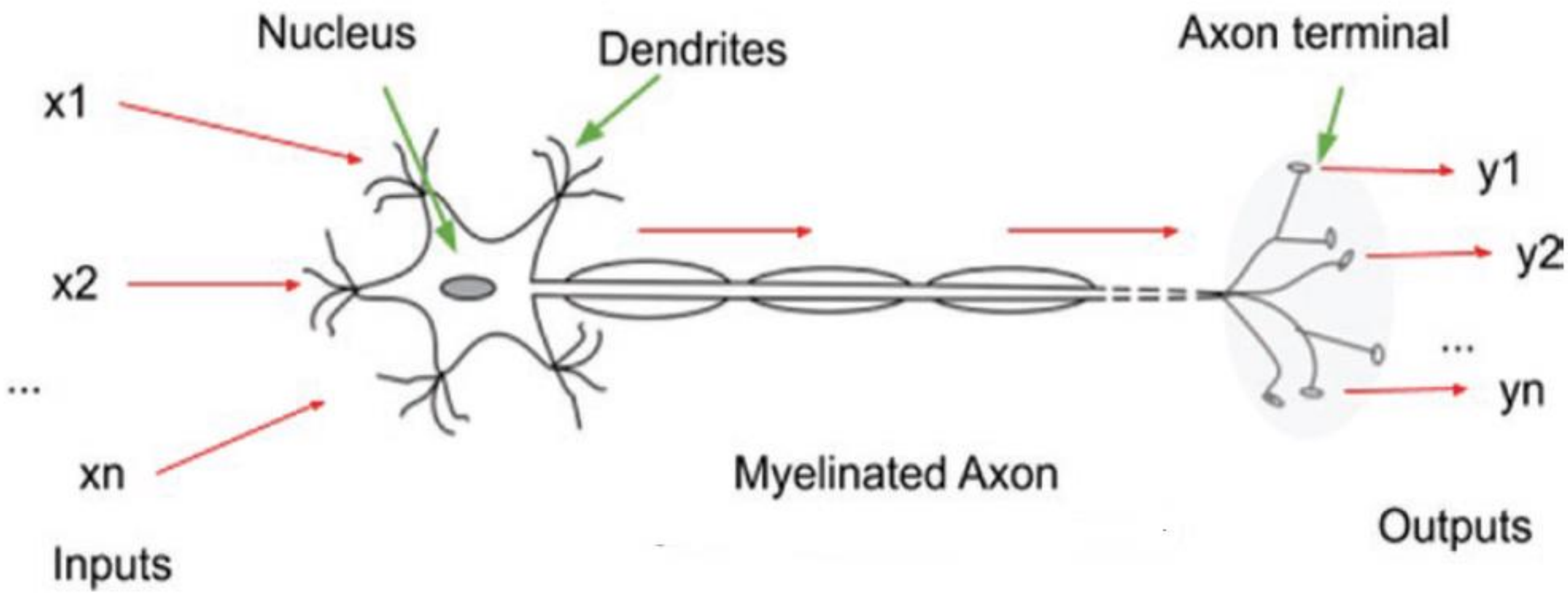
DEEP LEARNING

Dr. Mithun Kumar Kar
Assistant Professor (Sr-Gr)
School of Artificial Intelligence
Amrita Vishwa Vidyapeetham Coimbatore

Artificial Neural Networks

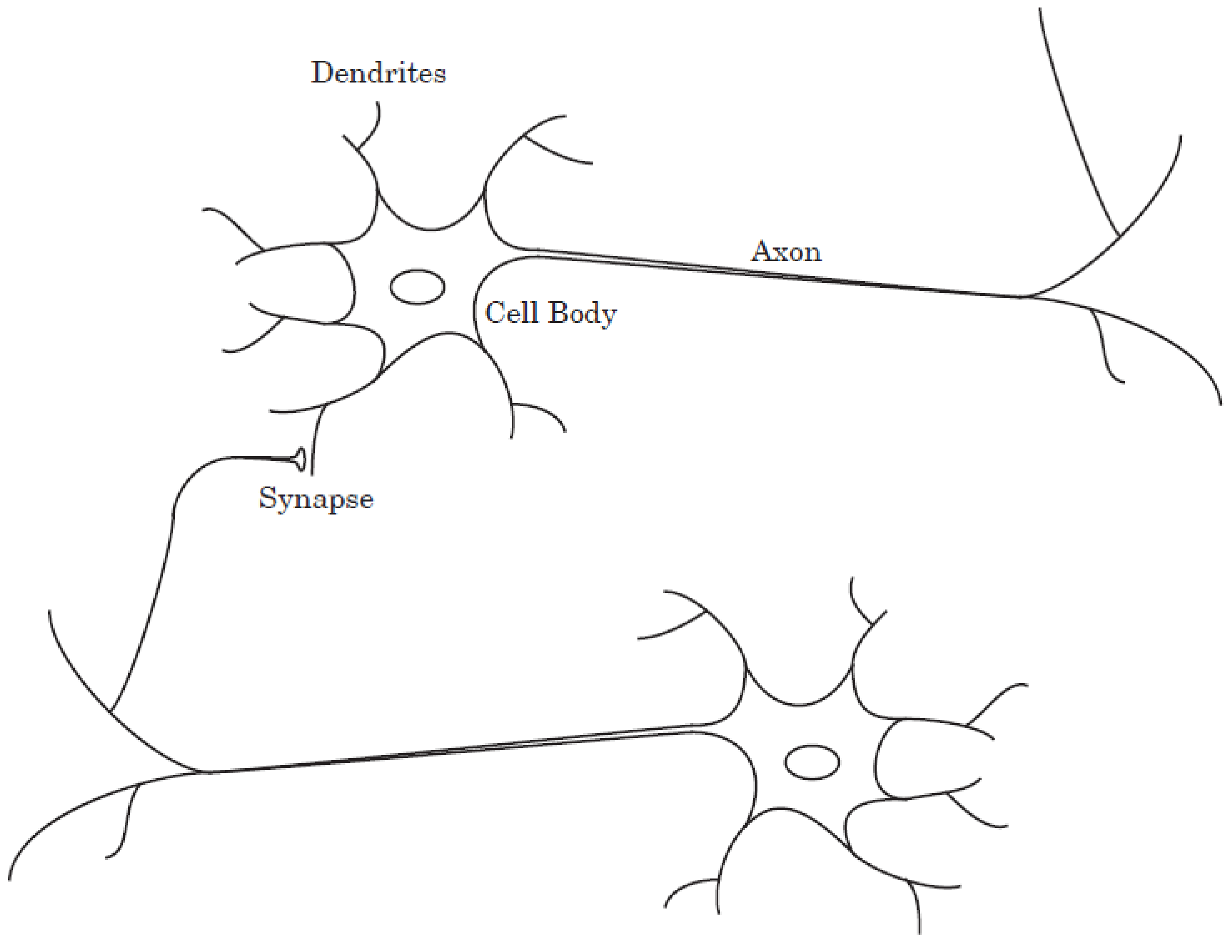
- An artificial neural network (ANN) is a computing system that is designed to work the way the human brain works.



- **When we go in to a new environment, we adapt to the new environment that is we learn**.

Artificial Neural Networks

- Neurons have three principal components: the **dendrites**, the **cell body** and the **axon**.

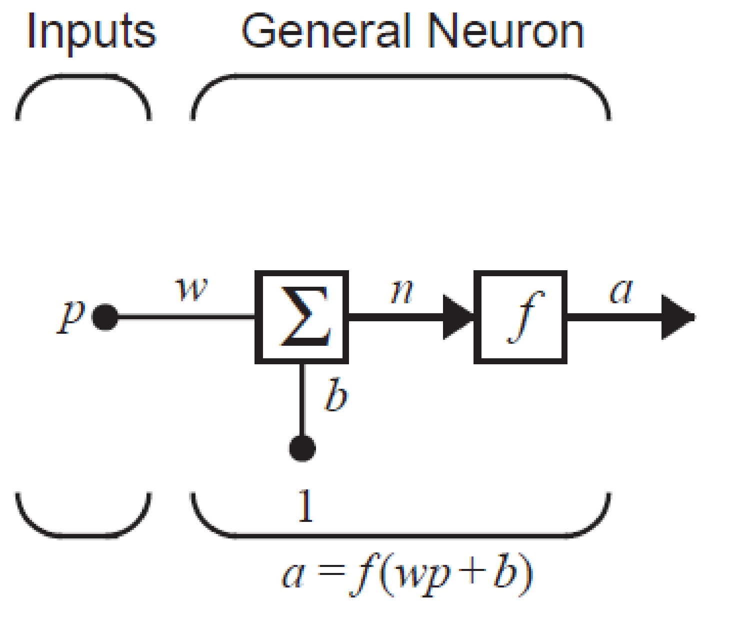


Artificial Neural Networks

- The **dendrites** are tree-like **receptive networks** of nerve fibers that carry electrical signals into the cell body.
- The **cell body** effectively **sums and thresholds** these incoming signals.
- The **axon** is a single long fiber that **carries the signal** from the cell body out to other neurons.
- The point of **contact between** an **axon** of one cell and a **dendrite** of another cell is called a **synapse**.

Neuron Models

- **Single-Input Neuron**:
- P is the input,
  w is the weight,
  b is the bias or offset,
  f is called the activation
  function or transfer function.



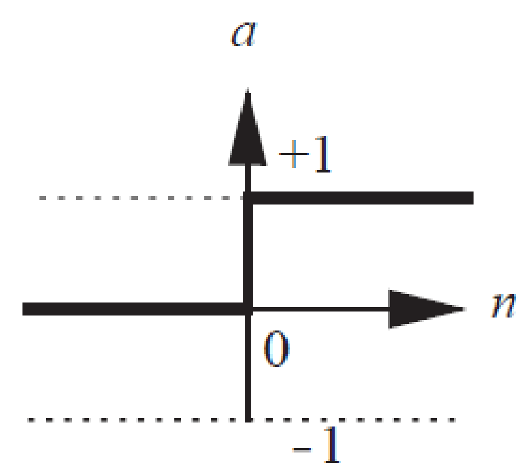
- The summer output , often ref
  goes

into a transfer function , which produces the scalar neuron output .

- If we take p = 2, w = 3, b = -1.5
  a = f (wp + b) = f[2(3)-1.5)]=f(4.5)
- The actual output depends on the particular transfer function that is chosen.

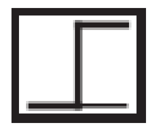
Neuron Models

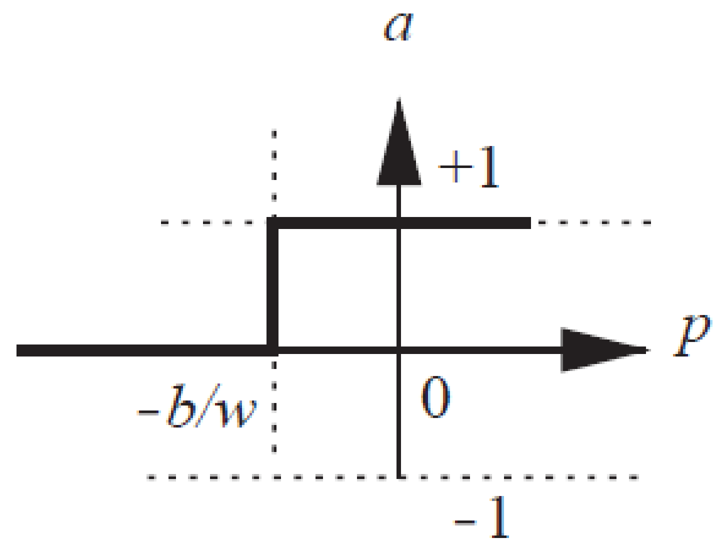
- **Transfer Functions:** The transfer function may be a linear or a nonlinear function of net input 'n'.
- A variety of transfer functions are used depending on the required application of the neural network.
- **Hard Limit Transfer Function:**
- The hard limit transfer function sets the output of the neuron to 0 if the function argument is less than 0, or 1 if its argument is greater than or equal to 0.



Hard Limit Transfer Function

Neuron Models

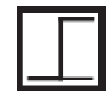
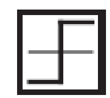
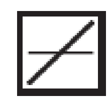
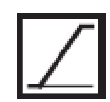
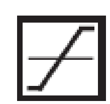
- **a = f (wp + b)** → 



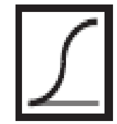
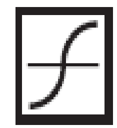
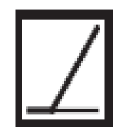
$$a = f\left(\sum_i w_i p_i\right) = f(W.P) = f(W^T P)$$

- The above figure illustrates the input/output characteristic of a single-input neuron that uses a hard limit transfer function.

Transfer functions

Name	Input/Output Relation	Icon	MATLAB Function
Hard Limit	$a = 0 \quad n < 0$ $a = 1 \quad n \geq 0$		hardlim
Symmetrical Hard Limit	$a = -1 \quad n < 0$ $a = +1 \quad n \geq 0$		hardlims
Linear	$a = n$		purelin
Saturating Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n \leq 1$ $a = 1 \quad n > 1$		satlin
Symmetric Saturating Linear	$a = -1 \quad n < -1$ $a = n \quad -1 \leq n \leq 1$ $a = 1 \quad n > 1$		satlins

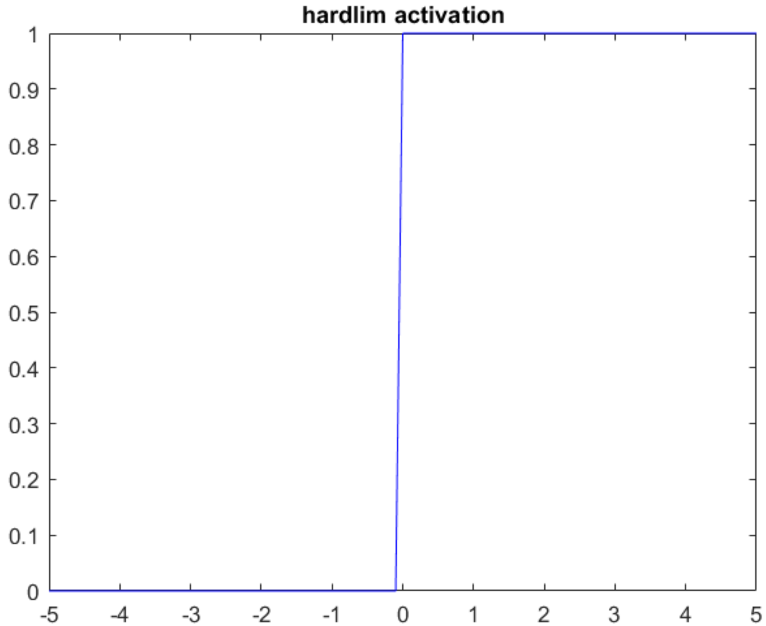
Transfer functions

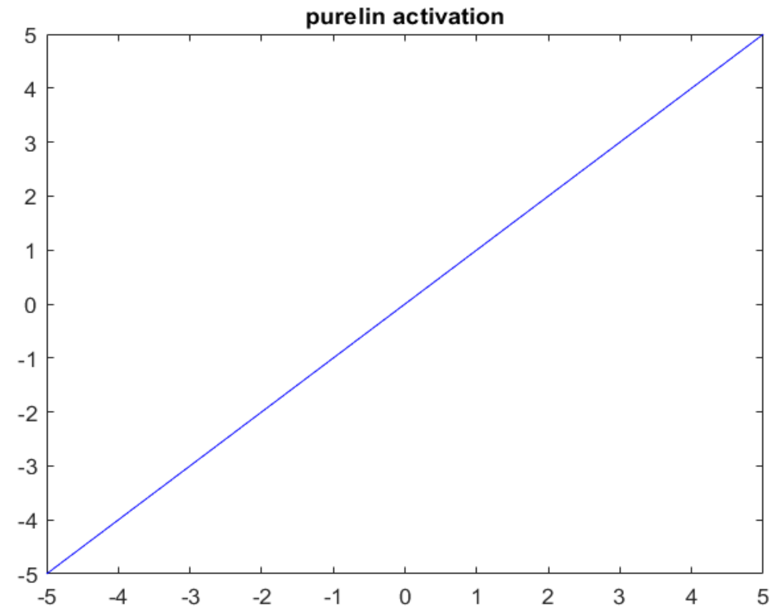
Log-Sigmoid	$a = \dfrac{1}{1+e^{-n}}$		logsig
Hyperbolic Tangent Sigmoid	$a = \dfrac{e^{n}-e^{-n}}{e^{n}+e^{-n}}$		tansig
Positive Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \le n$		poslin
Competitive	$a = 1$ neuron with max n $a = 0$ all other neurons	$\boxed{\mathrm{C}}$	compet

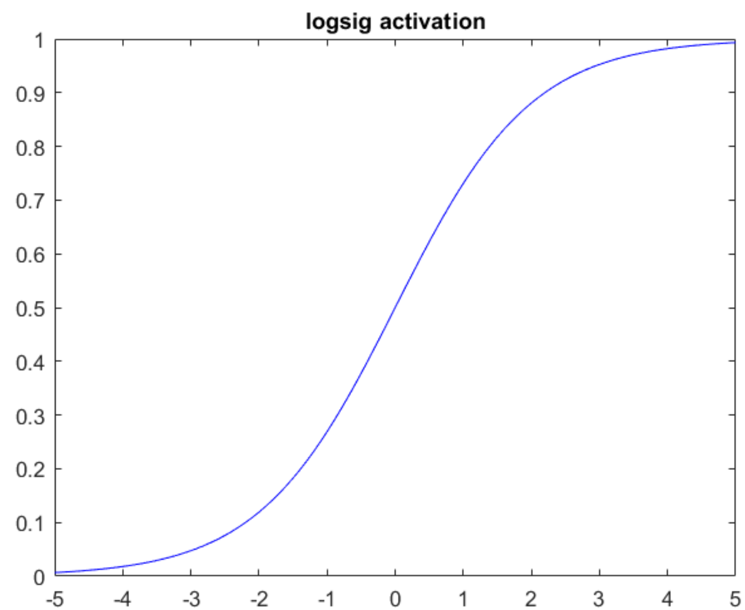
Matlab Implementation

- n = -5:0.1:5;
- figure()
- plot(n,hardlim(n),'b');
- title('hardlim activation')
- figure();
- plot(n,purelin(n),'b');
- title('purelin activation')
- figure();
- plot(n,logsig(n),'b');
- title('logsig activation')

Matlab Implementation



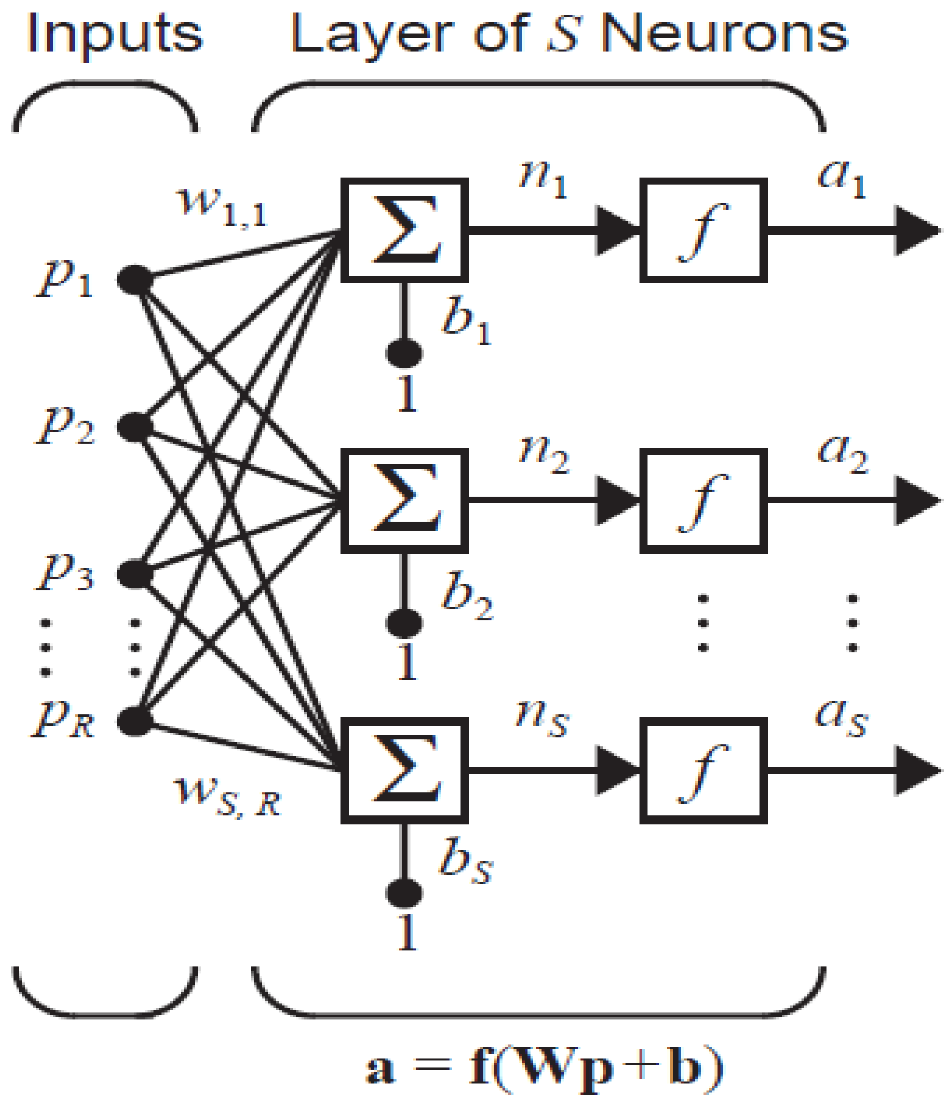




A single-layer network of S neurons

- Layers of n neurons:
- Each of the R inputs is connected to each of the neurons.

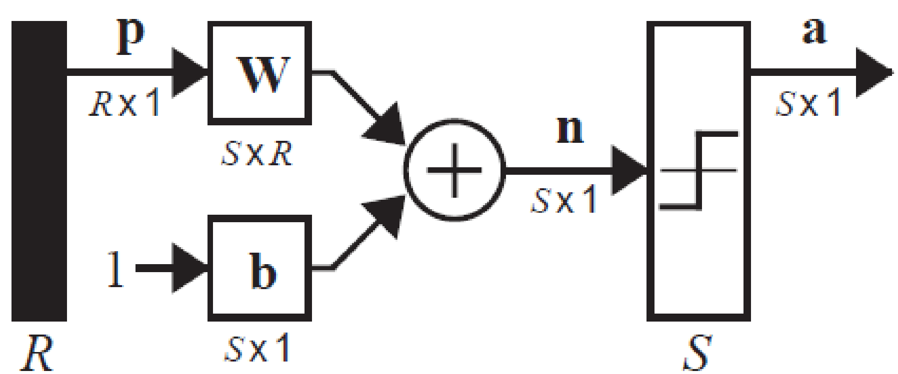
$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$



Layer of S Neurons

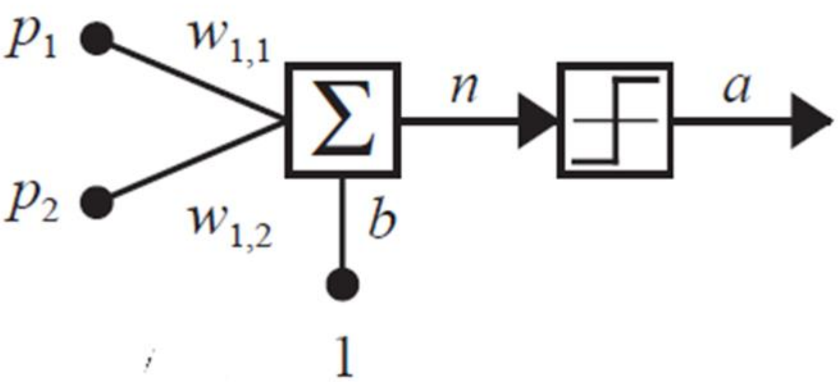
Perceptron

- Figure shows a single-layer perceptron with symmetric hard limit transfer function (Single-neuron perceptron).
- Single-neuron perceptron can classify input vectors into two categories.



Perceptron

- For a two-input perceptron,



$$a = f(wp + b)$$

$$a = hardlim(n) = hardlim(\mathbf{W}\mathbf{p} + b)$$

$$= hardlim({}_1\mathbf{w}^T\mathbf{p} + b) = hardlim(w_{1,1}p_1 + w_{1,2}p_2 + b)$$

Perceptron

- The decision boundary between the categories is determined by $wp + b = 0$
- The decision boundary is determined by the input vectors for which the net input 'n' is zero.
- To draw the line, we can find the points where it intersects the and axes
- To draw the line, we can find the points where it intersects the p_1 and p_2 axes.

Perceptron Learning Rule

- Let $w_{11} = 1, w_{12} = 1$ and $b =$ -1
- The decision boundary is then given by $n = w_i^T p + b = 0$

$$w_{11} p_1 + w_{12} p_2 + b = 0$$

- This defines a line in the input space.
- On the line and on one side of the line the network output will be 0 , whereas on the other side of the line the output will be 1.
- We can find the intercepts as

$$p_1 = -\frac{b}{w_{11}} \qquad p_2 = -\frac{b}{w_{12}}$$

Perceptron Learning Rule

- To find the p_1 intercept, set $p_2 = 0$

$$p_1 = -\frac{b}{w_{11}} = -\frac{-1}{1} = 1$$

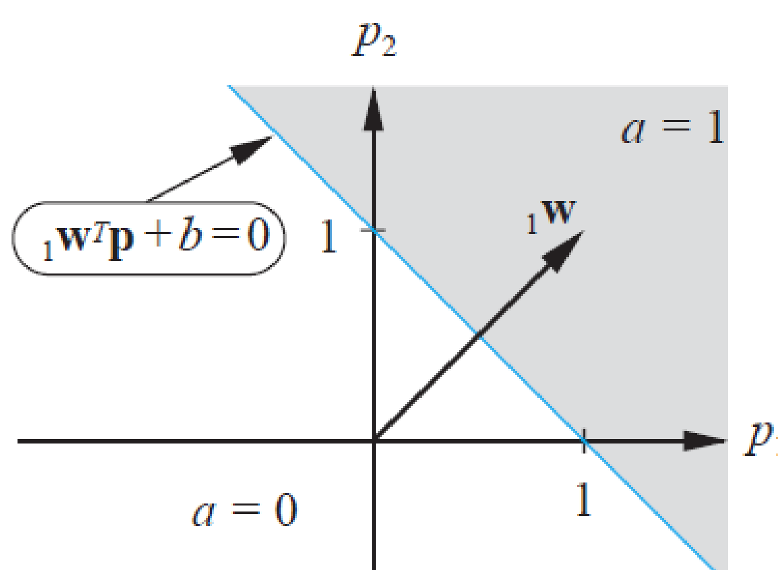
- To find the p_2 intercept, set $p_1 = 0$

$$p_2 = -\frac{b}{w_{12}} = -\frac{-1}{1} = 1$$

- The resulting decision boundary is
- For the input $p = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$

$$a = activation(w_i^T p + b)$$

$$= activation([1,1]\begin{bmatrix} 2 \\ 0 \end{bmatrix} - 1) = 1$$



Perceptron Learning Rule

- For the input $p = \begin{bmatrix} -2 \\ 0 \end{bmatrix}$

$$a = activation(w_i^T p + b) \qquad w_{11} = 1, w_{12} = 1 \text{ and } b = \text{-}1$$

$$a = activation\left([1,1] \begin{bmatrix} -2 \\ 0 \end{bmatrix} - 1 \right) = activation(-3) = 0$$

- For the input

$$p = \begin{bmatrix} -2 \\ +4 \end{bmatrix} \quad a = activation\left([1,1] \begin{bmatrix} -2 \\ 4 \end{bmatrix} - 1 \right) = activation(1) = 1$$

Perceptron

- If the vector inputs are three dimensional with a single neuron then

$$y = activation\left(\left[w_{11}, w_{12}, w_{13} \right] \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} + b \right)$$

Perceptron Learning Rule

- By learning rule we mean a procedure for updating the weights and biases of a network to perform some specific task.
- Perceptron learning algorithm converges to a correct W within a finite number of iterations, k, over the data if the classes are linearly separable.
- There are many types of neural network learning rules. They fall into three broad categories: **supervised learning**, **unsupervised learning** and **reinforcement (or graded) learning.**

Perceptron Learning Rule

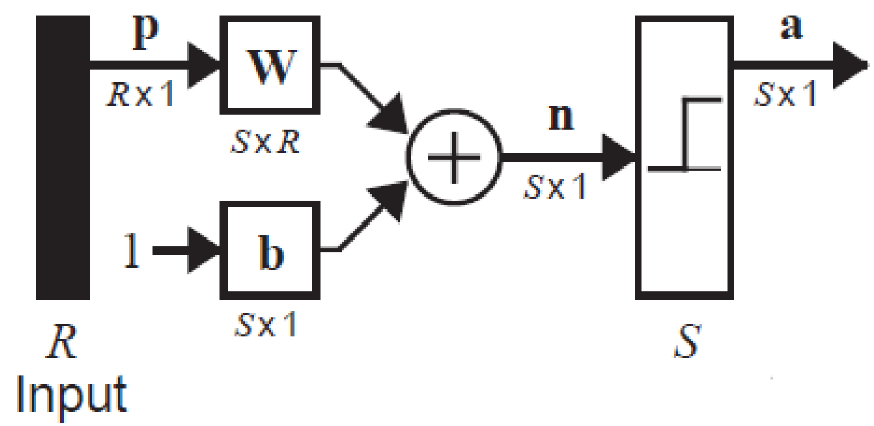
- In **supervised learning**, the learning rule is provided with a set of examples (the training set) of proper network behavior: $\{p_1, t_1\}, \{p_2, t_2\}, \ldots \{p_n, t_n\}$

'p_n' is the input to the model and 't_n' is corresponding correct (target) output.

- **In unsupervised learning**, the weights and biases are modified in response to network inputs only. There are no target outputs available.

Perceptron Learning Rule

- Perceptron Architecture:



$$ {}_i\mathbf{w} = \begin{bmatrix} w_{i,1} \\ w_{i,2} \\ \vdots \\ w_{i,R} \end{bmatrix}. $$

$$ \mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}. $$

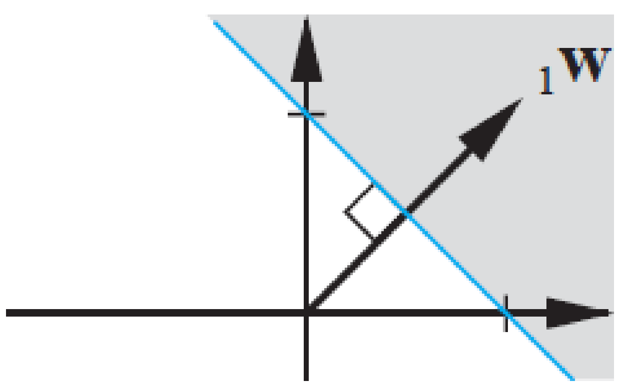
$$ \mathbf{W} = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \\ \vdots \\ {}_S\mathbf{w}^T \end{bmatrix}. $$

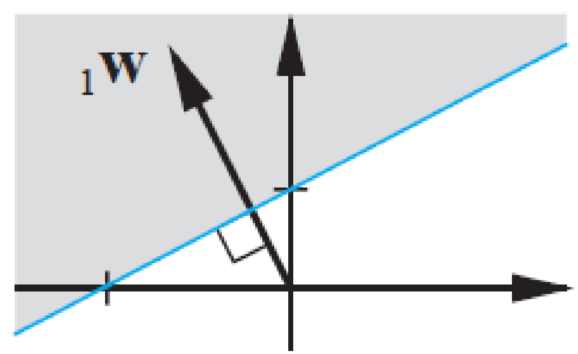
$$ a_i = Activation(WP + b) $$

$$ a_i = Activation(w_i^T p + b) $$

Perceptron Learning Rule

- For all points on the boundary, the inner product of the input vector with the weight vector is the same.
- This implies that these input vectors will all have the same projection onto the weight vector, so they must lie on a line orthogonal to the weight vector.
- Therefore the weight vector will always point toward the region where the neuron output is 1.



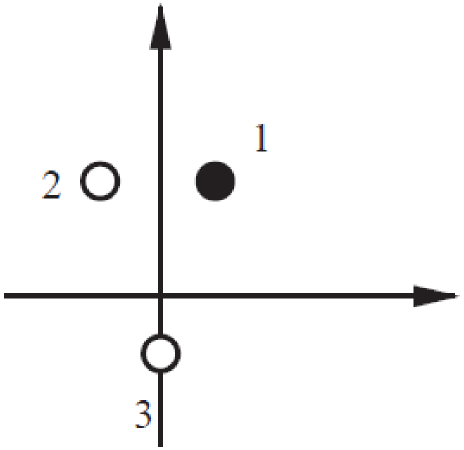


Perceptron Learning Rule

- Perceptron learning rule we will begin with a simple test problem.
- Let the input/target pairs are

$$\left\{\mathbf{p}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, t_1 = 1\right\} \left\{\mathbf{p}_2 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, t_2 = 0\right\} \left\{\mathbf{p}_3 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, t_3 = 0\right\}.$$

- There must be an allowable decision boundary that can separate the vectors p_2 and p_3 from p_1.



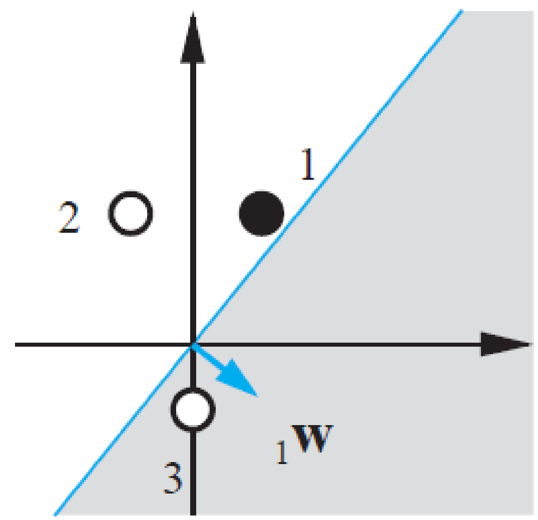
Constructing Learning Rules

- Let set the weight vectors randomly $w_{11} = \begin{bmatrix} 1 \\ -0.8 \end{bmatrix}$
- We begin with P1.

$$a = hardlim({}_1\mathbf{w}^T\mathbf{p}_1) = hardlim\left(\begin{bmatrix} 1.0 & -0.8 \end{bmatrix}\begin{bmatrix} 1 \\ 2 \end{bmatrix}\right)$$

$$a = hardlim(-0.6) = 0 .$$

- The network has not returned
- the correct value.
- **The network output is 0**,
- while the target response is $t_1 = 1$



Perceptron Learning Rule

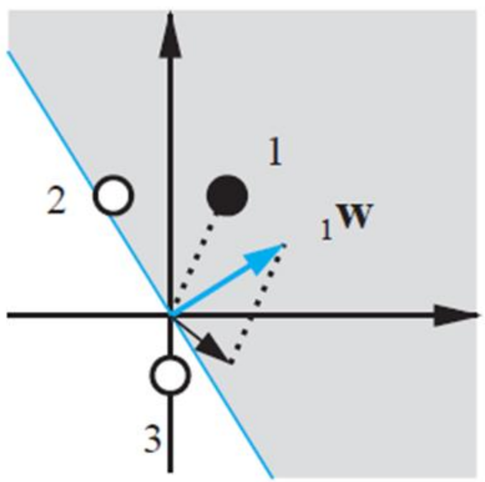
- Let add p_1 to w_{11} with the condition

$$\text{If } t = 1 \text{ and } a = 0, \text{ then } {}_{1}\mathbf{w}^{new} = {}_{1}\mathbf{w}^{old} + \mathbf{p}.$$

$${}_{1}\mathbf{w}^{new} = {}_{1}\mathbf{w}^{old} + \mathbf{p}_1 = \begin{bmatrix} 1.0 \\ -0.8 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2.0 \\ 1.2 \end{bmatrix}.$$

- We now move on to the next input vector and will continue making changes to the weights and cycling through the inputs until they are all classified correctly.
- The next input vector is p_2.

$$a = activation\left([2,1.2]\begin{bmatrix} -1 \\ 2 \end{bmatrix} \right)$$
$$= activation(0.4) = 1$$



- A class 0 vector was misclassified as a 1.

Perceptron Learning Rule

- Since we would now like to move the weight vector w_{11} away from the input, we can simply change the addition in to subtraction.

$$\text{If } t = 0 \text{ and } a = 1, \text{ then } \mathbf{w}_{11}^{new} = \mathbf{w}_{11}^{old} - \mathbf{p}.$$

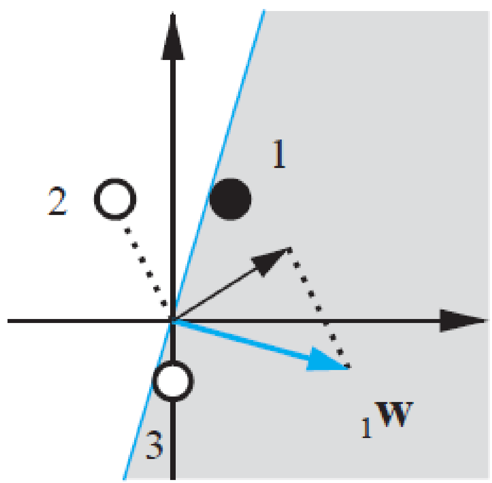
- If we apply this to the test problem we find

$$\mathbf{w}_{11}^{new} = \mathbf{w}_{11}^{old} - \mathbf{p}_2 = \begin{bmatrix} 2.0 \\ 1.2 \end{bmatrix} - \begin{bmatrix} -1 \\ 2 \end{bmatrix} = \begin{bmatrix} 3.0 \\ -0.8 \end{bmatrix}$$

- The next input vector is p_3

$$a = activation\left([3.0, -0.8] \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right)$$

$$= activation(0.8) = 1$$



Perceptron Learning Rule

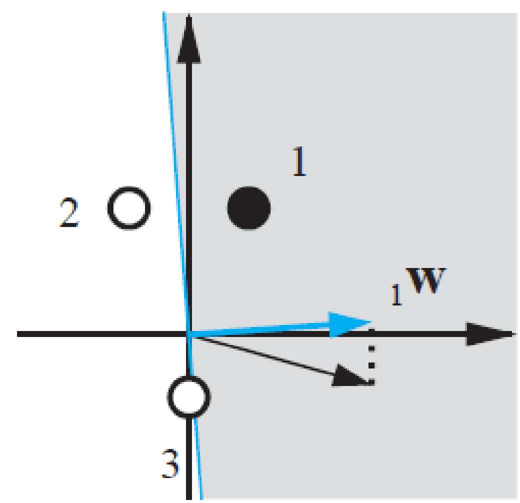
- The current weight w_{11} results in a decision boundary that misclassifies p_3 .
- So w_{11} will updated using the same condition.

$$\mathbf{w}_{11}^{new} = \mathbf{w}_{11}^{old} - \mathbf{p}_3 = \begin{bmatrix} 3.0 \\ -0.8 \end{bmatrix} - \begin{bmatrix} 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 3.0 \\ 0.2 \end{bmatrix}$$

$$a = activation\left([3.0, 0.2] \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right)$$

$$= activation(-0.2) = 0$$

If $t = a$, then $\mathbf{w}_{11}^{new} = \mathbf{w}_{11}^{old}$



Perceptron Learning Rule

- Here are the three rules, which cover all possible combinations of output and target values:

$$\text{If } t = 1 \text{ and } a = 0, \text{ then } \quad {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}.$$

$$\text{If } t = 0 \text{ and } a = 1, \text{ then } \quad {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}.$$

$$\text{If } t = a, \text{ then } \quad {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old}.$$

- Now if we take **e = t – a**

Perceptron Learning Rule

- We have
$$\text{If } e = 1, \text{ then } \mathbf{w}_{11}^{new} = \mathbf{w}_{11}^{old} + \mathbf{p}.$$
$$\text{If } e = -1, \text{ then } \mathbf{w}_{11}^{new} = \mathbf{w}_{11}^{old} - \mathbf{p}.$$
$$\text{If } e = 0, \text{ then } \mathbf{w}_{11}^{new} = \mathbf{w}_{11}^{old}.$$

- we can see that the sign of **P** is the same as the sign on the error, e. Furthermore, the absence of in the third rule corresponds to an error e of 0.

$$\mathbf{w}_{11}^{new} = \mathbf{w}_{11}^{old} + e\mathbf{p} = \mathbf{w}_{11}^{old} + (t - a)\mathbf{p}.$$

$$b^{new} = b^{old} + e.$$

Perceptron Learning Rule

- The perceptron rule can be written conveniently in matrix notation

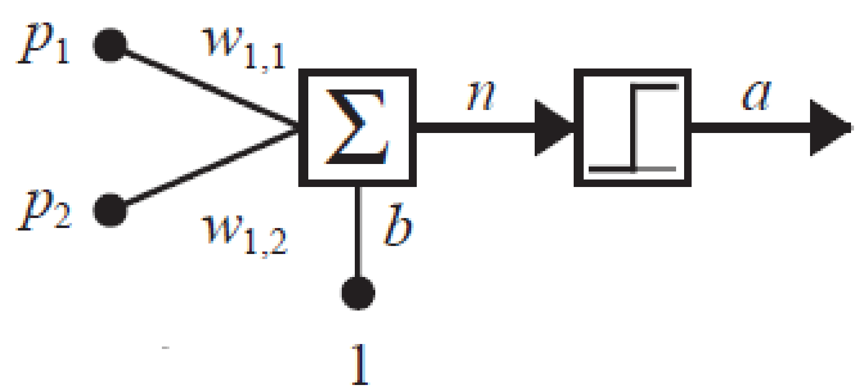
$$W^{new} = W^{old} + eP^{T}$$

$$b^{new} = b^{old} + e$$

- **Training Multiple-Neuron Perceptron:**

$$W_i^{new} = W_i^{old} + e_i P$$

$$b_i^{new} = b_i^{old} + e_i$$



Perceptron Learning Rule

- Example: $\left\{ \mathbf{p}_1 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}, t_1 = \begin{bmatrix} 0 \end{bmatrix} \right\}$ $\left\{ \mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, t_2 = \begin{bmatrix} 1 \end{bmatrix} \right\}$

- Typically the weights and biases are initialized to small random numbers. Let

$$W = [0.5 \text{ -1 -0.5}] \qquad b = 0.5$$

- The first step is to apply the first input vector P1

$$a = hardlim(\mathbf{W}\mathbf{p}_1 + b) = hardlim\left(\begin{bmatrix} 0.5 & -1 & -0.5 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} + 0.5 \right)$$

$$= hardlim(2.5) = 1$$

Perceptron Learning Rule

- Then we calculate the error: $e = t_1 - a = 0 - 1 = -1$
- The weight update is

$$\mathbf{W}^{new} = \mathbf{W}^{old} + e\mathbf{p}^T = \begin{bmatrix} 0.5 & -1 & -0.5 \end{bmatrix} + (-1)\begin{bmatrix} 1 & -1 & -1 \end{bmatrix}$$
$$= \begin{bmatrix} -0.5 & 0 & 0.5 \end{bmatrix} .$$

- The bias update is

$$b^{new} = b^{old} + e = 0.5 + (-1) = -0.5 .$$

- This completes the first iteration.

Perceptron Learning Rule

- The second iteration of the perceptron rule is:

$$a = hardlim(\mathbf{W}\mathbf{p}_2 + b) = hardlim\left(\begin{bmatrix} -0.5 & 0 & 0.5 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + (-0.5)\right)$$

$$= hardlim(-0.5) = 0$$

$$e = t_2 - a = 1 - 0 = 1$$

$$\mathbf{W}^{new} = \mathbf{W}^{old} + e\mathbf{p}^T = \begin{bmatrix} -0.5 & 0 & 0.5 \end{bmatrix} + 1\begin{bmatrix} 1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 0.5 & 1 & -0.5 \end{bmatrix}$$

$$b^{new} = b^{old} + e = -0.5 + 1 = 0.5$$

Perceptron Learning Rule

- The third iteration begins again with the first input

$$a = hardlim(\mathbf{W}\mathbf{p}_1 + b) = hardlim\left(\begin{bmatrix} 0.5 & 1 & -0.5 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} + 0.5\right)$$

$$= hardlim(0.5) = 1$$

$$e = t_1 - a = 0 - 1 = -1$$

$$\mathbf{W}^{new} = \mathbf{W}^{old} + e\mathbf{p}^T = \begin{bmatrix} 0.5 & 1 & -0.5 \end{bmatrix} + (-1)\begin{bmatrix} 1 & -1 & -1 \end{bmatrix}$$
$$= \begin{bmatrix} -0.5 & 2 & 0.5 \end{bmatrix}$$

$$b^{new} = b^{old} + e = 0.5 + (-1) = -0.5 .$$

Perceptron Learning Rule

- This weight and bias is applied to 2nd input

$$a = \text{hardlim}\left([-0.5\ 2\ 0.5]\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + -0.5 \right) = \text{hardlim}(.5) = 1$$

$$e = t_2 - a = 1 - 1 = 0$$

$$W^{new} = W^{old} + eP^{T} \qquad b^{new} = b^{old} + e$$

$$W^{new} = [-0.5\ 2\ 0.5] \qquad b^{new} = -0.5$$

Matlab implementation

- % You can create a perceptron with the following:
- %
- % net = perceptron;
- % net = configure(net,P,T);
- % where input arguments are as follows:
- %
- % P is an R-by-Q matrix of Q input vectors of R elements each.
- %
- % T is an S-by-Q matrix of Q target vectors of S elements each.

- P = [0 2];
- T = [0 1];
- net = perceptron;
- net = configure(net,P,T);
- inputweights = net.inputweights{1,1}
- biases = net.biases{1}

Matlab implementation

- %Start with a single neuron having an input vector with just two elements.
- net = perceptron;
- net = configure(net,[0;0],0);
- net.b{1} = [0];
- w = [1 -0.8];
- net.IW{1,1} = w;
- % The input target pair is given by
- p = [1; 2];
- t = [1];
- %compute the output and error
- a = net(p)
- e = t-a
- dw = learnp(w,p,[],[],[],[],e,[],[],[],[],[])
- wnew = w + dw

Perceptron learning

- Solve the following classification problem with the perceptron rule.

$$\left\{\mathbf{p}_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, t_1 = 0\right\} \left\{\mathbf{p}_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, t_2 = 1\right\} \left\{\mathbf{p}_3 = \begin{bmatrix} -2 \\ 2 \end{bmatrix}, t_3 = 0\right\} \left\{\mathbf{p}_4 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, t_4 = 1\right\}$$

- Using initial weights and bias, we can start

$$\mathbf{W}(0) = \begin{bmatrix} 0 & 0 \end{bmatrix} \qquad b(0) = 0 .$$

- For the first input vector p1, using the initial weights and bias, the output a is

$$a = hardlim(\mathbf{W}(0)\mathbf{p}_1 + b(0))$$

$$= hardlim\left(\begin{bmatrix} 0 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \end{bmatrix} + 0\right) = hardlim(0) = 1$$

Perceptron learning

- The output a does not equal the target value t1.

$$e = t_1 - a = 0 - 1 = -1$$

$$\mathbf{W}(1) = \mathbf{W}(0) + e\mathbf{p}_1^T = \begin{bmatrix} 0 & 0 \end{bmatrix} + (-1)\begin{bmatrix} 2 & 2 \end{bmatrix} = \begin{bmatrix} -2 & -2 \end{bmatrix}$$

$$b(1) = b(0) + e = 0 + (-1) = -1$$

- Apply the second input vector p2 , using the updated weights and bias.

$$a = hardlim(\mathbf{W}(1)\mathbf{p}_2 + b(1))$$

$$= hardlim\left(\begin{bmatrix} -2 & -2 \end{bmatrix}\begin{bmatrix} 1 \\ -2 \end{bmatrix} - 1\right) = hardlim(1) = 1$$

- The output a is equal to the target t2, hence no change in weight and bias.

Perceptron learning

- Hence $\mathbf{W}(2) = \mathbf{W}(1)$

$$b(2) = b(1)$$

- We now apply this to the third input vector p3.

$$a = hardlim(\mathbf{W}(2)\mathbf{p}_3 + b(2))$$

$$= hardlim\left(\begin{bmatrix} -2 & -2 \end{bmatrix} \begin{bmatrix} -2 \\ 2 \end{bmatrix} - 1\right) = hardlim(-1) = 0$$

- The output in response to input vector p3 is equal to the target t3. Hence no change.

$$\mathbf{W}(3) = \mathbf{W}(2)$$

$$b(3) = b(2)$$

Perceptron learning

- Move on to the last input vector p4 using this weight.

$$a = hardlim(\mathbf{W}(3)\mathbf{p}_4 + b(3))$$

$$= hardlim\left(\begin{bmatrix}-2 & -2\end{bmatrix}\begin{bmatrix}-1\\1\end{bmatrix} - 1\right) = hardlim(-1) = 0$$

- This time the output a does not equal the appropriate target t4, update the weight

$$e = t_4 - a = 1 - 0 = 1$$

$$\mathbf{W}(4) = \mathbf{W}(3) + e\mathbf{p}_4^T = \begin{bmatrix}-2 & -2\end{bmatrix} + (1)\begin{bmatrix}-1 & 1\end{bmatrix} = \begin{bmatrix}-3 & -1\end{bmatrix}$$

$$b(4) = b(3) + e = -1 + 1 = 0$$

Perceptron learning

- We now must check the first vector p1 again.

$$a = hardlim(\mathbf{W}(4)\mathbf{p}_1 + b(4))$$

$$= hardlim\left(\begin{bmatrix}-3 & -1\end{bmatrix}\begin{bmatrix}2\\2\end{bmatrix} + 0\right) = hardlim(-8) = 0$$

- Therefore there are no changes.

$$\mathbf{W}(5) = \mathbf{W}(4)$$

$$b(5) = b(4)$$

Now apply this to p2.

$$a = hardlim(\mathbf{W}(5)\mathbf{p}_2 + b(5))$$

$$= hardlim\left(\begin{bmatrix}-3 & -1\end{bmatrix}\begin{bmatrix}1\\-2\end{bmatrix} + 0\right) = hardlim(-1) = 0$$

Perceptron learning

- Calculating the error

$$e = t_2 - a = 1 - 0 = 1$$

$$\mathbf{W}(6) = \mathbf{W}(5) + e\mathbf{p}_2^T = \begin{bmatrix} -3 & -1 \end{bmatrix} + (1)\begin{bmatrix} 1 & -2 \end{bmatrix} = \begin{bmatrix} -2 & -3 \end{bmatrix}$$

$$b(6) = b(5) + e = 0 + 1 = 1.$$

- Apply this to p3

$$a = hardlim(\mathbf{W}(6)\mathbf{p}_3 + b(6)) = hardlim\left(\begin{bmatrix} -2 & -3 \end{bmatrix}\begin{bmatrix} -2 \\ 2 \end{bmatrix} + 1\right) = 0 = t_3$$

$$a = hardlim(\mathbf{W}(6)\mathbf{p}_4 + b(6)) = hardlim\left(\begin{bmatrix} -2 & -3 \end{bmatrix}\begin{bmatrix} -1 \\ 1 \end{bmatrix} + 1\right) = 1 = t_4$$

$$a = hardlim(\mathbf{W}(6)\mathbf{p}_1 + b(6)) = hardlim\left(\begin{bmatrix} -2 & -3 \end{bmatrix}\begin{bmatrix} 2 \\ 2 \end{bmatrix} + 1\right) = 0 = t_1$$

Perceptron learning

$$a = hardlim(\mathbf{W}(6)\mathbf{p}_2 + b(6)) = hardlim\left(\begin{bmatrix}-2 & -3\end{bmatrix}\begin{bmatrix}1 \\ -2\end{bmatrix} + 1\right) = 1 = t_2$$

- Therefore the algorithm has converged. The final solution is

$$\mathbf{W} = \begin{bmatrix}-2 & -3\end{bmatrix} \qquad b = 1\,.$$

- Now we can graph the training data and the decision boundary of the solution.

$$n = \mathbf{W}\mathbf{p} + b = w_{1,1}p_1 + w_{1,2}p_2 + b = -2p_1 - 3p_2 + 1 = 0\,.$$

To find the p_2 intercept of the decision boundary, set $p_1 = 0$:

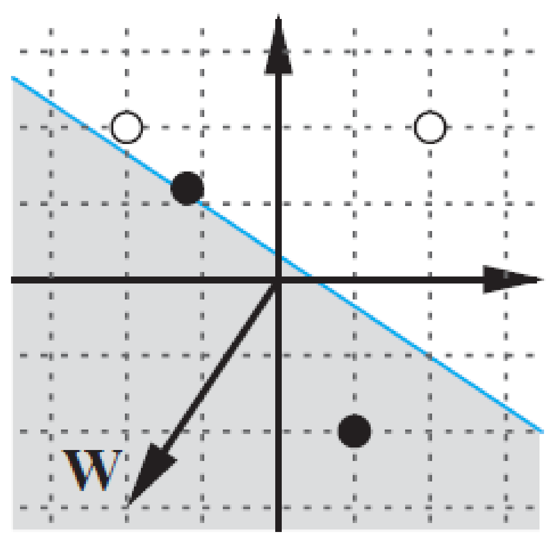
$$p_2 = -\frac{b}{w_{1,2}} = -\frac{1}{-3} = \frac{1}{3} \qquad \text{if } p_1 = 0$$

Perceptron learning

To find the p_1 intercept, set $p_2 = 0$:

$$p_1 = -\frac{b}{w_{1,1}} = -\frac{1}{-2} = \frac{1}{2} \qquad \text{if } p_2 = 0$$

- The resulting decision boundary is like



Matlab implementation

- Now

$$\left\{\mathbf{p}_1 = \begin{bmatrix}2\\2\end{bmatrix}, t_1 = 0\right\}\left\{\mathbf{p}_2 = \begin{bmatrix}1\\-2\end{bmatrix}, t_2 = 1\right\}\left\{\mathbf{p}_3 = \begin{bmatrix}-2\\2\end{bmatrix}, t_3 = 0\right\}\left\{\mathbf{p}_4 = \begin{bmatrix}-1\\1\end{bmatrix}, t_4 = 1\right\}$$

- Let the initial values are W(0) and b(0)

$$\mathbf{W}(0) = [0 \quad 0] \quad b(0) = 0$$

- Then starting with initial weights

$$\alpha = hardlim(\mathbf{W}(0)\mathbf{p}_1 + b(0))$$

$$= hardlim\left([0 \quad 0]\begin{bmatrix}2\\2\end{bmatrix} + 0\right) = hardlim(0) = 1$$

- The output **a** does not equal the target value **t1.**

Matlab implementation

- We get the error $e = t_1 - \alpha = 0 - 1 = -1$

$$\Delta\mathbf{W} = e\mathbf{p}_1^T = (-1)[2 \quad 2] = [-2 \quad -2]$$

$$\Delta b = e = (-1) = -1$$

- Updated weights

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^T = [0 \quad 0] + [-2 \quad -2] = [-2 \quad -2] = \mathbf{W}(1)$$

$$b^{new} = b^{old} + e = 0 + (-1) = -1 = b(1)$$

$$\alpha = hardlim(\mathbf{W}(1)\mathbf{p}_2 + b(1))$$

$$= hardlim\left([-2 \quad -2]\begin{bmatrix} 1 \\ -2 \end{bmatrix} - 1\right) = hardlim(1) = 1$$

- No change in weight in next iteration. The final values are W(6) = [−2 −3] and b(6) = 1.

Matlab implementation

- Matlab:
- net = perceptron;
- p = [2; 2];
- t = [0];
- net.trainParam.epochs = 1;
- net = train(net,p,t);
- w = net.iw{1,1}, b = net.b{1}
- w =
- -2 -2
- b =
- -1

Matlab implementation

- p = [[2;2] [1;-2] [-2;2] [-1;1]]
- t = [0 1 0 1]
- net = perceptron;
- net.trainParam.epochs = 10;
- net = train(net,p,t);
- w = net.iw{1,1}, b = net.b{1}

- w = -2 -3
- b = 1

Matlab implementation

- Now classify with the perceptron rule

$$\left\{\mathbf{p}_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, t_1 = 0\right\} \left\{\mathbf{p}_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, t_2 = 1\right\} \left\{\mathbf{p}_3 = \begin{bmatrix} -2 \\ 2 \end{bmatrix}, t_3 = 0\right\} \left\{\mathbf{p}_4 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, t_4 = 1\right\}$$

Use the initial weights and bias:

$$\mathbf{W}(0) = \begin{bmatrix} 0 & 0 \end{bmatrix} \qquad b(0) = 0.$$

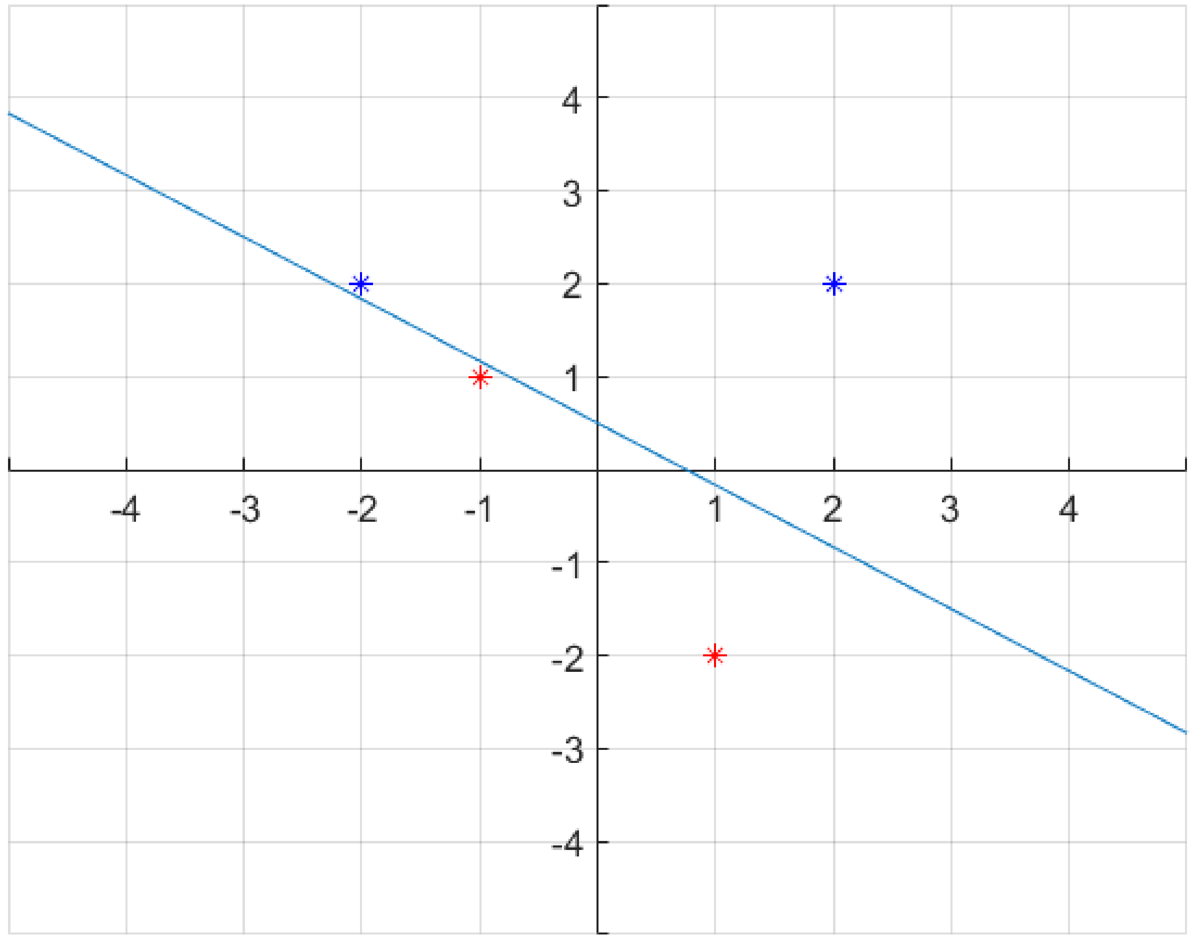
Matlab implementation

- p = [[2;2] [1;-2] [-2;2] [-1;1]]
- t = [0 1 0 1]
- net = perceptron;
- net.trainParam.epochs = 1;
- net = train(net,p,t);
- a = net(p);
- net.trainParam.epochs = 10;
- net = train(net,p,t);
- w = net.iw{1,1}, b = net.b{1}
- %plotting the line
- p1=-b/w(1),p2=-b/w(2),
- %plot([p1 0], [0 p2])
- A = [p1 0];
- B = [0 p2];
- %plot(A,B,'*')
- hold on
- plot(2,2,'*','color','blue')
- hold on
- plot(1,-2,'*','color','red')
- hold on
- plot(-2,2,'*','color','blue')
- hold on

Matlab implementation

- plot(-1,1,'*','color','red')
- axis([-5 5 -5 5])
- ax = gca;
- ax.XAxisLocation = 'origin';
- ax.YAxisLocation = 'origin';
- hold on
- %line(A,B)
- %hold off
- xlim = get(gca,'XLim');
- m = (0-p2)/(p1-0);
- n = p1;
- y1 = m*xlim(1) + n;
- y2 = m*xlim(2) + n;
- grid on, hold on
- line([xlim(1) xlim(2)],[y1 y2])
- hold off
-

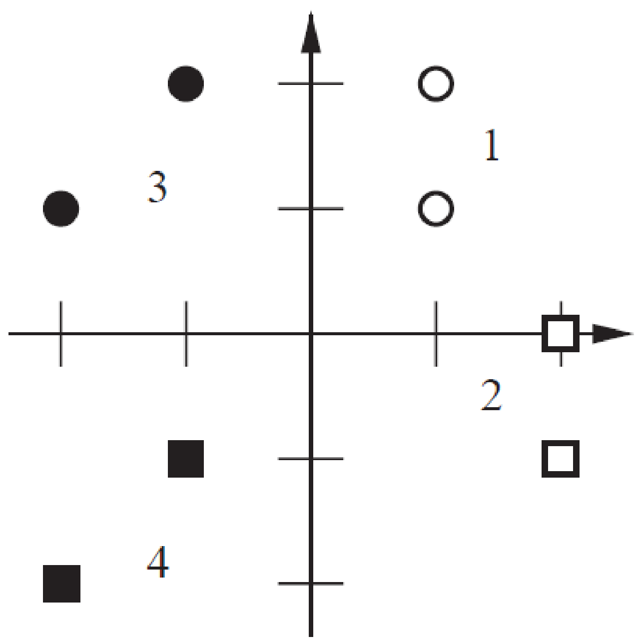
Matlab implementation



Example with two neurons

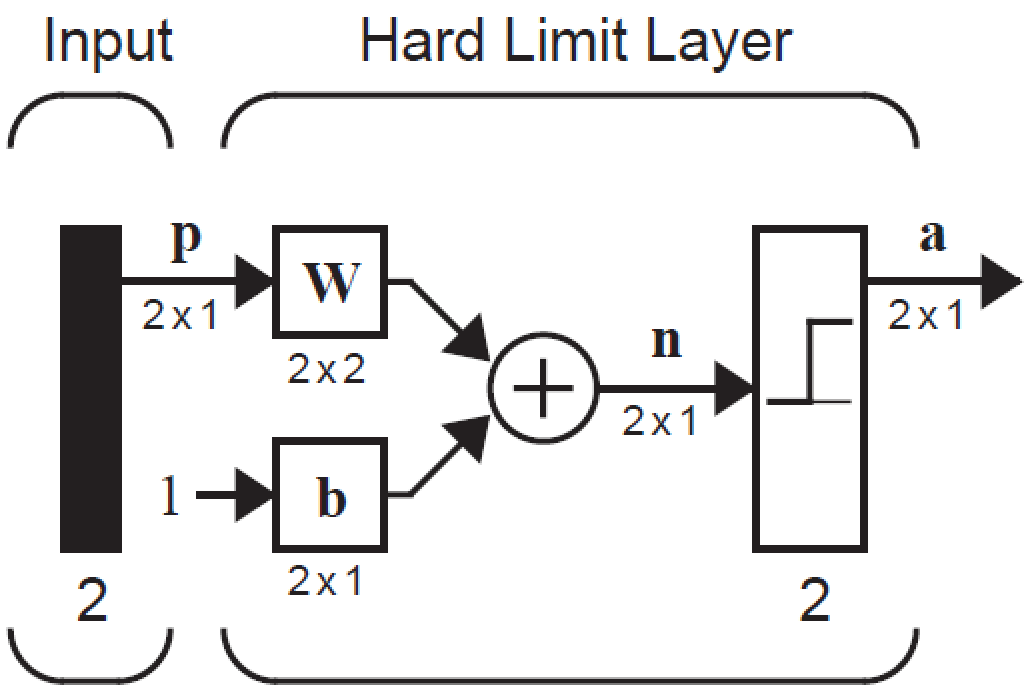
- $$\text{class 1:}\left\{\mathbf{p}_1 = \begin{bmatrix}1\\1\end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix}1\\2\end{bmatrix}\right\}, \text{class 2:}\left\{\mathbf{p}_3 = \begin{bmatrix}2\\-1\end{bmatrix}, \mathbf{p}_4 = \begin{bmatrix}2\\0\end{bmatrix}\right\},$$

$$\text{class 3:}\left\{\mathbf{p}_5 = \begin{bmatrix}-1\\2\end{bmatrix}, \mathbf{p}_6 = \begin{bmatrix}-2\\1\end{bmatrix}\right\}, \text{class 4:}\left\{\mathbf{p}_7 = \begin{bmatrix}-1\\-1\end{bmatrix}, \mathbf{p}_8 = \begin{bmatrix}-2\\-2\end{bmatrix}\right\}.$$



Example

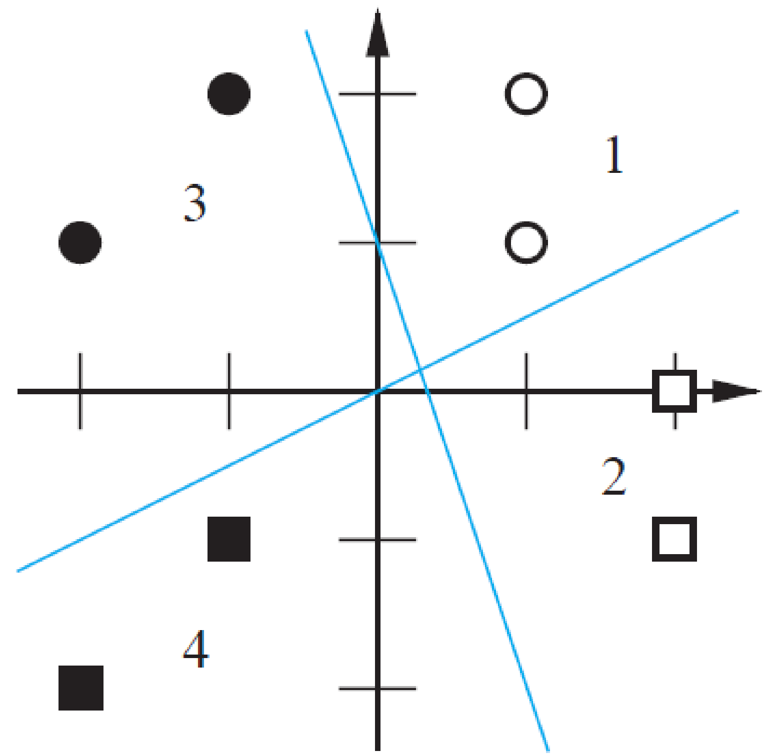
- To solve a problem with four classes of input vector we will need a perceptron with at least two neurons.



$$\mathbf{a} = \mathrm{hardlim}(\mathbf{W}\mathbf{p} + \mathbf{b})$$

Example

- A two-neuron perceptron creates two decision boundaries.
- we need to have one decision boundary divide the four classes into two sets of two. The remaining boundary must then isolate each class.
- The weight vectors should be orthogonal to the decision boundaries and should point toward the regions where the neuron outputs are 1



Example

- This solution corresponds to target values of

$$\text{class 1:}\left\{\mathbf{t}_1 = \begin{bmatrix}0\\0\end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix}0\\0\end{bmatrix}\right\}, \text{class 2:}\left\{\mathbf{t}_3 = \begin{bmatrix}0\\1\end{bmatrix}, \mathbf{t}_4 = \begin{bmatrix}0\\1\end{bmatrix}\right\},$$

$$\text{class 3:}\left\{\mathbf{t}_5 = \begin{bmatrix}1\\0\end{bmatrix}, \mathbf{t}_6 = \begin{bmatrix}1\\0\end{bmatrix}\right\}, \text{class 4:}\left\{\mathbf{t}_7 = \begin{bmatrix}1\\1\end{bmatrix}, \mathbf{t}_8 = \begin{bmatrix}1\\1\end{bmatrix}\right\}.$$

Let initialized the weight vectors as ${}_1\mathbf{w} = \begin{bmatrix}-3\\-1\end{bmatrix}$ and ${}_2\mathbf{w} = \begin{bmatrix}1\\-2\end{bmatrix}$.

- Now we can calculate the bias by picking a point on a boundary.

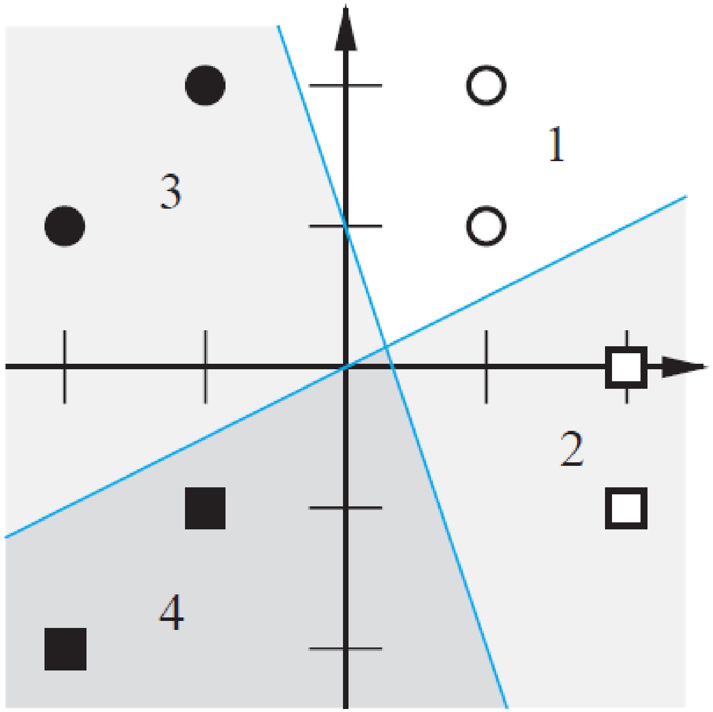
$$b_1 = -{}_1\mathbf{w}^T\mathbf{p} = -\begin{bmatrix}-3 & -1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} = 1, \qquad b_2 = -{}_2\mathbf{w}^T\mathbf{p} = -\begin{bmatrix}1 & -2\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix} = 0.$$

Example

- In matrix form we have

$$\mathbf{W} = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \end{bmatrix} = \begin{bmatrix} -3 & -1 \\ 1 & -2 \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} 1 \\ 0 \end{bmatrix},$$

- With this the neural network
- Can be iterated.



Example

- Let $\left\{\mathbf{p}_1 = \begin{bmatrix}1\\1\end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix}0\\0\end{bmatrix}\right\} \left\{\mathbf{p}_2 = \begin{bmatrix}1\\2\end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix}0\\0\end{bmatrix}\right\} \left\{\mathbf{p}_3 = \begin{bmatrix}2\\-1\end{bmatrix}, \mathbf{t}_3 = \begin{bmatrix}0\\1\end{bmatrix}\right\}$

$$\left\{\mathbf{p}_4 = \begin{bmatrix}2\\0\end{bmatrix}, \mathbf{t}_4 = \begin{bmatrix}0\\1\end{bmatrix}\right\} \left\{\mathbf{p}_5 = \begin{bmatrix}-1\\2\end{bmatrix}, \mathbf{t}_5 = \begin{bmatrix}1\\0\end{bmatrix}\right\} \left\{\mathbf{p}_6 = \begin{bmatrix}-2\\1\end{bmatrix}, \mathbf{t}_6 = \begin{bmatrix}1\\0\end{bmatrix}\right\}$$

$$\left\{\mathbf{p}_7 = \begin{bmatrix}-1\\-1\end{bmatrix}, \mathbf{t}_7 = \begin{bmatrix}1\\1\end{bmatrix}\right\} \left\{\mathbf{p}_8 = \begin{bmatrix}-2\\-2\end{bmatrix}, \mathbf{t}_8 = \begin{bmatrix}1\\1\end{bmatrix}\right\}.$$

with initial weights

$$\mathbf{W}(0) = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}, \ \mathbf{b}(0) = \begin{bmatrix}1\\1\end{bmatrix}.$$

Example

- The first iteration is

$$\mathbf{a} = hardlim\,(\mathbf{W}(0)\mathbf{p}_1 + \mathbf{b}(0)) = hardlim\,\left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix}\right) = \begin{bmatrix} 1 \\ 1 \end{bmatrix},$$

$$\mathbf{e} = \mathbf{t}_1 - \mathbf{a} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \end{bmatrix},$$

$$\mathbf{W}(1) = \mathbf{W}(0) + \mathbf{e}\mathbf{p}_1^T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} -1 \\ -1 \end{bmatrix}\begin{bmatrix} 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix},$$

$$\mathbf{b}(1) = \mathbf{b}(0) + \mathbf{e} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} -1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Example

- The second iteration is

$$\mathbf{a} = hardlim\,(\mathbf{W}(1)\mathbf{p}_2 + \mathbf{b}(1)) = hardlim\,(\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}) = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

$$\mathbf{e} = \mathbf{t}_2 - \mathbf{a} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

$$\mathbf{W}(2) = \mathbf{W}(1) + \mathbf{e}\mathbf{p}_2^T = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 & 2 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix},$$

$$\mathbf{b}(2) = \mathbf{b}(1) + \mathbf{e} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Example

- The third iteration is

$$\mathbf{a} = hardlim\,(\mathbf{W}(2)\mathbf{p}_3 + \mathbf{b}(2)) = hardlim\,\left(\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}\begin{bmatrix} 2 \\ -1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} 1 \\ 0 \end{bmatrix},$$

$$\mathbf{e} = \mathbf{t}_3 - \mathbf{a} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix},$$

$$\mathbf{W}(3) = \mathbf{W}(2) + \mathbf{e}\mathbf{p}_3^T = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \end{bmatrix}\begin{bmatrix} 2 & -1 \end{bmatrix} = \begin{bmatrix} -2 & 0 \\ 1 & -1 \end{bmatrix},$$

$$\mathbf{b}(3) = \mathbf{b}(2) + \mathbf{e} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}.$$

Example

- Iterations four through eight produce no changes in the weights.

$$\mathbf{W}(8) = \mathbf{W}(7) = \mathbf{W}(6) = \mathbf{W}(5) = \mathbf{W}(4) = \mathbf{W}(3)$$

$$\mathbf{b}(8) = \mathbf{b}(7) = \mathbf{b}(6) = \mathbf{b}(5) = \mathbf{b}(4) = \mathbf{b}(3)$$

- The ninth iteration produces the result:

Example

$$\mathbf{a} = hardlim\,(\mathbf{W}(8)\mathbf{p}_1 + \mathbf{b}(8)) = hardlim\,\left(\begin{bmatrix} -2 & 0 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 1 \end{bmatrix},$$

$$\mathbf{e} = \mathbf{t}_1 - \mathbf{a} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \end{bmatrix},$$

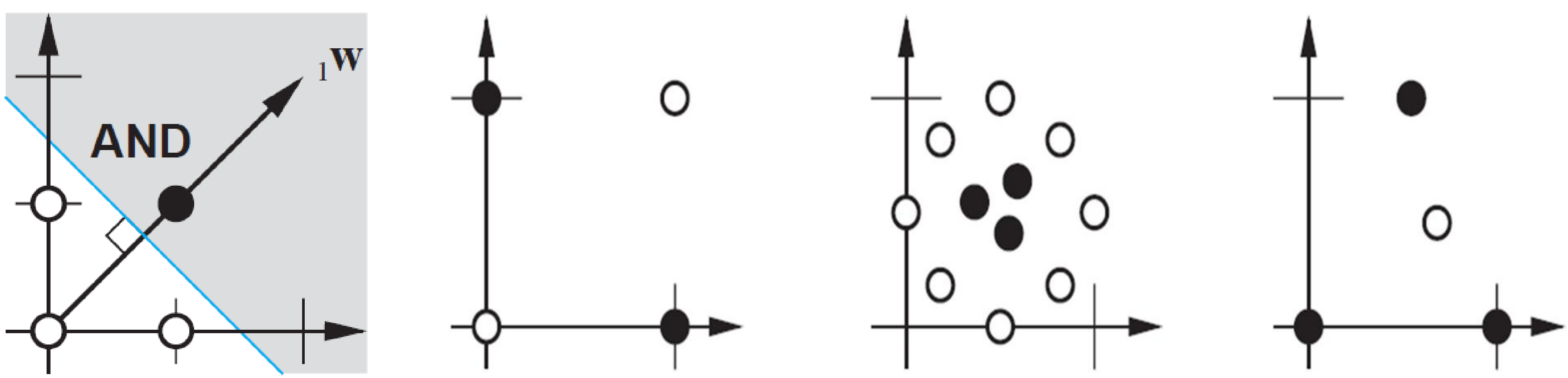
$$\mathbf{W}(9) = \mathbf{W}(8) + \mathbf{e}\mathbf{p}_1^T = \begin{bmatrix} -2 & 0 \\ 1 & -1 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix}\begin{bmatrix} 1 & 1 \end{bmatrix} = \begin{bmatrix} -2 & 0 \\ 0 & -2 \end{bmatrix},$$

$$\mathbf{b}(9) = \mathbf{b}(8) + \mathbf{e} = \begin{bmatrix} -1 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}.$$

- At this point the algorithm has converged.

Limitations of perceptron

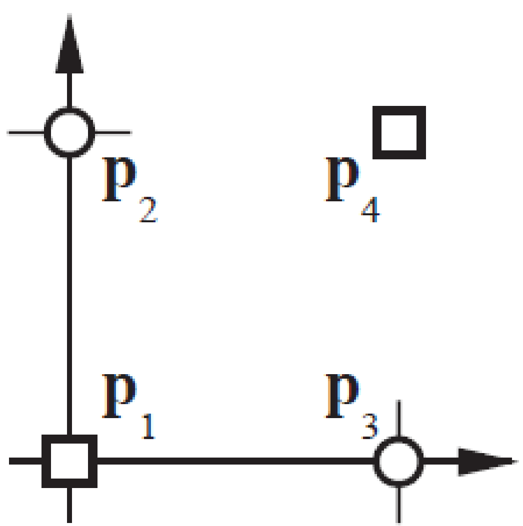
- What problems can a perceptron solve?
- The perceptron can be used to classify input vectors that can be separated by a linear boundary. We call such (linearly separable).
- The logical AND gate example is a two-dimensional example of a linearly separable problem.



Limitations of perceptron

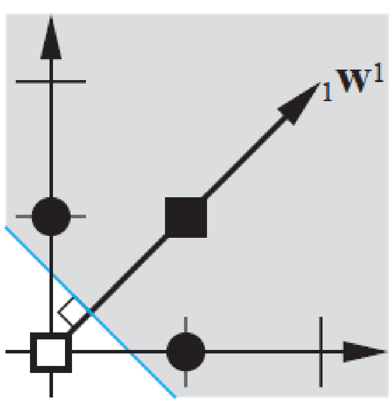
- Unfortunately, many problems are not linearly separable. The classic example is the XOR gate. The input/target pairs for the XOR gate are

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

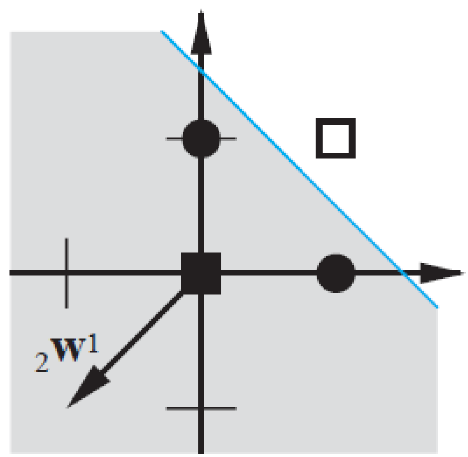


Limitations of perceptron

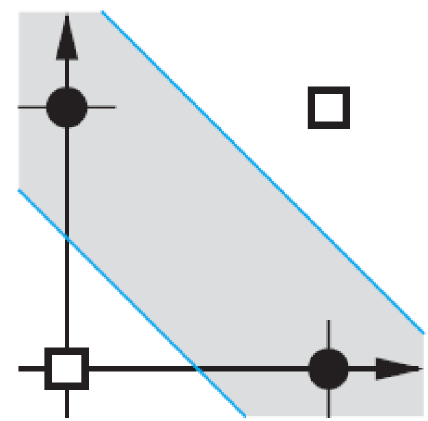
- A two-layer network can solve the XOR problem.
- One solution is to use two neurons in the first layer to create two decision boundaries.
- The first boundary separates p1 from the other patterns, and the second boundary separates p4.
- Then the second layer is used to combine the two boundaries together using an AND operation



Layer 1/Neuron 1

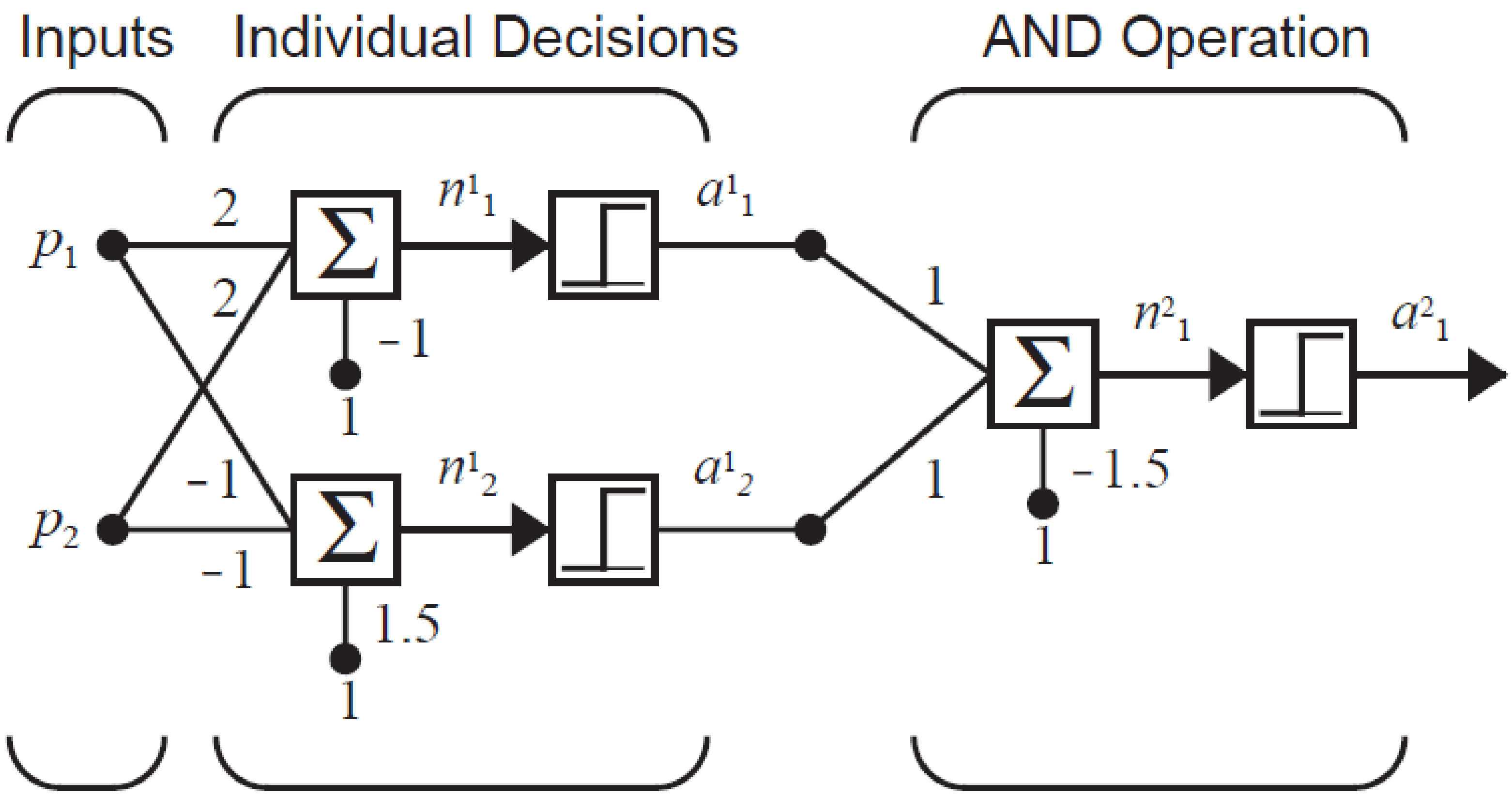


Layer 1/Neuron 2

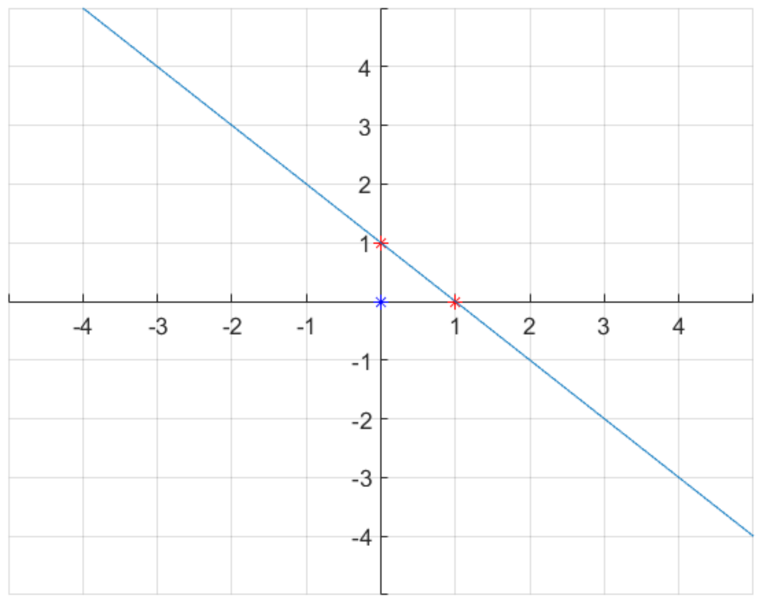


XOR gate

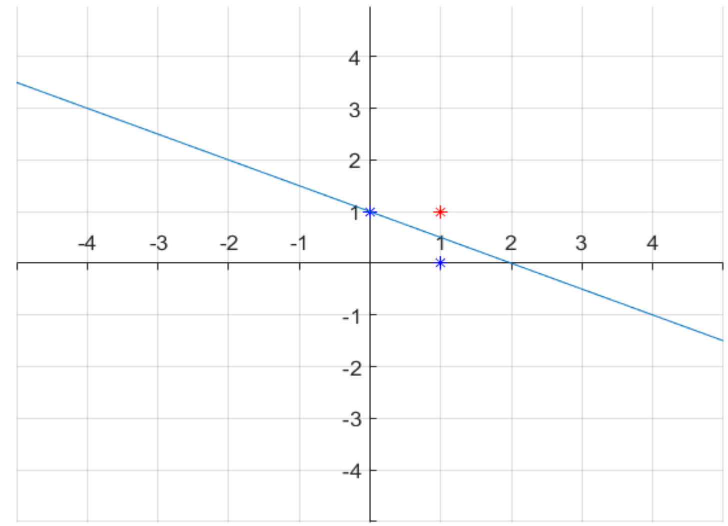
-



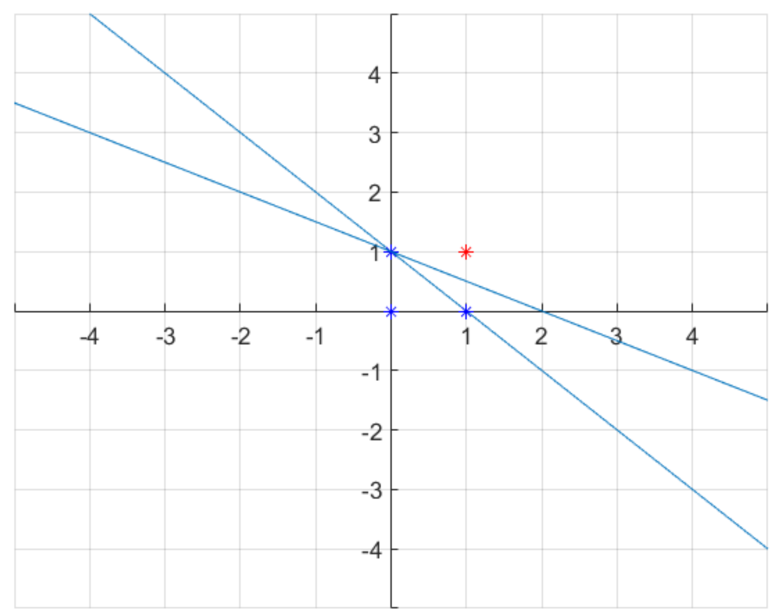
Two-Layer XOR Network



4
3
2
1
-1
-2
-3
-4
-4 -3 -2 -1 1 2 3 4



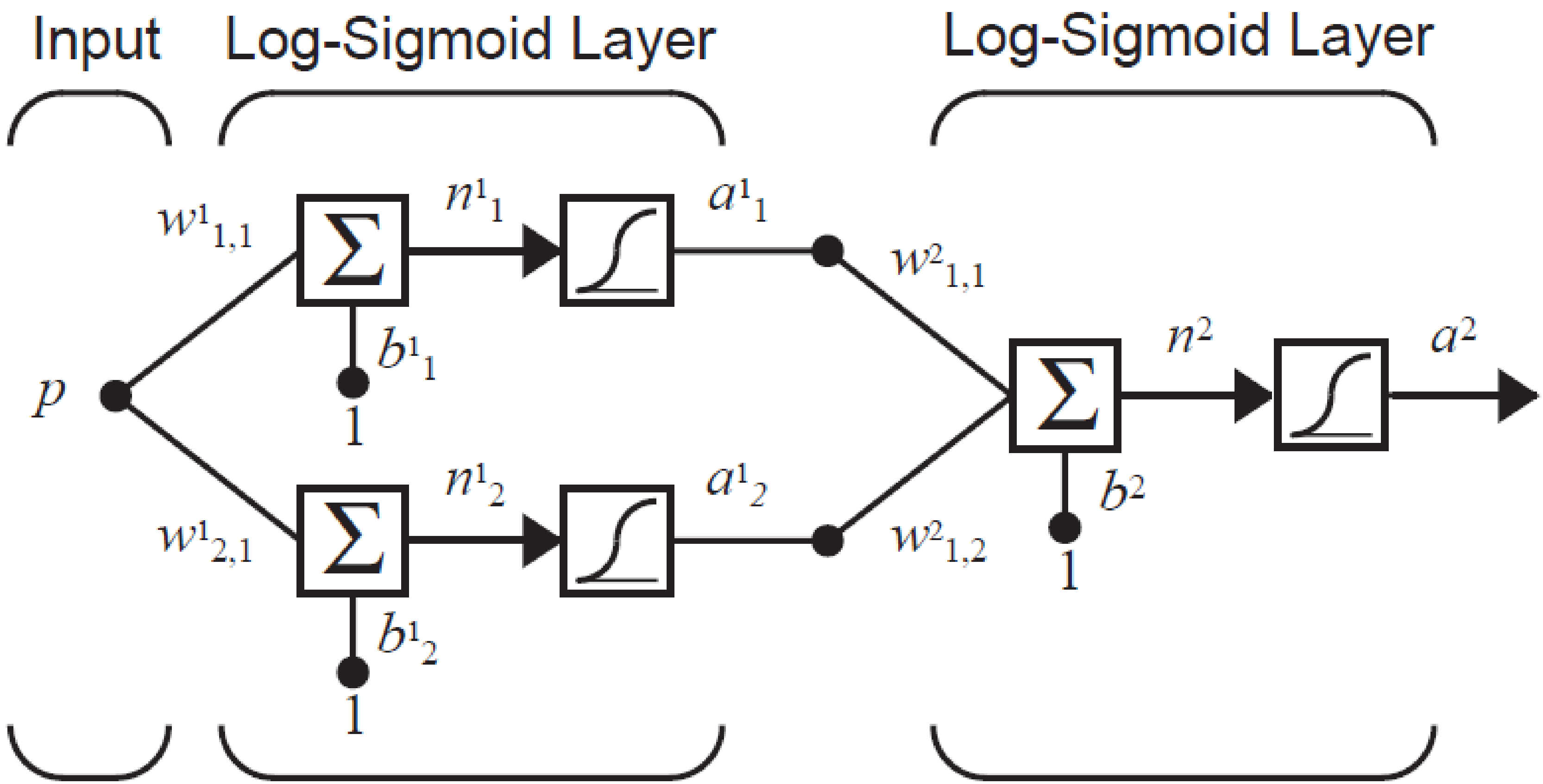
4
3
2
1
-1
-2
-3
-4
-4 -3 -2 -1 1 2 3 4



4
3
2
1
-1
-2
-3
-4
-4 -3 -2 -1 1 2 3 4

XOR gate with MLP

- Using multilayer perceptron we can the XOR gate classification with sigmoid activation function.



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

Python implementation of XOR gate

- import numpy as np
- import numpy as np

- def sigmoid (x):
-     return 1/(1 + np.exp(-x))

- def sigmoid_derivative(x):
-     return x * (1 - x)

- #Input datasets
- inputs = np.array([[0,0],[0,1],[1,0],[1,1]])
- expected_output = np.array([[0],[1],[1],[0]])

Python implementation of XOR gate

- epochs = 100000
- lr = 0.1
- inputLayerNeurons, hiddenLayerNeurons, outputLayerNeurons = 2,2,1

- #**Random weights and bias initialization**
- **hidden_weights** = np.random.uniform(size=(inputLayerNeurons,hiddenLayerNeurons))
- **hidden_bias** =np.random.uniform(size=(1,hiddenLayerNeurons))
- **output_weights** = np.random.uniform(size=(hiddenLayerNeurons,outputLayerNeurons))
- **output_bias** = np.random.uniform(size=(1,outputLayerNeurons))

Python implementation of XOR gate

```
for _ in range(epochs):
        #Forward Propagation
        hidden_layer_activation = np.dot(inputs,hidden_weights)
        hidden_layer_activation += hidden_bias
        hidden_layer_output = sigmoid(hidden_layer_activation)

        output_layer_activation = np.dot(hidden_layer_output,output_weights)
        output_layer_activation += output_bias
        predicted_output = sigmoid(output_layer_activation)

        #Backpropagation
        error = expected_output - predicted_output
        d_predicted_output = error * sigmoid_derivative(predicted_output)

        error_hidden_layer = d_predicted_output.dot(output_weights.T)
        d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)

        #Updating Weights and Biases
        output_weights += hidden_layer_output.T.dot(d_predicted_output) * lr
        output_bias += np.sum(d_predicted_output,axis=0,keepdims=True) * lr
        hidden_weights += inputs.T.dot(d_hidden_layer) * lr
        hidden_bias += np.sum(d_hidden_layer,axis=0,keepdims=True) * lr
```

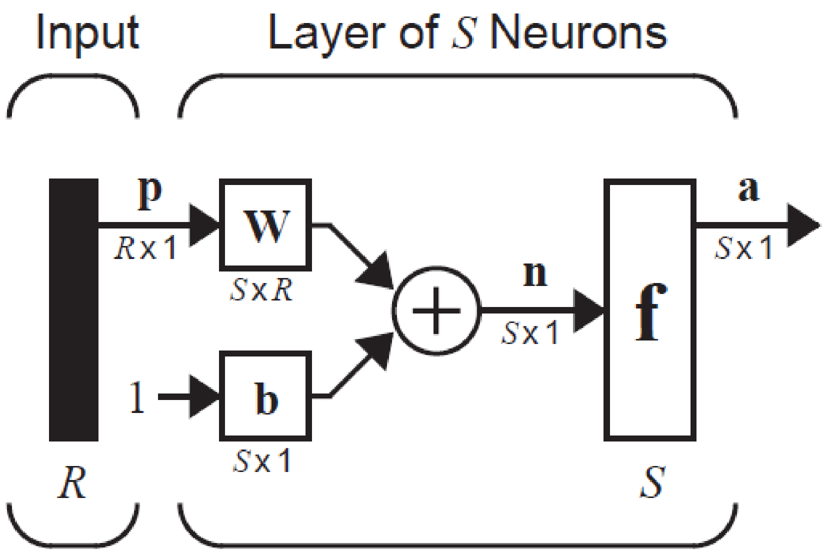
Python implementation of XOR gate

- print("Final hidden weights: ",end='')
- print(*hidden_weights)
- print("Final hidden bias: ",end='')
- print(*hidden_bias)
- print("Final output weights: ",end='')
- print(*output_weights)
- print("Final output bias: ",end='')
- print(*output_bias)

- print("\nOutput from neural network after 10,000 epochs: ",end='')
- print(*predicted_output)

Multilayer Perceptron Learning Rule

•



$$\mathbf{a} = \mathbf{f}(\mathbf{W}\mathbf{p} + \mathbf{b})$$

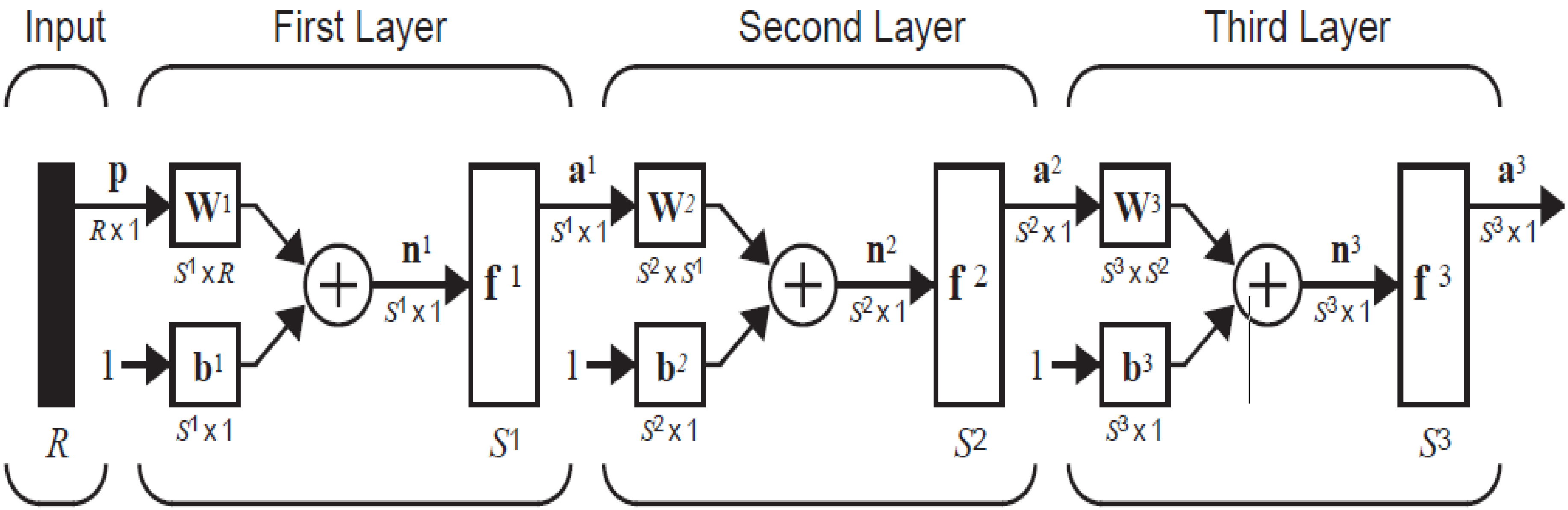
$$\mathbf{W} = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \\ \vdots \\ {}_S\mathbf{w}^T \end{bmatrix}.$$

$$W_i^{new} = W_i^{old} + e_i P$$

$$b_i^{new} = b_i^{old} + e_i$$

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}.$$

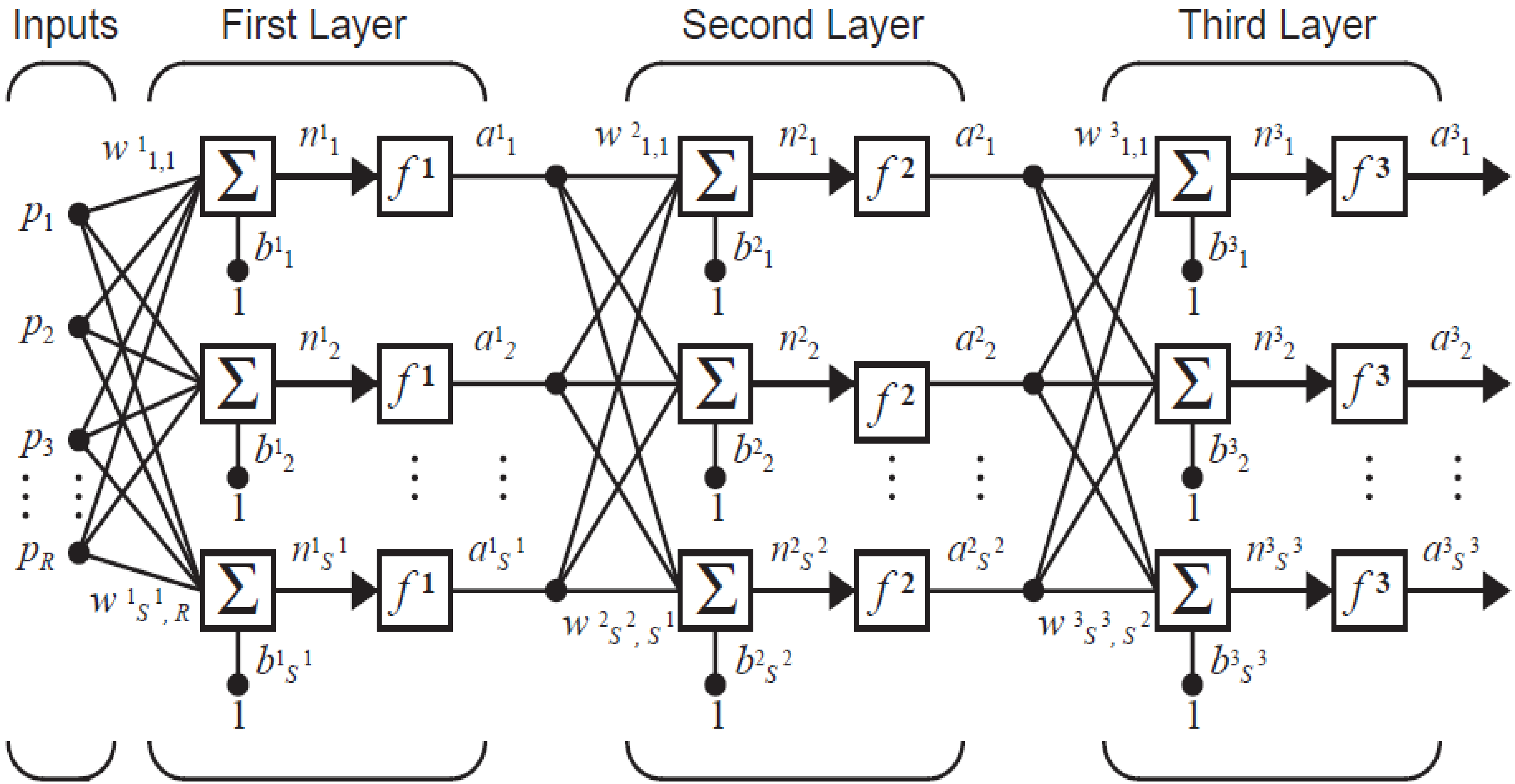
Multi-stage Networks



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

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

Multi-stage Networks

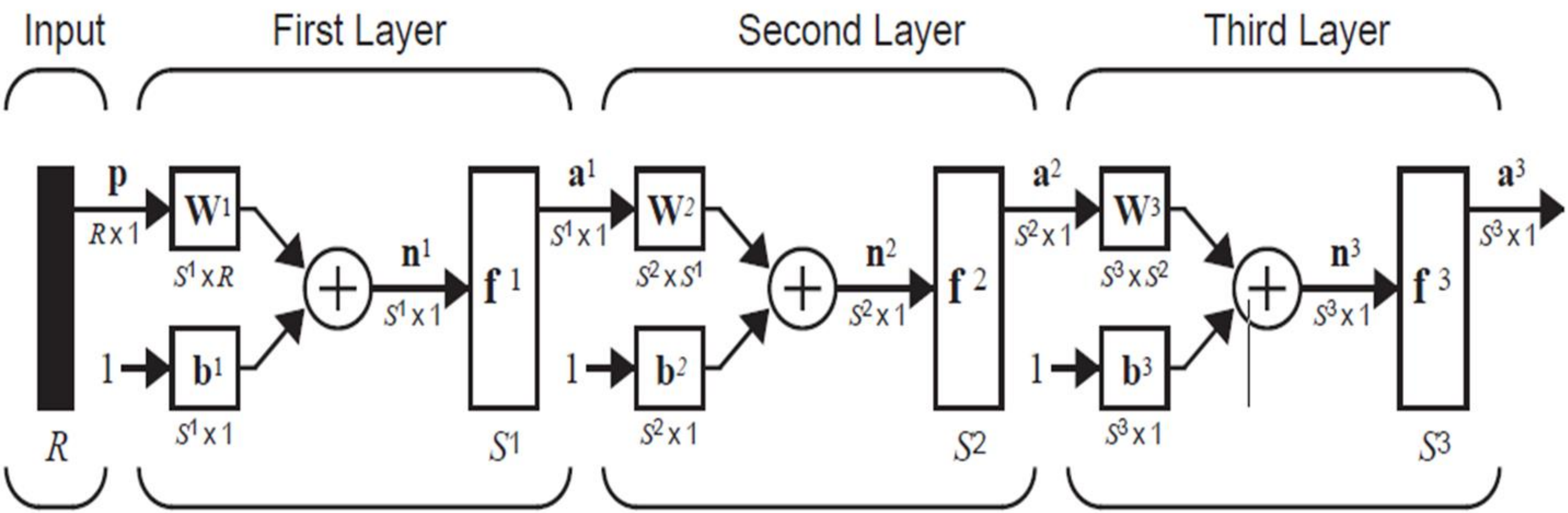


$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1) \qquad \mathbf{a}^2 = \mathbf{f}^2(\mathbf{W}^2\mathbf{a}^1+\mathbf{b}^2) \qquad \mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{a}^2+\mathbf{b}^3)$$

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

Backpropagation algorithm

- For multilayer networks the output of one layer becomes the input to the following layer.



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

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

Back-propagation algorithm

- First layer – $a^1 = f^1(W^1 p + b_1)$
- Second layer – $a^2 = f^2(W^2 a^1 + b_2)$

$$a^2 = f^2(W^2(f^1(W^1 p + b_1)) + b_2)$$

- Third layer – $a^3 = f^3(W^3 a^2 + b_3)$

$$a^3 = f^3(W^3((f^2(W^1(f^1(W^1 p + b_1)) + b_2)) + b_3)$$

- The equations that describe this operation are

$$a^{m+1} = f^{m+1}(W^3 a^m + b^{m+1})$$

where **m** is the number of layers in the network.

Back-propagation algorithm(error in different layers)

- Performance Index: The back-propagation algorithm uses **mean square error** as performance measure that is $E[e^2]$ where (**e = actual-predicted**)
- The algorithm is provided with a set of examples of proper network behavior:

$$\{p_{1,}t_1\},\{p_{2,}t_2\},...,\{p_{Q,}t_Q\}$$

- Where p_Q is the input to the network and t_Q is the corresponding target output.
- The algorithm should adjust the network parameters in order to minimize the mean square error:

$$F(x) = E[e^2] = E[(t-a)^2]$$

- where x is the vector of network weights and biases.

Back-propagation algorithm

- If the network has multiple outputs this generalizes to

$$F(x) = E[e^T e] = E[(t-a)^T (t-a)]$$

- The approximated MSE can be represented by

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

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

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

$$w_{i,j}^m(k+1) = w_{i,j}^m(k) - \alpha \frac{\partial \tilde{F}}{\partial w_{i,j}^m} \quad \text{and} \quad b_i^m(k+1) = b_i^m(k) - \alpha \frac{\partial \tilde{F}}{\partial b_i^m}$$

Back-propagation algorithm

- Using partial derivative and chain rule we have

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

$$\frac{\partial \hat{F}}{\partial b_i^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m}.$$

- Since the net input to layer m is an explicit function of the weights and bias in that layer:

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

Back-propagation algorithm

- If we now define $s_i^m = \dfrac{\partial \tilde{F}}{\partial n_i^m}$

as the sensitivity of $\tilde{F}$ to changes in the ith element of the net input at layer m. Then

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

Hence the weight updating can be written as

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

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

Back-propagation algorithm

- In matrix form this becomes:

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

where

$$s^m = \frac{\partial \tilde{F}}{\partial n^m} = \begin{bmatrix} \frac{\partial \tilde{F}}{\partial n_1^m} \\ \frac{\partial \tilde{F}}{\partial n_2^m} \\ . \\ . \\ . \\ \frac{\partial \tilde{F}}{\partial n_{s^m}^m} \end{bmatrix}$$

Backpropagating the Sensitivities

- The term back-propagation comes actually from the computation of sensitivities s^m.
- It describes a recurrence relationship in which the sensitivity at layer is **m** computed from the sensitivity at layer **m+1** .
- To derive the recurrence relationship the following Jacobian matrix is used.

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

Backpropagating the Sensitivities

- Consider the i, j element of the Jacobian matrix

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

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

- where

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

Backpropagating the Sensitivities

- Therefore the Jacobian matrix can be written

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

where

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

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

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

Backpropagating the Sensitivities

- The sensitivities are propagated backward through the network from the last layer to the first layer.

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

- We need the starting point s^{M} for the recurrence relation which is obtained at the final layer.

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

Backpropagating the Sensitivities

- Now, since $$\frac{\partial a_i}{\partial n_i^M} = \frac{\partial a_i^M}{\partial n_i^M} = \frac{\partial f^M(n_i^M)}{\partial n_i^M} = \dot{f}^M(n_i^M),$$

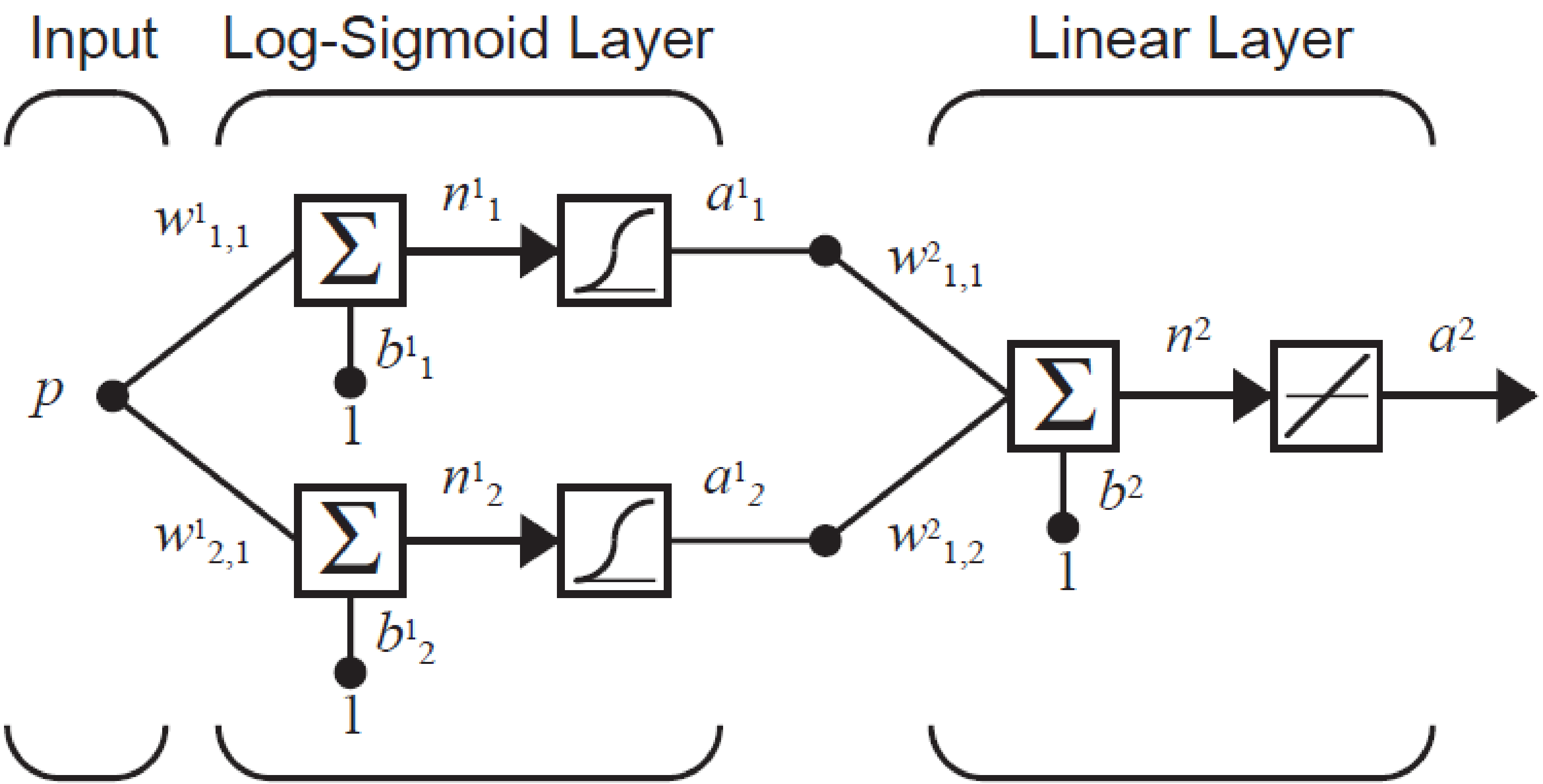
$$s_i^M = -2(t_i - a_i)\dot{f}^M(n_i^M).$$

- This can be expressed in matrix form as

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

Back-propagation example

- Suppose that we want to use the network to approximate the function for $g(p) = 1 + \sin\left(\frac{\pi}{4}p\right)$ for $-2 \le p \le 2$.



$\mathbf{a}^1 = \mathbf{logsig}(\mathbf{W}^1 p + \mathbf{b}^1)$ $\quad a^2 = purelin(\mathbf{W}^2 \mathbf{a}^1 + b^2)$

Back-propagation example

- We need to choose some initial values for the network weights and biases.

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

- For our initial input we will choose p=1. Hence

$a^0 = p = 1.$ The output of the first layer is then

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

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

Back-propagation example

- 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

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

- The next stage of the algorithm is to backpropagate the sensitivities.
- we will need the derivatives of the transfer functions, $\dot{f}^1(n)$ and $\dot{f}^2(n)$.

Back-propagation example

- 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

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

- The starting point of backpropagation is found at the second layer

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

Back-propagation example

- The first layer sensitivity is then computed by backpropagating the sensitivity from the second layer

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

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

$$= \begin{bmatrix}0.218 & 0 \\ 0 & 0.233\end{bmatrix}\begin{bmatrix}-0.227 \\ 0.429\end{bmatrix} = \begin{bmatrix}-0.0495 \\ 0.0997\end{bmatrix}.$$

Back-propagation example

- The final stage of the algorithm is to update the weights

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

$$= \begin{bmatrix} 0.171 & -0.0772 \end{bmatrix},$$

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

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

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

Back-propagation example

- This completes the first iteration of the back propagation algorithm.
- 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.

Batch vs. Incremental Training

- The algorithm described above is the stochastic gradient descent algorithm, which involves 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.

Batch vs. Incremental Training

- 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

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

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

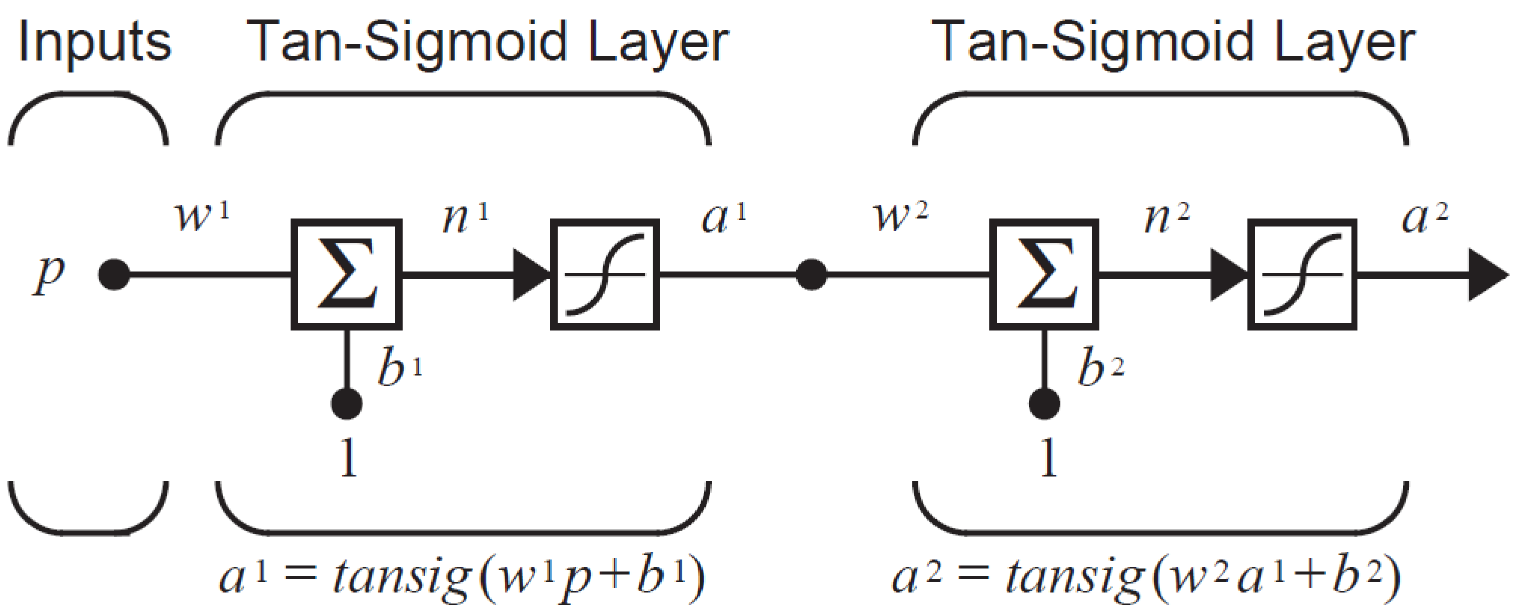
$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \frac{\alpha}{Q}\sum_{q=1}^{Q}\mathbf{s}_q^m$$

Back-propagation example

- For the network shown in Figure the initial weights and biases are chosen to be

$$w^1(0) = -1\,,\; b^1(0) = 1\,,\; w^2(0) = -2\,,\; b^2(0) = 1$$

- An input/target pair is given to be $((p = -1),(t = 1))$
- Perform one iteration of backpropagation with alpha=1



Back-propagation example

$$n^1 = w^1 p + b^1 = (-1)(-1) + 1 = 2$$

$$a^1 = tansig(n^1) = \frac{\exp(n^1) - \exp(-n^1)}{\exp(n^1) + \exp(-n^1)} = \frac{\exp(2) - \exp(-2)}{\exp(2) + \exp(-2)} = 0.964$$

$$n^2 = w^2 a^1 + b^2 = (-2)(0.964) + 1 = -0.928$$

$$a^2 = tansig(n^2) = \frac{\exp(n^2) - \exp(-n^2)}{\exp(n^2) + \exp(-n^2)} = \frac{\exp(-0.928) - \exp(0.928)}{\exp(-0.928) + \exp(0.928)}$$

$$= -0.7297$$

$$e = (t - a^2) = (1 - (-0.7297)) = 1.7297$$

Back-propagation example

- Now we back propagate the sensitivities

$$\mathbf{s}^2 = -2\dot{\mathbf{F}}^2(\mathbf{n}^2)(\mathbf{t}-\mathbf{a}) = -2[1-(a^2)^2](e) = -2[1-(-0.7297)^2]1.7297$$

$$= -1.6175$$

$$\mathbf{s}^1 = \dot{\mathbf{F}}^1(\mathbf{n}^1)(\mathbf{W}^2)^T\mathbf{s}^2 = [1-(a^1)^2]w^2\mathbf{s}^2 = [1-(0.964)^2](-2)(-1.6175)$$

$$= 0.2285$$

- Finally, the weights and biases are updated

$$w^2(1) = w^2(0) - \alpha s^2(a^1)^T = (-2) - 1(-1.6175)(0.964) = -0.4407$$

$$w^1(1) = w^1(0) - \alpha s^1(a^0)^T = (-1) - 1(0.2285)(-1) = -0.7715\,,$$

$$b^2(1) = b^2(0) - \alpha s^2 = 1 - 1(-1.6175) = 2.6175\,,$$

$$b^1(1) = b^1(0) - \alpha s^1 = 1 - 1(0.2285) = 0.7715\,.$$

Performance Surfaces and Optimum Points

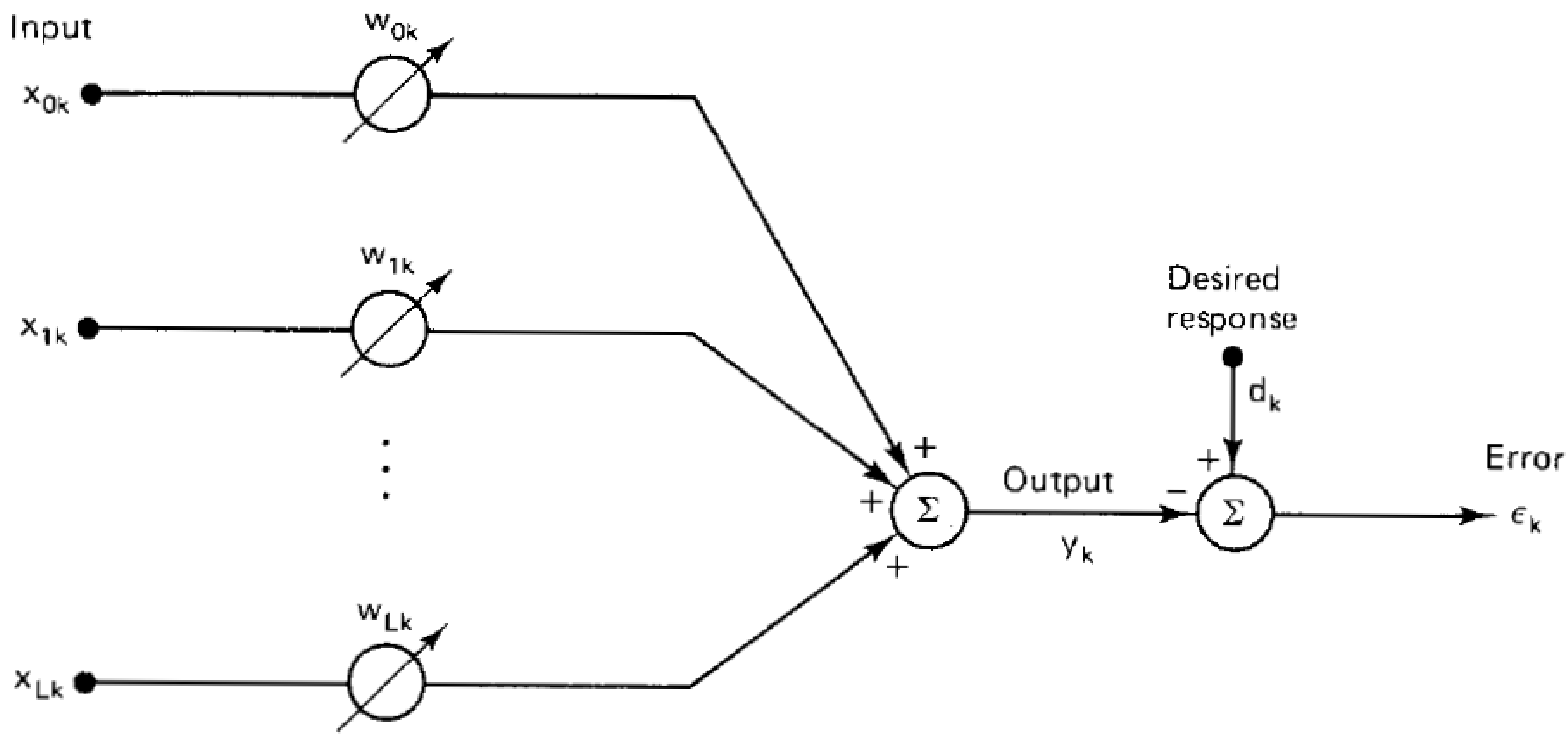
- Performance Learning: During training the network parameters (weights and biases) are adjusted in an effort to optimize the “performance” of the network.

- There are two steps involved in this optimization process. The first step is to define what we mean by performance, that is find a quantitative measure of network performance, called the **performance index**.

- The second step of the optimization process is to search the parameter space (adjust the network weights and biases) in order to reduce the performance index.

The performance function

- Suppose we multiple inputs with associate weight vectors and the output as shown in figure.



- We have to estimate the error function

The performance function

- The error signal with time index k is represented by

$$\varepsilon_k = d_k - y_k$$

Where the output is $y_k = \sum_{l=0}^{L} w_{lk} x_{lk}$, in matrix notation we have

$$y_k = \mathbf{X}_k^{\mathrm{T}} \mathbf{W}_k = \mathbf{W}_k^{\mathrm{T}} \mathbf{X}_k$$

Let d is the desired response or target value.

$$\varepsilon_k = d_k - \mathbf{X}_k^{\mathrm{T}} \mathbf{W} = d_k - \mathbf{W}^{\mathrm{T}} \mathbf{X}_k$$

$$\varepsilon_k^2 = d_k^2 + \mathbf{W}^{\mathrm{T}} \mathbf{X}_k \mathbf{X}_k^{\mathrm{T}} \mathbf{W} - 2 d_k \mathbf{X}_k^{\mathrm{T}} \mathbf{W}$$

The performance function

- We assume that ε_k, d_k, and $\mathbf{X}_k$ are statistically stationary and take the expected value

$$E\left[\varepsilon_k^2\right] = E\left[d_k^2\right] + \mathbf{W}^{\mathrm{T}} E\left[\mathbf{X}_k \mathbf{X}_k^{\mathrm{T}}\right]\mathbf{W} - 2E\left[d_k \mathbf{X}_k^{\mathrm{T}}\right]\mathbf{W}$$

- Let R defined as a square matrix where

$$\mathbf{R} = E\left[\mathbf{X}_k \mathbf{X}_k^{\mathrm{T}}\right] = E\begin{bmatrix} x_{0k}^2 & x_{0k}x_{1k} & x_{0k}x_{2k} & \cdots & x_{0k}x_{Lk} \\ x_{1k}x_{0k} & x_{1k}^2 & x_{1k}x_{2k} & \cdots & x_{1k}x_{Lk} \\ \vdots & \vdots & \vdots & & \vdots \\ x_{Lk}x_{0k} & x_{Lk}x_{1k} & x_{Lk}x_{2k} & \cdots & x_{Lk}^2 \end{bmatrix}$$

$$\mathbf{P} = E\left[d_k \mathbf{X}_k\right] = E\left[d_k x_{0k} \quad d_k x_{1k} \quad \cdots \quad d_k x_{Lk}\right]^{\mathrm{T}}$$

The performance function

- Now the MSE can be represented as

$$\text{MSE} \triangleq \xi = E\left[\varepsilon_k^2\right] = E\left[d_k^2\right] + \mathbf{W}^{\mathrm{T}}\mathbf{R}\mathbf{W} - 2\mathbf{P}^{\mathrm{T}}\mathbf{W}$$

- It is shown that MSE is precisely a quadratic function of the components of the weight vector W when the input components and desired response input are stationary stochastic variables.
- The gradient of MSE is given by

$$\nabla \triangleq \frac{\partial \xi}{\partial \mathbf{W}} = \left[\frac{\partial \xi}{\partial w_0} \quad \frac{\partial \xi}{\partial w_1} \quad \cdots \quad \frac{\partial \xi}{\partial w_L}\right]^{\mathrm{T}}$$

$$= 2\mathbf{R}\mathbf{W} - 2\mathbf{P}$$

The performance function

- To obtain the minimum MSE the weight vector W is set at its optimum value W*, where the gradient is zero.

$$\nabla = \mathbf{0} = 2\mathbf{R}\mathbf{W}^* - 2\mathbf{P}$$

$$\mathbf{W}^* = \mathbf{R}^{-1}\mathbf{P}$$

$$\begin{aligned}\xi_{\min} &= E\left[d_k^2\right] + \mathbf{W}^{*\mathrm{T}}\mathbf{R}\mathbf{W}^* - 2\mathbf{P}^{\mathrm{T}}\mathbf{W}^* \\ &= E\left[d_k^2\right] + [\mathbf{R}^{-1}\mathbf{P}]^{\mathrm{T}}\mathbf{R}\mathbf{R}^{-1}\mathbf{P} - 2\mathbf{P}^{\mathrm{T}}\mathbf{R}^{-1}\mathbf{P}\end{aligned}$$

$$\xi_{\min} = E\left[d_k^2\right] - \mathbf{P}^{\mathrm{T}}\mathbf{R}^{-1}\mathbf{P} = E\left[d_k^2\right] - \mathbf{P}^{\mathrm{T}}\mathbf{W}^*$$

- Also we can express MSE in the form

$$\xi = \xi_{\min} + (\mathbf{W} - \mathbf{W}^*)^{\mathrm{T}}\mathbf{R}(\mathbf{W} - \mathbf{W}^*)$$

$$\mathbf{V} = \mathbf{W} - \mathbf{W}^* = [v_0 \quad v_1 \quad \cdots \quad v_L]^{\mathrm{T}}$$

$$\xi = \xi_{\min} + \mathbf{V}^{\mathrm{T}}\mathbf{R}\mathbf{V}$$

The performance function

$$\text{MSE} \triangleq \xi = E\left[\varepsilon_k^2\right] = E\left[d_k^2\right] + \mathbf{W}^{\mathrm{T}}\mathbf{R}\mathbf{W} - 2\mathbf{P}^{\mathrm{T}}\mathbf{W}$$

$$\xi = \xi_{\min} + (\mathbf{W} - \mathbf{W}^*)^{\mathrm{T}}\mathbf{R}(\mathbf{W} - \mathbf{W}^*)$$

$$\xi = \xi_{\min} + \mathbf{W}^{*\mathrm{T}}\mathbf{R}\mathbf{W}^* + \mathbf{W}^{\mathrm{T}}\mathbf{R}\mathbf{W} - \mathbf{W}^{\mathrm{T}}\mathbf{R}\mathbf{W}^* - \mathbf{W}^{*\mathrm{T}}\mathbf{R}\mathbf{W}$$

$$\xi = E\left[d_k^2\right] - \mathbf{P}^{\mathrm{T}}\mathbf{W}^* + \mathbf{W}^{*\mathrm{T}}\mathbf{R}\mathbf{W}^* + \mathbf{W}^{\mathrm{T}}\mathbf{R}\mathbf{W} - 2\mathbf{W}^{\mathrm{T}}\mathbf{R}\mathbf{W}^*$$

$$\xi = E\left[d_k^2\right] - \mathbf{P}^{\mathrm{T}}\mathbf{R}^{-1}\mathbf{P} + \mathbf{P}^{\mathrm{T}}\mathbf{R}^{-1}\mathbf{R}\mathbf{R}^{-1}\mathbf{P} + \mathbf{W}^{\mathrm{T}}\mathbf{R}\mathbf{W} - 2\mathbf{W}^{\mathrm{T}}\mathbf{R}\mathbf{R}^{-1}\mathbf{P}$$

$$= E\left[d_k^2\right] + \mathbf{W}^{\mathrm{T}}\mathbf{R}\mathbf{W} - 2\mathbf{W}^{\mathrm{T}}\mathbf{P}$$

$$= E\left[d_k^2\right] + \mathbf{W}^{\mathrm{T}}\mathbf{R}\mathbf{W} - 2\mathbf{P}^{\mathrm{T}}\mathbf{W}$$

The performance function

- The gradient of mean square error wrt V is obtained by

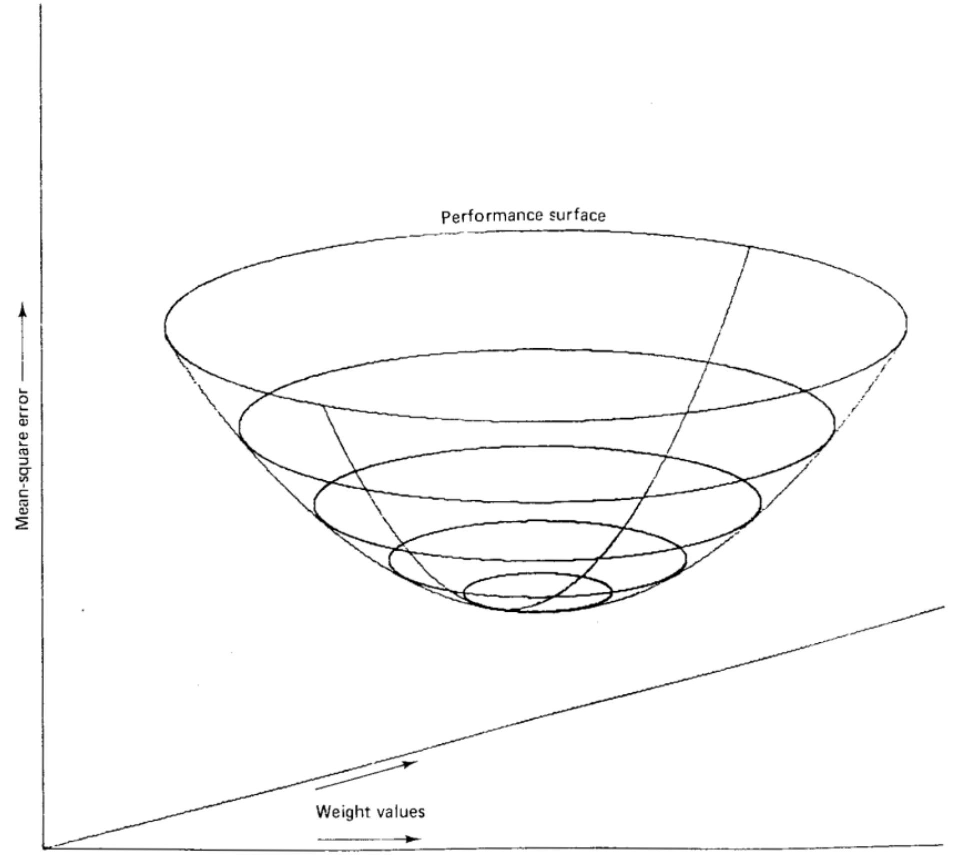
$$\frac{\partial \xi}{\partial \mathbf{V}} = \begin{bmatrix} \frac{\partial \xi}{\partial v_0} & \frac{\partial \xi}{\partial v_1} & \cdots & \frac{\partial \xi}{\partial v_L} \end{bmatrix} = 2\mathbf{RV}$$

$$\nabla = \frac{\partial \xi}{\partial \mathbf{W}} = \frac{\partial \xi}{\partial \mathbf{V}} = 2\mathbf{RV} = 2(\mathbf{RW} - \mathbf{P})$$

Observations

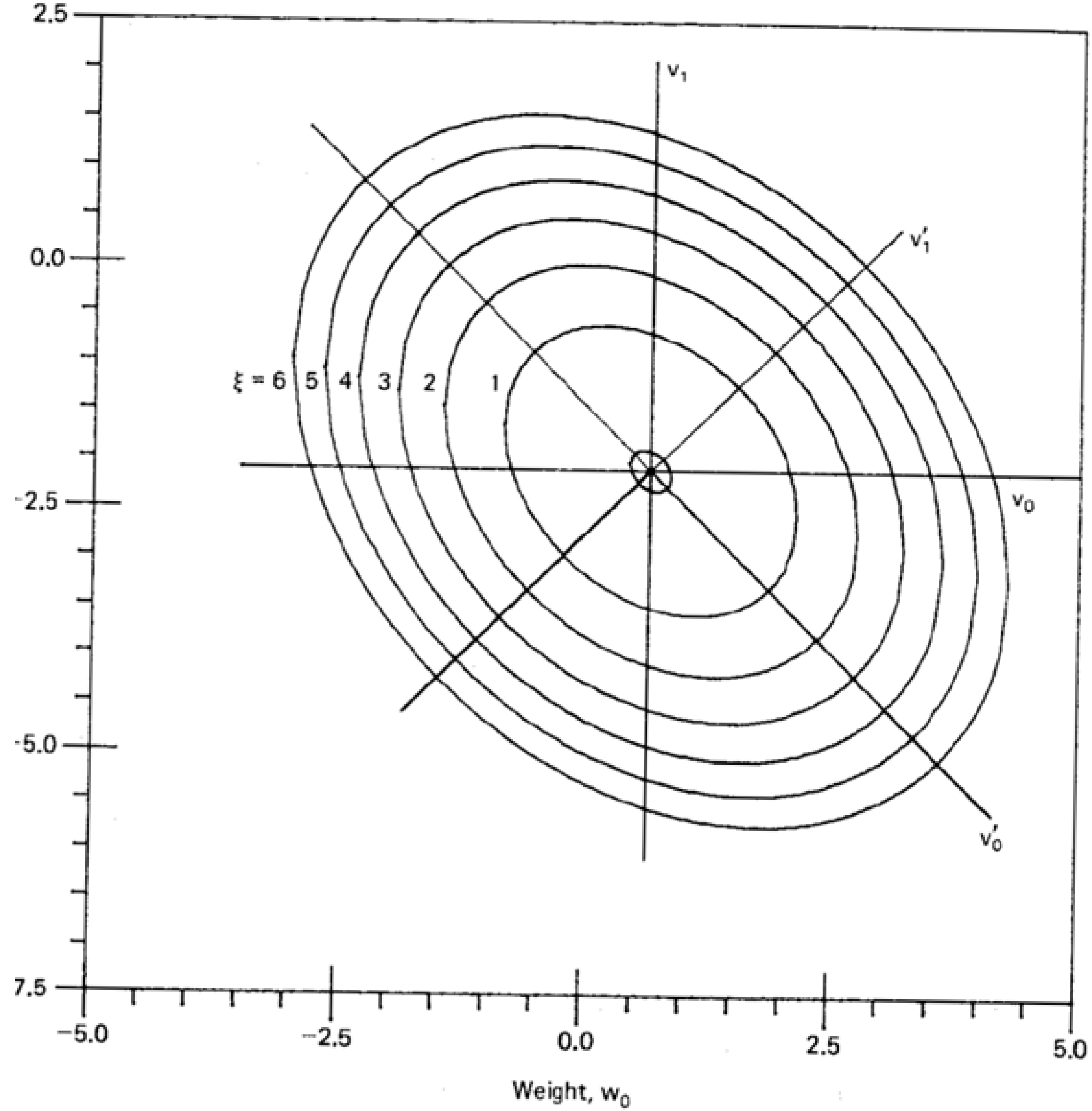
1) The eigenvectors of the input correlation matrix (R) define the principal axes of the error surface.

2) The eigenvalues of the input correlation matrix (R), give the second derivatives of the error surface with respect to the principal axes of error surface.



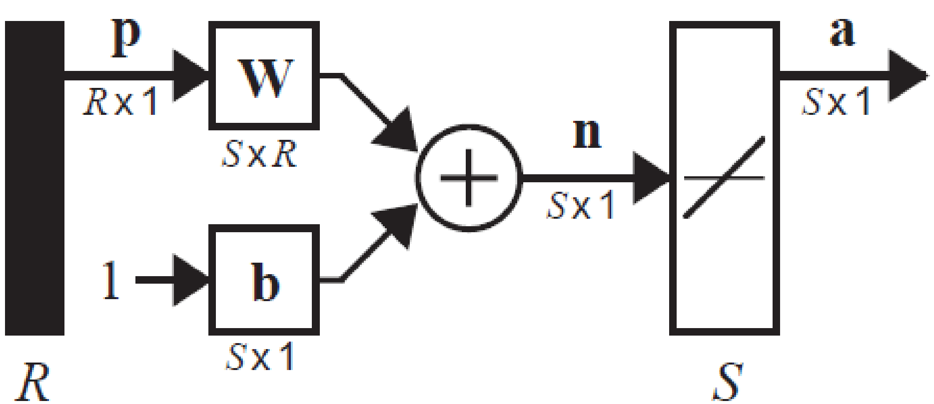
Observations

- .



ADALINE Network

- The ADALINE network is shown in Figure which has a linear transfer function.



ADALINE Network

Performance Optimization

- The objective is to develop algorithms to optimize a performance index F(x) which means to find the value of x that minimizes F(x).
- Consider the following function of n variables:

$$F(x) = F(x_1, x_1, ..., x_n)$$

- The gradient can be defined as

$$\nabla F(\mathbf{x}) = \left[\frac{\partial}{\partial x_1}F(\mathbf{x}) \; \frac{\partial}{\partial x_2}F(\mathbf{x}) \; \ldots \; \frac{\partial}{\partial x_n}F(\mathbf{x})\right]^T$$

Performance Optimization

- The $\nabla^2 F(x)$ is called Hessian and is represented by

$$
\nabla^2 F(\mathbf{x}) = \begin{bmatrix}
\frac{\partial^2}{\partial x_1^2} F(\mathbf{x}) & \frac{\partial^2}{\partial x_1 \partial x_2} F(\mathbf{x}) & \dots & \frac{\partial^2}{\partial x_1 \partial x_n} F(\mathbf{x}) \\
\frac{\partial^2}{\partial x_2 \partial x_1} F(\mathbf{x}) & \frac{\partial^2}{\partial x_2^2} F(\mathbf{x}) & \dots & \frac{\partial^2}{\partial x_2 \partial x_n} F(\mathbf{x}) \\
\vdots & \vdots & & \vdots \\
\frac{\partial^2}{\partial x_n \partial x_1} F(\mathbf{x}) & \frac{\partial^2}{\partial x_n \partial x_2} F(\mathbf{x}) & \dots & \frac{\partial^2}{\partial x_n^2} F(\mathbf{x})
\end{bmatrix}
$$

Performance Optimization

- Consider the following function of two variables

$$F(x) = x_1^4 + x_2^2$$

$$\nabla F(x) = \begin{bmatrix} 4x_1^3 \\ 2x_2 \end{bmatrix} \qquad \nabla^2 F(x) = \begin{bmatrix} 12\mathrm{x}_1^2 & 0 \\ 0 & 2 \end{bmatrix}_{x=0} = \begin{bmatrix} 0 & 0 \\ 0 & 2 \end{bmatrix}$$

- Consider the following function of two variables

$$F(x) = x_1^2 + x_1 x_2 + x_2^2$$

$$\nabla F(x) = \begin{bmatrix} 2x_1 + x_2 \\ 2x_2 + x_1 \end{bmatrix} \qquad \nabla^2 F(x) = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

Quadratic Functions

- The general form of a quadratic function is

$$F(x) = \frac{1}{2} X^T A X + d^T X + c$$

- To find the gradient for this function, we will use the following useful properties of the gradient

1) $\nabla(h^T X) = \nabla(h X^T) = h$

where h is a constant vector.

2) $\nabla(X^T Q X) = QX + Q^T X = 2QX$

- The gradient of $F(x) = \frac{1}{2} X^T A X + d^T X + c$ is
- $\nabla F(x) = AX + d$ and $\nabla^2 F(x) = A$

Eigensystem of the Hessian

- Consider a quadratic function that has a stationary point at the origin,

$$F(x) = \frac{1}{2} x^T A x$$

- where A represents the the Hessian of the quadratic equation. A is also a symmetric matrix.

$$F(x) = x_1^2 + x_2^2 \;\; = \frac{1}{2} X^T A X$$

$$= \frac{1}{2} X^T \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} X$$

Perceptron Learning Rule

- Now $F(x) = \frac{1}{2} x^T A x$ $F(x) = x_1^2 + x_2^2$
- Let $x_1 = 3$ and $x_2 = 4$, then if $x = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$

$$F(x) = \frac{1}{2} \begin{bmatrix} 3 & 4 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \end{bmatrix} = 25$$

Performance Optimization

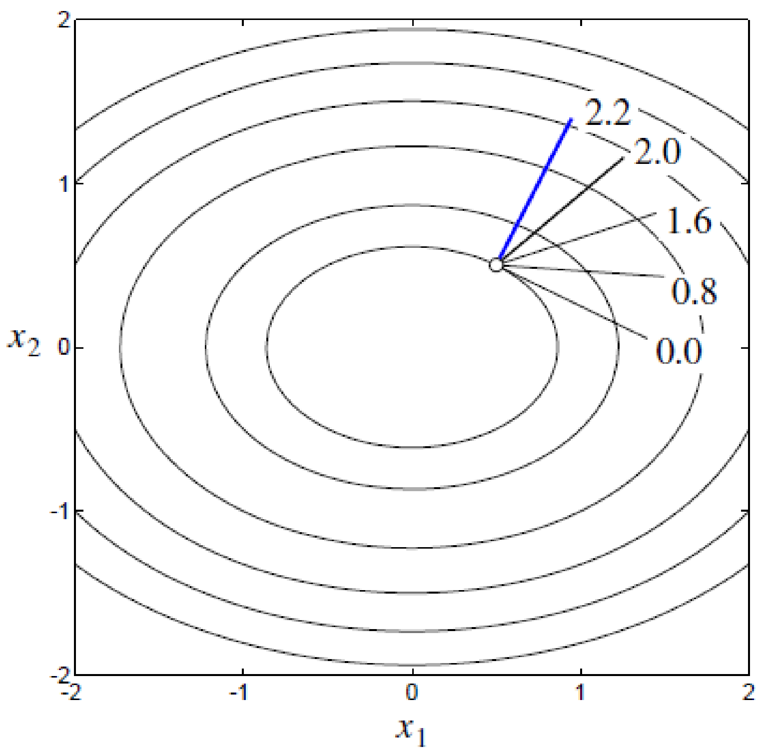
- Let **P** be a vector in the direction along which we wish to know the derivative of F(x). The first and second derivative in the direction of **P** can then be computed as

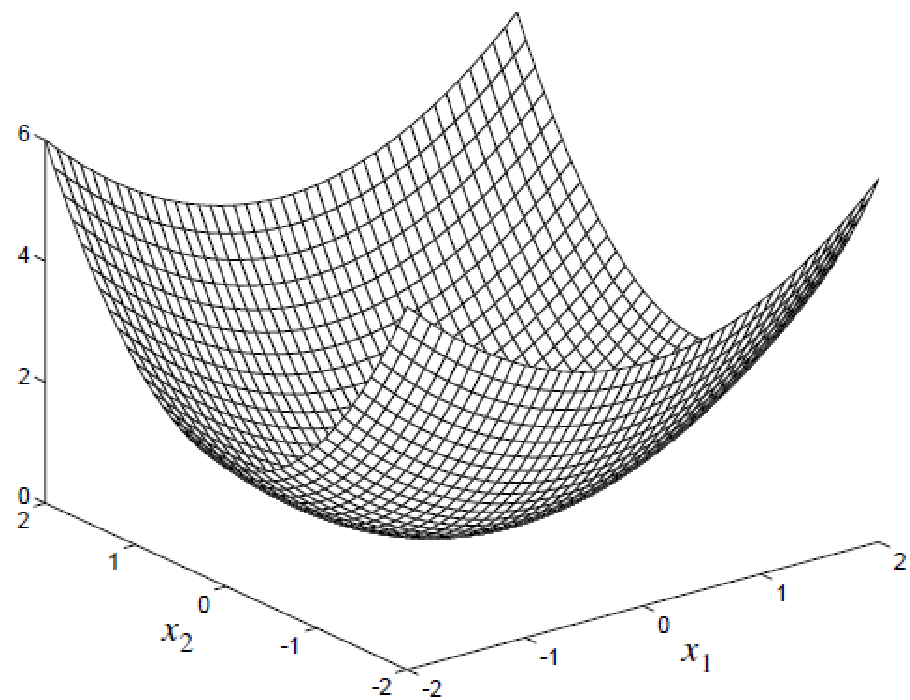
$$\frac{P^T \nabla F(x)}{\|P\|} \quad \text{and} \quad \frac{\mathbf{p}^T \nabla^2 F(\mathbf{x}) \mathbf{p}}{\|\mathbf{p}\|^2}.$$

- Where P is the direction vector.
- Which direction has the greatest slope?
- Any direction that is orthogonal to the gradient will have zero slope.
- The maximum slope will occur when the inner product of the direction vector and the gradient is a maximum.

Performance Optimization

- Figure shows a contour plot and a 3-D plot of F(x) having the expression $F(x) = x_1^2 + 2x_2^2$





Performance Optimization

- Suppose that we want to know the derivative of the function at the point $\mathbf{x}^* = \begin{bmatrix} 0.5 & 0.5 \end{bmatrix}^T$ in the direction $\mathbf{p} = \begin{bmatrix} 2 & -1 \end{bmatrix}^T$
- First we evaluate the gradient at $\mathbf{x}^*$

$$\nabla F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*} = \left.\begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix}\right|_{\mathbf{x}=\mathbf{x}^*} = \left.\begin{bmatrix} 2x_1 \\ 4x_2 \end{bmatrix}\right|_{\mathbf{x}=\mathbf{x}^*} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

- The derivative in the direction P can then be computed as

Performance Optimization

$$\frac{\mathbf{p}^T \nabla F(\mathbf{x})}{\|\mathbf{p}\|} = \frac{\begin{bmatrix} 2 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix}}{\left\| \begin{bmatrix} 2 \\ -1 \end{bmatrix} \right\|} = \frac{\begin{bmatrix} 0 \end{bmatrix}}{\sqrt{5}} = 0 .$$

- Therefore the function has zero slope in the direction P.
- What can we say about those directions that have zero slope? **Therefore any direction that is orthogonal to the gradient will have zero slope**.
- The maximum slope will occur when the inner product of the direction vector and the gradient is a maximum. **This happens when the direction vector is the same as the gradient**.

Steepest Descent

- To optimize a performance index F(x).
- To find the value of x that minimizes F(x).
- We begin from some initial guess x_0 and then update the guess in stages according to an equation of the form

$$x_{k+1} = x_k + \alpha_k p_k$$

- where the vector pk represents a search direction, and the positive scalar α_k is the learning rate, which determines the length of the step.
- Now let $\Delta x_k = x_{k+1} - x_k = \alpha_k p_k$

Steepest Descent

- We would like to have the function decreases at each iteration $F(x_{k+1}) < F(x_k)$. Now if
- Consider the first-order Taylor series expansion of $F(x_k)$ about the old guess x_k is

$$F(x_{k+1}) = F(x_k) + \nabla F(x_k)^T (x_{k+1} - x_k)$$

- Taylor series expansion:

$$F(\mathbf{x}) = F(\mathbf{x}^*) + \nabla F(\mathbf{x})^T \Big|_{\mathbf{x} = \mathbf{x}^*} (\mathbf{x} - \mathbf{x}^*)$$

$$+ \frac{1}{2}(\mathbf{x} - \mathbf{x}^*)^T \nabla^2 F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}^*} (\mathbf{x} - \mathbf{x}^*) + \cdots$$

Steepest Descent

- We can write $F(x_{k+1}) = F(x_k) + g_k^T \Delta x_k$ where

$$g_k = \nabla F(x_k)$$

- For $F(x_{k+1}) < F(x_k)$ the value of $g_k^T \Delta x_k = g_k^T \alpha_k p_k < 0$
- We will select an α_k that is small, but greater than zero and the condition is

$$g_k^T p_k < 0$$

- Any vector p_k that satisfies this equation is called a descent directional vector.
- The function must go down if we take a small enough step in this direction.

Steepest Descent

- What is the direction of steepest descent?
- In what direction will the function decrease most rapidly?
- It will be most negative when the direction vector is the negative of the gradient.
- Therefore a vector that points in the steepest descent direction is $p_k = -g_k^T$
- Using this in the iterative equation, that produces the method of steepest descent is

$$x_{k+1} = x_k - \alpha_k g_k$$

Learning Rate

- For steepest descent there are two general methods for determining the learning rate α_k .
- One approach is to minimize the performance index F(x) with respect to α_k at each iteration.
- The other method for selecting is to use a fixed value (e.g.,$\alpha_k = 0.02$), or to use variable, but predetermined, values like

$$\alpha_k = \frac{1}{k}$$

Learning Rate

- Apply the steepest descent algorithm to the following function $F(x) = x_1^2 + 25x_2^2$, starting from the initial guess

$$x_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

- The first step is to find the gradient:

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 2x_1 \\ 50x_2 \end{bmatrix}.$$

Learning Rate

- If we evaluate the gradient at the initial guess we find

$$\mathbf{g}_0 = \nabla F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_0} = \begin{bmatrix} 1 \\ 25 \end{bmatrix}.$$

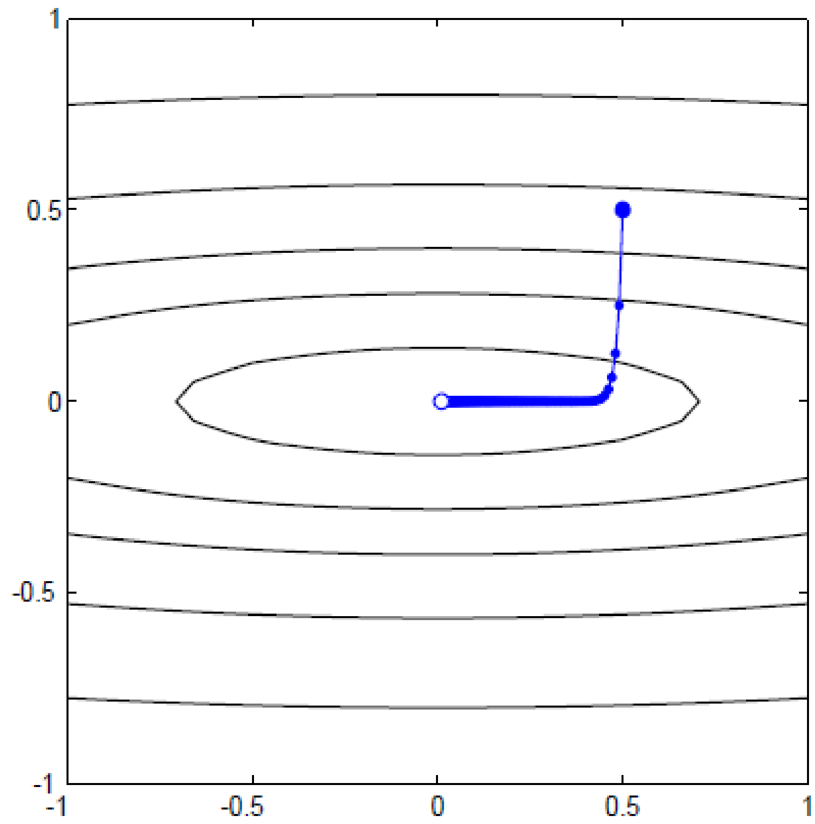
- Assuming a fixed learning rate of $\alpha = 0.01$

$$\mathbf{x}_1 = \mathbf{x}_0 - \alpha \mathbf{g}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - 0.01 \begin{bmatrix} 1 \\ 25 \end{bmatrix} = \begin{bmatrix} 0.49 \\ 0.25 \end{bmatrix}.$$

$$\mathbf{x}_2 = \mathbf{x}_1 - \alpha \mathbf{g}_1 = \begin{bmatrix} 0.49 \\ 0.25 \end{bmatrix} - 0.01 \begin{bmatrix} 0.98 \\ 12.5 \end{bmatrix} = \begin{bmatrix} 0.4802 \\ 0.125 \end{bmatrix}.$$

Learning Rate

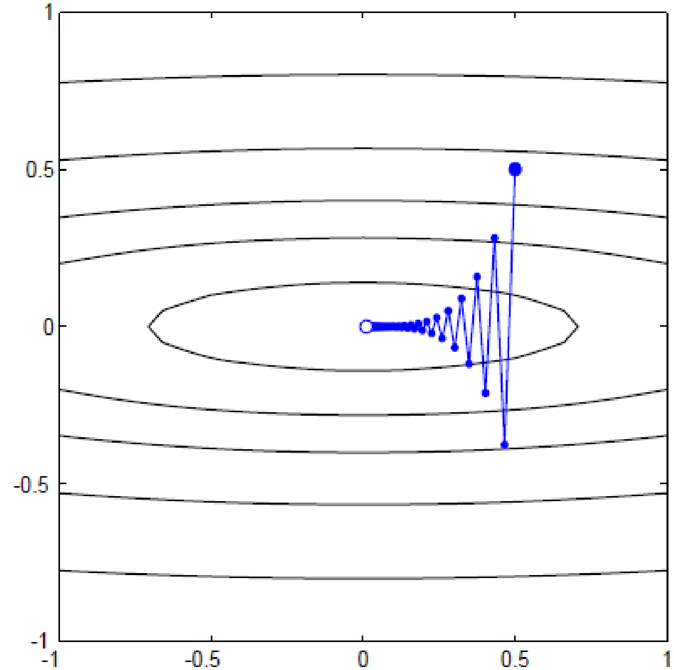
- If we continue the iterations we obtain the trajectory



Trajectory for Steepest Descent with $\alpha = 0.01$

Learning Rate

- Note that the trajectory now oscillates. If we make the learning rate too large the algorithm will become unstable; the oscillations will increase instead of decaying.



Trajectory for Steepest Descent with $\alpha = 0.035$

Stable Learning Rate

- Suppose that the performance index is a quadratic function

$$F(x) = \frac{1}{2} X^T A X + d^T X + c$$

- The gradient of the quadratic function is

$$\nabla F(x) = AX + d$$

- If we now insert this expression into our expression for the steepest descent algorithm assuming a constant learning rate

$$x_{k+1} = x_k - \alpha g_k = x_k - \alpha(Ax_k + d)$$

$$x_{k+1} = (I - \alpha A)x_k - \alpha d$$

Stable Learning Rate

- This is a linear dynamic system, which will be stable if the eigenvalues of the matrix $(I - \alpha A)$ are less than one in magnitude.
- Our condition for the stability of the steepest descent algorithm is then

$$\left|(1 - \alpha\lambda_i)\right| < 1 \quad \text{or} \quad \alpha < \frac{2}{\lambda_i} \quad \text{or} \quad \alpha < \frac{2}{\lambda_{\max}}$$

- The maximum stable learning rate is inversely proportional to the maximum curvature of the quadratic function.

Stable Learning Rate

- Example: $F(x) = x_1^2 + 25x_2^2$ starting from the initial guess

$$x_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

- The Hessian matrix for this quadratic function is

$$A = \begin{bmatrix} 2 & 0 \\ 0 & 50 \end{bmatrix}$$

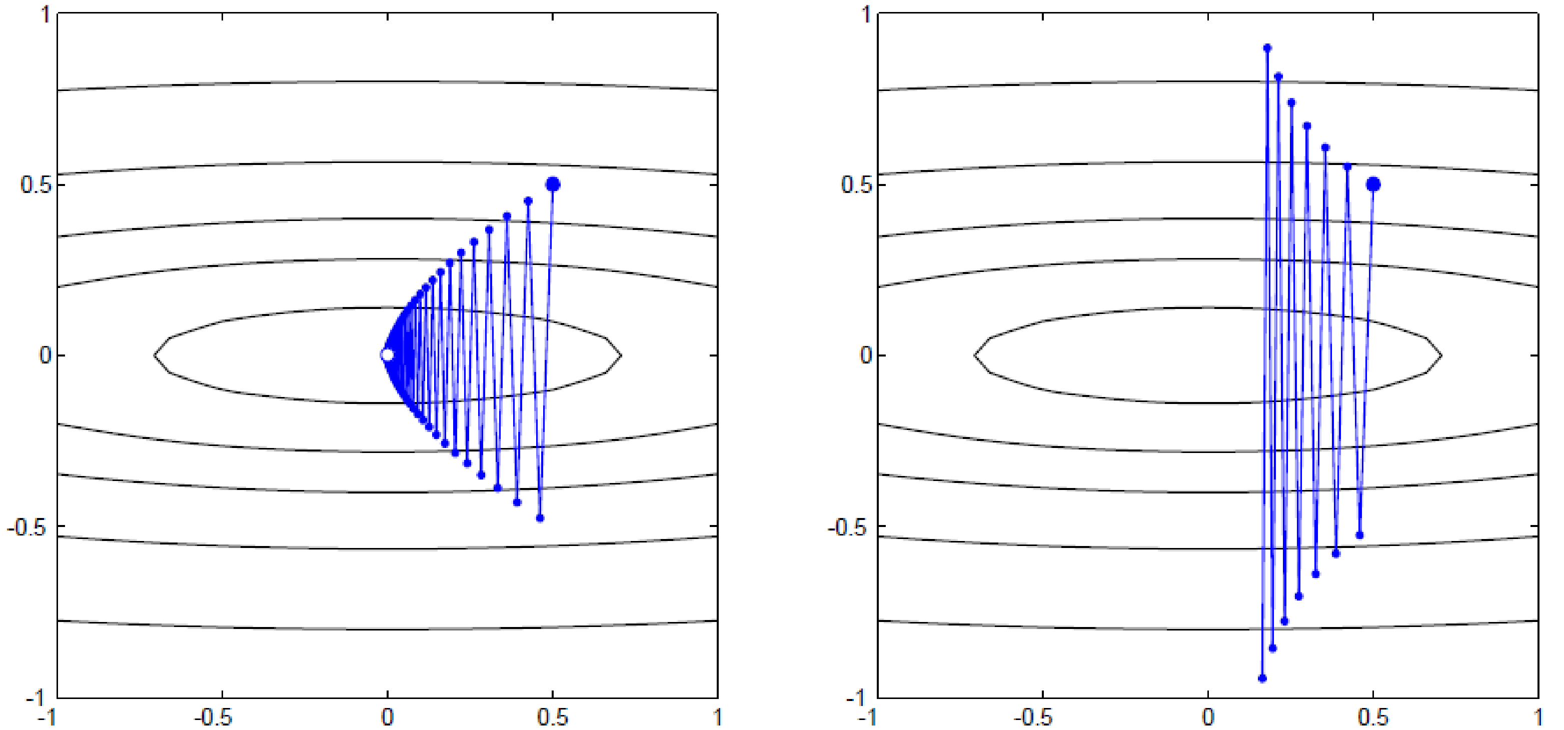
- The eigenvalues and eigenvectors of A are

$$\left\{(\lambda_1 = 2), \left(\mathbf{z}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right)\right\}, \left\{(\lambda_2 = 50), \left(\mathbf{z}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}\right)\right\}$$

Perceptron Learning Rule

- Therefore the maximum allowable learning rate is

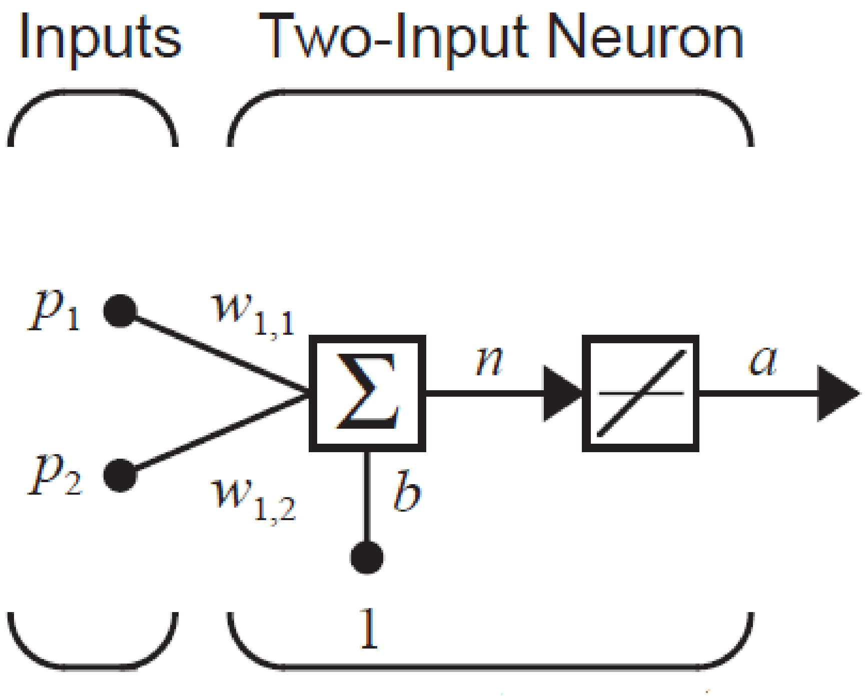
$$\alpha < \frac{2}{\lambda_{\max}} = \frac{2}{50} = 0.04$$



Trajectories for $\alpha = 0.039$ (left) and $\alpha = 0.041$ (right).

Mean Square Error

- Consider a single ADALINE with two inputs.

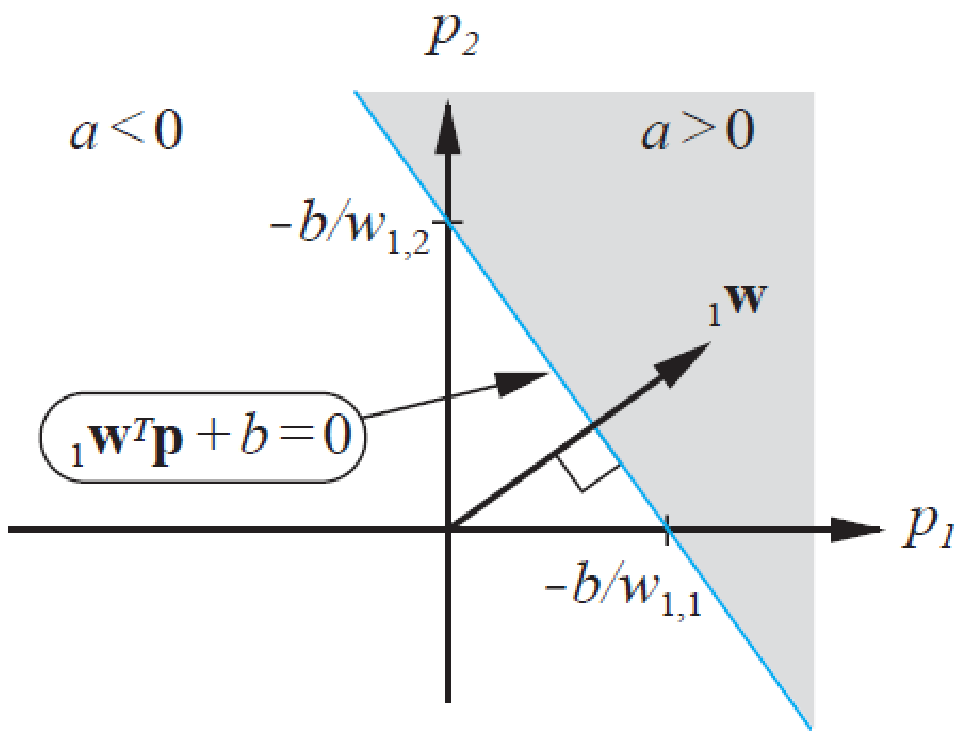


Two-Input Linear Neuron

$$a = {}_1\mathbf{w}^T\mathbf{p} + b = w_{1,1}p_1 + w_{1,2}p_2 + b$$

Mean Square Error

- If we set n=0, then ${}_1W^T X + b = 0$ specifies such a line which can classify objects into two categories., as shown in Figure.



Decision Boundary for Two-Input ADALINE

Mean Square Error

- Let network behavior is given by

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

- Where $\mathbf{p}_Q$ is an input to the network and $\mathbf{t}_Q$ is the corresponding target.
- $$\mathbf{x} = \begin{bmatrix} {}_1\mathbf{w} \\ b \end{bmatrix} \quad \mathbf{z} = \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} \quad a = {}_1\mathbf{w}^T\mathbf{p} + b, \quad a = \mathbf{x}^T\mathbf{z}.$$

- Expression for the ADALINE network mean square error:

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

Mean Square Error

- .

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

$$F(\mathbf{x}) = E[t^2 - 2t\mathbf{x}^T\mathbf{z} + \mathbf{x}^T\mathbf{z}\mathbf{z}^T\mathbf{x}]$$

$$= E[t^2] - 2\mathbf{x}^T E[t\mathbf{z}] + \mathbf{x}^T E[\mathbf{z}\mathbf{z}^T]\mathbf{x}$$

$$F(\mathbf{x}) = c - 2\mathbf{x}^T\mathbf{h} + \mathbf{x}^T\mathbf{R}\mathbf{x},$$

$$c = E[t^2],\ \mathbf{h} = E[t\mathbf{z}] \text{ and } \mathbf{R} = E[\mathbf{z}\mathbf{z}^T].$$

- $$F(\mathbf{x}) = c + \mathbf{d}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x}.$$ where

$$\mathbf{d} = -2\mathbf{h} \text{ and } \mathbf{A} = 2\mathbf{R}$$

Mean Square Error

- Here the vector **h** gives the cross-correlation between the **input vector** and its **associated target**.
- **R** is the input **correlation matrix**..
- We can see that the mean square error performance index for the ADALINE network is a quadratic function, $F(\mathbf{x}) = c + \mathbf{d}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x}\,.$
- Where $\mathbf{d} = -2\mathbf{h}$ and $\mathbf{A} = 2\mathbf{R}$

$$F(\mathbf{x}) = c - 2\mathbf{x}^T\mathbf{h} + \mathbf{x}^T\mathbf{R}\mathbf{x}\,,$$

Mean Square Error

- we know that the characteristics of the quadratic function depend primarily on the Hessian matrix A.
- If the eigenvalues of the Hessian are all positive, then the function will have one unique global minimum.
- It can be shown that all correlation matrices are either positive definite or positive semi-definite, which means that they can never have negative eigenvalues.

Mean Square Error

- Let's locate the stationary point of the performance index.
- Gradient:

$$\nabla F(\mathbf{x}) = \nabla\left(c + \mathbf{d}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x}\right) = \mathbf{d} + \mathbf{A}\mathbf{x} = -2\mathbf{h} + 2\mathbf{R}\mathbf{x}$$

- The stationary point of can be found by setting the gradient equal to zero:

$$-2\mathbf{h} + 2\mathbf{R}\mathbf{x} = 0 \qquad\qquad \mathbf{x}^* = \mathbf{R}^{-1}\mathbf{h}$$

- Therefore, if the correlation matrix is positive definite there will be a unique stationary point, which will be a strong minimum.

LMS Algorithm

- Using algorithm we find the estimated gradient and find the minimum point.
- The key insight that we could estimate the MSE error F(x) by

$$\hat{F}(\mathbf{x}) = (t(k) - a(k))^2 = e^2(k),$$

- where the expectation of the squared error has been replaced by the squared error at iteration k.
- Then, at each iteration we have a gradient estimate

of the form: $$\hat{\nabla} F(\mathbf{x}) = \nabla e^2(k)$$

Mean Square Error

- This is sometimes referred to as the stochastic gradient. When this is used in a gradient descent algorithm, it is referred to as “on-line” or incremental learning, since the weights are updated as each input is presented to the network.
- The first **R th** elements of $\nabla^2 e(k)$ are derivatives with respect to the network weights and **(R+1)th** element is the derivative with respect to the bias.

$$[\nabla e^2(k)]_j = \frac{\partial e^2(k)}{\partial w_{1,j}} = 2e(k)\frac{\partial e(k)}{\partial w_{1,j}} \text{ for } j = 1, 2, \ldots, R$$

Mean Square Error

$$[\nabla e^2(k)]_{R+1} = \frac{\partial e^2(k)}{\partial b} = 2e(k)\frac{\partial e(k)}{\partial b}.$$

First evaluate the partial derivative of the error with respect to the weight

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial[t(k)-a(k)]}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}}[t(k)-({}_1\mathbf{w}^T\mathbf{p}(k)+b)]$$

$$= \frac{\partial}{\partial w_{1,j}}\left[t(k)-\left(\sum_{i=1}^{R} w_{1,i}p_i(k)+b\right)\right]$$

where $p_i(k)$ is the ith element of the input vector at the kth iteration.

LMS Algorithm

- This simplifies to $$\frac{\partial e(k)}{\partial w_{1,j}} = -p_j(k)$$
- In a similar way we can obtain $$\frac{\partial e(k)}{\partial b} = -1$$
- Noting point is that $p_j(k)$ and 1 are the elements of the input vector **z** .
- So the **gradient of the squared error** at iteration **k** can be written

$$\hat{\nabla} F(\mathbf{x}) = \nabla e^2(k) = -2e(k)\mathbf{z}(k)$$

- To calculate this approximate gradient we need **only multiply the error times the input**.

LMS Algorithm

- This approximation to $\nabla F(x)$ can now be used in the steepest descent algorithm.

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \nabla F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}_k}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + 2\alpha e(k)\mathbf{z}(k),$$

- In term of weights we can have the update equation

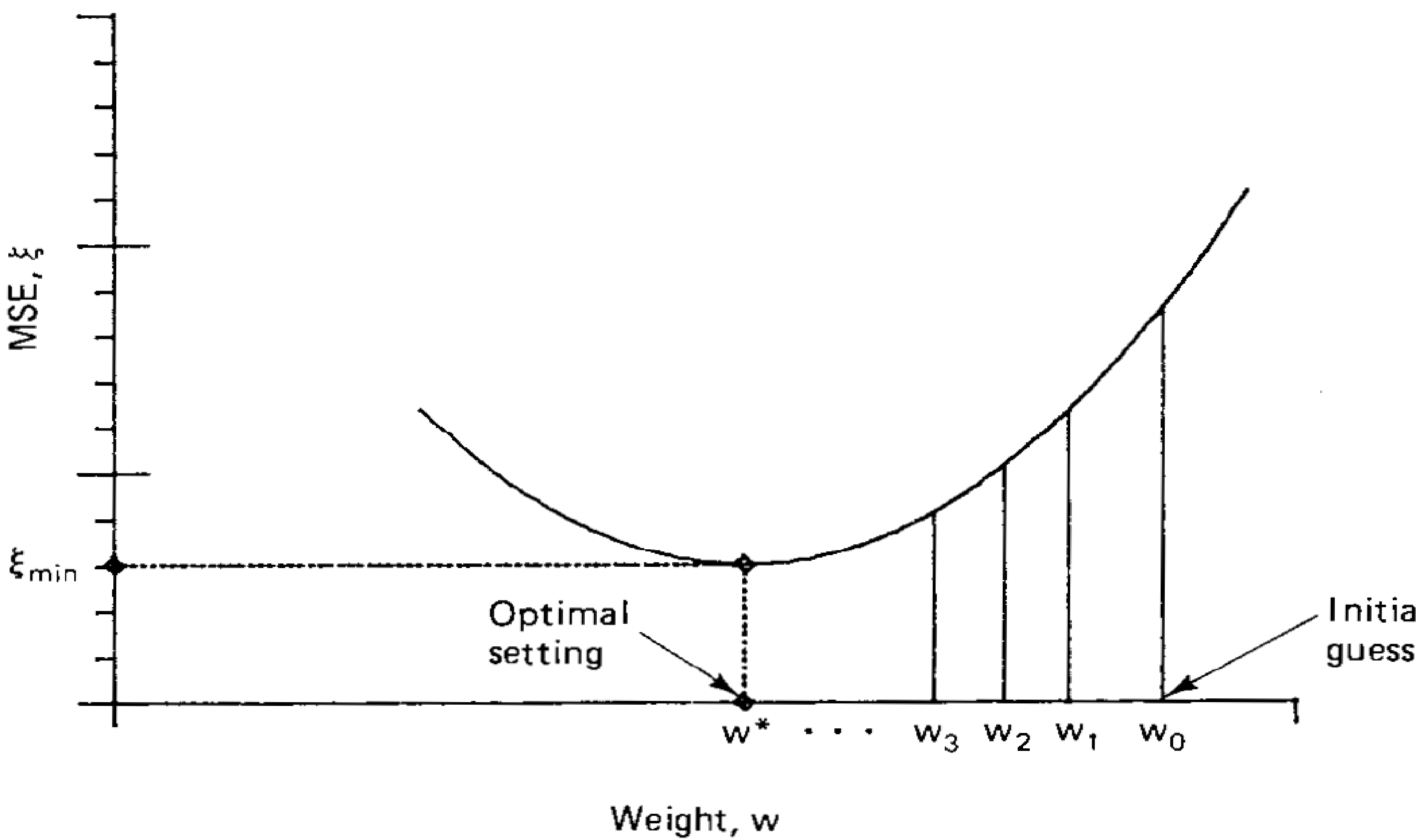
$${}_1\mathbf{w}(k+1) = {}_1\mathbf{w}(k) + 2\alpha e(k)\mathbf{p}(k),$$

$$b(k+1) = b(k) + 2\alpha e(k).$$

- These last two equations make up the **least mean square (LMS) algorithm**. This is also referred to as the delta rule or the **Widrow-Hoff learning algorithm**.

Basic idea of gradient search

$$\xi = \xi_{\min} + \lambda (w - w^*)^2$$



Basic idea of gradient search

- $\xi = \xi_{\min} + \lambda(w - w^*)^2 \qquad \frac{d\xi}{dw} = 2\lambda(w - w^*) \qquad \frac{d^2\xi}{dw^2} = 2\lambda$

- We begin with arbitrary value w_0 and measure the slope of the curve at this point.
- Then choose a new value w_1 which is equal to the initial value w_0 plus an increment proportion to the negative of the slop. This procedure is repeated until the optimal value w^* is reached.
- The value obtained by measuring different slopes of the performance surface at discrete intervals $w_0, w_1, w_2 ...$ are called gradient estimates.

Simple gradient search algorithm

- The repetitive gradient search algorithm can be defined as $w_{k+1} = w_k + \mu(-\nabla_k)$
- Where k is the iteration number.
- The gradient at $w = w_k$ is given by ∇_k.
- The gradient is obtained by

$$\nabla_k = \left.\frac{d\xi}{dw}\right|_{w=w_k} = 2\lambda(w_k - w^*)$$

$$w_{k+1} = w_k - 2\mu\lambda(w_k - w^*)$$

$$w_{k+1} = (1 - 2\mu\lambda)w_k + 2\mu\lambda w^*$$

Perceptron Learning Rule

- $$w_1 = (1 - 2\mu\lambda)w_0 + 2\mu\lambda w^*$$
$$w_2 = (1 - 2\mu\lambda)^2 w_0 + 2\mu\lambda w^* [(1 - 2\mu\lambda) + 1]$$
$$w_3 = (1 - 2\mu\lambda)^3 w_0 + 2\mu\lambda w^* \left[(1 - 2\mu\lambda)^2 + (1 - 2\mu\lambda) + 1\right]$$

$$w_k = (1 - 2\mu\lambda)^k w_0 + 2\mu\lambda w^* \sum_{n=0}^{k-1} (1 - 2\mu\lambda)^n$$

$$= (1 - 2\mu\lambda)^k w_0 + 2\mu\lambda w^* \frac{1 - (1 - 2\mu\lambda)^k}{1 - (1 - 2\mu\lambda)}$$

$$= w^* + (1 - 2\mu\lambda)^k (w_0 - w^*)$$

Perceptron Learning Rule

$$r = 1 - 2\mu\lambda$$