

CS:4420 Artificial Intelligence

Spring 2019

Neural Networks

Cesare Tinelli

The University of Iowa

Copyright 2004–19, Cesare Tinelli and Stuart Russell ^a

^a These notes were originally developed by Stuart Russell and are used with permission. They are copyrighted material and may not be used in other course settings outside of the University of Iowa in their current or modified form without the express written consent of the copyright holders.

Readings

- Chap. 18 of [Russell and Norvig, 3rd Edition]

Brains as Computational Devices

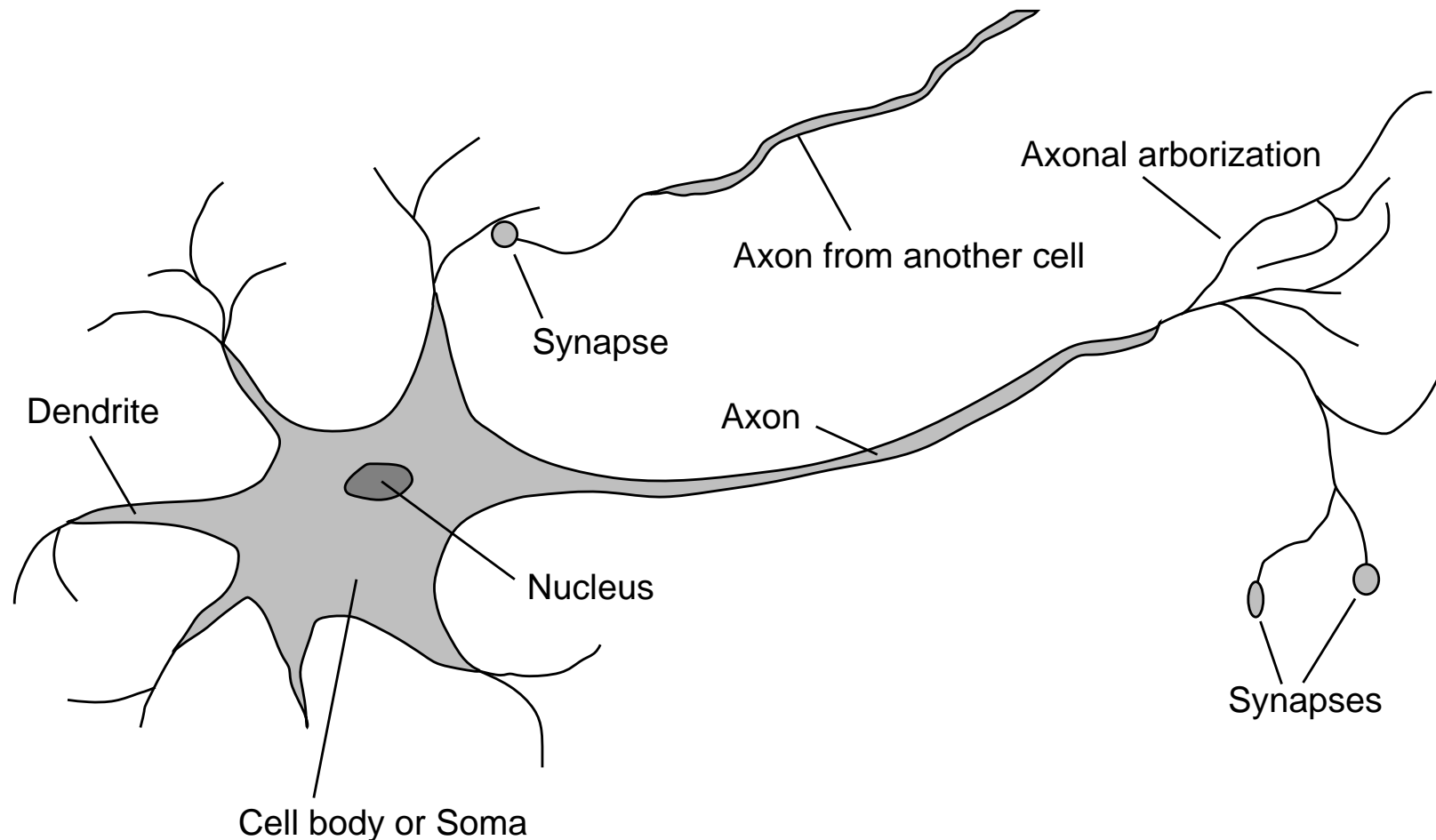
Brain's advantages with respect to digital computers:

- Massively parallel
- Fault-tolerant
- Reliable
- Graceful degradation

Brains and Neurons

10^{11} neurons of > 20 types, 10^{14} synapses, 1ms–10ms cycle time

Signals are noisy “spike trains” of electrical potential



Artificial Neural Network

Artificial neural networks are inspired by brains and neurons

A *neural network* is a graph with nodes, or *units*, connected by *links*

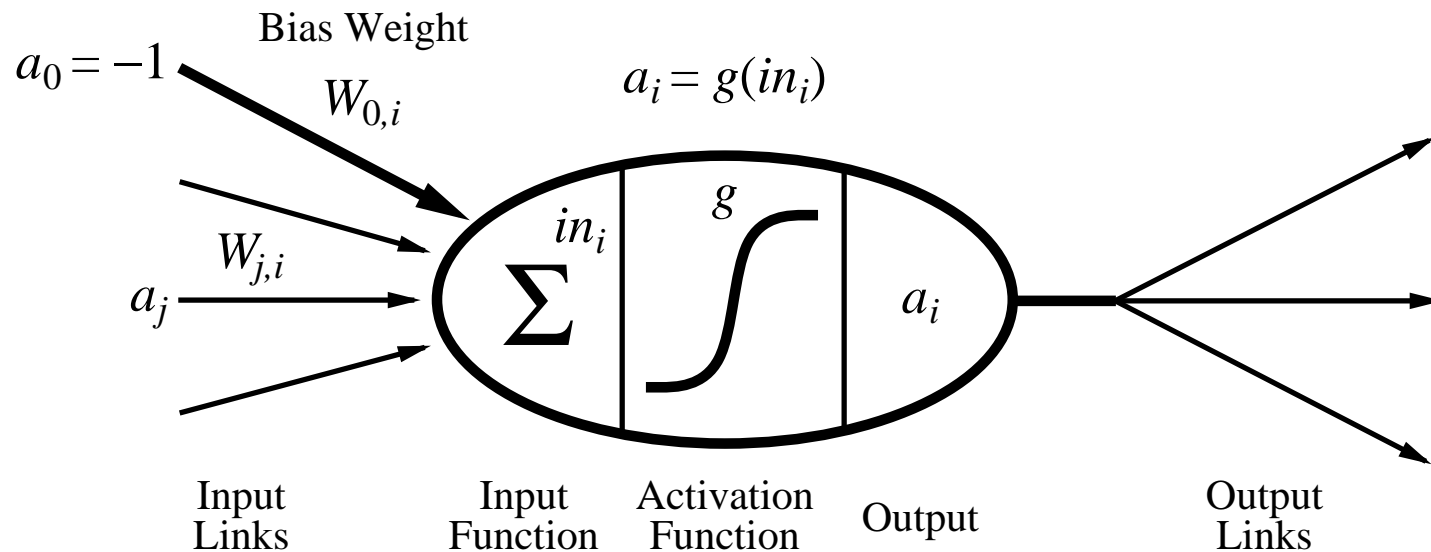
Each link has an associated *weight*, a real number

Typically, each node i outputs a real number, which is fed as input to the nodes connected to i

The output of a node is a function of the weighted sum of the node's inputs

A Neural Network Unit

McCulloch & Pitts model:

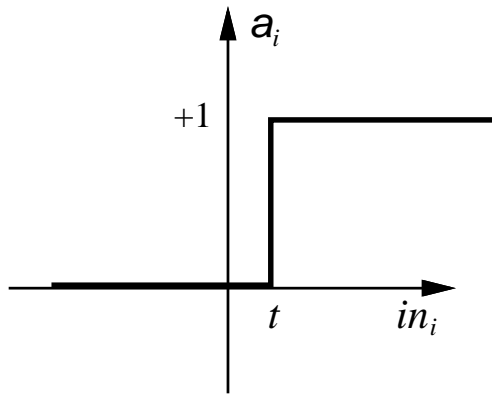


Output is a “squashed” linear function of the inputs:

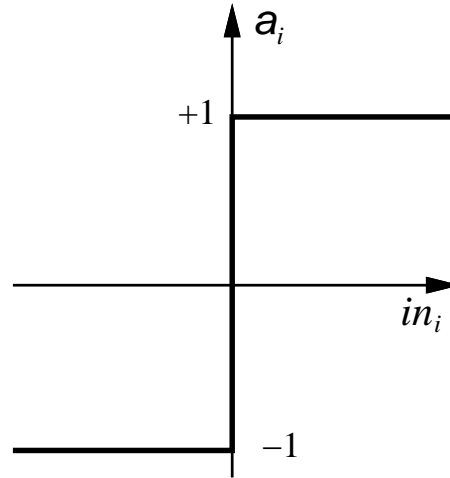
$$a_i \leftarrow g(in_i) = g\left(\sum_j W_{j,i} a_j\right)$$

This is a gross oversimplification of real neurons, but is meant to develop understanding of what networks of simple units can do

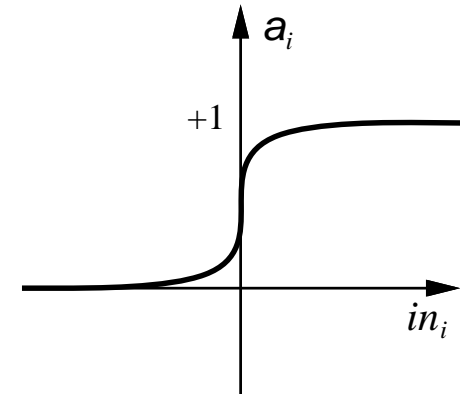
Possible Activation Functions



(a) Step function



(b) Sign function



(c) Sigmoid function

$$\text{step}_t(x) = \begin{cases} 1, & \text{if } x \geq t \\ 0, & \text{if } x < t \end{cases} \quad \text{sign}(x) = \begin{cases} +1, & \text{if } x \geq 0 \\ -1, & \text{if } x < 0 \end{cases}$$

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

Normalizing Unit Thresholds.

If t is the threshold value of the output unit, then

$$\text{step}_t\left(\sum_{j=1}^n W_j I_j\right) = \text{step}_0\left(\sum_{j=0}^n W_j I_j\right)$$

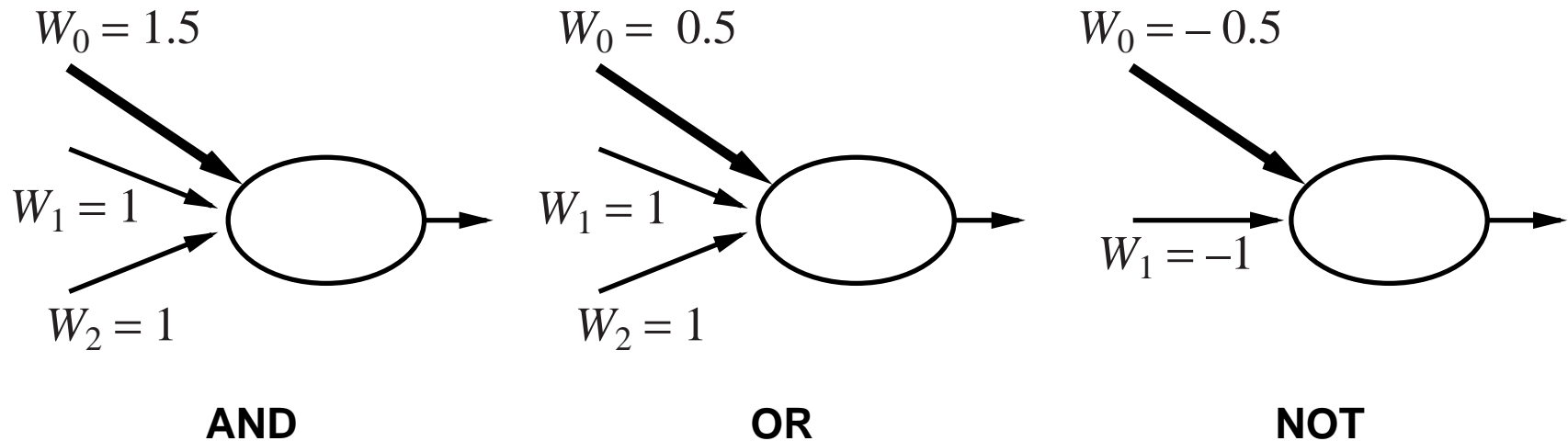
where $W_0 = t$ and $I_0 = -1$

So we can always assume that the unit's threshold is 0

This allows thresholds to be learned like any other weight

Then, we can even allow output values in $[0, 1]$ by replacing step_0 by the sigmoid function

Units as Logic Gates



Activation function: step function

Since units can implement the \wedge, \vee, \neg boolean operators, neural nets are *Turing-complete*: they can implement **any** computable function

Computing with NNs

Different functions are implemented by different network **topologies** and unit **weights**

The allure of NNs is that a network *does not need to be explicitly programmed to compute a certain function f*

Given enough nodes and links, a NN can **learn** the function by itself

It does so by

- looking at a training set of **input/output pairs** for f and
- modifying its topology and weights so that its own input/output behavior agrees with the training pairs

In other words, NNs too learn by **induction**

Learning Network Structures

The structure of a NN is given by its nodes and links

The class of functions a network can represent depends on the network structure

Fixing the network structure in advance can make the task of learning a certain function impossible

On the other hand, using a large network is also potentially problematic

If a network has too many parameters (i.e., weights), it will simply learn the examples by memorizing them in its weights (*overfitting*)

Learning Network Structures

Two main ways to modify a network structure in accordance with the training set:

Optimal brain damage: Start with a large, fully-connected network and remove connections that do not seem to matter

Tiling: Start with a very small network and increasingly add units to cover correctly more and more examples

Neither technique is completely satisfactory in practice

Often, the network structure is established **manually** by trial and error (using cross-validation, etc.)

Learning procedures are then used to **learn the network weights only**

Network structures

Feed-forward networks:

- single-layer perceptrons
- multi-layer perceptrons

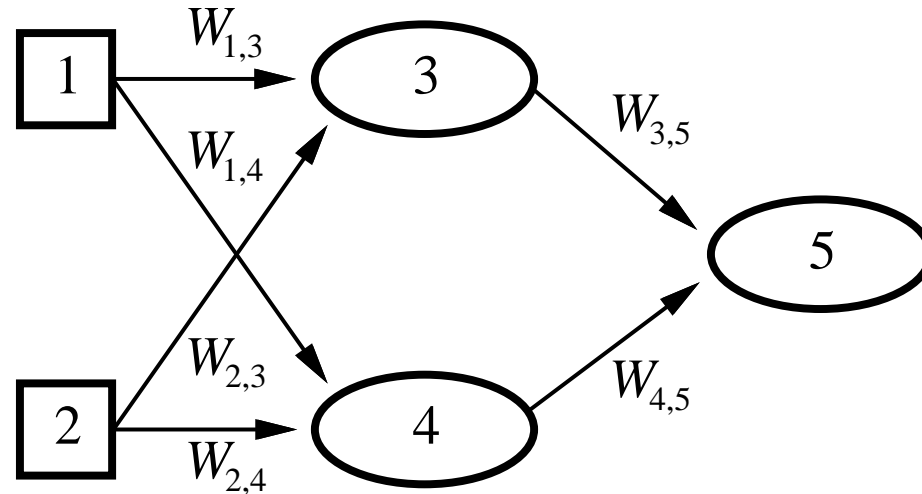
Feed-forward networks implement functions, have no internal state

Recurrent networks:

- Hopfield networks have symmetric weights ($W_{i,j} = W_{j,i}$)
 $g(x) = \text{sign}(x)$, $a_i = \pm 1$; holographic associative memory
- Boltzmann machines use stochastic activation functions

Recurrent networks have directed cycles with delays, hence have internal state (like flip-flops), can oscillate etc.

Feed-forward Network Example



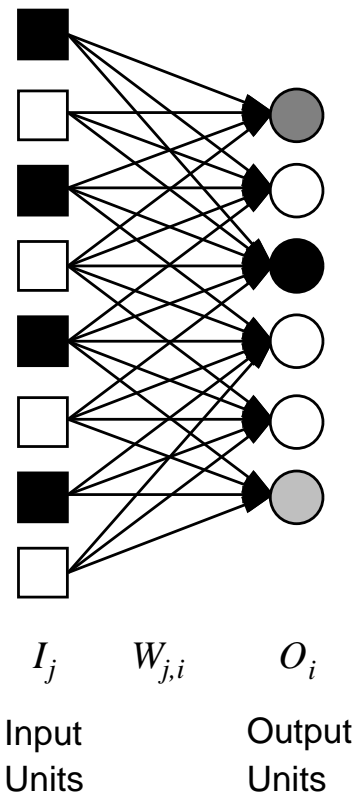
If we consider the weights as parameters, a network represents an **entire family** of nonlinear functions:

$$\begin{aligned} a_5 &= g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \\ &= g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + \\ &\quad W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2)) \end{aligned}$$

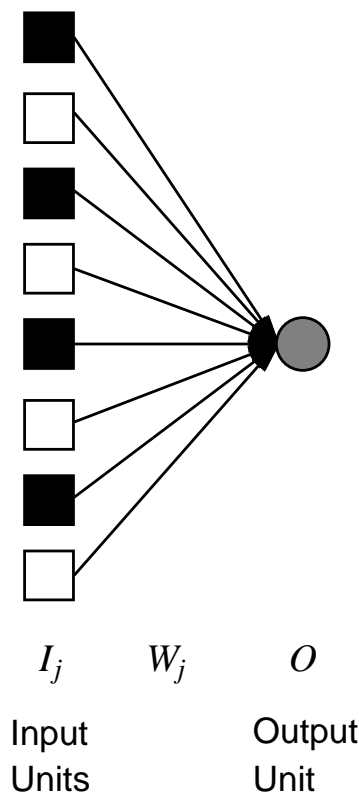
Changing weights changes the function: do learning this way!

(Single-layer) Perceptrons

Single-layer, feed-forward networks whose units use a step/sigmoid function as activation function

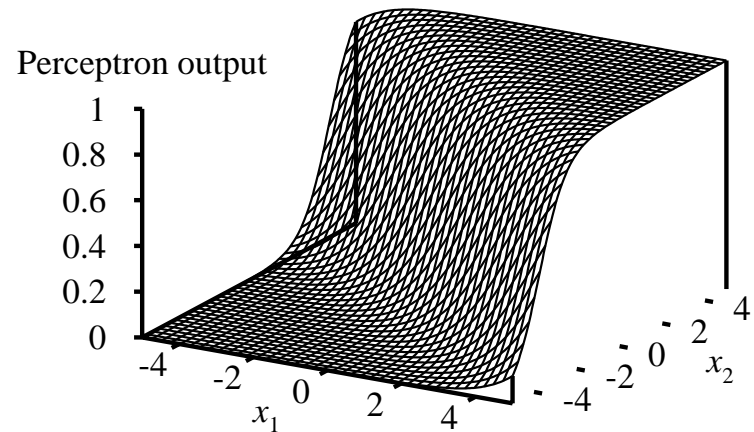
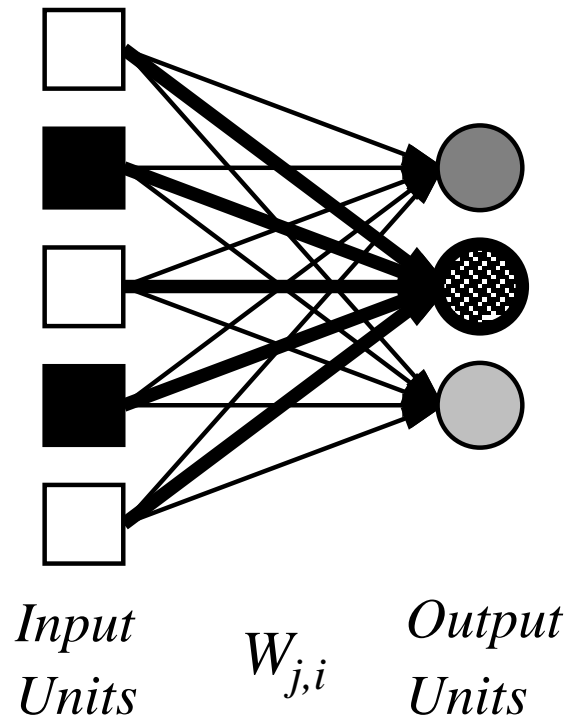


Perceptron Network



Single Perceptron

Perceptrons



Output units all operate separately—no shared weights

Adjusting weights changes the cliff's location, orientation, and steepness

Perceptron Learning

Perceptrons caused a great stir when they were invented because it was shown that

If a function is representable by a perceptron, then it is learnable with 100% accuracy, given enough training examples

Perceptron Learning

Perceptrons caused a great stir when they were invented because it was shown that

If a function is representable by a perceptron, then it is learnable with 100% accuracy, given enough training examples

Problem: perceptrons can only represent **linearly-separable functions**

Perceptron Learning

Perceptrons caused a great stir when they were invented because it was shown that

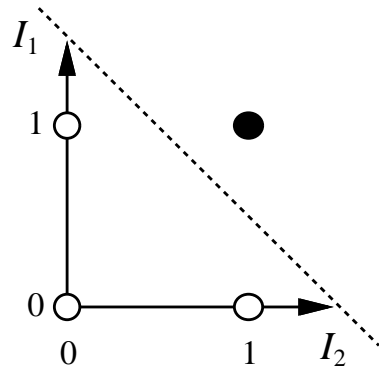
If a function is representable by a perceptron, then it is learnable with 100% accuracy, given enough training examples

Problem: perceptrons can only represent **linearly-separable functions**

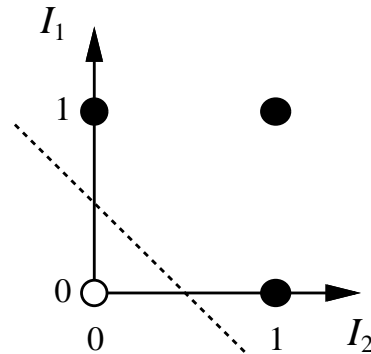
It was soon shown that most of the functions we would like to compute **are not** linearly-separable

Linearly Separable Functions

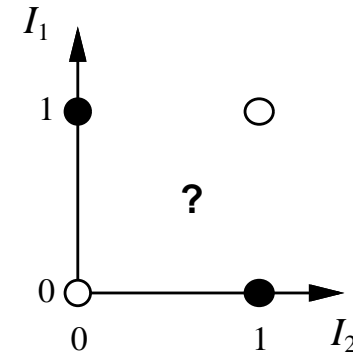
2-dimensional space:



(a) I_1 and I_2



(b) I_1 or I_2



(c) I_1 xor I_2

A black dot corresponds to an output value of 1; an empty dot corresponds to an output value of 0

Can represent **and**, **or**, **not**, majority, etc., but not **xor**

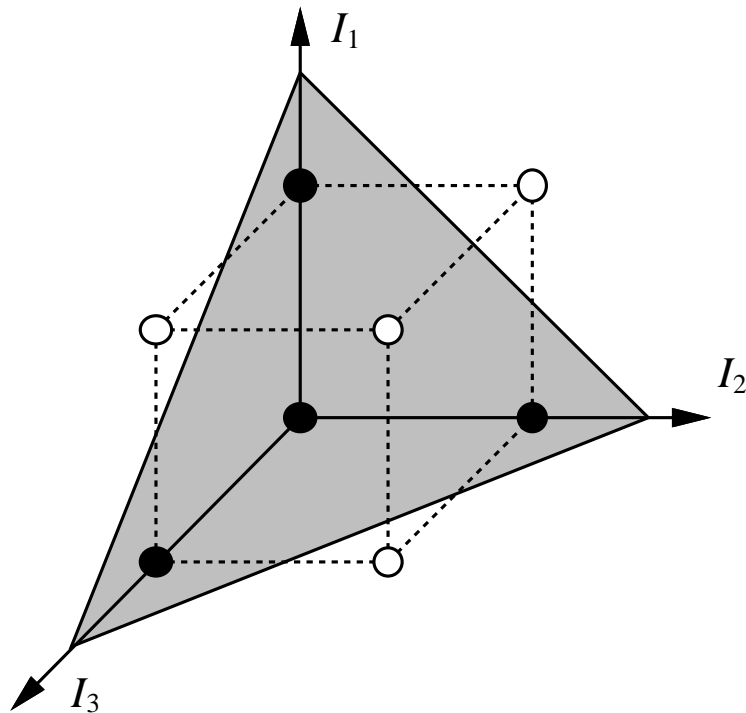
Represents a *linear separator* in input space:

$$\sum_j W_j I_j > 0 \quad \text{or} \quad \mathbf{W} \cdot \mathbf{I} > 0$$

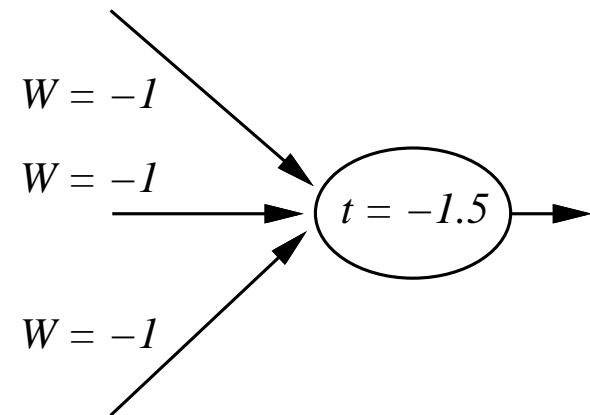
A Linearly Separable Function

3-dimensional space:

The *minority* function: return 1 if the input vector contains less than 1s than 0s; return 0 otherwise



(a) Separating plane



(b) Weights and threshold

Learning with NNs

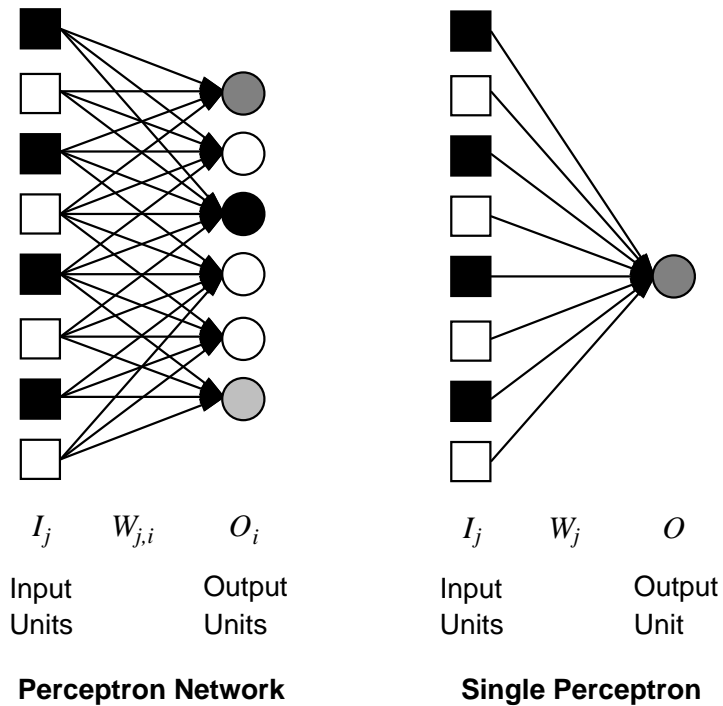
Most NN learning methods are *current-best-hypothesis* methods

```
function NEURAL-NETWORK-LEARNING(examples) returns network  
  
  network ← a network with randomly assigned weights  
  repeat  
    for each e in examples do  
      O ← NEURAL-NETWORK-OUTPUT(network, e)  
      T ← the observed output values from e  
      update the weights in network based on e, O, and T  
    end  
  until all examples correctly predicted or stopping criterion is reached  
  return network
```

Each cycle in the procedure above is called an *epoch*

The Perceptron Learning Method

Weight updating in perceptrons is very simple because each output node is independent of the other output nodes.



So we can consider a perceptron with a single output node

The Perceptron Learning Method

If O is the value returned by the output unit for a given example and T is the expected output, then the unit's error is

$$E = T - O$$

If the error E is positive we need to increase O ; otherwise, we need to decrease it

The Perceptron Learning Method

- Since $O = g(\sum_{j=0}^n W_j I_j)$ where g is the sigmoid function, we can change O by changing each W_j
- to increase O we should increase W_j if I_j is positive, decrease W_j if I_j is negative
- to decrease O we should decrease W_j if I_j is positive, increase W_j if I_j is negative
- This is done by updating each W_j in parallel as follows:

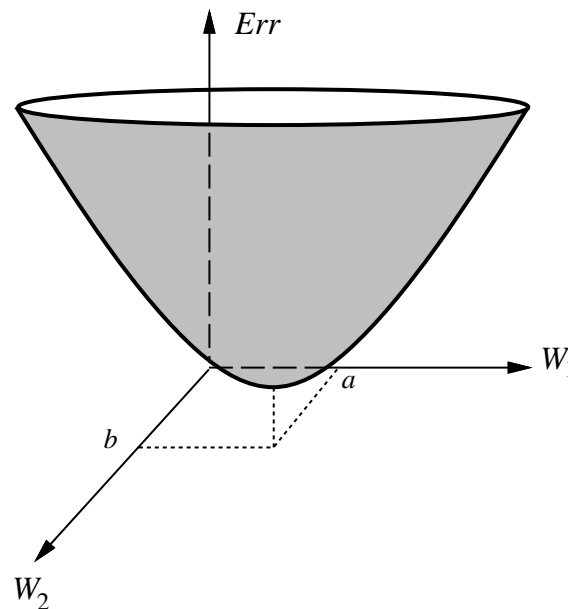
$$W_j \leftarrow W_j + \alpha \cdot I_j \cdot g'(\sum_{j=0}^n W_j I_j) \cdot (T - O)$$

where $g'(x) = g(x) \cdot (1 - g(x))$ is the first derivative of g and α is a positive constant, the *learning rate*

Perceptron Learning as Search

Provided that the learning rate constant is not too high, the perceptron **will** learn any linearly-separable function. Why?

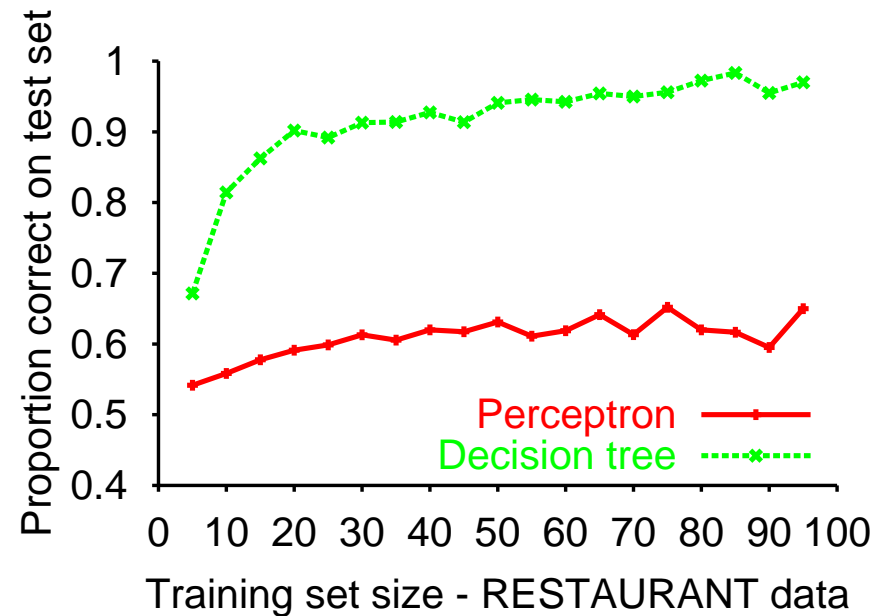
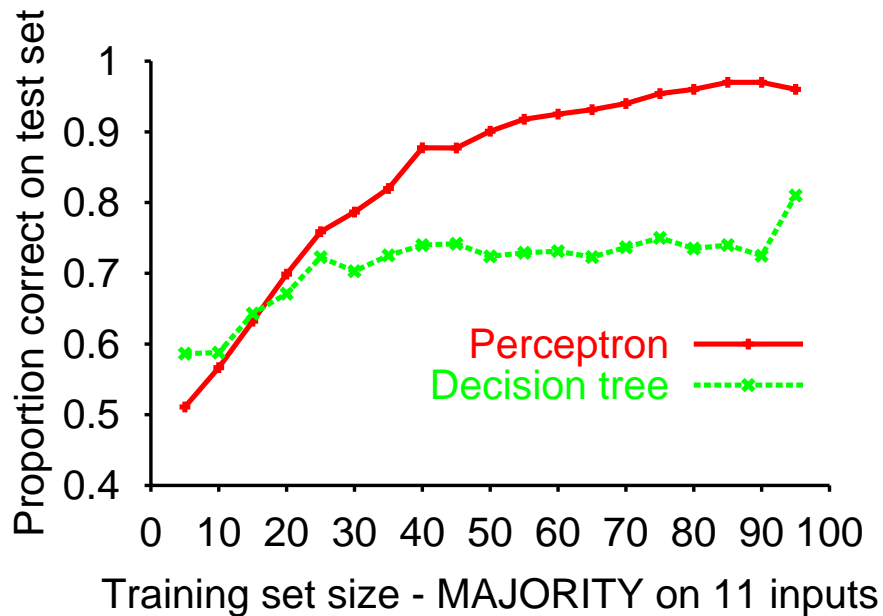
The perceptron learning procedure is a **gradient descent** search procedure whose search space has no local minima.



Each possible configuration of weights for the perceptron is a state in the search space

Perceptron learning contd.

Perceptron learning rule converges to a consistent function **for any linearly separable data set**



Perceptron learns majority function easily, DTL is hopeless

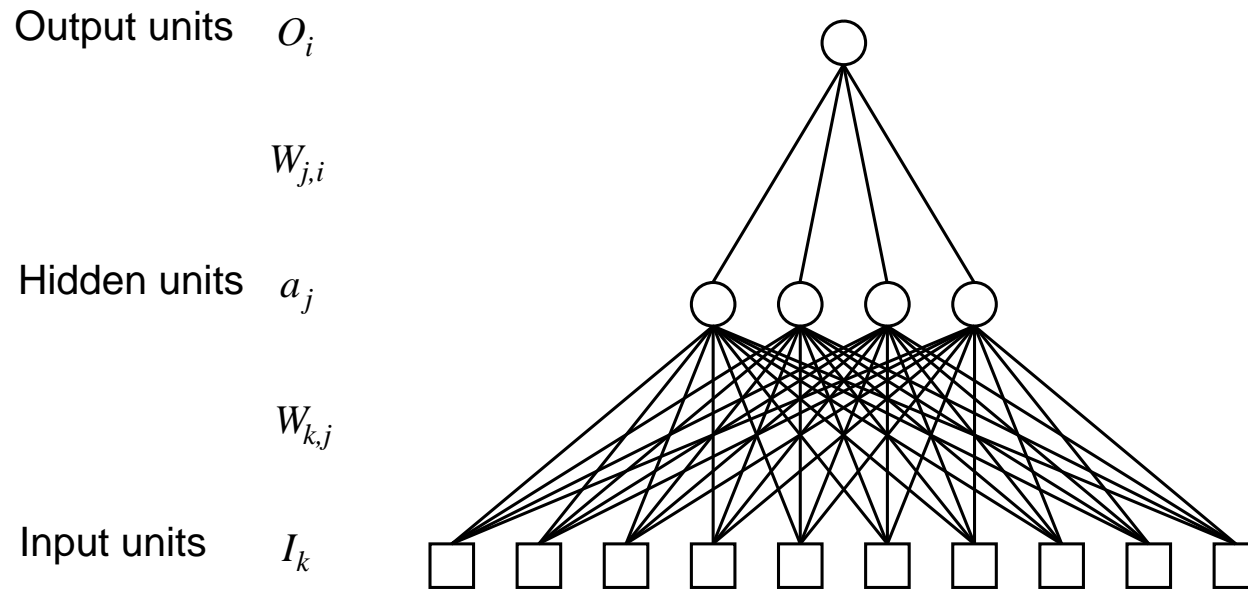
DTL learns restaurant function easily, perceptron cannot represent it

Multilayer, Feed-forward Networks

A kind of neural network in which

- links are unidirectional and form no cycles (the net is a **directed acyclic graph**)
- the root nodes of the graph are *input units*, their activation value is determined by the environment
- the leaf nodes are *output units*
- the remaining nodes are *hidden units*
- units can be divided into *layers*: a unit in a layer is connected only to units in the next layer

A Two-layer, Feed-forward Network



Notes:

- The roots of the graph are at the bottom and the (only) leaf at the top
- The layer of input units is generally not counted (which is why this is a **two**-layer net)
- Layers are usually fully connected; numbers of hidden units is typically chosen by hand

Multilayer, Feed-forward Networks

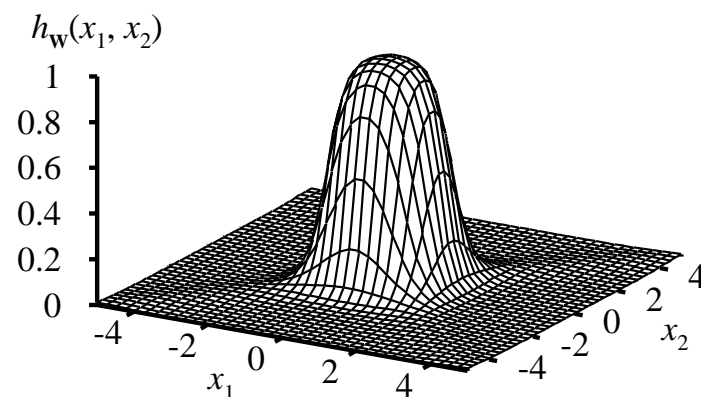
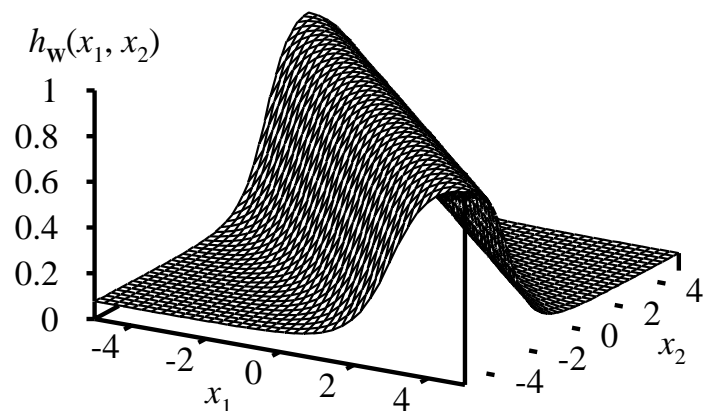
Are a powerful computational device:

- with just **one hidden layer**, they can approximate **any continuous function**
- with just **two hidden layers**, they can approximate **any computable function**

However, the number of needed units per layer may **grow exponentially** with the number of input units

Expressiveness of MLNs

All continuous functions w/ 2 layers, all functions w/ 3 layers



Combine two opposite-facing threshold functions to make a ridge

Combine two perpendicular ridges to make a bump

Add bumps of various sizes and locations to fit any surface

Proof requires exponentially many hidden units (cf. DTL proof)

Back-Propagation Learning

Extends the the main idea of perceptron learning to multilayer networks:

Assess the blame for a unit's error and divide it among the contributing weights

1. start from the units in the output layer
2. propagate the error back to previous layers up to the input layer

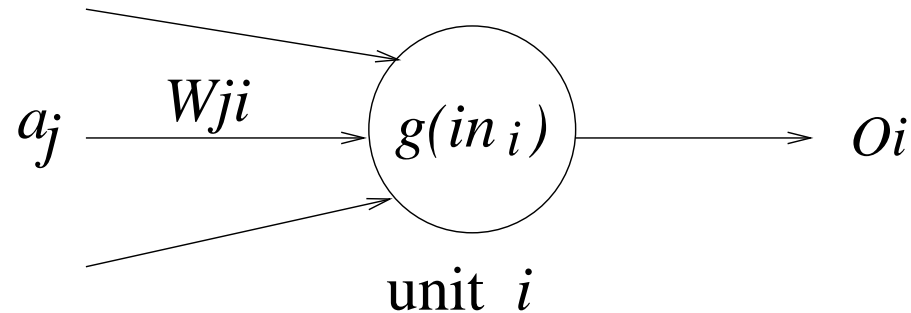
Weight updates:

Output layer: as in the perceptron case

Hidden layer: by back-propagation

Updating Weights: Output Layer

Exactly as in perceptrons:

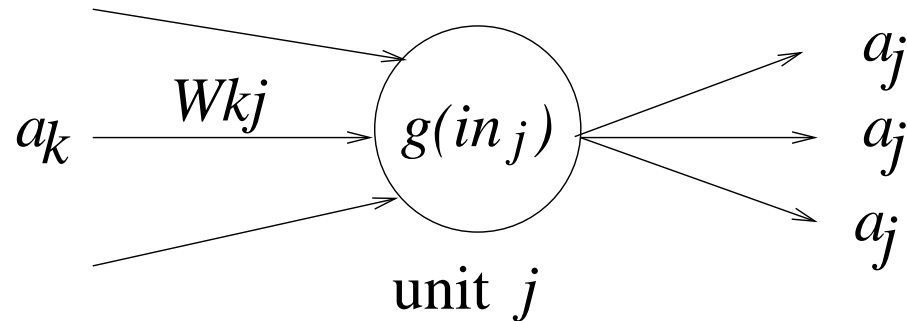


$$W_{ji} \leftarrow W_{ji} + \alpha \cdot a_j \cdot \Delta_i$$

where

- $\alpha > 0$ is the learning rate
- $\Delta_i = g'(in_i) \cdot (T_i - O_i)$ is the error of unit i
- g is the sigmoid function, $in_i = \sum_j W_{ji} a_j$
- T_i is the expected output

Updating Weights: Hidden Layers



$$W_{kj} \leftarrow W_{kj} + \alpha \cdot a_k \cdot \Delta_j$$

where

- $\Delta_j = g'(in_j) \cdot \sum_i W_{ji} \Delta_i$
- $\Delta_i =$ error of unit in the next layer that is connected to unit j

The Back-propagation Procedure

1. Choose a learning rate α
2. Choose (small) values for the weights randomly
3. Repeat until network performance is satisfactory

For each training example e

- a. Propagate e 's inputs forward to compute output O_i for each output node i
- b. For each output node i , compute

$$\Delta_i := g'(in_i) \cdot (T_i - O_i)$$

- c. For each previous level l and node j in l , compute

$$\Delta_j := g'(in_j) \cdot \sum_i W_{ji} \Delta_i$$

- d. Update each weight W_{rs} by

$$W_{rs} \leftarrow W_{rs} + \alpha \cdot a_r \cdot \Delta_s$$

Why Back-Propagation Works

Back-propagation learning too is a gradient descent search in the weight space over a certain error surface

If \mathbf{W} is the vector of all the weights in the network, the error surface is given by

$$E(\mathbf{W}) := \frac{\sum_i (T_i - O_i)^2}{2}$$

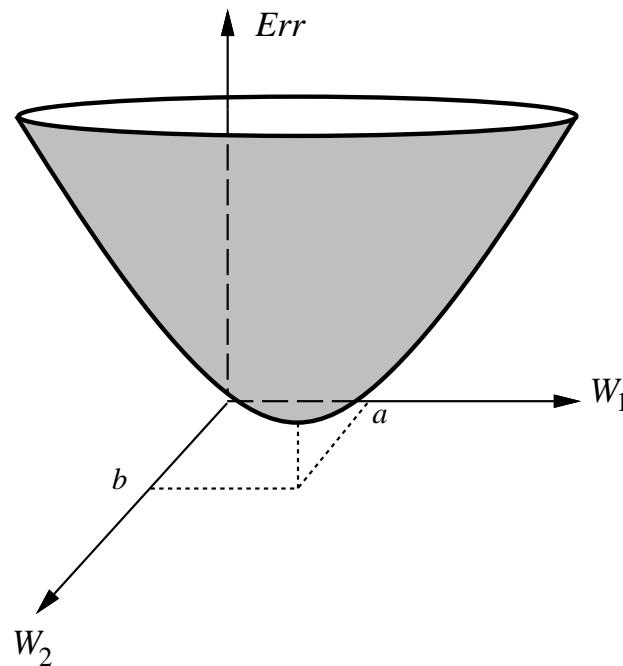
The update for each weight W_{ji} of a unit i is the opposite of the gradient (slope) of the error surface along the direction W_{ji} :

$$a_j \cdot \Delta_i = -\frac{\partial E(\mathbf{W})}{\partial W_{ji}}$$

Why BP doesn't Always Work

Producing a new vector \mathbf{W}' by adding to each W_{ji} in \mathbf{W} the opposite of E 's slope along W_{ji} guarantees that

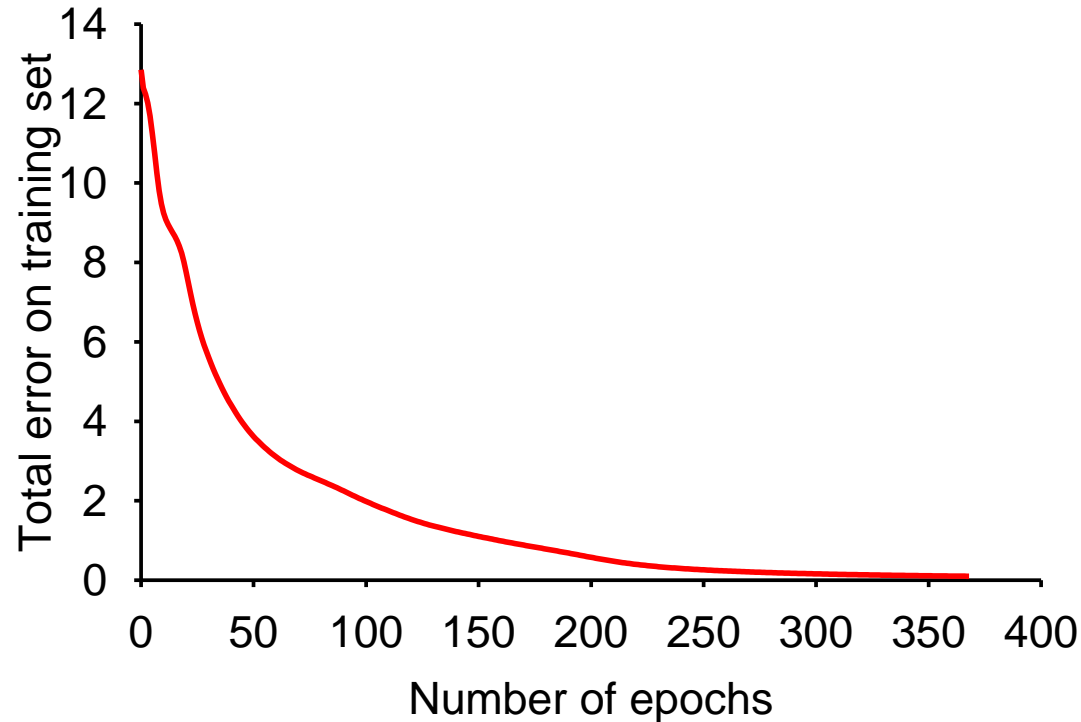
$$E(\mathbf{W}') \leq E(\mathbf{W})$$



In general, however, the error surface may contain local minima

Back-propagation learning contd.

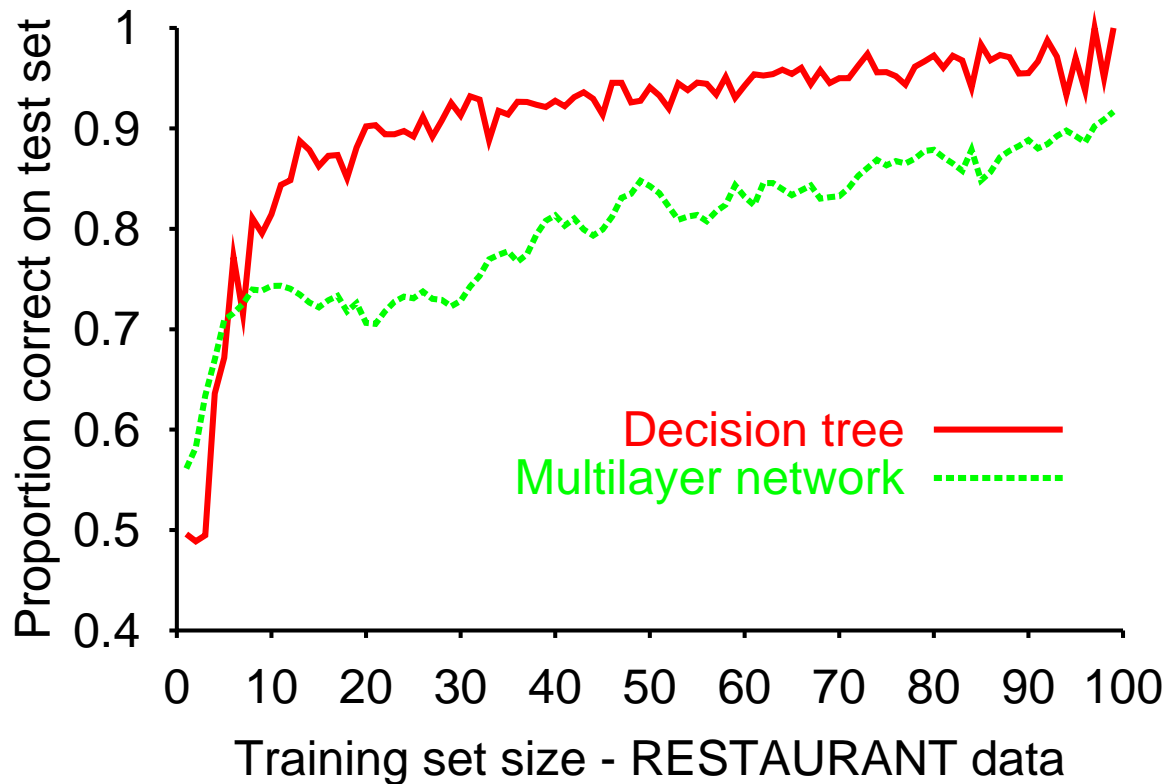
Training curve for 100 restaurant examples: finds exact fit



Typical problems: slow convergence, local minima

Back-propagation learning contd.

Learning curve for MLP with 4 hidden units:



MLNs are quite good for complex pattern recognition tasks, but resulting hypotheses cannot be understood easily

Evaluating Back-propagation

To assess the goodness of back-propagation learning for multilayer networks one must consider several issues:

- Expressiveness
- Computational efficiency
- Generalization power
- Sensitivity to noise
- Transparency
- Background Knowledge

Handwritten digit recognition



	3-NN	FFN	LeNet	B LeNet	SVM	V SVM	Match
ER	2.4	1.6	0.9	0.7	1.1	0.56	0.63
RT	1K	10	30	50	2K	200	
Mem	12	0.49	0.12	0.21	11		
T	0	7	14	30	10		
R	8.1	3.2	1.8	0.5	1.8		

ER = error rate, RT = runtime (ms/digit), M = memory (MB), TT = training time (days), R = % rejected for 0.5% error