Machine Learning

CSE 6363 (Fall 2016)

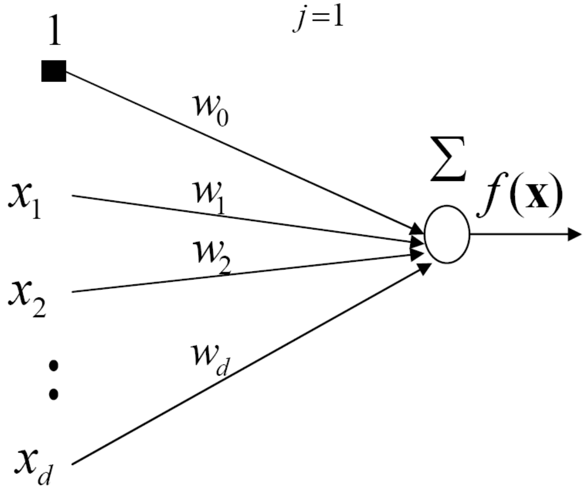
Lecture 11 Neural Networks

Heng Huang, Ph.D.
Department of Computer Science and Engineering

Linear Units

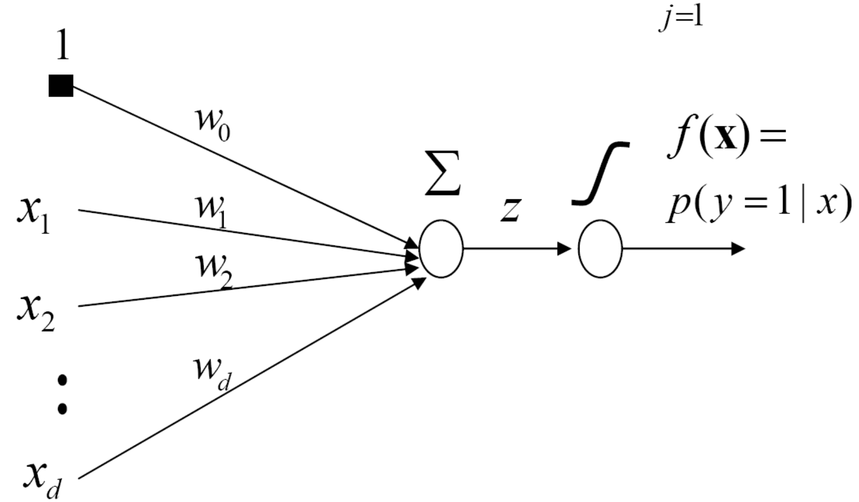
Linear regression

$$f(\mathbf{x}) = w_0 + \sum_{j=1}^{d} w_j x_j$$



Logistic regression

$$f(\mathbf{x}) = p(y=1 \mid \mathbf{x}, \mathbf{w}) = g(w_0 + \sum_{j=1}^{d} w_j x_j)$$



On-line gradient update:

$$w_0 \leftarrow w_0 + \alpha(y - f(\mathbf{x}))$$

$$\vdots$$

$$w_j \leftarrow w_j + \alpha(y - f(\mathbf{x}))x_j$$

The same

On-line gradient update:

$$w_0 \leftarrow w_0 + \alpha(y - f(\mathbf{x}))$$

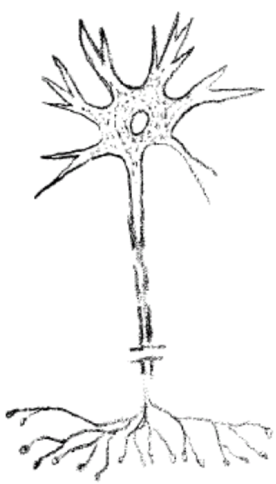
$$\vdots$$

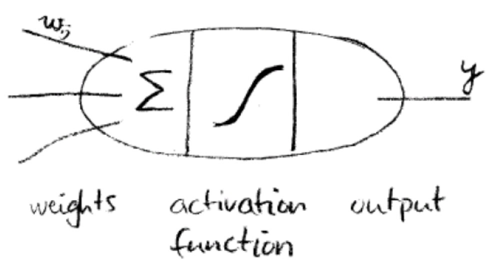
$$w_j \leftarrow w_j + \alpha(y - f(\mathbf{x}))x_j$$

Biological Neural Networks

Biological Neural Networks — *common features* — Artificial Neural Networks

Biological Neural Networks	*common features*	Articficial Neural Networks
models used in	parallel computations	models used in
Theoretical Neuroscience	distributed	Function Approximation
	non linear units	Classification
	local processing	Data Processing
	adaptation	





Extensions of Simple Linear Units

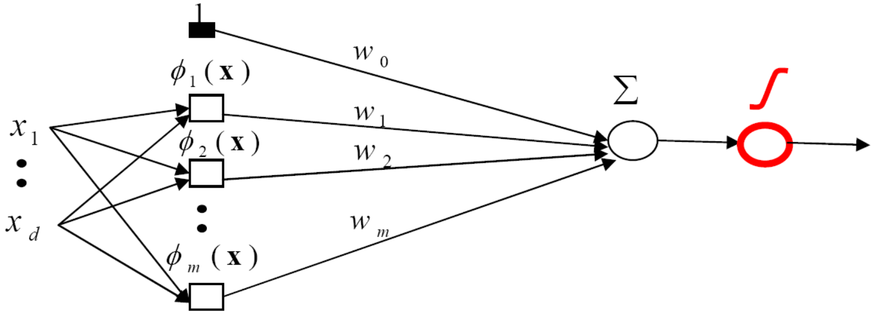
Feature (basis) functions model **nonlinearities**

Linear regression

$$f(\mathbf{x}) = w_0 + \sum_{j=1}^{m} w_j \phi_j(\mathbf{x})$$

Logistic regression

$$f(\mathbf{x}) = g(w_0 + \sum_{j=1}^{m} w_j \phi_j(\mathbf{x}))$$



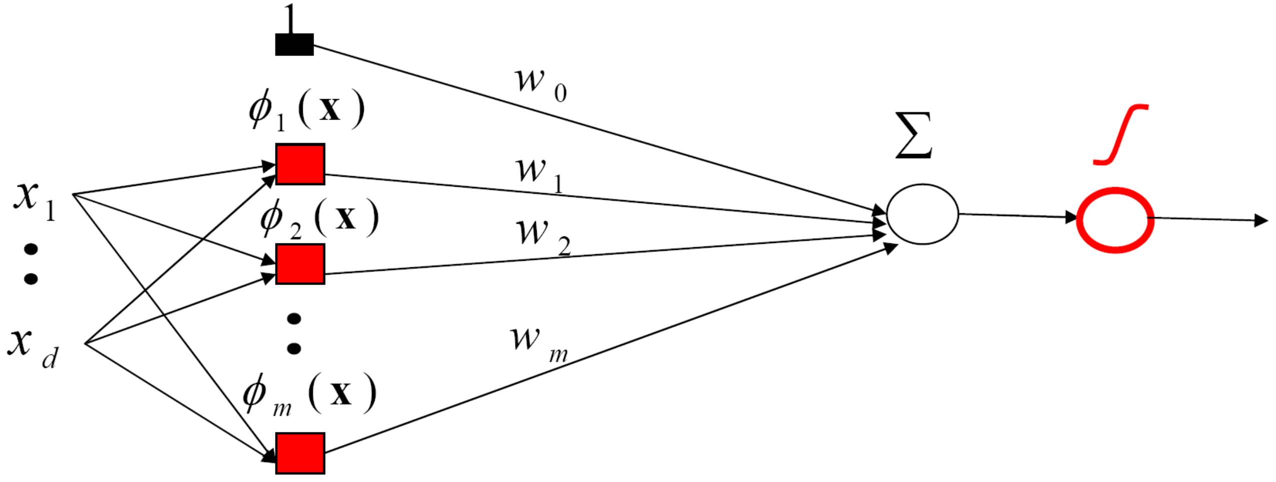
Important property:

- The same problem as learning of the weights for linear units , the input has changed– but the weights are linear in the new input

Problem: too many weights to learn

Multi-layer Neural Networks

- **Problems of extended linear units:**
  - fixed basis functions,
  - too many weights
- **One possible solution:**
  - Assume parametric feature (basis) functions
  - Learn the parameters together with the remaining weights

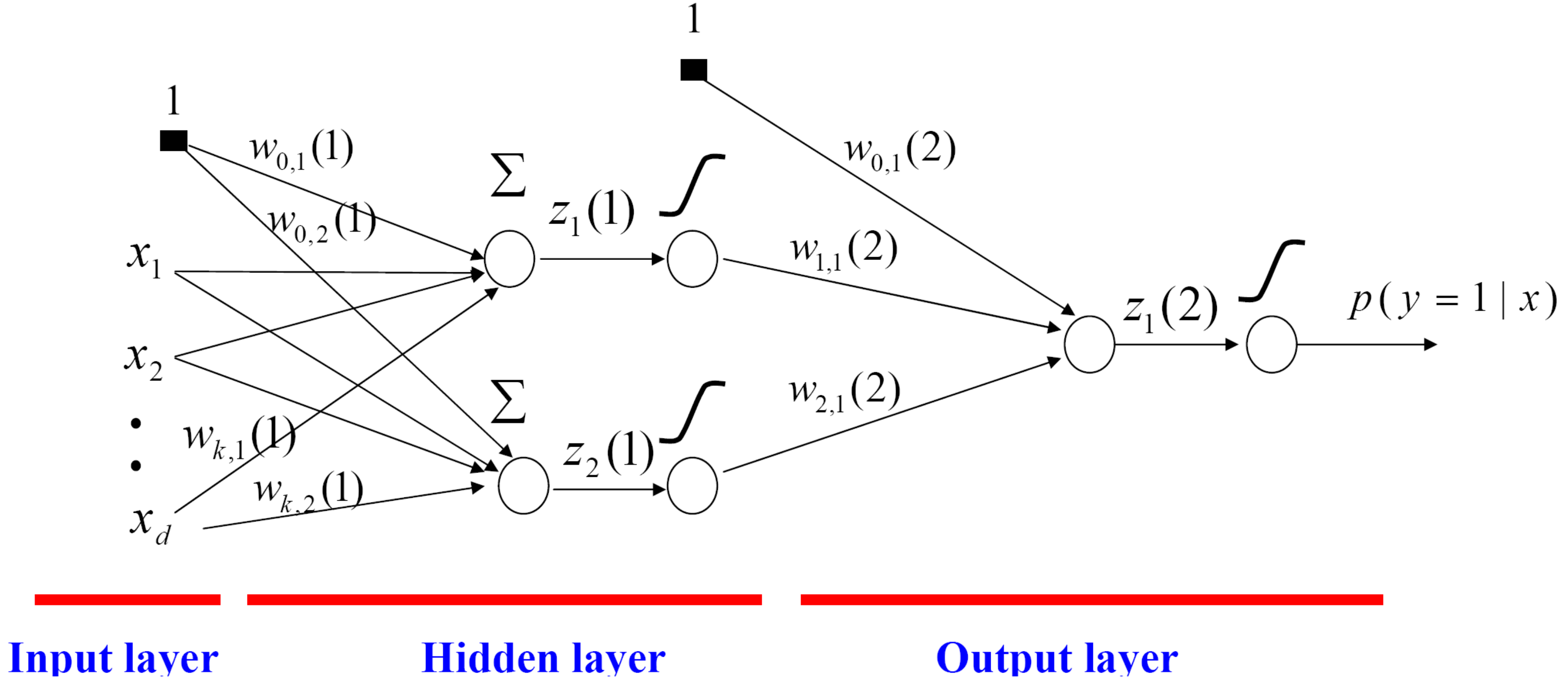


Multi-layer Neural Networks

Also called a **multilayer perceptron (MLP)**

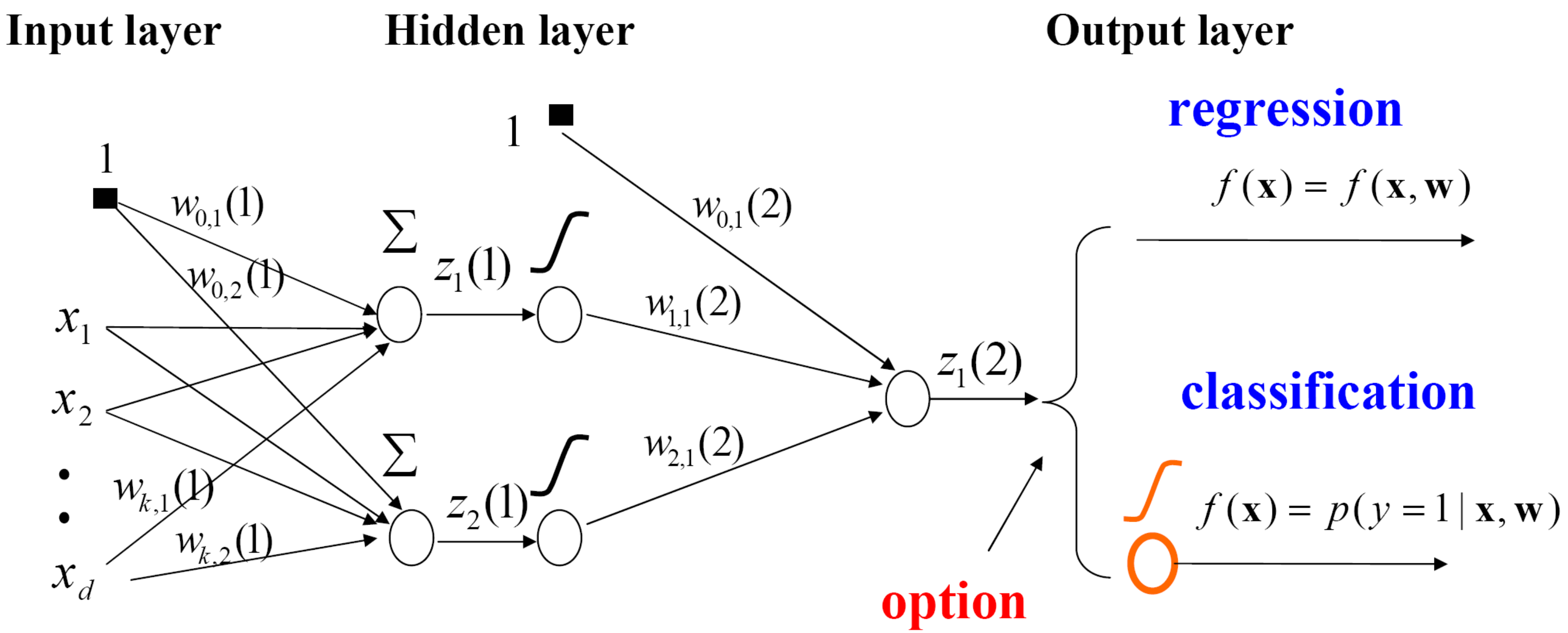
Cascades multiple logistic regression units

Example: (2 layer) classifier with non-linear decision boundaries



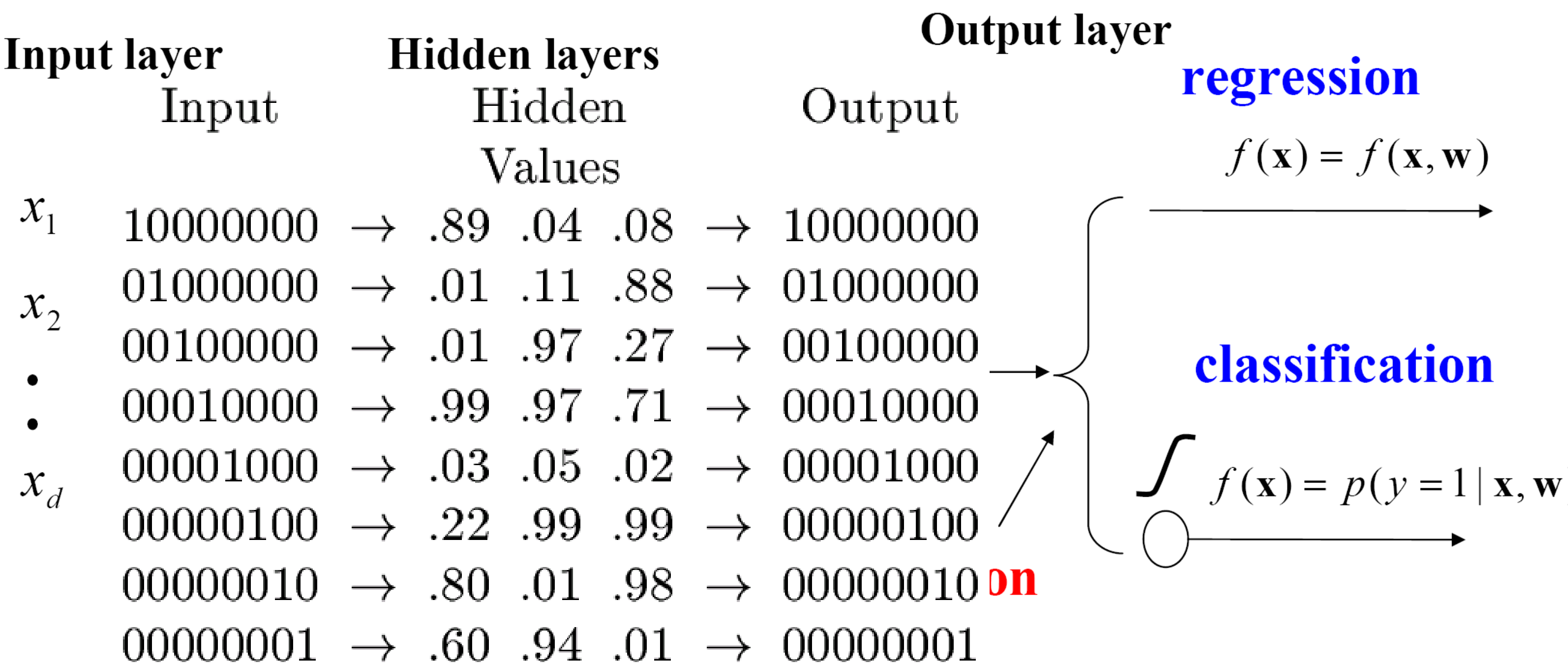
Multi-layer Neural Networks

- Models **non-linearities through logistic regression units**
- Can be applied to both **regression and binary classification problems**



Multi-layer Neural Networks

- **Non-linearities are modeled using multiple hidden logistic regression units (organized in layers)**
- The output layer determines whether it is a **regression or a binary classification problem**

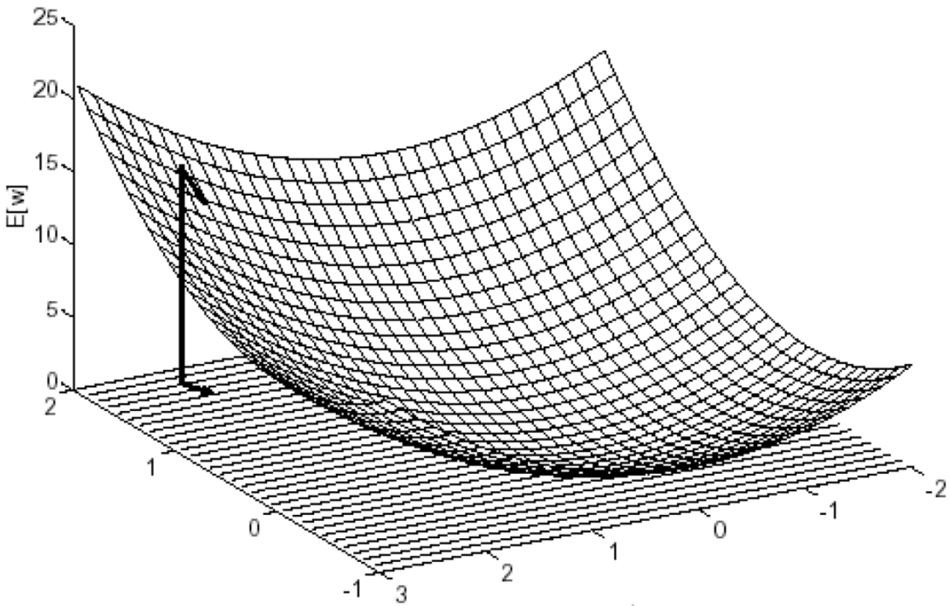


Network Training

- Online Learning
  - weights updated after every training sample
  - significantly faster than offline learning
  - better suited for large datasets
- Offline/Batch Learning
  - weights updated after one epoch

Weight updates based on the error: $J_{\text{online}}(D_i, \mathbf{w})$

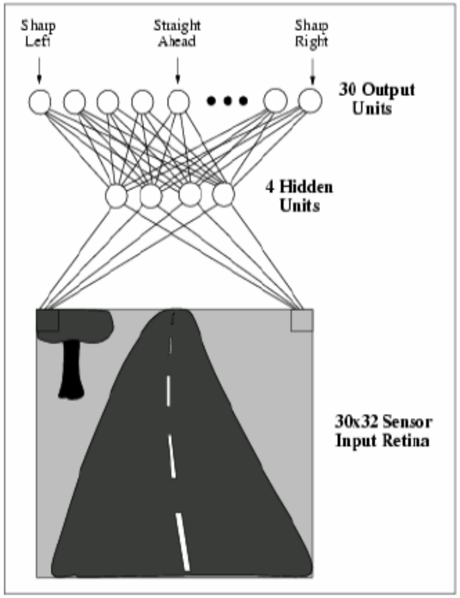
$$w_j \leftarrow w_j - \alpha \frac{\partial}{\partial w_j} J_{\text{online}}(D_i, \mathbf{w})$$

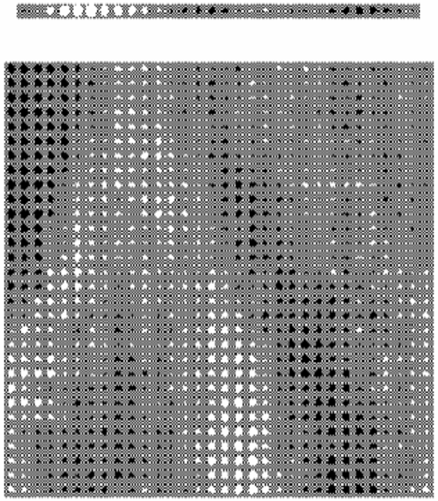


Batch-mode vs Online Mode Learning

- In batch-mode
  - Samples provided and processed together to construct model
  - Need to store samples (not the model)
  - Classical approach for data mining
- In online-mode
  - Samples provided and processed one by one to update model
  - Need to store the model (not the samples)
  - Classical approach for adaptive systems
- But both approaches can be adapted to handle both contexts
  - Samples available together can be exploited one by one
  - Samples provided one by one can be stored and then exploited together

Example Application -- Driving A Car

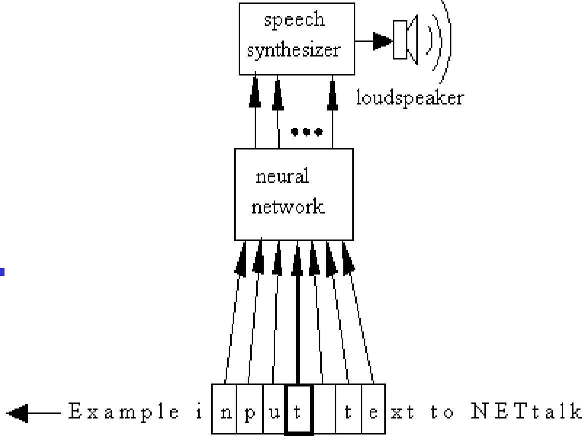




Example Application -- Driving A Car

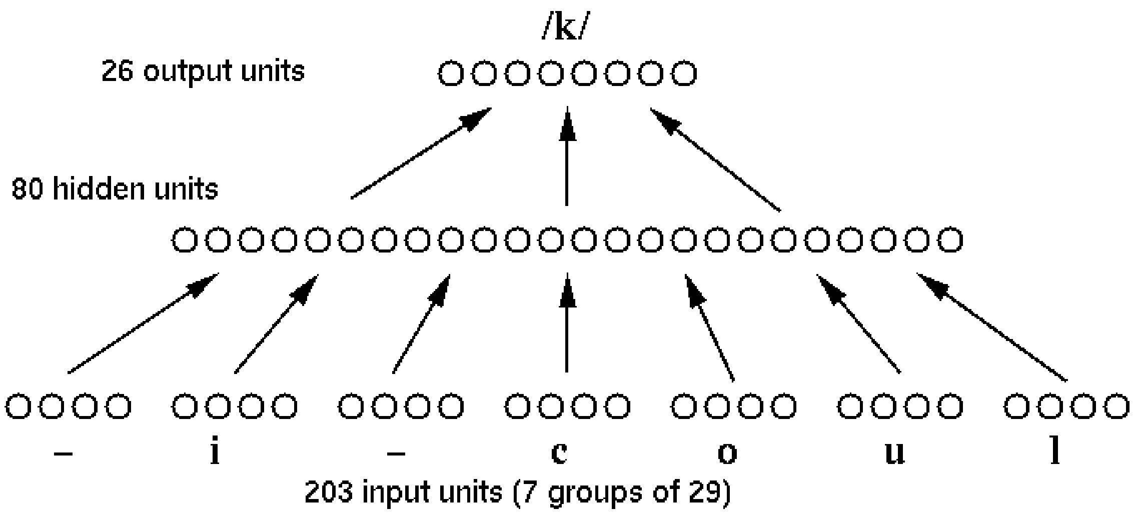
- **Input:** Video image (30 x 32) + range-finder distance (8 x 32)
- **Output:** 45 units representing steering angle
- **Hidden:** 45 units completely connected to input and output
- **Training:** Originally on simulated images. Then in real use
- **Results:** Drives 3mph along woodland road
- **Current Developments:** Recently up to 15 decisions per second, good for 55mph driving along dirt, gravel, and other difficult surfaces. Trained by a real driver.

Nettalk



- Built in response to development of a chip which converts phonemes into sounds of language. Needed something to convert English words into phonemes.
- **Input:** 29 bits per character, 7 character window (for context). *one-of-n-encoding* — for each character, 1 bit is on, the rest are off.
- **Hidden units:** One layer consisting of 80 hidden units, completely connected to inputs and outputs.
- **Output:** 26 units to represent 54 phonemes.
- **Training:** Training on dictionary pronunciation and on transcriptions of speech.

Results

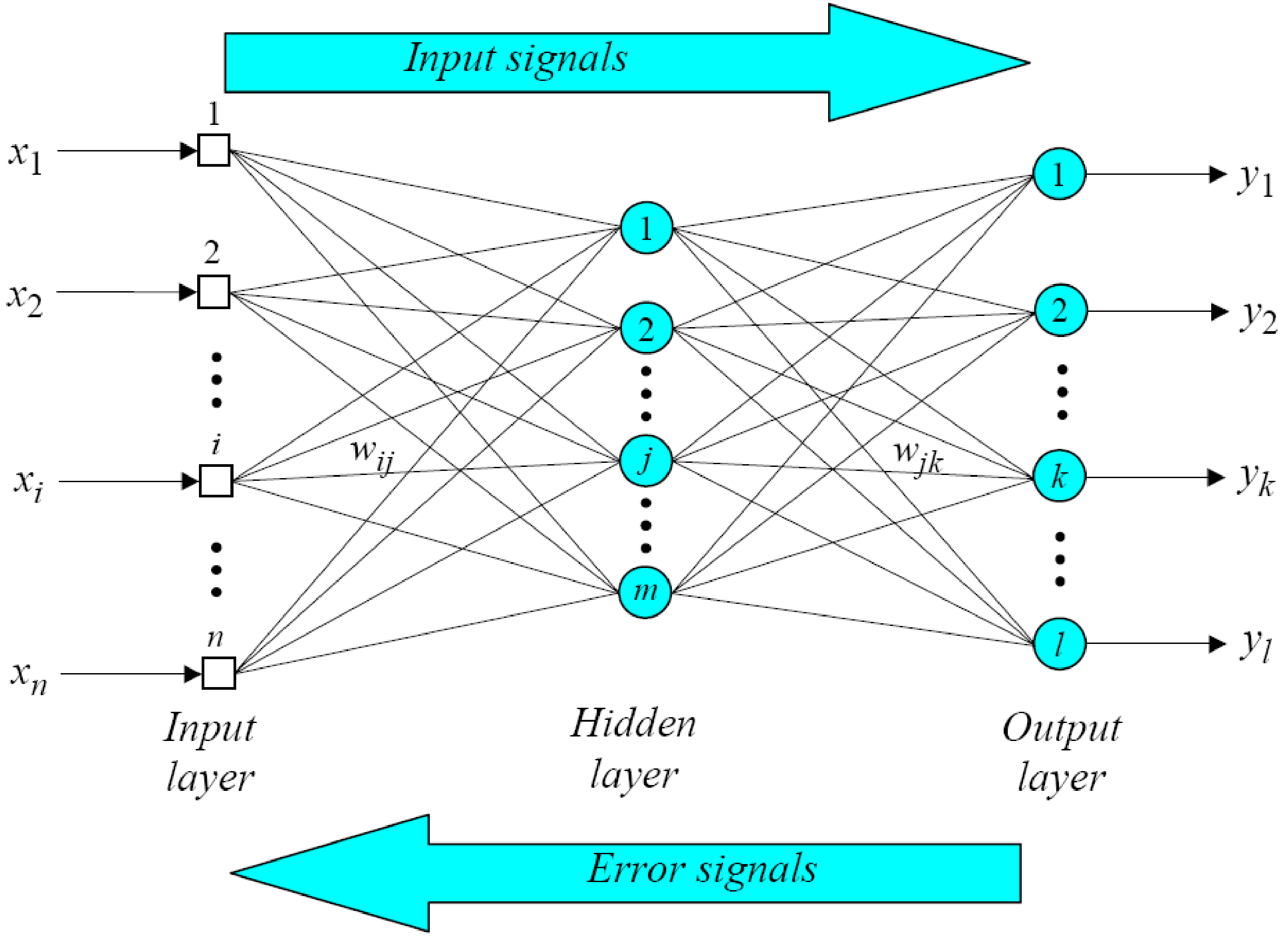


- Understandable speech after 10 learning cycles. After 50 training cycles, apparent error was 5%
- Existing expert system (DecTalk) performed better, but required about 10 person-years of linguistic analysis to generate rules; Nettalk require about 1 month to produce
- Neural Networks can function as an expert system without the need to codify the expertise. Learns from examples

When to Consider Neural Networks

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete or real valued
- Output is a vector of values
- Possibly noise data
- Form of target function is unknown
- Human readability of result is unimportant
- Examples:
  - Speech phoneme recognition
  - Image classification
  - Financial prediction

Three-layer Back-propagation Neural Network



Output

- The response function is normally nonlinear
- Samples include
  - Sigmoid

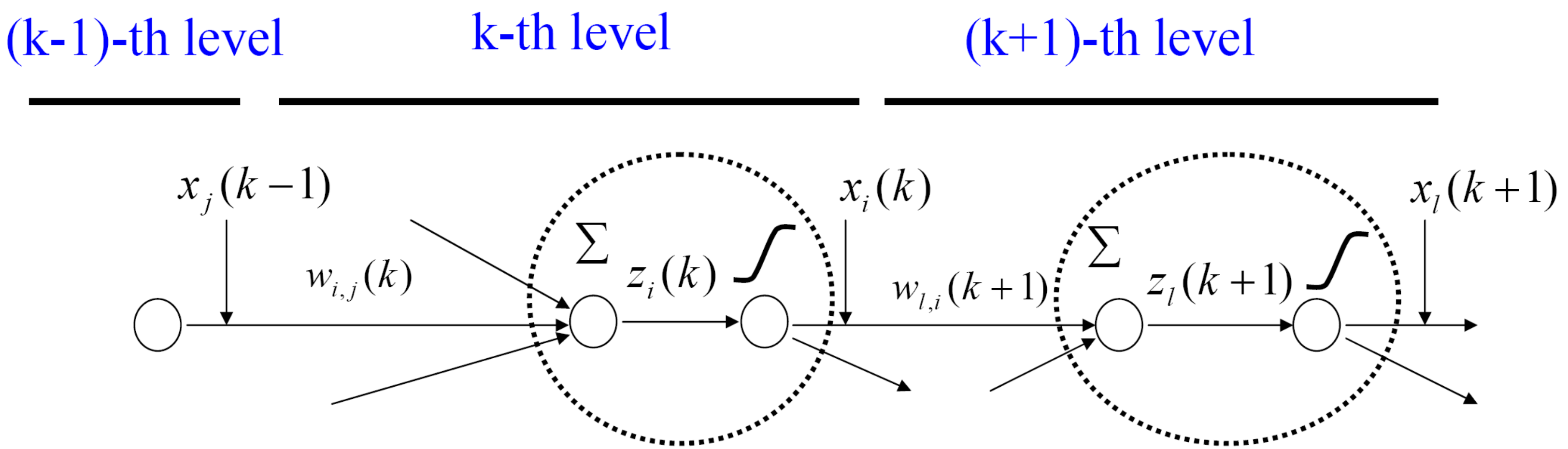
$$f(x) = \frac{1}{1+e^{-\lambda x}}$$

  - Piecewise linear

$$f(x) = \begin{cases} x, if \;\; x \geq \theta \\ 0, if \;\; x < \theta \end{cases}$$

Ref: Tagliarini

Backpropagation



$x_i(k)$ - output of the unit i on level k

$z_i(k)$ - input to the sigmoid function on level k

$w_{i,j}(k)$ - weight between units j and i on levels (k-1) and k

$$z_i(k) = w_{i,0}(k) + \sum_j w_{i,j}(k) x_j(k-1)$$

$$x_i(k) = g(z_i(k))$$

Backpropagation

Update weight $w_{i,j}(k)$ using a data point $D_u = < \mathbf{x}, y >$

$$w_{i,j}(k) \leftarrow w_{i,j}(k) - \alpha \frac{\partial}{\partial w_{i,j}(k)} J_{online}(D_u, \mathbf{w})$$

Let $\delta_i(k) = \frac{\partial}{\partial z_i(k)} J_{online}(D_u, \mathbf{w})$

Then: $\frac{\partial}{\partial w_{i,j}(k)} J_{online}(D_u, \mathbf{w}) = \frac{\partial J_{online}(D_u, \mathbf{w})}{\partial z_i(k)} \frac{\partial z_i(k)}{\partial w_{i,j}(k)} = \delta_i(k) x_j(k-1)$

S.t. $\delta_i(k)$ is computed from $x_i(k)$ and the next layer $\delta_l(k+1)$

$$\delta_i(k) = \left[\sum_l \delta_l(k+1) w_{l,i}(k+1) \right] x_i(k)(1 - x_i(k))$$

Last unit (is the same as for the regular linear units):

$$\delta_i(K) = -(y - f(\mathbf{x}, \mathbf{w}))$$

It is the same for the classification with the log-likelihood measure of fit and linear regression with least-squares error!!!

Learning with MLP

- **Online gradient descent algorithm**
  - Weight update for example D_u:

$$w_{i,j}(k) \leftarrow w_{i,j}(k) - \alpha \frac{\partial}{\partial w_{i,j}(k)} J_{\text{online}}(D_u, \mathbf{w})$$

$$\frac{\partial}{\partial w_{i,j}(k)} J_{online}(D_u, \mathbf{w}) = \frac{\partial J_{online}(D_u, \mathbf{w})}{\partial z_i(k)} \frac{\partial z_i(k)}{\partial w_{i,j}(k)} = \delta_i(k) x_j(k-1)$$

$$w_{i,j}(k) \leftarrow w_{i,j}(k) - \alpha \delta_i(k) x_j(k-1)$$

$x_j(k-1)$ - j-th output of the (k-1) layer

$\delta_i(k)$ - derivative computed via backpropagation

α - a learning rate

Online Gradient Descent Algorithm for MLP

Online-gradient-descent (*D, number of iterations*)

Initialize all weights $w_{i,j}(k)$

for i=1:1: *number of iterations*

do **select** a data point D_u=<$\mathbf{x}$,y> from D

set learning rate α

compute outputs $x_j(k)$ for each unit

compute derivatives $\delta_i(k)$ via **backpropagation**

update all weights (in parallel)

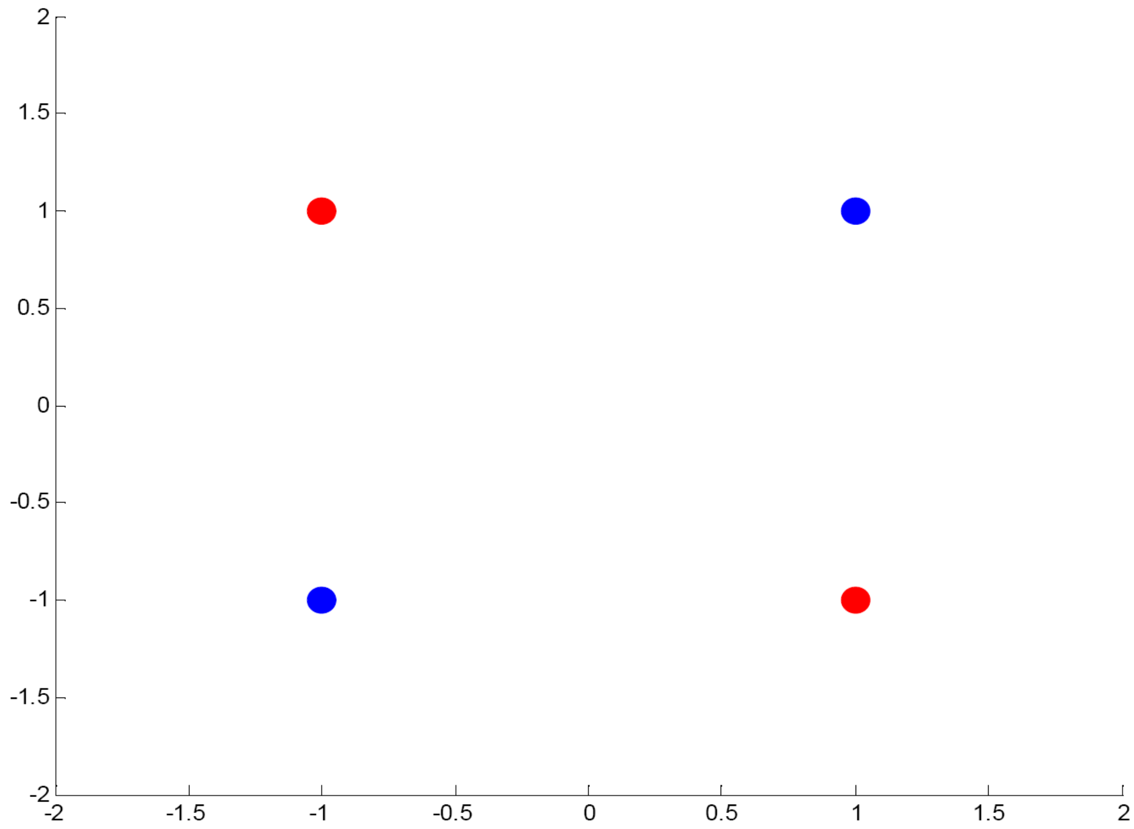
$$w_{i,j}(k) \leftarrow w_{i,j}(k) - \alpha\delta_i(k)x_j(k-1)$$

end for

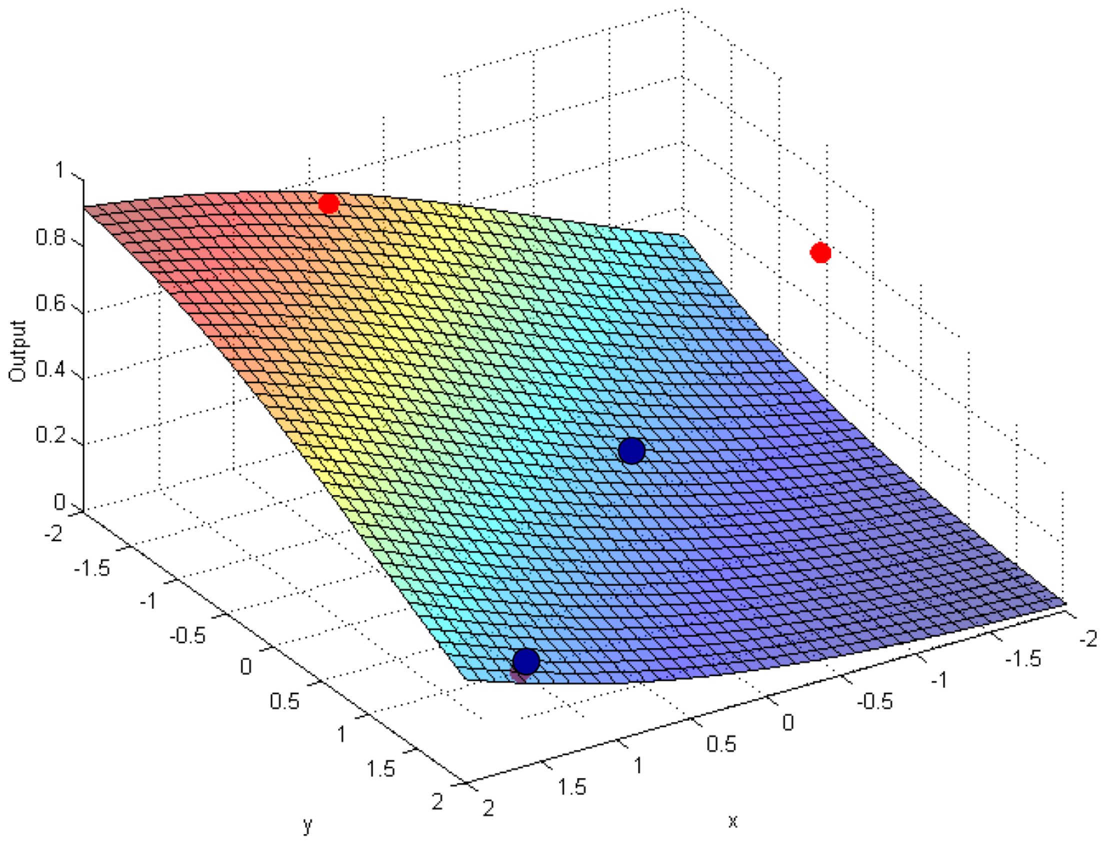
return weights **w**

Xor Example

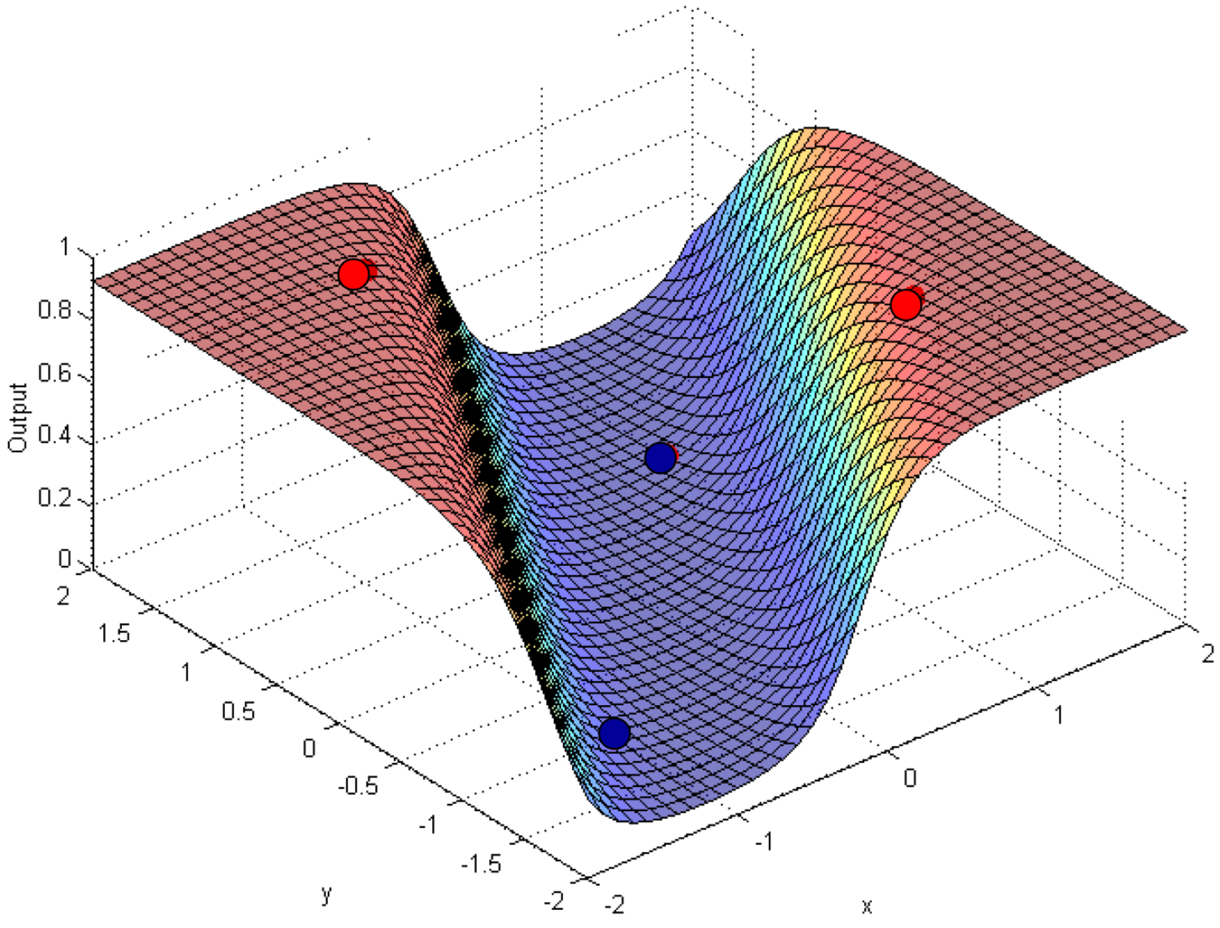
- Linear decision boundary does not exist



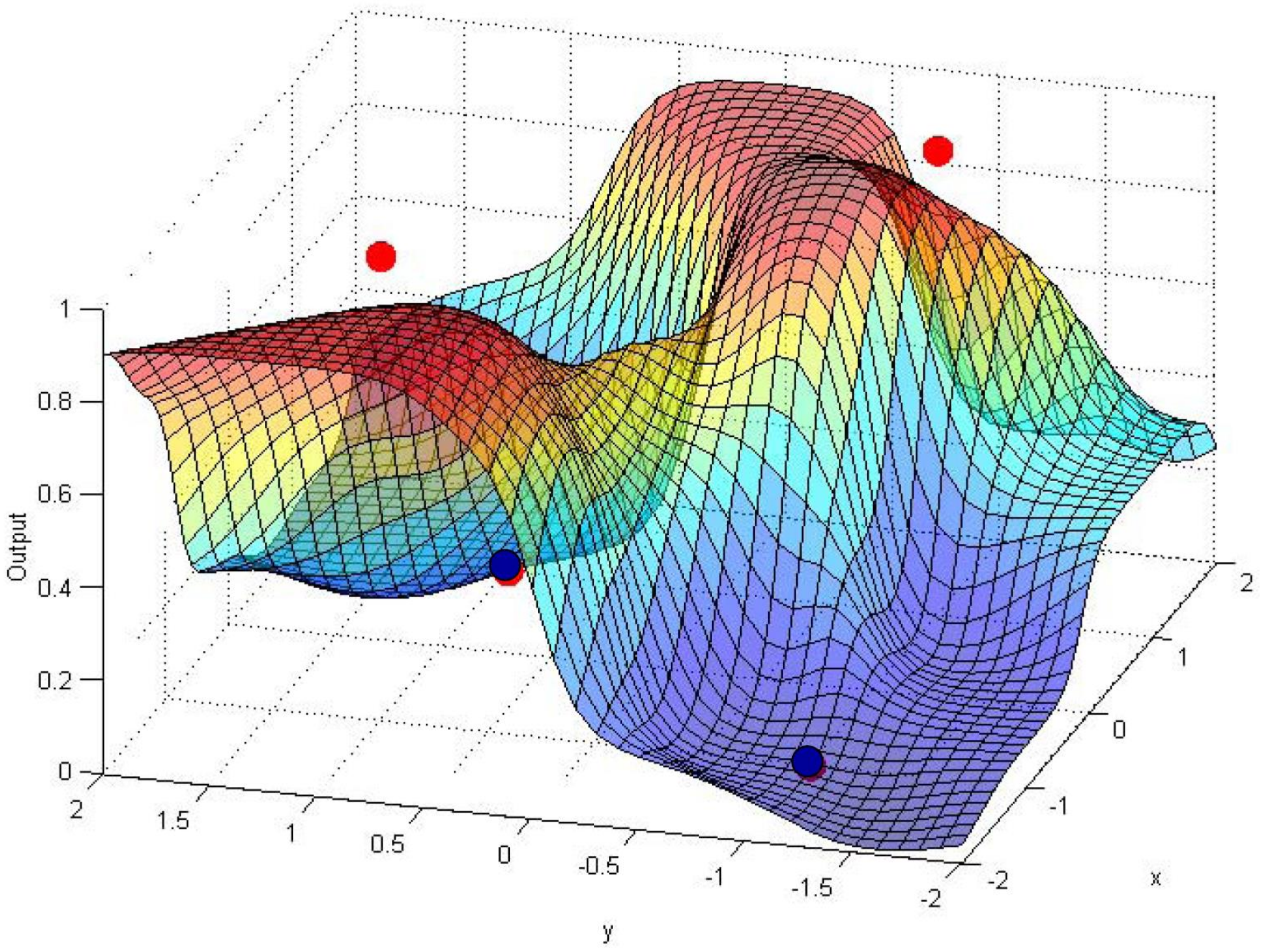
Xor Example Using Linear Unit



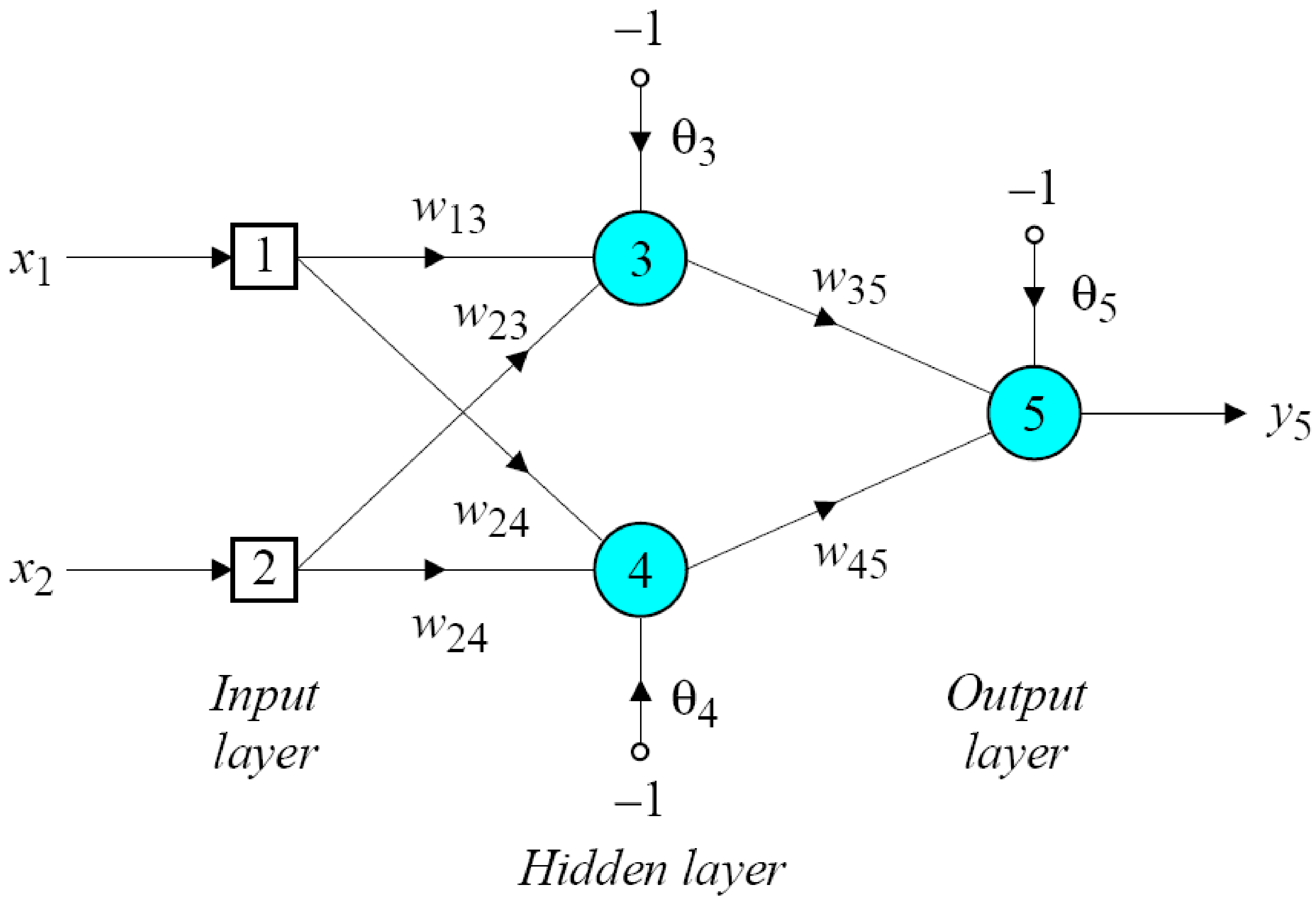
Neural Network with 2 Hidden Units



Neural Network with 10 Hidden Units



Example of Error Back-propagation



Example of Error Back-propagation

- The effect of the threshold applied to a neuron in the hidden or output layer is represented by its weight, θ, connected to a fixed input equal to -1
- The initial weights and threshold levels are set randomly as follows:

$w_{13} = 0.5$, $w_{14} = 0.9$, $w_{23} = 0.4$, $w_{24} = 1.0$, $w_{35} = -1.2$, $w_{45} = 1.1$, $\theta_3 = 0.8$, $\theta_4 = -0.1$, and $\theta_5 = 0.3$.

Example of Error Back-propagation

- We consider a training set where inputs x_1 and x_2 are equal to 1 and desired output $y_{d,5}$ is 0. The actual outputs of neuron 3 and 4 in the hidden layer are calculated as

$$y_3 = sigmoid\ (x_1 w_{13} + x_2 w_{23} - \theta_3) = 1 / \left[1 + e^{-(1 \cdot 0.5 + 1 \cdot 0.4 - 1 \cdot 0.8)} \right] = 0.5250$$

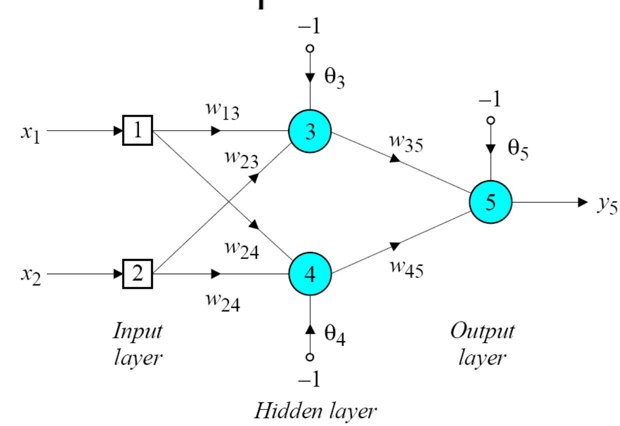
$$y_4 = sigmoid\ (x_1 w_{14} + x_2 w_{24} - \theta_4) = 1 / \left[1 + e^{-(1 \cdot 0.9 + 1 \cdot 1.0 + 1 \cdot 0.1)} \right] = 0.8808$$

- Now the actual output of neuron 5 in the output layer is determined as:

$$y_5 = sigmoid(y_3 w_{35} + y_4 w_{45} - \theta_5) = 1 / \left[1 + e^{-(-0.52501.2 + 0.88081.1 - 1 \cdot 0.3)} \right] = 0.5097$$

- Thus, the following error is obtained:

$$e = y_{d,5} - y_5 = 0 - 0.5097 = -0.5097$$



Example of Error Back-propagation

- The next step is weighting training. To update the weights and threshold levels in our network, we propagate the error, e, from the output layer backward to the input layer.
- First, we calculate the error gradient for neuron 5 in the output layer:

$$\delta_5 = y_5\,(1 - y_5)\,e = 0.5097 \cdot (1 - 0.5097) \cdot (-0.5097) = -0.1274$$

- Then we determine the weight corrections assuming that the learning rate parameter, α, is equal to 0.1:

$$\Delta w_{35} = \alpha \cdot y_3 \cdot \delta_5 = 0.1 \cdot 0.5250 \cdot (-0.1274) = -0.0067$$
$$\Delta w_{45} = \alpha \cdot y_4 \cdot \delta_5 = 0.1 \cdot 0.8808 \cdot (-0.1274) = -0.0112$$
$$\Delta \theta_5 = \alpha \cdot (-1) \cdot \delta_5 = 0.1 \cdot (-1) \cdot (-0.1274) = -0.0127$$

Example of Error Back-propagation

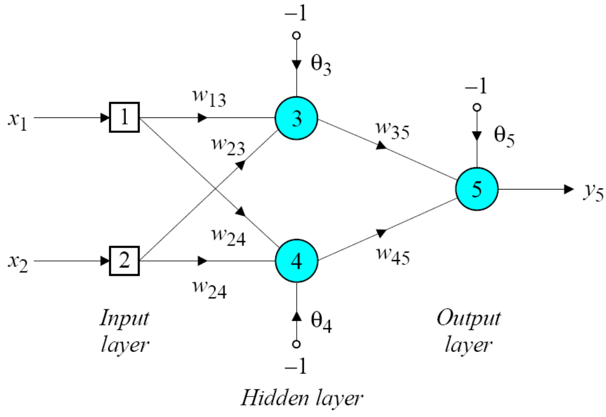
- Next we calculate the error gradients for neurons 3 and 4 in the hidden layer:

$$\delta_3 = y_3(1-y_3)\cdot\delta_5\cdot w_{35} = 0.5250\cdot(1-0.5250)\cdot(-0.1274)\cdot(-1.2) = 0.0381$$

$$\delta_4 = y_4(1-y_4)\cdot\delta_5\cdot w_{45} = 0.8808\cdot(1-0.8808)\cdot(-0.127\,4)\cdot 1.1 = -0.0147$$

- We determine the weight corrections:

$$\Delta w_{13} = \alpha\cdot x_1\cdot\delta_3 = 0.1\cdot 1\cdot 0.0381 = 0.0038$$
$$\Delta w_{23} = \alpha\cdot x_2\cdot\delta_3 = 0.1\cdot 1\cdot 0.0381 = 0.0038$$
$$\Delta\theta_3 = \alpha\cdot(-1)\cdot\delta_3 = 0.1\cdot(-1)\cdot 0.0381 = -0.0038$$
$$\Delta w_{14} = \alpha\cdot x_1\cdot\delta_4 = 0.1\cdot 1\cdot(-0.0147) = -0.0015$$
$$\Delta w_{24} = \alpha\cdot x_2\cdot\delta_4 = 0.1\cdot 1\cdot(-0.0147) = -0.0015$$
$$\Delta\theta_4 = \alpha\cdot(-1)\cdot\delta_4 = 0.1\cdot(-1)\cdot(-0.0147) = 0.0015$$



Example of Error Back-propagation

- At last, we update all weights and threshold:

$$w_{13} = w_{13} + \Delta w_{13} = 0.5 + 0.0038 = 0.5038$$

$$w_{14} = w_{14} + \Delta w_{14} = 0.9 - 0.0015 = 0.8985$$

$$w_{23} = w_{23} + \Delta w_{23} = 0.4 + 0.0038 = 0.4038$$

$$w_{24} = w_{24} + \Delta w_{24} = 1.0 - 0.0015 = 0.9985$$

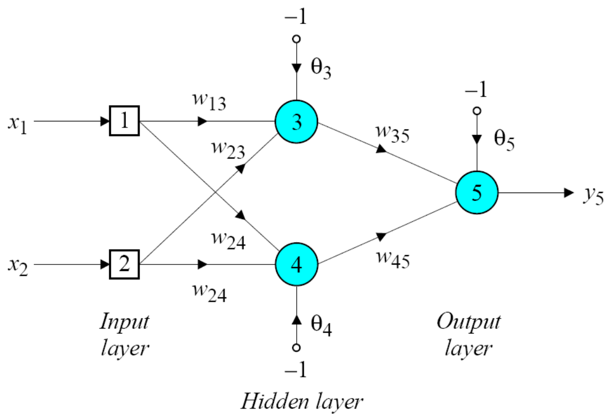
$$w_{35} = w_{35} + \Delta w_{35} = -1.2 - 0.0067 = -1.2067$$

$$w_{45} = w_{45} + \Delta w_{45} = 1.1 - 0.0112 = 1.0888$$

$$\theta_3 = \theta_3 + \Delta\theta_3 = 0.8 - 0.0038 = 0.7962$$

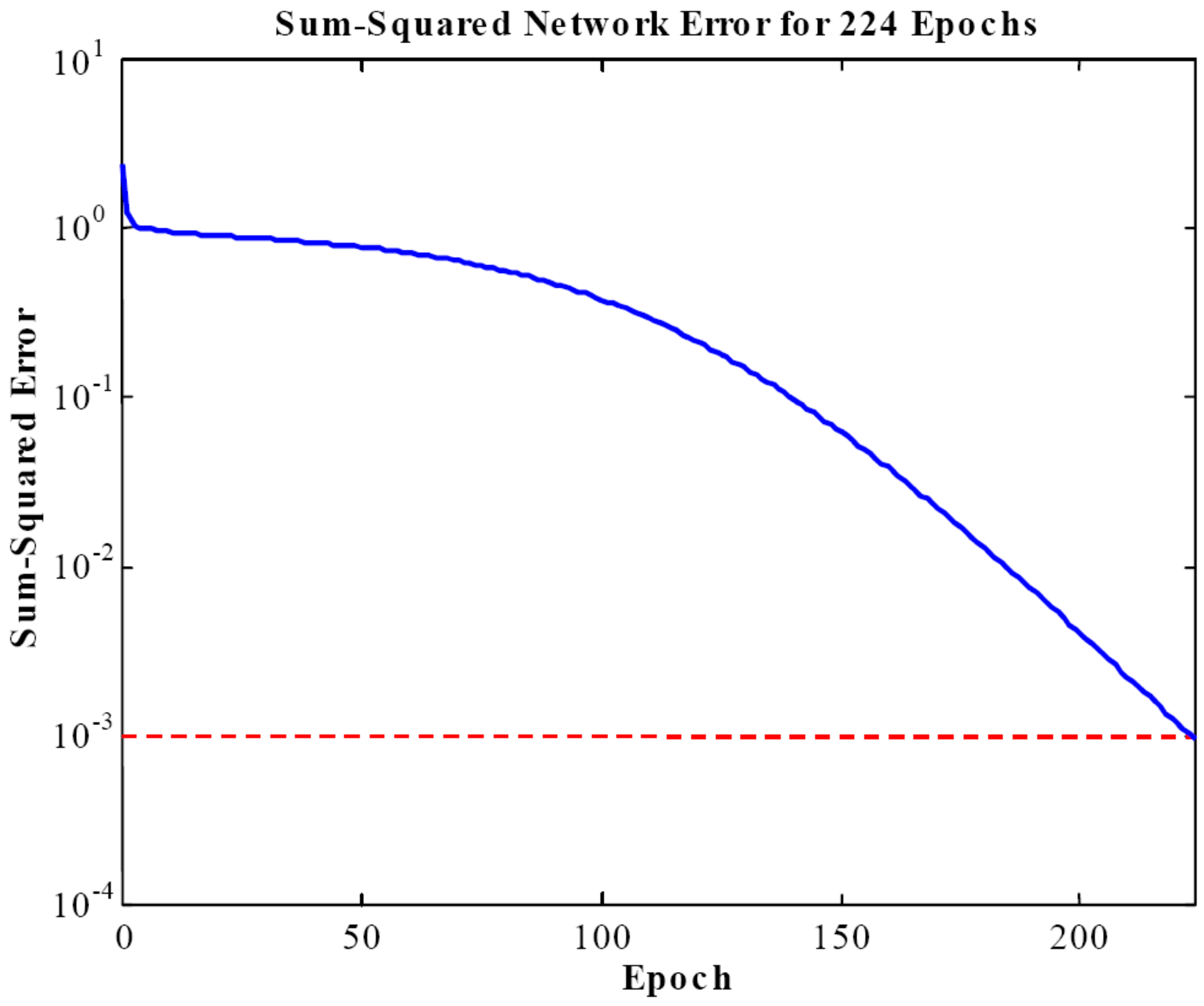
$$\theta_4 = \theta_4 + \Delta\theta_4 = -0.1 + 0.0015 = -0.0985$$

$$\theta_5 = \theta_5 + \Delta\theta_5 = 0.3 + 0.0127 = 0.3127$$



- The training process is repeated until the sum of squared errors is less than 0.001.

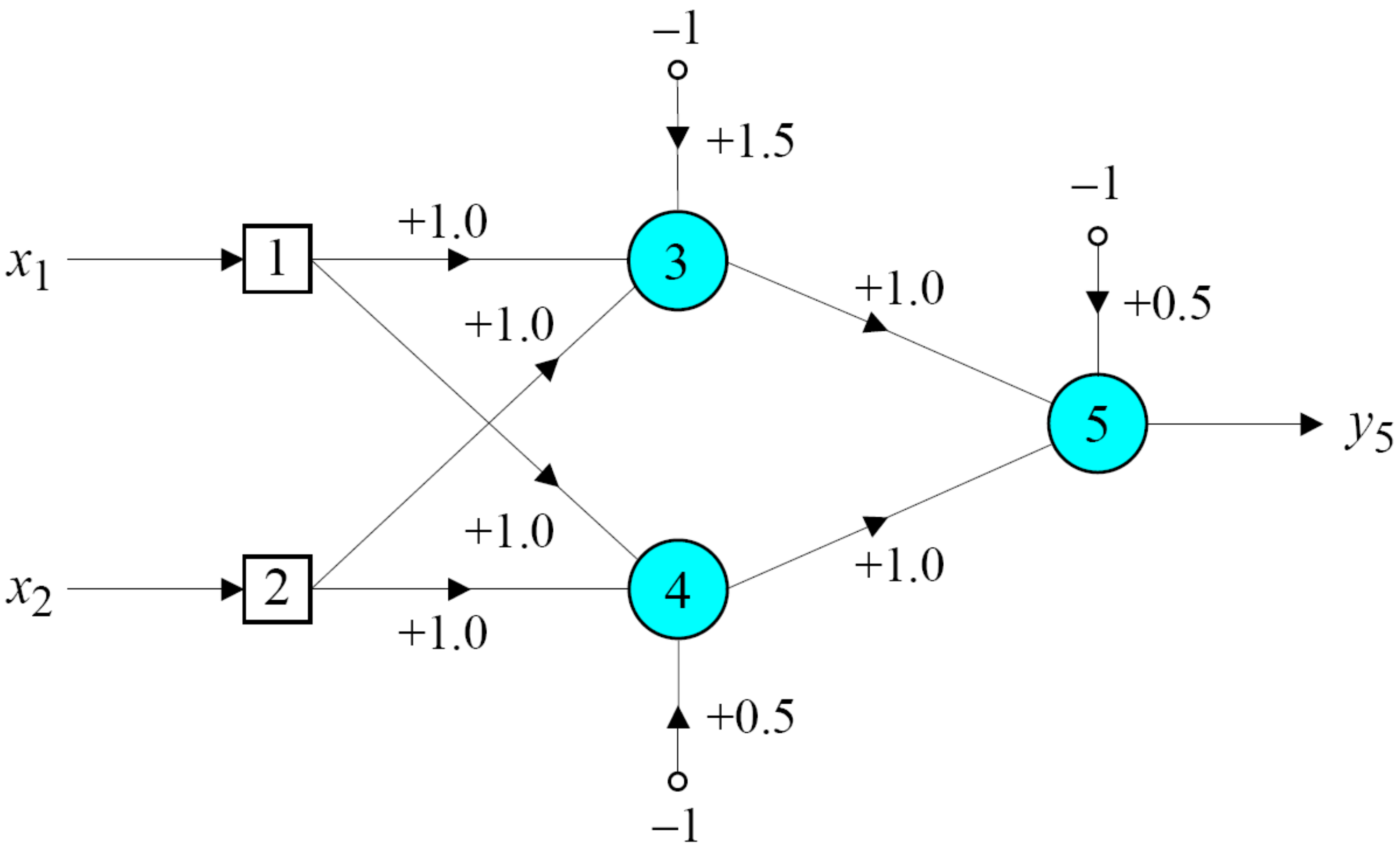
Learning Curve for Operation Exclusive-OR



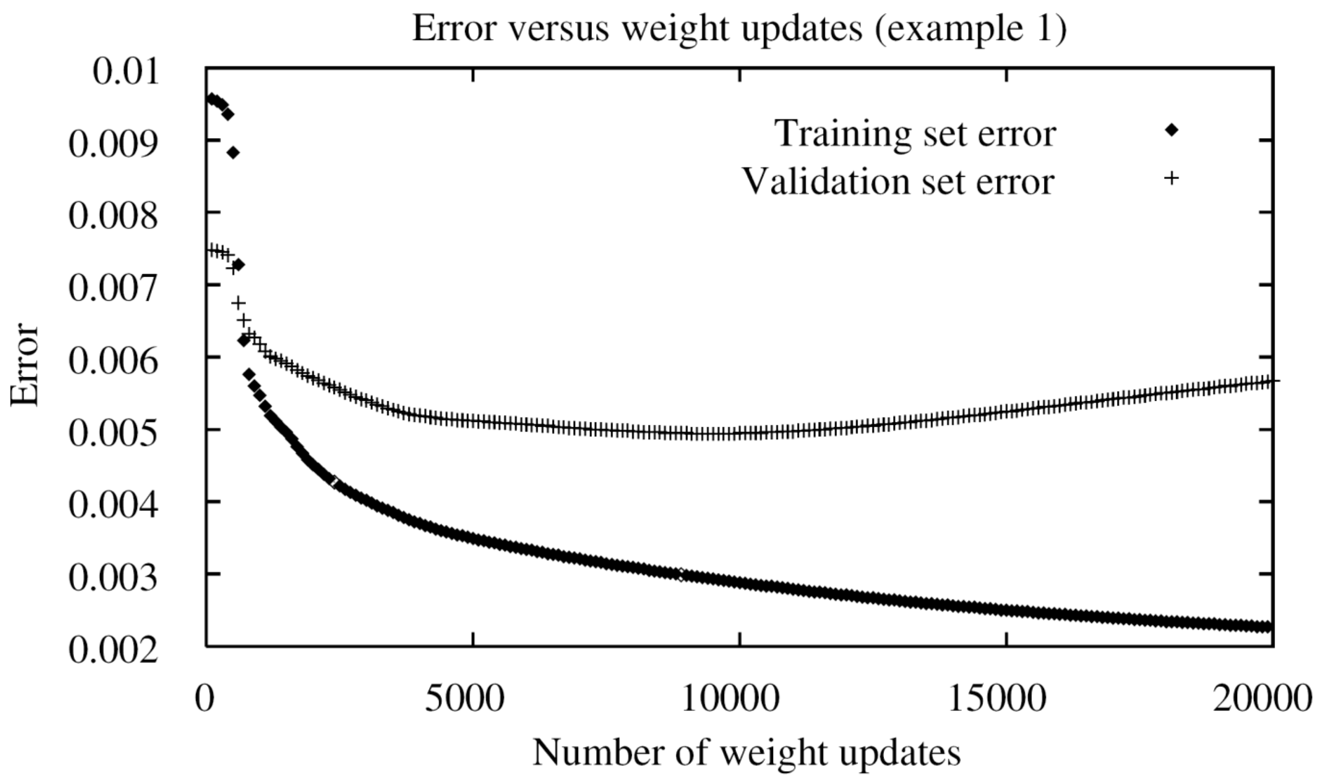
Final Results of Three-layer Network Learning

Inputs		Desired output	Actual output	Error	Sum of squared errors
x_1	x_2	y_d	y_5	e	
1	1	0	0.0155	–0.0155	0.0010
0	1	1	0.9849	0.0151	
1	0	1	0.9849	0.0151	
0	0	0	0.0175	–0.0175	

Solution for Exclusive-OR operation



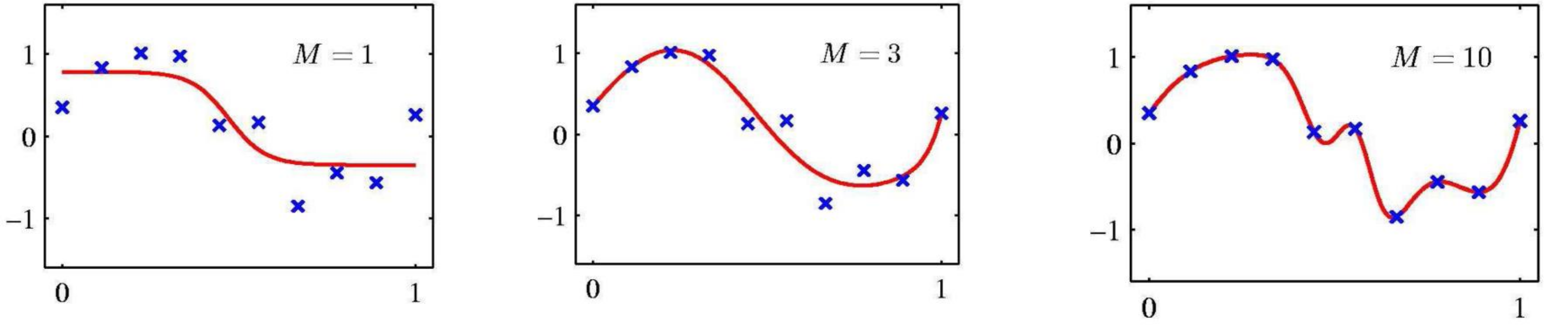
Overfitting in Neural Network



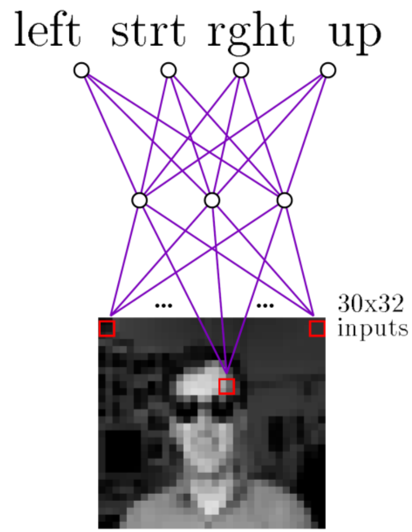
Alternative Error Function

- Penalize large weights:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$



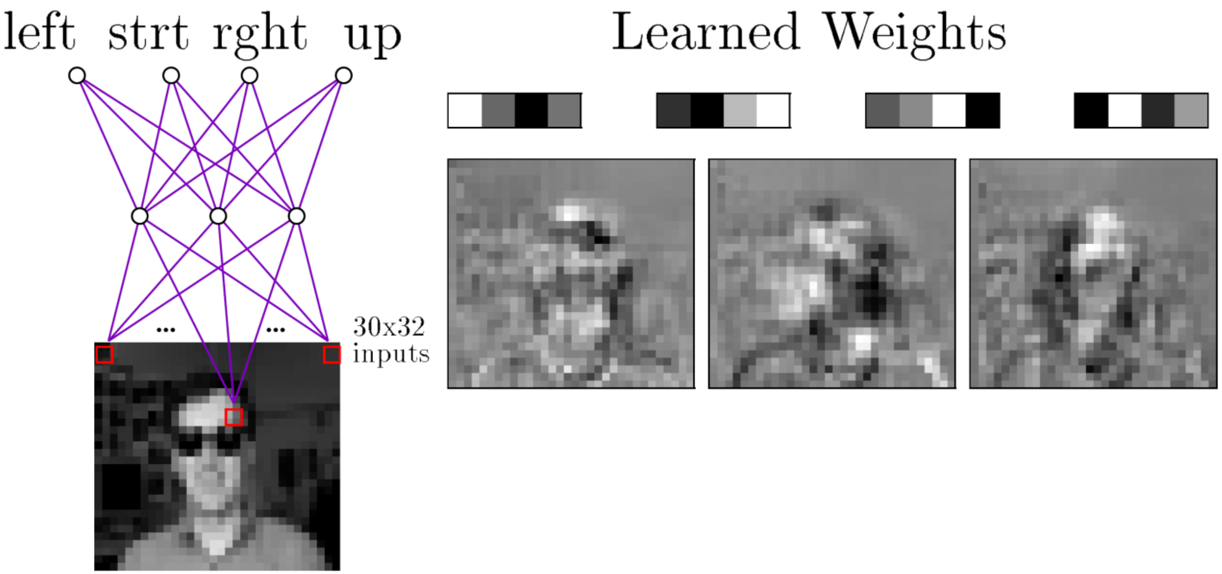
Neural Networks for Face Recognition



Typical input images

- 90% accurate learning head pose, and recognizing 1-of-20 faces

Learned Hidden Unit Weights



Typical input images