# Introduction To Perceptron Networks

Jan Jantzen <jj@iau.dtu.dk>[1]

## Abstract

When it is time-consuming or expensive to model a plant using the basic laws of physics, a neural network approach can be an alternative. From a control engineer's viewpoint a two-layer perceptron network is sufficient. It is indicated how to model a dynamic plant using a perceptron network.

## Contents



[1] Technical University of Denmark, Department of Automation, Bldg 326, DK-2800 Lyngby, DENMARK. Tech. report no 98-H 873 (nnet), 25 Oct 1998.

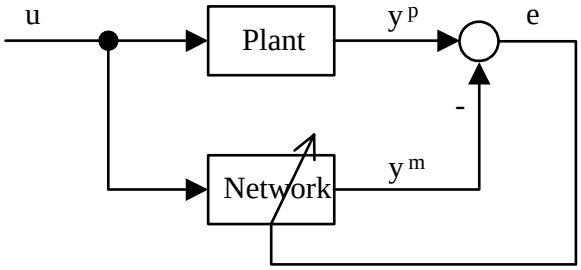


Figure 1: A neural network models a plant in a forward learning manner.

# 1. Introduction

A neural network is basically a model structure and an algorithm for fitting the model to some given data. The *network approach* to modelling a plant uses a generic nonlinearity and allows all the parameters to be adjusted. In this way it can deal with a wide range of nonlinearities. *Learning* is the procedure of training a neural network to represent the dynamics of a plant, for instance in accordance with Fig. 1. The neural network is placed in parallel with the plant and the error $e$ between the output of the system and the network outputs, the *prediction error*, is used as the training signal. Neural networks have a potential for intelligent control systems because they can learn and adapt, they can approximate nonlinear functions, they are suited for parallel and distributed processing, and they naturally model multivariable systems. If a physical model is unavailable or too expensive to develop, a neural network model might be an alternative.

Networks are also used for *classification.* An example of an industrial application concerns acoustic quality control (Meier, Weber & Zimmermann, 1994). A factory produces ceramic tiles and an experienced operator is able to tell a bad tile from a good one by hitting it with a hammer; if there are cracks inside, it makes an unusual sound. In the quality control system the tiles are hit automatically and the sound is recorded with a microphone. A neural network (a so-called Kohonen network) tells the bad tiles from the good ones, with an acceptable rate of success, after being presented with a number of examples.

The objective here is to present the subject from a *control* engineer's viewpoint. Thus, there are two immediate application areas:

- Models of (industrial) processes, and
- controllers for (industrial) processes.

In the early 1940s McCulloch and Pitts studied the connection of several basic elements based on a model of a neuron, and Hebb studied the adaptation in neural systems. Rosenblatt devised in the late 50s the *Perceptron,* now widely used. Then in 1969 Minsky and Papert pointed to several limitations of the perceptron, and as a consequence the research in the field slowed down for lack of funding. The catalyst for today's level of research was a

series of results and algorithms published in 1986 by Rumelhart and his co-workers. In the 90s neural networks and fuzzy logic came together in neurofuzzy systems since both techniques are applied where there is uncertainty. There are now many real-world applications ranging from finance to aerospace.

There are many neural network architectures such as the perceptron, multilayer perceptrons, networks with feedback loops, self-organising systems, and dynamical networks, together with several different learning methods such as error-correction learning, competitive learning, supervised and unsupervised learning (see the textbook by Haykin, 1994). Neural network types and learning methods have been organised into a brief classification scheme, a *taxonomy* (Lippmann, 1987).

Neural networks have already been examined from a control engineer's viewpoint (Miller, Sutton & Werbos in Hunt, Sbarbaro, Zbikowski & Gawthrop, 1992). Neural networks can be used for system identification (forward modelling, inverse modelling) and for control, such as supervised control, direct inverse control, model reference control, internal model control, and predictive control (see the overview article by Hunt et al., 1992). Within the realm of modelling, identification, and control of nonlinear systems there are applications to pattern recognition, information processing, design, planning, and diagnosis (see the overview article by Fukuda & Shibata, 1992). Hybrid systems using neural networks, fuzzy sets, and artificial intelligence technologies exist, and these are surveyed also in that article. A systematic investigation of neural networks in control confirms that neural networks can be trained to control dynamic, nonlinear, multivariable, and noisy processes (see the PhD thesis by Sørensen, 1994). Somewhat related is Nørgaards investigation (1996) of their application to system identification, and he also proposes improvements to specific control designs. A comprehensive systematic classification of the control schemes proposed in the literature has been attempted by Agarwal (1997). His taxonomy is a tree with 'control schemes using neural networks' at the top node, broken down into two classes: 'neural network only as aid' and 'neural network as controller'. These are then further refined.

There are many commercial tools for building and using neural networks, either alone or together with fuzzy logic tools; for an overview, see the database CITE (MIT, 1995). Neural network computations are naturally expressed in matrix notation, and there are several toolboxes in the matrix language Matlab, for example the commercial neural network toolbox (Demuth & Beale, 1992), and a university developed toolbox for identification and control, downloadable from the World Wide Web (Nørgaard, NNSYSID with NNCTRL ).

In the wider perspective of *supervisory* control, there are other application areas, such as robotic vision, planning, diagnosis, quality control, and data analysis (data mining). The strategy in this lecture note is to aim at all these application areas, but only present the necessary and sufficient neural network material for understanding the basics.

## 2. The perceptron

Given a classification problem, the set of data $\mathbf{u}$ is to be classified into the classes $C_1$ and $C_2$. A neural network can learn from data and improve its performance through learning, and that is the ability we are interested in. In a *learning phase* we wish to present a subset

| 475 Hz | 557 Hz | Quality Ok? |
|---|---|---|
| 0.958 | 0.003 | Yes |
| 1.043 | 0.001 | Yes |
| 1.907 | 0.003 | Yes |
| 0.780 | 0.002 | Yes |
| 0.579 | 0.001 | Yes |
| 0.003 | 0.105 | No |
| 0.001 | 1.748 | No |
| 0.014 | 1.839 | No |
| 0.007 | 1.021 | No |
| 0.004 | 0.214 | No |

Table 1: Frequency intensities for ten tiles

of examples to the network in order to train it. In a following *generalisation phase* we wish the trained network makes the correct classification with data it has never seen before.

**Example 1 (tiles)** *Tiles are made from clay moulded into the right shape, brushed, glazed, and baked. Unfortunately, the baking may produce invisible cracks. Operators can detect the cracks by hitting the tiles with a hammer, and in an automated system the response is recorded with a microphone, filtered, Fourier transformed, and normalised. A small set of data (adapted from MIT, 1997) is given in TABLE* 1.

□

The *perceptron,* the simplest form of a neural network, is able to classify data into two classes. Basically it consists of a single *neuron* with a number of adjustable weights. The neuron is the fundamental processor of a neural network (Fig. 2). It has three basic elements:

1. A set of connecting *links* (or *synapses*); each link carries a *weight* (or *gain*) $w_0, w_1, w_2$.
2. A *summation* (or *adder*) sums the input signals after they are multiplied by their respective weights.
3. An *activation function* $f(x)$ limits the output of the neuron. Typically the output is limited to the interval $[0, 1]$ or alternatively $[-1, 1]$.

The summation in the neuron also includes an *offset* $w_0$ for lowering or raising the net input to the activation function. Mathematically the input to the neuron is represented by a vector $\mathbf{u} = (1, u_1, u_2, \ldots, u_n)^T$, and the output is a scalar $y = f(x)$. The weights of the connections are represented by the vector $\mathbf{w} = (w_0, w_1, \ldots, w_n)^T$ where $w_0$ is the offset. The output is calculated as

$$y = \mathrm{f}(\mathbf{w}^T\mathbf{u}) \tag{1}$$

Fig. 2a is a perceptron with two inputs and an offset. With a hard limiter as activation function (b), the neuron produces an output equal to $+1$ or $-1$ that we can associate with $\mathrm{C}_1$ and $\mathrm{C}_2$, respectively.

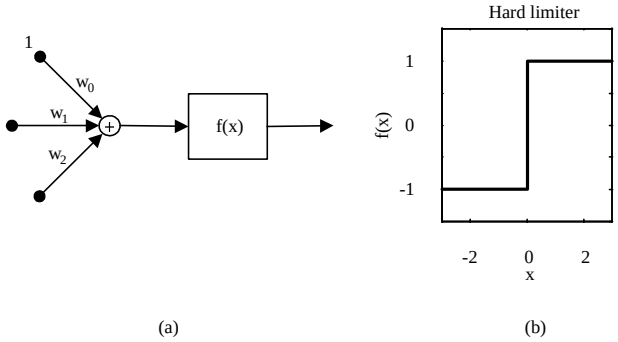


Figure 2: Perceptron consisting of a neuron (a) with an offset $w_0$ and an activation function $f(x)$, which is a hard limiter (b).

## 2.1 Perceptron training

The weights **w** are adjusted using an adaptive learning rule. One such learning rule is the *perceptron convergence algorithm*. If the two classes $C_1$ and $C_2$ are *linearly separable* (i.e., they lie on opposite sides of a straight line or, in general, a hyperplane), then there exists a weight vector **w,** with the properties

$$\begin{aligned} \mathbf{w}^T\mathbf{u} &\geq 0 \text{ for every input vector } \mathbf{u} \text{ belonging to class } C_1 \\ \mathbf{w}^T\mathbf{u} &< 0 \text{ for every input vector } \mathbf{u} \text{ belonging to class } C_2 \end{aligned} \tag{2}$$

Assuming, to be general, that the perceptron has $m$ inputs, then the equation $\mathbf{w}^T\mathbf{u} = 0$, in an $m$ dimensional space with coordinates $u_1, u_2, \ldots, u_m$, defines a hyperplane as the switching surface between the two different classes of input. The training process adjusts the weights **w** to satisfy the two inequalities (2). A training set consists of, say, $K$ samples of the input vector **u,** together with each sample's class membership (0 or 1). A presentation of the complete training set to the multilayer perceptron is called an *epoch.* The learning is continued epoch by epoch until the weights stabilise.

**Algorithm** The core of the perceptron convergence algorithm for adapting the weights of the elementary perceptron has two steps.

(a) If the $k$th member of the training set, $\mathbf{u}_k$, $(k = 1, 2 \ldots, K)$ is correctly classified by the weight vector $\mathbf{w}_k$ computed at the $k$th iteration of the algorithm, no correction is made to $\mathbf{w}_k$ as shown by

$$\mathbf{w}_{k+1} = \mathbf{w}_k \quad \text{if } \mathbf{w}_k^T\mathbf{u}_k \geq 0 \text{ and } \mathbf{u}_k \text{ belongs to class } C_1 \tag{3}$$

and

$$\mathbf{w}_{k+1} = \mathbf{w}_k \quad \text{if } \mathbf{w}_k^T\mathbf{u}_k < 0 \text{ and } \mathbf{u}_k \text{ belongs to class } C_2 \tag{4}$$

(b) Otherwise the perceptron weights are updated according to the rule

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta_k\mathbf{u}_k \quad \text{if } \mathbf{w}_k^T\mathbf{u}_k \geq 0 \text{ and } \mathbf{u}_k \text{ belongs to class } C_2 \tag{5}$$

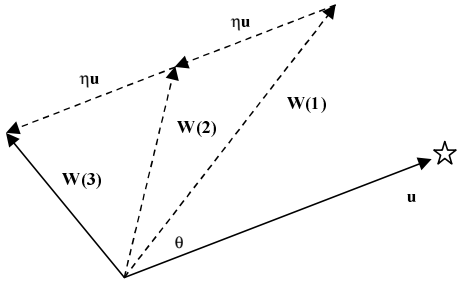


Figure 3: After two updates $\theta$ is larger than 90 degrees, and $\mathbf{w}^T\mathbf{u} = |\mathbf{w}|\,|\mathbf{u}|\cos(\theta)$ changes sign.

and

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \eta_k \mathbf{u}_k \quad \text{if } \mathbf{w}_k^T \mathbf{u}_k < 0 \text{ and } \mathbf{u}_k \text{ belongs to class } \mathrm{C}_1 \tag{6}$$

□

Notice the interchange of the class numbers from step 1 to step 2. The learning-rate $\eta_k$ controls the adjustment applied to the weight vector at iteration $k$. If $\eta$ is a constant, independent of the iteration number $k$, we have a *fixed increment* adaptation rule for the perceptron. The algorithm has been proved to converge (Haykin, 1994; Lippmann, 1987).

The perceptron learning rule is illustrated in Fig. 3. The input vector **u** points at some point in the $m$-dimensional space marked by a star. The update rule is based on the definition of the dot-product of two vectors, which relates the angle $\theta$ between the input vector and the weight vector w,

$$\mathbf{w}^T\mathbf{u} = |\mathbf{w}|\,|\mathbf{u}|\cos(\theta)$$

The figure shows a situation where the weight vector $\mathbf{w}(1)$ needs to be changed. The angle $\theta$ is less than 90 degrees, $\cos(\theta)$ is positive, and the update rule changes the weight vector into $\mathbf{w}(2)$ by the amount $\eta\mathbf{u}$ in the direction opposite of **u.** The weight vector turned, but not enough, so another pass is necessary to bring it to $\mathbf{w}(3)$. Now the angle is larger than 90 degrees, and the sign of $\mathbf{w}^T\mathbf{u}$ is correct. The vector **w** is orthogonal to the hyperplane that separates the classes (Fig. 4). When **w** turns, the hyperplane turns with it, until the hyperplane separates the classes correctly.

**Example 2 (tiles)** *We will train the perceptron in Fig.* 2 *to detect bad tiles. The inputs to the neuron are the two first columns in TABLE* 1*. Let us associate the good tiles with class* $C_1$ *and the bad ones with* $C_2$*. The first test input to the perceptron, including a* 1 *for the offset weight, is then*

$$\mathbf{u}^T = (1, 0.958, 0.003) \tag{7}$$

*This corresponds to a tile from* $C_1$*. The weights, including the offset weight, have to be initialised, and let us arbitrarily start with*

$$\mathbf{w}^T = (0.5, 0.5, 05) \tag{8}$$

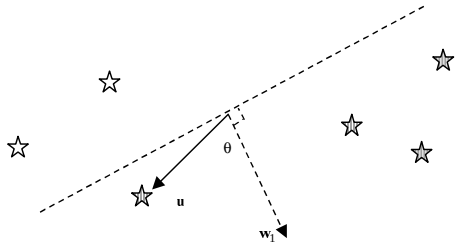


Figure 4: The weight vector defines the line (hyperplane), that separates the white class from the shaded class.

*The learning rate is fixed at*

$$\eta = 0.5 \tag{9}$$

*According to the perceptron convergence algorithm, we have to test the product*

$$\begin{aligned} \mathbf{w}^T\mathbf{u} &= (1, 0.958, 0.003)(0.5, 0.5, 05)^T & (10)\\ &= 0.981 & (11)\end{aligned}$$

*The sign is positive corresponding to the case in (3); therefore we leave the weights of the perceptron as they are. We repeat the procedure with the next set of inputs, and the next, and so on. The results from a hand calculation of two passes through the whole set of data are collected in TABLE* 2*. There are no updates to the weights before we reach the first bad tile, then we apply (5), and the weights change.*

*After the first pass it looks like the perceptron can recognise the bad tiles, but as the weights have changed since the initial run, we have to go back and check the good tiles again. In the second pass (TABLE* 2*, second pass) the weights are updated in the first iteration, but after that there are no further changes. A third pass (not shown) will show that the adaptation has stopped.*

*The perceptron is indeed able to distinguish between good and bad tiles. Figure* 5 *shows a logarithmic plot of the data and a separating line between the two classes. The final weights are*

$$\mathbf{w}^T = (0, 0.977, -0.425) \tag{12}$$

*That is, no offset, and the line where the sign switches is defined by*

$$0.977u_1 - 0.425u_2 = 0 \tag{13}$$

*This is equivalent to the straight line* $u_2 = (0.977/0.425)u_1$ *drawn in the figure. With a hard limiter (*$\mathrm{signum}$ *function) as the activation function the perceptron produces an output according to*

$$y = \mathrm{sgn}(\mathbf{w}^T\mathbf{u}) \tag{14}$$

*Any new data, drawn from beyond the training set, but above the line will result in* $y = -1$ *(bad) and any data below the line will result in* $y = +1$ *(good).*

□

| $\mathbf{u}^T$ | $\mathbf{w}^T$ | OK? | $\mathbf{w}^T\mathbf{u}$ | Equation | Updated $\mathbf{w}^T$ |
|---|---|---|---|---|---|
| (1,0.958,0.003) | (0.5,0.5,0.5) | 1 | 0.981 | (3) | (0.5,0.5,0.5) |
| (1,1.043,0.001) | (0.5,0.5,0.5) | 1 | 1.022 | (3) | (0.5,0.5,0.5) |
| (1,1.907,0.003) | (0.5,0.5,0.5) | 1 | 1.455 | (3) | (0.5,0.5,0.5) |
| (1,0.780,0.002) | (0.5,0.5,0.5) | 1 | 0.891 | (3) | (0.5,0.5,0.5) |
| (1,0.579,0.001) | (0.5,0.5,0.5) | 1 | 0.790 | (3) | (0.5,0.5,0.5) |
| (1,0.003,0.105) | (0.5,0.5,0.5) | 0 | 0.554 | (5) | (0,0.499,0.448) |
| (1,0.001,1.748) | (0,0.499,0.448) | 0 | 0.783 | (5) | (-0.5,0.498,-0.427) |
| (1,0.014,1.839) | (-0.5,0.498,-0.427) | 0 | -1.277 | (3) | (-0.5,0.498,-0.427) |
| (1,0.007,1.021) | (-0.5,0.498,-0.427) | 0 | -0.932 | (3) | (-0.5,0.498,-0.427) |
| (1,0.004,0.214) | (-0.5,0.498,-0.427) | 0 | -0.589 | (3) | (-0.5,0.498,-0.427) |
| 2nd pass | | | | | |
| (1,0.958,0.003) | (-0.5,0.498,-0.427) | 1 | -0.024 | (6) | (0,0.977,-0.425) |
| (1,1.043,0.001) | (0,0.977,-0.425) | 1 | 1.019 | (3) | (0,0.977,-0.425) |
| (1,1.907,0.003) | (0,0.977,-0.425) | 1 | 1.862 | (3) | (0,0.977,-0.425) |
| (1,0.780,0.002) | (0,0.977,-0.425) | 1 | 0.761 | (3) | (0,0.977,-0.425) |
| (1,0.579,0.001) | (0,0.977,-0.425) | 1 | 0.565 | (3) | (0,0.977,-0.425) |
| (1,0.003,0.105) | (0,0.977,-0.425) | 0 | -0.042 | (3) | (0,0.977,-0.425) |
| (1,0.001,1.748) | (0,0.977,-0.425) | 0 | -0.742 | (3) | (0,0.977,-0.425) |
| (1,0.014,1.839) | (0,0.977,-0.425) | 0 | -0.768 | (3) | (0,0.977,-0.425) |
| (1,0.007,1.021) | (0,0.977,-0.425) | 0 | -0.427 | (3) | (0,0.977,-0.425) |
| (1,0.004,0.214) | (0,0.977,-0.425) | 0 | -0.087 | (3) | (0,0.977,-0.425) |

Table 2: After two passes the the perceptron algorithm converges.

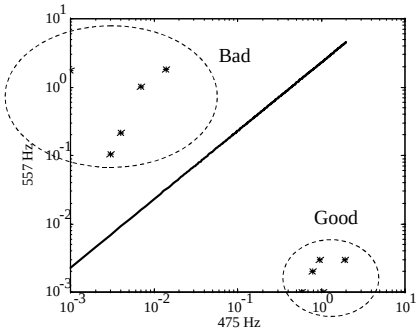


Figure 5: Classification of tiles in the input space (logarithmic axes).

## 2.2 Single layer perceptron

Given a problem which calls for more than two classes, several perceptrons can be combined into a network. The simplest form of a *layered* network just has an input layer of source nodes that connect to an output layer of neurons, but not vice versa. Fig. 6 shows a single layer perceptron with five nodes capable of recognising three linearly separable classes (three output nodes) by means of two features (two input nodes). Each output neuron defines a weight vector which in turn defines a hyperplane. The hyperplanes separate the classes as in Fig. 7.

The activation function could have various shapes depending on the application; Fig. 8 shows six different types. The mathematical expressions for the three in the bottom row are, respectively

$$\mathrm{f}(x) = \frac{1}{1+\exp(-ax)} \tag{15}$$

$$\mathrm{f}(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)} \tag{16}$$

$$\mathrm{f}(x) = \exp(-x^2) \tag{17}$$

Here (15) is a *logistic* function, an example of the widely used *sigmoid* function; in general a sigmoid is an $s-$ shaped function, strictly increasing and asymptotic. The parameter $a$ is used to adjust the slope; in the limit as $a$ approaches infinity, the sigmoid becomes a step. Equation (16) is a *hyperbolic tangent*; it is symmetric about the origin, which can be convenient. The third function (17) is a *Gaussian* function.

## 2.3 Gradient descent learning

If we introduce the *desired* response $d$ of the network,

$$d = \left\{ \begin{array}{ll} +1 & \text{if } \mathbf{u} \text{ belongs to class } \mathrm{C}_1 \\ -1 & \text{if } \mathbf{u} \text{ belongs to class } \mathrm{C}_2 \end{array} \right\} \tag{18}$$

the learning rules (3)-(6) can be expressed in a more convenient way. They can be summed up nicely in the form of an *error-correction* learning rule, the so-called *delta* rule, as shown by

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \Delta\mathbf{w}_k \tag{19}$$

$$= \mathbf{w}_k + \eta_k(d_k - y_k)\mathbf{u}_k \tag{20}$$

The difference $d_k - y_k$ plays the role of an error signal and $k$ corresponds to the current learning sample. The learning rate parameter $\eta$ is a positive constant limited to the range $0 < \eta \leq 1$. The learning rule increases the output $y$ by increasing $\mathbf{w}$ when the error $e = d-y$ is positive; therefore $w_i$ increases if $u_i$ is positive, and it decreases if $u_i$ is negative (and similarly when the error is negative). Features of the delta rule are as follows:

- It is simple,
- learning can be performed locally at each neuron, and
- weights are updated on-line after presentation of each pattern.

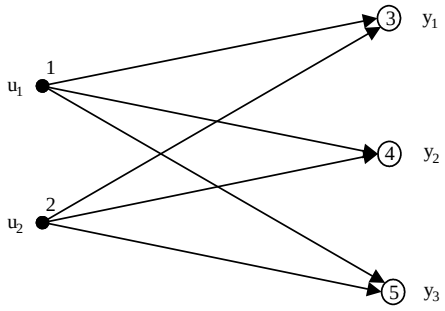


Figure 6: Single layer perceptron network with three output neurons.

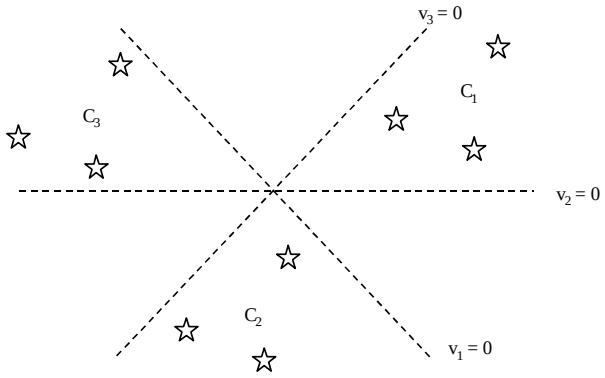


Figure 7: Three linearly separable classes.

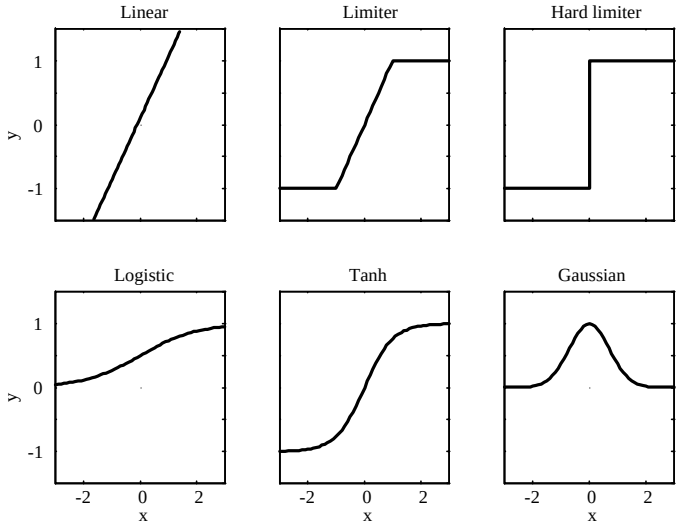


Figure 8: Examples of six activation functions.

The ultimate purpose of error-correction learning is to minimise a cost function based on the error signal $e$. Then the response of each output neuron approaches the target response for that neuron in some statistical sense. Indeed, once a cost function is selected, the learning is strictly an optimisation problem.

A common cost function is the sum of the squared errors,

$$\mathcal{E}(\mathbf{w}) = \frac{1}{2}\sum_{r} e_r^2 \tag{21}$$

The summation is over all the output neurons of the network (cf. index $r$). The network is trained by minimising $\mathcal{E}$ with respect to the weights, and this leads to the *gradient descent* method. The factor $1/2$ is used to simplify differentiation when minimising $\mathcal{E}$. A plot of the cost function $\mathcal{E}$ versus the weights is a multidimensional *error-surface*. Depending on the type of activation functions used in the network we may encounter two situations:

1. *The network has entirely linear neurons*. The error surface is a quadratic function of the weights, and it is bowl-shaped with a unique minimum point.
2. *The network has nonlinear neurons*. The surface has a global minimum (perhaps multiple) as well as local minima.

In both cases, the objective of the learning is to start from an arbitrary point on the surface, determined by the initial weights and the initial inputs, and move toward a global minimum in a step-by-step fashion. This is feasible in the first case, whereas in the second case the algorithm may get trapped in a local minimum, never reaching a global minimum.

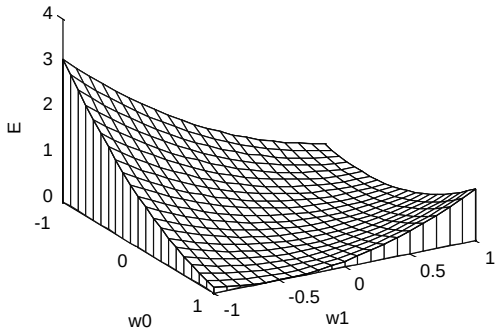


Figure 9: Example of error surface for a simple perceptron.

**Example 3 (error surface)** *To see how the error surface is formed we will examine a simple perceptron with one neuron* $(r = 1)$*, one input* $(m = 1)$*, an offset, and a linear activation function. The network output is*

$$y = \mathrm{f}(\mathbf{w}^T\mathbf{u}) = \mathbf{w}^T\mathbf{u} = (w_0, w_1)(1, u)^T = w_0 + w_1 u \tag{22}$$

*Given a particular input* $u$ *and the corresponding desired output* $d$*, we can plot the error surface. Assume*

$$u = 1, d = 0.5 \tag{23}$$

*then the error function becomes*

$$\begin{aligned} \mathcal{E}(\mathbf{w}) &= \frac{1}{2}e^2 && (24)\\ &= \frac{1}{2}(d-y)^2 && (25)\\ &= \frac{1}{2}\left(d-(w_0+w_1u)\right)^2 && (26)\\ &= \frac{1}{2}\left(0.5-(w_0+w_1)\right)^2 && (27) \end{aligned}$$

*This is a parabolic bowl, as the plot indicates in Fig.* 9*.*

*In case there are* $K$ *examples in the training set, then* $u$*,* $y$*, and* $d$ *become (row) vectors of length* $K$*, and the squared errors are summed for each instance of* $w_0$ *and* $w_1$

$$\mathcal{E}(\mathbf{w}) = \frac{1}{2}\sum_K \left(\mathbf{d}^T - \left(w_0 + w_1 * \mathbf{u}^T\right)\right)^2 \tag{28}$$

*The squaring operation is element-by-element and we have a vector under the summation symbol. The summation is over the elements of that vector. The result is a scalar for each combination of the scalars* $w_0$ *and* $w_1$*.*

□

To sum up, in order to make $\mathcal{E}(\mathbf{w})$ small, the idea is to change the weights **w** in the

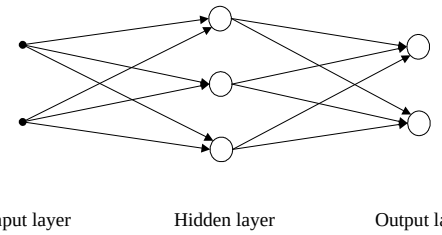


Figure 10: Fully connected multi-layer perceptron.

direction of the negative gradient of $\mathcal{E}$, that is

$$\Delta \mathbf{w} = -\eta \frac{\partial \mathcal{E}}{d\mathbf{w}} = -\eta e \frac{\partial e}{d\mathbf{w}} \tag{29}$$

Here $e$ has been substituted for $\sum_r e_r^2$. The partial derivatives $\partial e / d\mathbf{w}$ tells how much the error is influenced by changes in the weights.

**Example 4** *There are alternatives to the cost function (21). If it is chosen to be*

$$\mathcal{E}(\mathbf{w}) = |e|$$

*the gradient method gives*

$$\Delta \mathbf{w} = -\eta \frac{\partial e}{d\mathbf{w}} sign(e)$$

□

## 2.4 Multi-layer perceptron

The single-layer perceptron can only classify linearly separable problems. For non-separable problems it is necessary to use more layers. A *multilayer* (feedforward) network has one or more *hidden* layers whose neurons are called *hidden neurons*. The graph in Fig. 10 illustrates a multilayer network with one hidden layer. The network is *fully connected,* because every node in a layer is connected to all nodes in the next layer. If some of the links are missing, the network is *partially connected.* When we say that a network has $n$ layers, we only count the hidden layers and the output layer; the input layer of source nodes does not count, because the nodes do not perform any computations. A single layer network thus refers to a network with just an output layer.

**Example 5 (XOR)** *In Fig.* 11*, left, the points marked by an 'x' belong to class $C_1$ and the points marked by an 'o' belong to $C_2$. The relation between $(u_1, u_2)$ versus the class*

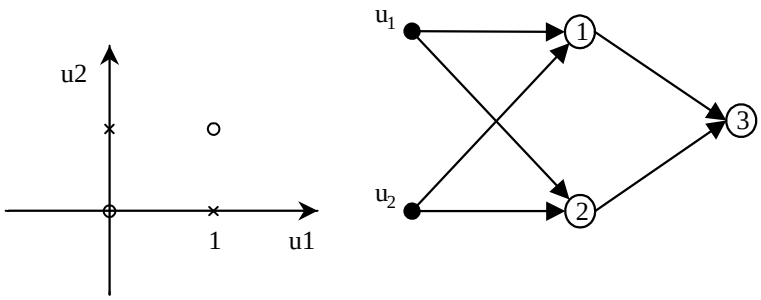


Figure 11: Two classes 'x' and 'o' that cannot be separated by a line (left) require a network with more than one layer (right).

*membership $y$ can be described in the table*

| $u_1$ | $u_2$ | $y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

(30)

*The table also expresses the logical* exclusive-or *(XOR) operation; it was used many years ago to demonstrate that the (single layer) perceptron was unable to classify a very simple set of data.*

*In symbols, a two-input perceptron with one neuron, an offset, and a linear activation function has the following function*

$$y = \mathrm{f}(\mathbf{w}^T\mathbf{u}) = \mathbf{w}^T\mathbf{u} = (w_0, w_1, w_2)(1, u_1, u_2)^T = w_0 + w_1u_1 + w_2u_2 \tag{31}$$

*It is required to satisfy the following four inequalities:*

$$0 * w_1 + 0 * w_2 + w_0 \leq 0 \Rightarrow w_0 \leq 0 \tag{32}$$
$$0 * w_1 + 1 * w_2 + w_0 > 0 \Rightarrow w_0 > -w_2 \tag{33}$$
$$1 * w_1 + 0 * w_2 + w_0 > 0 \Rightarrow w_0 > -w_1 \tag{34}$$
$$1 * w_1 + 1 * w_2 + w_0 \leq 0 \Rightarrow w_0 \leq -w_1 - w_2 \tag{35}$$

*Using (32) in (33)-(35) implies*

$$w_2 > 0 \tag{36}$$
$$w_1 > 0 \tag{37}$$
$$w_1 + w_2 \leq 0 \tag{38}$$

*It is of course impossible to satisfy (36)-(38) all at the same time.*

□

**Example 6** *It is possible, nevertheless, to solve the problem with the two layer perceptron*

*in Fig.* 11*, right, if neuron 1 in the hidden layer has the weights* $\mathbf{w}_1 = (-0.5, 1, -1)$*, neuron 2 in the hidden layer* $\mathbf{w}_2 = (0.5, 1, -1)$ *, and the output neuron* $\mathbf{w}_3 = (0.5, 1, -1)$ *. Assuming a hard limiter activation function in each neuron, and representing the bits* 0 *and* 1 *by* $-1$ *respectively* $+1$*, then the overall function of the network is*

$$y_3 = sgn\left(\mathbf{w}_3^T (1, y_1, y_2)^T\right) = sgn(0.5 + y_1 - y_2) \quad (39)$$

$$y_1 = sgn\left(\mathbf{w}_1^T (1, u_1, u_2)^T\right) = sgn(-0.5 + u_1 - u_2) \quad (40)$$

$$y_2 = sgn\left(\mathbf{w}_2^T (1, u_1, u_2)^T\right) = sgn(0.5 + u_1 - u_2) \quad (41)$$

*By insertion, the input-output relation is*

$$y_3 = sgn(0.5 + (sgn(-0.5 + u_1 - u_2)) - sgn(0.5 + u_1 - u_2)) \quad (42)$$

*With* $\mathbf{u}_1 = (-1, -1, 1, 1)^T$ *and* $\mathbf{u}_2 = (-1, 1 - 1, 1)^T$ *the output becomes* $\mathbf{y} = (-1, 1, 1, -1)^T$ *as desired, cf. (30).*

□

## 2.5 Back-propagation

The *error back-propagation* algorithm is a popular learning rule for multilayer perceptrons. It is based on the delta rule, and it uses the squared error measure (21) for output nodes. A perceptron weight $w_{ji}$, the weight on the connection from neuron $i$ to neuron $j$, is updated according to the *generalised delta rule*

$$w_{ji} = w_{ji} + \Delta w_{ji} = w_{ji} - \eta \frac{\partial \mathcal{E}}{\partial w_{ji}} \quad (43)$$

To make the notation clearer the index of the training instance $k$ has been omitted from the expression; it is understood that the equation is recursive such that the $w_{ji}$ on the left hand side is the new weight, while the $w_{ji}$ on the right hand side is the old weight. In the generalised delta rule the correction to the weight is proportional to the gradient of the error or, in other words, to the sensitivity of the error to changes in the weight. To apply the algorithm two passes of computation are necessary, a *forward pass* and a *backward pass*.

In the forward pass the weights remain unchanged. The forward pass begins at the first hidden layer by presenting it with the input vector, and terminates at the output layer by computing the error signal for each output neuron. The backward pass starts at the output layer by passing the error signals backwards through the network, layer by layer.

For the backward pass, assume that neuron $j$ is an output neuron. The derivative may be expressed using the chain rule,

$$\frac{\partial \mathcal{E}}{\partial w_{ji}} = \frac{\partial \mathcal{E}}{\partial e_j} \frac{\partial e_j}{\partial y_j} \frac{\partial y_j}{\partial v_j} \frac{\partial v_j}{\partial w_{ji}} = e_j * (-1) * \mathrm{f}_j' * y_i \quad (44)$$

Here $e_j$ is the error of neuron $j$, $y_j$ is the output of the neuron, and $v_j$ is the internal input to neuron $j$ after summation, but before the activation function, i.e., $y_j = \mathrm{f}(v_j)$. The $\mathrm{f}'$

notation means differentiation of $\mathrm{f}$ with respect to its argument. Insertion into (43) yields

$$\Delta w_{ji} = \eta e_j \mathrm{f}_j' y_i \quad j \text{ is an output neuron} \tag{45}$$

The error $e_j$ of neuron $j$ is associated with a desired output $d_j$, and it is simply $e_j = d_j - y_j$.

For the hidden node $i$, we have to *back-propagate* the error recursively, since it occurs in the two first partial derivatives in (44). But there is a problem, in that the error is unknown for a hidden node. We will instead compute directly

$$\frac{\partial \mathcal{E}}{\partial e_i} \frac{\partial e_i}{\partial y_i} = \frac{\partial \mathcal{E}}{\partial y_i} \tag{46}$$

The partial derivative of $\mathcal{E}$ with respect to the output of hidden neuron $i$, connected directly to output neuron $j$, is

$$\begin{aligned} \frac{\partial \mathcal{E}}{\partial y_i} &= \sum_j e_j \frac{\partial e_j}{\partial y_i} && (47) \\ &= \sum_j e_j \frac{\partial e_j}{\partial v_j} \frac{\partial v_j}{\partial y_i} && (48) \\ &= \sum_j e_j \frac{\partial \left(d_j - \mathrm{f}_j\left(v_j\right)\right)}{\partial v_j} \frac{\partial v_j}{\partial y_i} && (49) \\ &= \sum_j e_j \left[-\mathrm{f}_j'\left(\cdot\right)\right] w_{ji} && (50) \end{aligned}$$

In this case the update rule becomes

$$\Delta w_{ih} = \eta \sum_j e_j \mathrm{f}_j'\left(\cdot\right) w_{ji} \mathrm{f}_i'\left(\cdot\right) y_h \tag{51}$$

Here $i$ is a hidden neuron, $h$ is an immediate predecessor of $i$, and $j$ is an output neuron. The factor $\mathrm{f}_i'\left(\cdot\right)$ depends solely on the activation function of hidden neuron $i$. The summation over $j$ requires knowledge of the error signals $e_j$ for all the output neurons. The term $w_{ji}$ consists of the weights associated with the output connections. Finally, $y_h$ is the input to neuron $i$, and it stems from the partial derivative $\partial v_j / \partial y_i$. For a neuron $g$ in the next layer, going towards the input layer, the update rule is applied recursively.

In summary, the correction to the weight $w_{qp}$ on the connection from neuron $p$ to neuron $q$ is defined by

$$\Delta w_{qp} = \eta \delta_q y_p \tag{52}$$

Here $\eta$ is the learning rate parameter, $\delta_q$ is called the *local gradient,* cp. (29) or (43), and $y_p$ is the input signal to neuron $q$. The local gradient is computed recursively for each neuron, and it depends on whether the neuron is an output neuron or a hidden neuron:

1. If neuron $q$ is an output neuron, then

$$\delta_q = e_q \mathrm{f}_q'\left(\cdot\right) \tag{53}$$

Both factors in the product are associated with neuron $q$.

2. If neuron $q$ is a hidden node, then

$$\delta_q = \sum_r \delta_r w_{rq} \mathrm{f}_q'(\cdot) \tag{54}$$

The summation is a weighted sum of $\delta_r$'s of the immediate successor neurons.

The weights on the connections feeding the output layer are updated using the delta rule (52), where the local gradient $\delta_q$ is as in (53). Given the $\delta$'s for the neurons of the output layer, we next use (54) to compute the $\delta$'s for all the neurons in the next layer and the changes to all the weights of the connections feeding it. To compute the $\delta$ for each neuron, the activation function $\mathrm{f}(\cdot)$ of that neuron must be differentiable.

**Algorithm** (after Lippmann, 1987) The back-propagation algorithm has five steps.

(a) *Initialise weights*. Set all weights to small random values.

(b) *Present inputs and desired outputs (training pairs)*. Present an input vector **u** and specify the desired outputs **d**. If the net is used as a classifier then all desired outputs are typically set to zero, except one (set to 1). The input could be new on each trial, or samples from a training set could be presented cyclically until weights stabilise.

(c) *Calculate actual outputs*. Calculate the outputs by successive use of $\mathbf{y} = \mathbf{f}(\mathbf{w}^T\mathbf{u})$ where **f** is a vector of activation functions.

(d) *Adapt weights*. Start at the output neurons and work back to the first hidden layer. Adjust weights by

$$w_{ji} = w_{ji} + \eta \delta_j y_i \tag{55}$$

In this equation $w_{ji}$ is the weight from hidden neuron $i$ (or an input node) to neuron $j$, $y_i$ is the output of neuron $i$ (or an input), $\eta$ is the learning rate parameter, and $\delta_j$ is the gradient; if neuron $j$ is an output neuron then it is defined by (53), and if neuron $j$ is hidden then it is defined by (54). Convergence is sometimes faster if a momentum term is added according to (78).

(e) *Go to step 2*.

□

**Example 7 (local gradient)** *The logistic function* $\mathrm{f}(x) = 1/(1+\exp(-x))$ *is differentiable with the derivative* $\mathrm{f}'(x) = \exp(-x)/(1+\exp(-x))^2$. *We can eliminate the exponential term and express it as*

$$f'(x) = f(x)\,[1 - f(x)] \tag{56}$$

*For a neuron* $j$ *in the output layer, we may then express the local gradient as*

$$\delta_j = e_j f_j'(\cdot) = (d_j - y_j)\, y_j\, (1 - y_j) \tag{57}$$

*For a hidden neuron* $i$ *the local gradient is*

$$\delta_i = f_i'(\cdot) \sum_j \delta_j w_{ji} = y_i\,(1 - y_i) \sum_j \delta_j w_{ji} \tag{58}$$

□

**Example 8 (backpropagation)** *(after Demuth & Beale, 1992) To study back-propagation*

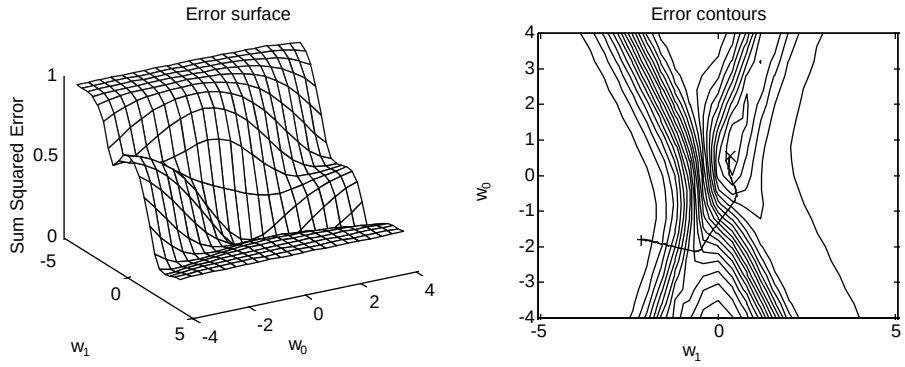


Figure 12: Nonlinear error surface for back-propagation.

*we will consider a simple example with the training set*

$$\begin{array}{rr} u & y \\ \hline -3 & 0.4 \\ 2 & 0.8 \end{array} \tag{59}$$

*Thus the network will have one input, there will be no hidden layer, and the output layer will have one output neuron. To make the example more interesting we choose a nonlinear activation function (logistic function). The overall function is then*

$$y = \mathrm{f}\left((w_0, w_1) \cdot (1, u)^T\right) = \mathrm{f}\left(w_0 + w_1 u\right) \tag{60}$$

*The error is*

$$\mathcal{E} = \frac{1}{2}\sum e^2 = \frac{1}{2}\sum \left(d - y\right)^2 = \frac{1}{2}\sum \left((0.4, 0.8) - \mathrm{f}\left(w_0 + w_1\left(-3, 2\right)\right)\right)^2 \tag{61}$$

*With some help from the neural network toolbox (Demuth & Beale, 1992) we can plot the error surface, see Fig.* 12 *(the toolbox omits the factor* $1/2$ *in (61)).*

*As the learning progresses, the weights are adjusted to move the network down the gradient. The error contour graph on the right side of Fig.* 12 *shows how the network moved from its original values to a solution. Notice that it more or less chooses the steepest path.*

*The network error per epoch is saved throughout the training and the error per epoch is plotted in Fig.* 13*. The error decreases throughout the training and the training stops after about 70 epochs when the error drops below the preset limit of* $0.001$*. At this point the final values of the weights are* $(w_0, w_1) = (0.5497, 0.3350)$ *.*

□

## 2.6 A general model

A standard, matrix oriented model for neural networks has been developed (Hunt, Sbar-

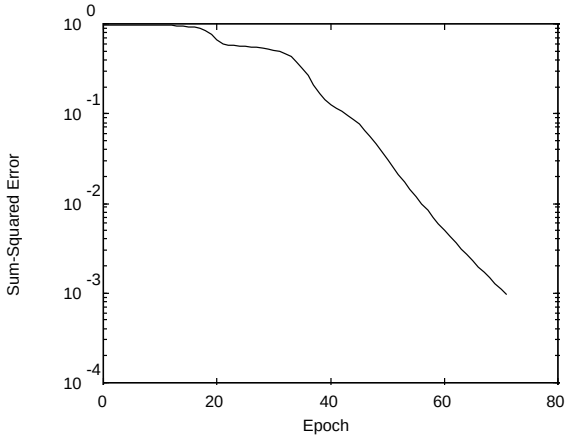


Figure 13: Network error.

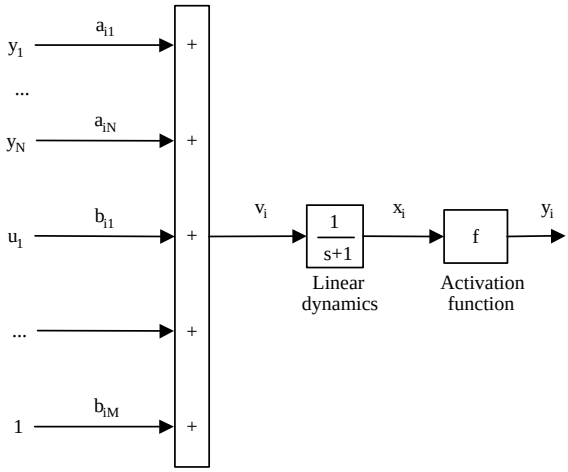


Figure 14: Generalised neuron model.

baro, Zbikowski and Gawthrop, 1992), and many well-known structures fall within that model. The basic processing element is considered to have three elements (see Fig. 14): a weighted summer, a linear dynamic single-input, single-output function, and a non-dynamic nonlinear function.

- The *weighted summer* is described by

$$v_i(t) = \sum_{j=1}^{N} a_{ij} f(x_j)(t) + \sum_{k=1}^{M} b_{ik} u_k(t), \tag{62}$$

  giving a weighted sum $v_i$ in terms of the outputs of all elements $f(x_j)$, external inputs $u_k$, and corresponding weights $a_{ij}$ and $b_{ik}$ with offsets included in $b_{ik}$. To simplify the notation the time index will be omitted in the following, and the equation may be written in matrix notation as:

$$\mathbf{v} = \mathbf{Af(x)} + \mathbf{Bu} \tag{63}$$

  where **A** is an $N \times N$ matrix of weights $a_{ij}$, and **B** is an $N \times M$ matrix of weights $b_{ik}$. The neurons are enumerated consecutively from top to bottom, layer by layer in the forward direction. The offsets are incorporated with the inputs **u**, but it could be useful to represent them explicitly.
- The *linear dynamic function* has input $v_i$ and output $x_i$. In transfer function form it is described by

$$x_i(s) = \mathrm{H}(s) v_i(s) \tag{64}$$

  Five common choices are

$$H(s) = 1 \tag{65}$$

$$H(s) = \frac{1}{s} \tag{66}$$

$$H(s) = \frac{1}{1 + \tau s} \tag{67}$$

$$H(s) = \frac{1}{p_1 s + p_2} \tag{68}$$

$$H(s) = \exp(-s\tau) \tag{69}$$

  Clearly, the first three are special cases of the fourth.
- The *non-dynamic nonlinear function* returns the output

$$f(x_i) \tag{70}$$

  in terms of the input $x_i$. The function $f(\cdot)$ corresponds to the previously mentioned activation functions (see for example Fig. 8).

The three components of the neuron can be combined in various ways. For example, if the neurons are all non-dynamic $(\mathrm{H}(s) = 1)$ then an array of neurons can be written as the set of *algebraic* equations obtained by combining (62), (64), and (70):

$$\mathbf{x} = \mathbf{Af(x)} + \mathbf{Bu} \tag{71}$$

where **x** is an $N \times 1$ vector. If, on the other hand, each neuron has integrator dynamics

$H(s) = 1/s$, then an array of neurons can be written as the set of *differential* equations

$$\dot{\mathbf{x}} = \mathbf{Af(x)} + \mathbf{Bu} \tag{72}$$

Clearly, the solutions of (71) form possible steady state solutions of (72).

The behaviour of such a network depends on the interconnection matrix **A** and on the form of $\mathrm{H}(s)$ . We shall use the model now to give an alternative formulation of the back-propagation algorithm.

In a *static two-layer feedforward* network, $\mathrm{H}(s) = 1$. In order to study more closely how the calculations proceed, Fig. 15 shows a signal flow graph of a simple network with one input, two nodes, and one output. It is a two layer network with offsets. The arrows are weighted (sometimes nonlinearly) and the nodes are summation points. Generalising to any number of nodes,

$$\begin{aligned} \mathbf{x} &= \mathbf{Af(x)} + \mathbf{Bu} \\ \mathbf{y} &= \mathbf{Cf(x)} \end{aligned} \tag{73}$$

Keeping the two-layer structure, the **x**, **y**, and **u** vectors are partitioned according to the layers, and the forward pass of the backpropagation algorithm can be written in details as

$$\begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{A}_{21} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathrm{f}(\mathbf{x}_1) \\ \mathrm{f}(\mathbf{x}_2) \end{bmatrix} + \begin{bmatrix} \mathbf{B}_{11} & \mathbf{b}_{12} \\ \mathbf{0} & \mathbf{b}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ u_2 = 1 \end{bmatrix} \tag{74}$$

$$\mathbf{y} = \begin{bmatrix} \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathrm{f}(\mathbf{x}_1) \\ \mathrm{f}(\mathbf{x}_2) \end{bmatrix} \tag{75}$$

The subscripts denote the layers in the network, such that the hidden layer has subscript 1 and the output layer has subscript 2. The **A** and **B** matrices have a block structure and the matrix $\mathbf{A}_{21}$ holds the weights from layer 1 to layer 2; **I** is the identity matrix. The inputs are partitioned according to real inputs $\mathbf{u}_1$ and one additional input $u_2 = 1$ which is used for the offsets of all nodes. The input matrix $\mathbf{B}_{11}$ holds the weights from the input nodes $\mathbf{u}_1$ to the hidden layer, $\mathbf{b}_{12}$ is a vector of offset weights for the first layer, and $\mathbf{b}_{22}$ is for the second layer.

The forward pass proceeds in three iterations, after the vector **x** is initialised to **0**.

Iteration 1

$$\begin{aligned} \mathbf{x}_1 &= \mathbf{B}_{11}\mathbf{u}_1 + \mathbf{b}_{12} \\ \mathbf{x}_2 &= \mathbf{b}_{22} \end{aligned}$$

Iteration 2

$$\begin{aligned} \mathbf{x}_1 &= \mathbf{B}_{11}\mathbf{u}_1 + \mathbf{b}_{12} \\ \mathbf{x}_2 &= \mathbf{A}_{21} f\left(\mathbf{B}_{11}\mathbf{u}_1 + \mathbf{b}_{12}\right) + \mathbf{b}_{22} \end{aligned}$$

Iteration 3

$$\mathbf{y} = f\left(\mathbf{A}_{21} f\left(\mathbf{B}_{11}\mathbf{u}_1 + \mathbf{b}_{12}\right) + \mathbf{b}_{22}\right)$$

Each iteration progresses on level farther into the flow graph, and after three iterations, the outputs are reached. The calculation is straight forward to implement.

The backward pass is somewhat tricky, at least regarding the understanding of the difference between errors and gradients. Figure 16 is a flow graph of a backward pass in our

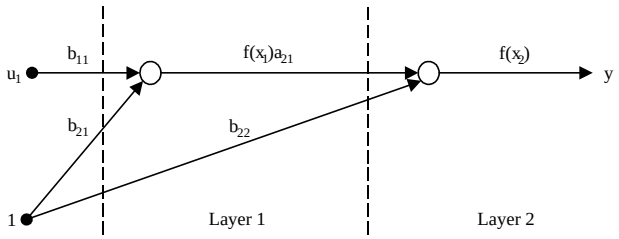


Figure 15: Signal flow graph of a forward pass in a two node network.

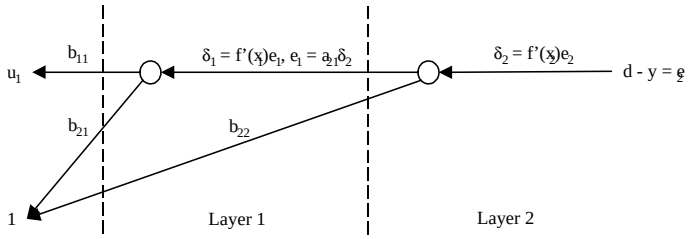


Figure 16: Signal flow graph of a backward pass showing the difference between errors and gradients.

simple network. The network is the same as in Fig. 15, except the arrows are reversed to emphasise the direction of the calculations. The input is now the error, which is the difference between the desired value and the network output, or $e_2 = d - y$. The error is multiplied by the derivative of the activation function in the current operating point to produce the first local gradient $\delta_2$. The error corresponding to node 1 is then $e_1 = a_{21}\delta_2$. The errors are used in the following matrix calculations similar to state variables in the state space model. Reversing the arrows corresponds to transposing the matrices. The backward pass can thus be expressed as,

$$\begin{aligned} \mathbf{e} &= \mathbf{A}^T \left[ f'(\mathbf{x})\, \mathbf{e} \right] + \mathbf{C}^T(\mathbf{d}\text{-}\mathbf{y}) \\ \mathbf{e}_u &= \mathbf{B}^T \left[ f'(\mathbf{x})\, \mathbf{e} \right] \end{aligned} \tag{76}$$

The expression in square brackets $[\cdot]$ is a product of two vectors, but it is element-to-element, rather than a vector dot product, such that $f'(\mathbf{x}(1))$ is multiplied by $\mathbf{e}(1)$, $f'(\mathbf{x}(2))$ by $\mathbf{e}(2)$, and so on, producing a column vector. The bottom system of equations is not used in practice, but included for completeness; it expresses the backpropagation of the error all the way to the inputs. Note that **f'(x)e** is the vector of gradients $\boldsymbol{\delta}$. For implementation purposes (76) can be written in more detail,

$$\begin{aligned} \begin{bmatrix} \mathbf{e}_1 \\ \mathbf{e}_2 \end{bmatrix} &= \begin{bmatrix} \mathbf{0} & \mathbf{A}_{21}^T \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} f'(\mathbf{x}_1)\, \mathbf{e}_1 \\ f'(\mathbf{x}_2)\, \mathbf{e}_2 \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \mathbf{I} \end{bmatrix} [\mathbf{d} - \mathbf{y}] \\ \mathbf{e}_u &= \begin{bmatrix} \mathbf{B}_{11}^T & \mathbf{0} \\ \mathbf{b}_{12}^T & \mathbf{b}_{22}^T \end{bmatrix} \begin{bmatrix} f'(\mathbf{x}_1)\, \mathbf{e}_1 \\ f'(\mathbf{x}_2)\, \mathbf{e}_2 \end{bmatrix} \end{aligned} \tag{77}$$

The backward pass also proceeds in three iterations. The vector **e** is initialised to **0**.

Iteration 1

$$\begin{aligned} \mathbf{e}_1 &= \mathbf{0} \\ \mathbf{e}_2 &= \mathbf{d} - \mathbf{y} \end{aligned}$$

Iteration 2

$$\begin{aligned} \mathbf{e}_1 &= \mathbf{A}_{21}^T f'(\mathbf{x}_2)\, (\mathbf{d} - \mathbf{y}) \\ \mathbf{e}_2 &= \mathbf{d} - \mathbf{y} \end{aligned}$$

Iteration 3

$$\begin{aligned} \mathbf{e}_{u1} &= \mathbf{B}_{11}^T f'(\mathbf{x}_1)\, \mathbf{A}_{21}^T f'(\mathbf{x}_2)\, (\mathbf{d} - \mathbf{y}) \\ \mathbf{e}_{u2} &= \mathbf{b}_{12}^T f'(\mathbf{x}_1)\, \mathbf{A}_{21}^T f'(\mathbf{x}_2)\, (\mathbf{d} - \mathbf{y}) + \mathbf{b}_{22}^T f'(\mathbf{x}_2)\, (\mathbf{d} - \mathbf{y}) \end{aligned}$$

Once the backward pass is over, the weights can be updated. Both matrices **A** and **B** have to be updated, according to the delta rule,

$$\begin{aligned} \mathbf{A} &= \mathbf{A} + \Delta\mathbf{A} \\ \mathbf{B} &= \mathbf{B} + \Delta\mathbf{B} \end{aligned}$$

Or, in more detail,

$$\mathbf{A} = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{A}_{21} & \mathbf{0} \end{bmatrix} + \eta \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \delta_2 f'(\mathbf{x}_1)^T & \mathbf{0} \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{b}_{12} \\ \mathbf{0} & \mathbf{b}_{22} \end{bmatrix} + \eta \begin{bmatrix} \delta_1 \mathbf{u}_1^T & \delta_1 \\ \mathbf{0} & \delta_2 \end{bmatrix}$$

The vectors $\boldsymbol{\delta}_1$ and $\boldsymbol{\delta}_2$ are column vectors, and the products $\delta_2 f'(\mathbf{x}_1)^T$ and $\delta_1 \mathbf{u}_1^T$ are outer products, i.e., the results are matrices the same shape as $\mathbf{A}_{21}$ and $\mathbf{B}_{11}$ respectively. The model is for a two-layer, fully connected network. It should be fairly easy to extend it to more layers than two.

# 3. Practical issues

It is attractive that neural networks can model nonlinear relationships. From the treatment above, it may even seem easy to obtain a model of an unknown plant. In practice, however, several difficulties jeopardise the development, such as deciding the necessary and sufficient number of nodes in a network. There are no rigid mathematics regarding such design choices, but this section provides a few rules of thumb.

## 3.1 Rate of learning

The smaller we make the learning parameter $\eta$ the smaller will the changes to the weights be, but then the learning takes longer. If $\eta$ is too large, the learning may become unstable. According to Sørensen (1994) it is normal practice to gradually decrease $\eta$ from $0.1$ to $0.001$.

The backpropagation algorithm may reach only a local minimum of the error function, and the search may be slow, especially near the minimum. A simple way to improve the situation, and yet avoid instability, is to modify the delta rule by including a *momentum* term

$$\Delta w_{ji} = \eta \delta y_i + \alpha \Delta w_{ji} \tag{78}$$

where $\alpha$ is usually a positive constant called the *momentum constant*. The effect is a low-pass filtering of the increments of the weight updates, thus reducing the risk of getting stuck in a local minimum. The following rules of thumb apply.

- For the learning to be stable, the momentum constant must be in the range $0 \leq |\alpha| < 1$. When $\alpha$ is zero the back-propagation algorithm operates without momentum. Note that $\alpha$ can be negative, although it is unlikely to be used in practice. According to Sørensen (1994) $\alpha$ is typically chosen to be $0.2, 0.3, ..., 0.9$.
- When the partial derivative $\partial \mathcal{E}/\partial w_{ji}$ has the same sign on consecutive iterations, the sum $\Delta w_{ji}$ grows in magnitude, and so the weight is adjusted by a large amount. Hence the momentum term tends to accelerate the descent.
- When the partial derivative $\partial \mathcal{E}/\partial w_{ji}$ has opposite signs on consecutive iterations, the sum $\Delta w_{ji}$ shrinks in magnitude, and the weight is adjusted by a small amount. Hence the momentum term has a stabilising effect in directions that alternate in sign.

The momentum term may also prevent the learning from terminating in a shallow local minimum on the error surface.

The learning parameter $\eta$ has been assumed constant, but in reality it should be connec-

tion dependent. We may in fact constrain any number of weights to remain fixed by simply making the learning rate $\eta_{ji}$ for weight $w_{ji}$ equal to zero.

## 3.2 Pattern and batch modes of training

Back-propagation may proceed in one of two basic ways:

- *Pattern mode*. Weights are updated after each training example; that is how it was presented here. To be precise, consider an epoch consisting of $K$ training examples (*patterns*) arranged in the order $[\mathbf{u}_1, \mathbf{d}_1], \ldots, [\mathbf{u}_K, \mathbf{d}_K]$. The first example $[\mathbf{u}_1, \mathbf{d}_1]$ is presented to the network, and the forward and backward computations are performed. Then the second example $[\mathbf{u}_2, \mathbf{d}_2]$ is presented, and the forward and backward computations are repeated. This continues until the last example.
- *Batch mode*. Weights are updated *after* presenting *all* training examples; that constitutes an epoch.

From an on-line point of view, where the data are presented as time series, the pattern mode uses less storage. Moreover, given a random presentation of the patterns, the search in weight space becomes stochastic in nature, and the back-propagation gets trapped less likely in a local minimum. The batch mode, however, is appropriate off-line providing a more accurate estimate of the gradient vector. The choice between the two methods therefore depends on the problem at hand.

## 3.3 Initialisation

A good choice for the initial values of the weights can be important. The customary practice is to initialise all weights randomly and uniformly distributed within a small range of values. The wrong choice of initial weights can lead to *saturation* when the value of $\mathcal{E}$ remains constant for some period of time during the learning process, corresponding to a saddle point in the error surface. Assuming sigmoidal activation functions, it happens if the input to one (or several) activation functions is numerically so large that it operates on the tails of the sigmoids. Here the slope is rather small, the neuron is in saturation. Consequently the adjustments to the weights of the neuron will be small, and the network may take a long time to escape from it. The following observations have been made (Lee, Oh & Kim in Haykin, 1994):

- Incorrect saturation is avoided by choosing the initial values of the weights inside a *small* range of values.
- Incorrect saturation is less likely to occur when the number of hidden neurons is low.
- Incorrect saturation rarely occurs when the neurons operate in their linear regions.

## 3.4 Scaling

The training data most often have to be scaled. The data are measured in physical units, which often are of quite different orders of magnitude. This is inconvenient when the ac-

tivation functions are defined on standard universes. In the case of s-shaped activation functions, there is a risk that neurons will operate on the tails of the activation function, causing saturation phenomena and increased training time. Another problem is that the error function $\mathcal{E}$ favours elements accidentally measured in a unit which implies a large magnitude.

Scaling may avoid these problems. One way is to map the data such that the scaled signals have a mean value $m$ of zero, and a standard deviation $s$ of one, i.e.,

$$\mathbf{x}^* = \frac{\mathbf{x} - m}{s},$$

where $\mathbf{x}$ is a set of physical measurements, and $\mathbf{x}^*$ the corresponding vector in the scaled world of the network.

## 3.5 Stopping criteria

In principle training continues until the weights stabilise and the average squared error over the entire training set converges to some minimum value. There are actually no well-defined criteria for stopping the algorithm's operation, but there are some reasonable criteria.

It is logical to look for a zero gradient, since the gradient of the error surface is zero in a minimum. A stopping criterion could therefore be to stop when the Euclidean norm of the gradient vector is sufficiently small. Learning may take a long time, however, and it requires computing the gradient vector $\mathbf{g}(\mathbf{w})$.

Another possibility is to exploit that the error function is stationary at a minimum. The stopping criterion could therefore be to stop when the absolute rate of change in the average squared error per epoch is sufficiently small. A typical value is 0.1 to 1 percent per epoch; sometimes a value as small as 0.01 percent per epoch is used.

## 3.6 The number of training examples

Back-propagation learning starts with presenting a *training set* to the network. Hopefully the network is designed to *generalise*, that is, it is able to perform a nonlinear interpolation. The network generalises well when its output is nearly correct when presented with a *test data set* never used in training. The network can interpolate mainly because continuous activation functions lead to continuous output functions.

If a network learns too many input-output relations, i.e., there is *overfitting*, in the presence of noise, it is probably because it uses too many hidden neurons. When this happens the error is low; it looks like a desirable situation, but it is not. In the presence of noise it is undesirable to perfectly fit the data points. Instead a smoother approximation is more correct. A high error goal stops a network from *overtraining*, because it gives the network some tolerance in the fit.

The opposite effect, *underfitting*, is also possible. This occurs when there are too few weights, then the network cannot produce outputs reasonably close to the desired outputs. An example is illustrated in Fig. 17.

Overfitting and underfitting are affected by the size and the efficiency of the training

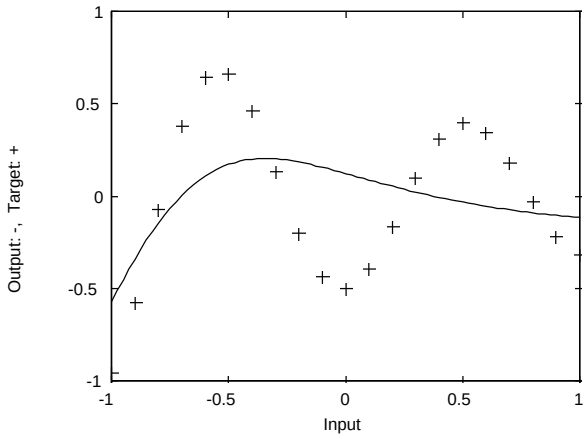


Figure 17: Due to underfitting, a network (solid line) has not been able to model the variations in the training data $(+)$.

set, and the architecture of the network. If the architecture is fixed, then a right size of the training set must be determined.

For the case of a network containing a single hidden layer used as a binary classifier, some guidelines are available (Baum & Haussler in Haykin, 1994). The network will almost certainly provide generalisation, if

1. the fraction of errors made on the training set is less than $\varepsilon/2$, and
2. the number of examples, $K$, used in training is

$$K \geq \frac{32W}{\varepsilon} \ln\left(\frac{32M}{\varepsilon}\right) \tag{79}$$

Here $M$ is the number of hidden neurons, $W$ is the total number of weights in the network, $\varepsilon$ is the ratio of the permissible error to the desired output, and $\ln$ is the natural logarithm. The equation provides a worst case formula for estimating the training set size for a single-layer neural network; in practice there can be a huge gap between the actual size of the training set needed and that predicted by (79). In practice, all we need is to satisfy the condition

$$K \geq \frac{W}{\varepsilon} \tag{80}$$

Thus, with an error ratio of, say, 0.1 (10 percent) the number of training examples should be approximately 10 times the number of weights in the network.

## 3.7 The number of layers and neurons

A single layer perceptron forms half-plane decision regions (Lippmann, 1987). A two-

layer perceptron can form any, possibly unbounded, convex region in the input space. Such regions are formed from intersections of the half-plane regions formed by each node in the first layer of the multi-layer perceptron. These convex regions have at most as many sides as there are nodes in the first layer. The number of nodes must be large enough to form a decision region that is as complex as is required. It must not, however, be so large that the weights required can not be readily estimated from the available training data. A three layer perceptron (with two hidden layers) can form arbitrarily complex decision regions. Thus no more than three layers are required in perceptron networks.

The number of nodes in the second (hidden) layer must be greater than one when decision regions are disconnected or meshed and cannot be formed from one convex region. The number of second layer nodes required in the worst case is equal to the number of disconnected regions in the input space. The number of nodes in the first layer must typically be sufficient to provide three or more edges for each convex area generated by every second layer node. There should thus typically be more than three times as many nodes in the second as in the first layer.

The above concerns primarily multi-layer perceptrons with one output when hard limiters (*signum* functions) are used as activation functions. Similar rules apply to multi-output networks where the resulting class corresponds to the output node with the largest output.

For control applications, not classification, Sørensen (1994) chooses to only consider multi-layer perceptrons with one hidden layer since it is sufficient.

# 4. How to model a dynamic plant

To return to the learning problem posed in the introduction, forward learning is the procedure of training a neural network to represent the dynamics of a plant (Fig. 1). The neural network is placed in parallel with the plant and the error between the system and the network outputs, the *prediction error*, is used as the training signal. In the case of a multi-layer perceptron the network can be trained by back-propagating the prediction error.

But how does the network allow for the dynamics of the plant? One possibility is to introduce dynamics into the network itself. This can be done using internal feedback in the network (a *recurrent* network) or by introducing transfer functions into the neurons. A straight forward approach is to augment the network input with signals corresponding to past inputs and outputs.

Assume that the plant is governed by a nonlinear discrete-time difference equation

$$y^p(t+1) = \mathrm{f}\left[y^p(t), \ldots, y^p(t-n+1); u(t), \ldots, u(t-m+1)\right] \tag{81}$$

Thus the system output $y^p$ at time $t+1$ is a nonlinear function $\mathrm{f}$ of the past $n$ output values and of the past $m$ values of the input $u$. The superscript $^p$ refers to the plant. An immediate idea is to choose the input-output structure of the network equal to the believed structure of the system. Denoting the output of the network as $y^m$ where superscript $^m$ refers to the model, we then have

$$y^m(t+1) = \widehat{\mathrm{f}}\left[y^p(t), \ldots, y^p(t-n+1); u(t), \ldots, u(t-m+1)\right] \tag{82}$$

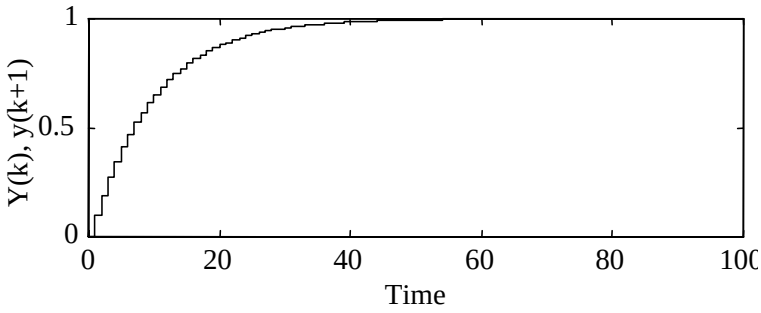


Figure 18: Experimental step response data for training.

Here $\widehat{\mathrm{f}}$ represents the nonlinear input-output map of the network (i.e. the approximation of $\mathrm{f}$). Notice that the input to the network includes the past values of the plant output (the network has no memory). This dependence is not included in Fig. 1 for simplicity. If we assume that after a suitable training period the network gives a good representation of the plant (i.e. $y^m \approx y^p$), then for subsequent post-training purposes the network output itself can be fed back and used as part of the network input. In this way the network can be used independently of the plant. This may be preferred when there is noise since it avoids problems of bias caused by noise in the plant.

A pure delay of $n_d$ samples in the plant can be directly incorporated. In the case of (81) the equation becomes

$$y^p(t+1) = \mathrm{f}\left[y^p(t), \ldots, y^p(t-n+1); u(t-n_d+1), \ldots, u(t-n_d-m+2)\right]$$

**Example 9 (first order plant)** *Given a dynamic plant with one input $u$ and one output $y$, we wish to model it with a neurofuzzy network using forward learning. Its step response to a unit step input $u(t) = 1$ $(0 \le t \le 100)$ will be used as training data; it is plotted in Fig.* 18*. Assume the plant can be modelled using a continuous time Laplace transfer function with one time constant $\tau$,*

$$y = \frac{1}{1+\tau s} u \tag{83}$$

*The discrete time (zero-order-hold) equivalent is*

$$y(k+1) = ay(k) + (1-a)u(k) \tag{84}$$

*where $a$ is a constant that depends on $\tau$ and the sampling time.*

*The network inputs and output are, cf. (82),*

$$Input_1 = y(k) \tag{85}$$

$$Input_2 = u(k) \tag{86}$$

$$Output = y(k+1) \tag{87}$$

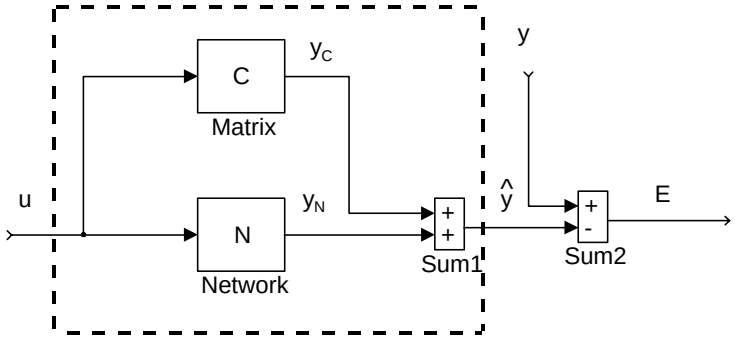


Figure 19: Parallel coupling with a matrix.

*Using the step response, we can produce a training set. Two examples from the set are*

| $y(k)$ | $u(k)$ | $y(k+1)$ |
|---|---|---|
| *0* | *1* | *0.1* |
| *1* | *1* | *1* |

(88)

*One may argue, assuming that (84) is an accurate model, that the first row determines $a$, and the problem is solved without neural nets. However, in case the plant contains a nonlinearity, not accounted for in (84), a single neuron with two inputs and one output might model the plant better.*

□

If a nonlinear plant is known to have partly linear, partly nonlinear gains from the inputs to the outputs, a network can be trained to model just the nonlinear behaviour of the plant (Sørensen, 1994). The network is coupled in parallel with a known constant gain matrix **C,** and the network and **C** are both fed with the data **u**(k), the input data to the plant (Fig. 19). The outputs from the network are $\mathbf{y}_N(k)$ and from the matrix $\mathbf{y}_C(k)$, and

$$\mathbf{y}_N(k) = \mathrm{F}\left(\mathbf{u}(k), \boldsymbol{\theta}\right) \tag{89}$$

$$\mathbf{y}_C(k) = \mathbf{C}\mathbf{u}(k) \tag{90}$$

The vector $\boldsymbol{\theta}$ contains all the weights and offsets in the network ordered in some way. The total output of the arrangement is

$$\widehat{\mathbf{y}}(k) = \mathbf{y}_N(k) + \mathbf{y}_C(k) \tag{91}$$

$$= \mathrm{F}\left(\mathbf{u}(k), \boldsymbol{\theta}\right) + \mathbf{C}\mathbf{u}(k) \tag{92}$$

The total gain matrix is

$$\mathbf{N}(k) = \frac{d\widehat{y}_i(k)}{du_j(k)}, \tag{93}$$

where index $i$ runs over all outputs and index $j$ over all inputs. The gain matrix **N,** evaluated at a particular input $\mathbf{u}(k)$ at time instance $k$, is called the *Jacobian* matrix of the function from **u** to $\widehat{\mathbf{y}}$. Each element is the gain on a small change of an input $u_j$, and the gain in

position $(i, j)$ refers to the signal path from input $u_j$ to output $\widehat{y}_i$. A further computation shows that

$$\mathbf{N}(k) = \frac{d\,\mathrm{F}\,(\mathbf{u}(k), \boldsymbol{\theta})}{d\mathbf{u}^T(k)} + \mathbf{C} \quad (94)$$

$$= \mathbf{N}_N\,(k) + \mathbf{C} \quad (95)$$

where $\mathbf{N}_N\,(k)$ is the gain matrix of the network. Equations (92) and (95) simply show that the combined parallel structure has a total output equal to the sum of the outputs from the nonlinear network and the linear matrix, and that the combined gain matrix is the sum of the two individual gain matrices. The configuration inside the dotted box in Fig. 19 behaves as an ordinary network, and can be trained as a network.

# 5. Summary

Back-propagation is an example of *supervised* learning. That means the network is shown (by a 'teacher') a desired output given an example from a training set. A performance measure is the sum of the squared errors visualised as an error surface. For the system to learn (from the 'teacher'), the operating point is forced to move toward a minimum point, local or global. The gradient points in the direction of the steepest descent. Given an adequate training set and enough time, the network is usually able to perform classification and function approximation satisfactorily. A network can be described in terms of matrices resulting in a model similar in form to the state-space model known from control engineering. A dynamic plant can be modelled using a static nonlinear regression in the discrete time domain.

# References


Agarwal, M. (1997). A systematic classification of neural-network-based control, *IEEE control systems* **17**(2): 75–93.

Demuth, H. and Beale, M. (1992). *Neural Network Toolbox: For Use with Matlab*, The MathWorks, Inc, Natick, MA, USA.

Fukuda, T. and Shibata, T. (1992). Theory and applications of neural networks for industrial control systems, *IEEE Transactions on industrial electronics* **39**(6): 472–489.

Haykin, S. (1994). *Neural Networks: A Comprehensive Foundation*, Macmillan College Publishing Company, Inc., 866 Third Ave, New York, NY 10022.

Hunt, K., Sbarbaro, D., Zbikowski, R. and Gawthrop, P. (1992). Neural networks for control systems - a survey, *Automatica* **28**(6): 1083–1112.

Lippmann, R. (1987). An introduction to computing with neural nets, *IEEE ASSP Magazine* pp. 4–22.

Meier, W., Weber, R. and Zimmermann, H.-J. (1994). Fuzzy data analysis-methods and industrial applications, *Fuzzy Sets and Systems* **61**: 19–28.

MIT (1995). *CITE Literature and Products Database*, MIT GmbH / ELITE, Promenade 9, D-

52076 Aachen, Germany.

MIT (1997). *DataEngine: Part II, Tutorials*, MIT GmbH, Promenade 9, D-52076 Aachen, Germany.

Nørgaard, P. M. (1996). *System Identification and Control with Neural Networks*, PhD thesis, Technical University of Denmark, Dept. of Automation, Denmark.

Nørgaard, P. M. (n.d.a). *NNCTRL Toolkit*, Technical University of Denmark: Dept. of Automation, http://www.iau.dtu.dk/research/control/nnctrl.html.

Nørgaard, P. M. (n.d.b). *NNSYSID Toolbox*, Technical University of Denmark: Dept. of Automation, http://www.iau.dtu.dk/Projects/proj/nnhtml.html.

Sørensen, O. (1994). *Neural Networks in Control Applications*, PhD thesis, Aalborg University, Institute of Electronic Systems, Dept. of Control Engineering, Denmark. ISSN 0908-1208.