Biologically Inspired Computing: Neural Computation

Lecture 4

Patricia A. Vargas

Lecture 4

I. Lecture 3 – Revision
II. Artificial Neural Networks (Part II)
    I. Multi-Layer Perceptrons
        I. The Back-propagation Algorithm

Artificial Neural Networks

- Learning Paradigms

$$w(t+1) = w(t) + \Delta w(t)$$

I. Supervised Learning

II. Unsupervised Learning

III. Reinforcement Learning

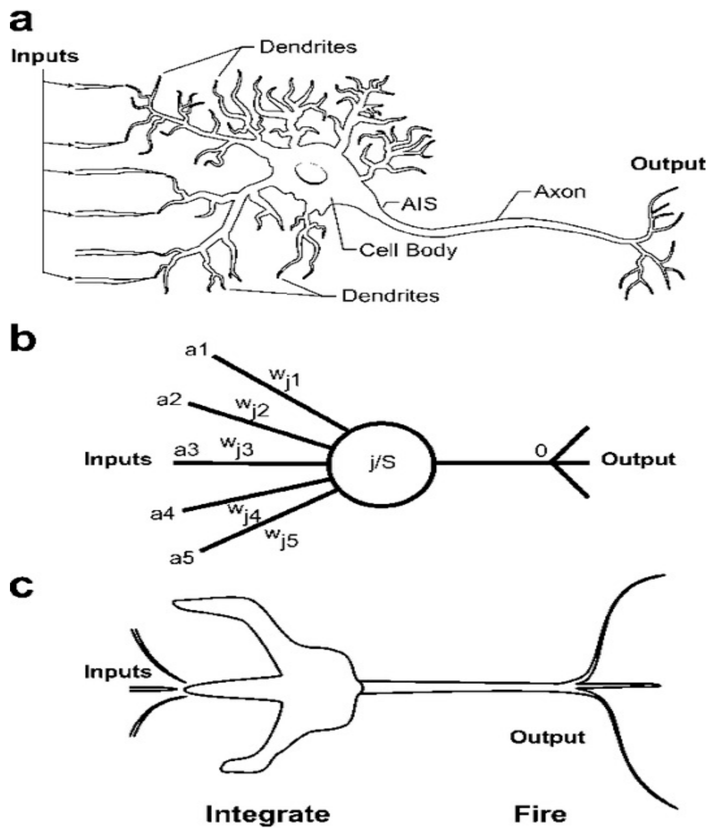
Two main forms of learning

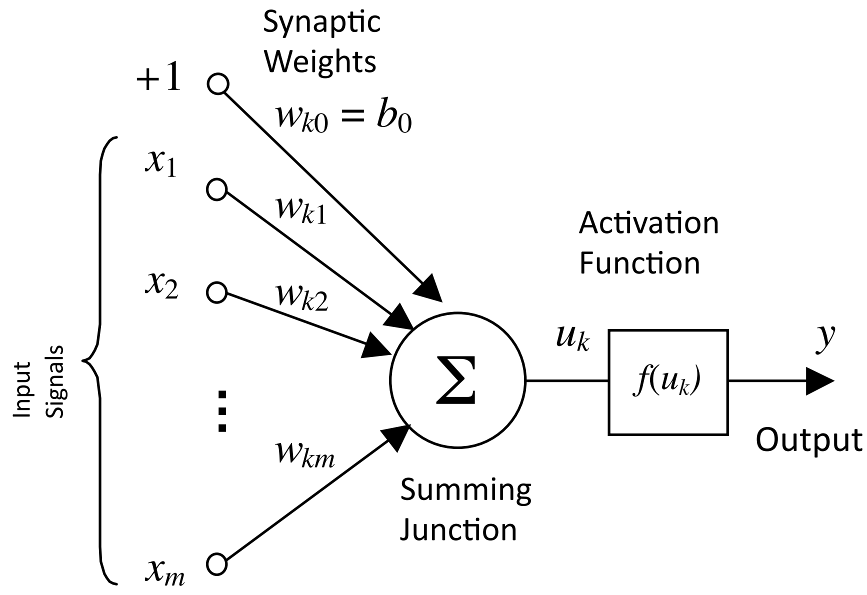
- Supervised Learning
  - Error-correcting learning
    - ✓Perceptron
      - delta rule
    - Multi-Layer Perceptron (MLP)
      - Back-propagation (generalized delta rule)

- Unsupervised Learning
  - Hopfield Neural Network

Artificial Neural Networks

- Frank Rosenblatt (1957)
  - Perceptron





$$y_k = f(u_k) = f\left(\sum_{j=0}^{m} w_{kj} x_j\right)$$

Perceptron algorithm in pseudo-code

```
Start with random initial weights (e.g., uniform random in [-.3,.3])

Do
{
  For All Patterns p
  {
    For All Output Nodes j
    {
      CalculateActivation(j)

      Error_j = TargetValue_j_for_Pattern_p - Activation_j

      For All Input Nodes i To Output Node j
      {
        DeltaWeight = LearningConstant * Error_j * Activation_i
        Weight = Weight + DeltaWeight
      }
    }
  }
}
Until "Error is sufficiently small" Or "Time-out"
```

Limitations of the Perceptron

- Can only represent linear separable problems…

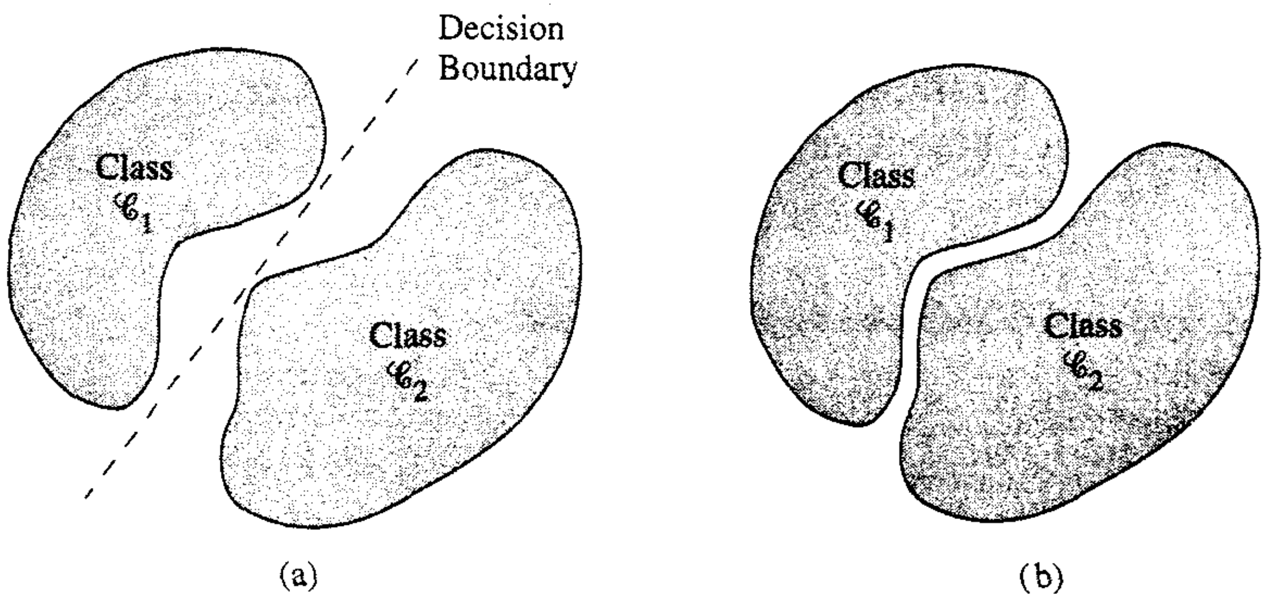
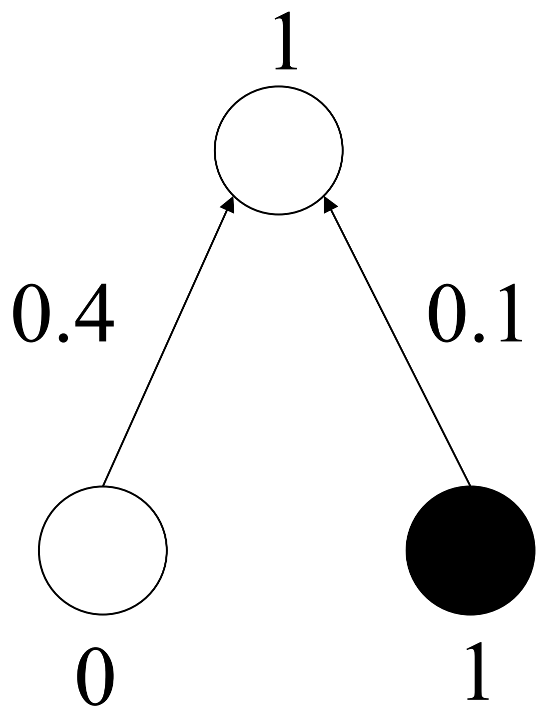


FIGURE 3.9 (a) A pair of linearly separable patterns. (b) A pair of non-linearly separable patterns.

Exclusive OR (XOR)

In	**Out**
0 1	1
1 0	1
1 1	0
0 0	0

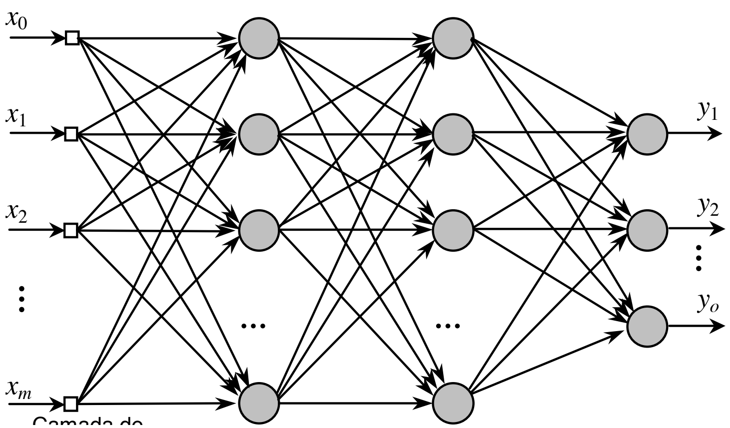


Error-backpropagation ?

- What was needed, was an algorithm to train Perceptrons with more than two layers
- Preferably also one that used continuous activations and non-linear activation rules
- Such an algorithm was developed by
  - Paul Werbos in 1974
  - David Parker in 1982
  - LeCun in 1984
  - Rumelhart, Hinton, and Williams in 1986

Artificial Neural Networks

The Multi-Layer Perceptron (MLP)

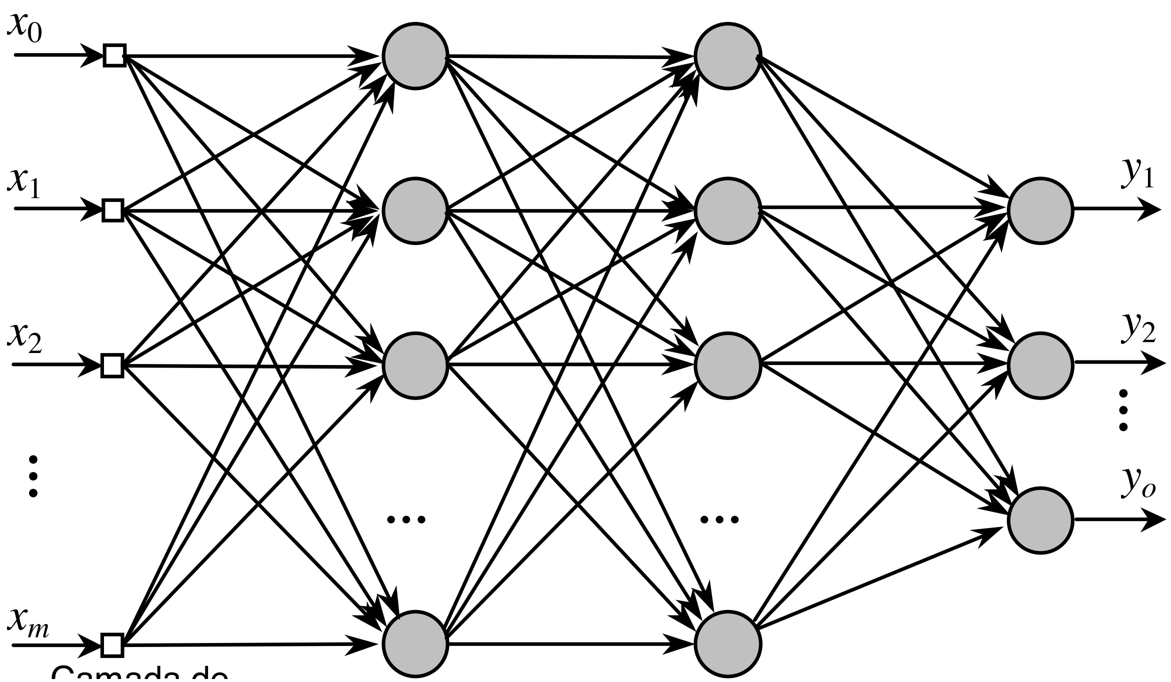


- The XOR problem is solvable if we add an extra "layer" to a Perceptron

- MLPs become more manageable, mathematically and computationally, if we formalise them into a standard structure (or *topology* or *architecture*)…

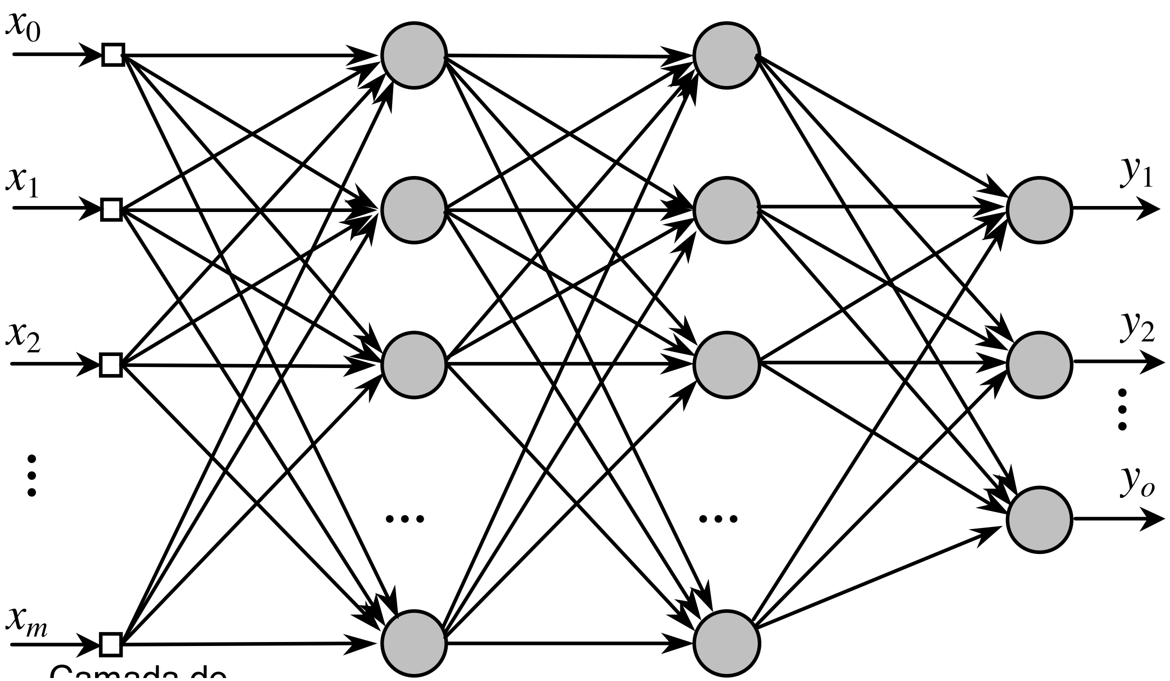
The Multi-Layer Perceptron (MLP)

- Each node is connected to EVERY node in the adjacent layers and NO nodes in the same or any other layers



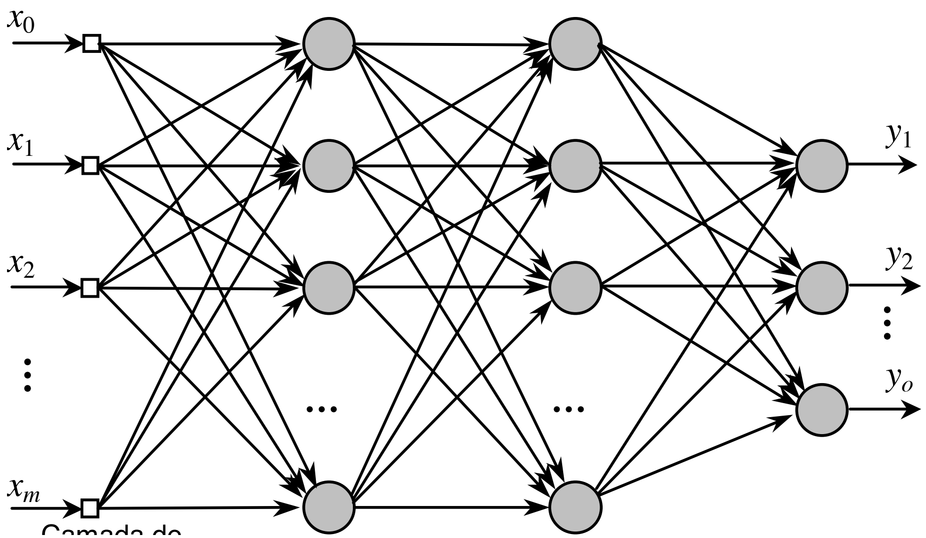
The Multi-Layer Perceptron (MLP)

How do I represent the weights of a MLP in a matrix notation?



The Multi-Layer Perceptron (MLP)

How do I represent the MLP using a matrix notation?



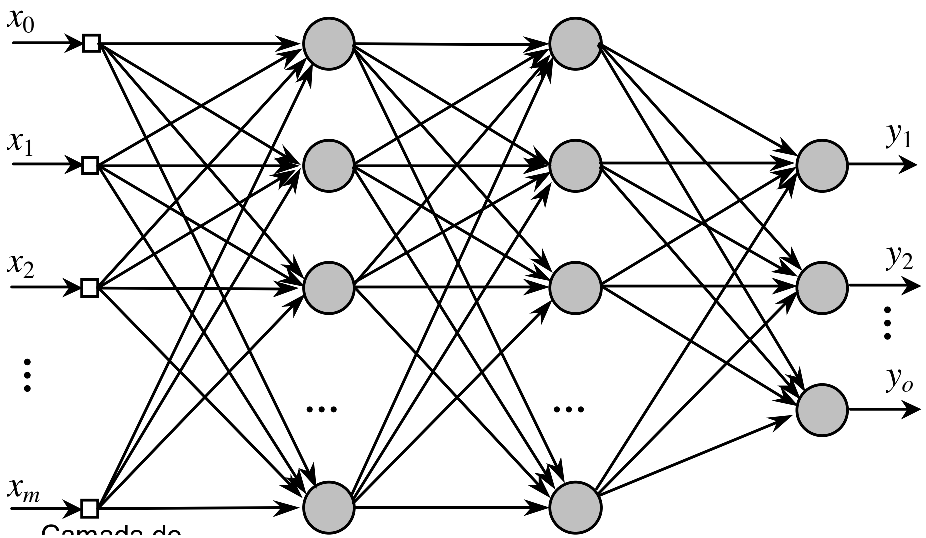
$$\mathbf{W} = \begin{bmatrix} w_{10} & w_{11} & \cdots & w_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{o0} & w_{o1} & \cdots & w_{om} \end{bmatrix}$$

$$y_i = f(\mathbf{w}_i.\mathbf{x}) = f(\Sigma_j \, w_{ij}.x_j) \;, j = 1,\ldots,m.$$

BUT we have many layers....

The Multi-Layer Perceptron (MLP)

How do I represent the MLP using a matrix notation?



$$\mathbf{W} = \begin{bmatrix} w_{10} & w_{11} & \cdots & w_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{o0} & w_{o1} & \cdots & w_{om} \end{bmatrix}$$

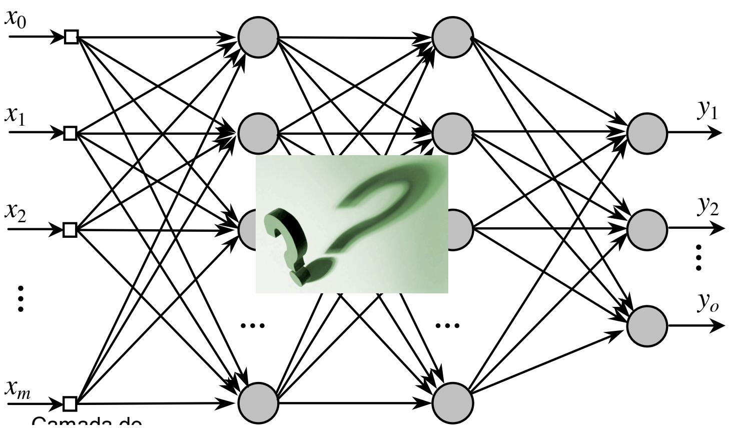
$$y_i = f(\mathbf{w}_i.\mathbf{x}) = f(\Sigma_j\, w_{ij}.x_j) \;, j = 1,\ldots,m.$$

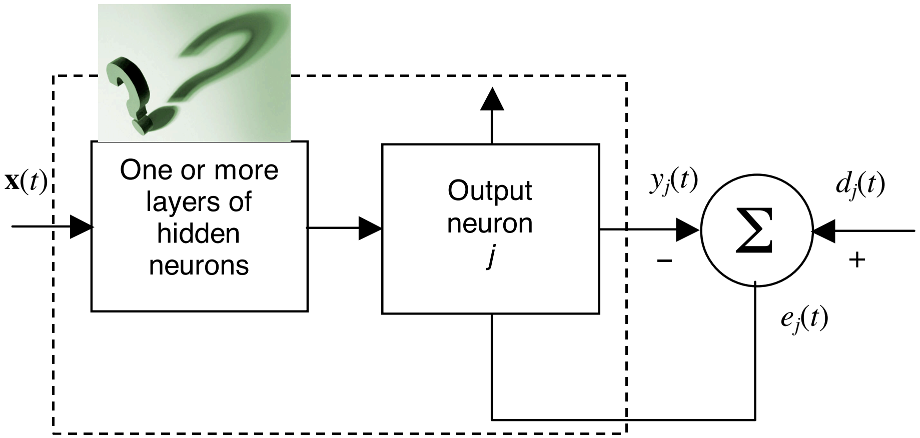
W^k is the synaptic weight matrix of layer *k*

$$\mathbf{y} = \mathbf{f}^3(\mathbf{W}^3\, \mathbf{f}^2(\mathbf{W}^2\, \mathbf{f}^1(\mathbf{W}^1\mathbf{x})))$$

How to train a Multi-Layer Perceptron (MLP)?

- How do we find the weights needed to perform a particular function?
- The problem lies in determining an error at the hidden nodes
- We have no desired value at the hidden nodes with which to compare their actual output and determine an error
- We have a desired output which can deliver an error at the output nodes but how should this error be divided up amongst the hidden nodes?

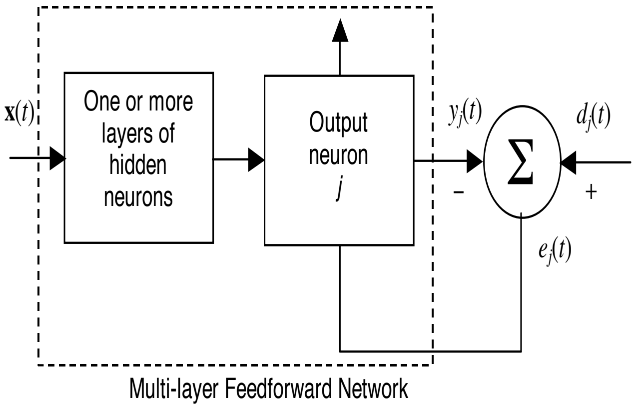


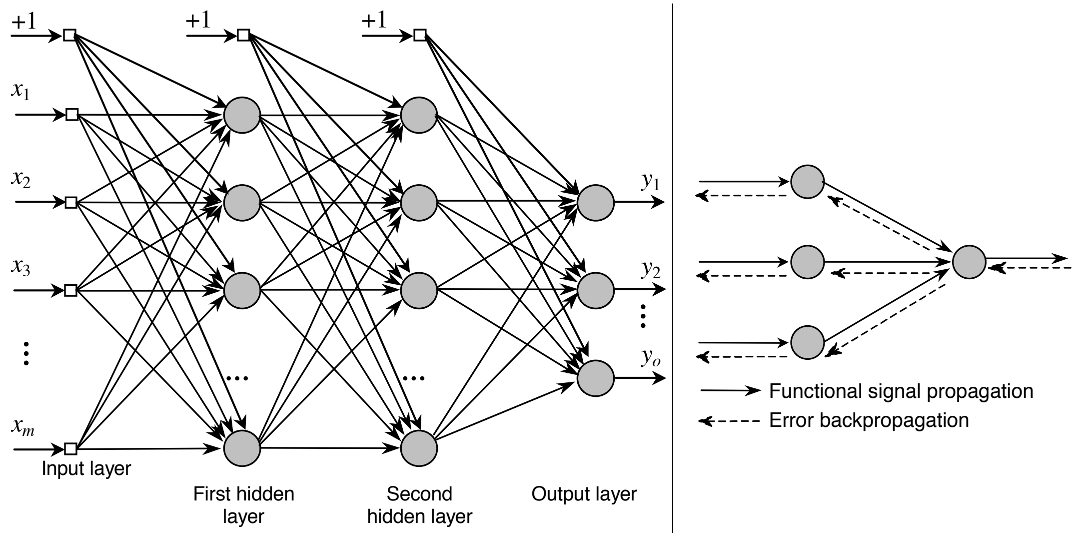


Multi-layer Feedforward Network

The Back-Propagation Algorithm

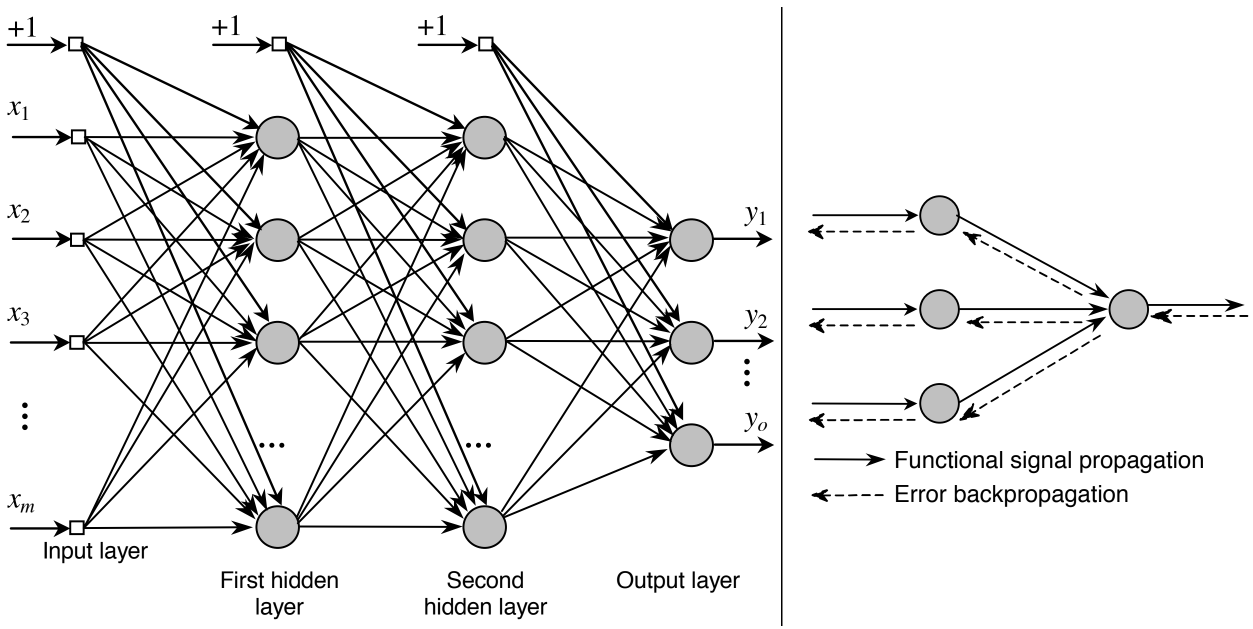
- In 1986 Rumelhart, Hinton and Williams proposed a "Generalised Delta Rule"
- also known as Error Back-Propagation or Gradient Descent Learning
- This rule, as its name implies, is an extension of the Delta Rule or "Widrow-Hoff Rule"





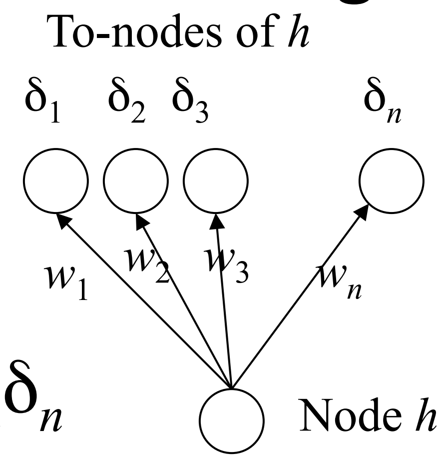
The Back-Propagation Algorithm:

1. Feed inputs forward through network
2. Determine error at outputs
3. Feed error backwards towards inputs
4. Determine weight adjustments
5. Repeat for next input pattern
6. Repeat until all errors acceptably small



The backprop trick

- To find the error value for a given node *h* in a hidden layer, …
- Simply take the weighted sum of the errors of all nodes connected from node *h*
- i.e., of all nodes that have an incoming connection from node *h*:



This is backpropgation of errors

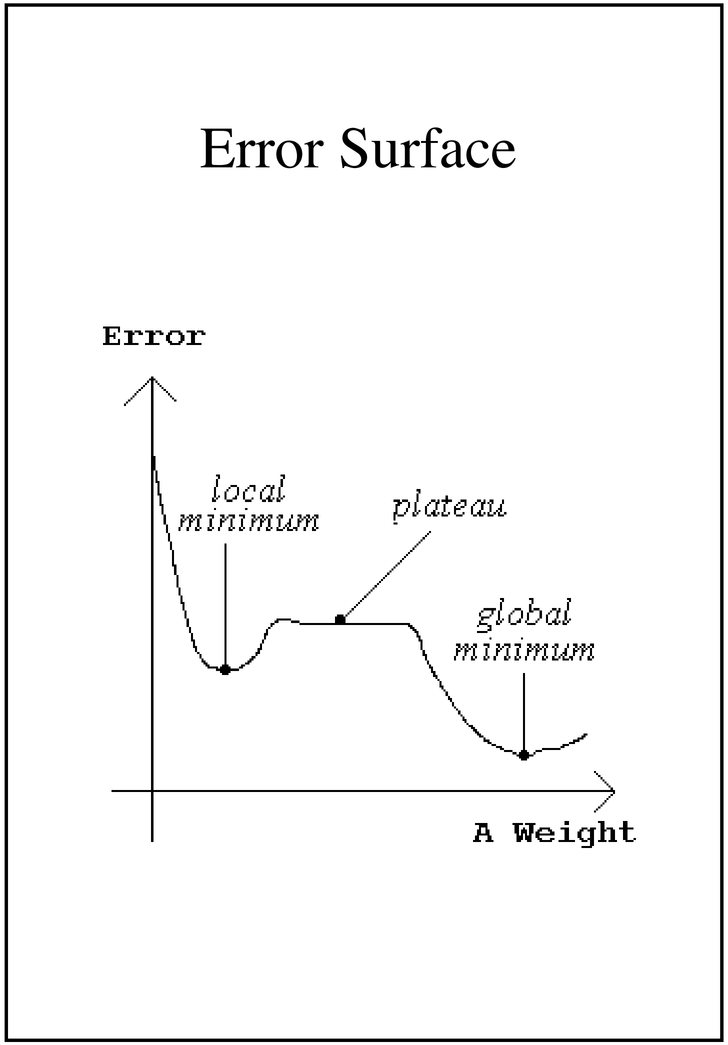
$$\delta_h = w_1\delta_1 + w_2\delta_2 + w_3\delta_3 + \ldots + w_n\delta_n$$

Characteristics of backpropagation

- Any number of layers
- Only feedforward, no cycles (though a more general versions does allow this)
- Use continuous nodes
  - Must have differentiable activation rule
  - Typically, *logistic*: S-shape between 0 and 1
- Initial weights are random

The gradient descent makes sense mathematically

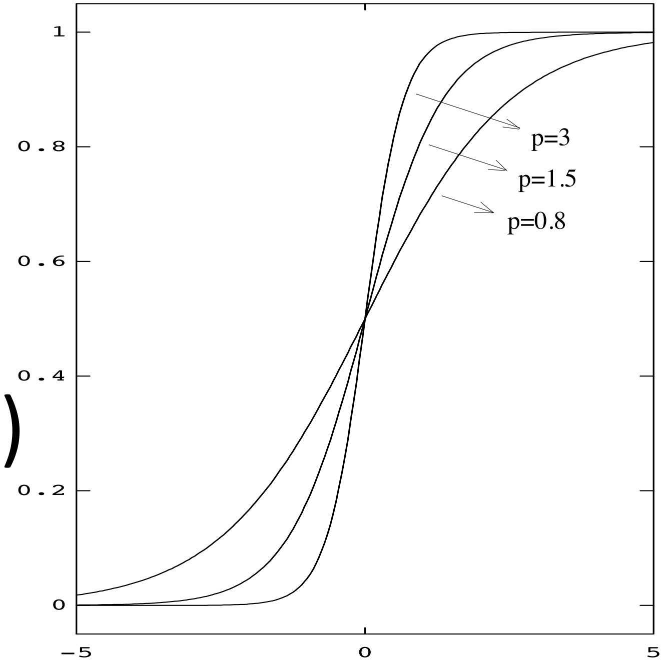
- It does not guarantee high performance
- It does not prevent local minima



Logistic function

- S-shaped between 0 and 1
- Approaches a linear function around $x = 0$
- Its rate-of-change (derivative) for a node with a given activation is:

  activation × (1 - activation)



Backpropagation algorithm in rules

- weight change = some small constant × error × input activation
- For an output node, the error is:

  error = (target activation - output activation) × output activation × (1 - output activation)

- For a hidden node, the error is:

  error = weighted sum of to-node errors × hidden activation × (1 - hidden activation)

Weight change and momentum

- backpropagation algorithm often takes a long time to learn
- So, the learning rule is often augmented with a so called momentum term
- This consist in adding a fraction of the old weight change
- The learning rule then looks like:

  weight change = some small constant × error × input activation + momentum constant × old weight change

Backpropagation in equations I

- If j is a node in an output layer, the error δ_j is:

  $\delta_j = (t_j - a_j)\ a_j(a_j - 1)$

- where a_j is the activation of node j
- t_j is its target activation value, and
- δ_j its error value

Backpropagation in equations II

- If *j* is a node in a hidden layer, and if there are *k* nodes 1, 2, …, *k*, that receive a connection from *j*, the error δ_j is:
- $\delta_j = (w_{1j}\delta_1 + w_{2j}\delta_1 + \ldots + w_{kj}\delta_k)\, a_j\,(a_j - 1)$
- where the weights w_{1j}, w_{2j}, …, w_{kj} belong to the connections from hidden node *j* to nodes 1, 2, …, *k*.

Backpropagation in equations III

- The backpropagation learning rule (applied at time t) is:

$$\Delta w_{ji}(t) = \mu \, \delta_j a_i + \beta \Delta w_{ji}(t\text{-}1)$$

- where $\Delta w_{ji}(t)$ is the change in the weight from node i to node j at time t,
- The learning constant μ is typically chosen rather small (e.g., 0.05).
- The momentum term β is typically chosen around 0.5.

NetTalk: Backpropagation's 'killer-app'

- Text-to-speech converter
- Developed by Sejnowski and Rosenberg (1987)

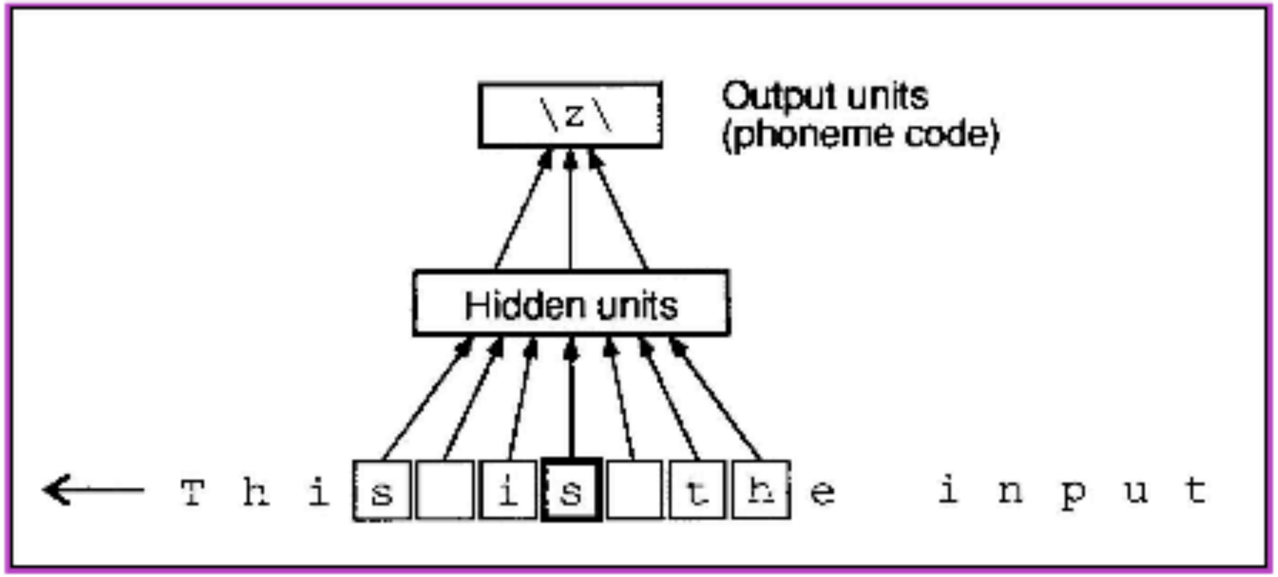
(Sejnowski & Rosenberg, 1987 *"Parallel Networks that Learn to Pronounce English Text", Complex Systems 1, 145-168)*

- Connectionism's answer to DECTalk
- Learned to pronounce text with an error score comparable to DECTalk
- Was trained, not programmed
- Input was letter-in-context, output phoneme

NetTalk: Backpropagation's 'killer-app'

- Project for pronouncing English text: for each character, the network should give the code of the corresponding phoneme:
  - A stream of words is given to the network, along with the phoneme pronunciation of each in symbolic form
  - A speech generation device is used to convert the phonemes to sound
- The same character is pronounced differently in different contexts: Head, Beach, Leech, Sketch

NetTalk: Backpropagation’s ‘killer-app’



- Input is rolling sequence of 7 characters
- 7 x 29 possible characters = 203 binary inputs
- 80 neurons in one hidden layer
- 26 output neurons (one for each phoneme code)
- 16,240 weights in the first layer; 2,080 in the second

→ 203-80-26 two-layer network

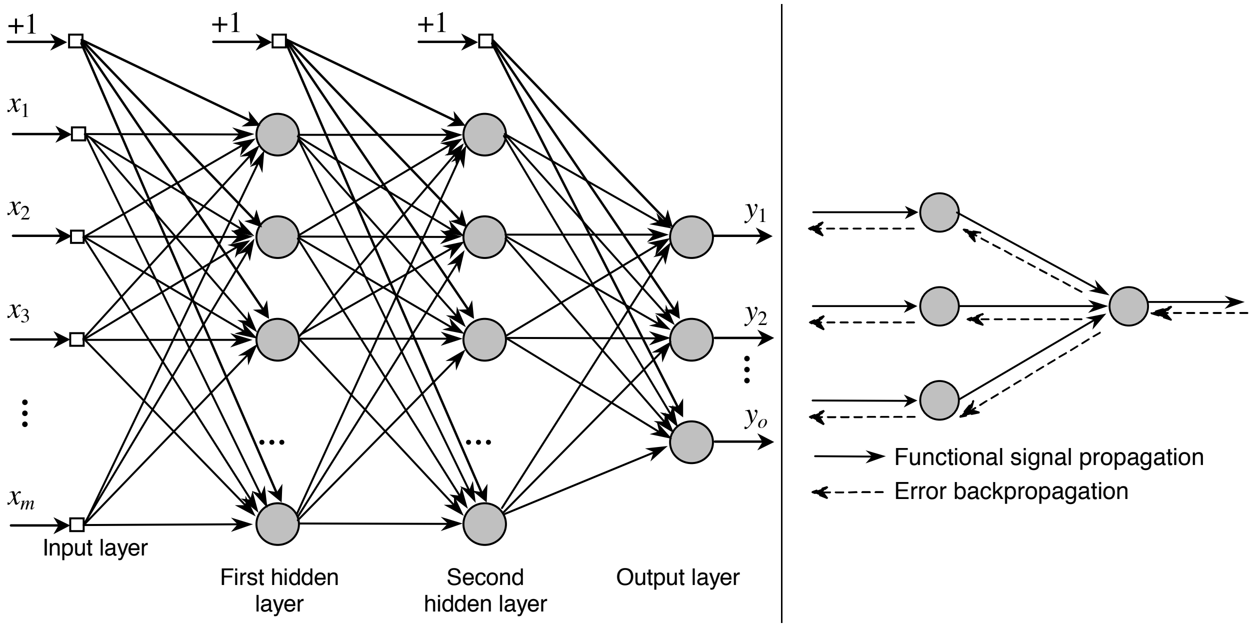
NetTalk: Backpropagation's 'killer-app'

- Training set: database of 1,024 words
- After 10 epochs the network obtains intelligible speech; after 50 epochs 95% accuracy is achieved
  - generalization: 78% accuracy on continuation of training text
  - Since three characters on each side are not always enough to determine the correct pronunciation, 100% accuracy cannot be obtained
- The learning process
  - Gradually performs better and better discrimination
  - Sounds like a child learning to talk
  - damaging network produced graceful degradation, with rapid recovery on retraining
- Analysis of the hidden neurons reveals that some of them represent meaningful properties of the input (e.g., vowels vs. consonants)

http://homepages.cae.wisc.edu/~ece539/data

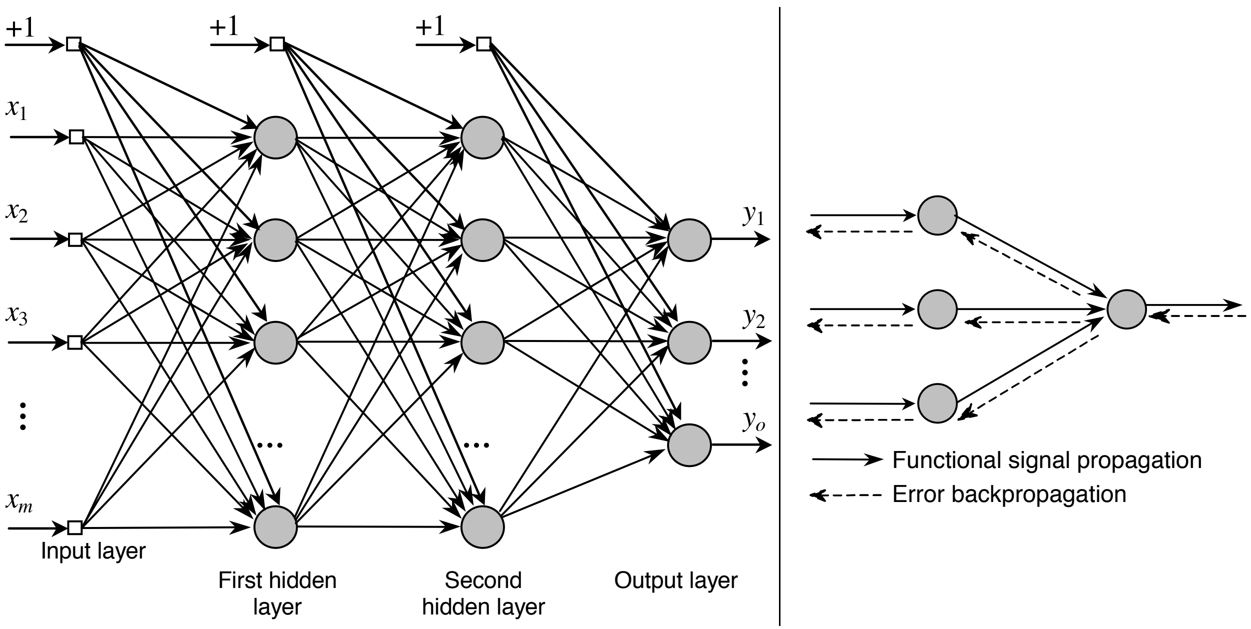
The Back-Propagation Algorithm

- Batch Learning
  - sum the weight updates for each input pattern and apply them after a complete set of training patterns has been presented (after one "**epoch of training**")
  - therefore, adjustments to the synaptic weights are made on an epoch-by-epoch basis



The Back-Propagation Algorithm

- On-Line Learning
  - update weights as each input pattern is presented
  - therefore, adjustments to the synaptic weights are made on an example-by-example basis



Despite its popularity backpropagation has some disadvantages

- Learning is slow
- New learning will rapidly *overwrite* old representations, unless these are interleaved (i.e., repeated) with the new patterns
- This makes it hard to keep networks up-to-date with new information (e.g., dollar rate)
- This also makes it very implausible from as a psychological model of human memory

Good points

- Easy to use
  - Few parameters to set
  - Algorithm is easy to implement
- Can be applied to a wide range of data
- Is very popular
- Has contributed greatly to the 'new connectionism' (second wave)

Conclusion

- Error-correcting learning has been very important in the brief history of connectionism
- Despite its limited plausibility as a psychological model of learning and memory, it is nevertheless used widely (also in psychology)

Lecture 4

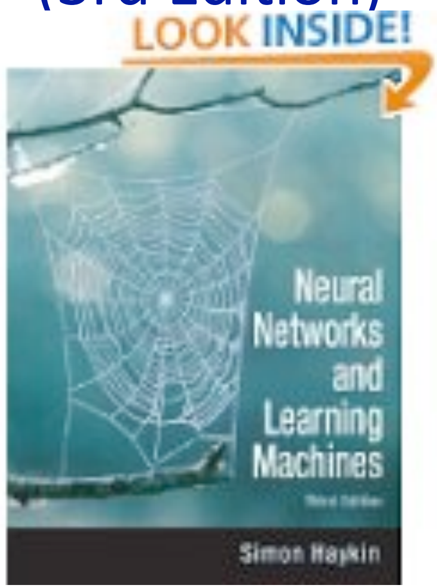
I. Lecture 3 – Revision
II. Artificial Neural Networks (Part II)
    I. Multi-Layer Perceptron
        I. The Back-propagation Algorithm

Lecture 4

Reading list/Homework

- Take a look at Chapter 4 (“Multilayer Perceptrons”) from the book:

  “Neural Networks and Learning Machines” (3rd Edition) by Simon O. Haykin (Nov 28, 2008)



- Answer questions 17 to 20 from the Tutorial material

Lecture 5

What's next?

Artificial Neural Networks

(Part III)