R·I·T

Neural Networks (Overview)

Prof. Richard Zanibbi

Introduction

Inspired by Biology

But as used in pattern recognition research, have little relation with real neural systems (studied in *neurology* and *neuroscience*)

Kuncheva: the literature 'on NNs is excessive and continuously growing.'

Early Work

McCullough and Pitts (1943)

Introduction, Continued

Neural Nets Encode a Function

Represents function f : $R^n \rightarrow R^c$ where n is the dimensionality of the input space, c the output space

- Classification: map feature space to values for c discriminant functions: choose maximum discriminant value (most 'activated' output node)
- Regression: learn continuous outputs directly (e.g. learn to fit the sin function - see Bishop text)

Training (for Classification)

$$E = \frac{1}{2}\sum_{j=1}^{N}\sum_{i=1}^{c}\{g_i(\mathbf{z}_j) - \mathcal{I}(\omega_i, l(\mathbf{z}_j))\}^2$$

Minimize error on outputs (i.e. maximize function approximation) for a training set, most often the *squared error* (above)

Introduction, Continued

Granular Representation

A set of interacting elements ('neurons' or nodes) map input values to output values using structured series of interactions

Properties

- Instable: like decision trees, small changes in training data can alter NN behavior significantly
  - Also like decision trees, prone to overfitting: validation set often used to stop training
- Expressive: With proper design and training, can approximate any function to a specified precision

Expressive Power of NNs

Using Squared Error for Learning Classification Functions

For infinite data, the discriminant functions learned by a NN approach the true posterior probabilities classes (for multi-layer perceptrons (MLP), and radial basis function (RBF) networks):

$$\lim_{N \to \infty} g_i(\mathbf{x}) = P(\omega_i|\mathbf{x}), \qquad \mathbf{x} \in \Re^n$$

Note

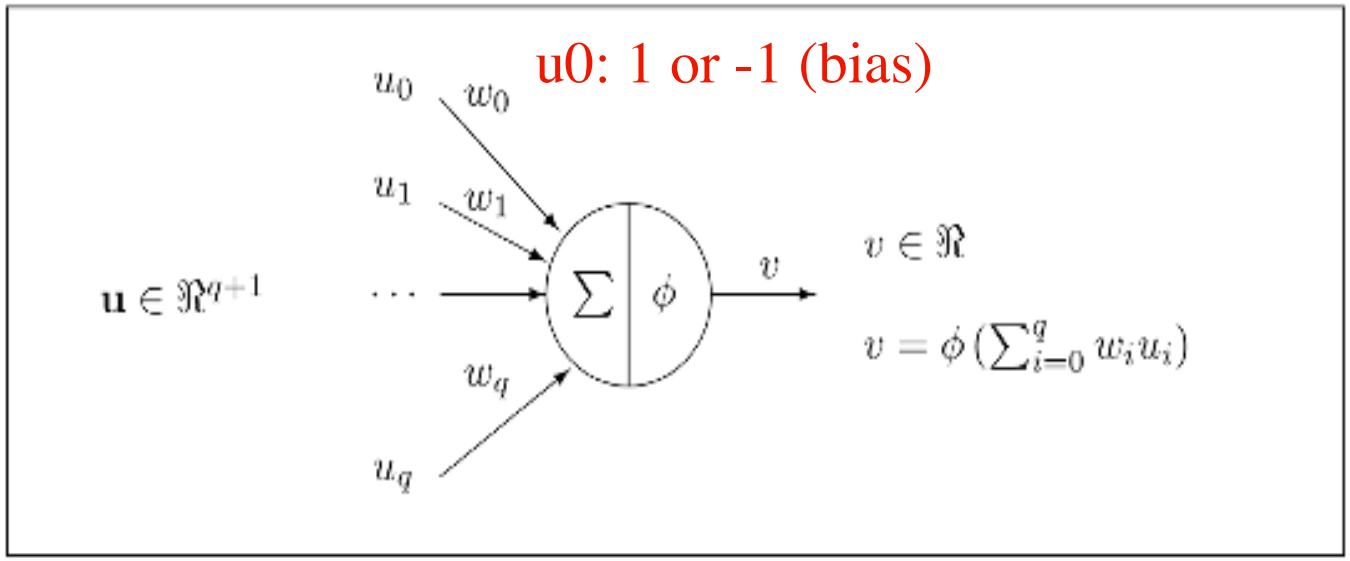
This result applies to any classifier that can approximate an arbitrary discriminant function with a specified precision (not specific to NNs)

A Single Neuron (Node)

Let $\mathbf{u} = [u_0, \ldots, u_q]^T \in \Re^{q+1}$ be the input vector to the node and $v \in \Re$ be its output. We call $\mathbf{w} = [w_0, \ldots, w_q]^T \in \Re^{q+1}$ a vector of *synaptic weights*. The processing element implements the function

$$v = \phi(\xi); \qquad \xi = \sum_{i=0}^{q} w_i u_i \tag{2.79}$$

where $\phi : \Re \to \Re$ is the *activation function* and ξ is the *net sum*.



R·I·T

Common Activation Functions

- The threshold function

ξ : (net sum)

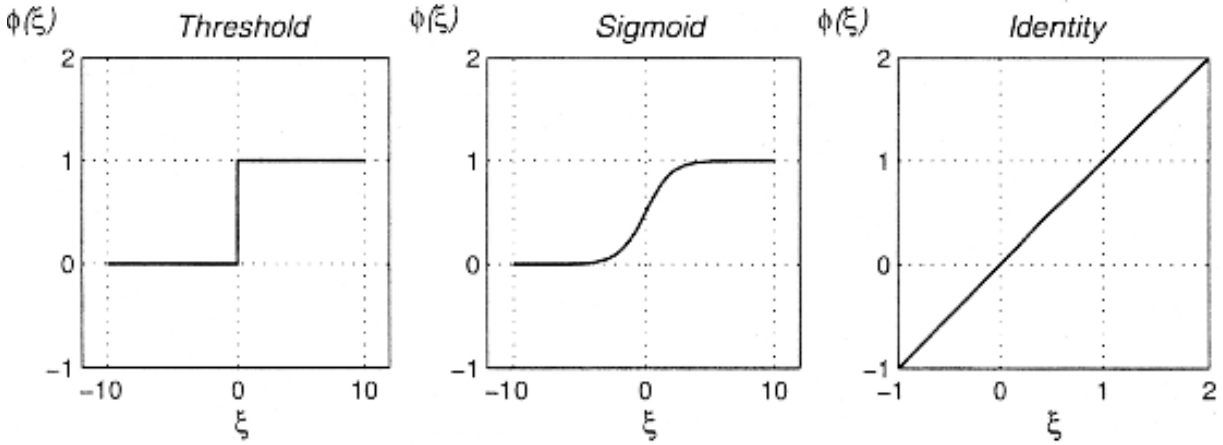
$$\phi(\xi) = \begin{cases} 1, & \text{if } \xi \geq 0, \\ 0, & \text{otherwise.} \end{cases}$$

- The sigmoid function

$$\phi(\xi) = \frac{1}{1 + \exp(-\xi)} \qquad \boxed{\phi'(\xi) = \phi(\xi)[1 - \phi(\xi)]}$$

- The identity function

$$\phi(\xi) = \xi \qquad \textit{(used for input nodes)}$$



R·I·T

Bias: Offset/Translation for Activation Functions

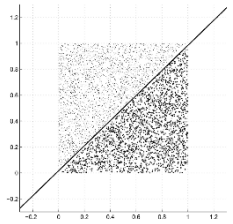
The weight "$-w_0$" is used as a *bias*, and the corresponding input value u_0 is set to 1. Equation (2.79) can be rewritten as

$$v = \phi[\zeta - (-w_0)] = \phi\left[\sum_{i=1}^{q} w_i u_i - (-w_0)\right] \tag{2.83}$$

where ζ is now the weighted sum of the weighted inputs from 1 to q. Geometrically, the equation

$$\sum_{i=1}^{q} w_i u_i - (-w_0) = 0 \tag{2.84}$$

defines a hyperplane in $\Re^q$. A node with a threshold activation function (2.80) responds with value $+1$ to all inputs $[u_1, \ldots, u_q]^T$ on the one side of the hyperplane, and value 0 to all inputs on the other side.



The Perceptron (Rosenblatt, 1962)

$$v = \phi(\xi); \qquad \xi = \sum_{i=0}^{q} w_i u_i$$

Defines linear decision boundary

$$\phi(\xi) = \begin{cases} 1, & \text{if } \xi \geq 0, \\ -1, & \text{otherwise.} \end{cases}$$

Where activation had value 0.

Update Rule:

$$\mathbf{w} \longleftarrow \boxed{\mathbf{w} - v\eta\mathbf{z}_j} \qquad (2.86)$$

where v is the output of the perceptron for $\mathbf{z}_j$ and η is a parameter specifying the *learning rate*

Learning Algorithm:

- Set all input weights (**w**) randomly (e.g. in [0,1])
- Apply the weight update rule when a misclassification is made
- Pass over training data (Z), applying the update rule whenever there is an error, until a pass is made where no errors occur. One pass = one *epoch*

R·I·T

Properties of Perceptron Learning

Convergence and Zero Error!

If two classes are linearly separable in feature space, always converges to a function producing no error on the *training* set

Infinite Looping and No Guarantees!

If classes not linearly separable. If stopped early, no guarantee that last function learned is the best considered during training

Example: Perceptron Learning

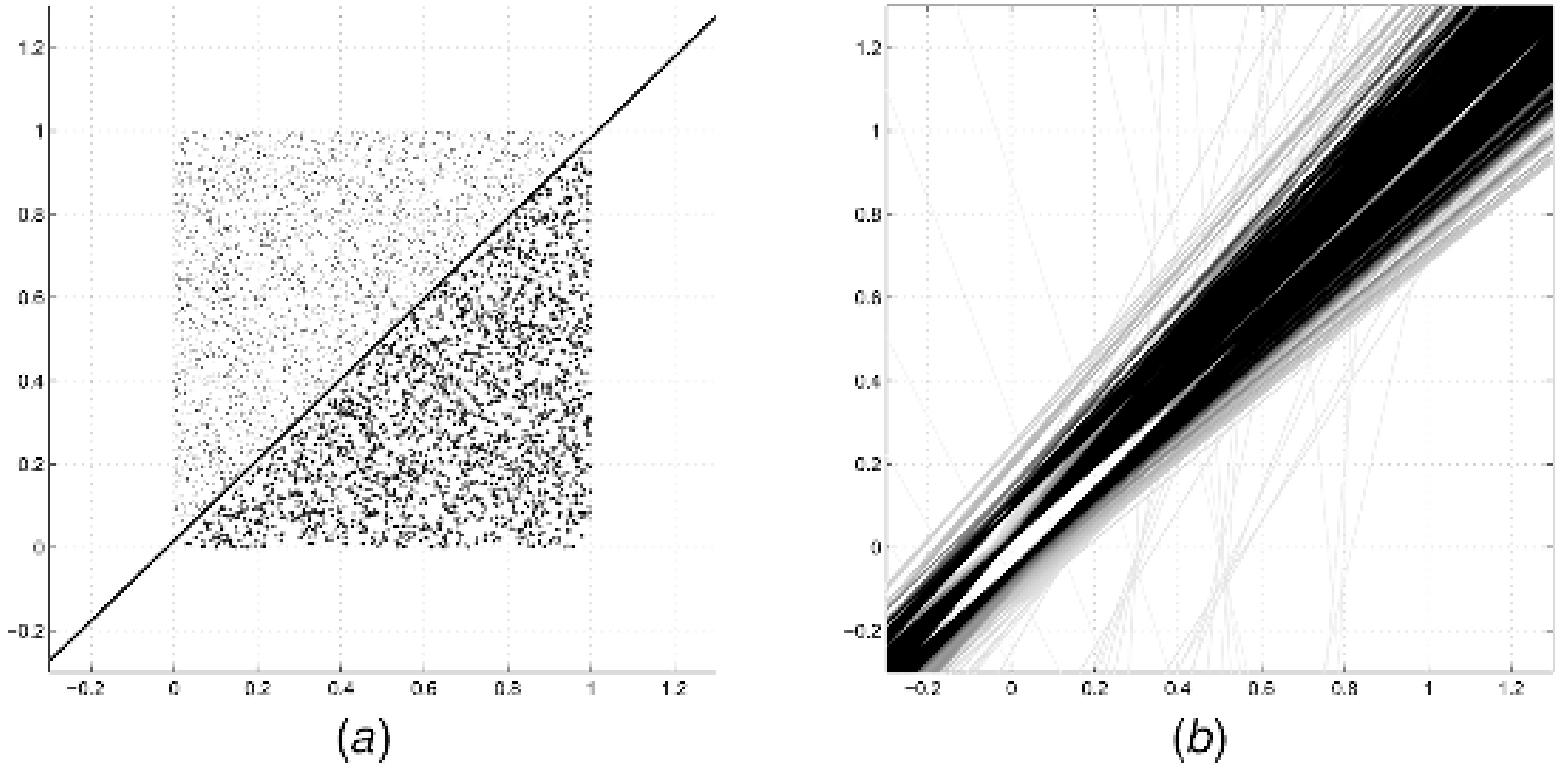
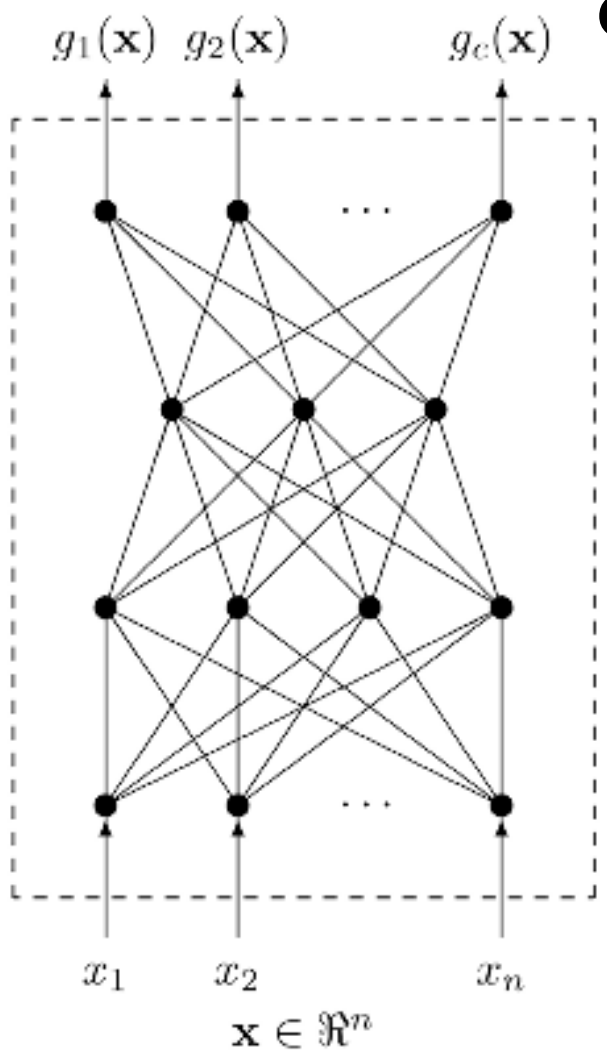


Fig. 2.16 (*a*) *Uniformly distributed two-class data and the boundary found by the perceptron training algorithm.* (*b*) *The "evolution" of the class boundary.*

Multi-Layer Perceptron

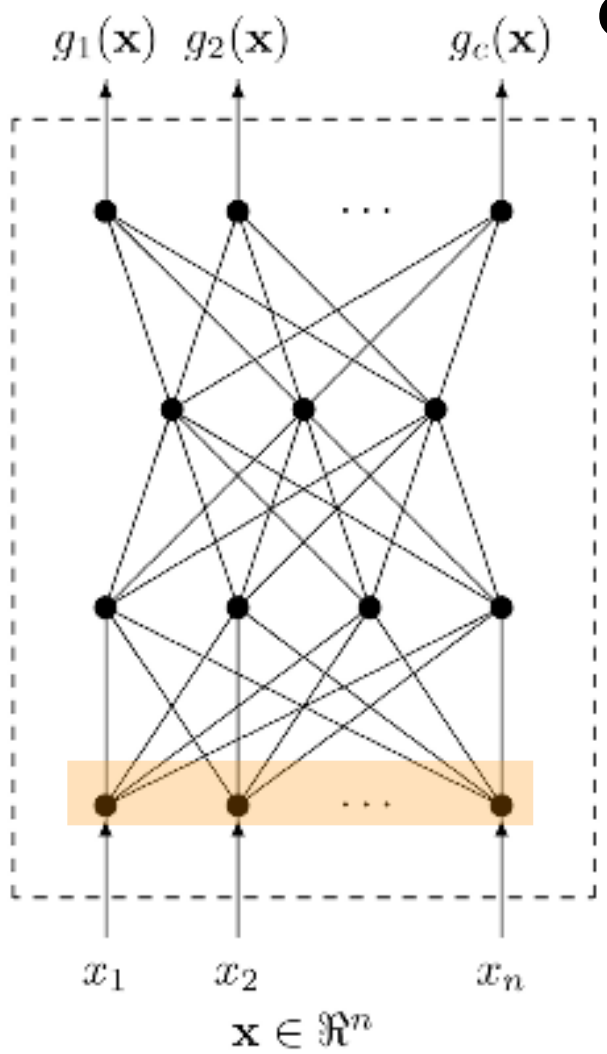


Correct Output:
(0,0,..., 1)

- Hidden layers have the same activation function (threshold or sigmoid)
- Classification is feed-forward: compute activations one layer at a time, input to ouput: decide ω_i for *max* $g_i(X)$
- Learning is through backpropagation: error metric used to update input weights based on observed activations (outputs)

Fig. 2.17 *A generic model of an MLP classifier.*

Multi-Layer Perceptron



Correct Output:
(0,0,..., 1)

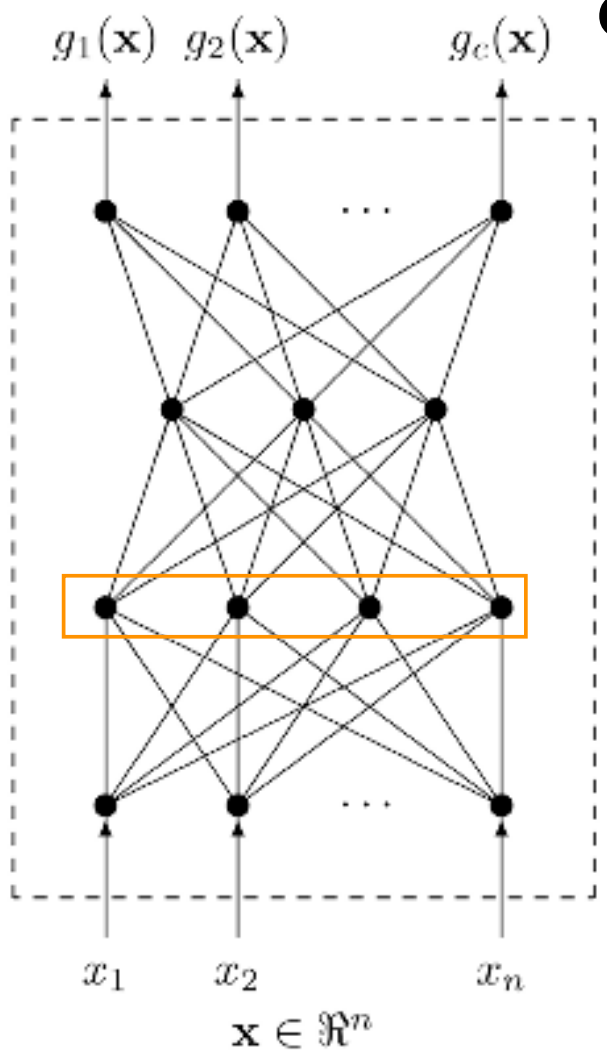
(activation: identity fn)

- Hidden layers have the same activation function (threshold or sigmoid)
- Classification is feed-forward: compute activations one layer at a time, input to ouput: decide ω_i for *max* $g_i(X)$
- Learning is through backpropagation: error metric used to update input weights based on observed activations (outputs)

R·I·T

Fig. 2.17 *A generic model of an MLP classifier.*

Multi-Layer Perceptron



Correct Output:
(0,0,..., 1)

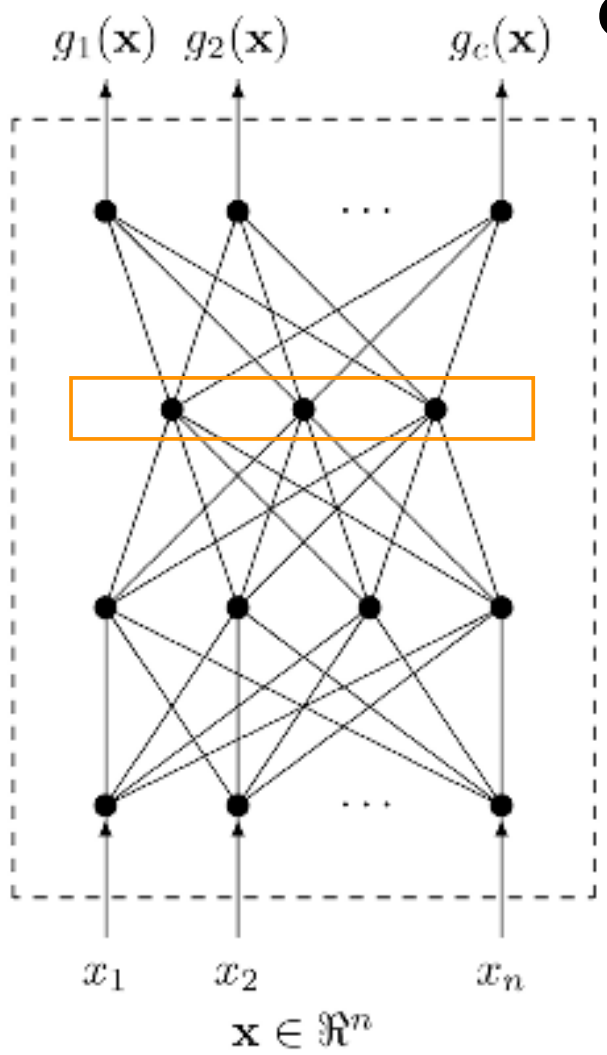
(activation: identity fn)

- Hidden layers have the same activation function (threshold or sigmoid)
- Classification is feed-forward: compute activations one layer at a time, input to ouput: decide ω_i for *max* $g_i(X)$
- Learning is through backpropagation: error metric used to update input weights based on observed activations (outputs)

R·I·T

Fig. 2.17 *A generic model of an MLP classifier.*

Multi-Layer Perceptron



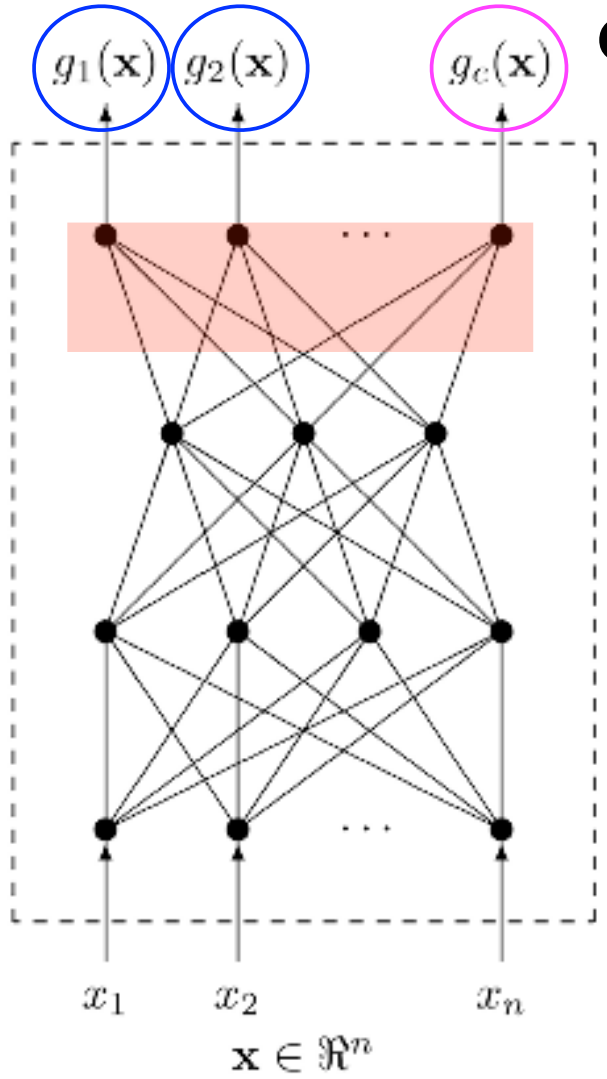
Correct Output:
(0,0,..., 1)

(activation: identity fn)

- Hidden layers have the same activation function (threshold or sigmoid)
- Classification is feed-forward: compute activations one layer at a time, input to ouput: decide ω_i for *max* $g_i(X)$
- Learning is through backpropagation: error metric used to update input weights based on observed activations (outputs)

Fig. 2.17 *A generic model of an MLP classifier.*

Multi-Layer Perceptron



Correct Output:
(0,0,..., 1)

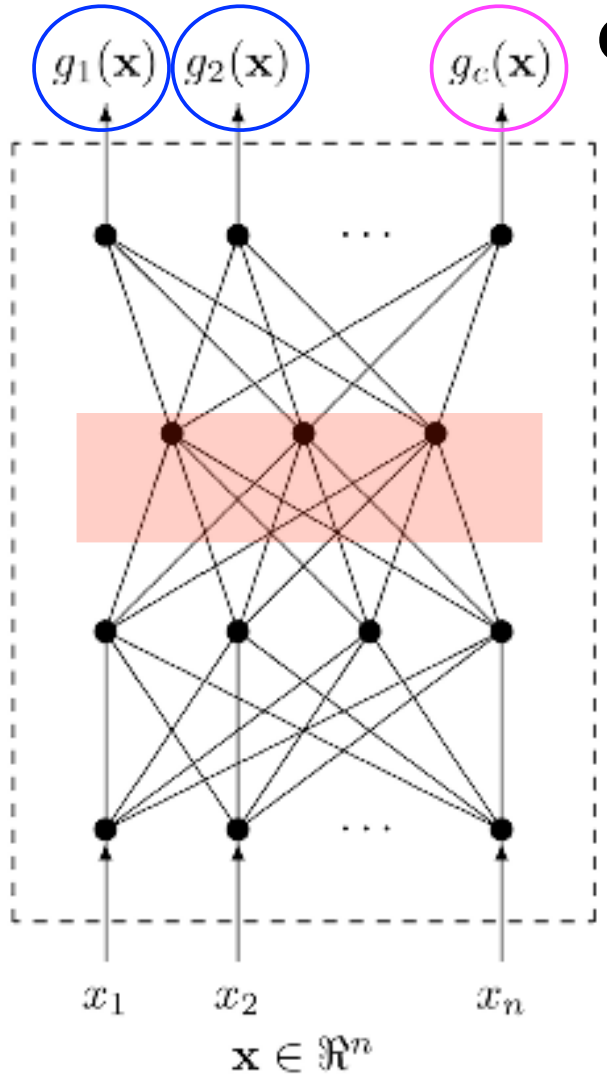
(activation: identity fn)

- Hidden layers have the same activation function (threshold or sigmoid)
- Classification is feed-forward: compute activations one layer at a time, input to ouput: decide ω_i for *max* $g_i(X)$
- Learning is through backpropagation: error metric used to update input weights based on observed activations (outputs)

R·I·T

Fig. 2.17 *A generic model of an MLP classifier.*

Multi-Layer Perceptron



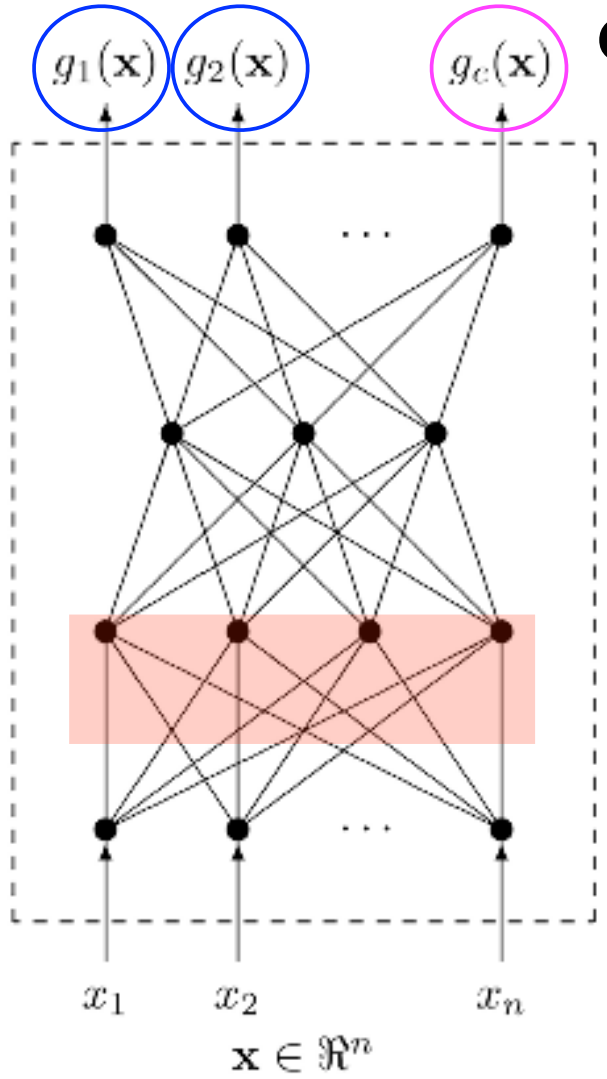
Correct Output:
(0,0,..., 1)

(activation: identity fn)

- Hidden layers have the same activation function (threshold or sigmoid)
- Classification is feed-forward: compute activations one layer at a time, input to ouput: decide ω_i for *max* $g_i(X)$
- Learning is through backpropagation: error metric used to update input weights based on observed activations (outputs)

Fig. 2.17 *A generic model of an MLP classifier.*

Multi-Layer Perceptron



Correct Output:
(0,0,..., 1)

(activation: identity fn)

- Hidden layers have the same activation function (threshold or sigmoid)
- Classification is feed-forward: compute activations one layer at a time, input to ouput: decide ω_i for *max* $g_i(X)$
- Learning is through backpropagation: error metric used to update input weights based on observed activations (outputs)

Fig. 2.17 *A generic model of an MLP classifier.*

MLP Properties

Approximating Classification Regions

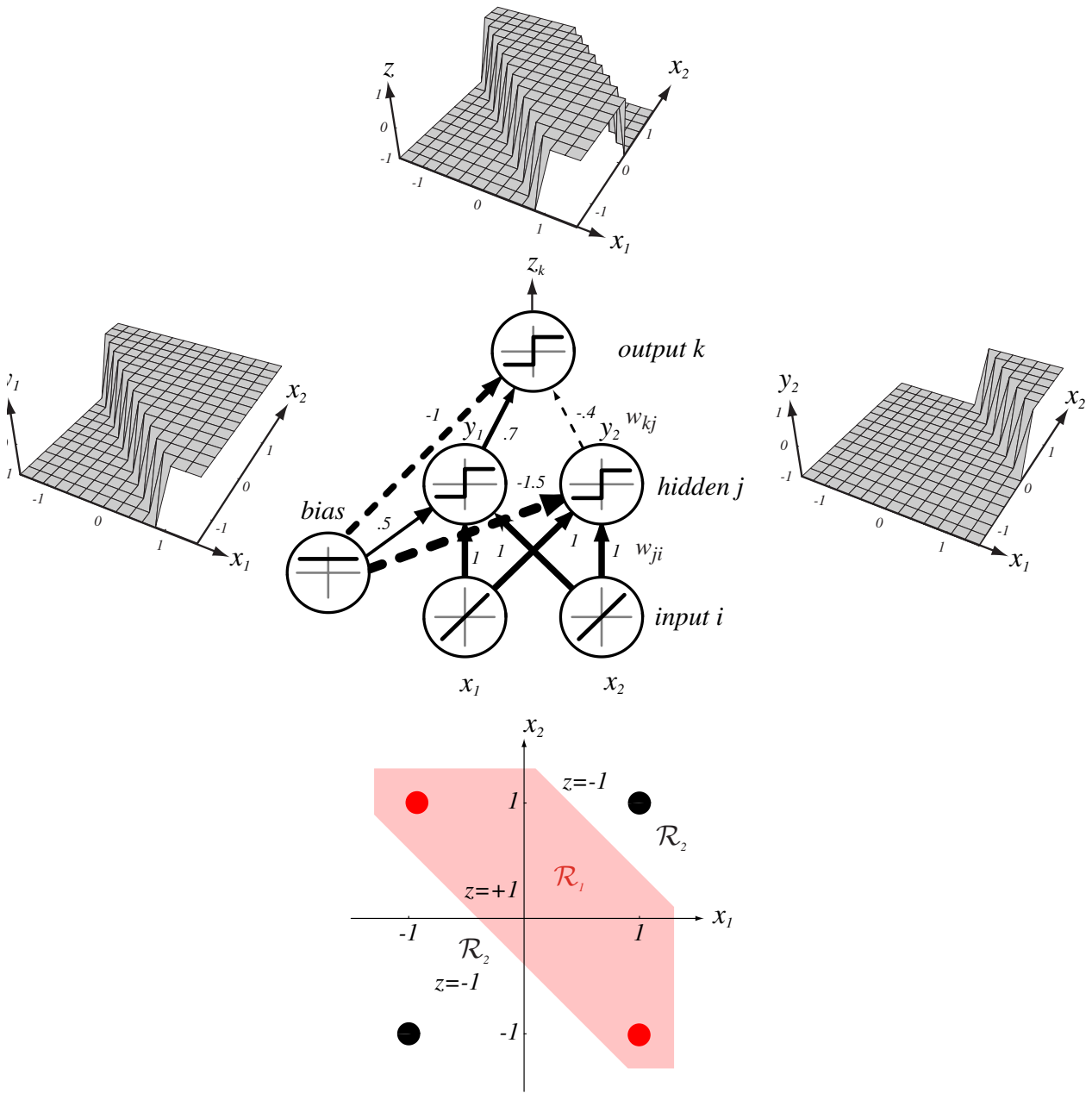
MLP shown in previous slide with *threshold* nodes can approximate any classification regions in R^n to a specified precision

Approximating *Any Function*

Later found that an MLP with one hidden layer and threshold nodes can approximate *any* function with a specified precision

In Practice...

These results tell us what is possible, but not how to achieve it (still uncertainties regarding appropriate network structure and training algorithm)


z
x2
x1
zk
output k
bias
.5
-1
y1
.7
-.4
y2
wkj
-1.5
hidden j
1
1
1
1
wji
input i
x1
x2
y1
y2
z=-1
z=+1
R1
R2
R2
z=-1

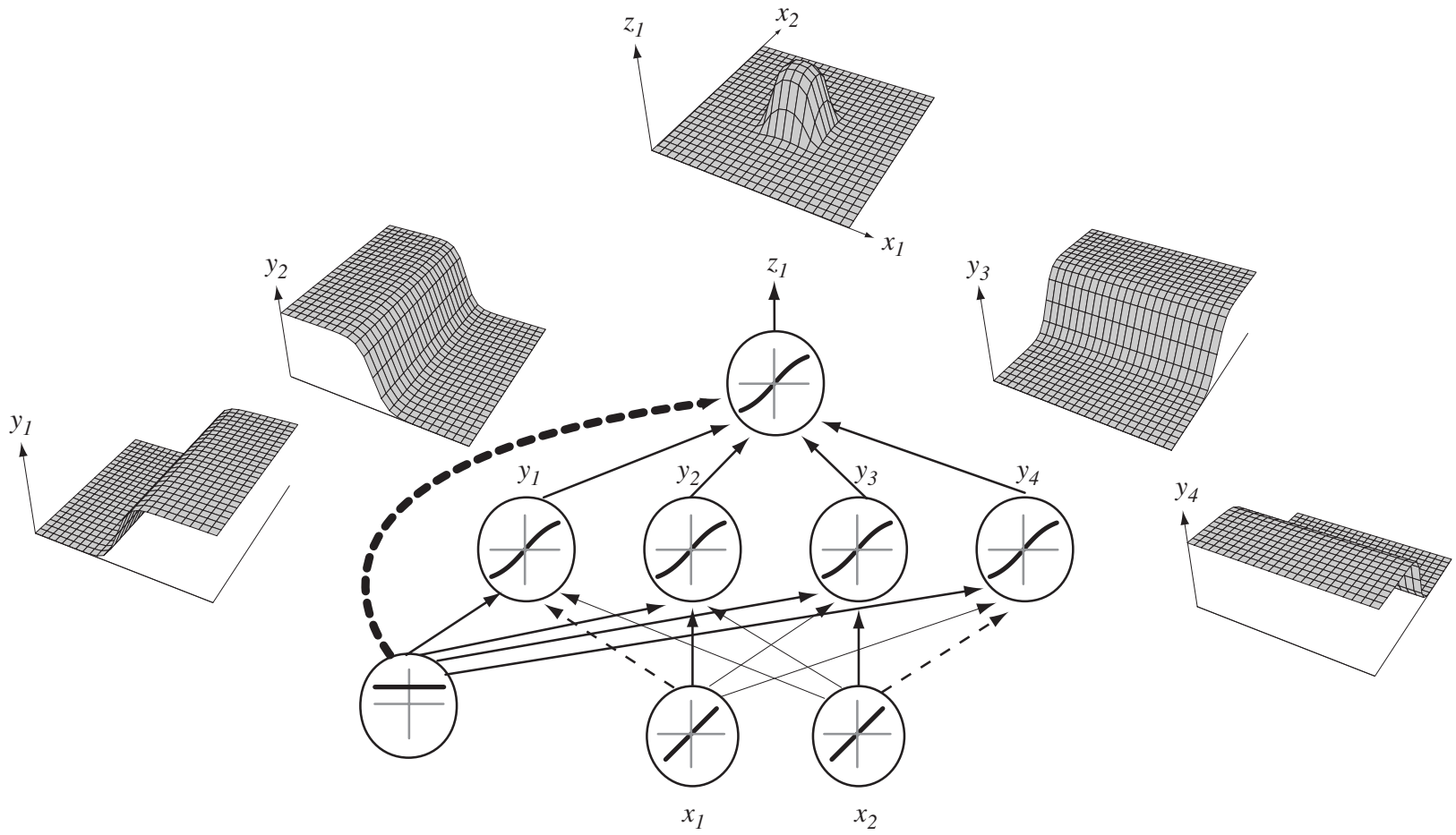


FIGURE 6.2. A 2-4-1 network (with bias) along with the response functions at different units; each hidden output unit has sigmoidal activation function $f(\cdot)$. In the case shown, the hidden unit outputs are paired in opposition thereby producing a "bump" at the output unit. Given a sufficiently large number of hidden units, any continuous function from input to output can be approximated arbitrarily well by such a network. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

(Threshold nodes)

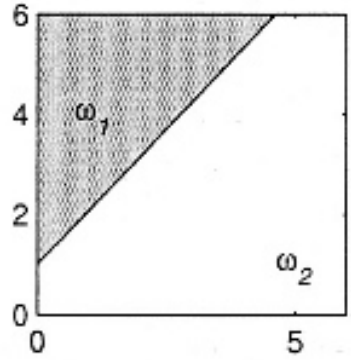
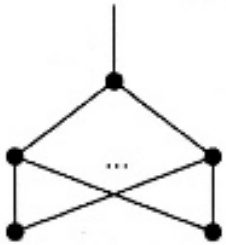
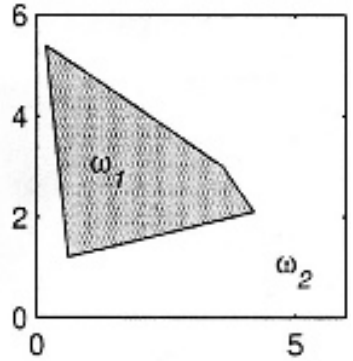
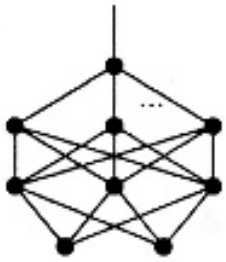
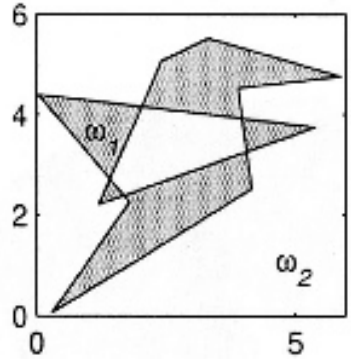
NN configuration	*Type of region*	*An example*
	Half space bounded by a hyperplane	
	Convex regions (open or closed)	
	Any regions	

Fig. 2.18 *Possible classification regions for an MLP with one, two, and three layers of threshold nodes.* (*Note that the "NN configuration" column only indicates the number of hidden layers and not the number of nodes needed to produce the regions in column "An example".*)

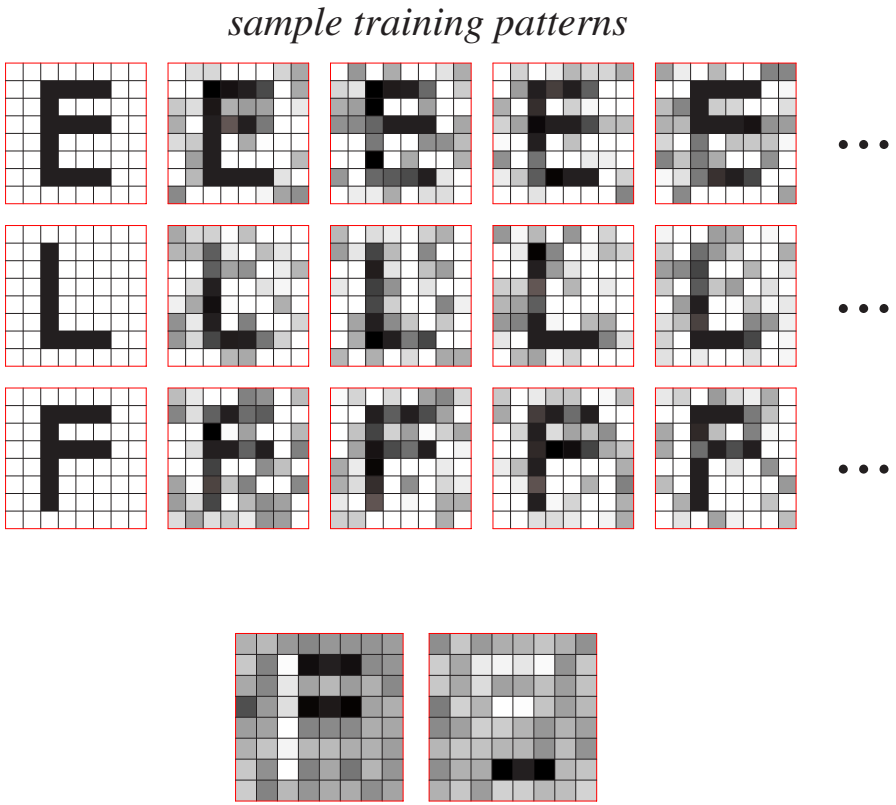


FIGURE 6.13. The top images represent patterns from a large training set used to train a 64-2-3 sigmoidal network for classifying three characters. The bottom figures show the input-to-hidden weights, represented as patterns, at the two hidden units after training. Note that these learned weights indeed describe feature groupings useful for the classification task. In large networks, such patterns of learned weights may be difficult to interpret in this way. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*.

Backpropagation Update Rules

Modify Perceptron Update Rule

Update weights using 1) partial derivative of error for node output w.r.t. net sum ξ, 2) output of node *k* from the preceding layer, 3) η (*eta*, learning rate)

Output Layer (depth/level 0)

$$w_{ik}^{o} \leftarrow w_{ik}^{o} - \eta \delta_i^o v_k^h, \qquad k = 0, \ldots, M, \qquad i = 1, \ldots, c$$

Hidden Layers (depth 'h')

$$w_{ik}^{h} \leftarrow w_{ik}^{h} - \eta \delta_i^h v_k^{h-1}, \qquad k = 0, \ldots, S, \quad i = 1, \ldots, M$$

Computing the Error

Derivative of Error at Output w.r.t w_j

$$\frac{\partial E}{\partial w_j} = \frac{\partial E}{\partial \xi}\frac{\partial \xi}{\partial w_j} = \boxed{\frac{\partial E}{\partial \xi}} u_j$$

input value j (node output from previous layer)

Derivative of Error w.r.t. *Output* Net Sum

$$\boxed{\frac{\partial E}{\partial g_i(\mathbf{x})}} = g_i(\mathbf{x}) - \mathcal{I}(l(\mathbf{x}), \omega_i)$$

$$\boxed{\delta_i^o} = \frac{\partial E}{\partial \xi_i^o} = \boxed{\frac{\partial E}{\partial g_i(\mathbf{x})}}\frac{\partial g_i(\mathbf{x})}{\partial \xi_i^o} = [g_i(\mathbf{x}) - \mathcal{I}(l(\mathbf{x}), \omega_i)]\boxed{\frac{\partial \phi(\xi_i^o)}{\partial \xi_i^o}}$$

(chain rule)

For Sigmoid Activation Function:

$$\boxed{\delta_i^o} = \frac{\partial E}{\partial \xi_i^o} = [g_i(\mathbf{x}) - \mathcal{I}(\mathbf{x}, \omega_i)]\boxed{g_i(\mathbf{x})[1 - g_i(\mathbf{x})]}$$

(2.91): Output Node Error $\delta_i^o = \frac{\partial E}{\partial \xi_i^o} = [g_i(\mathbf{x}) - \mathcal{I}(l(\mathbf{x}), \omega_i)]\, g_i(\mathbf{x})[1 - g_i(\mathbf{x})]$

(2.96): Hidden Node Error

$$\delta_k^h = \frac{\partial E}{\partial \xi_k^h} = \left(\sum_{i=1}^{c} \delta_i^o w_{ik}^o \right) v_k^h (1 - v_k^h)$$

(2.77) (Squared Error):

$$E = \frac{1}{2} \sum_{j=1}^{N} \sum_{i=1}^{c} \{ g_i(\mathbf{z}_j) - \mathcal{I}(\omega_i, l(\mathbf{z}_j)) \}^2$$

Stopping Criterion:

Error less than epsilon OR
Exceed max # epochs, T

Output/Hidden Activation:

Sigmoid function

**Online training (vs. batch or stochastic)

Backpropagation MLP training

1. Choose an MLP structure: pick the number of hidden layers, the number of nodes at each layer and the activation functions.
2. Initialize the training procedure: pick small random values for all weights (including biases) of the NN. Pick the learning rate $\eta > 0$, the maximal number of epochs T and the error goal $\epsilon > 0$.
3. Set $E = \infty$, the epoch counter $t = 1$ and the object counter $j = 1$.
4. While ($E > \epsilon$ and $t \leq T$) do
   (a) Submit $\mathbf{z}_j$ as the next training example.
   (b) Calculate the output of every node of the NN with the current weights (forward propagation).
   (c) Calculate the error term δ at each node at the output layer by (2.91).
   (d) Calculate recursively all error terms at the nodes of the hidden layers using (2.95) (backward propagation).
   (e) For each hidden and each output node update the weights by

$$w_{new} = w_{old} - \eta \delta u, \qquad (2.98)$$

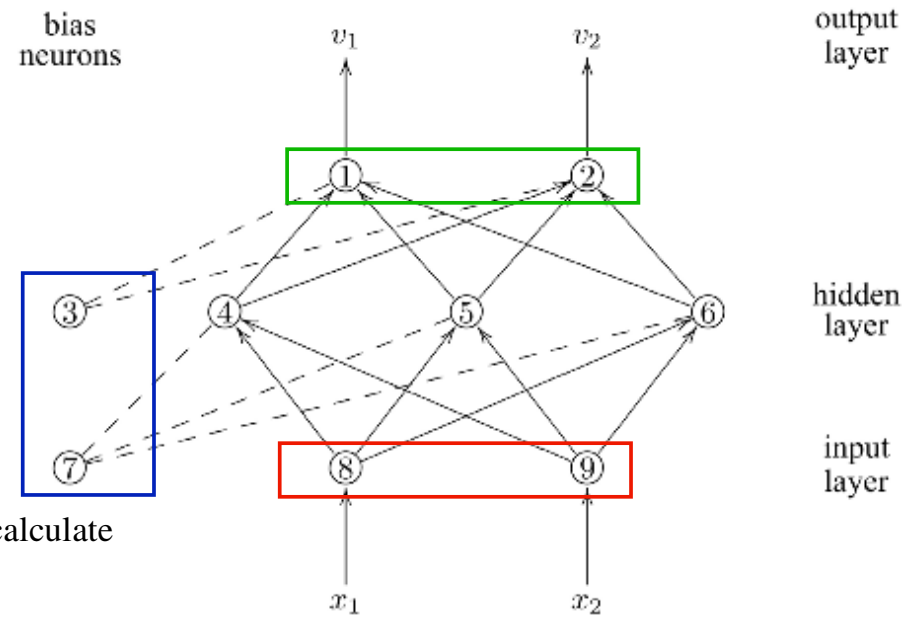
   using the respective δ and u.
   (f) Calculate E using the current weights and Eq. (2.77).
   (g) If $j = N$ (a whole pass through $\mathbf{Z}$ (epoch) is completed), then set $t = t + 1$ and $j = 0$. Else, set $j = j + 1$.
5. End % (While)

Input: (2,-1) (class 2)

$v_1 = 0.8488 \quad v_2 = 0.9267$

$v_3 = 1 \quad v_4 = 0.7738 \quad v_5 = 0.8235 \quad v_6 = 0.9038$

$v_7 = 1 \quad v_8 = 2 \quad v_9 = -1$



The target value (class ω_2) is [0, 1]. Using Eq. (2.91), we have

$$\delta_1 = (0.8488 - 0) \times 0.8488 \times (1 - 0.8488) = 0.1089$$

$$\delta_2 = (0.9267 - 1) \times 0.9267 \times (1 - 0.9267) = -0.0050$$

(sigmoid deriv.)

Propagating the error to the hidden layer as in Eq. (2.96), we calculate

$$\begin{aligned}\delta_4 &= (\delta_1 \times w_{41} + \delta_2 \times w_{42}) \times v_4 \times (1 - v_4) \\ &= (0.1089 \times 0.05 - 0.0050 \times 0.57) \times 0.7738 \times (1 - 0.7738) \\ &\approx 0.0005 \end{aligned} \quad (2.101)$$

Update Weights

In the same way we obtain $\delta_5 = 0.0104$ and $\delta_6 = 0.0068$. We can now calculate he new values of all the weights through Eq. (2.98). For example,

$$\begin{aligned} w_{42} &= w_{42} - \eta \times \delta_2 \times v_4 \\ &= 0.57 - 0.1 \times (-0.0050) \times 0.7738 \quad (2.102) \\ &= 0.5704 \quad (2.103) \end{aligned}$$

For input-to-hidden layer weights we use again Eq. (2.98); for example,

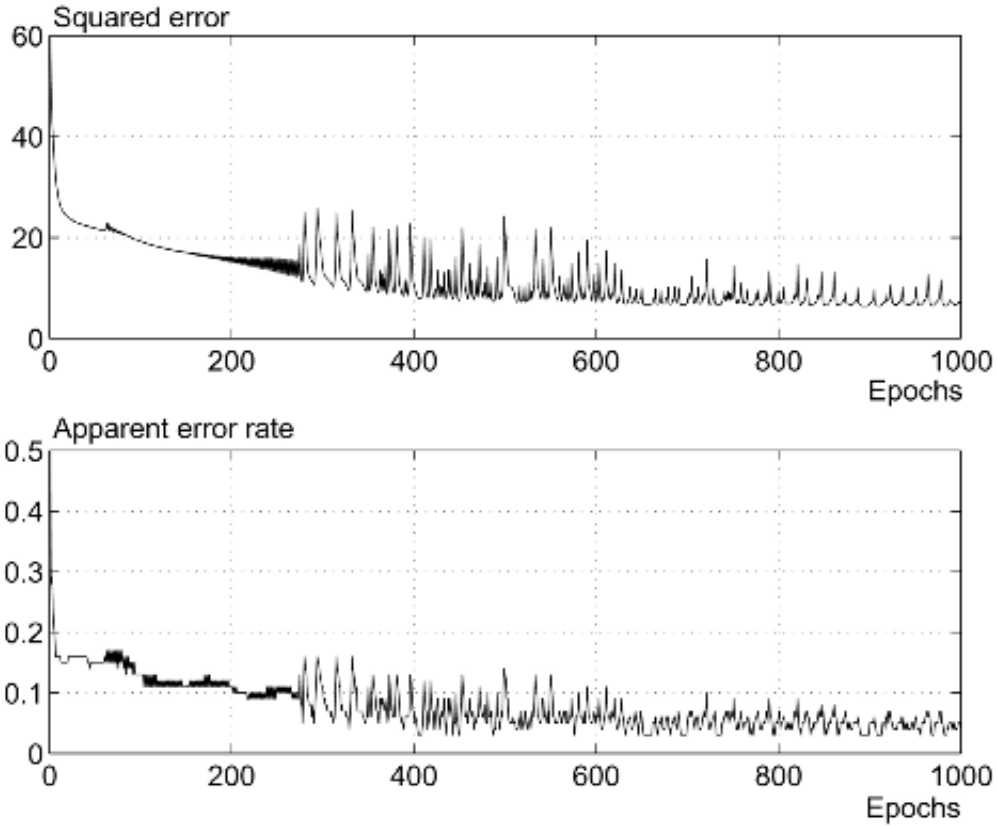
$$w_{95} = w_{95} - \eta \times \delta_5 \times v_9 = w_{95} - \eta \times \delta_5 \times x_2 \quad (2.104)$$

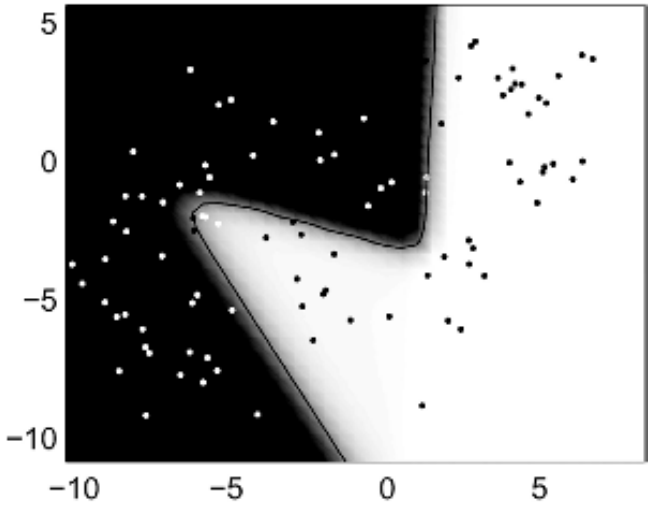
$$= 0.06 - 0.1 \times 0.0104 \times (-1) = 0.0610 \quad (2.105)$$

nodes 3, 7 are bias nodes: always output 1

activation:
input: identity
hidden/output: sigmoid

2:3:2 MLP (see previous slide)
Batch training (updates at end of epoch)
Max Epochs: 1000, η = 0.1, error goal: 0
Initial weights: random, in [0,1]





Final train error: 4%
Final test error: 9%

Fig. 2.21 *Squared error and the apparent error rate versus the number of epochs for the backpropagation training of a 2 : 3 : 2 MLP on the banana data.*

Final Note

Backpropogation Algorithms

Are numerous: many designed for faster convergence, increased stability, etc.

Other Network Types

Many, many variations: RBF, Graph Transformer Networks (GTN), Convolutional Networks

- e.g. LeNet5 example (digit recognition): http://yann.lecun.com/exdb/lenet/