Νευρο-Ασαφής Υπολογιστική
Neuro-Fuzzy Computing

Διδάσκων –
Δημήτριος Κατσαρός

@ Τμ. ΗΜΜΥ
Πανεπιστήμιο Θεσσαλίας

Διάλεξη 6η

Backpropagation

A bit of history

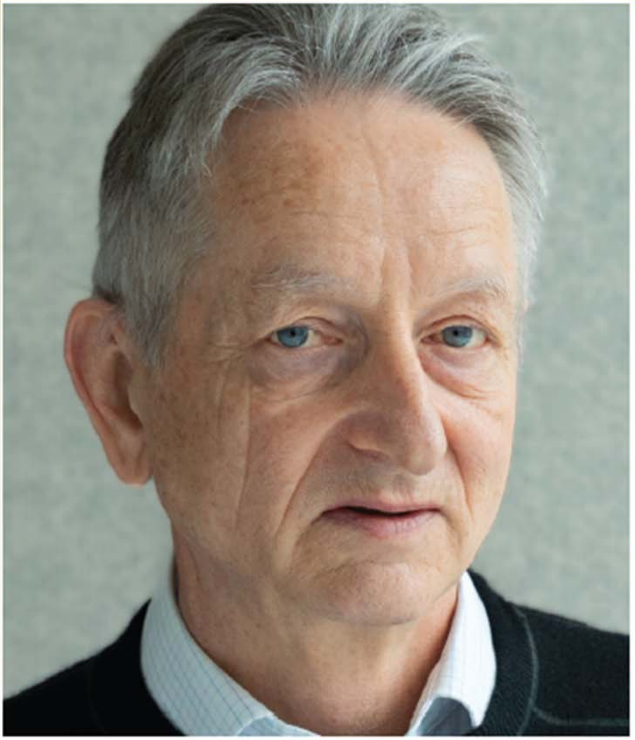
- The first description of an algorithm to train multilayer networks was contained in the thesis of Paul Werbos in 1974
- It was not until the mid 1980s that the backpropagation algorithm was rediscovered and widely publicized. It was rediscovered independently by
  - David Rumelhart, Geoffrey Hinton, Ronald Williams in 1986

AI Research, Google
Resigned

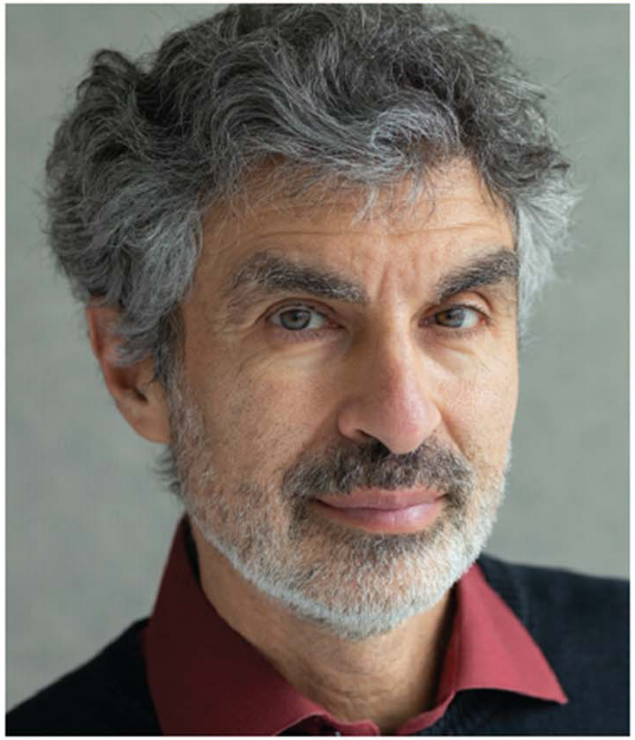
  - David Parker in 1985
  - Yann Le Cun in 1985 (director of AI Research, Facebook)

P. J. Werbos, “Beyond regression: New tools for prediction and analysis in the behavioral sciences,” **Ph.D. Thesis**, Harvard University, Cambridge, MA, 1974. Also published as The Roots of Backpropagation, New York: John Wiley & Sons, 1994.
D. E. Rumelhart, G. E. Hinton and R. J. Williams, “Learning representations by back-propagating errors,” **Nature**, vol. 323, pp. 533–536, 1986.
D. B. Parker, “Learning-logic: Casting the cortex of the human brain in silicon,” **Technical Report TR-47**, Center for Computational Research in Economics and Management Science, MIT, Cambridge, MA, 1985.
Y. Le Cun, “Une procedure d’apprentissage pour reseau a seuil assymetrique,” **Cognitiva**, vol. 85, pp. 599–604, 1985.

Turing Award (called the Nobel Prize of computing) 2018



Geoffrey Hinton



Yoshua Bengio

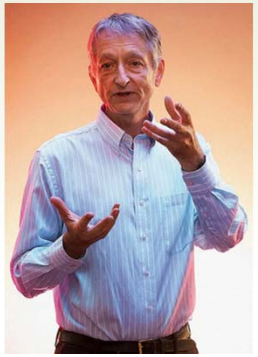
Yann LeCun

Nobel Prize in Physics 2024

https://www.nobelprize.org/prizes/physics/2024/summary/

“for foundational discoveries and inventions that enable machine learning with artificial neural networks”

John J. Hopfield
Born: 15 July 1933, Chicago, IL, USA
Affiliation at the time of the award: Princeton University, Princeton, NJ, USA



Geoffrey E. Hinton
Born: 1947, London, UK.
PhD 1978 from The University of Edinburgh, UK.
Affiliation at the time of the award: Professor at University of Toronto, Canada

Nobel Prize in Chemistry 2024

https://www.nobelprize.org/prizes/chemistry/2024/summary/

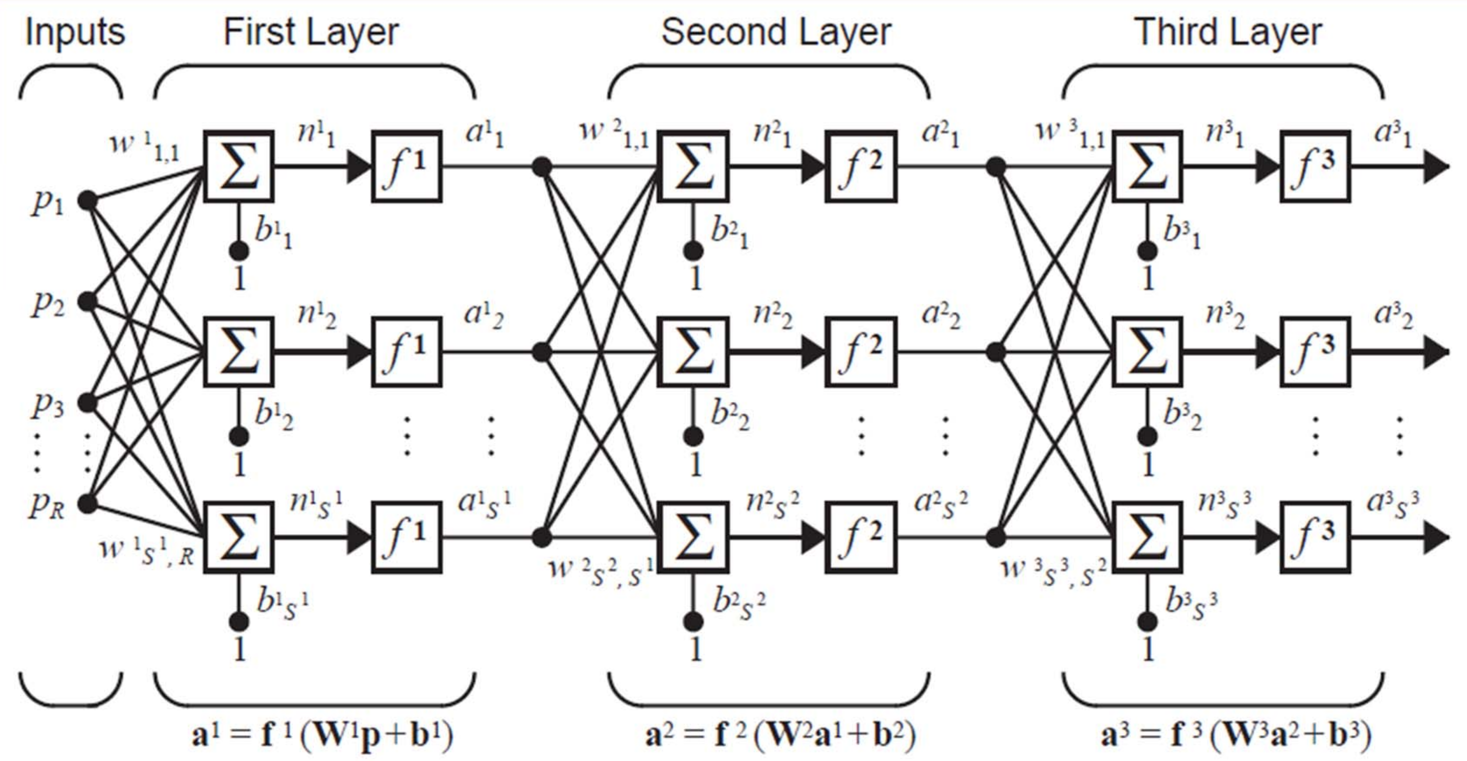
“one half awarded to David Baker "for computational protein design", the other half jointly to Demis Hassabis and John M. Jumper "for protein structure prediction"”

Demis Hassabis
Born: 27 July 1976, London, United Kingdom
Affiliation at the time of the award: Google DeepMind, London, United Kingdom

John M. Jumper
Born: 1985, Little Rock, AR, USA
Affiliation at the time of the award: Google DeepMind, London, United Kingdom

Three layer network

- Recall from previous lecture a multilayer neural network
- To identify the structure of a multilayer network, we will use R-S^1-S^2-S^3, where the number of inputs is followed by the number of neurons in each layer

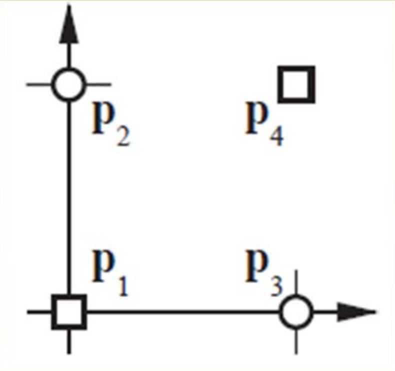


$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1) \qquad \mathbf{a}^2 = \mathbf{f}^2(\mathbf{W}^2\mathbf{a}^1+\mathbf{b}^2) \qquad \mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{a}^2+\mathbf{b}^3)$$

$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{f}^2(\mathbf{W}^2\mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1)+\mathbf{b}^2)+\mathbf{b}^3)$$

The XOR example

- The input/target pairs for the XOR gate are:

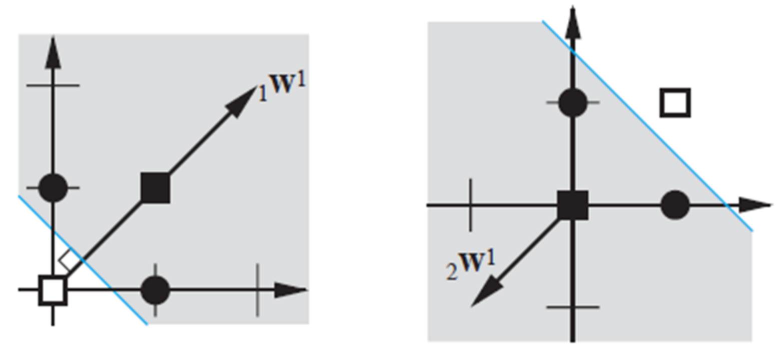


$$\left\{\mathbf{p}_1 = \begin{bmatrix}0\\0\end{bmatrix}, t_1 = 0\right\} \left\{\mathbf{p}_2 = \begin{bmatrix}0\\1\end{bmatrix}, t_2 = 1\right\} \left\{\mathbf{p}_3 = \begin{bmatrix}1\\0\end{bmatrix}, t_3 = 1\right\} \left\{\mathbf{p}_4 = \begin{bmatrix}1\\1\end{bmatrix}, t_4 = 0\right\}$$

- Because the two categories are not linearly separable, a single-layer perceptron cannot perform the classification
- A two-layer network can solve the XOR problem
- There are many different multilayer solutions. One solution is to use two neurons in the first layer to create two decision boundaries
  - The first boundary separates $\mathbf{p}_1$ from the other patterns, and the second boundary separates $\mathbf{p}_4$

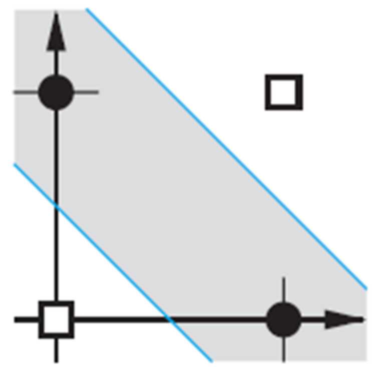
The XOR example

- Then the second layer is used to combine the two boundaries together using an AND operation

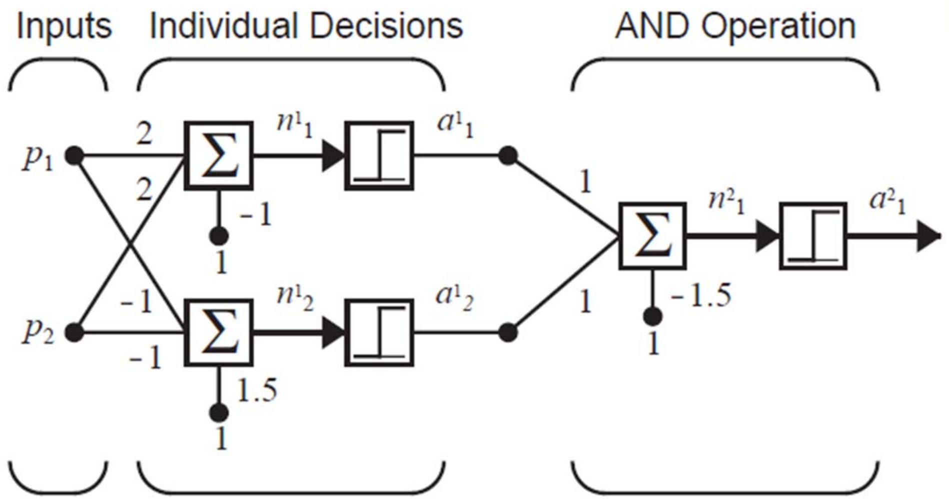


Layer 1/Neuron 1

Layer 1/Neuron 2

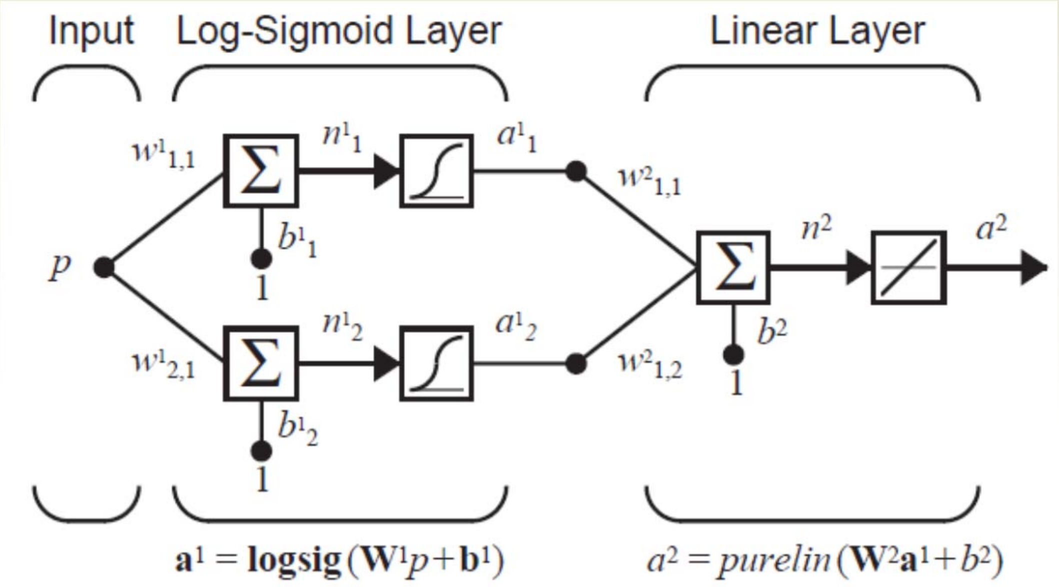


- The resulting 2-2-1 neural network is the following



The function approximation example

- In control systems, for example, the objective is to find an appropriate feedback function that maps from measured outputs to control inputs
- The following example illustrates the flexibility of the multilayer perceptron for implementing functions
- Consider the following two-layer 1-2-1 network
  - The transfer function for the first layer is log-sigmoid
  - the transfer function for the second layer is linear



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

$$f^2(n) = n$$

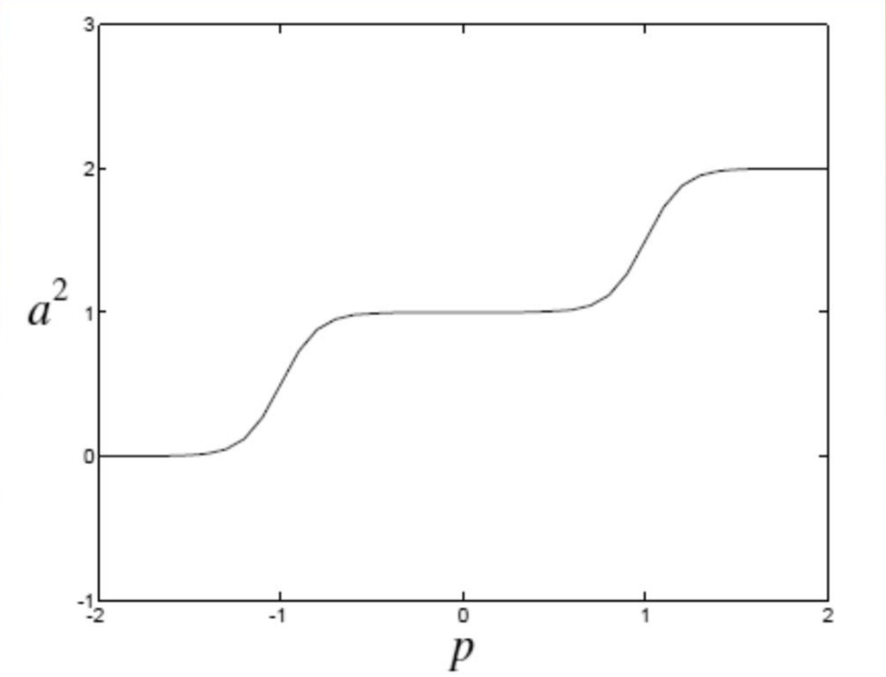
The function approximation example

- Suppose that the nominal values of the weights and biases for this network are: $w^1_{1,1} = 10, \quad w^1_{2,1} = 10, \quad b^1_1 = -10, \quad b^1_2 = 10$

$$w^2_{1,1} = 1, \quad w^2_{1,2} = 1, \quad b^2 = 0$$

- The network response for these parameters is shown below, which plots the network output a^2 as the input *p* is varied over the range [-2, 2]

- Notice that the response consists of two steps, one for each of the log-sigmoid neurons in the first layer. By adjusting the network parameters we can change the shape and location of each step



The function approximation example

- The centers of the steps occur where the net input to a neuron in the first layer is zero

$$n_1^1 = w_{1,1}^1 p + b_1^1 = 0 \Rightarrow p = -\frac{b_1^1}{w_{1,1}^1} = -\frac{-10}{10} = 1$$

$$n_2^1 = w_{2,1}^1 p + b_2^1 = 0 \Rightarrow p = -\frac{b_2^1}{w_{2,1}^1} = -\frac{10}{10} = -1$$

- The steepness of each step can be adjusted by changing the network weights

The function approximation example

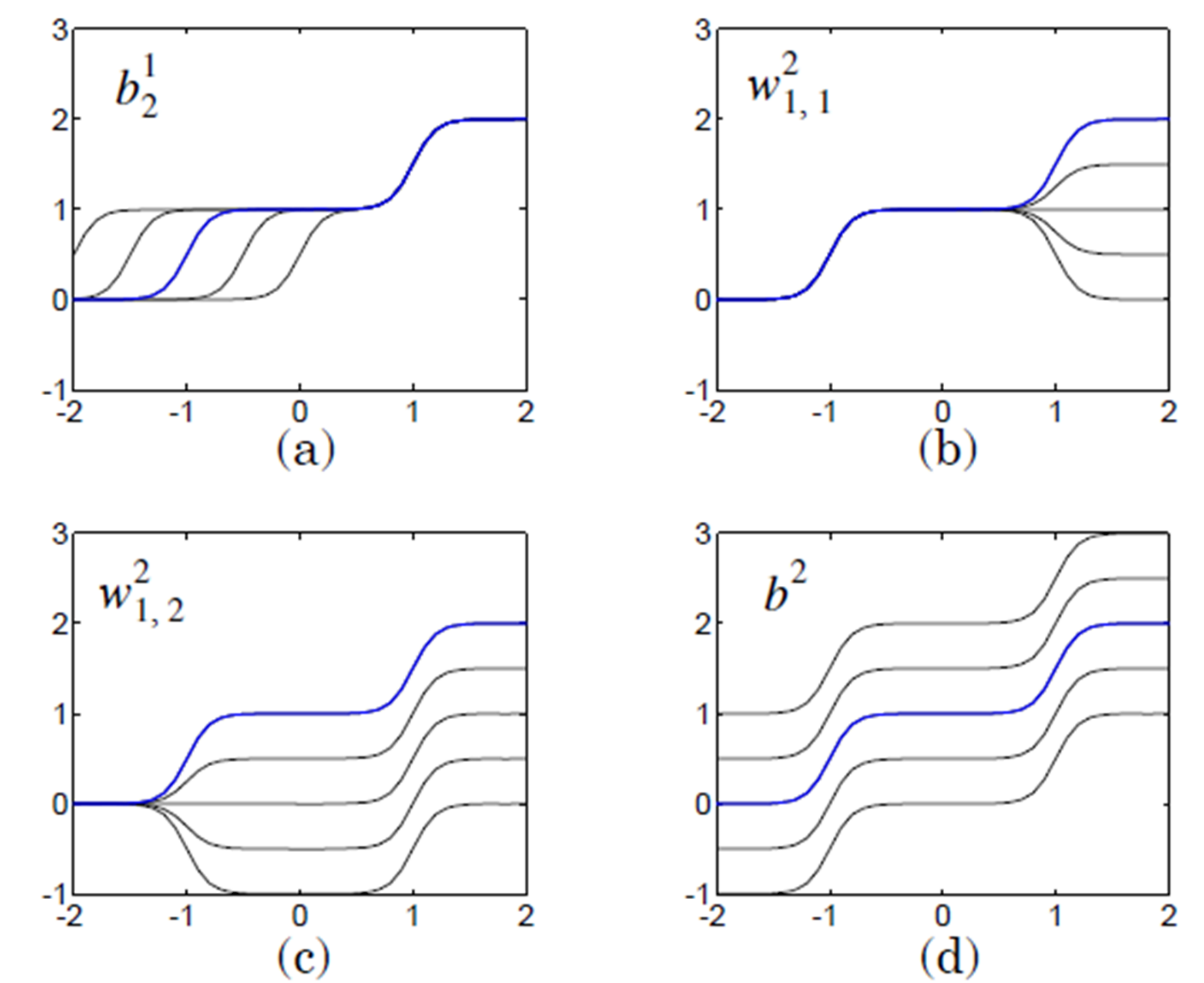
- See the effects of parameter changes on the network response
- The blue curve is the nominal response
- The other curves correspond to the network response when one parameter at a time is varied over the following ranges:

$$-1 \le w_{1,1}^2 \le 1$$

$$-1 \le w_{1,2}^2 \le 1$$

$$0 \le b_2^1 \le 20$$

$$-1 \le b^2 \le 1$$



The function approximation example

- Plot (a) shows how the network biases in the first (hidden) layer can
- be used to locate the position of the steps
- Plot (b) illustrates how the weights determine the slope of the steps
  - The bias in the second (output) layer shifts the entire network response up or down, as can be seen in Plot (d)

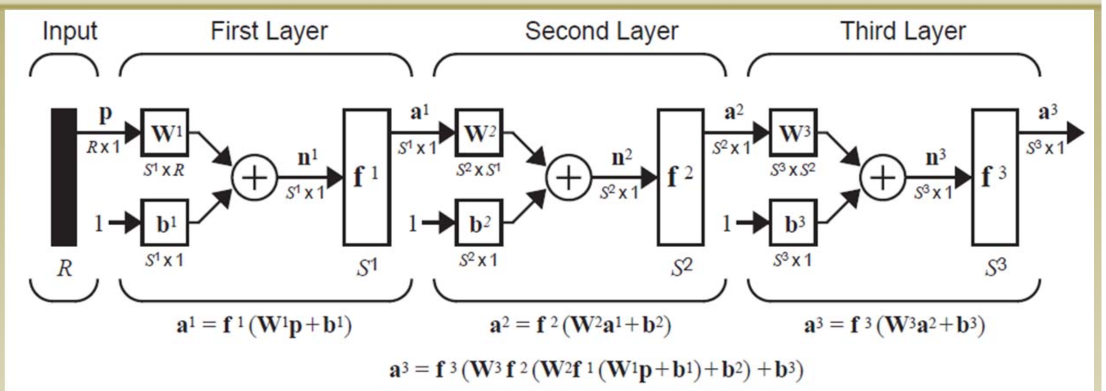
- It would appear that we could use such networks to approximate almost any function, if we had a sufficient number of neurons in the hidden layer.
- In fact, it has been shown that two-layer networks, with sigmoid transfer functions in the hidden layer and linear transfer functions in the output layer, can approximate virtually any function of interest to any degree of accuracy, provided sufficiently many hidden units are available [see the Universal Approximation Theorem in our textbook]

The Universal Approximation Theorem

- The Universal Approximation Theorem (https://www.sciencedirect.com/science/article/pii/0893608089900208) states that a feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function (such as the logistic sigmoid activation function) can approximate any Borel measurable function from one finite-dimensional space to another with any desired non-zero amount of error, provided that the network is given enough hidden units. The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well (article mentioned above)
- The concept of Borel measurability is beyond the scope of this course; for our purposes it suffices to say that any continuous function on a closed and bounded subset of R^{n} is Borel measurable and therefore may be approximated by a neural network. A neural network may also approximate any function mapping from any finite dimensional discrete space to another
- While the original theorems were first stated in terms of units with activation functions that saturate both for very negative and for very positive arguments, universal approximation theorems have also been proved for a wider class of activation functions, which includes the now commonly used REctified Linear Unit (https://www.sciencedirect.com/science/article/pii/S0893608005801315)

The Backpropagation algorithm

- In abbreviated notation



- For multilayer networks the output of one layer becomes the input to the following layer. The equations that describe this operation are

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^{m} + \mathbf{b}^{m+1}) \quad \text{for} \quad m = 0, 1, \ldots, M-1$$

where ***M*** is the number of layers in the network

The neurons in the first layer receive external inputs:

$$\mathbf{a}^0 = \mathbf{p}$$

which provides the starting point for the above equation. The outputs of the neurons in the last layer are considered the network outputs $\mathbf{a} = \mathbf{a}^{\mathbf{M}}$

Performance index

- The backpropagation algorithm for multilayer networks is a generalization of the LMS algorithm, and both algorithms use the same performance index: *mean square error*
- The algorithm is provided with a set of examples of proper network behavior : $\{\mathbf{p}_1, \mathbf{t}_1\}\ \{\mathbf{p}_2, \mathbf{t}_2\}\ \ldots\ \{\mathbf{p}_Q, \mathbf{t}_Q\}$
- The algorithm should adjust the network parameters in order to minimize the mean square error:

$$F(\mathbf{x}) = E[e^2] = E[(t\text{-}a)^2]$$

where is the vector of network weights and biases. If the network has multiple outputs this generalizes to:

$$F(\mathbf{x}) = E[\mathbf{e}^T\mathbf{e}] = E[(\mathbf{t}\text{-}\boldsymbol{\alpha})^{\mathrm{T}}(\mathbf{t}\text{-}\mathbf{a})]$$

- As with the LMS algorithm, we will approximate the mean square error by

$$\hat{F}(\mathbf{x}) = (\mathbf{t}(k)\text{-}\mathbf{a}(k))^{\mathrm{T}}(\mathbf{t}(k)\text{-}\mathbf{a}(k)) = \mathbf{e}^{\mathrm{T}}(k)\mathbf{e}(k)$$

where the expectation of the squared error has been replaced by the squared error at iteration *k*

Performance index

- The steepest descent algorithm for the approximate mean square error (stochastic gradient descent) is

$$w_{i,j}^{m}(k+1) = w_{i,j}^{m}(k) - \alpha \frac{\partial \hat{F}}{\partial w_{i,j}^{m}}$$

$$b_{i}^{m}(k+1) = b_{i}^{m}(k) - \alpha \frac{\partial \hat{F}}{\partial b_{i}^{m}}$$

where α is the learning rate

- So far, this development is identical to that for the LMS algorithm
- Now we come to the difficult part – the computation of the partial derivatives

Chain rule

- For a single-layer linear network (the ADALINE) these partial derivatives are conveniently computed
  - Recall the rule: $\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^T(k)$
- For the multilayer network the error is not an explicit function of the weights in the hidden layers, therefore these derivatives are not computed so easily
- Because the error is an indirect function of the weights in the hidden layers, we will use the chain rule of calculus to calculate the derivatives
- To review the chain rule, suppose that we have a function *f* that is an explicit function only of the variable *n*. We want to take the derivative of with respect to a third variable *w*. The chain rule is then

$$\frac{df(n(w))}{dw} = \frac{df(n)}{dn} \times \frac{dn(w)}{dw}$$

- For example: f(n)= e^n and n=2w, so that $f(n(w))= e^{2w}$, then:

$$\frac{df(n(w))}{dw} = \frac{df(n)}{dn} \times \frac{dn(w)}{dw} = (e^n)(2)$$

Chain rule

- We will use this concept to find the derivatives

$$\frac{\partial \hat{F}}{\partial w_{i,j}^{m}} = \frac{\partial \hat{F}}{\partial n_{i}^{m}} \times \frac{\partial n_{i}^{m}}{\partial w_{i,j}^{m}}$$

$$\frac{\partial \hat{F}}{\partial b_{i}^{m}} = \frac{\partial \hat{F}}{\partial n_{i}^{m}} \times \frac{\partial n_{i}^{m}}{\partial b_{i}^{m}}$$

- The second term in each of these equations can be easily computed, since the net input to layer *m* is an explicit function of the weights and bias in that layer:

$$n_{i}^{m} = \sum_{j=1}^{S^{m-1}} w_{i,j}^{m} a_{j}^{m-1} + b_{i}^{m}$$

- Therefore: $\frac{\partial n_{i}^{m}}{\partial w_{i,j}^{m}} = a_{j}^{m-1}, \quad \frac{\partial n_{i}^{m}}{\partial b_{i}^{m}} = 1$

- If we now define: $s_{i}^{m} = \frac{\partial \hat{F}}{\partial n_{i}^{m}}$

Chain rule

- (the *sensitivity* of to changes in the i-th element of the net input at layer), then the following equations

$$\frac{\partial \hat{F}}{\partial w_{i,j}^{m}} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^{m}} \qquad \frac{\partial \hat{F}}{\partial b_i^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m}$$

- can be simplified to

$$\frac{\partial \hat{F}}{\partial w_{i,j}^{m}} = s_i^m a_j^{m-1} \qquad \frac{\partial \hat{F}}{\partial b_i^m} = s_i^m$$

- We can now express the approximate steepest descent algorithm as

$$w_{i,j}^{m}(k+1) = w_{i,j}^{m}(k) - \alpha s_i^m a_j^{m-1}$$

$$b_i^m(k+1) = b_i^m(k) - \alpha s_i^m$$

Chain rule

- In matrix form this becomes

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha \mathbf{s}^m$$

where: $$\mathbf{s}^m \equiv \frac{\partial \hat{F}}{\partial \mathbf{n}^m} = \begin{bmatrix} \frac{\partial \hat{F}}{\partial n_1^m} \\ \frac{\partial \hat{F}}{\partial n_2^m} \\ \vdots \\ \frac{\partial \hat{F}}{\partial n_n^m} \end{bmatrix}$$

Backpropagating the sensitivities

- It now remains for us to compute the sensitivities $\mathbf{s}^m$, which requires another application of the chain rule
- It is this process that gives us the term backpropagation, because it describes a recurrence relationship in which the sensitivity at layer *m* is computed from the sensitivity at layer *m+1*
- To derive the recurrence relationship for the sensitivities, we will use the following *Jacobian* matrix:

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^{m}} \equiv \begin{bmatrix} \frac{\partial n_1^{m+1}}{\partial n_1^m} & \frac{\partial n_1^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_1^{m+1}}{\partial n_{S^m}^m} \\ \frac{\partial n_2^{m+1}}{\partial n_1^m} & \frac{\partial n_2^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_2^{m+1}}{\partial n_{S^m}^m} \\ \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_1^m} & \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_{S^m}^m} \end{bmatrix}$$

Backpropagating the sensitivities

- Next we want to find an expression for this matrix. Consider the i, j element of the matrix:

$$\frac{\partial n_i^{m+1}}{\partial n_j^m} = \frac{\partial\left(\sum_{l=1}^{S^m} w_{i,l}^{m+1} a_l^m + b_i^{m+1}\right)}{\partial n_j^m} = w_{i,j}^{m+1} \frac{\partial a_j^m}{\partial n_j^m} =$$

$$= w_{i,j}^{m+1} \frac{\partial f^m(n_j^m)}{\partial n_j^m} = w_{i,j}^{m+1} f'^{\,m}(n_j^m)$$

- Thus, the Jacobian matrix can be written: $\dfrac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} = \mathbf{W}^{m+1}\mathbf{F}'^{\mathbf{m}}(\mathbf{n}^{\mathbf{m}})$

where: $\mathbf{F}'^{\,m}(\mathbf{n}^m) = \begin{bmatrix} f'^{\,m}(n_1^m) & 0 & \dots & 0 \\ 0 & f'^{\,m}(n_2^m) & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \vdots & f'^{\,m}(n_{S^m}^m) \end{bmatrix}$

Backpropagating the sensitivities

- We can now write out the recurrence relation for the sensitivity by using the chain rule in matrix form:

$$\mathbf{s}^m = \frac{\partial \hat{F}}{\partial \mathbf{n}^m} = \left(\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m}\right)^T \frac{\partial \hat{F}}{\partial \mathbf{n}^{n+1}} = \mathbf{F}'^{\mathbf{m}}(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \frac{\partial \hat{F}}{\partial \mathbf{n}^{m+1}}$$

$$\Rightarrow \mathbf{s}^m = \mathbf{F}'^{\mathbf{m}}(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \mathbf{s}^{m+1}$$

- Now we can see where the backpropagation algorithm derives its name. The sensitivities are propagated backward through the network from the last layer to the first layer

$$\mathbf{s}^{M} \rightarrow \mathbf{s}^{M\text{-}1} \rightarrow \ \ldots \rightarrow \mathbf{s}^{2} \rightarrow \mathbf{s}^{1}$$

The final step

- It is worth emphasizing that the backpropagation algorithm uses the same approximate steepest descent technique that we used in the LMS algorithm. The only complication is that in order to compute the gradient we need to first backpropagate the sensitivities. The beauty of backpropagation is that we have a very efficient implementation of the chain rule
- We still have one more step to make in order to complete the backpropagation algorithm. We need the starting point, $\mathbf{s}^M$, for the recurrence relation. This is obtained at the final layer:

$$s_i^M = \frac{\partial \hat{F}}{\partial n_i^M} = \frac{\partial (\mathbf{t}-\mathbf{a})^T(\mathbf{t}-\mathbf{a})}{\partial n_i^M} = \frac{\partial \sum_{j=1}^{S^M}(t_j - a_j)^2}{\partial n_i^M} = -2(t_i - a_i)\frac{\partial a_i}{\partial n_i^M}$$

- Now since $\frac{\partial a_i}{\partial n_i^M} = \frac{\partial a_i^M}{\partial n_i^M} = \frac{\partial f^M(n_i^M)}{\partial n_i^M} = f'^M(n_i^M)$
- We can write: $s_i^M = -2(t_i - a_i)f'^M(n_i^M)$ or
- in matrix from $\mathbf{s}_i^M = -2\mathbf{F}'^{\mathbf{M}}(\mathbf{n}^M)(\mathbf{t}-\mathbf{a})$

Summary of backpropagation algorithm

- The first step is to propagate the input forward through the network:

$$\mathbf{a}^0 = \mathbf{p}$$

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}) \quad \text{for } m = 0, 1, \ldots, M-1$$

$$\mathbf{a} = \mathbf{a}^M$$

- The next step is to propagate the sensitivities backward through the network:

$$\mathbf{s}^M = -2\mathbf{F}'^{\mathbf{M}}(\mathbf{n}^M)(\mathbf{t} - \mathbf{a})$$

$$\mathbf{s}^m = \mathbf{F}'^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T\mathbf{s}^{m+1}, \quad \text{for} \quad m = M-1, \ldots, 2, 1$$

- Finally, the weights and biases are updated using the approximate steepest descent rule:

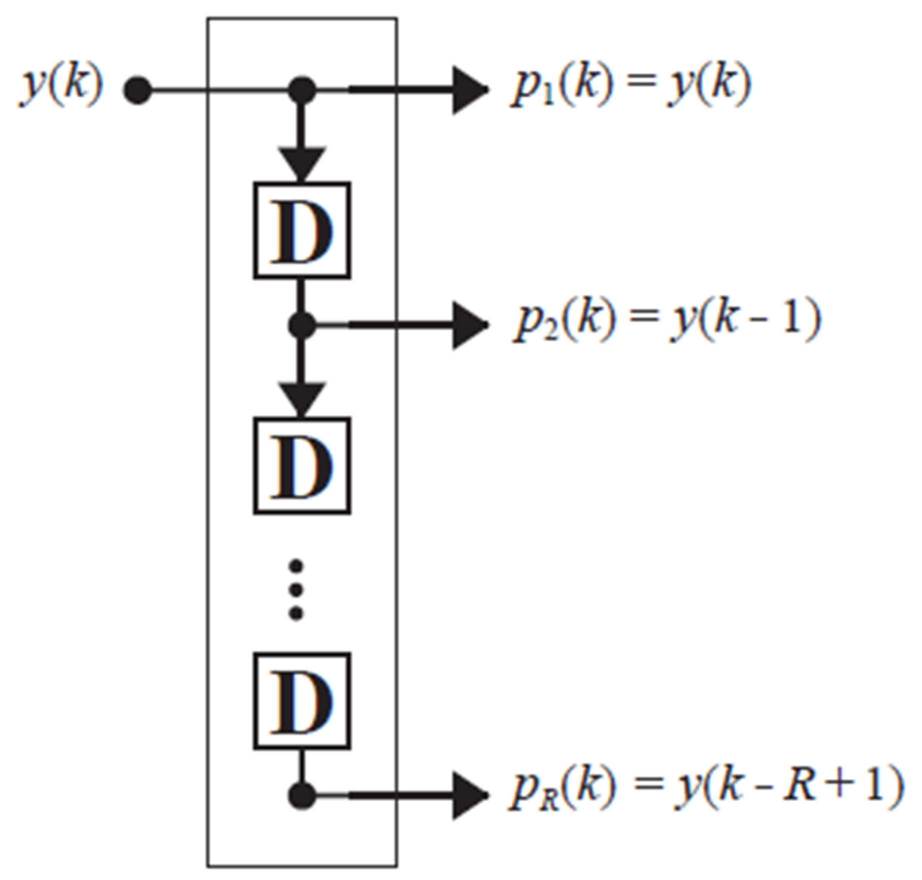
$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha\mathbf{s}^m(\mathbf{a}^{m-1})^T$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha\mathbf{s}^m$$

Practice on ADALINE

Tapped delay line

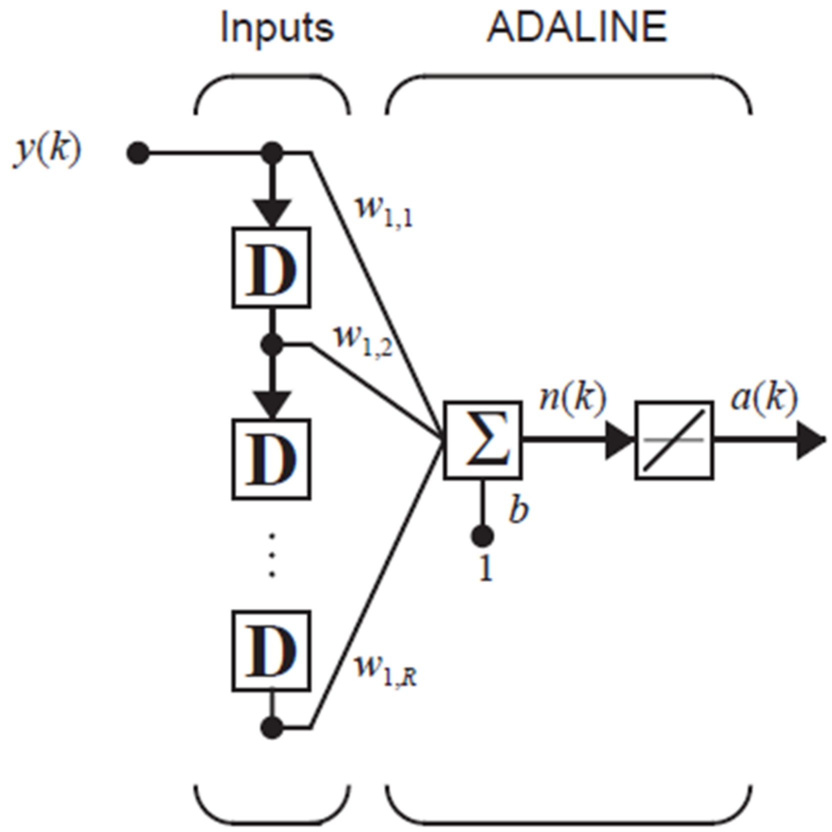
- ADALINE has been much more widely used than the perceptron network. In fact, it is safe to say that it is one of the most widely used neural networks in practical applications
- One of the major application areas of the ADALINE has been adaptive filtering, where it is still used extensively
- In order to use the ADALINE network as an adaptive filter, we need to introduce a new building block, the *tapped delay line*. A tapped delay line with R outputs is shown here



Adaptive filter: Tapped delay line+ADALINE

- If we combine a tapped delay line with an ADALINE network, we can create an *adaptive filter*
- You might have recognized this network as a finite impulse response (FIR) filter

$$a(k) = purelin(\mathbf{Wp}+b) = \sum_{i=1}^{R} w_{1,i}\,y(k-i+1)+b\,.$$



$$a(k) = purelin(\mathbf{Wp}(k)+b)$$

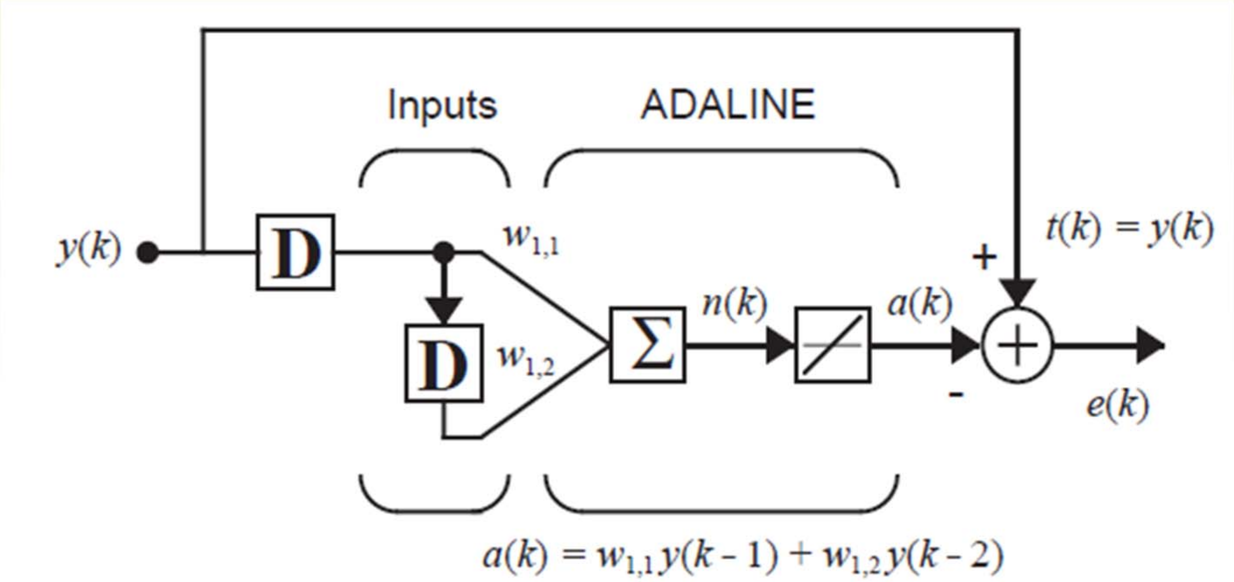
Exercise-03

Consider the adaptive filter ADALINE shown here. The purpose of this filter is to predict the next value of the input signal from the two previous values. Suppose that the input signal is a stationary random process, with autocorrelation function given by

$$C_y(n) = E[y(k)y(k+n)]$$

$$C_y(0) = 3,\ C_y(1) = -1,\ C_y(2) = -1$$

- ➢ Sketch the contour plot of the performance index (mean square error)
- ➢ What is the maximum stable value of the learning rate (α) for the LMS algorithm?


Inputs
ADALINE
$y(k)$
D
D
$w_{1,1}$
$w_{1,2}$
Σ
$n(k)$
$a(k)$
+
-
$t(k) = y(k)$
$e(k)$
$a(k) = w_{1,1}y(k-1) + w_{1,2}y(k-2)$