Multi-Layer Networks

&

Backpropagation

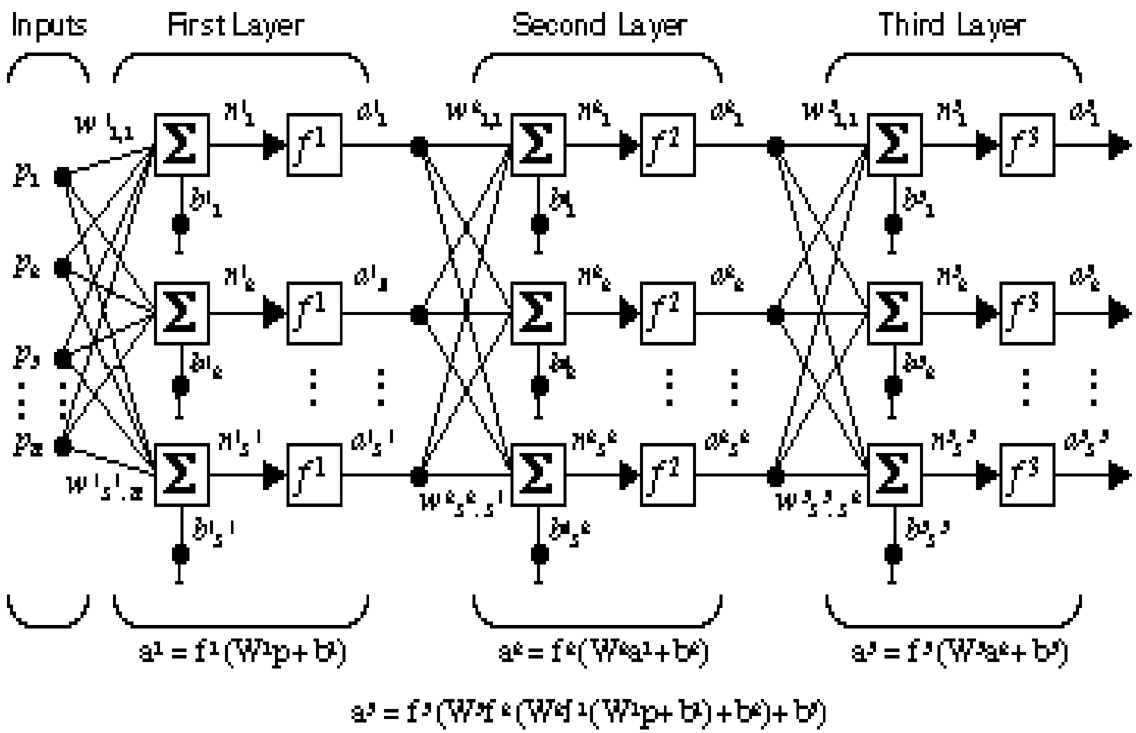
Multi-Layer Networks

- Generally much more versatile than single neurons
- No linear-separability requirement for problem space!
- Training approach is less obvious and potentially more time consuming.

Multi-Level Networks

- Several varieties, the most common of which is known as:
  - MLP (Multi-Level Perceptron)
  - Feed-forward network
  - Backpropagation Network (alluding to a common method of training these networks; other training methods could conceivably be used, so this is not a good name for the networks themselves.)

MLP Network



Note that sometimes the input is counted as a “layer”.

The real layers other than the output are called “hidden” layers.

Demo nnd11f

- Shows a simple 2-level network:
  - 1 input, 1 output
  - 2 neurons in first layer, with 1 weight and 1 bias each, logsig activation function
  - 1 neuron in output layer, with 2 weights and 1 bias
  - output activation function selectable from: purelin (identity), tansig, logsig
- Plot is network output vs. input

Function Approximation Demo nnd11f

Input Log-Sigmoid Layer Linear Layer

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

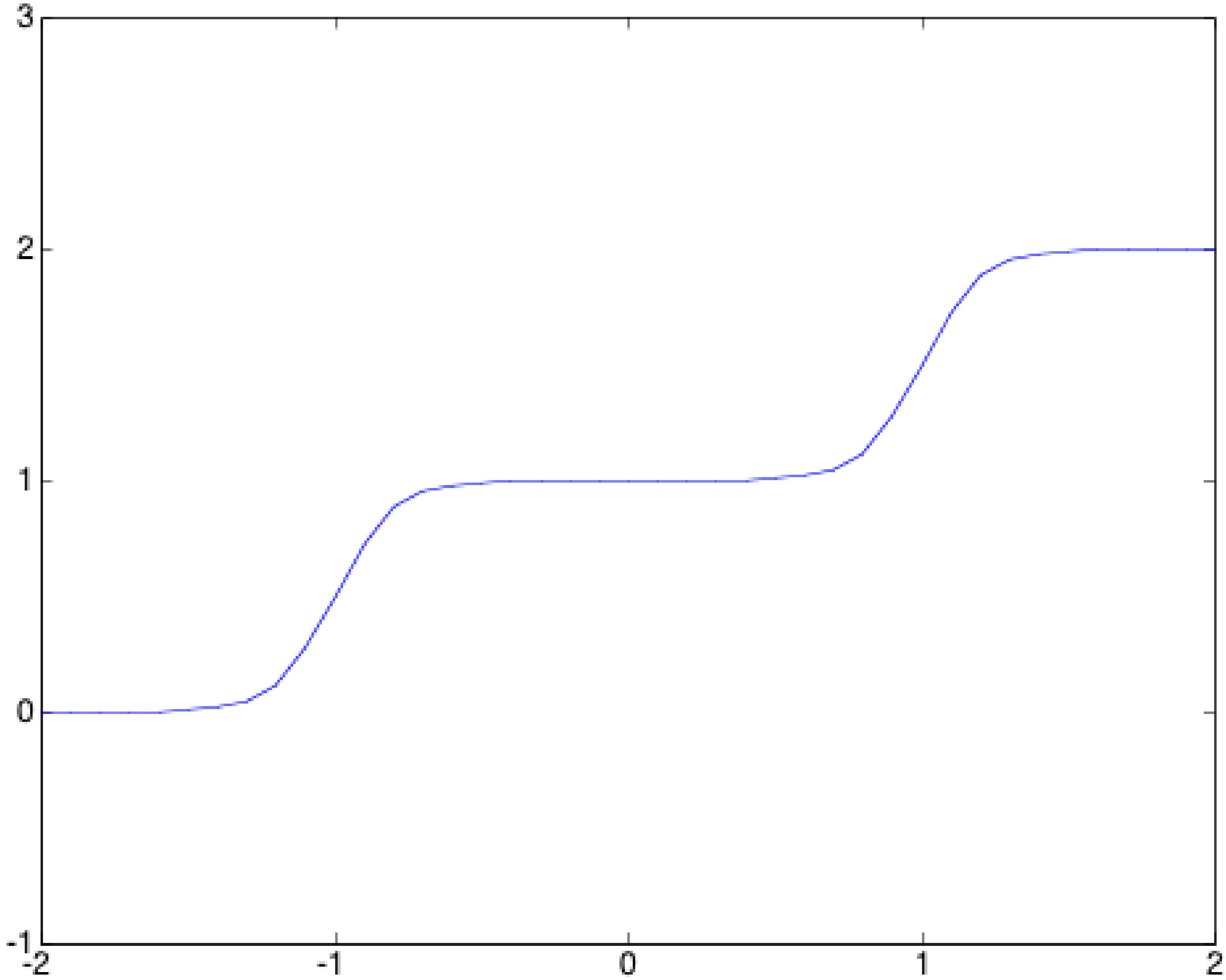
$$f^2(n) = n$$

Nominal Parameter Values

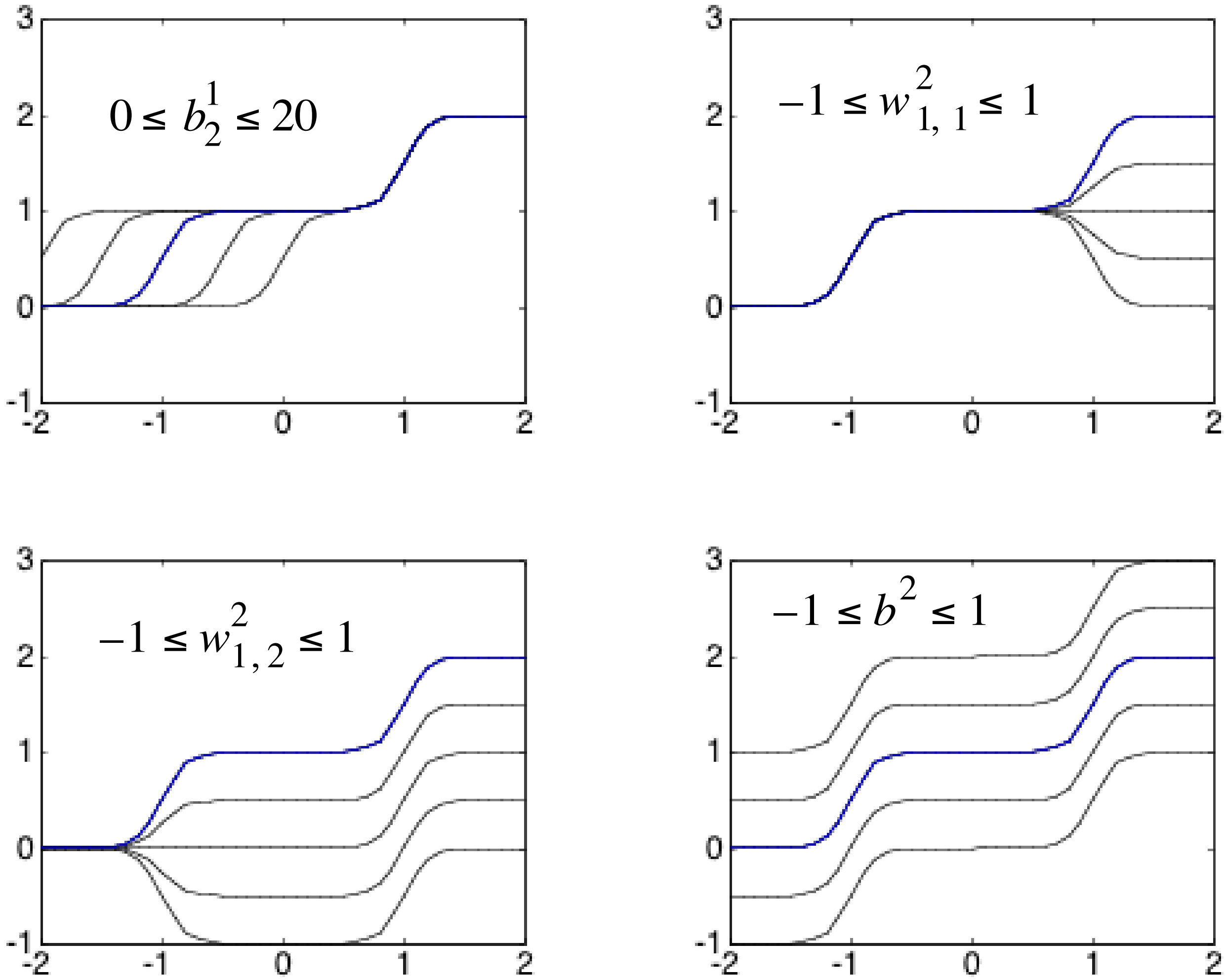
$$w^1_{1,1} = 10 \qquad w^1_{2,1} = 10 \qquad b^1_1 = -10 \qquad b^1_2 = 10$$

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

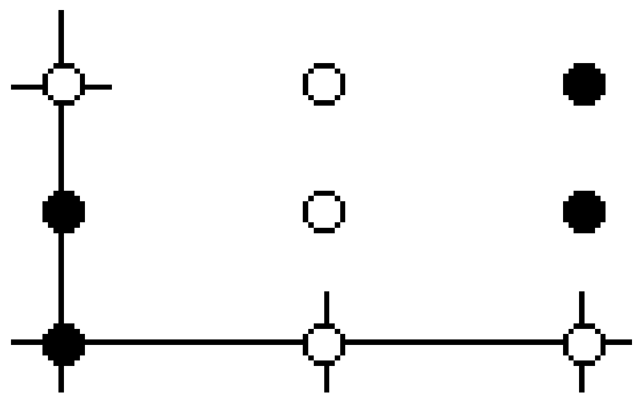
Nominal Response



Parameter Variations

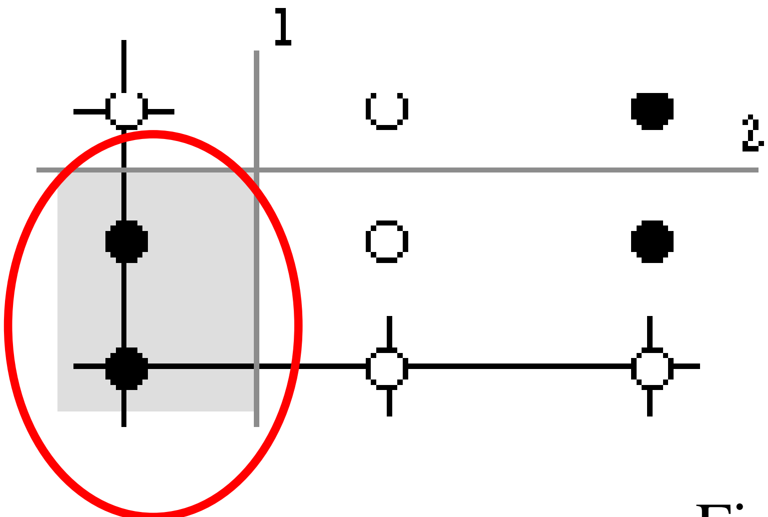


Constructing a Multi-Layer Classifier



Design a network by hand that implements this decision problem.

Elementary Decision Boundaries



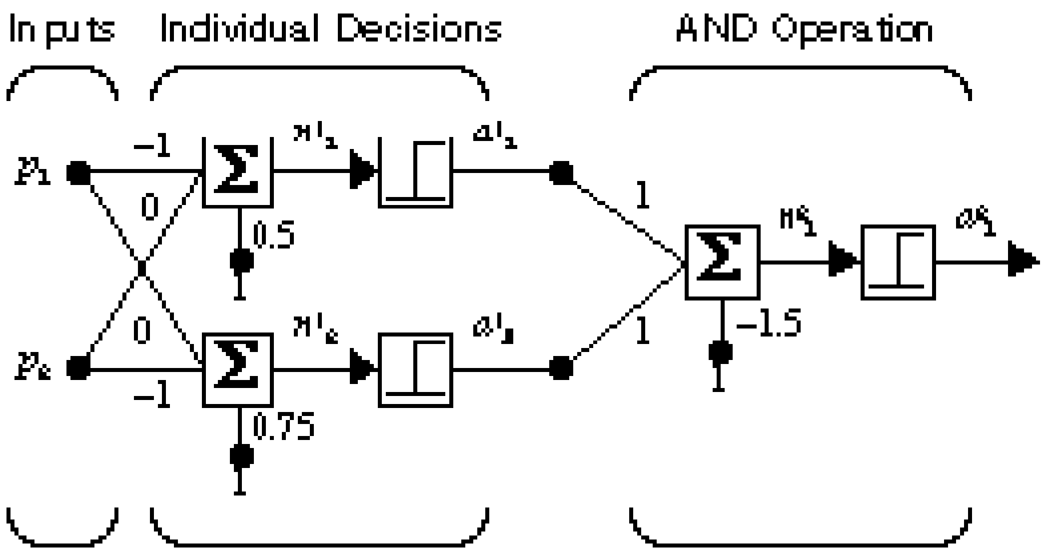
First Boundary:

$$a_1^1 = hardlim\left(\begin{bmatrix}-1 & 0\end{bmatrix}\mathbf{p} + 0.5\right)$$

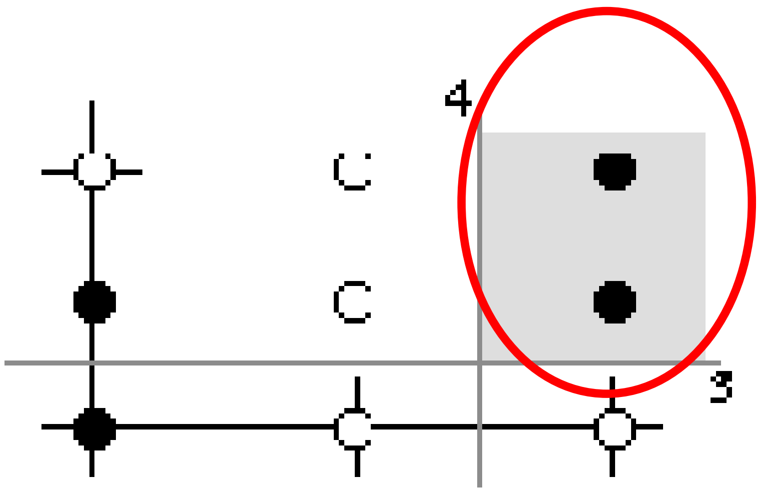
Second Boundary:

$$a_2^1 = hardlim\left(\begin{bmatrix}0 & -1\end{bmatrix}\mathbf{p} + 0.75\right)$$

First Subnetwork



Elementary Decision Boundaries



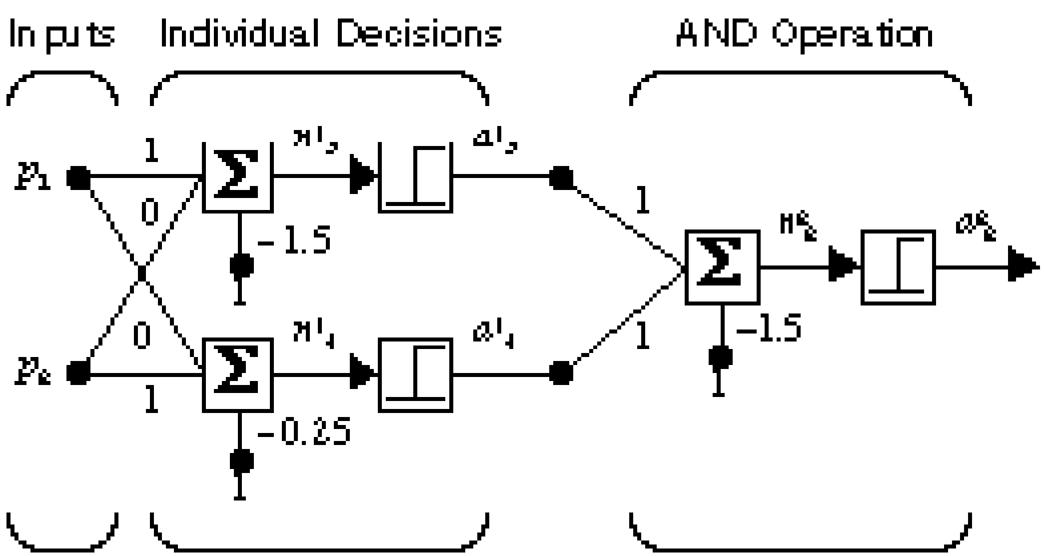
Third Boundary:

$$a_3^1 = hardlim(\begin{bmatrix}1 & 0\end{bmatrix}\mathbf{p} - 1.5)$$

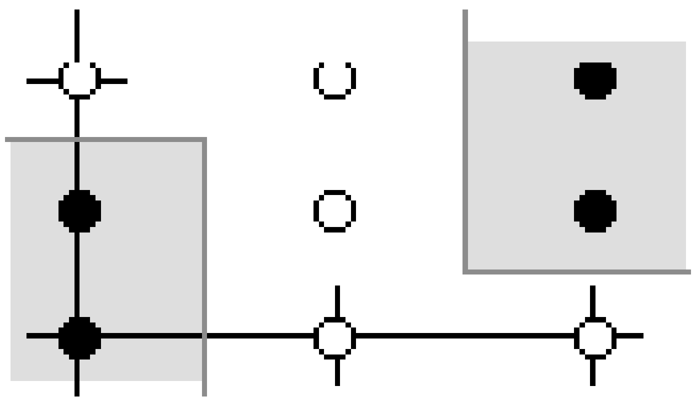
Fourth Boundary:

$$a_4^1 = hardlim(\begin{bmatrix}0 & 1\end{bmatrix}\mathbf{p} - 0.25)$$

Second Subnetwork



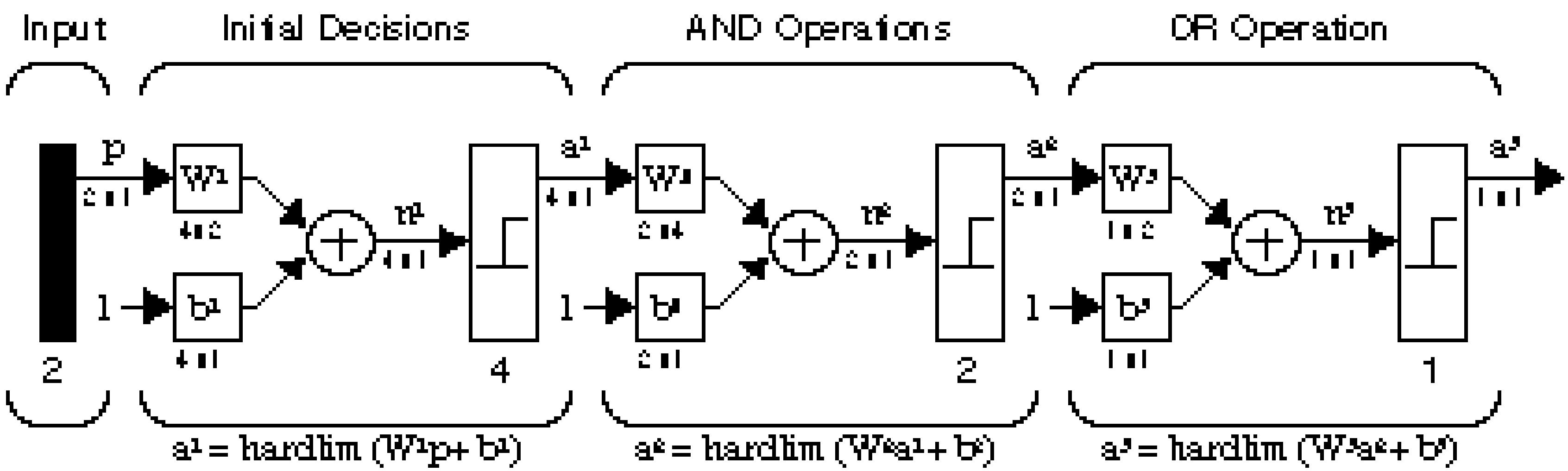
Total Network



$$\mathbf{W}^1 = \begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \mathbf{b}^1 = \begin{bmatrix} 0.5 \\ 0.75 \\ -1.5 \\ -0.25 \end{bmatrix}$$

$$\mathbf{W}^2 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \qquad \mathbf{b}^2 = \begin{bmatrix} -1.5 \\ -1.5 \end{bmatrix}$$

$$\mathbf{W}^3 = \begin{bmatrix} 1 & 1 \end{bmatrix} \qquad \mathbf{b}^3 = \begin{bmatrix} -0.5 \end{bmatrix}$$



How to Train a MLP?

- With a single neuron, it is not too hard to see how to adjust the weights based upon the error values. We've already seen a couple of ways.

- With a multi-layer network, it is less obvious. For one thing, **what is the "error" for the neurons in non-final layers**? Without these, we don't know how to adjust.

- This is called the "credit assignment" problem (maybe should be "blame assignment").

Backpropagation

- Werbos, in his Harvard PhD thesis in 1974 found a method, but it was not widely disseminated.
- Rumelhart and McClelland, in 1985, discovered the method, presumably independently, and popularized it under the current name.
- In mathematics, such methods are in the category of “optimization”.

Backpropagation

- The technique is gradient descent, as explained for Adalines.
- However, the computation of the gradient is less clear.

Backpropagation Training Cycle

- **Forward propagation**: Derive the activation values (the inputs to the activation functions) at each neuron, and the final output.
- **Compute the error** in the output.
- **Backpropagate** the error through the network to get "**sensitivities**" at each neuron. (The gradient approximation is derivable from the sensitivities.)
- Use the sensitivities to **derive weight changes**.
- Apply the weight changes.

Backpropagation Training Cycle

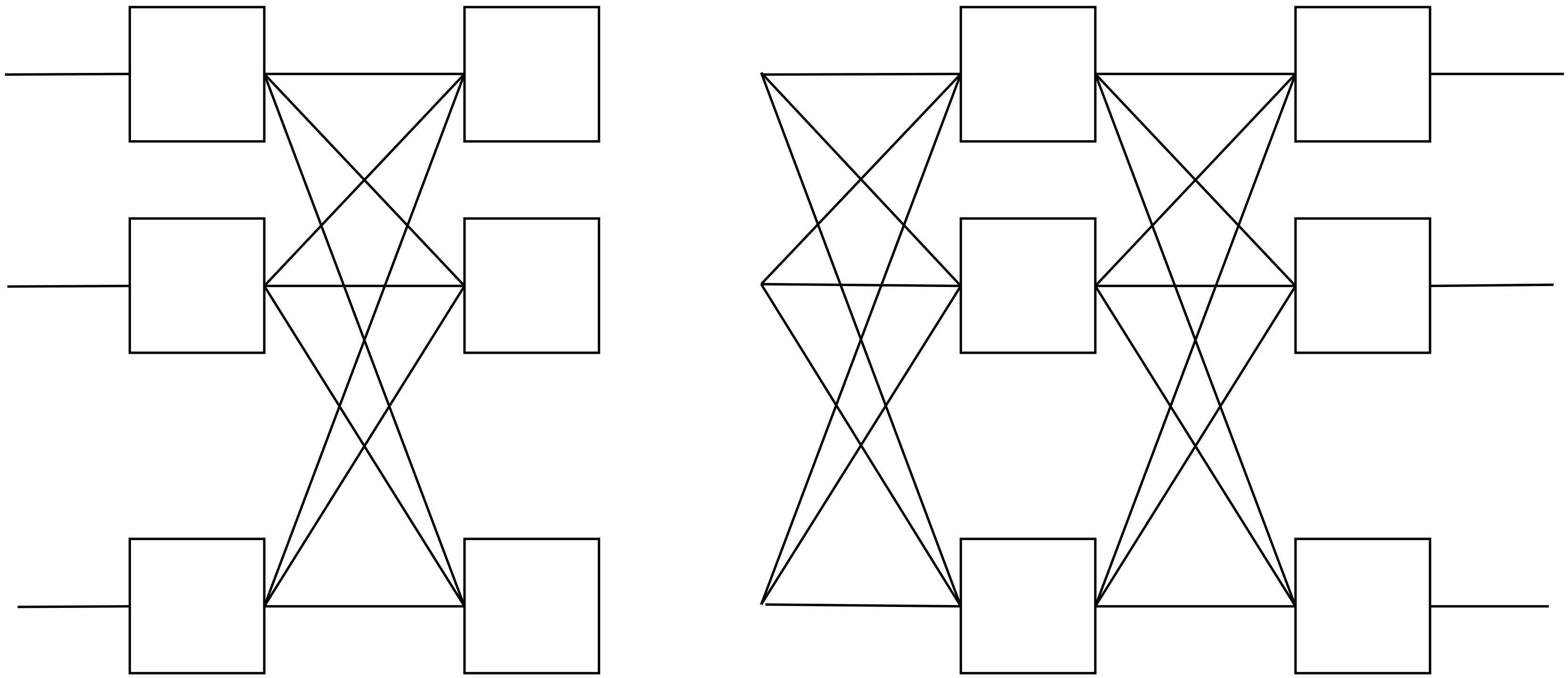
- Backpropagate is mathematically a lot like forward propagate.
- Sensitivities are used instead of signal values.
- The sensitivities are the partial derivatives of the MSE with respect to the activation values.
- Basically both are iterated matrix multiplications and applications of the activation functions of the neurons.

How to Adjust Weights?

- With a single neuron, it is not too hard to see how to adjust the weights based upon the error values. We've already seen a couple of ways.
- With a multi-layer network, it is less obvious. For one thing, what is the "error" for the neurons in non-final layers? Without these, we don't know how to adjust.
- This is called the "credit assignment" problem (maybe should be "blame assignment").

Multi-Layer Network

Each box has a row-vector of weights and a bias.



Each layer has a matrix of weights and a column vector of biases.

Multi-Layer Network

- Given an input vector, can compute the outputs.
- Given a sample, can compute the errors in output.
- Knowing gradient, can adjust the weights.
- Big Question:

  How to compute the gradient?

Multi-Layer Network

- Recall that the gradient consists of components

  $$\partial J/\partial w$$

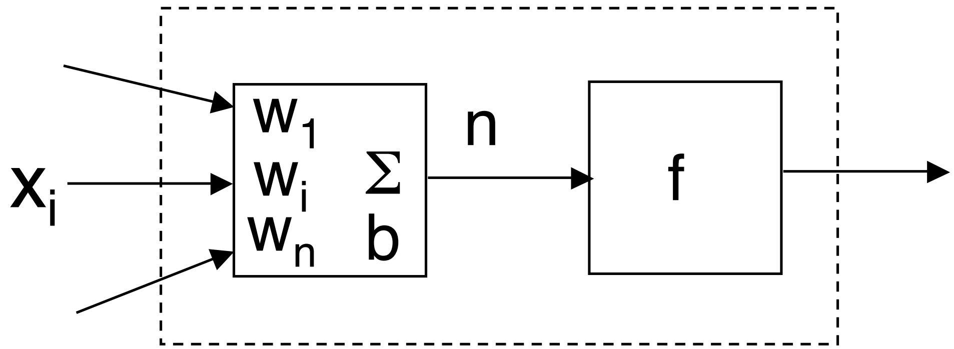
  where J is the mean-squared error and w is some weight (including bias) in the network.

- For the generalized Adaline, already derived:

  $$\partial J/\partial w_i = -2\, \varepsilon\, x_i\, f'(n),$$

  where x_i is the input corresponding to weight w_i, and v (**net**) is the weighted sum. This works **as is** for the multi-layer case at the **output** layer.

Inside one neuron



$\partial J/\partial w_i = (\partial J/\partial n)\ (\partial n/\partial w_i)$ chain rule

$= (\partial(d-f(n))^2/\partial v)\ (\partial n/\partial w_i)$

$= -2\ \varepsilon\ f'(n) \qquad x_i$

$= s\ x_i$

where $s = (\partial J/\partial n)$ is called the *sensitivity*

Chain Rule Refresher

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

Example

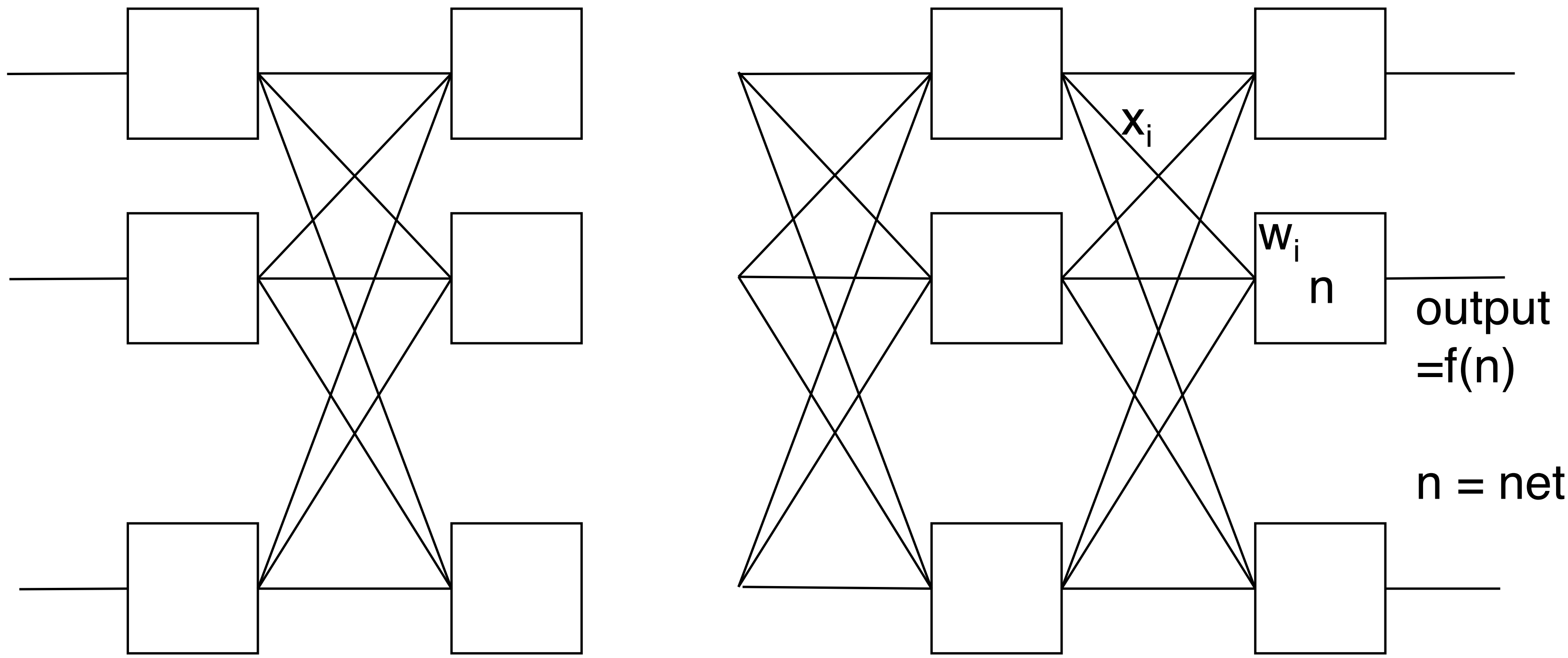
$$f(n) = \cos(n) \qquad n = e^{2w} \qquad f(n(w)) = \cos(e^{2w})$$

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

Application to Gradient Calculation

$$\frac{\partial J}{\partial w_{i,j}^{m}} = \frac{\partial J}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^m}$$

Last Layer

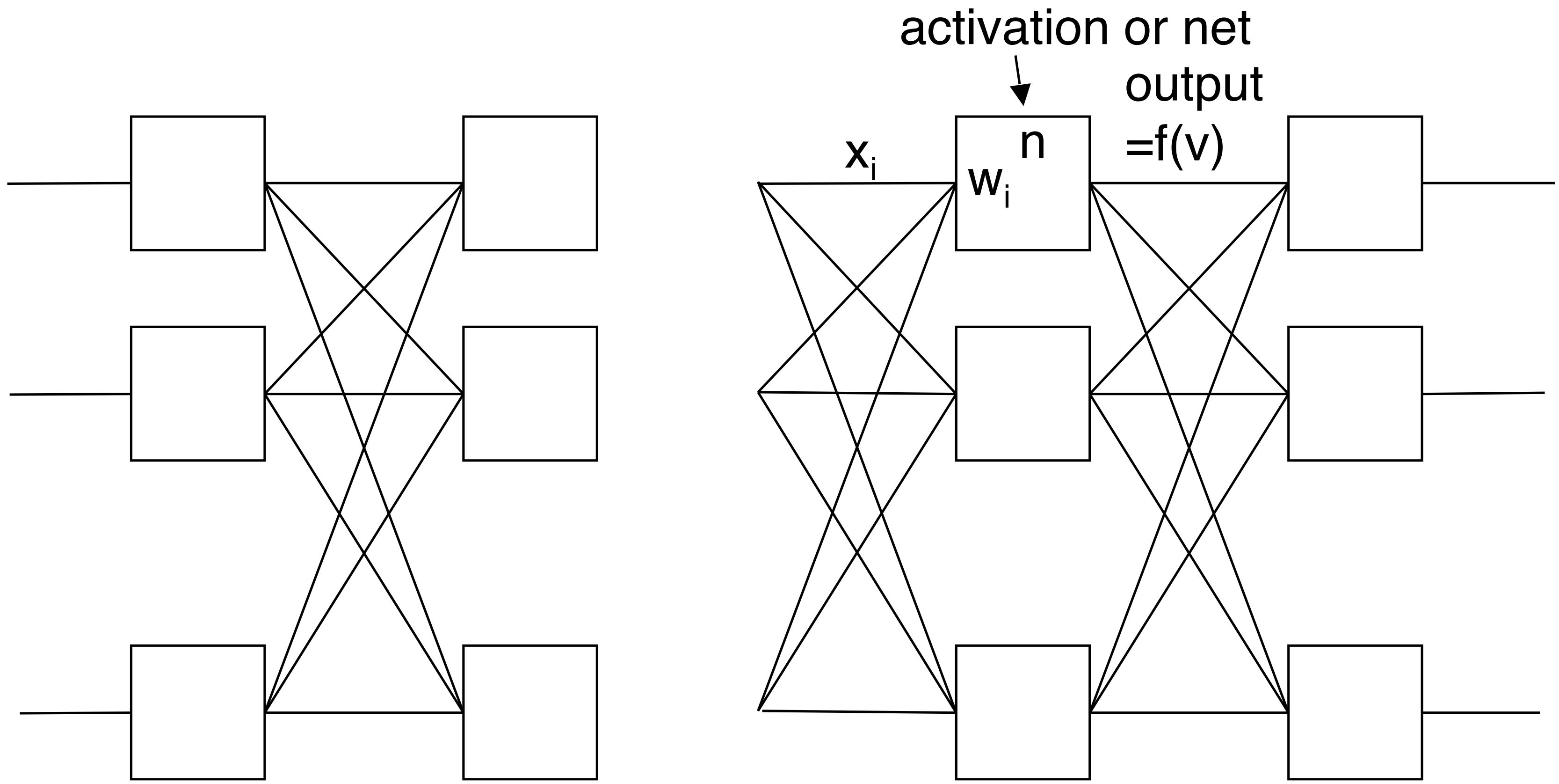


Utility of Sensitivity

$$\partial J/\partial w_i = (\partial J/\partial n)\ (\partial n/\partial w_i)$$
$$= s\ x_i$$

anywhere in the network, not just at the final layer.

e.g. Next-Last Layer



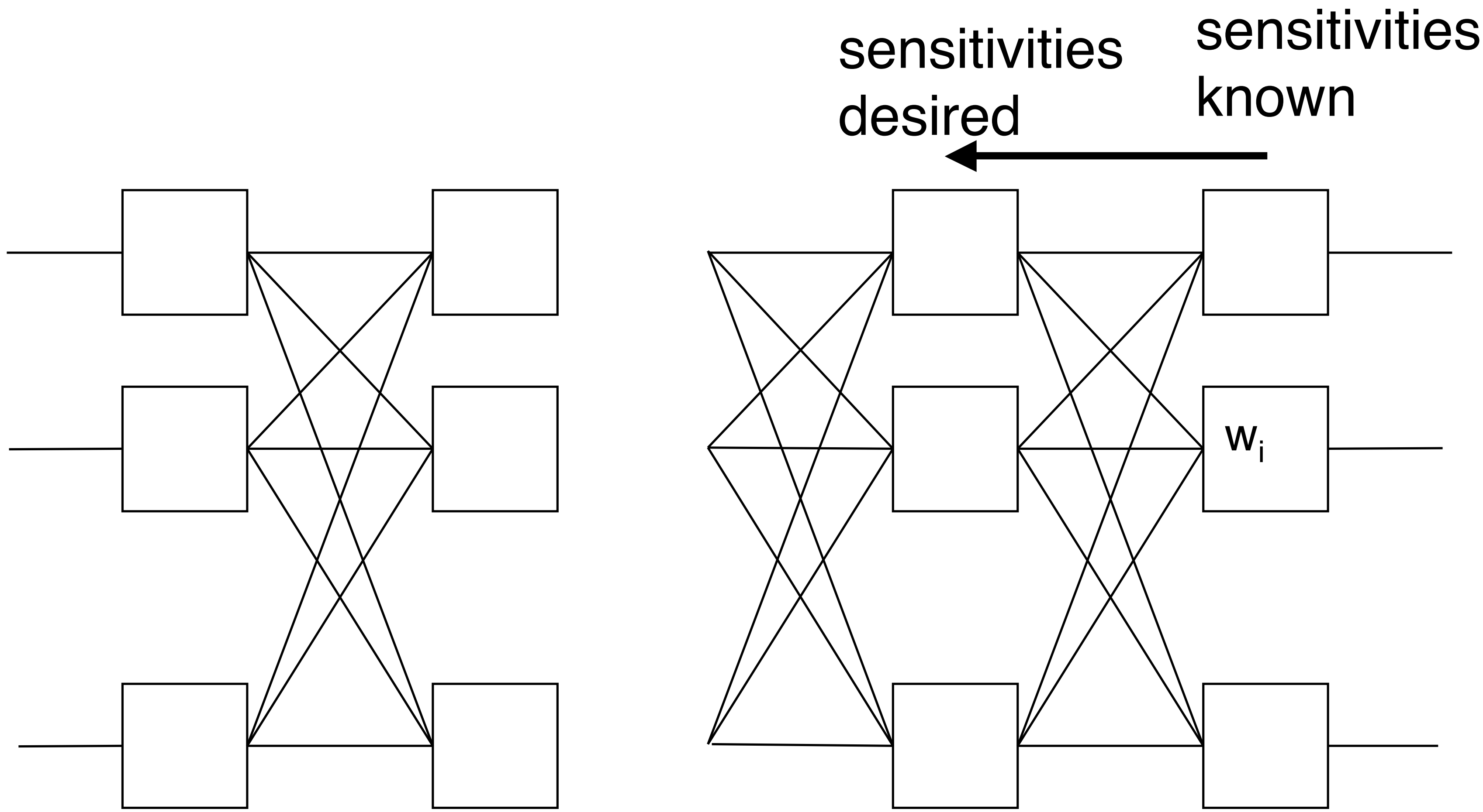
$$\partial J/\partial w_i = (\partial J/\partial n)\ (\partial n/\partial w_i)$$
$$= s\, x_i \quad \text{(s is “sensitivity”)}$$

Computing of Sensitivity

Unlike at the output, it is less clear how to compute the sensitivity at an ***arbitrary*** neuron.

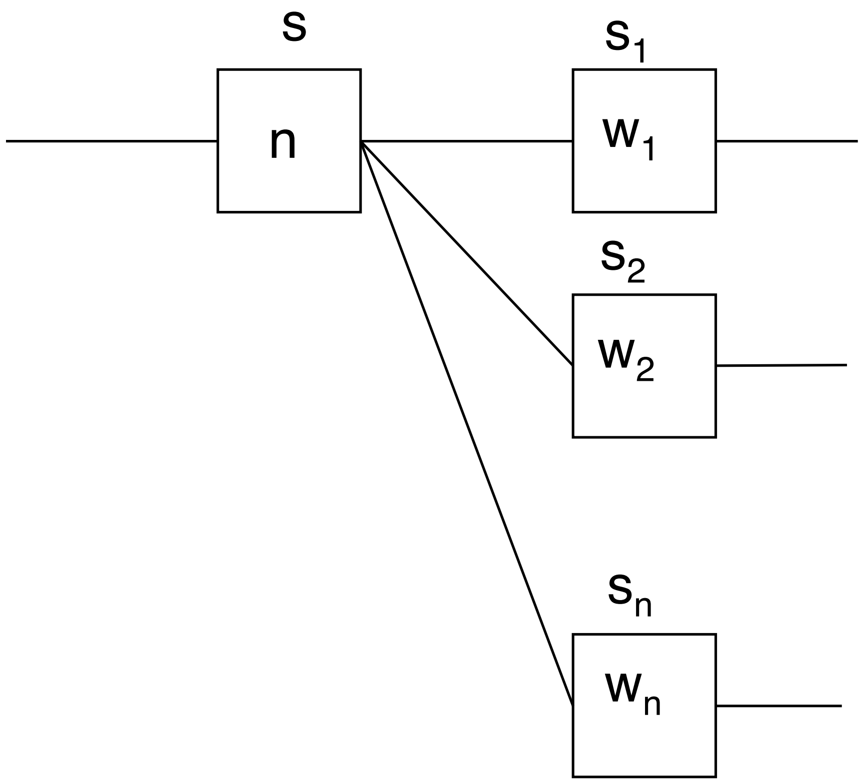
The key idea is to use an iterative approach that starts with the sensitivities at the last layer and works backward toward the first layer.

Backward Propagation of Sensitivity



Backward Propagation of Sensitivity

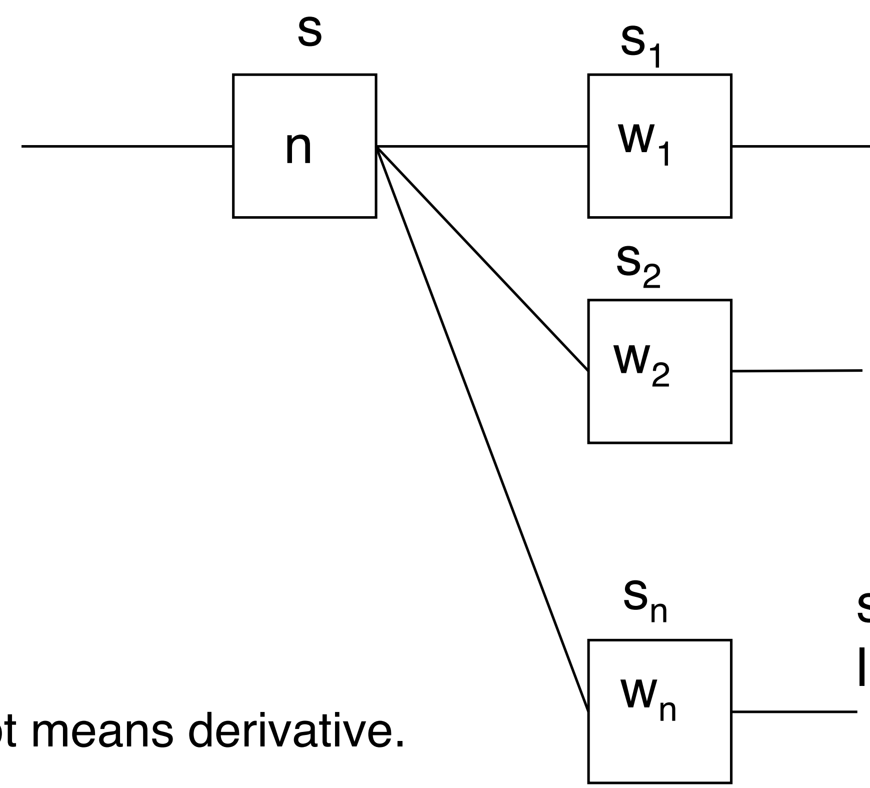
Express desired as a weighted sum of known:



$$s = f'(n)\ \Sigma\ w_j\, s_j$$

Vector Form

Express desired as a weighted sum of known:



$s = f'(n)\ \Sigma\ w_j\ s_j$

Note: over-dot means derivative.

sens layer m

derivs (diagonal matrix)

weight matrix **transposed!!**

sens layer m+1

Vector Form for entire *m*th layer: $\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T\mathbf{s}^{m+1}$

Correctness

Why should $s = f'(n)\ \Sigma\ w_j\ s_j$

$s = \partial J/\partial n^m$

$= \Sigma\ (\partial n^{m+1}/\partial n^m)\ (\partial J/\partial n^{m+1})$

from the *vector* form of the chain rule

$$\mathbf{s}^m = \frac{\partial J}{\partial \mathbf{n}^m} = \left(\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m}\right)^T \frac{\partial J}{\partial \mathbf{n}^{m+1}}$$

Vector Form for entire *m*th layer: $\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T\mathbf{s}^{m+1}$

Vector Form Expansion

In the vector form $\mathbf{s}^m = \frac{\partial J}{\partial \mathbf{n}^m} = \left(\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m}\right)^T \frac{\partial J}{\partial \mathbf{n}^{m+1}}$

derivatives times weights transposed

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

The s *subscripts* here are not related to sensitivities.

the “Jacobian”

Correctness, continued

$s = \Sigma\ (\partial n^{m+1}/\partial n^{m}) \qquad (\partial J/\partial n^{m+1})$

$= (\partial/\partial n^{m})\ \Sigma\ w_j f(n_j^{m}) \qquad s_j$ by def. of n^{m+1}

$= \Sigma\ w_j\ f'(n^{m})\ s_j$

$= f'(n^{m})\ \Sigma\ w_j\ s_j$

Vector Form

$$\mathbf{s}^m = \frac{\partial J}{\partial \mathbf{n}^m} = \left(\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m}\right)^T \frac{\partial J}{\partial \mathbf{n}^{m+1}} = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \frac{\partial J}{\partial \mathbf{n}^{m+1}}$$

$$\boxed{\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \mathbf{s}^{m+1}}$$

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} = \mathbf{W}^{m+1}\dot{\mathbf{F}}^m(\mathbf{n}^m)$$

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

Fully-Subscripted Alternatives to the Vector Forms

$$n_i^m = \sum_{j=1}^{S^{m-1}} w_{i,j}^m a_j^{m-1} + b_i^m$$

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

Sensitivity

$$s_i^m \equiv \frac{\partial J}{\partial n_i^m}$$

Gradient

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

Fully-Subscripted Alternatives to the Vector Forms

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

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

$$\dot{f}^m(n_j^m) = \frac{\partial f^m(n_j^m)}{\partial n_j^m}$$

Vector Form for Last Layer, M

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

$$\frac{\partial a_i}{\partial n_i^M} = \frac{\partial a_i^M}{\partial n_i^M} = \frac{\partial f^M(n_i^M)}{\partial n_i^M} = \dot{f}^M(n_i^M)$$

$$s_i^M = -2(t_i - a_i)\dot{f}^M(n_i^M)$$

$$\boxed{\mathbf{s}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t}-\mathbf{a})}$$

Backpropagation Training Cycle

- **Forward propagation**: Derive the activation values (the inputs to the activation functions) at each neuron, and the final output.
- **Compute the error** in the output.
- **Backpropagate** the error through the network to get "sensitivities" at each neuron. (The gradient approximation is derivable from the sensitivities.)
- Use the sensitivities to **derive weight changes**.
- Apply the weight changes.

Backpropagation (Sensitivities)

The sensitivities are computed by starting at the last layer, and then propagating backwards through the network to the first layer.

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

$$\mathbf{s}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t}-\mathbf{a}) \longleftarrow \text{basis}$$

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T\mathbf{s}^{m+1} \longleftarrow \text{induction step}$$

diagonal matrix of activation function derivative values

$$\dot{\mathbf{F}}^m(\mathbf{n}^m) = \begin{bmatrix} \dot{f}^m(n_1^m) & 0 & \ldots & 0 \\ 0 & \dot{f}^m(n_2^m) & \ldots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & \dot{f}^m(n_{S^m}^m) \end{bmatrix}$$

Weight and Bias Update

(Here we are using α instead of η for the learning rate.)

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

senstitivities of layer m

input of layer m = output of layer m-1

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

Fully-Subscripted Version of Weight Update

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

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

$$\mathbf{s}^m \equiv \frac{\partial J}{\partial \mathbf{n}^m} = \begin{bmatrix} \dfrac{\partial J}{\partial n_1^m} \\ \dfrac{\partial J}{\partial n_2^m} \\ \vdots \\ \dfrac{\partial J}{\partial n_{S^m}^m} \end{bmatrix}$$

Summary

Forward Propagation

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

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m) \qquad m = 0, 2, \ldots, M-1$$

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

Backpropagation

$$\mathbf{s}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t}-\mathbf{a})$$

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T\mathbf{s}^{m+1} \qquad m = M-1, \ldots, 2, 1$$

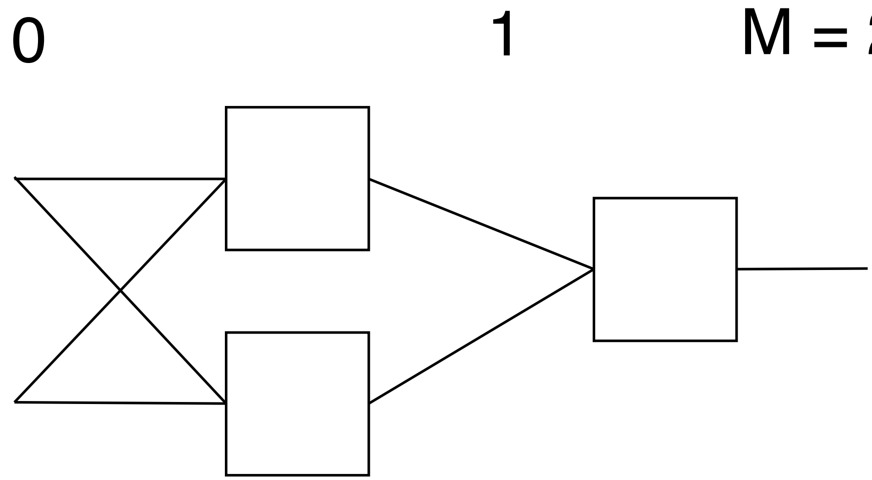
Weight Update

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

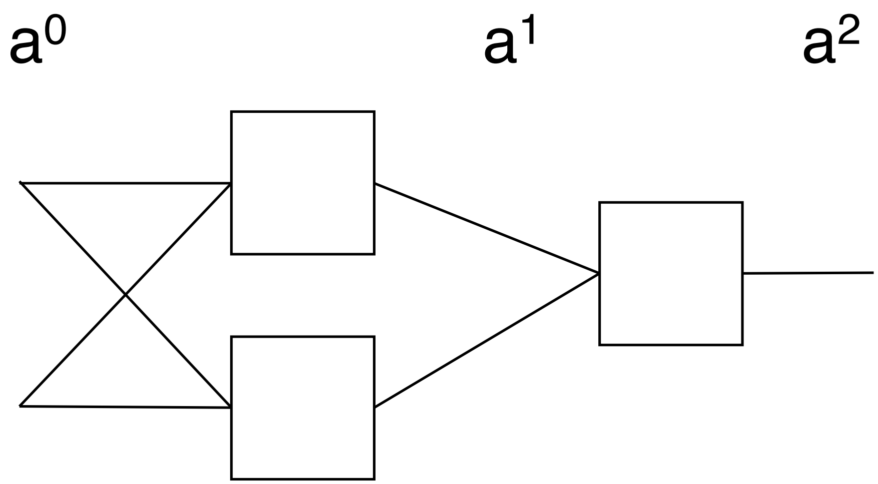
Exercise

- Derive the backprop equations symbolically for a simple network.
- Then use the equations to train the network.

Label the Levels

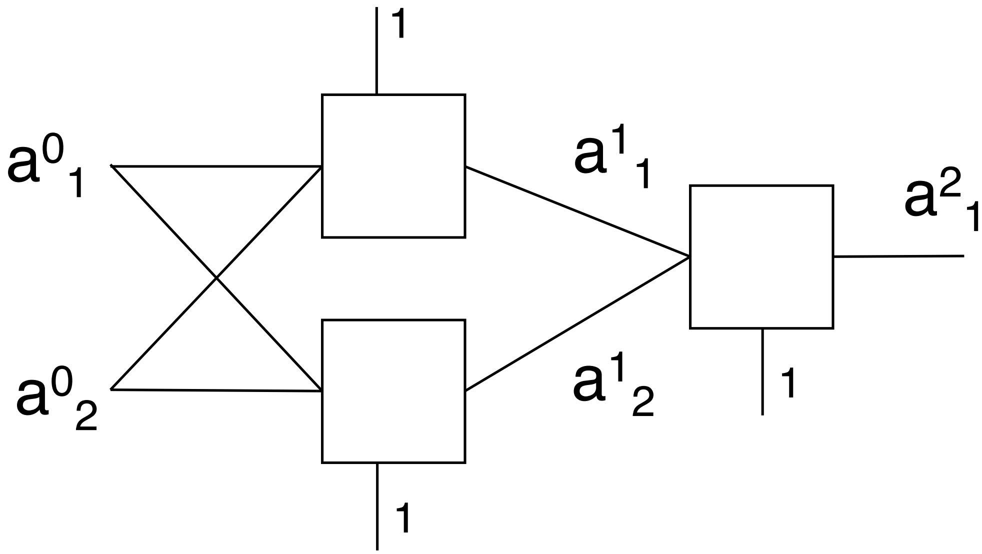


Label the Signal Vectors or Lines



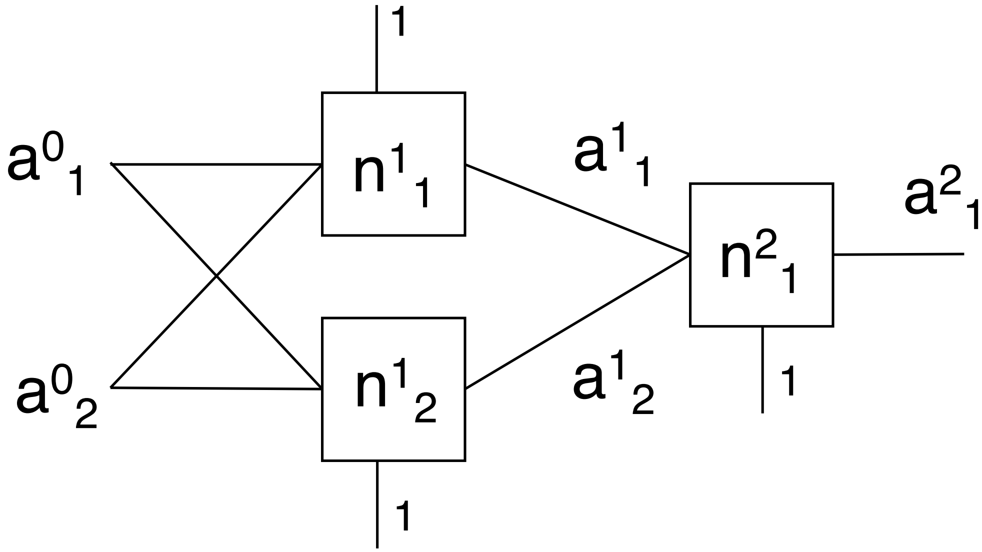
Vectors, superscript = level

Label the Signal Vectors or Lines

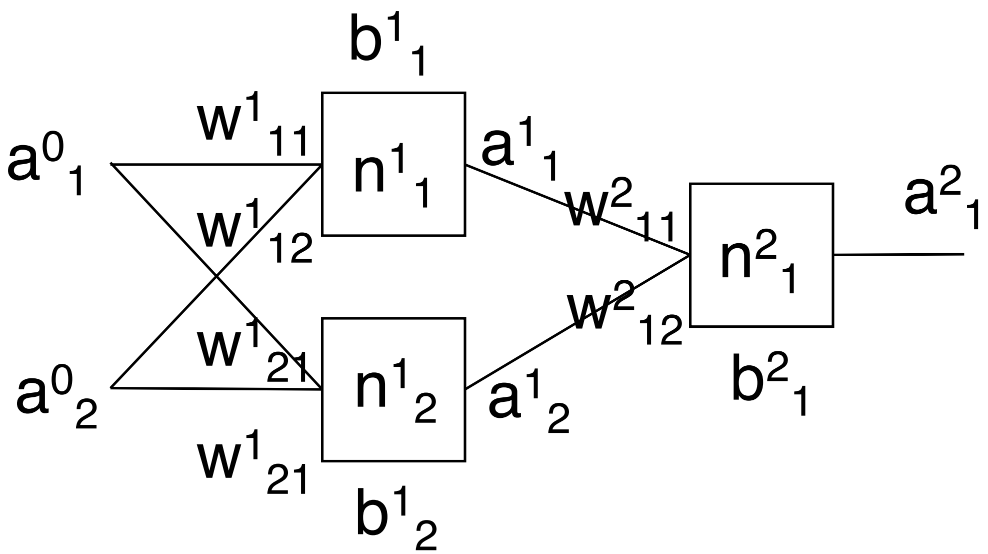


Lines, superscript = level,

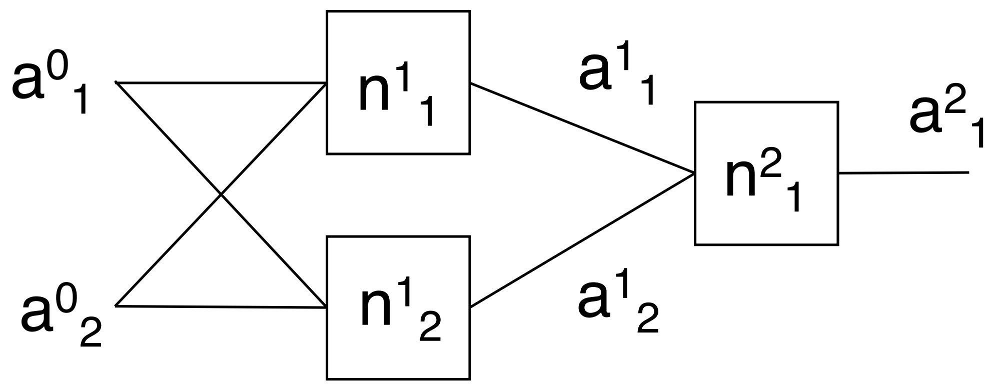
Label the Net (Activation) Values



Label the Weights and Biases



Write the forward equations for activations

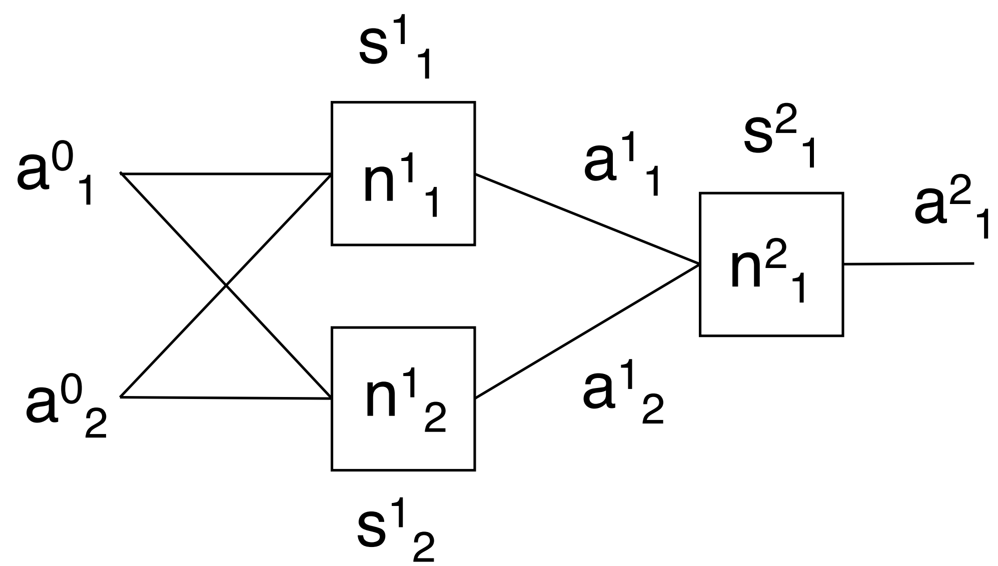


$$n^1_1 = w^1_{11}\, a^0_1 + w^1_{12}\, a^0_2 + b^1_1$$
$$a^1_1 = f(n^1_1)$$

$$n^1_2 = w^1_{21}\, a^0_1 + w^1_{22}\, a^0_2 + b^1_2$$
$$a^1_2 = f(n^1_2)$$

$$n^2_1 = w^2_{11}\, a^1_1 + w^2_{12}\, a^1_2 + b^2_1$$
$$a^2_1 = f(n^2_1)$$

Write the backward equations for sensitivities



$s^1_1 = w^2_{11}\, s^2_1\, f'(n^1_1)$

$s^2_1 = -2(d^2_1 - a^2_1)\, f'(n^2_1)$

actual

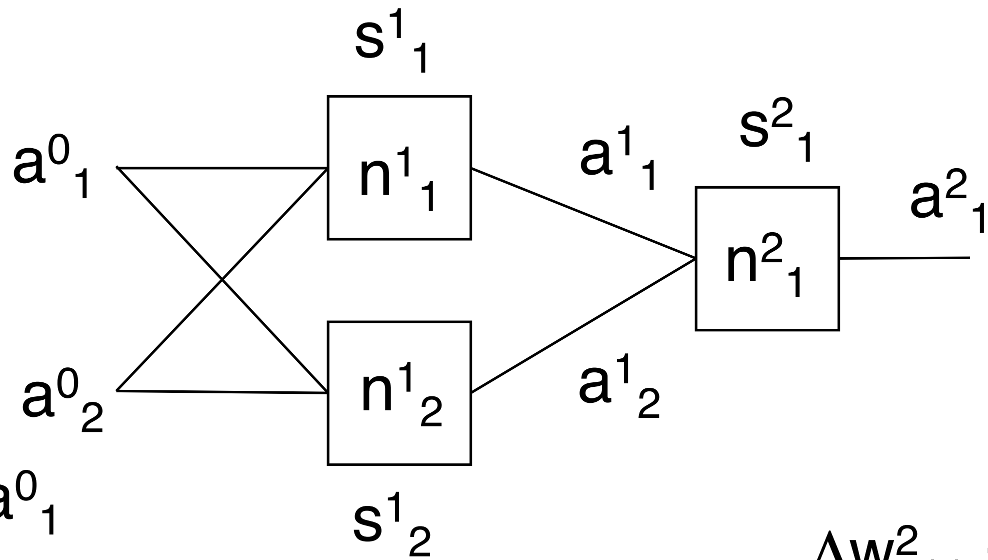
desired

$s^1_2 = w^2_{12}\, s^2_1\, f'(n^1_2)$

Note

- The summations for the backpropagated senstivities have only one term in this example, since the following layer has only one neuron.

Write the Equations for Weight and Bias Update



$$\Delta w^1_{11} = -\alpha s^1_1 \, a^0_1$$
$$\Delta w^1_{12} = -\alpha s^1_1 \, a^0_2$$
$$\Delta b^1_1 = -\alpha s^1_1$$

$$\Delta w^1_{21} = -\alpha s^1_2 \, a^0_1$$
$$\Delta w^1_{22} = -\alpha s^1_2 \, a^0_2$$
$$\Delta b^1_2 = -\alpha s^1_2$$

$$\Delta w^2_{11} = -\alpha s^2_1 \, a^1_1$$
$$\Delta w^2_{12} = -\alpha s^2_1 \, a^1_2$$
$$\Delta b^2_1 = -\alpha s^2_1$$

Exercises

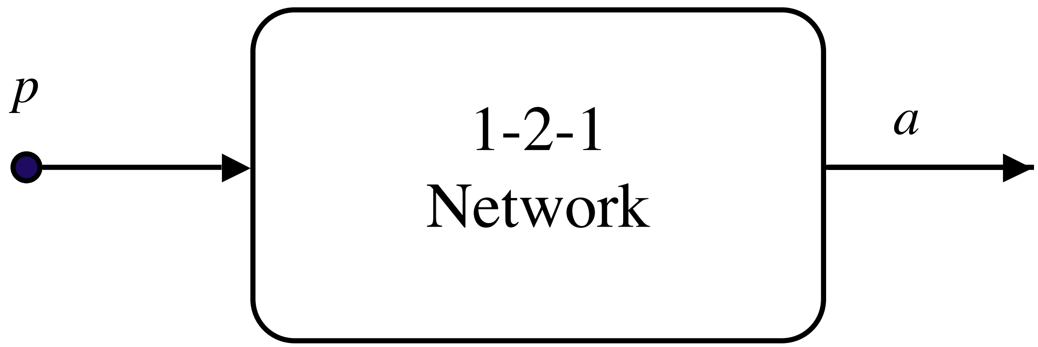
- Use the derivation to train the network to realize xor, where the first layer activations are logsig and the second layer is a hardlim.
- Carry out the preceding type of derivation for a 2-3-2 network (2 inputs, 3 middle neurons, 2 outputs).

Note on Training vs. Use

- Discontinuous functions such as hardlim, hardlims, etc. don’t have derivatives.
- Therefore we train the network with continuous approximations to these functions, then replace them with the discontinuous versions during usage:

<u>usage</u>	<u>train with</u>
hardlim	logsig
hardlims	tansig

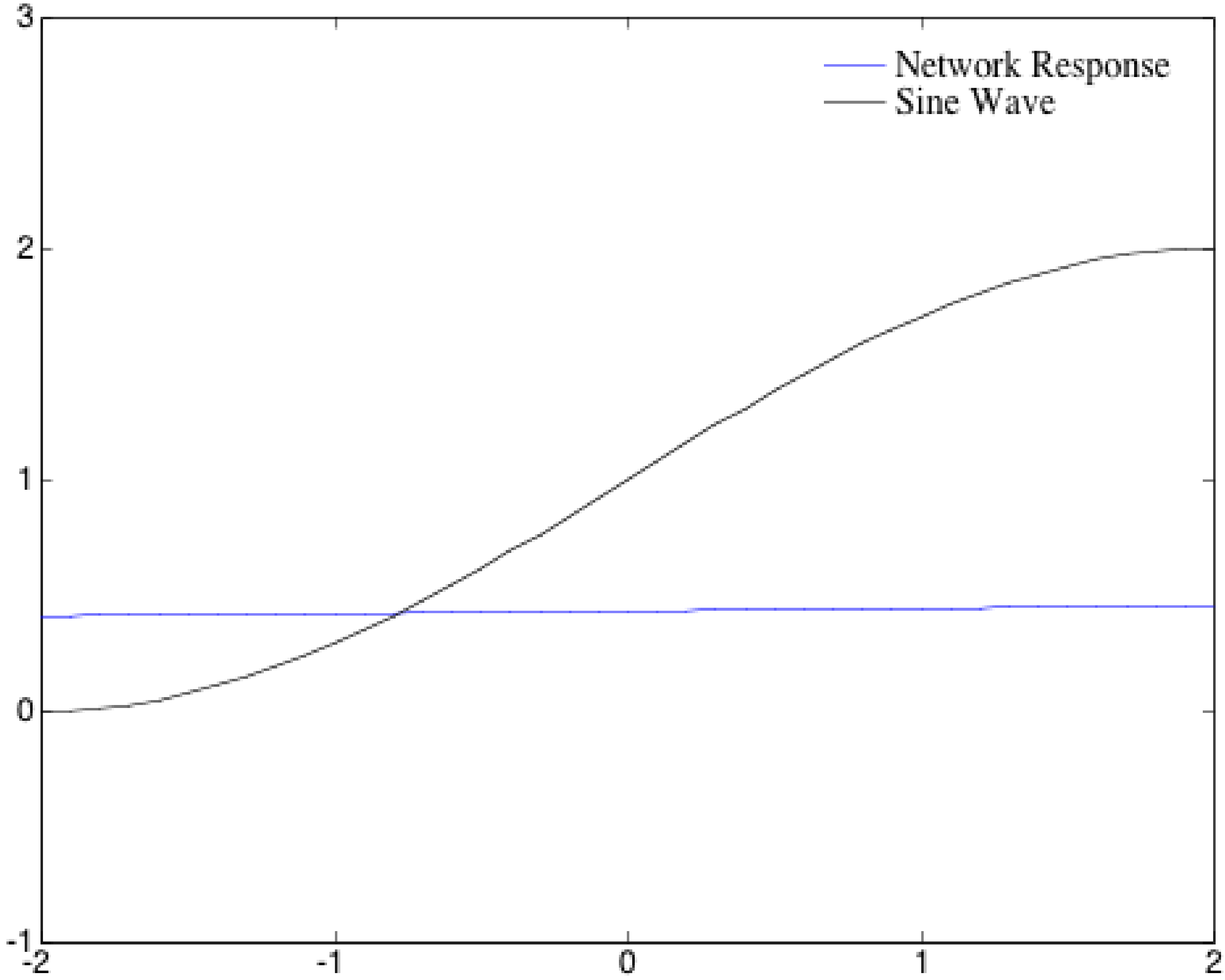
Numeric Example



Input Log-Sigmoid Layer Linear Layer

Initial Conditions

$$\mathbf{W}^1(0) = \begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix} \quad \mathbf{b}^1(0) = \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix} \quad \mathbf{W}^2(0) = \begin{bmatrix} 0.09 & -0.17 \end{bmatrix} \quad \mathbf{b}^2(0) = \begin{bmatrix} 0.48 \end{bmatrix}$$



Forward Propagation

$$a^0 = p = 1$$

$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{W}^1\mathbf{a}^0 + \mathbf{b}^1) = \mathbf{logsig}\left(\begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix}\begin{bmatrix} 1 \end{bmatrix} + \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix}\right) = \mathbf{logsig}\left(\begin{bmatrix} -0.75 \\ -0.54 \end{bmatrix}\right)$$

$$\mathbf{a}^1 = \begin{bmatrix} \dfrac{1}{1 + e^{0.75}} \\ \dfrac{1}{1 + e^{0.54}} \end{bmatrix} = \begin{bmatrix} 0.321 \\ 0.368 \end{bmatrix}$$

$$a^2 = \mathrm{f}^2(\mathbf{W}^2\mathbf{a}^1 + \mathbf{b}^2) = purelin\left(\begin{bmatrix} 0.09 & -0.17 \end{bmatrix}\begin{bmatrix} 0.321 \\ 0.368 \end{bmatrix} + \begin{bmatrix} 0.48 \end{bmatrix}\right) = \begin{bmatrix} 0.446 \end{bmatrix}$$

$$e = t - a = \left\{1 + \sin\left(\frac{\pi}{4}p\right)\right\} - a^2 = \left\{1 + \sin\left(\frac{\pi}{4}1\right)\right\} - 0.446 = 1.261$$

Transfer Function Derivatives

$$\dot{f}^{1}(n) = \frac{d}{dn}\left(\frac{1}{1+e^{-n}}\right) = \frac{e^{-n}}{(1+e^{-n})^{2}} = \left(1-\frac{1}{1+e^{-n}}\right)\left(\frac{1}{1+e^{-n}}\right) = (1-a^{1})(a^{1})$$

$$\dot{f}^{2}(n) = \frac{d}{dn}(n) = 1$$

Backpropagation

$$\mathbf{s}^2 = -2\dot{\mathbf{F}}^2(\mathbf{n}^2)(\mathbf{t}-\mathbf{a}) = -2\left[\dot{f}^2(n^2)\right](1.261) = -2\left[1\right](1.261) = -2.522$$

$$\mathbf{s}^1 = \dot{\mathbf{F}}^1(\mathbf{n}^1)(\mathbf{W}^2)^T\mathbf{s}^2 = \begin{bmatrix}(1-a_1^1)(a_1^1) & 0 \\ 0 & (1-a_2^1)(a_2^1)\end{bmatrix}\begin{bmatrix}0.09 \\ -0.17\end{bmatrix}\begin{bmatrix}-2.522\end{bmatrix}$$

$$\mathbf{s}^1 = \begin{bmatrix}(1-0.321)(0.321) & 0 \\ 0 & (1-0.368)(0.368)\end{bmatrix}\begin{bmatrix}0.09 \\ -0.17\end{bmatrix}\begin{bmatrix}-2.522\end{bmatrix}$$

$$\mathbf{s}^1 = \begin{bmatrix}0.218 & 0 \\ 0 & 0.233\end{bmatrix}\begin{bmatrix}-0.227 \\ 0.429\end{bmatrix} = \begin{bmatrix}-0.0495 \\ 0.0997\end{bmatrix}$$

Weight Update

$$\alpha = 0.1$$

$$\mathbf{W}^2(1) = \mathbf{W}^2(0) - \alpha\mathbf{s}^2(\mathbf{a}^1)^T = \begin{bmatrix} 0.09 & -0.17 \end{bmatrix} - 0.1\begin{bmatrix} -2.522 \end{bmatrix}\begin{bmatrix} 0.321 & 0.368 \end{bmatrix}$$

$$\mathbf{W}^2(1) = \begin{bmatrix} 0.171 & -0.0772 \end{bmatrix}$$

$$\mathbf{b}^2(1) = \mathbf{b}^2(0) - \alpha\mathbf{s}^2 = \begin{bmatrix} 0.48 \end{bmatrix} - 0.1\begin{bmatrix} -2.522 \end{bmatrix} = \begin{bmatrix} 0.732 \end{bmatrix}$$

$$\mathbf{W}^1(1) = \mathbf{W}^1(0) - \alpha\mathbf{s}^1(\mathbf{a}^0)^T = \begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix} - 0.1\begin{bmatrix} -0.0495 \\ 0.0997 \end{bmatrix}\begin{bmatrix} 1 \end{bmatrix} = \begin{bmatrix} -0.265 \\ -0.420 \end{bmatrix}$$

$$\mathbf{b}^1(1) = \mathbf{b}^1(0) - \alpha\mathbf{s}^1 = \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix} - 0.1\begin{bmatrix} -0.0495 \\ 0.0997 \end{bmatrix} = \begin{bmatrix} -0.475 \\ -0.140 \end{bmatrix}$$