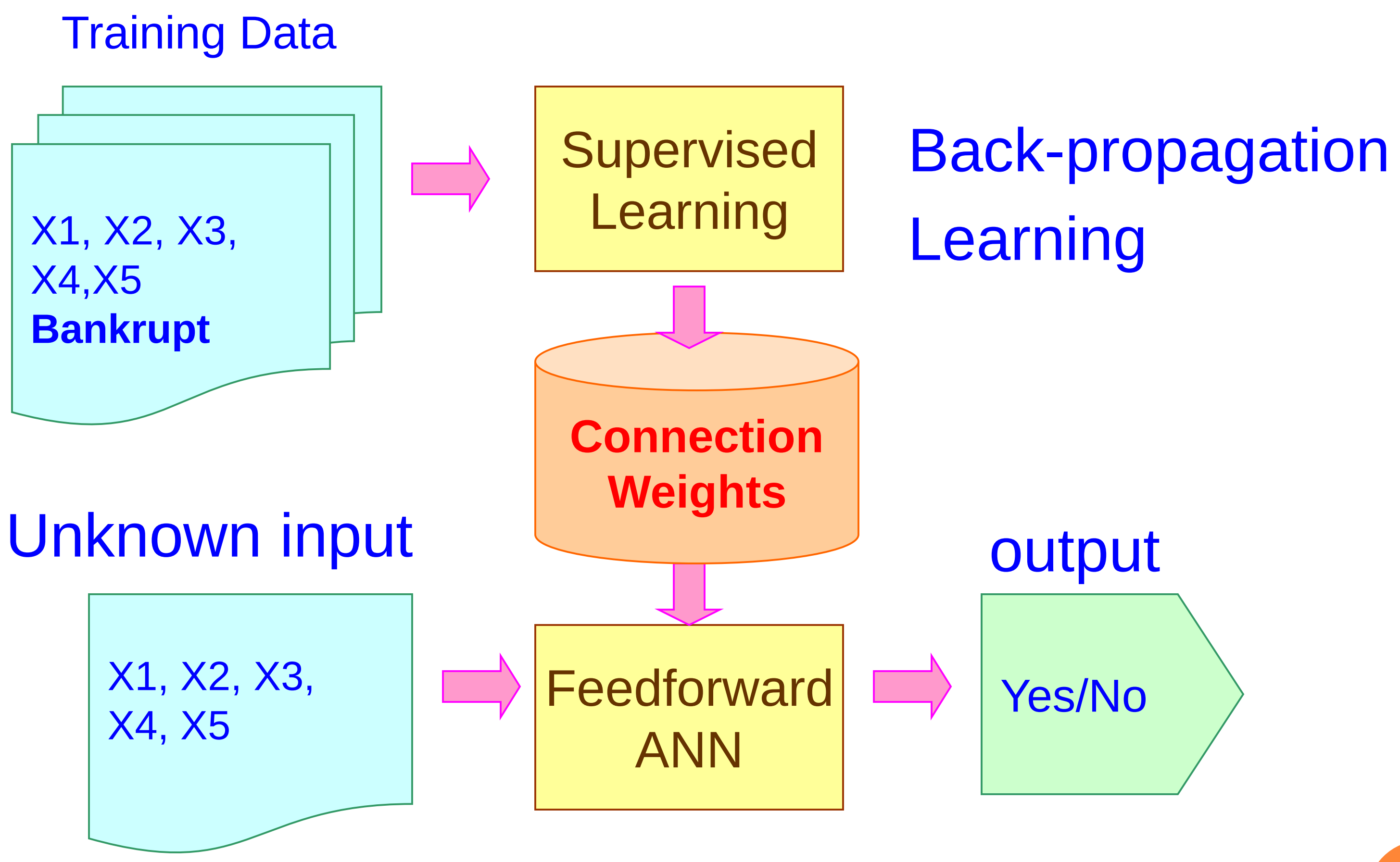


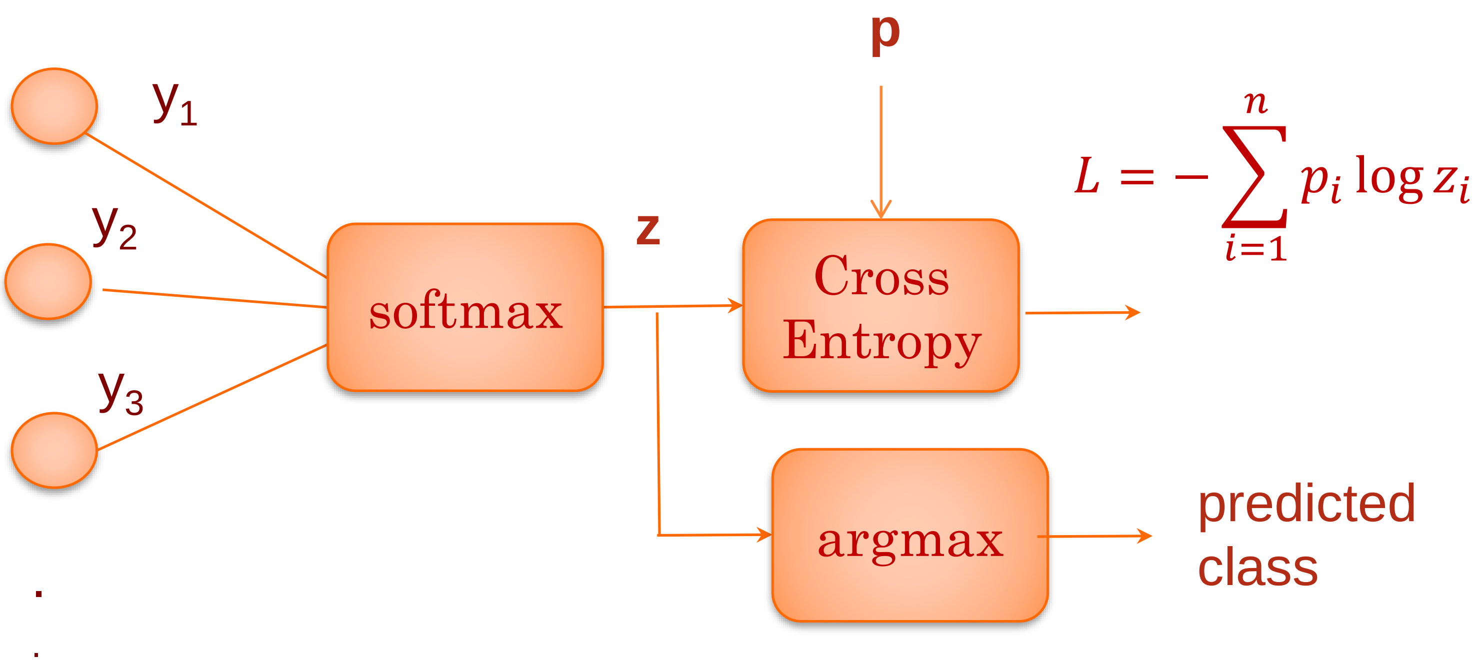
# Binary Classifier



# MLP-based Classifier



# Softmax for M-ary Classifier



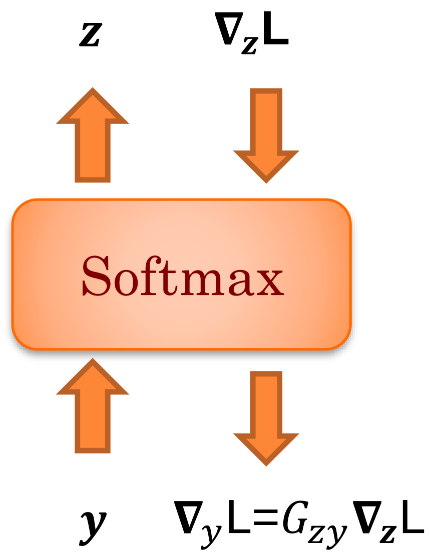

- $z_i = \frac{e^{y_i}}{Z}$, $Z = \sum_{k=1}^{n} e^{y_k}$ where $\sum z_i = 1$
- $p_i' s$ are the ground truths
- Cross entropy $L = -\sum_{i=1}^{n} p_i \log z_i$

# Gradient Propagation for Softmax

- $z_i = \frac{e^{y_i}}{Z} = e^{y_i} Z^{-1}, Z = \sum_{k=1}^{n} e^{y_k}$
- $\frac{\partial z_i}{\partial y_i} = e^{y_i} Z^{-1} + e^{y_i}(-1) Z^{-2} \frac{\partial Z}{\partial y_i}$

  $= z_i - e^{y_i} Z^{-2} e^{y_i} = z_i - {z_i}^2 = z_i(1 - z_i)$
- $\frac{\partial z_i}{\partial y_j} = e^{y_i}(-1) Z^{-2} \frac{\partial Z}{\partial y_j}$

  $= -e^{y_i} Z^{-2} e^{y_j} = -z_i z_j \; for \; j \neq i$
- $$G_{zy} = \begin{bmatrix} z_1(1 - z_1) & -z_1 z_2 & \cdots & -z_1 z_n \\ -z_1 z_2 & z_2(1 - z_2) & \cdots & -z_2 z_n \\ \vdots & \vdots & \ddots & \vdots \\ -z_1 z_n & -z_2 z_n & \cdots & z_n(1 - z_n) \end{bmatrix}$$



# Minimizing Cross Entropy

- Objective: $\mathrm{L} = -\sum_{i=1}^{n} p_i \log z_i$
- Let $\{ z_i \}$ to approximate $\{ p_i \}$ where.$\sum p_i = 1$
- Since $\frac{\partial \mathrm{L}}{\partial z_i} = -\frac{p_i}{z_i}$,

$$\nabla_z \mathrm{L} = \begin{bmatrix} \frac{\partial \mathrm{L}}{\partial z_1} \\ \vdots \\ \frac{\partial \mathrm{L}}{\partial z_n} \end{bmatrix} = -\begin{bmatrix} \frac{p_1}{z_1} \\ \vdots \\ \frac{p_n}{z_n} \end{bmatrix}.$$

$$\nabla_y \mathrm{L} = G_{zx} \nabla_{\boldsymbol{z}} \mathrm{L} = -\begin{bmatrix} z_1(1-z_1) & -z_1 z_2 & \cdots & -z_1 z_n \\ -z_1 z_2 & z_2(1-z_2) & \cdots & -z_2 z_n \\ \vdots & \vdots & \ddots & \vdots \\ -z_1 z_n & -z_2 z_n & \cdots & z_n(1-z_n) \end{bmatrix} \begin{bmatrix} \frac{p_1}{z_1} \\ \vdots \\ \frac{p_n}{z_n} \end{bmatrix}$$

# Gradient Descent on Softmax

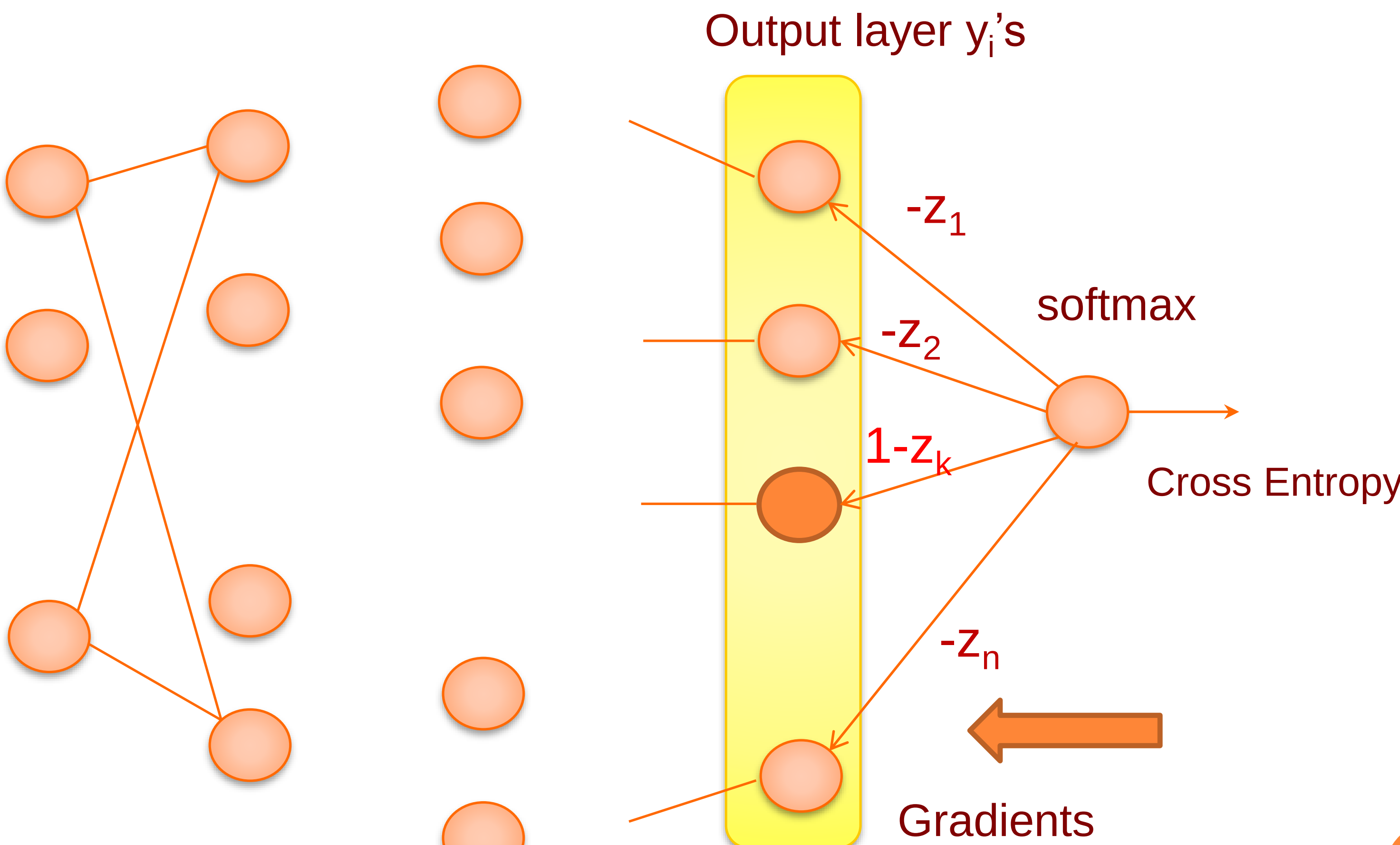
- $i$-th element of $\boldsymbol{g}_y$ is

$$\nabla_{y_i} = -(z_i(1-z_i)\frac{p_i}{z_i} - \sum_{j\neq i} z_i z_j \frac{p_j}{z_j})$$
$$= -(p_i(1-z_i) - \sum_{j\neq i} z_i p_j)$$
$$= -p_i + p_i z_i + z_i \sum_{j\neq i} p_j \quad (\sum p_j = 1)$$
$$= -p_i + p_i z_i + z_i(1-p_i) = z_i - p_i.$$

To minimize $J$, $-\nabla_{y_i} = (p_i - z_i)$ is propagated to $y_i$.

- Notice that classification is a special case
  - $p_k = 1$ and $p_j = 0$ for all $j \neq k$ for gold branch $k$ (one-hot)
  - i.e. $1 - z_k$ for target node and $-z_j$ for the others.
- $\boldsymbol{z}$ is an estimate of the target distribution, $\boldsymbol{p}$.

# Softmax: Propagating Gradients




Gradients
Gold branch: 1-z]k]
others: -z[j]

# Improving Speed of Convergence

- Use generalized delta rule for learning
  - $\Delta w_{ij}(t) = \alpha \cdot x_i \cdot \delta_j + \beta \cdot \Delta w_{ij}(t-1)$
  - Momentum $\beta$ has a typical value of 0.95
  - Weighted sum between training data and original value (learn incrementally)
- Use hyperbolic tangent function instead of sigmoid
  - $\tanh(x) = 2a/(1+e^{-bx}) - a$ ($a = 1.7$, $b = 0.7$)
- Use dynamic learning rate $\alpha$
  - Increasing $\alpha$ if square-error changes in the same direction for several epochs, and decreasing $\alpha$ otherwise.

# Adaptive Learning Rate

- AdaGrad
  - $\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,i}+\epsilon}} g_{t,i}$
  - $g_{t,i} = \nabla_\theta J(\theta_{t,i})$
  - $\epsilon$ is smoothing value (typically 1e-8)
  - $G_{t,i}$ is energy of gradients: summation of the squares of past gradients (${g_{t,i}}^2$)
- AdaDelta
  - $E\left[g^2\right]_t = \gamma E\left[g^2\right]_{t-1} + (1-\gamma){g_t}^2$
  - $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t+\epsilon}} g_t$
  - Incremental estimation of average energy $E\left[g^2\right]_t$
  - Momentum $\gamma$=0.99 typically

# Adam (Adaptive Moment Estimation)

- $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$
- $v_t = \beta_2 v_{t-1} + (1-\beta_2) {g_t}^2$
- $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} m_t$
  - It is required to normalize $m_t$ and $v_t$ by

    $\hat{m}_t = \frac{m_t}{1-{\beta_1}^t}$, $\hat{v}_t = \frac{v_t}{1-{\beta_2}^t}$ if $m_0 = v_0 = 0$
  - But unnecessary if $m_0 = g_0$, $v_0 = {g_0}^2$
- Exponential decay ($\beta < 1$) of influence
  - $m_1 = \beta_1 g_0 + (1-\beta_1) g_1$
  - $m_2 = \beta_1(\beta_1 g_0 + (1-\beta_1) g_1) + (1-\beta_1) g_2$

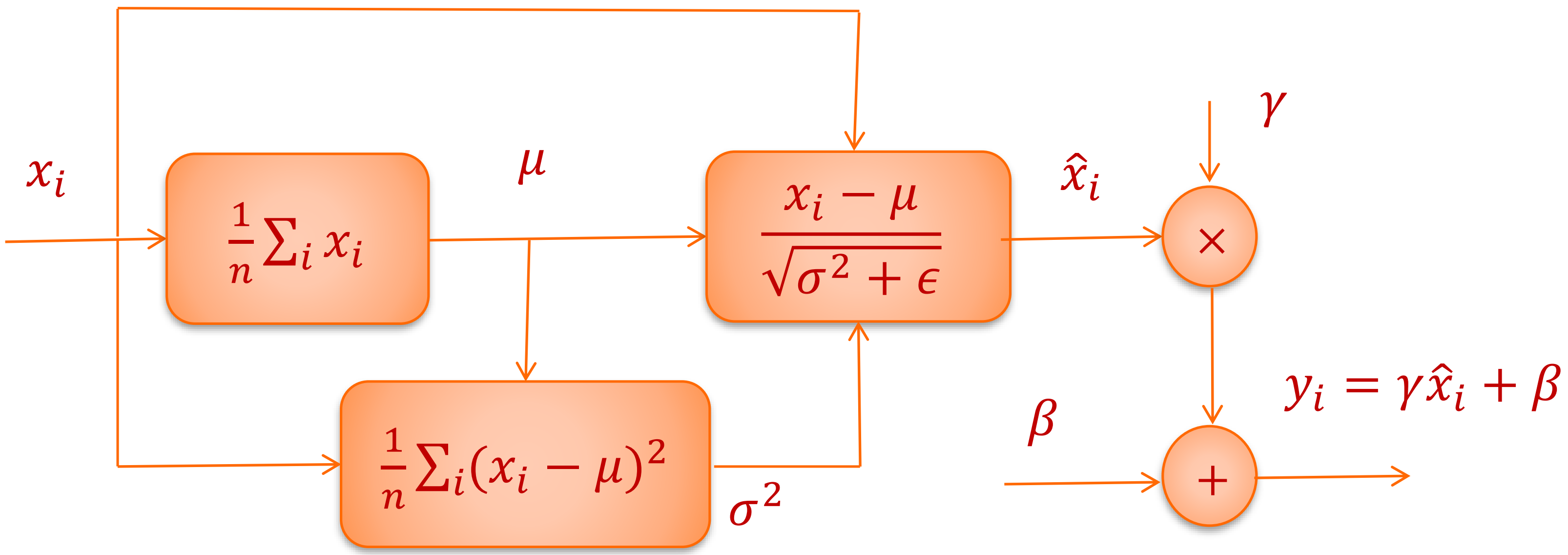
    $={\beta_1}^2 g_0 + \beta_1 (1-\beta_1) g_1 + (1-\beta_1) g_2$
  - $m_t = {\beta_1}^t g_0 + {\beta_1}^{t-1}(1-\beta_1) g_1 + \cdots + (1-\beta_1) g_t = \sum_{i=0}^{t} w_i g_i$
  - $\sum_{i=0}^{t} w_i = 1 \rightarrow m_t$ as weighting sum of gradients

# Batch Normalization

- Adjust the distribution (means and variances) of neuron inputs to avoid gradient vanishing (especially for deep network)
- May accelerate the convergence when training the network
- Reference:

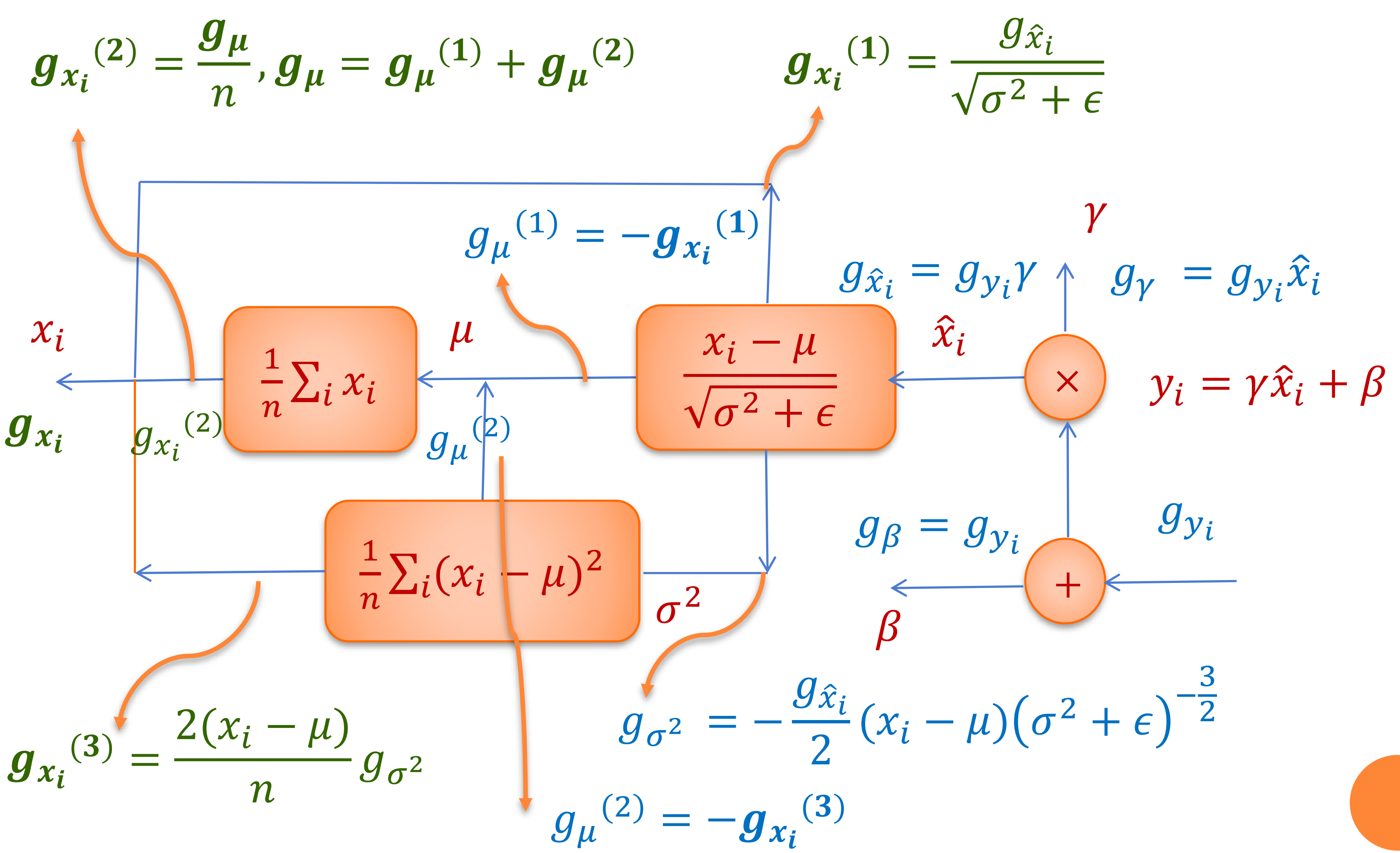
"Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", Sergey Ioffe and Christian Szegedy, 2015.

# Batch Normalization




- $\hat{x}_i$ has zero mean and unit variance.
- $y_i$ with shifted mean and scaled variance.
- $\gamma$ *and* $\beta$ are learnable.
- When $\beta$ equals to mean and $\gamma$ equals to standard deviation, BN is identity transform.

# Batch Normalization: Gradient



# Normalization

- $z_i = \frac{y_i}{Y} = y_i Y^{-1}, Y = |\boldsymbol{y}| = S^{1/2}$

  $\mathrm{S} = \sum_{k=1}^{n} {y_k}^2$
- $\frac{\partial z_i}{\partial y_i} = Y^{-1} + y_i(-1)Y^{-2}\frac{\partial Y}{\partial y_i}$

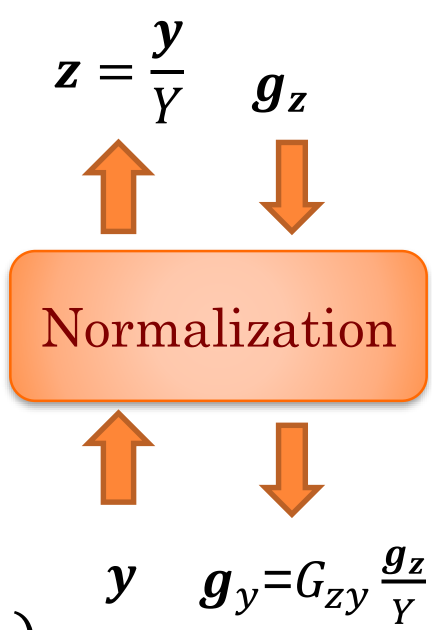
  $= Y^{-1} - y_i Y^{-2}(\frac{1}{2})S^{-\frac{1}{2}}(2y_i)$

  $= Y^{-1}\left(1 - y_i Y^{-2} y_i\right) = Y^{-1}(1 - {z_i}^2)$
- $\frac{\partial z_i}{\partial y_j} = y_i(-1)Y^{-2}\frac{\partial Y}{\partial y_j} = -y_i Y^{-2}\ (\frac{1}{2})S^{-\frac{1}{2}}(2y_j)$

  $= -y_i Y^{-2} Y^{-1} y_i {=} {-} Y^{-1} z_i z_j$
- Define $G_{zy}$ such that

  $\mathrm{G(i,i)} = 1 - {z_i}^2$

  $\mathrm{G(i,\,j)} = -z_i z_j$



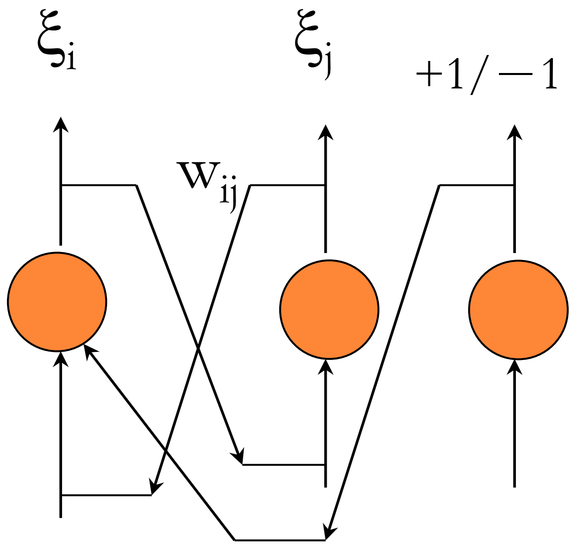
# Recurrent Networks

- Feed forward networks do not have memory.
- Recurrent networks can have connections between nodes in any layer, which enables them to store data – a memory.
  - flip-flop
- Capability of sequence prediction
  - Recurrent networks can be used to solve problems where the solution depends on previous inputs as well as current inputs (e.g. predicting stock market movements).

# Hopfield Network

- $\underline{\xi} = (\xi_1, \xi_2, \ldots)$ to be memorized with values of +1/-1
- $\xi_i = \text{sign}(\Sigma_j w_{ij} \xi_j - \theta_j)$
  $w_{ij} = (1/N)\ \xi_i\ \xi_j$
- $h_i = \Sigma_j w_{ij} \xi_j = (1/N) \Sigma_j \xi_i \xi_j \xi_j$
  $= \xi_i$ (since $\xi_j^2 = 1$)
- $h_i = \Sigma_j w_{ij} S_j = (1/N) \Sigma_j \xi_i \xi_j S_j$
  $= (1/N)\ \xi_i (\Sigma_j \xi_j S_j)$
  $= (1/N)\ \xi_i \cdot \langle \underline{S}, \underline{\xi} \rangle$

  If $\underline{S}$ is close to $\underline{\xi}$ (in Hamming distance) such that $\langle \underline{S}, \underline{\xi} \rangle > 0$, then $S_i$ will be attracted into $\xi_i$ and ($\underline{S} = \underline{\xi}$)




$\underline{S} = (S_1, S_2, \ldots, S_n)$

# Hopfield Network (cont'd)

$$W = \sum_{i=1}^{N} X_i X_i^t - N\,I$$

$$Y = sign(WX - \theta)$$

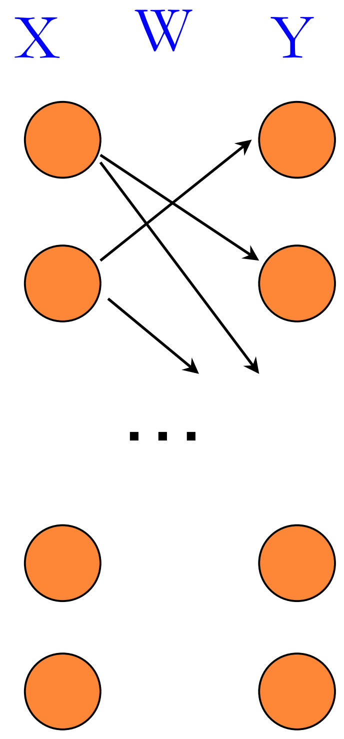
$X_i$'s are patterns to be memorized

Weights:N patterns ($X_i$'s)

$Y_i = \text{Sign}(WX_i - \theta) = X_i$

Input X close to $X_i$ will be attracted to $X_i$

Hopfield network is a memory that usually maps an input vector to the memorized vector whose **Hamming distance** from the input vector is least. (auto-associative memory) (memory→recall)



# Example of Hopfield Network

$$X_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad X_2 = \begin{bmatrix} -1 \\ -1 \\ -1 \\ -1 \\ -1 \end{bmatrix} \quad X_3 = \begin{bmatrix} 1 \\ -1 \\ 1 \\ 1 \\ -1 \end{bmatrix}$$

$$Y_2 = WX_2 = X_2 \qquad Y_3 = WX_3 = X_3$$

$$X_4 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \\ 1 \end{bmatrix}$$

$$W = \sum_{i=1}^{3} X_i X_i^t - 3I = \begin{bmatrix} 0 & 1 & 3 & 3 & 1 \\ 1 & 0 & 1 & 1 & 3 \\ 3 & 1 & 0 & 3 & 1 \\ 3 & 1 & 3 & 0 & 1 \\ 1 & 3 & 1 & 1 & 0 \end{bmatrix}$$

$$Y_4 = sign(WX_4) = sign(\begin{bmatrix} 2 \\ 4 \\ 8 \\ 2 \\ 4 \end{bmatrix}) = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = X_1$$

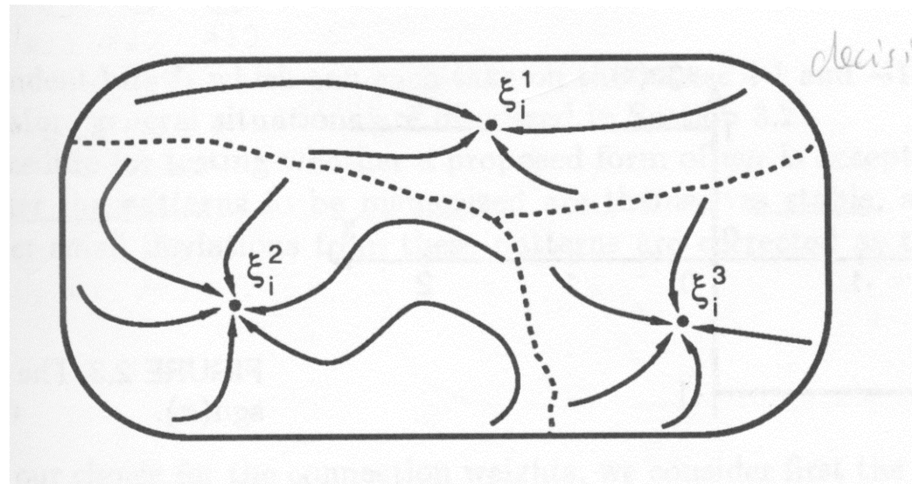
$$Y_1 = sign(WX_1) = sign(\begin{bmatrix} 8 \\ 6 \\ 8 \\ 8 \\ 6 \end{bmatrix}) = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = X_1$$

# Example of Hopfield Network (cont'd)

$$X_5 = \begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \\ 1 \end{bmatrix}$$

$$Y_5 = sign(WX_5) = sign(\begin{bmatrix} 2 \\ 2 \\ 2 \\ -4 \\ 2 \end{bmatrix}) = \begin{bmatrix} 1 \\ 1 \\ 1 \\ -1 \\ 1 \end{bmatrix}$$

$$Y_5' = WY_5 = sign(\begin{bmatrix} 2 \\ 4 \\ 2 \\ 8 \\ 4 \end{bmatrix}) = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = X_1$$




Auto-associative Memory

Storage: setting the weights

Retrieval: retrieving data

# Application of Hopfield Network

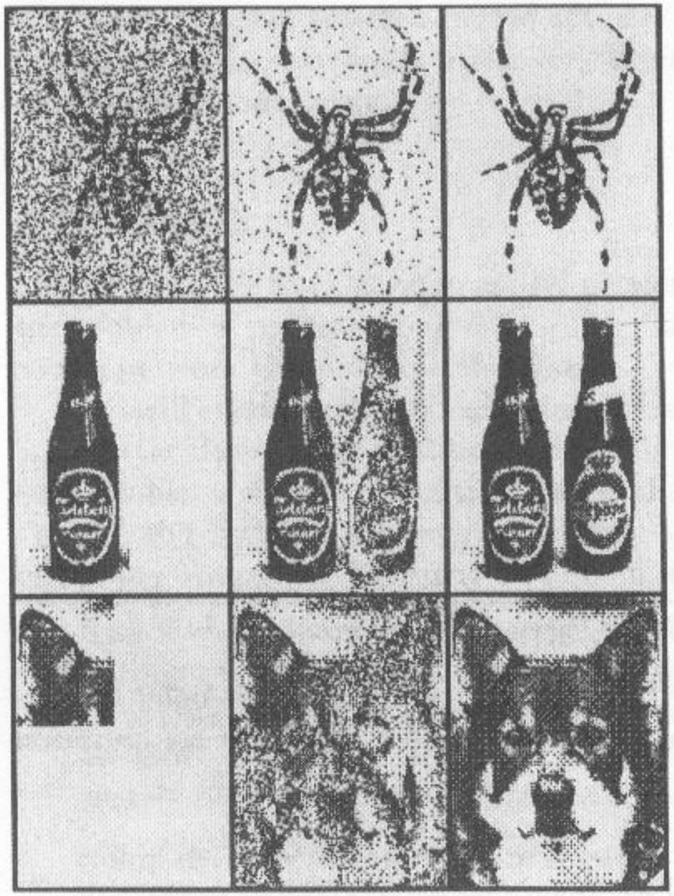


FIGURE 2.1 Example of how an associative memory can reconstruct images. These are binary images with 130 × 180 pixels. The images on the right were recalled by the memory after presentation of the corrupted images shown on the left. The middle column shows some intermediate states. A sparsely connected Hopfield network with seven stored images was used.

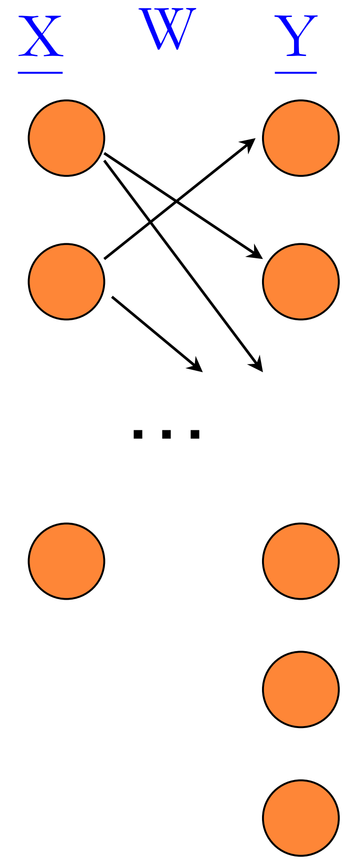
# Bidirectional Associative Memories (BAM)

$$W = \sum_{i-1}^{n} X_i Y_i^t$$

$$sign(W^t X_j) = Y_j$$

$$sign(WY_j) = X_j$$

- association X → Y



# Example of BAM

$$X_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad X_2 = \begin{bmatrix} -1 \\ -1 \end{bmatrix} Y_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \quad Y_2 = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}$$

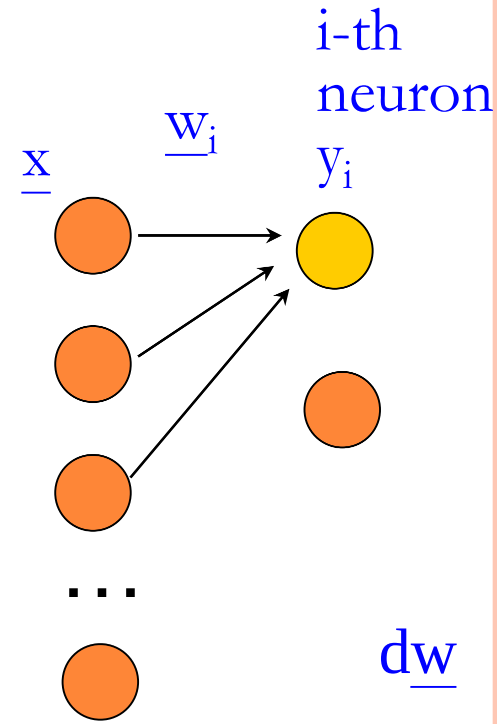
$$W = \sum_{i=1}^{2} X_i Y_i^t = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}$$

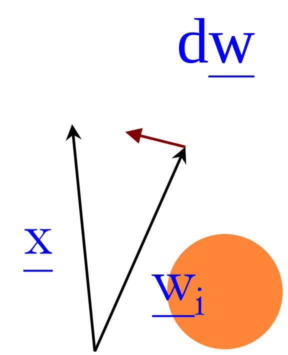
$$sign(W^t X_1) = sign(\begin{bmatrix} 4 \\ 4 \\ 4 \end{bmatrix}) = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = Y_1$$

$$sign(WY_1) = sign(\begin{bmatrix} 6 \\ 6 \end{bmatrix}) = \begin{bmatrix} 1 \\ 1 \end{bmatrix} = X_1$$

# Kohonen Maps (Self-Organizing Map)

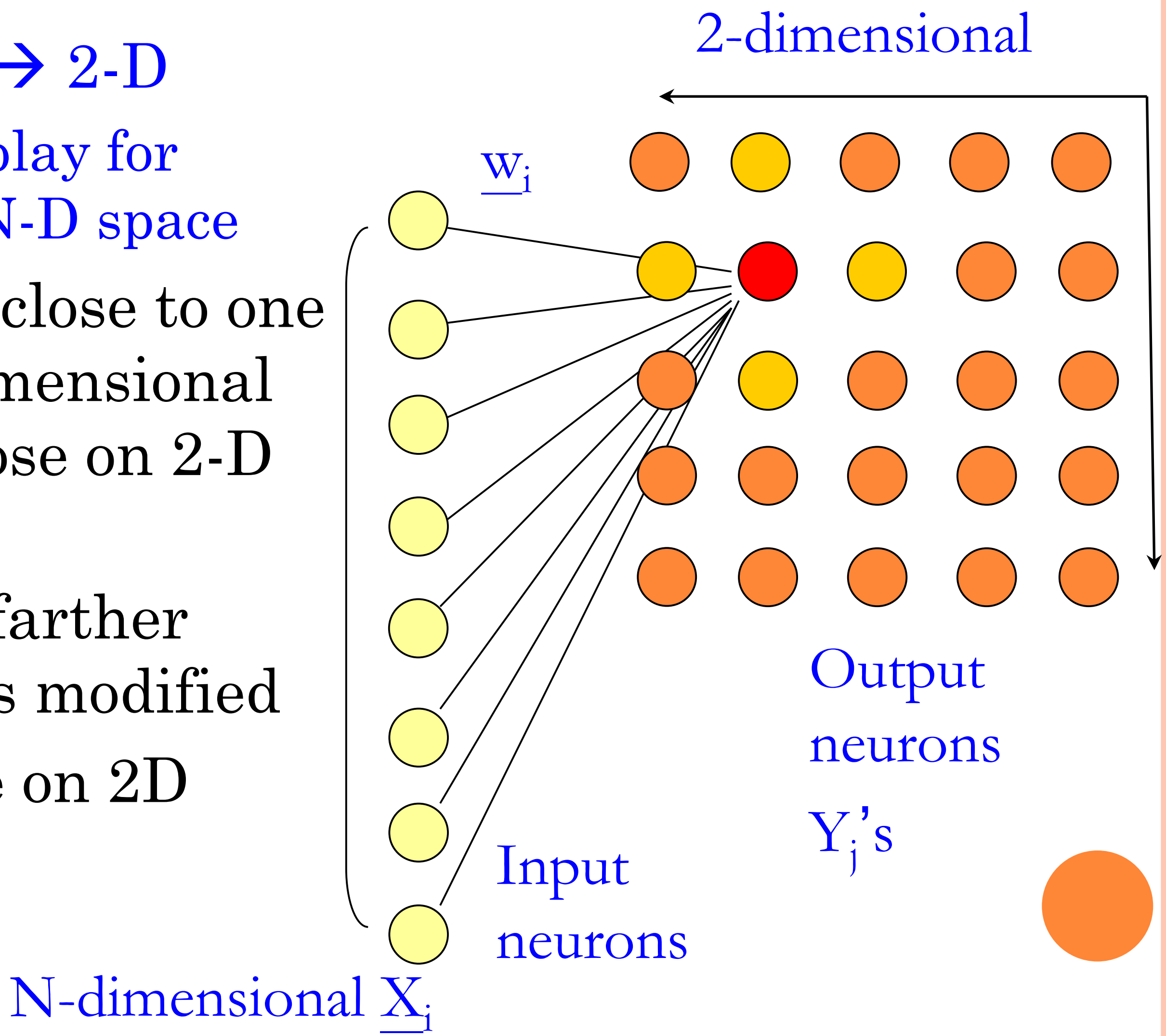
- Unsupervised learning
  - Clustering for training data $\{\underline{x}_j\}$
- Winner-take-all
  - The winner is the neuron with highest output ($\underline{w}_i$ closest to $\underline{x}$)
  - The winner and its spatial neighborhoods can update weights
  - $w_{ij}' = w_{ij} + \alpha(x_j - w_{ij})$ or $d\underline{w}_i = \alpha(\underline{x} - \underline{w}_i)$
- $y_i = \text{sign}(<\underline{x}, \underline{w}_i>)$
  - $<\underline{x}, \underline{w}_i>$ similarity
- 和KNN相似, K個output neurons的weights代表K個centroids



# Kohonen Maps (cont'd)

- N-dimensional → 2-D
  - 2D Visual display for closeness for N-D space
- Codewords ($\underline{w}_i$) close to one another in n-dimensional space will be close on 2-D plane
- The weights of farther neurons are less modified
- View N-D space on 2D plane



# References


- *Introduction to the Theory of Neural Computing*
  John Hertz, Anders Krogh, Richard G. Palmer
- *Artificial Intelligence A modern Approach*
  Stuart Russel, Peter Norvig
- *Artificial Intelligence, illuminated*
  Ben Coppin
- *An overview of gradient descent optimization algorithms*, Sebastian Ruder, 2016.