Neural Networks

(Reading: Kuncheva Section 2.5)

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)

R·I·T

Introduction, Continued

Black-Box View of a Neural Net

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 class with maximum discriminant value
- Regression: learn continuous outputs directly (e.g. learn to fit the sin function - see Bishop text)

Training (for Classification)

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

$$E = \frac{1}{2} \sum_{j=1}^{N} \sum_{i=1}^{c} \{ g_i(\mathbf{z}_j) - \mathcal{I}(\omega_i, l(\mathbf{z}_j)) \}^2 \tag{2.77}$$

Introduction, Continued

Granular Representation

A set of interacting elements ('neurons' or nodes) map input values to output values using a 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 set of discriminant functions learned by a network approach the true posterior probabilities for each class (for multi-layer perceptrons (MLP), and radial basis function (RBF) networks):

$$\lim_{N \to \infty} g_i(\mathbf{x}) = P(\omega_i|\mathbf{x}), \qquad \mathbf{x} \in \Re^n \tag{2.78}$$

Note:

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

R·I·T

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*.

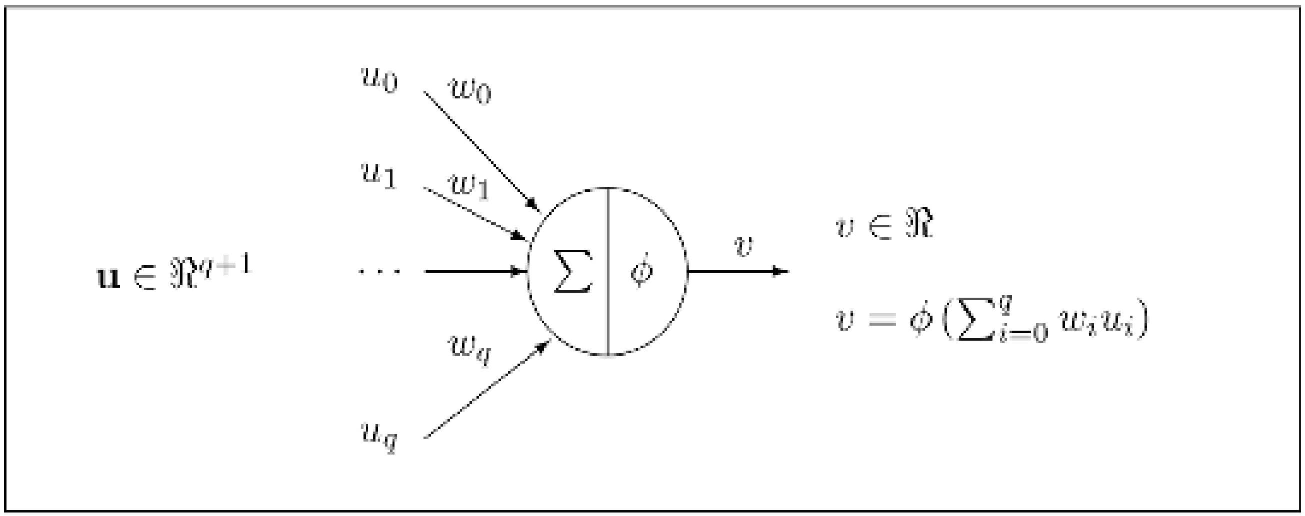


Fig. 2.14 *The NN processing unit.*

Common Activation Functions

ξ : (net sum)

- The threshold function

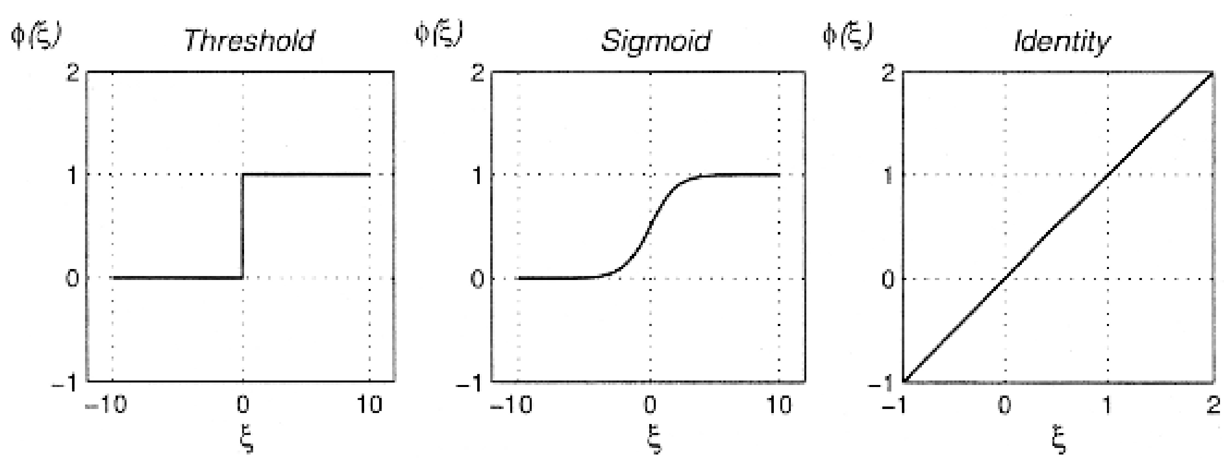
$$\phi(\xi) = \begin{cases} 1, & \text{if } \xi \geq 0, \\ 0, & \text{otherwise.} \end{cases}$$

- The sigmoid function

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

- The identity function

$$\phi(\xi) = \xi$$ *(used for input nodes)*



R·I·T

Bias: Offset for Activation Functions

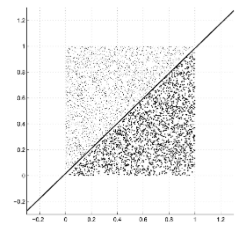
The weight "$-w_0$" is used as a *bias*, and the corresponding input value u_0 is set to 1. Equation (2.79) can be rewritten as

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

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.



R·I·T

The Perception (Rosenblatt, 1962)

Rosenblatt [8] defined the so called *perceptron* and its famous training algorithm. The perceptron is implemented as Eq. (2.79) with a threshold activation function

$$\phi(\xi) = \begin{cases} 1, & \text{if } \xi \geq 0, \\ -1, & \text{otherwise.} \end{cases} \qquad (2.85)$$

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

Update Rule:

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

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) until no errors are made. 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

Perceptron Learning

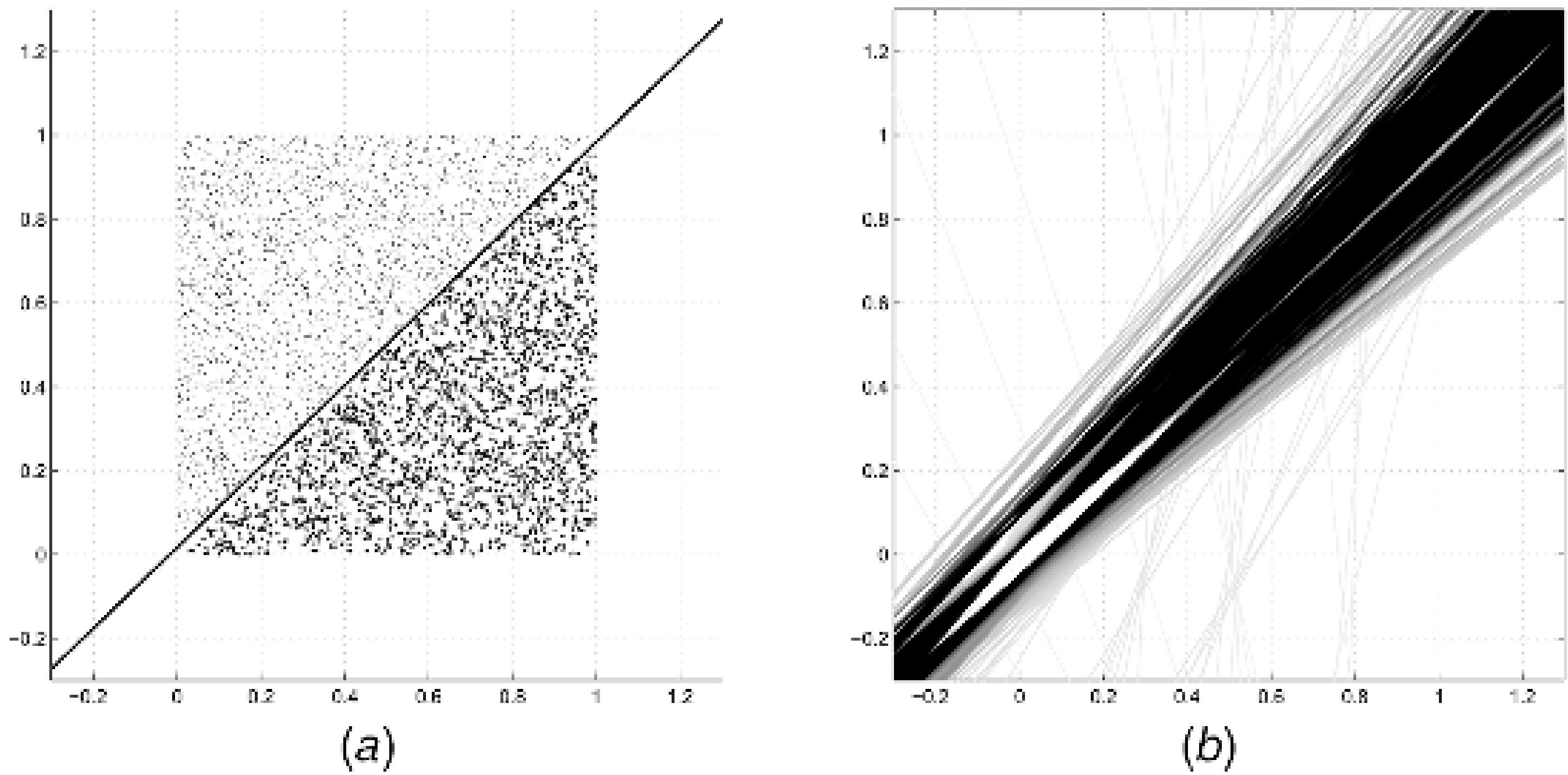


Fig. 2.16 *(a) Uniformly distributed two-class data and the boundary found by the perceptron training algorithm. (b) The "evolution" of the class boundary.*

Multi-Layer Perceptron

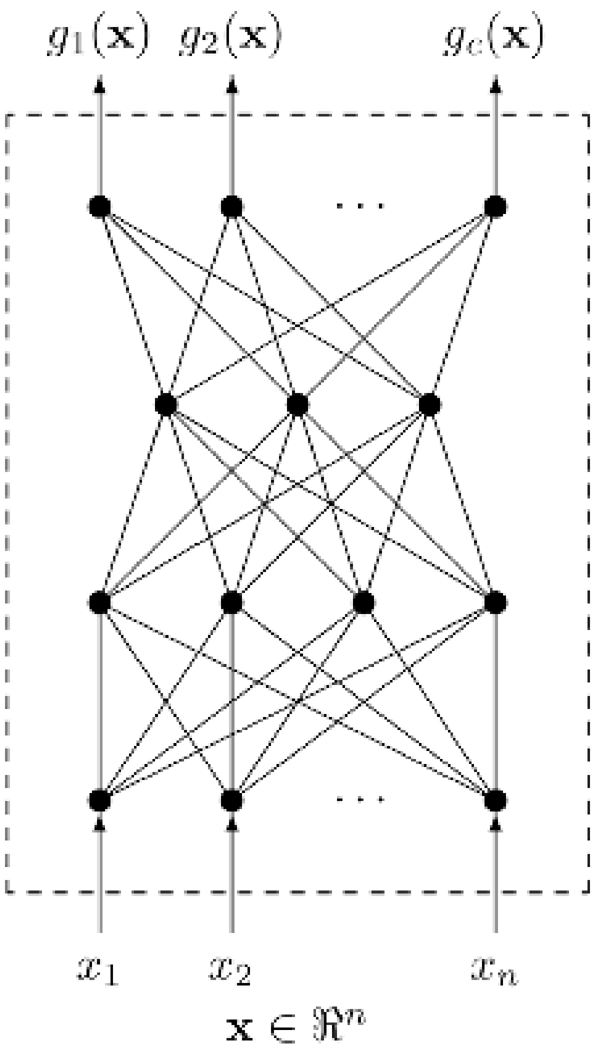


Fig. 2.17 A generic model of an MLP classifier.

- Nodes: perceptrons
- Hidden, output 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 (update input weights from output to input layer)

Multi-Layer Perceptron

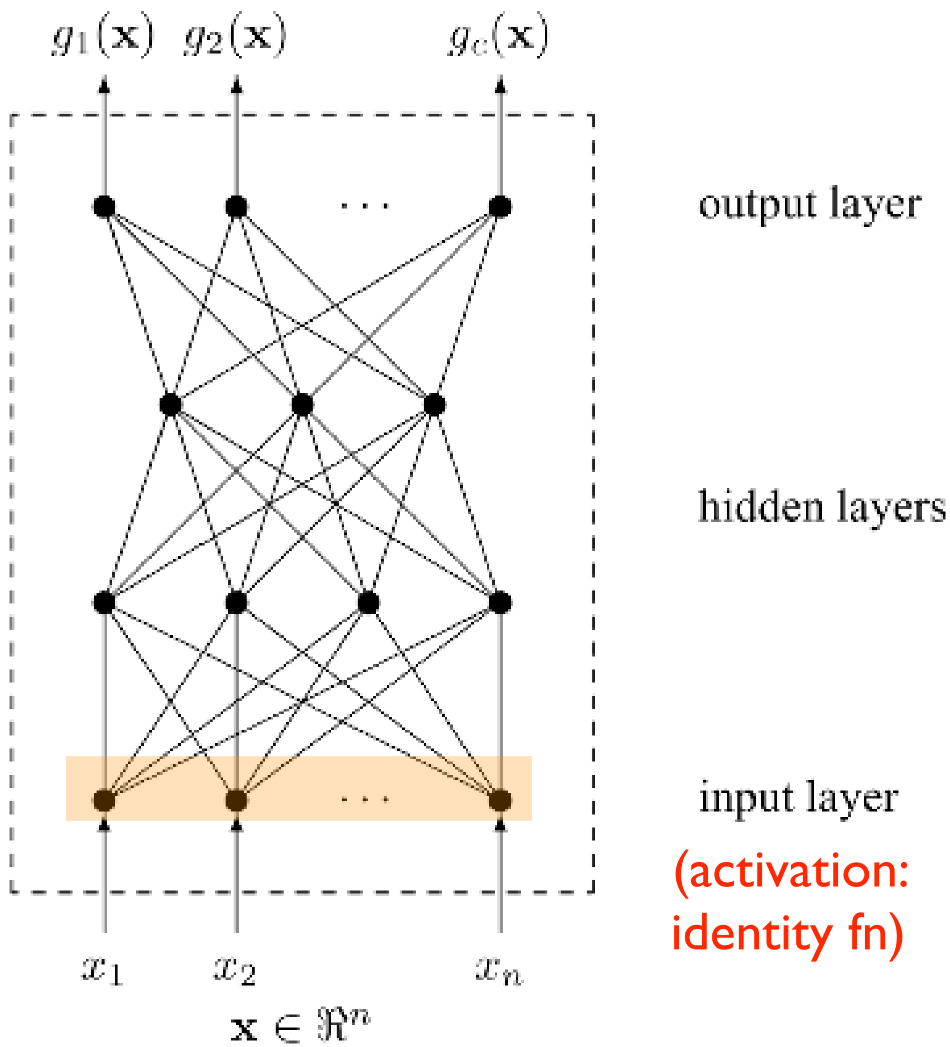


Fig. 2.17 *A generic model of an MLP classifier.*

- Nodes: perceptrons
- Hidden, output 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 (update input weights from output to input layer)

Multi-Layer Perceptron

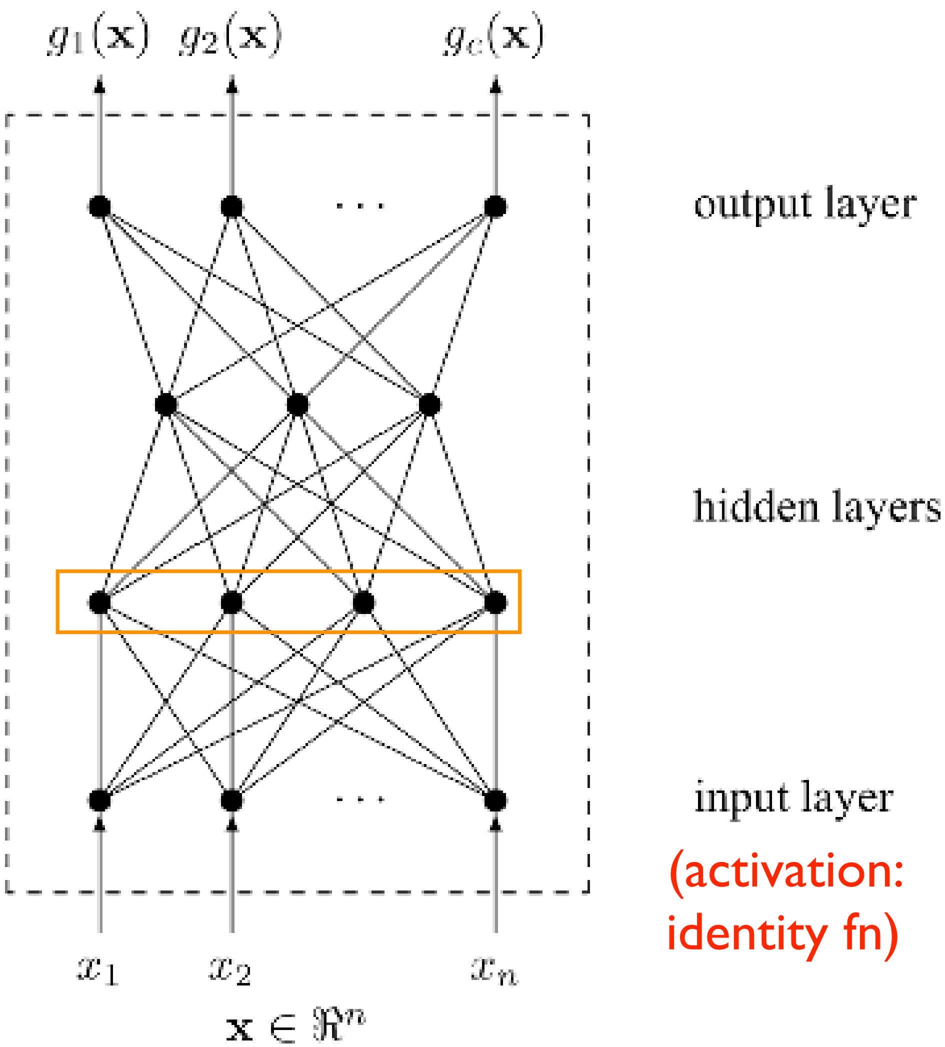


Fig. 2.17 *A generic model of an MLP classifier.*

- Nodes: perceptrons
- Hidden, output 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 (update input weights from output to input layer)

R·I·T

Multi-Layer Perceptron

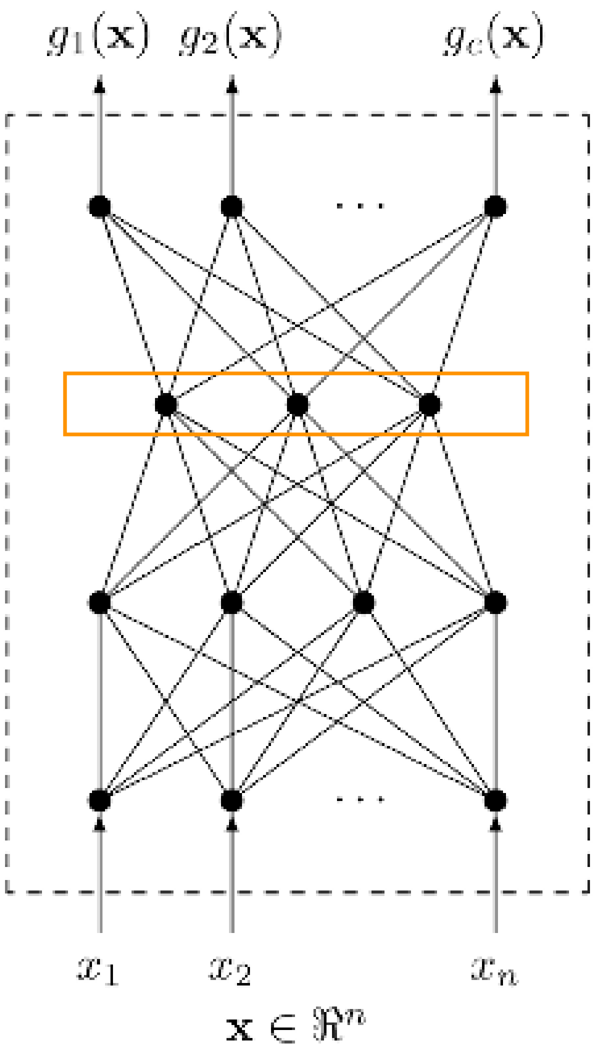


Fig. 2.17 A generic model of an MLP classifier.

- Nodes: perceptrons
- Hidden, output 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 (update input weights from output to input layer)

Multi-Layer Perceptron

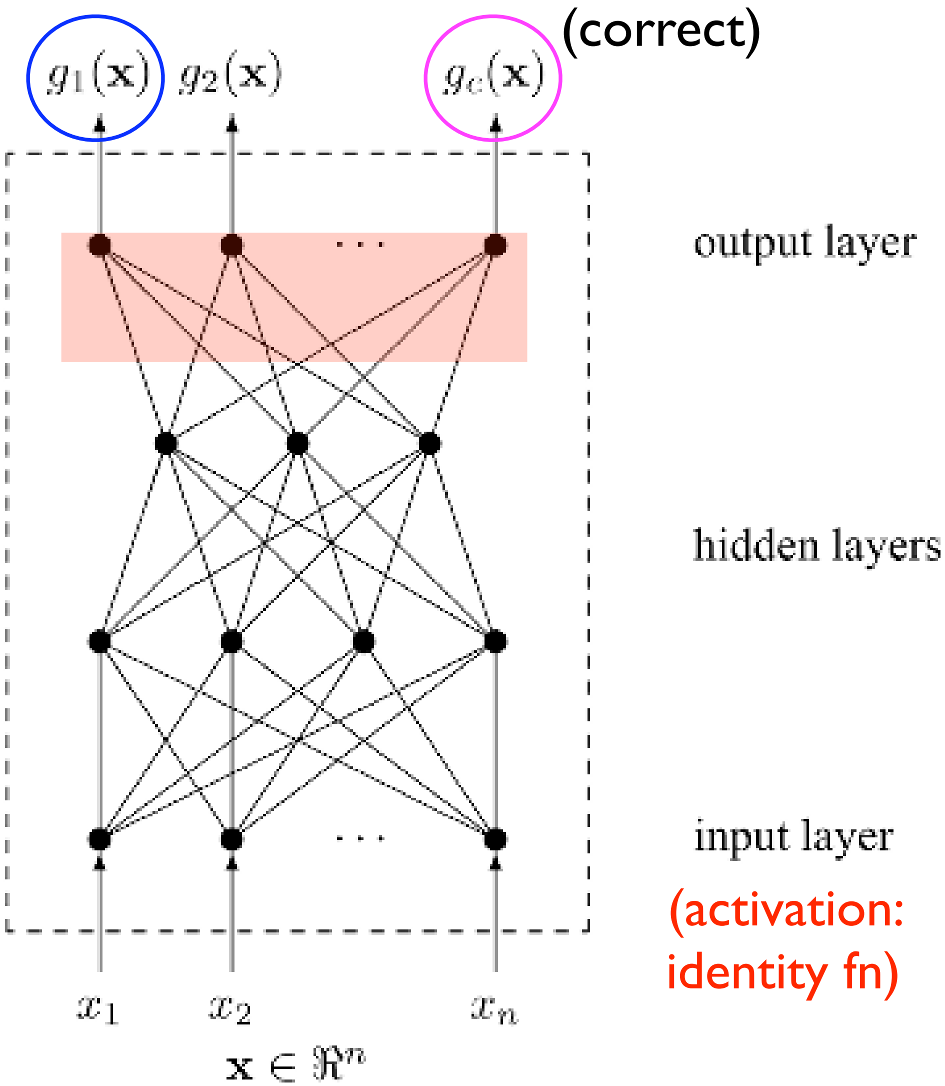


Fig. 2.17 *A generic model of an MLP classifier.*

- Nodes: perceptrons
- Hidden, output 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 (update input weights from output to input layer)

R·I·T

Multi-Layer Perceptron

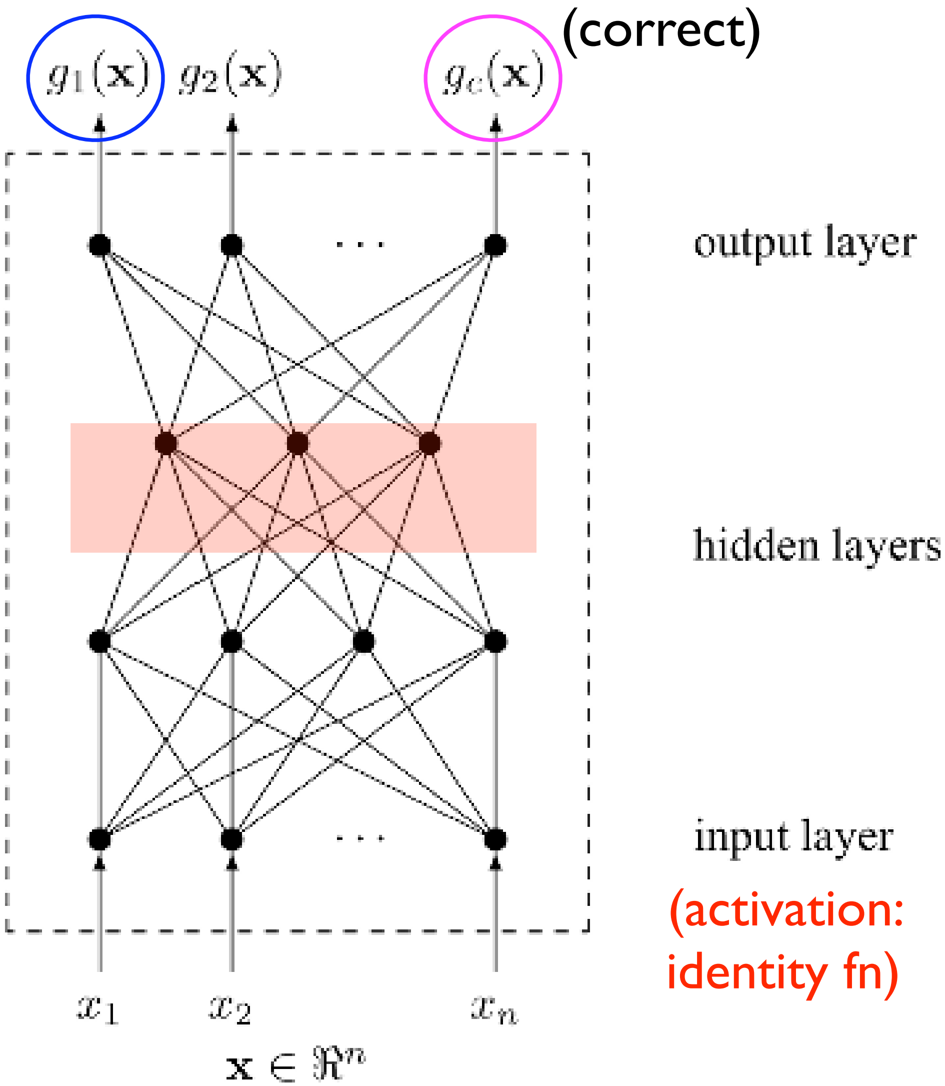


Fig. 2.17 A generic model of an MLP classifier.

- Nodes: perceptrons
- Hidden, output 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 (update input weights from output to input layer)

Multi-Layer Perceptron

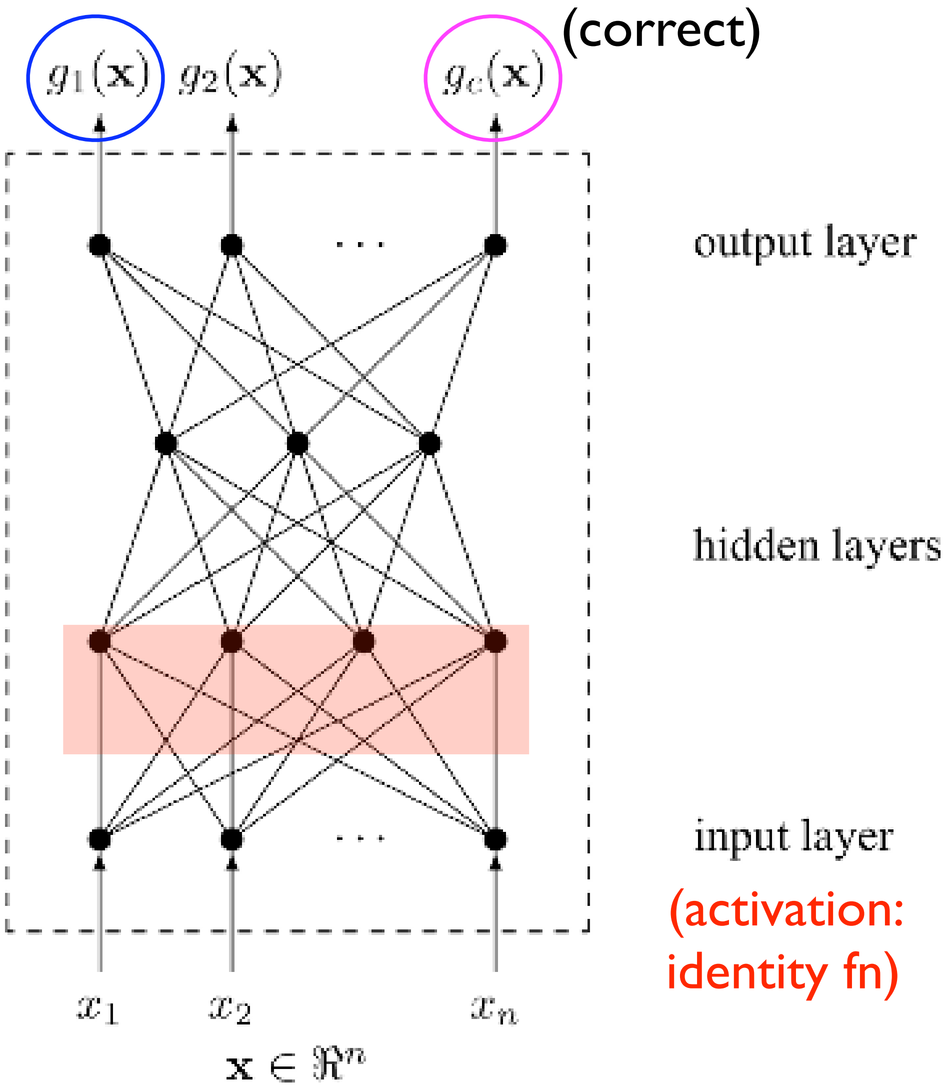


Fig. 2.17 *A generic model of an MLP classifier.*

- Nodes: perceptrons
- Hidden, output 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 (update input weights from output to input layer)

Multi-Layer Perceptron

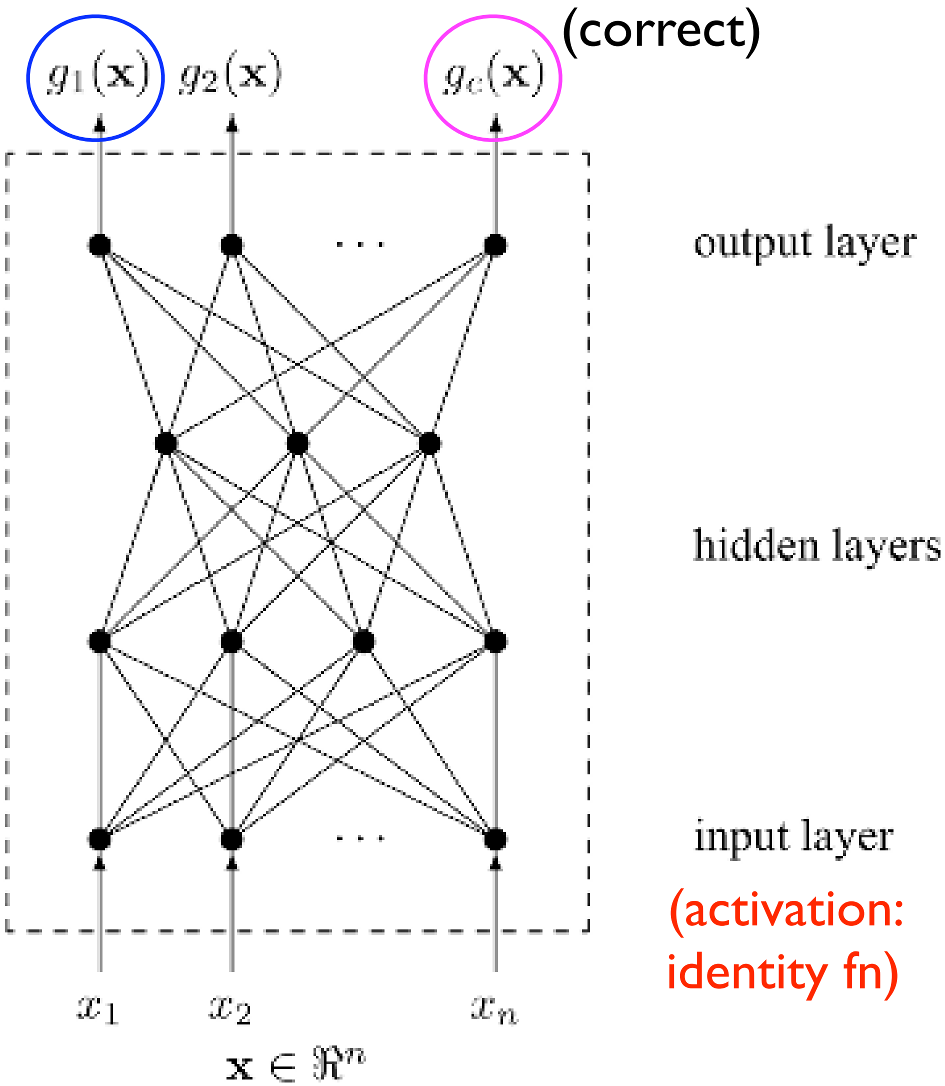


Fig. 2.17 *A generic model of an MLP classifier.*

- Nodes: perceptrons
- Hidden, output 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 (update input weights from output to input layer)

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 (network structure and training algorithms)

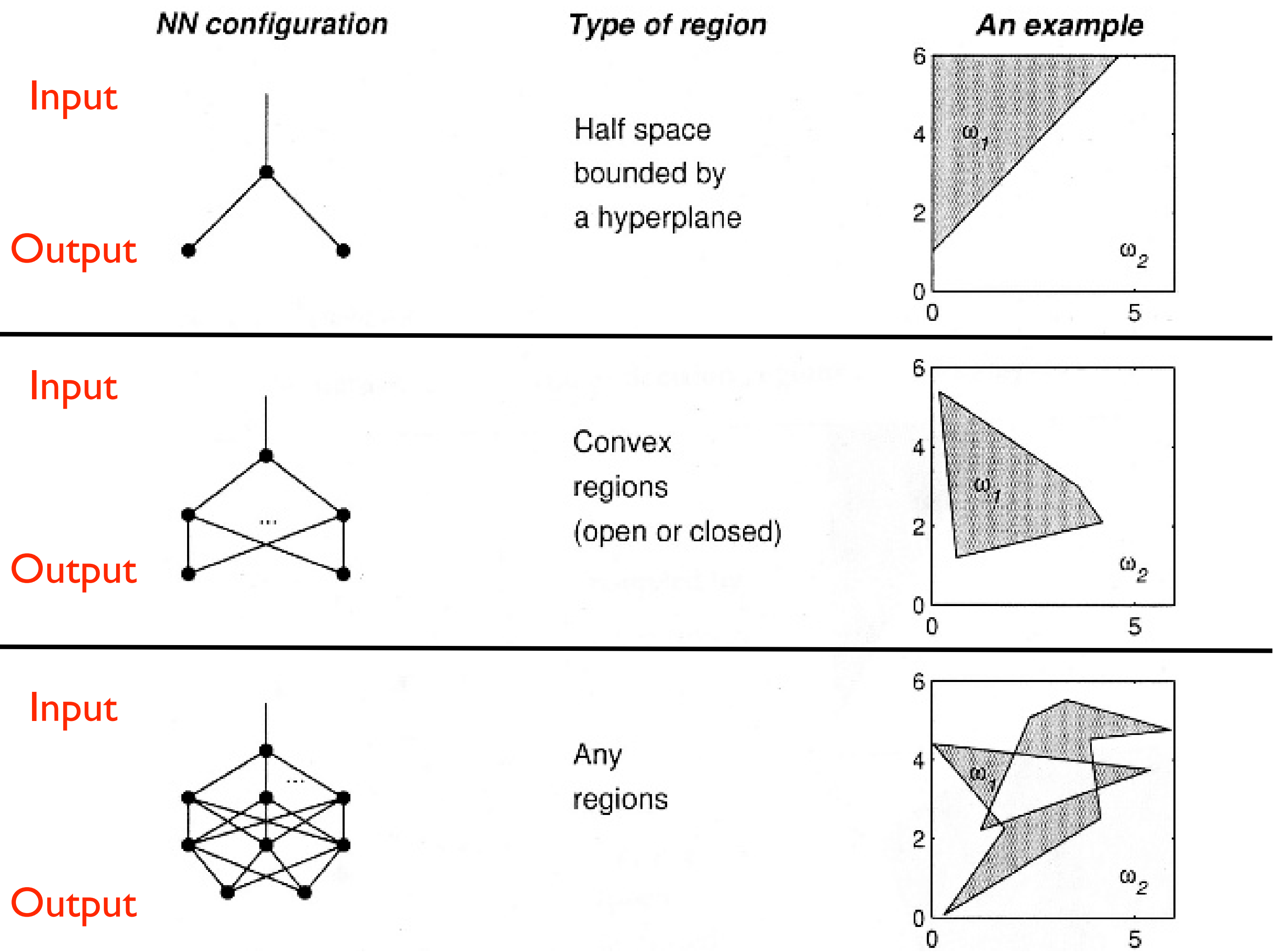


Fig. 2.18 Possible classification regions for an MLP with one, two, and three layers of threshold nodes. (Note that the “NN configuration” column only indicates the number of hidden layers and not the number of nodes needed to produce the regions in column “An example”.)

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

(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

$$w_{new} = w_{old} - \eta\delta u, \tag{2.98}$$

   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)

Fig. 2.19 Backpropagation MLP training.

nodes 3, 7 are bias nodes: always output 1

activation:
input: identity
hidden/output: sigmoid

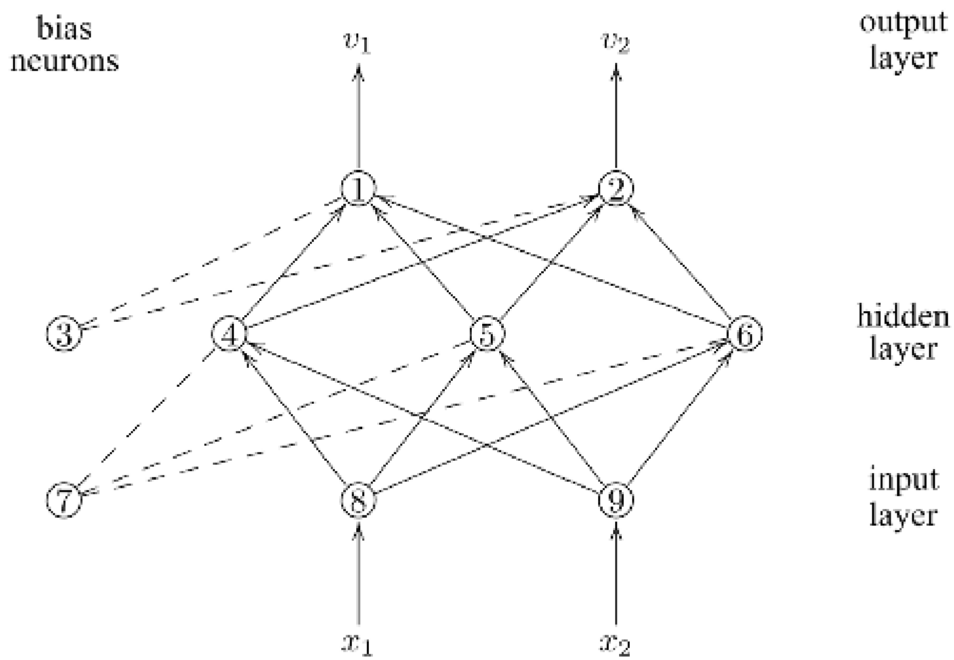


Fig. 2.20 *A 2 : 3 : 2 MLP configuration. Bias nodes are depicted outside the layers and are not counted as separate nodes.*

TABLE 2.5 (a) Random Set of Weights for a 2 : 3 : 2 MLP NN; (b) Updated Weights Through Backpropagation for a Single Training Example.

Neuron		Incoming Weights			
(*a*)	1	$w_{31} = 0.4300$	$w_{41} = 0.0500$	$w_{51} = 0.7000$	$w_{61} = 0.7500$
	2	$w_{32} = 0.6300$	$w_{42} = 0.5700$	$w_{52} = 0.9600$	$w_{62} = 0.7400$
	4	$w_{74} = 0.5500$	$w_{84} = 0.8200$	$w_{94} = 0.9600$	
	5	$w_{75} = 0.2600$	$w_{85} = 0.6700$	$w_{95} = 0.0600$	
	6	$w_{76} = 0.6000$	$w_{86} = 1.0000$	$w_{96} = 0.3600$	
(*b*)	1	$w_{31} = 0.4191$	$w_{41} = 0.0416$	$w_{51} = 0.6910$	$w_{61} = 0.7402$
	2	$w_{32} = 0.6305$	$w_{42} = 0.5704$	$w_{52} = 0.9604$	$w_{62} = 0.7404$
	4	$w_{74} = 0.5500$	$w_{84} = 0.8199$	$w_{94} = 0.9600$	
	5	$w_{75} = 0.2590$	$w_{85} = 0.6679$	$w_{95} = 0.0610$	
	6	$w_{76} = 0.5993$	$w_{86} = 0.9986$	$w_{96} = 0.3607$	

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]

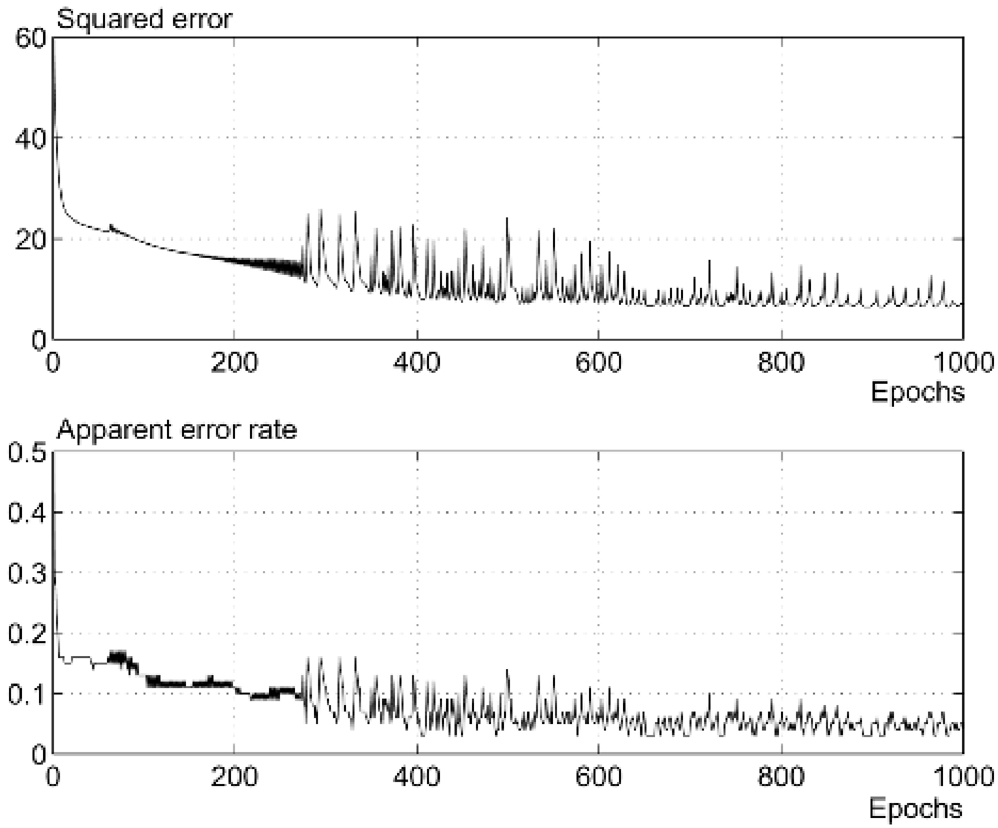
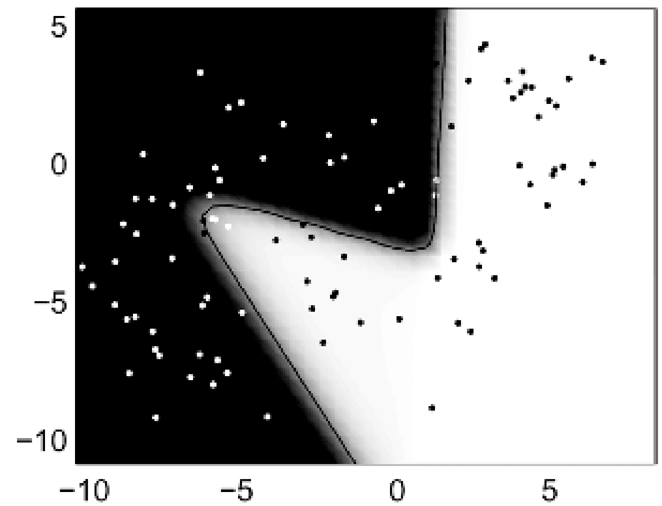


Fig. 2.21 *Squared error and the apparent error rate versus the number of epochs for the backpropagation training of a 2 : 3 : 2 MLP on the banana data.*



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

Final Note

Backpropogation Algorithms

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

R·I·T