Institute for Advanced Management Systems Research
Department of Information Technologies
Åbo Akademi University

The delta learning rule - Tutorial

Robert Fullér

Directory

November 4, 2010

rfuller@abo.fi

Table of Contents

1. Error correction learning

The error correction learning procedure is simple enough in conception. The procedure is as follows: During training an input is put into the network and flows through the network generating a set of values on the output units.

Then, the actual output is compared with the desired target, and a match is computed. If the output and target match, no change is made to the net.

However, if the output differs from the target a change must be made to some of the connections.

Let's first recall the definition of derivative of single-variable functions.

Definition 1.1. *The derivative of f at (an interior point of its domain) x, denoted by $f'(x)$, and defined by*

$$f'(x) = \lim_{x_n \to x} \frac{f(x) - f(x_n)}{x - x_n}$$

Let us consider a differentiable function

$$f : \mathbb{R} \to \mathbb{R}.$$

The derivative of f at (an interior point of its domain) x is denoted by $f'(x)$.

If $f'(x) > 0$ then we say that f is increasing at x, if $f'(x) < 0$ then we say that f is decreasing at x, if $f'(x) = 0$ then f can have a local maximum, minimum or inflextion point at x.

A differentiable function is always increasing in the direction

of its derivative, and decreasing in the opposite direction.

It means that if we want to find one of the local minima of a function f starting from a point x^0 then we should search for a second candidate:

- in the right-hand side of x^0 if $f'(x^0) < 0$, when f is decreasing at x^0;
- in the left-hand side of x^0 if $f'(x^0) > 0$, when f increasing at x^0.

The equation for the line crossing the point $(x^0, f(x^0))$ is given by

$$\frac{y - f(x^0)}{x - x^0} = f'(x^0)$$

that is

$$y = f(x^0) + (x - x^0)f'(x^0)$$

The next approximation, denoted by x^1, is a solution to the equation

$$f(x^0) + (x - x^0)f'(x^0) = 0$$

which is,

$$x^1 = x^0 - \frac{f(x^0)}{f'(x^0)}$$

This idea can be applied successively, that is

$$x^{n+1} = x^n - \frac{f(x^n)}{f'(x^n)}.$$

The above procedure is a typical descent method. In a descent method the next iteration w^{n+1} should satisfy the following property

$$f(w^{n+1}) < f(w^n)$$

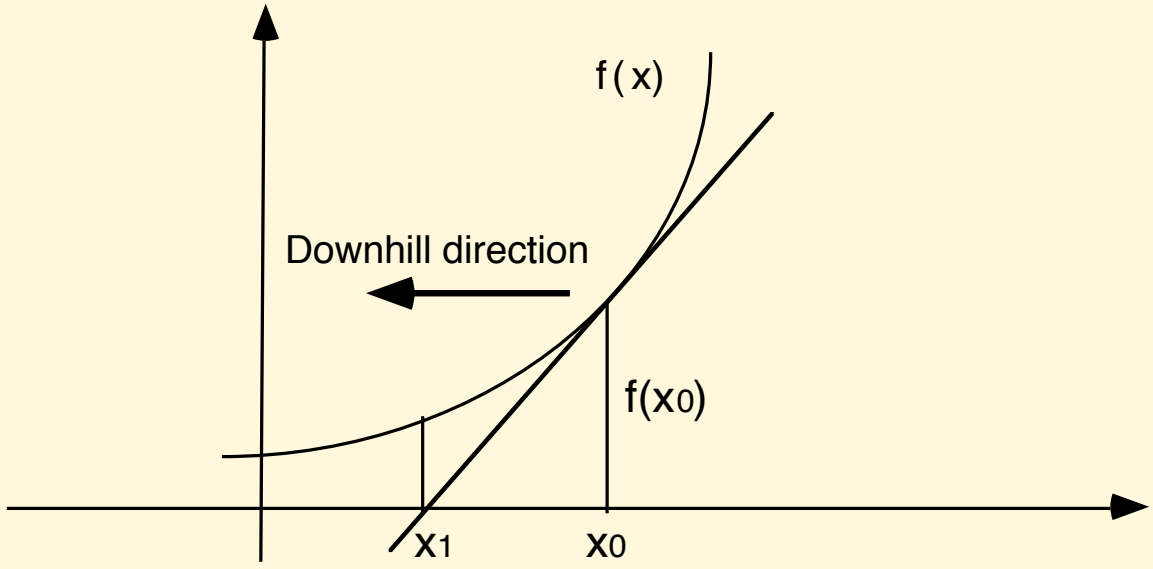

Figure 1: The downhill direction is negative at x_0.

i.e. the value of f at w^{n+1} is smaller than its previous value at w^n.

In error correction learning procedure, each iteration of a descent method calculates the downhill direction (opposite of the direction of the derivative) at w^n which means that for a suffi-

ciently small $\eta > 0$ the inequality

$$f(w^n - \eta f'(w^n)) < f(w^n)$$

should hold, and we let w^{n+1} be the vector

$$w^{n+1} = w^n - \eta f'(w^n)$$

Let $f : \mathbb{R}^n \to \mathbb{R}$ be a real-valued function. In a descent method, whatever is the next iteration, w^{n+1}, it should satisfy the property

$$f(w^{n+1}) < f(w^n)$$

i.e. the value of f at w^{n+1} is smaller than its value at previous approximation w^n.

Each iteration of a descent method calculates a downhill direction (opposite of the direction of the derivative) at w^n which

means that for a sufficiently small $\eta > 0$ the inequality

$$f(w^n - \eta f'(w^n)) < f(w^n)$$

should hold, and we let w^{n+1} be the vector

$$w^{n+1} = w^n - \eta f'(w^n).$$

Let $f : \mathbb{R}^n \to \mathbb{R}$ be a real-valued function and let $e \in \mathbb{R}^n$ with $\|e\| = 1$ be a given direction. The derivative of f with respect e at w is defined as

$$\partial_e f(w) = \lim_{t \to +0} \frac{f(w + te) - f(w)}{t}.$$

If

$$e = (0, \ldots \overbrace{1}^{i\text{-th}} \ldots, 0)^T,$$

i.e. e is the i-th basic direction then instead of $\partial_e f(w)$ we write $\partial_i f(w)$, which is defined by

$$\partial_i f(w) = \lim_{t \to +0} \frac{f(w_1, \ldots, w_i + t, \ldots w_n) - f(w_1, \ldots, w_i, \ldots, w_n)}{t}.$$

The gradient of f at w, denoted by $f'(w)$ is defined by

$$f'(w) = (\partial_1 f(w), \ldots, \partial_n f(w))^T$$

Example 1.1. *Let $f(w_1, w_2) = w_1^2 + w_2^2$ then the gradient of f is given by*

$$f'(w) = 2w = (2w_1, 2w_2)^T.$$

The gradient vector always points to the uphill direction of f. The downhill (steepest descent) direction of f at w is the

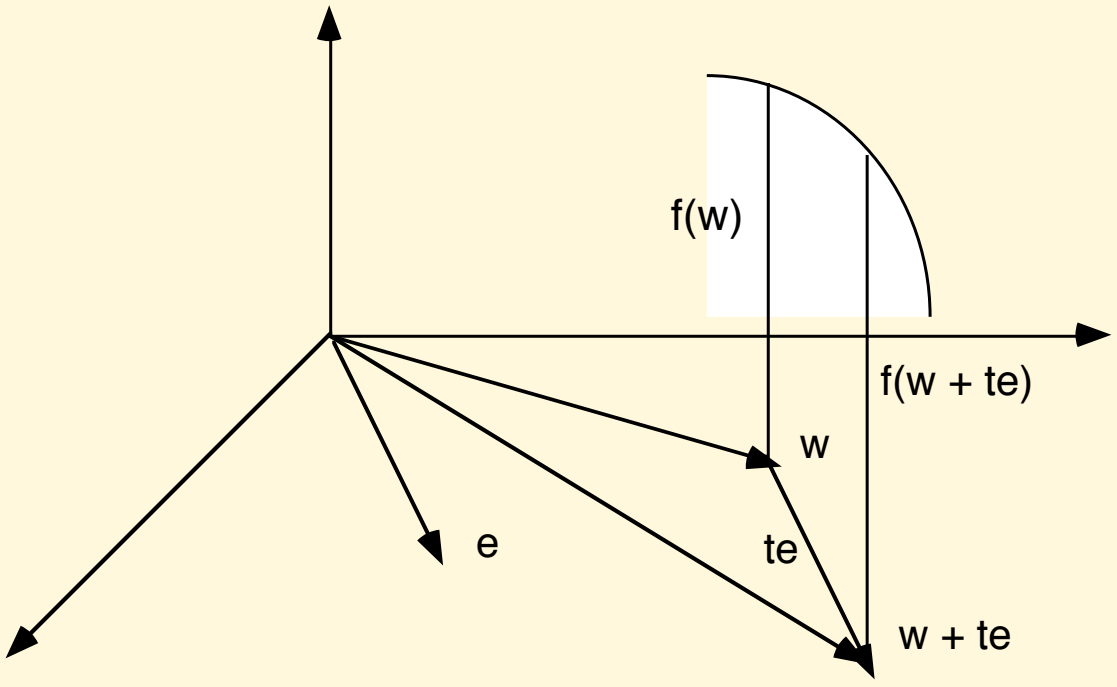

Figure 2: The derivative of f with respect to the direction e.

opposite of the uphill direction, i.e. the downhill direction is $-f'(w)$, which is

$$(-\partial_1 f(w), \ldots, -\partial_n f(w))^T.$$

2. Linear activation function

Definition 2.1. *A linear activation function is a mapping from* $f : \mathbb{R} \to \mathbb{R}$ *such that* $f(t) = t$ *for all* $t \in \mathbb{R}$.

Suppose we are given a single-layer network with n input units and m linear output units, i.e. the output of the i-th neuron can be written as

$$o_i = net_i =< w_i, x >= w_{i1}x_1 + \cdots + w_{in}x_n,$$

for $i = 1, \ldots, m$. Assume we have the following training set

$$\{(x^1, y^1), \ldots, (x^K, y^K)\}$$

where $x^k = (x_1^k, \ldots, x_n^k)$, $y^k = (y_1^k, \ldots, y_m^k)$, $k = 1, \ldots, K$.

The basic idea of the delta learning rule is to define a measure of the overall performance of the system and then to find a way

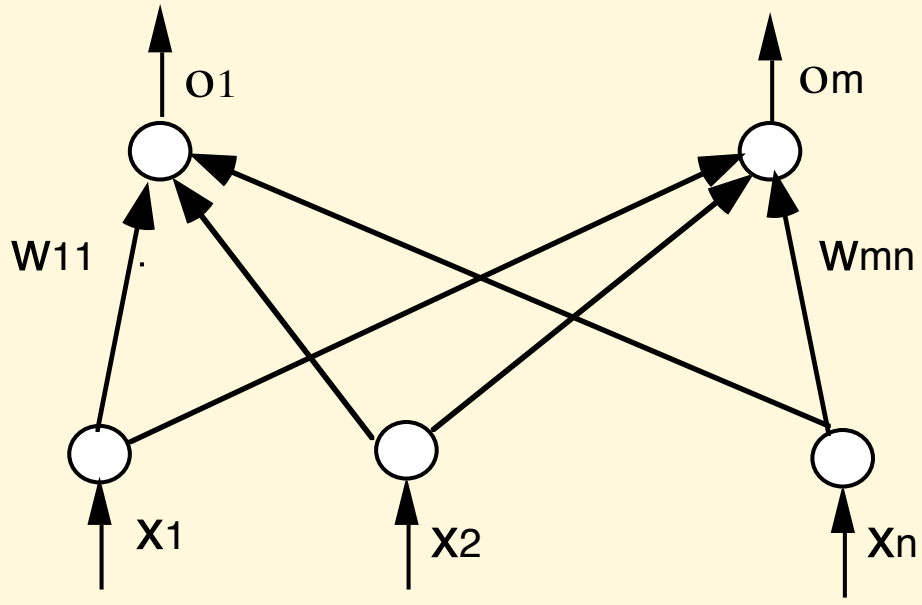

Figure 3: Single-layer feedforward network with m output units.

to optimize that performance.

In our network, we can define the performance of the system as

$$E = \sum_{k=1}^{K} E_k = \frac{1}{2} \sum_{k=1}^{K} \|y^k - o^k\|^2$$

That is

$$E = \frac{1}{2} \sum_{k=1}^{K} \sum_{i=1}^{m} (y_i^k - o_i^k)^2$$

$$= \frac{1}{2} \sum_{k=1}^{K} \sum_{i=1}^{m} (y_i^k - < w_i, x^k >)^2,$$

where i indexes the output units; k indexes the input/output pairs to be learned; y_i^k indicates the target for a particular output unit on a particular pattern;

$$o_i^k :=< w_i, x^k >,$$

indicates the actual output for that unit on that pattern; and E is

the total error of the system. The goal, then, is to minimize this function. It turns out, if the output functions are differentiable, that this problem has a simple solution: namely, we can assign a particular unit blame in proportion to the degree to which changes in that unit's activity lead to changes in the error.

That is, we change the weights of the system in proportion to the derivative of the error with respect to the weights.

The rule for changing weights following presentation of input/output pair (x, y) (we omit the index k for the sake of simplicity) is given by the gradient descent method, i.e. we minimize the quadratic error function by using the following iteration process

$$w_{ij} := w_{ij} - \eta \frac{\partial E_k}{\partial w_{ij}}$$

where $\eta > 0$ is the learning rate.

Let us compute now the partial derivative of the error function E_k with respect to w_{ij}

$$\frac{\partial E_k}{\partial w_{ij}} = \frac{\partial E_k}{\partial net_i}\frac{\partial net_i}{\partial w_{ij}} = -(y_i - o_i)x_j$$

where $net_i = w_{i1}x_1 + \cdots + w_{in}x_n$.

That is,

$$w_{ij} := w_{ij} + \eta(y_i - o_i)x_j$$

for $j = 1, \ldots, n$.

Definition 2.2. *The error signal term, denoted by δ_i^k and called* delta, *produced by the i-th output neuron is defined as*

$$\delta_i = -\frac{\partial E_k}{\partial net_i} = (y_i - o_i)$$

For linear output units δ_i is nothing else but the difference between the desired and computed output values of the i-th neu-

ron.

So the delta learning rule can be written as

$$w_i := w_i + \eta(y_i - o_i)x$$

where w_i is the weight vector of the i-th output unit, x is the actual input to the system, y_i is the i-th coordinate of the desired output, o_i is the i-th coordinate of the computed output and $\eta > 0$ is the learning rate.

If we have only one output unit then the delta learning rule collapses into

$$w := w + \eta(y - o)x = w + \eta\delta x$$

where δ denotes the difference between the desired and the

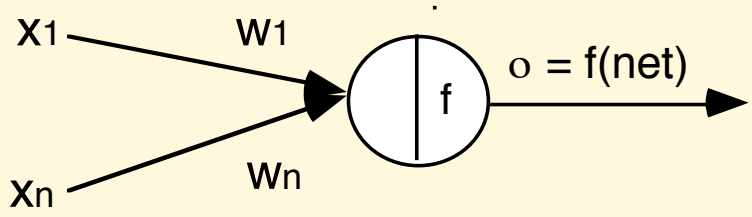

Figure 4: A single neuron net.

computed output.

A key advantage of neural network systems is that these simple, yet powerful learning procedures can be defined, allowing the systems to adapt to their environments.

The essential character of such networks is that they map similar input patterns to similar output patterns.

This characteristic is what allows these networks to make rea-

sonable generalizations and perform reasonably on patterns that have never before been presented. The similarity of patterns in a connectionist system is determined by their overlap.

The overlap in such networks is determined outside the learning system itself whatever produces the patterns.

The standard delta rule essentially implements gradient descent in sum-squared error for linear activation functions.

It should be noted that the delta learning rule was introduced only recently for neural network training by McClelland and Rumelhart.

This rule parallels the discrete perceptron training rule. It also can be called the **continuous perceptron training rule**.

3. The delta learning rule

The delta learning rule with linear activation functions. Given are K training pairs arranged in the training set

$$\{(x^1, y^1), \ldots, (x^K, y^K)\}$$

where $x^k = (x_1^k, \ldots, x_n^k)$ and $y^k = (y_1^k, \ldots, y_m^k)$, $k = 1, \ldots, K$.

- **Step 1** $\eta > 0$, $E_{\max} > 0$ are chosen
- **Step 2** Weights w_{ij} are initialized at small random values, $k := 1$, and the running error E is set to 0
- **Step 3** Training starts here. Input x^k is presented, $x := x^k$, $y := y^k$, and output $o = (o_1, \ldots, o_m)^T$ is computed

  $$o_i =< w_i, x >) = w_i^T x$$

  for $i = 1, \ldots, m$.

- **Step 4** Weights are updated

$$w_{ij} := w_{ij} + \eta(y_i - o_i)x_j$$

- **Step 5** Cumulative cycle error is computed by adding the present error to E

$$E := E + \frac{1}{2}\|y - o\|^2$$

- **Step 6** If $k < K$ then $k := k + 1$ and we continue the training by going back to **Step 3**, otherwise we go to **Step 7**

- **Step 7** The training cycle is completed. For $E < E_{\max}$ terminate the training session. If

$$E > E_{\max}$$

then E is set to 0 and we initiate a new training cycle by going back to **Step 3**

Exercise 1. *Construct a single-neuron network, which computes the following function. The training set is*

	x_1	x_2	x_3	$o(x_1, x_2, x_3)$
1.	*1*	*1*	*1*	*1*
2.	*1*	*1*	*0*	*1*
3.	*1*	*0*	*0*	*1*
4.	*0*	*1*	*1*	*0*
5.	*0*	*0*	*1*	*0*
6.	*0*	*1*	*0*	*0*
7.	*1*	*0*	*1*	*1*

Solution 1. *For example,* $w_1 = 1$ *and* $w_2 = w_3 = 0$.

Exercise 2. *The error function to be minimized is given by*

$$E(w_1, w_2, w_3, w_4) = \frac{1}{2}\Big[(w_1 - w_2 - w_3)^2 + (-w_1 + w_2 - w_3)^2 + (-w_1 - w_2 - 1)^2 + (w_4 + 1)^2\Big]$$

Find analytically the gradient vector

$$E'(w) = \begin{bmatrix} \partial_1 E(w) \\ \partial_2 E(w) \\ \partial_3 E(w) \\ \partial_4 E(w) \end{bmatrix}$$

Find analytically the weight vectors w^* that minimizes the error function such that $E'(w^*) = 0$. Derive the steepest descent method for the minimization of E.

Solution 2. *The gradient vector of E is*

$$E'(w) = \begin{bmatrix} 3w_1 - w_2 + 1 \\ -w_1 + 3w_2 + 1 \\ 2w_3 \\ w_4 + 1 \end{bmatrix}$$

and

$$w^* = (-1/2, -1/2, 0, -1)^T$$

is the unique solution to the equation $E'(w) = 0$.

The steepest descent method for the minimization of E reads

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} := \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} - \eta \begin{bmatrix} 3w_1 - w_2 + 1 \\ -w_1 + 3w_2 + 1 \\ 2w_3 \\ w_4 + 1 \end{bmatrix}.$$

where $\eta > 0$ is the learning constant.

That is,

$$
\begin{aligned}
w_1 &:= w_1 - \eta(3w_1 - w_2 + 1),\\
w_2 &:= w_2 - \eta(-w_1 + 3w_2 + 1),\\
w_3 &:= w_3 - \eta(2w_3),\\
w_4 &:= w_4 - \eta(w_4 + 1).
\end{aligned}
$$

4. The delta learning rule with semilinear activation function

The standard delta rule essentially implements gradient descent in sum-squared error for linear activation functions.

We shall describe now the delta learning rule with semilinear activation function. For simplicity we explain the learning algorithm in the case of a single-output network.

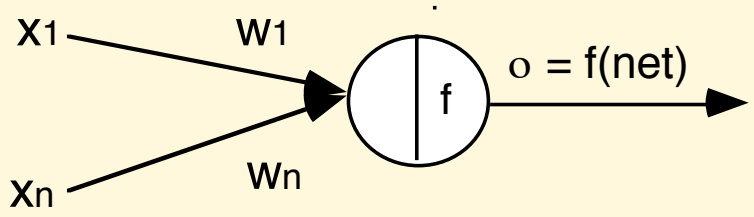

Figure 5: A single neuron net.

The output of the neuron is computed by unipolar sigmoidal activation function

$$o(< w, x >) = \frac{1}{1 + \exp(-w^T x)}.$$

Suppose we are given the following training set

No.	input values	desired output value
1.	$x^1 = (x_1^1, \ldots x_n^1)$	y^1
2.	$x^2 = (x_1^2, \ldots x_n^2)$	y^2
$\vdots$	$\vdots$	$\vdots$
K.	$x^K = (x_1^K, \ldots x_n^K)$	y^K

The system first uses the input vector, x^k, to produce its own output vector, o^k, and then compares this with the desired output, y^k.

If we use a linear output unit then whatever is the final weight vector, the output function of the network is a line (in two-dimensional case).

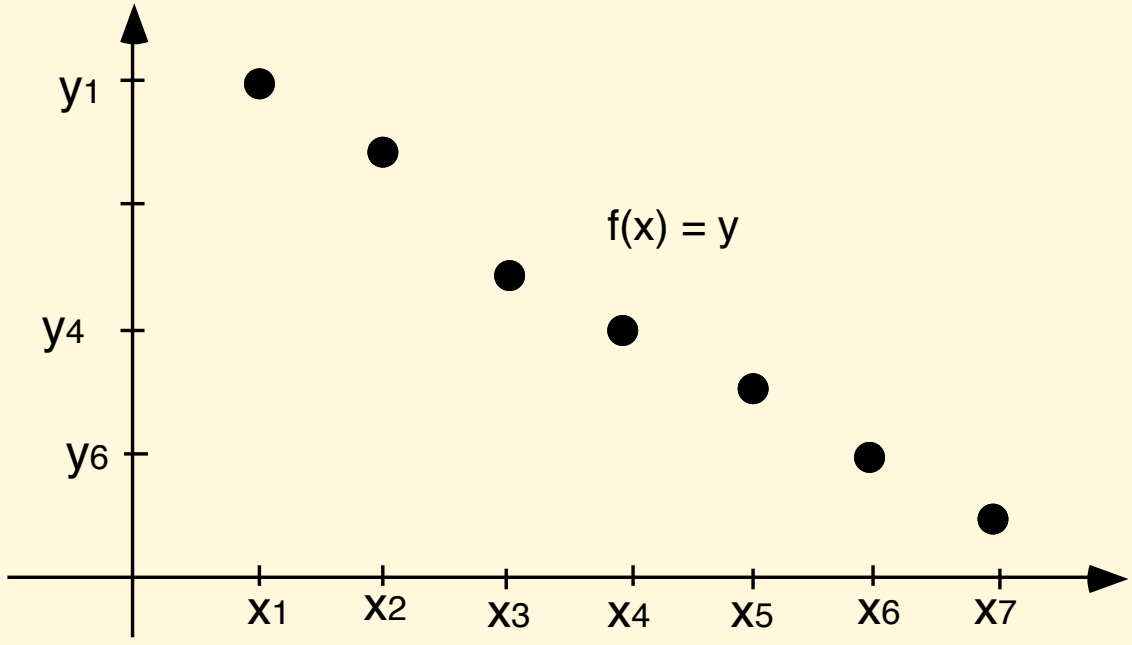

Figure 6: A pattern set from an almost linear function f.

It means that the delta learning rule with linear output function can approximate only a pattern set derived from an almost linear function.

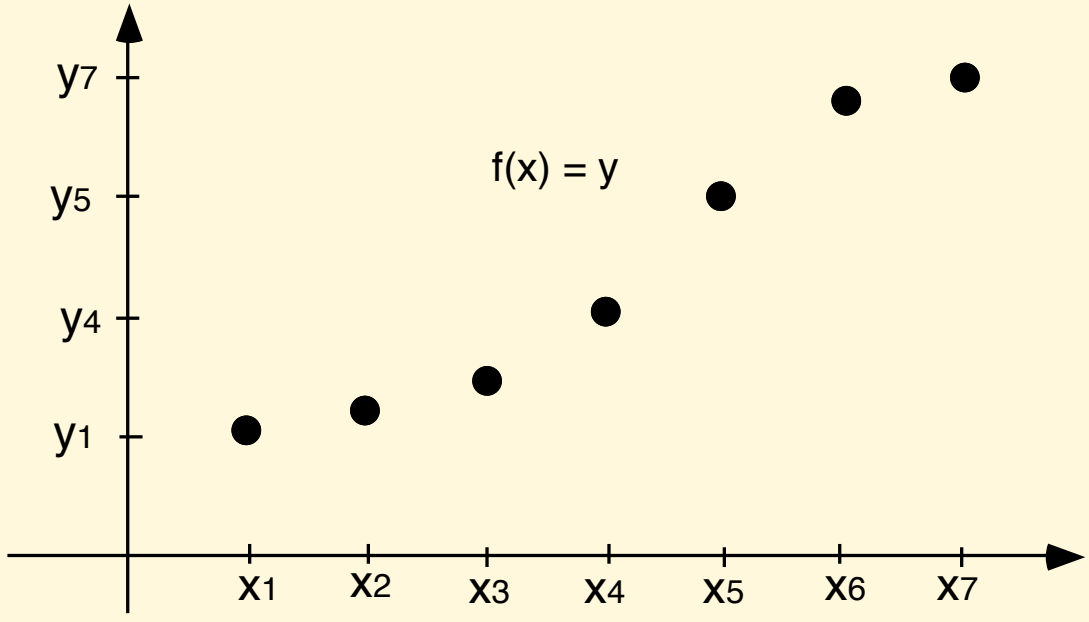

Figure 7: A pattern set from a nonlinear function f.

However, if the patterns are derived from a nonlinear function f, then the chances for a good approximation are small. It is why we use semilinear activation functions.

Let

$$E_k = \frac{1}{2}(y^k - o^k)^2 = \frac{1}{2}(y^k - o(< w, x^k >))^2 =$$

$$\frac{1}{2} \times \left[y^k - \frac{1}{1 + \exp\left(-w^T x^k\right)} \right]^2$$

be our measure of the error on input/output pattern k and let

$$E = \sum_{k=1}^{K} E_k = E_1 + E_2 + \cdots + E_K,$$

be our overall measure of the error.

The rule for changing weights following presentation of input/output pair k is given by the gradient descent method, i.e. we minimize the quadratic error function by using the follow-

ing iteration process

$$w := w - \eta E'_k(w).$$

Let us compute now the gradient vector of the error function E_k at point w:

$$E'_k(w) = \frac{d}{dw}\left(\frac{1}{2} \times \left[y^k - \frac{1}{1+\exp\left(-w^T x^k\right)}\right]^2\right) =$$

$$\frac{1}{2} \times \frac{d}{dw}\left[y^k - \frac{1}{1+\exp\left(-w^T x^k\right)}\right]^2 =$$

$$-(y^k - o^k)o^k(1-o^k)x^k$$

where $o^k = 1/(1+\exp\left(-w^T x^k\right))$.

Therefore our learning rule for w is

$$w := w + \eta(y^k - o^k)o^k(1 - o^k)x^k$$

which can be written as

$$w := w + \eta\delta_k o^k(1 - o^k)x^k$$

where

$$\delta_k = (y^k - o^k)o^k(1 - o^k).$$

The delta learning rule with unipolar sigmoidal activation function.

Given are K training pairs arranged in the training set

$$\{(x^1, y^1), \ldots, (x^K, y^K)\}$$

where $x^k = (x_1^k, \ldots, x_n^k)$ and $y^k \in \mathbb{R}$, $k = 1, \ldots, K$.

- **Step 1** $\eta > 0$, $E_{\max} > 0$ are chosen
- **Step 2** Weigts w are initialized at small random values, $k := 1$, and the running error E is set to 0
- **Step 3** Training starts here. Input x^k is presented, $x := x^k$, $y := y^k$, and output o is computed

$$o = o(< w, x >) = \frac{1}{1 + \exp(-w^T x)}$$

- **Step 4** Weights are updated

$$w := w + \eta(y - o)o(1 - o)x$$

- **Step 5** Cumulative cycle error is computed by adding the present error to E

$$E := E + \frac{1}{2}(y - o)^2$$

- **Step 6** If $k < K$ then $k := k + 1$ and we continue the training by going back to **Step 3**, otherwise we go to **Step 7**

- **Step 7** The training cycle is completed. For $E < E_{\max}$ terminate the training session. If $E > E_{\max}$ then E is set to 0 and we initiate a new training cycle by going back to **Step 3**

In this case, without hidden units, the error surface is shaped like a bowl with only one minimum, so gradient descent is guaranteed to find the best set of weights. With hidden units, how-

ever, it is not so obvious how to compute the derivatives, and the error surface is not concave upwards, so there is the danger of getting stuck in local minima.

We illustrate the delta learning rule with bipolar sigmoidal activation function

$$f(t) = \frac{2}{(1 + \exp -t) - 1}.$$

Example 4.1. *The delta learning rule with bipolar sigmoidal activation function. Given are K training pairs arranged in the training set*

$$\{(x^1, y^1), \ldots, (x^K, y^K)\}$$

where $x^k = (x_1^k, \ldots, x_n^k)$ and $y^k \in \mathbb{R}$, $k = 1, \ldots, K$.

- **Step 1** $\eta > 0$, $E_{\max} > 0$ *are chosen*
- **Step 2** *Weigts* w *are initialized at small random values,* $k := 1$, *and the running error* E *is set to 0*
- **Step 3** *Training starts here. Input* x^k *is presented,* $x := x^k$, $y := y^k$, *and output* o *is computed*

$$o = o(< w, x >) = \frac{2}{1 + \exp(-w^T x)} - 1$$

- **Step 4** *Weights are updated*

$$w := w + \frac{1}{2}\eta(y - o)(1 - o^2)x$$

- **Step 5** *Cumulative cycle error is computed by adding the present error to* E

$$E := E + \frac{1}{2}(y - o)^2$$

- **Step 6** *If* $k < K$ *then* $k := k + 1$ *and we continue the training by going back to* **Step 3**, *otherwise we go to* **Step 7**

- **Step 7** *The training cycle is completed. For* $E < E_{\max}$ *terminate the training session. If* $E > E_{\max}$ *then* E *is set to 0 and we initiate a new training cycle by going back to* **Step 3**

Exercise 3. *The error function to be minimized is given by*

$$E(w_1, w_2) = \frac{1}{2}[(w_2 - w_1)^2 + (1 - w_1)^2]$$

Find analytically the gradient vector

$$E'(w) = \begin{bmatrix} \partial_1 E(w) \\ \partial_2 E(w) \end{bmatrix}$$

Find analytically the weight vector w^* *that minimizes the error*

function such that

$$E'(w) = 0.$$

Derive the steepest descent method for the minimization of E.

Solution 3. *The gradient vector of* E *is*

$$E'(w) = \begin{bmatrix} (w_1 - w_2) + (w_1 - 1) \\ (w_2 - w_1) \end{bmatrix} = \begin{bmatrix} 2w_1 - w_2 - 1 \\ w_2 - w_1 \end{bmatrix}$$

and $w^*(1,1)^T$ *is the unique solution to the equation*

$$\begin{bmatrix} 2w_1 - w_2 - 1 \\ w_2 - w_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

The steepest descent method for the minimization of E *reads*

$$\begin{bmatrix} w_1(t+1) \\ w_2(t+1) \end{bmatrix} = \eta \begin{bmatrix} 2w_1(t) - w_2(t) - 1 \\ w_2(t) - w_1(t) \end{bmatrix}.$$

where $\eta > 0$ *is the learning constant and* t *indexes the number of iterations.*

That is,

$$w_1(t+1) = w_1(t) - \eta(2w_1(t) - w_2(t) - 1),$$
$$w_2(t+1) = w_2(t) - \eta(w_2(t) - w_1(t)),$$

or, equivalently,

$$w_1 := w_1 - \eta(2w_1(t) - w_2 - 1),$$
$$w_2 := w_2 - \eta(w_2(t) - w_1).$$

5. The generalized delta learning rule

We now focus on generalizing the delta learning rule for feed-forward layered neural networks. The architecture of the two-layer network considered below is shown in the figure. It has strictly speaking, two layers of processing neurons.

If, however, the layers of nodes are counted, then the network can also be labeled as a three-layer network. There is no agreement in the literature as to which approach is to be used to describe network architectures.

In this text we will use the term *layer* in reference to the actual number of existing and processing neuron layers. Layers with neurons whose outputs are not directly accesible are called internal or hidden layers. Thus the network of the figure is a two-layer network, which can be called a single hidden-layer network.

The generalized delta rule is the most often used supervised learning algorithm of feedforward multi-layer neural networks. For simplicity we consider only a neural network with one hidden layer and one output node.

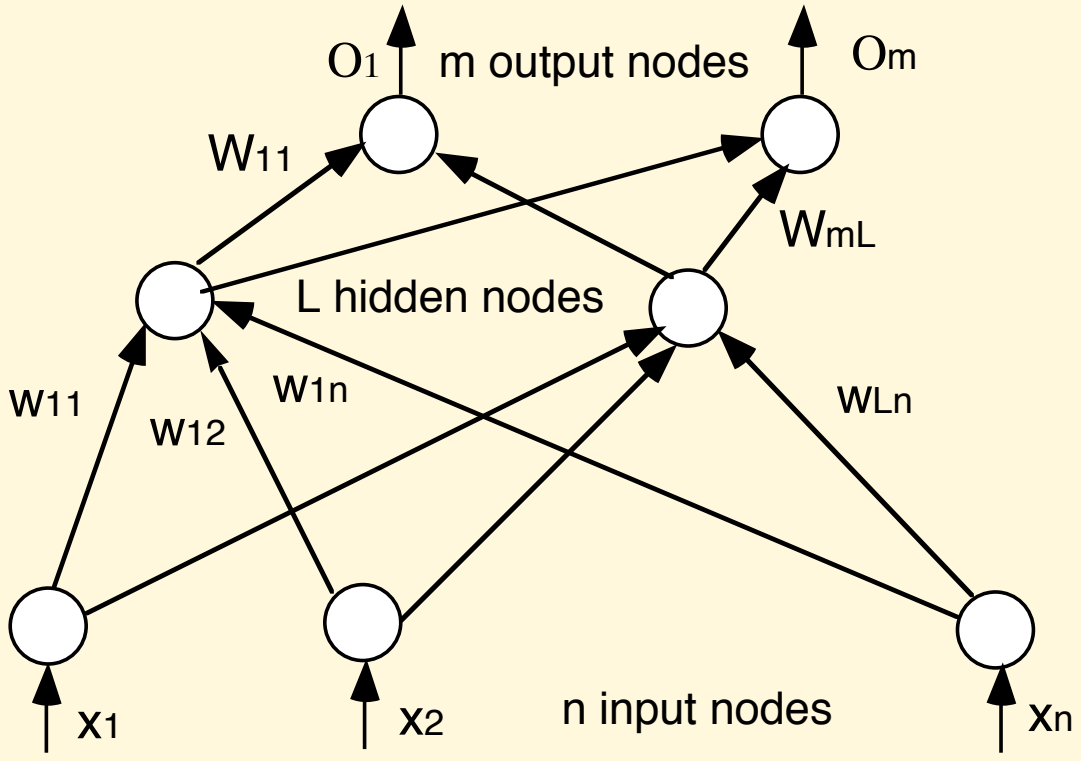

Figure 8: Layered neural network with two continuous perceptron layers.

The measure of the error on an input/output training pattern

$$(x^k, y^k)$$

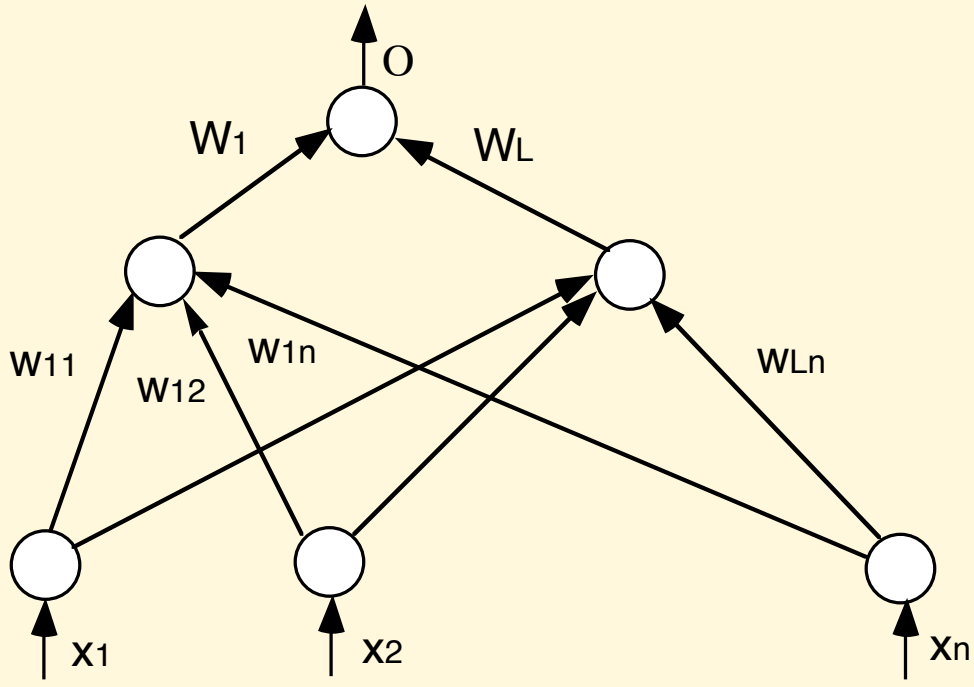

Figure 9: Two-layer neural network with one output node.

is defined by

$$E_k(W, w) = \frac{1}{2}(y^k - O^k)^2$$

where O^k is the computed output and the overall measure of

the error is

$$E(W, w) = \sum_{k=1}^{K} E_k(W, w).$$

If an input vector x^k is presented to the network then it generates the following output

$$O^k = \frac{1}{1 + \exp(-W^T o^k)}$$

where o^k is the output vector of the hidden layer

$$o_l^k = \frac{1}{1 + \exp(-w_l^T x^k)}$$

and w_l denotes the weight vector of the l-th hidden neuron, $l = 1, \ldots, L$.

The rule for changing weights following presentation of input/output pair k is given by the gradient descent method, i.e.

we minimize the quadratic error function by using the following iteration process

$$W := W - \eta \frac{\partial E_k(W, w)}{\partial W},$$

$$w_l := w_l - \eta \frac{\partial E_k(W, w)}{\partial w_l},$$

for $l = 1, \ldots, L$, and $\eta > 0$ is the learning rate.

By using the chain rule for derivatives of composed functions we get

$$\begin{aligned} \frac{\partial E_k(W, w)}{\partial W} &= \frac{1}{2} \frac{\partial}{\partial W} \left[y^k - \frac{1}{1 + \exp(-W^T o^k)} \right]^2 \\ &= -(y^k - O^k) O^k (1 - O^k) o^k \end{aligned}$$

So the rule for changing weights of the output unit is

$$W := W + \eta(y^k - O^k)O^k(1 - O^k)o^k = W + \eta\delta_k o^k$$

that is

$$W_l := W_l + \eta\delta_k o_l^k,$$

for $l = 1, \ldots, L$, and we have used the notation

$$\delta_k = (y^k - O^k)O^k(1 - O^k).$$

Let us now compute the partial derivative of E_k with respect to w_l

$$\frac{\partial E_k(W, w)}{\partial w_l} = -O^k(1 - O^k)W_l o_l^k(1 - o_l^k)x^k$$

i.e. the rule for changing weights of the hidden units is

$$w_l := w_l + \eta\delta_k W_l o_l^k(1 - o_l^k)x^k,$$

for $l = 1, \ldots, L$. That is

$$w_{lj} := w_{lj} + \eta \delta_k W_l o_l^k (1 - o_l^k) x_j^k,$$

for $j = 1, \ldots, n$.

Summary. *The generalized delta learning rule (error back-propagation learning). We are given the training set*

$$\{(x^1, y^1), \ldots, (x^K, y^K)\}$$

where $x^k = (x_1^k, \ldots, x_n^k)$ *and* $y^k \in \mathbb{R}$, $k = 1, \ldots, K$.

- **Step 1** $\eta > 0$, $E_{\max} > 0$ *are chosen*
- **Step 2** *Weigts* w *are initialized at small random values,* $k := 1$, *and the running error* E *is set to 0*
- **Step 3** *Training starts here. Input* x^k *is presented,* $x :=$

x^k, $y := y^k$, and output O is computed

$$O = \frac{1}{1 + \exp(-W^T o)}$$

where o_l is the output vector of the hidden layer

$$o_l = \frac{1}{1 + \exp(-w_l^T x)}$$

- **Step 4** *Weights of the output unit are updated*

$$W := W + \eta \delta o$$

where $\delta = (y - O)O(1 - O)$.

- **Step 5** *Weights of the hidden units are updated*

$$w_l = w_l + \eta \delta W_l o_l (1 - o_l) x, \; l = 1, \ldots, L$$

- **Step 6** *Cumulative cycle error is computed by adding the present error to* E

$$E := E + \frac{1}{2}(y - O)^2$$

- **Step 7** *If* $k < K$ *then* $k := k + 1$ *and we continue the training by going back to* **Step 2**, *otherwise we go to* **Step 8**

- **Step 8** *The training cycle is completed. For* $E < E_{\max}$ *terminate the training session. If* $E > E_{\max}$ *then* $E := 0$, $k := 1$ *and we initiate a new training cycle by going back to* **Step 3**

Exercise 4. *Derive the backpropagation learning rule with bipolar sigmoidal activation function*

$$f(t) = \frac{2}{(1 + \exp -t) - 1}.$$

6. Effectivity of neural networks

Funahashi (1989) showed that infinitely large neural networks with a single hidden layer are capable of approximating all continuous functions. Namely, he proved the following theorem

Theorem 6.1. *Let $\phi(x)$ be a nonconstant, bounded and monotone increasing continuous function. Let $K \subset \mathbb{R}^n$ be a compact set and*

$$f\colon K \to \mathbb{R}$$

be a real-valued continuous function on K. Then for arbitrary $\epsilon > 0$, there exists an integer N and real constants w_i, w_{ij} such that

$$\tilde{f}(x_1, \ldots, x_n) = \sum_{i=1}^{N} w_i \phi(\sum_{j=1}^{n} w_{ij} x_j)$$

satisfies

$$\|f - \tilde{f}\|_\infty = \sup_{x \in K} |f(x) - \tilde{f}(x)| \leq \epsilon.$$

In other words, any continuous mapping can be approximated in the sense of uniform topology on K by input-output mappings of two-layers networks whose output functions for the hidden layer are $\phi(x)$ and are linear for the output layer.

The Stone-Weierstrass theorem from classical real analysis can be used to show certain network architectures possess the universal approximation capability.

By employing the Stone-Weierstrass theorem in the designing of our networks, we also guarantee that the networks can compute certain polynomial expressions: if we are given networks exactly computing two functions, f_1 and f_2, then a larger net-

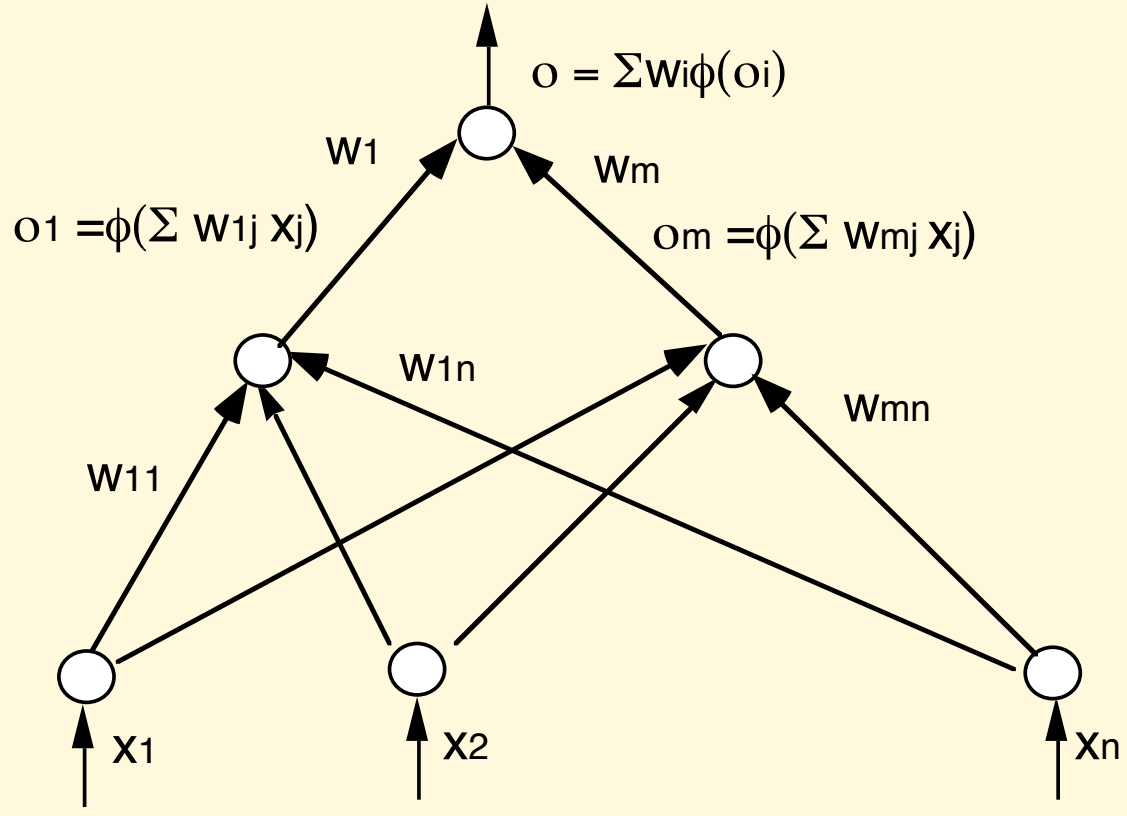

Figure 10: Funahashi's network.

work can exactly compute a polynomial expression of f_1 and f_2.

Theorem 6.2. *(Stone-Weierstrass) Let domain K be a compact space of n dimensions, and let $\mathcal{G}$ be a set of continuous real-valued functions on K, satisfying the following criteria:*

1. *The constant function $f(x) = \mathbf{1}$ is in $\mathcal{G}$.*
2. *For any two points $x_1 \neq x_2$ in K, there is an f in $\mathcal{G}$ such that $f(x_1) \neq f(x_2)$.*
3. *If f_1 and f_2 are two functions in $\mathcal{G}$, then $f\,g$ and $\alpha_1 f_1 + \alpha_2 f_2$ are in $\mathcal{G}$ for any two real numbers α_1 and α_2.*

Then $\mathcal{G}$ is dense in $\mathcal{C}(K)$, the set of continuous real-valued functions on K. In other words, for any $\epsilon > 0$ and any function g in $\mathcal{C}(K)$, there exists g in $\mathcal{G}$ such that

$$\|f - g\|_\infty = \sup_{x \in K} |f(x) - g(x)| \leq \epsilon.$$

The key to satisfying he Stone-Weierstrass theorem is to find functions that *transform multiplication into addition* so that products can be written as summations.

There are at least three generic functions that accomplish this transfomation: exponential functions, partial fractions, and step functions.

The following networks satisfy the Stone-Weierstrass theorem.

- **Decaying-exponential networks**

  Exponential functions are basic to the process of transforming multiplication into addition in several kinds of networks:

$$\exp(x_1)\exp(x_2) = \exp(x_1 + x_2).$$

Let $\mathcal{G}$ be the set of all continuous functions that can be computed by arbitrarily large decaying-exponential networks on domain $K = [0,1]^n$:

$$\mathcal{G} = \Bigg\{ f(x_1, \ldots, x_n)$$

$$= \sum_{i=1}^{N} w_i \exp(-\sum_{j=1}^{n} w_{ij} x_j), w_i, w_{ij} \in \mathbf{R} \Bigg\}.$$

Then $\mathcal{G}$ is dense in $\mathcal{C}(K)$

- **Fourier networks**
- **Exponentiated-function networks**
- **Modified logistic networks**
- **Modified sigma-pi and polynomial networks**

Let $\mathcal{G}$ be the set of all continuous functions that can be computed by arbitrarily large modified sigma-pi or polynomial networks on domain $K = [0,1]^n$:

$$\mathcal{G} = \left\{ f(x_1, \ldots, x_n) = \sum_{i=1}^{N} w_i \prod_{j=1}^{n} x_j^{w_{ij}}, w_i, w_{ij} \in \mathbf{R} \right\}.$$

Then $\mathcal{G}$ is dense in $\mathcal{C}(K)$.

- **Step functions and perceptron networks**
- **Partial fraction networks**