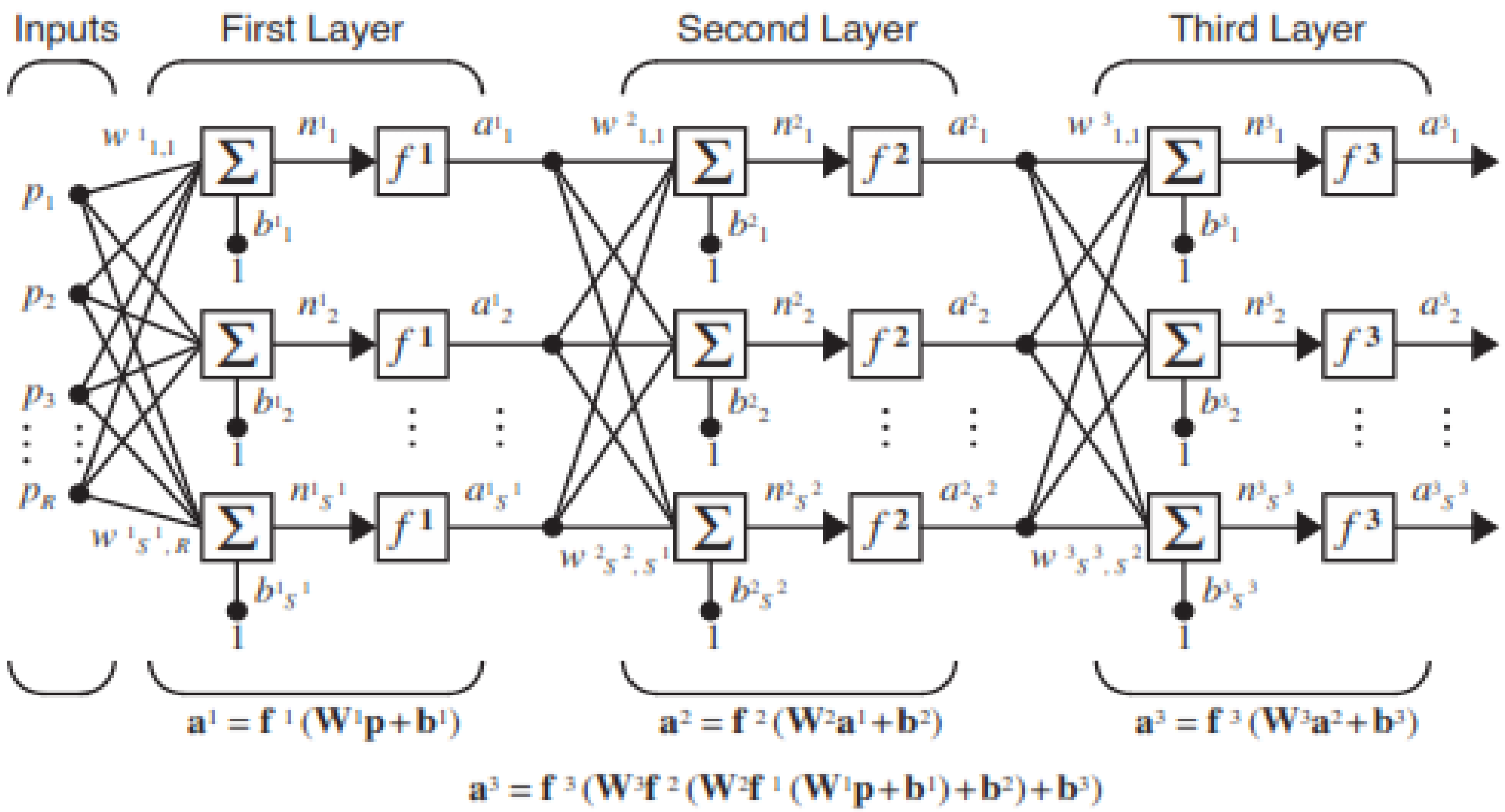
Backpropagation

Multilayer Perceptron



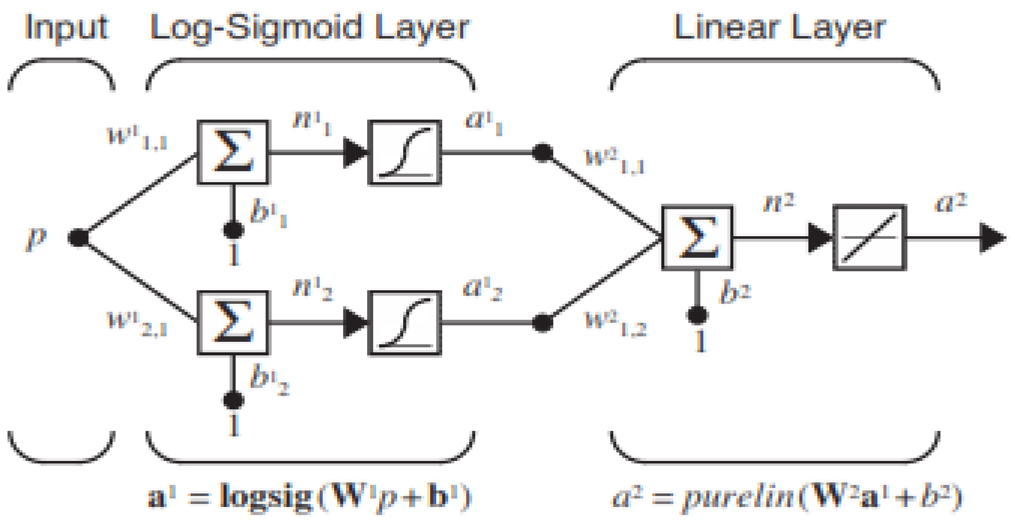
$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1)$ $\mathbf{a}^2 = \mathbf{f}^2(\mathbf{W}^2\mathbf{a}^1+\mathbf{b}^2)$ $\mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{a}^2+\mathbf{b}^3)$

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

$R - S^1 - S^2 - S^3$ Network

Function Approximation

The following example will illustrate the flexibility of the multilayer perceptron for implementing functions.



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

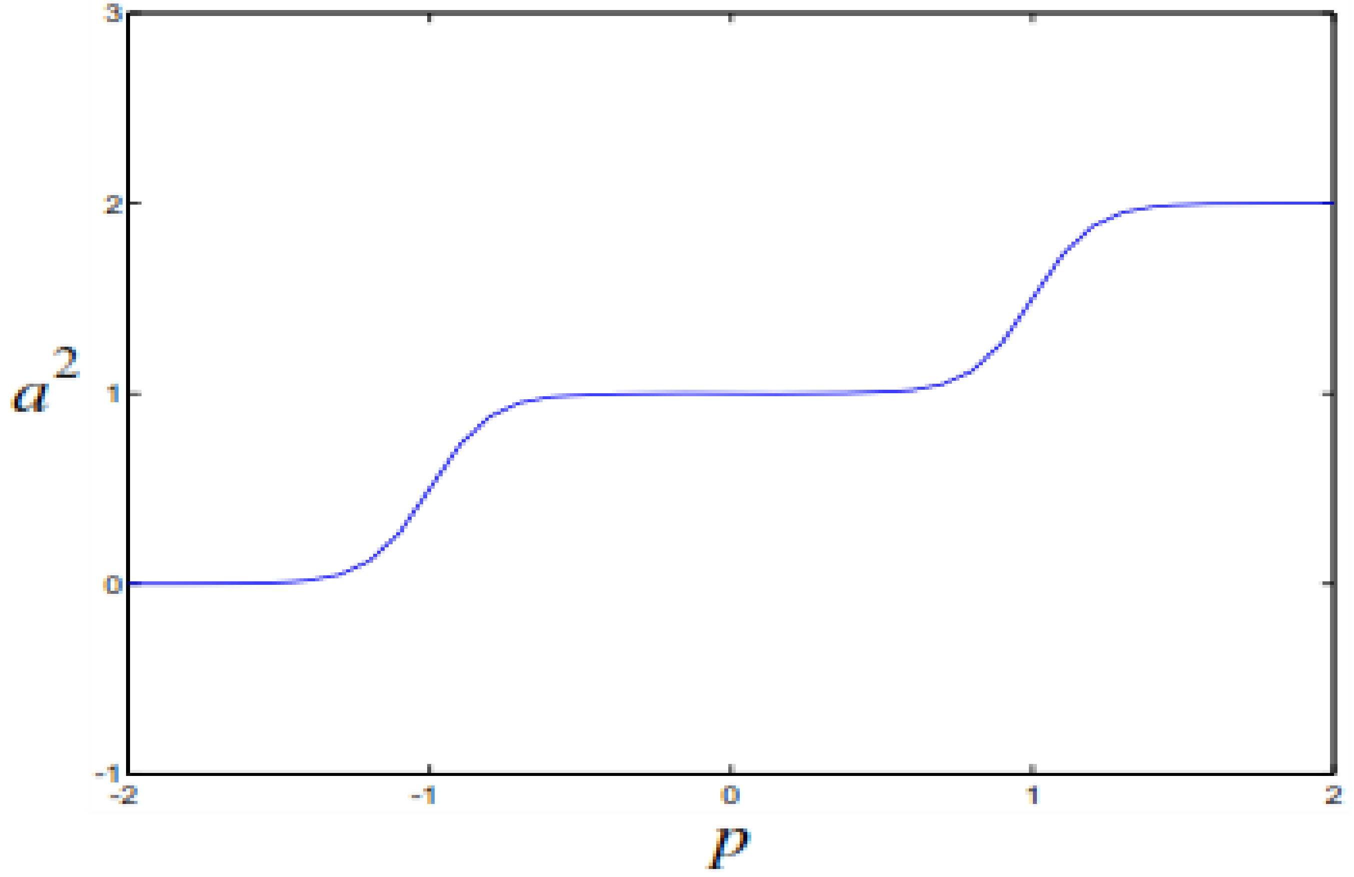
$$f^2(n) = n$$

Nominal Parameter Values

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

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

The network output response for these parameters as the input ***P*** is varied over the range [-2, 2].



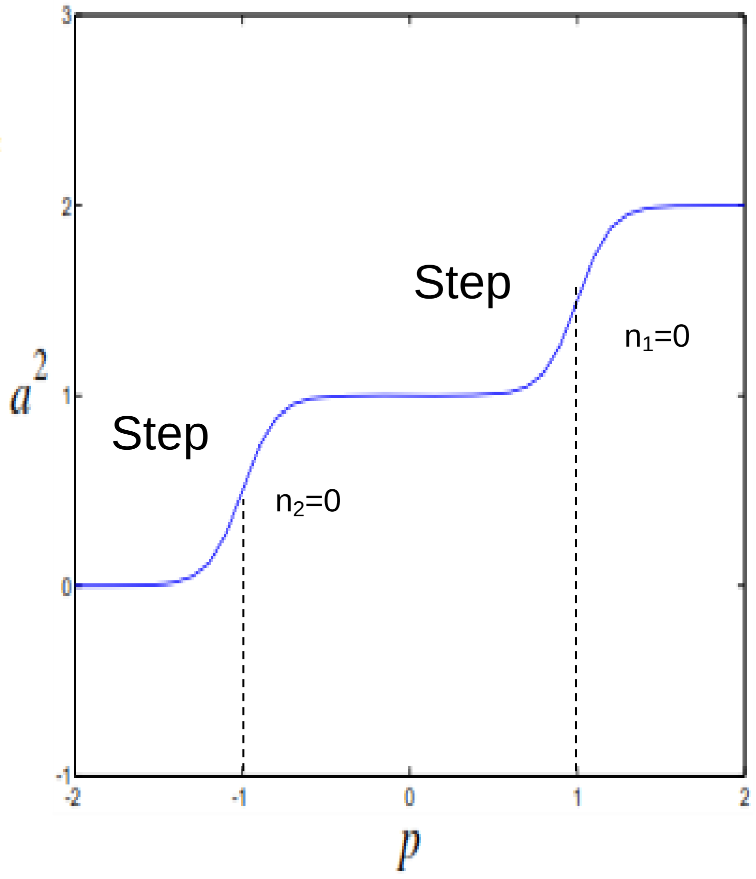
Notice that the response consists of two steps, one for each of the log-sigmoid neurons in the first layer. By adjusting the network parameters we can change the shape and location of each step.

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

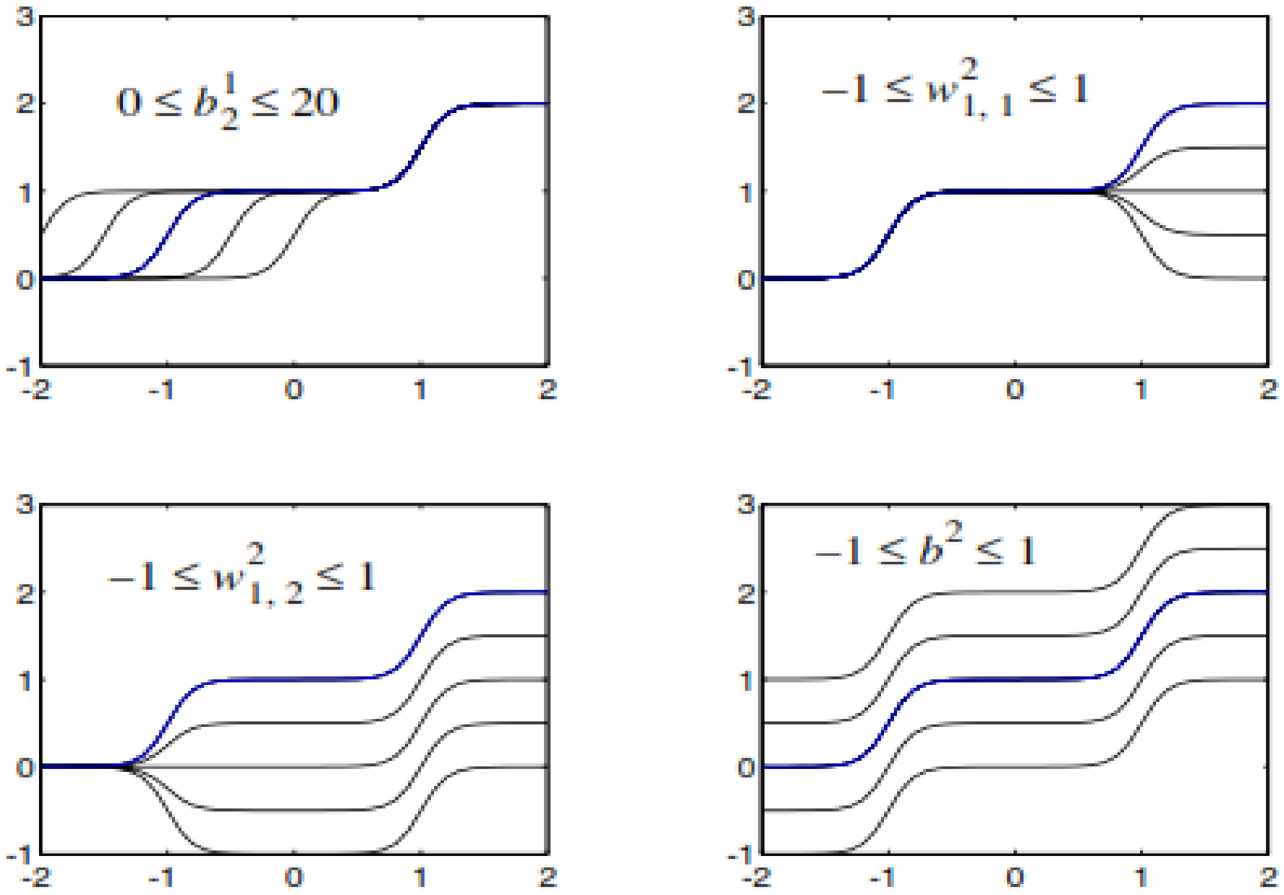
$$n_1^1 = w_{1,1}^1 p + b_1^1 = 0 \quad \Rightarrow \quad p = -\frac{b_1^1}{w_{1,1}^1} = -\frac{-10}{10} = 1$$

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

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

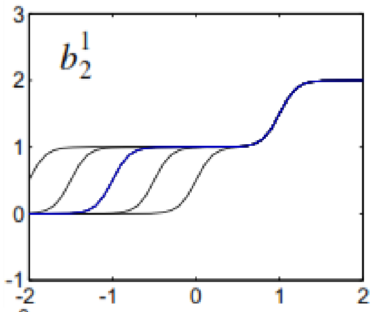


The blue curve is the nominal response. The other curves correspond to the network response when one parameter at a time is varied over the following ranges

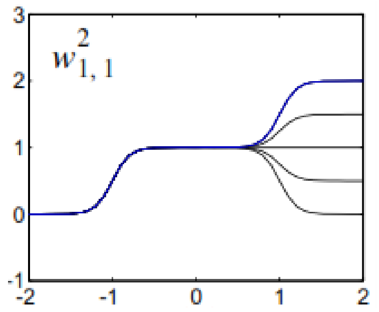


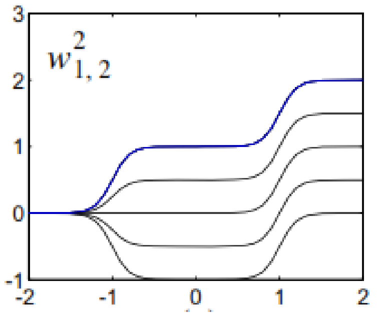
Effect of Parameter Changes on Network Response

The biases in the first (hidden) layer can be used to locate the position of the steps.

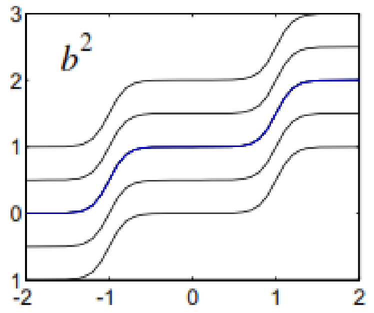


The weights determine the slope of the steps.



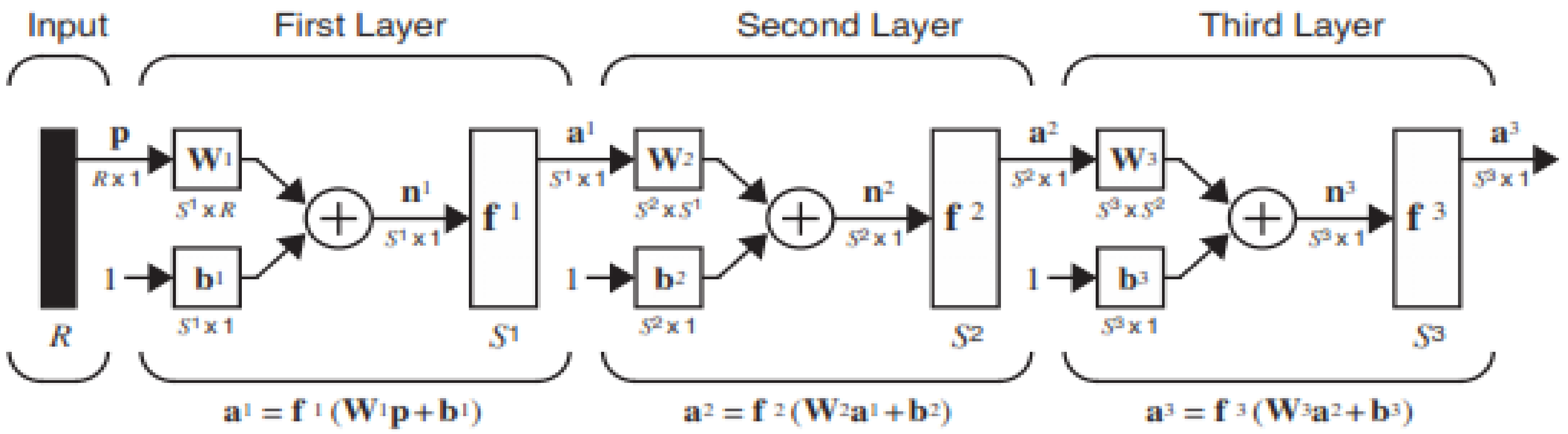


The bias in the second (output) layer shifts the entire network response up or down.



- In fact, any 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.
- The next step is to develop an algorithm to train such networks.

The Backpropagation Algorithm



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

Multilayer network

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

M is the number of layers in the net

The neurons in the first layer receive external inputs: $\mathbf{a}^0 = \mathbf{p}$

The outputs of the neurons in the last layer are considered the net outputs $\mathbf{a} = \mathbf{a}^M$

Performance Index of Backpropagation

- The performance index of the algorithms is the mean square error. (Least Mean Square error or LMS).
- The weights are adjusted, using a gradient descent method, so as to minimize the mean square error.
- The algorithm is provided with a set of examples of proper network behavior:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

- where $\mathbf{p}_q$ is an input to the network, and $\mathbf{t}_q$ is the corresponding target output.
- As each input is applied to the network, the network output is compared to the target.

- The algorithm should adjust the network parameters in order to minimize the mean square error:

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

- where E[ ] denotes the expected value, and **x** is the vector of network weights and biases

$$\mathbf{x} = \begin{bmatrix} \mathbf{w} \\ b \end{bmatrix}$$

- If the network has multiple outputs this generalizes to the vector case

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

- Approximate Mean Square Error (the expectation replaced by single iteration *k*)

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

Where $\hat{F}(\mathbf{x})$ is the estimated performance index.

Approximate Steepest Descent

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

- Now we come to the difficult part – the computation of the partial derivatives.
- For a single-layer linear network these partial derivatives are conveniently computed.
- 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 *f* with respect to a third variable *w*. The chain rule is then:

Chain Rule

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

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

$$\boxed{\frac{\partial \hat{F}}{\partial w^m_{i,j}} = \frac{\partial \hat{F}}{\partial n^m_i} \times \frac{\partial n^m_i}{\partial w^m_{i,j}}} \qquad \boxed{\frac{\partial \hat{F}}{\partial b^m_i} = \frac{\partial \hat{F}}{\partial n^m_i} \times \frac{\partial n^m_i}{\partial b^m_i}}$$

Gradient Calculation

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

$$\boxed{n_i^m = \sum_{j=1}^{S^{m-1}} w_{i,j}^m a_j^{m-1} + b_i^m} \rightarrow \boxed{\frac{\partial n_i^m}{\partial w_{i,j}^m} = a_j^{m-1}, \quad \frac{\partial n_i^m}{\partial b_i^m} = 1}$$

Sensitivity **s_i^m**: the *sensitivity* of $\hat{F}$ to changes in the *i*th element of the net input *n* at layer *m*

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

Steepest Descent

- We can now express the approximate steepest descent algorithm as

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

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

In matrix form this becomes:

$$\boxed{\begin{aligned} \mathbf{W}^m(k+1) &= \mathbf{W}^m(k) - \alpha \mathbf{s}^m(\mathbf{a}^{m-1})^T \\ \mathbf{b}^m(k+1) &= \mathbf{b}^m(k) - \alpha \mathbf{s}^m, \end{aligned}}$$

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

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

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

Next Step: Compute the Sensitivities (Backpropagation)

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:

Jacobian Matrix

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

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

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

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

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

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

Backpropagation (Sensitivities)

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

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

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

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

Initialization Last Layer

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

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

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

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

Summery

Forward Propagation

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

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

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

Backpropagation

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

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

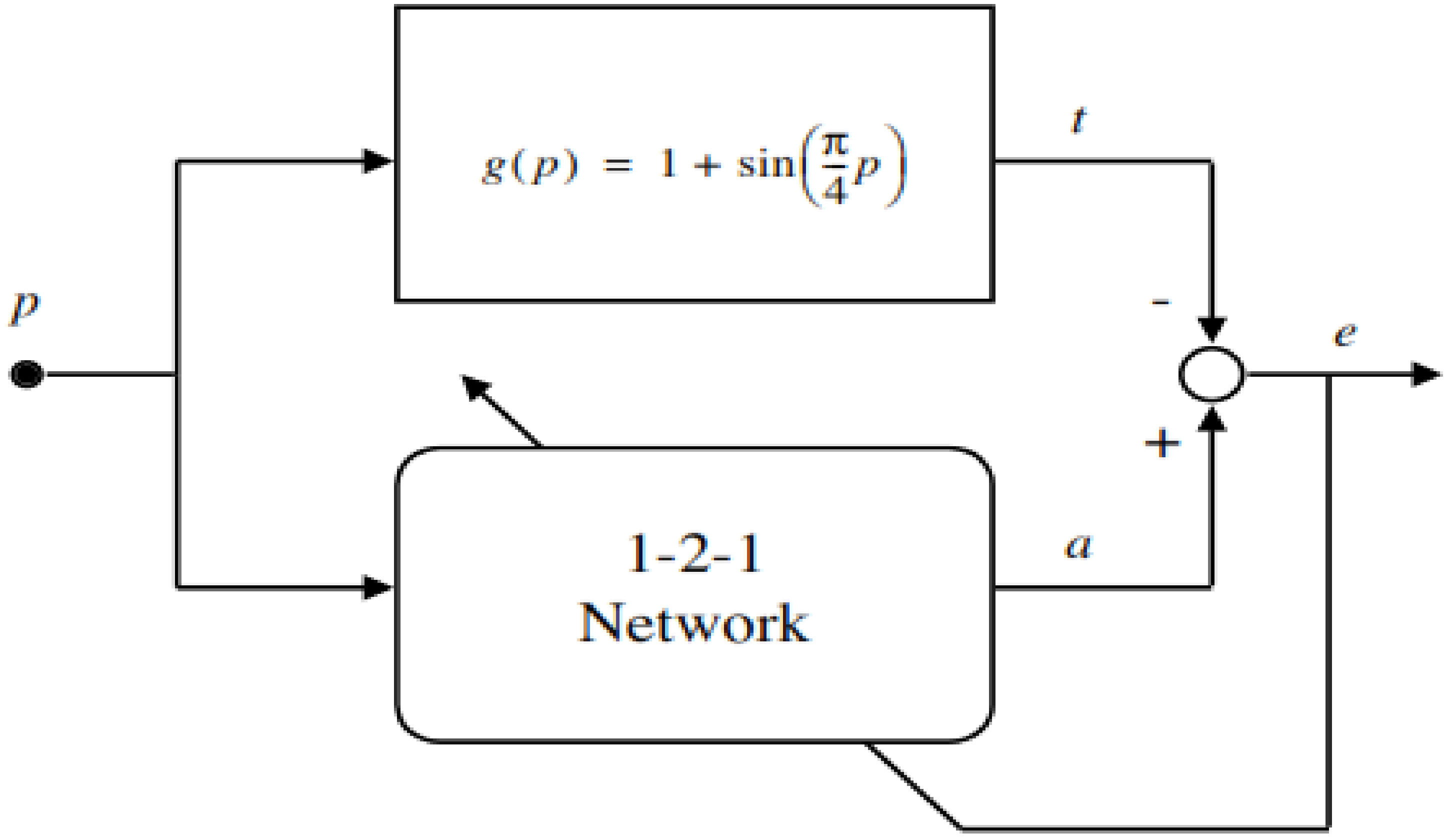
Weight Update

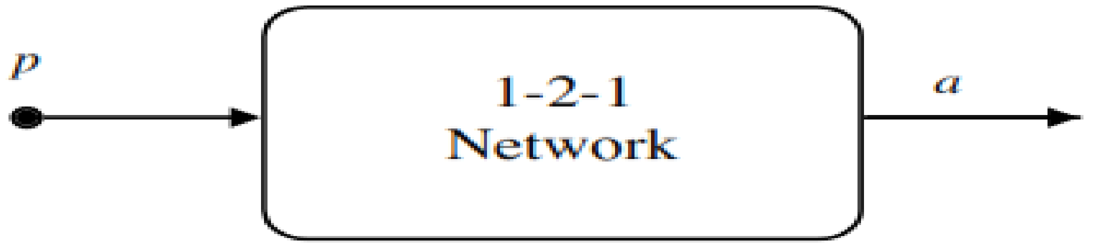
$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha\mathbf{s}^m(\mathbf{a}^{m-1})^T \qquad \mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha\mathbf{s}^m$$

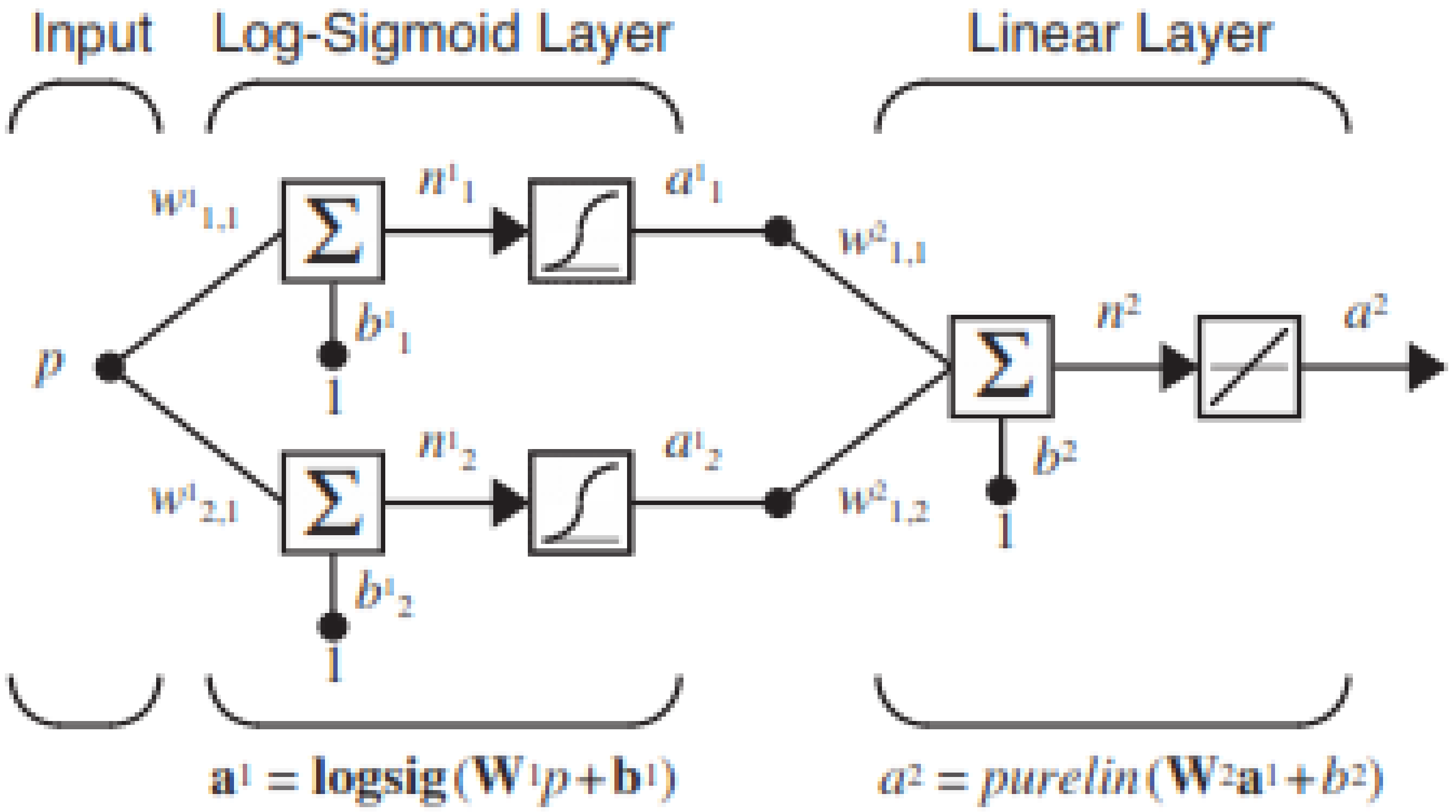
Example: Function Approximation

- To illustrate the backpropagation algorithm, let's choose a 1-2-1 network to approximate the function

$$g(p) = 1 + \sin\left(\frac{\pi}{4}p\right) \text{ for } -2 \le p \le 2 .$$




p
1-2-1
Network
a


Input
Log-Sigmoid Layer
Linear Layer
p
$w^1_{1,1}$
$w^1_{2,1}$
n^1_1
n^1_2
a^1_1
a^1_2
b^1_1
b^1_2
1
1
$w^2_{1,1}$
$w^2_{1,2}$
n^2
a^2
b^2
1
$\mathbf{a}^1 = \mathbf{logsig}(\mathbf{W}^1 p + \mathbf{b}^1)$
$a^2 = purelin(\mathbf{W}^2 \mathbf{a}^1 + b^2)$

Initial Conditions

- Before we begin the backpropagation algorithm we need to choose some initial values for the network weights and biases. Generally these are chosen to be small random values.

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

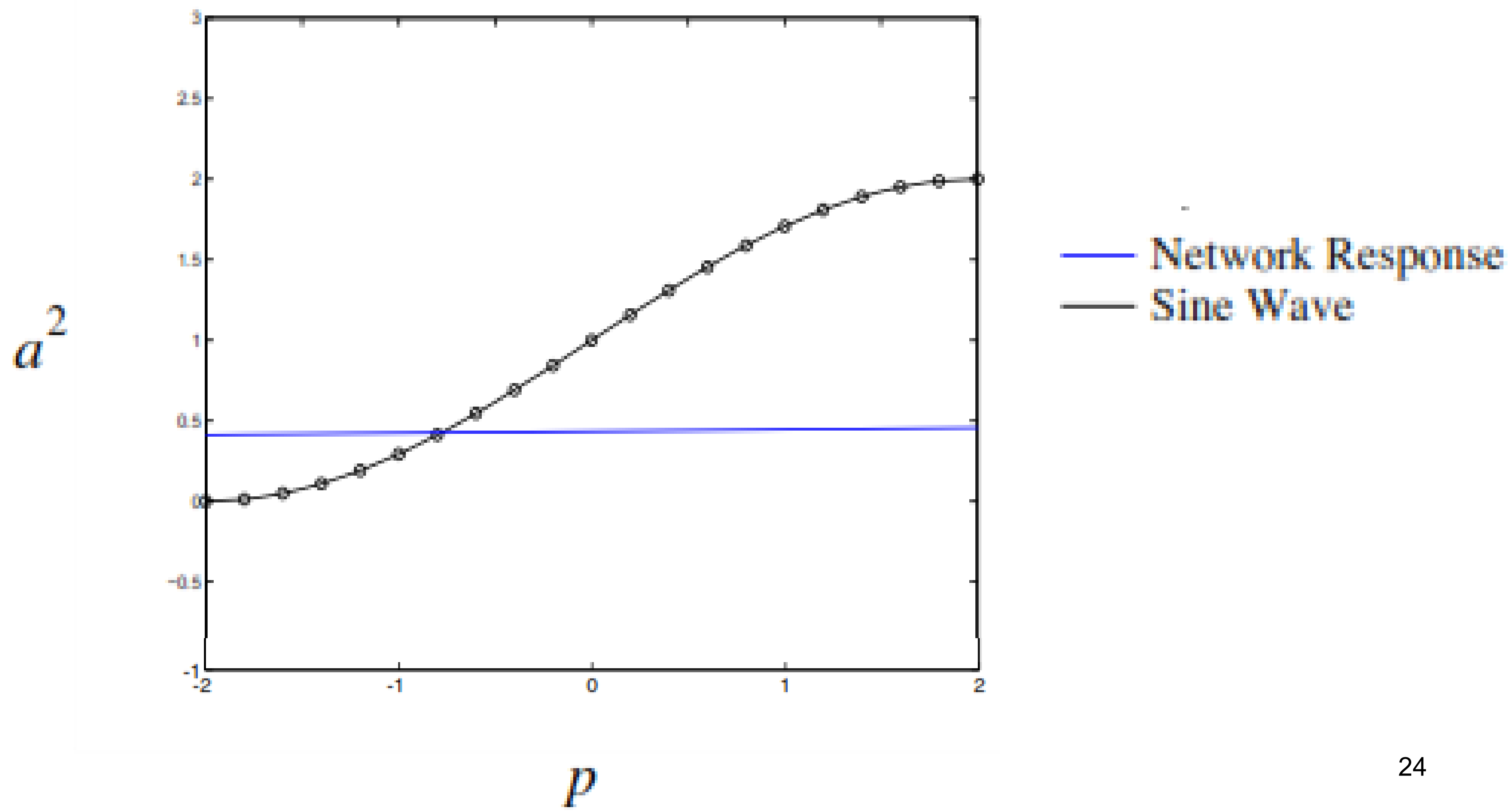
- To obtain our training set we will evaluate this function at several values of *p.*
- Next, we need to select a training set:

$$\{p_1, t_1\}, \{p_2, t_2, \ldots, \{p_q, t_q\} .$$

- In this case, we will sample the function at 21 points in the range [-2,2] at equally spaced intervals of 0.2.

Network Response

- The response of the network for these initial values is as shown, along with the sine function we wish to approximate.
- The training points are indicated by the circles



Forward Propagation

- The training points can be presented in any order, but they are often chosen randomly.
- For our initial input we will choose *p=1*, which is the 16th training point:

$$a^0 = p = 1$$

- The output of the first layer is then

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

$$= \begin{bmatrix}\dfrac{1}{1+e^{0.75}}\\[2ex]\dfrac{1}{1+e^{0.54}}\end{bmatrix} = \begin{bmatrix}0.321\\0.368\end{bmatrix}.$$

- The second layer output is

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

- The error would then be

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

$$= \left\{1 + \sin\left(\frac{\pi}{4}1\right)\right\} - 0.446 = 1.261$$

Transfer Function Derivatives

- The next stage of the algorithm is to backpropagate the sensitivities.
- Before we begin the backpropagation, we will need the derivatives of the transfer functions $f^1(n)$, and $f^2(n)$
- For the first layer

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

- For the second layer we have

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

Backpropagation

- We can now perform the backpropagation. The starting point is found at the second layer:

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

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

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

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

Weight Update

$$\alpha = 0.1$$

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

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

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

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

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

How to complete the training

- Up to this the first iteration of the back propagation algorithm is completed.
- We next proceed to randomly choose another input from the training set and perform another iteration of the algorithm.
- We continue to iterate until the difference between the network response and the target function reaches some acceptable level.
  - (Note that this will generally take many passes through the entire training set.)

Batch vs. Incremental Training

- The algorithm described above is the stochastic gradient descent algorithm, which involves "on-line" or *incremental training*, in which the network weights and biases are updated after each input is presented .
- It is also possible to perform *batch training*, in which the complete gradient is computed (after all inputs are applied to the network) before the weights and biases are updated.
- For example, if each input occurs with equal probability, the mean square error performance index can be written

$$F(\mathbf{x}) = E[\mathbf{e}^T\mathbf{e}] = E[(\mathbf{t}-\mathbf{a})^T(\mathbf{t}-\mathbf{a})] = \frac{1}{Q}\sum_{q=1}^{Q}(\mathbf{t}_q-\mathbf{a}_q)^T(\mathbf{t}_q-\mathbf{a}_q)$$

The total gradient of this performance index is

$$\nabla F(\mathbf{x}) = \nabla\left\{\frac{1}{Q}\sum_{q=1}^{Q}(\mathbf{t}_q-\mathbf{a}_q)^T(\mathbf{t}_q-\mathbf{a}_q)\right\} = \frac{1}{Q}\sum_{q=1}^{Q}\nabla\{(\mathbf{t}_q-\mathbf{a}_q)^T(\mathbf{t}_q-\mathbf{a}_q)\}$$

- Therefore, to implement a batch version of the backpropagation algorithm, we would step from the forward Eqs. through the sensitive calculations for all of the inputs in the training set.
- Then, the individual gradients would be averaged to get the total gradient.
- The update equations for the batch steepest descent algorithm would then be

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \frac{\alpha}{Q}\sum_{q=1}^{Q}\mathbf{s}_q^m(\mathbf{a}_q^{m-1})^T$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \frac{\alpha}{Q}\sum_{q=1}^{Q}\mathbf{s}_q^m .$$

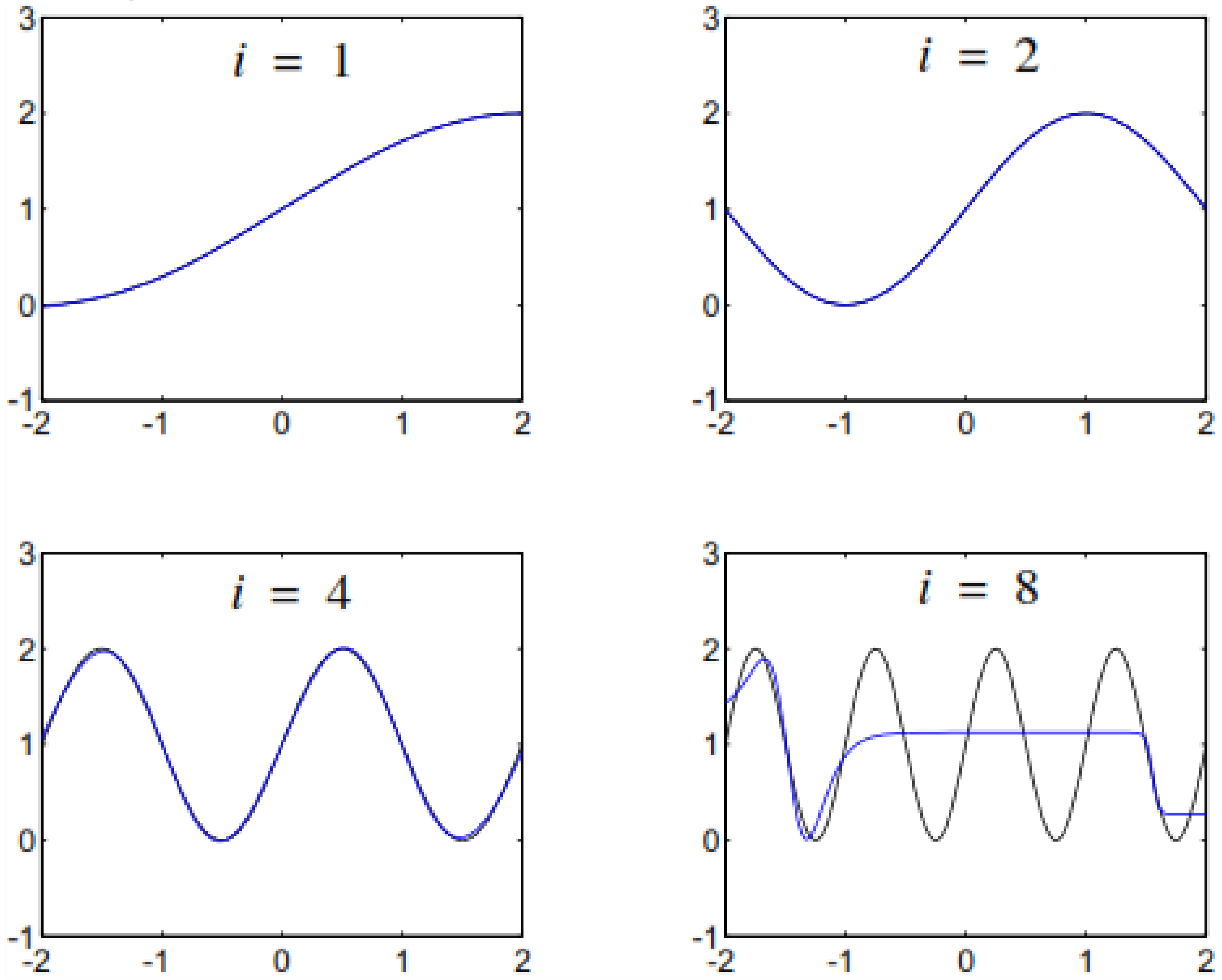
Choice of Network Architecture

- For our first example let's assume that we want to approximate the following functions:

$$g(p) = 1 + \sin\left(\frac{i\pi}{4}p\right) \text{ for } -2 \le p \le 2$$

- where ***i*** takes on the values 1, 2, 4 and 8.
- As ***i*** is increased, the function becomes more complex, because we will have more periods of the sine wave over the interval -2 ≤ *p* ≤ 2.
- It will be more difficult for a neural network with a fixed number of neurons in the hidden layers to approximate ***g(p)*** as ***i*** is increased.
- For this first example we will use a 1-3-1 network,
- The transfer function for the first layer is log-sigmoid and the transfer function for the second layer is linear.
- This type of two-layer network can produce a response that is a sum of three log-sigmoid functions (or as many log-sigmoids as there are neurons in the hidden layer).
- Clearly there is a limit to how complex a function this network can implement.

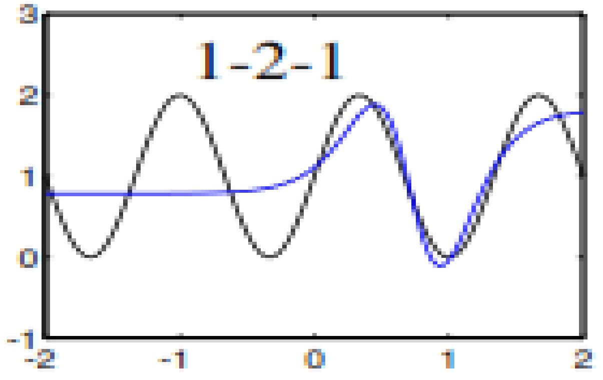
- The final network responses after it has been trained are shown by the blue lines

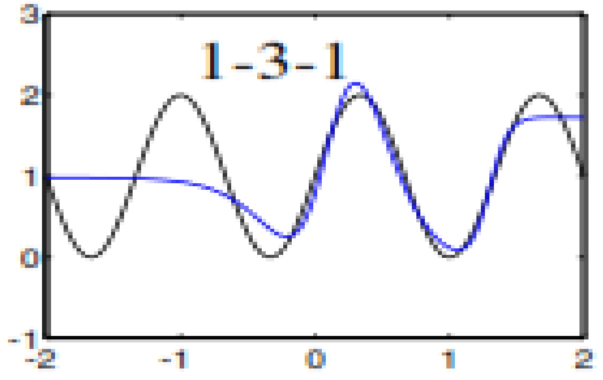


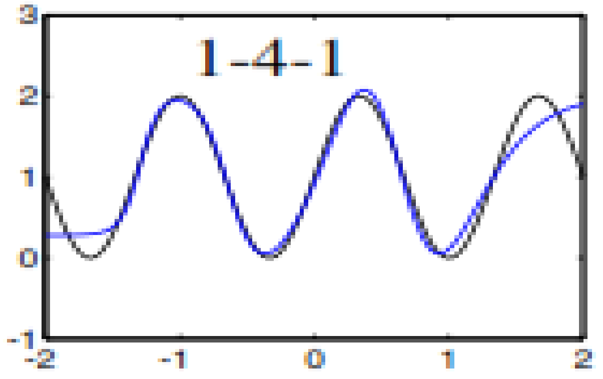
- We can see that for i=4 the 1-3-1 network reaches its maximum capability.
- When $i>4$ the network is not capable of producing an accurate approximation

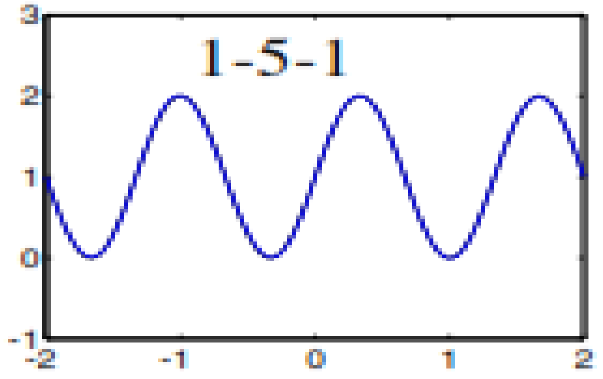
Choice of Network Architecture (1-S^1-1), the response of this network is a superposition of S^1 sigmoid functions

$$g(p) = 1 + \sin\left(\frac{6\pi}{4}p\right)$$









- To summarize these results:
- a 1- S^1 -1 network, with sigmoid neurons in the hidden layer and linear neurons in the output layer, can produce a response that is a superposition of S^1 sigmoid functions.
- If we want to approximate a function that has a large number of inflection points, we will need to have a large number of neurons in the hidden layer.