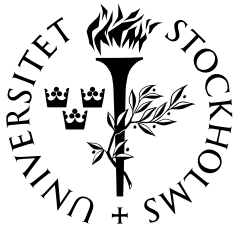


EXAMENSARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

The Mathematical Background of Artificial Neural Networks and their Application in the Medical Technology Project NIVA^{B}

av

Susanne Thon

2008 - No 14

The Mathematical Background of Artificial Neural Networks and their Application in the Medical Technology Project NIVA^{B}

Susanne Thon

Examensarbete i matematik 15 högskolepoäng, påbyggnadskurs

Handledare: Andreas Völkel

2008

Abstract

One of the most important features of the human brain is its ability to learn. The way in which the synapses between the brain's neurons are adapted in new situations is unique and the total capacity of biological neural networks has not been able to be simulated. Yet artificial neural networks are a powerful tool in pattern recognition and calculation as they are able to approximate any continuous multidimensional function. The proof of this property goes back to Kolmogorov and will be one of the main results which we will present in this thesis.
After giving the mathematical background of neural networks, we will turn to an application in medical technology. In the project NIVA^{B} (non-invasive determination of blood glucose level) neural networks are used for the calculation of blood glucose. On the basis of this project we will in the second part of this thesis demonstrate how neural networks can be realised with Matlab.

Acknowledgment

This work was carried out at the business enterprise TROUT GmbH in Kassel, Germany, during summer and autumn 2008. I would like to thank the TROUT GmbH for giving me the opportunity to work with this interesting project and my colleagues for their support and good cooperation.

Contents

1 Introduction

In this thesis we will investigate the concept of artificial neural networks for use in medical technology. Artificial neural networks are pattern recognition systems which are inspired by the functionality of biological neural networks. One of the most important neural systems is the human brain. The nervous system consists of biologically independent neurons which are connected by synapses. Each neuron is connected to thousands of other neurons. By transmitting impulses, the synapses pass information between the neurons. Each impulse causes a reaction. In the learning process the synapses between the neurons are modified. The reaction on an impulse is changed by trial and error until the feedback is optimal and recurring patterns are recognised.

Artificial neural networks simulate this way of learning and transmitting information. They are applied in a wide range of areas, such as in speech recognition, artificial intelligence, engineering and diagnostics in economy and medical science, where problems are hard to describe analytically and cannot be solved adequately by conventional methods.[1]

In this paper we will consider the mathematical background of artificial neural networks. After a short historical overview the concept of neural networks will be presented as a special area of graph theory. We will consider two- and three-layer feedforward networks and show as a main result that three-layer networks are able to approximate any continuous function. This result, which goes back to Kolmogorov, makes neural networks a powerful tool for calculation where the explicit functions are unknown.
To give an example of how learning can be carried out, we will consider the backpropagation training method, which is based on the method of gradient descent.

We will then show how neural networks can be used in Matlab, where a neural network toolbox is provided. We will describe how networks can be created and trained. Besides backpropagation a wide range of other training methods is implemented in the toolbox, some of which we will present without going into the mathematical details.
Finally we will present and analyse the medical technology project NIVA^{B}. This project is implemented with Matlab and applies neural networks as a tool for non-invasive determination of blood glucose values. On the background of this we will develop an own simplified version of a program for determining the blood glucose level.

[1] [Berns Kolb 1994] p. v

2 The Concept of Neural Networks

2.1 Historical Overview

A first idea of neural networks was presented by McCulloch and Pitts in the 1940s. In their work *"A logical calculus of ideas immanent in nervous activity"*[2] they construct a formal model of a neuron as a threshold-unit in two states, which can be seen as a foundation of artificial neural networks.[3]
In 1949 Hebb published his work *"The Organization of Behavior"*,[4] in which he presents an approach of learning in neural networks. In the Hebbian learning rule the connection between two neurons is strengthened, if they are active at the same time.
The first adaptive network model was developed in 1958 by Rosenblatt.[5] It was called perceptron and provides in its features a basis for most of the networks used today.
However, these networks turned out to be appropriate only for a limited class of problems and could therefore not provide a basis for problem solution in general. Research and public interest in neural networks abated for some decades, until the publication of Hopfield's work[6] in 1982 caused a new boom in neuroresearch. He describes recurrent networks as a totally new system of nonlinear networks. The output of each processing element in the binary, symmetrically weighted model is fully connected by weights to the inputs.
In the following years the first learning methods for multilayer networks were described. In 1986 Rumelhart, Hinton and Williams[7] presented backpropagation, a learning method that has its roots in the Ph.D. thesis of Werbos[8] in 1974. Backpropagation can be applied to any multilayer neural network and is one of the most popular methods still today.

2.2 The Topology of Neural Networks

In our presentation of the mathematical background of neural networks we will mainly follow Lenze's book *"Einführung in die Mathematik neuronaler Netze"*.[9] We will restrict ourselves to two- and three-layer networks, as this is sufficient for our purpose. However, the concept of two- and three-layer networks can be extended and in practical applications even networks with more than three layers are used.

We consider neural networks as a special case of directed graphs. Before introducing formal neural networks, we will present some notations of graph theory.

[2] [McCulloch Pitts 1943]
[3] [Berns Kolb 1994] p. 10
[4] [Hebb 1949]
[5] [Rosenblatt 1958]
[6] [Hopfield 1982]
[7] [Rumelhart Hinton Williams 1986]
[8] [Werbos 1974]
[9] [Lenze 2003]

2.2.1 Definition

Let X be a finite nonempty set. A **(finite) directed graph** $G := (X, H, \gamma)$ is made up of the elements of X, called the vertices of G, a finite set H such that $H \cap X = \{\}$, called the edges of G, and a mapping $\gamma : H \to X \times X$, called the incidence function of G.

2.2.2 Definition

Let $G := (X, H, \gamma)$ be a directed graph with vertices $v, w, v_0, v_1, ..., v_n \in X$ and edges $h, h_1, ..., h_k \in H$.

- If $\gamma(h) = (v, w)$ then v is called **origin** and w is called **terminus** of h.
- The number of edges for which v is origin is called **outgoing degree** of v and is denoted by $\delta^+(v)$, the number of edges for which v is terminus is called **incoming degree** of v and is denoted by $\delta^-(v)$.
- If there do not exist any edges h such that $\gamma(h) = (v, v)$ for any vertex v, then G is called **loop-free**.
- A loop-free graph which has at most one edge between any two different vertices is called **simple graph**.
- A finite sequence $C := (v_0, h_1, v_1, h_2, v_2, ..., v_{k-1}, h_k, v_k)$ such that $\gamma(h_i) = (v_{i-1}, v_i),\ 1 \leq i \leq k,$ is called **directed path** from v_0 to v_k. $\alpha(C) := v_0$ is called **start vertex**, $\omega(C) := v_k$ is called **end vertex** of the path.
- A directed path C is called **closed directed path**, or **cycle**, if $\alpha(C) = \omega(C)$. Otherwise C is called **open directed path**.

We are now able to introduce formal neural networks and formal neurons.

2.2.3 Definition

Let $G := (X, H, \gamma)$ be a simple directed graph. We define

$$\widetilde{X} := X \setminus \{v \in X : \delta^+(v) \cdot \delta^-(v) = 0\},$$
$$\widetilde{H} := H \setminus \{h \in H : \gamma(h) \in (X \setminus \widetilde{X}) \times (X \setminus \widetilde{X})\}.$$

Then $N := (X, \widetilde{X}, H, \widetilde{H}, \gamma)$ is called **(formal) neural network**.

The elements of $\widetilde{X}$ are those vertices which are both origin and terminus, the elements of $\widetilde{H}$ are the edges between vertices in $\widetilde{X}$, i.e. to get the set $\widetilde{H}$ from the set H, all edges between vertices $v, w \in X$ such that v is not terminus and w is not origin are removed. We will use the following notation:

- The elements $v \in \widetilde{X}$ are called **nodes** of N.
- The elements $h \in \widetilde{H}$ are called **vectors** of N.

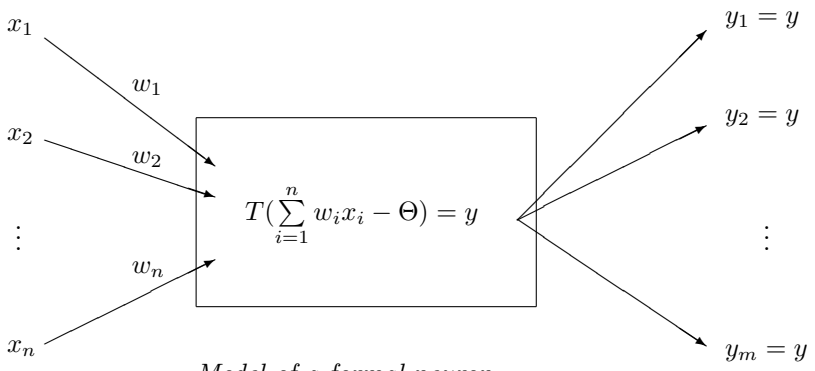
- All nodes $v \in \widetilde{X}$ for which exist a $w \in X \setminus \widetilde{X}$ and an $h \in \widetilde{H}$ such that $\gamma(h) = (w, v)$ are called **input nodes** of N. The vector h is then called **input vector** of N.
- All nodes $v \in \widetilde{X}$ for which exist a $w \in X \setminus \widetilde{X}$ and an $h \in \widetilde{H}$ such that $\gamma(h) = (v, w)$ are called **output nodes** of N. The vector h is then called **output vector** of N.
- If the simple directed graph G induced by N does not contain any cycles then N is called **feedforward neural network**. Otherwise N is called **recurrent neural network**.

2.2.4 Definition

A **(formal) neuron** is a function

$$\kappa : \mathbb{R}^n \to \mathbb{R}^m$$
$$(x_1, ..., x_n) \mapsto (T(\sum_{i=1}^{n} w_i x_i - \Theta), ..., T(\sum_{i=1}^{n} w_i x_i - \Theta))$$

with **weight vector** $\vec{w} = (w_1, ..., w_n) \in \mathbb{R}^n$, **threshold value** $\Theta \in \mathbb{R}$, and **transfer function** $T : \mathbb{R} \to \mathbb{R}$.



Model of a formal neuron

A neuron works in the following way: The input values $x_1, ..., x_n$ are weighted by $w_1, ..., w_n$. They are then added so that we get a total stimulation $\sum_{i=1}^{n} w_i x_i$. In this manner the weight vector regulates how much influence each neuron has on the total stimulation of the output neuron.

The threshold value is subtracted from the total stimulation: $\sum_{i=1}^{n} w_i x_i - \Theta$.

Θ indicates the sensibility of the network. If Θ is large, even the total stimulation

has to be large to give positive stimulation to the neuron.
The transfer function translates the stimulation of a neuron into some kind of neural activity by $T(\sum_{i=1}^{n} w_i x_i - \Theta)$. Some typical transfer functions are:

- The **linear function**

$$T(x) = \alpha x, \quad \alpha \in \mathbb{R}^+.$$

  The stimulation of the neuron is multiplied by a positive scalar.
  If $\alpha = 1$, we get as a special case the following function:

- The **identity function**

$$T(x) = x =: T_I(x).$$

- The **step function**

$$T(x) = \left\{ \begin{array}{ll} 0, & \text{if } x < 0 \\ 1, & \text{if } x \geq 0 \end{array} \right\} =: T_1(x).$$

  If the total stimulation of the neuron is positive, it "fires", otherwise it does not. This simulates the behavior of real neurons.

- The **Sigmoid functions**
  A sigmoid function is any bounded function $T : \mathbb{R} \to \mathbb{R}$ such that

$$\lim_{x \to -\infty} T(x) = 0 \quad \text{and} \quad \lim_{x \to \infty} T(x) = 1.$$

  The neuron fires with an intensity between 0 and 1.

- The **Fermi function**

$$T(x) = \frac{1}{1 + e^{-x}} =: T_F(x)$$

  is one example of a sigmoid function.

- The **hyperbolic tangent function**

$$T(x) = \tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} =: T_H(x)$$

  can be seen as an adjusted Fermi function as $T_H(x) = 2\, T_F(2x) - 1$.

Now we imagine each node in a neural network as such a formal neuron. The neurons are ordered in layers. To know which neuron is stimulated by which other neuron, they are numbered. This is called **time-discrete update** or scheduling.

The neurons in the first layer, i.e. the input neurons of the network N, are stimulated. The impulses are adapted in the way described above and propagated to the neurons of the second layer and so on. Finally, the output neurons return the network's reaction on the primary impulse.
Regarding all these conditions, a neural network N with formal neurons as nodes and given time-discrete update is an implementation of a function

$$\aleph : \mathbb{R}^n \to \mathbb{R}^m,$$

which in its turn is called "The Neural Network". We assume that N has n input vectors and m output vectors. $\aleph$ depends thus in a complex way on the weights, threshold values and transfer functions of all neurons as well as on the given update.

2.3 Learning in Artificial Neural Networks

How will the threshold values and the weight vectors be determined? The neural network has to learn them. In a first step they are initialised by random values. A finite set of training data $(\vec{x}^{(1)}, \vec{y}^{(1)}), ..., (\vec{x}^{(t)}, \vec{y}^{(t)})$, consisting of input values $\vec{x}^{(j)} \in \mathbb{R}^n$ and output values, or targets, $\vec{y}^{(j)} \in \mathbb{R}^m$, $1 \leq j \leq t$, is presented to a network with n input nodes and m output nodes. To each input $\vec{x}^{(j)}$ the net calculates an output, which is compared to the target $\vec{y}^{(j)}$. The weights and threshold values will then be adapted by a given learning rule. There are many learning rules in common use, some of which will be presented later.
When learning is finished, the network should be able to map each training vector of input values $\vec{x}^{(j)}$ to the right, or at least approximately right, output value $\vec{y}^{(j)}$, as well as to calculate an appropriate output to any other input. This form of learning is called **supervised learning**.

There exists even another form of learning, called **unsupervised learning**. In this case a finite amount of inputs $\vec{x}^{(1)}, ..., \vec{x}^{(t)}$ is presented to the neural network. The network should organise the presented data and discover collective properties. By searching for regularities or trends in the inputs, the network makes adaptions and should, after the learning process, be able to classify primary unknown inputs. In this thesis we will not discuss further issues on unsupervised learning as it is not relevant in our context.

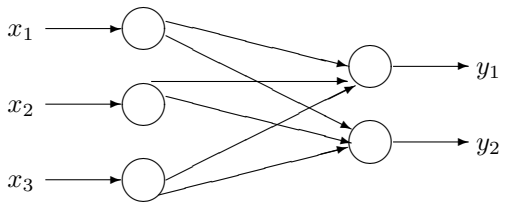
3 Two-Layer Feedforward Networks

3.1 Construction

3.1.1 Definition

A **two-layer feedforward network** is a neural network whose nodes and vectors have the following properties:

- Each node is either input or output node.
- To each input node exists exactly one input vector.
- To each output node exists exactly one output vector.
- Each input node is connected to each output node. Other connections do not exist.



A two-layer feedforward network with three input nodes and two output nodes

The **input layer** consists of all input nodes, the **output layer** consists of all output nodes.
If the net has n input and m output nodes, it should have the following features:

- The input nodes are formal neurons with identity transfer function T_I. The i-th input neuron has weight vector $\vec{w_i} = 1 \in \mathbb{R}$ and threshold value $\Theta_i = 0 \in \mathbb{R}$ for $1 \leq i \leq n$.
- The output nodes are formal neurons with a common transfer function T. For $1 \leq j \leq m$, the j-th output node has weight vector $\vec{w_j} = (w_{1j}, ..., w_{nj}) \in \mathbb{R}^n$ and threshold value $\Theta_j \in \mathbb{R}$.

Thus for an input $\vec{x} = (x_1, ..., x_n) \in \mathbb{R}^n$ the network calculates the output $\vec{y} = (y_1, ..., y_m) \in \mathbb{R}^m$ by

$$y_j = T(\sum_{i=1}^{n} w_{ij} x_i - \Theta_j), \quad 1 \leq j \leq m.$$

3.2 The Linear Associator with Hebbian Learning Rule

One of the simplest and first studied models of neural networks is the linear associator. It is a two-layer feedforward network which uses the identity transfer function in the output layer and the Hebbian learning rule. The Hebbian learning rule was introduced in 1949 by Donald Hebb in his book *"The Organization of Behavior"*.[10] It basically states that if one neuron is active and this activity is caused by another neuron's activity, the weight between these two neurons should be increased. The threshold value is equal to zero. The rule is formulated as follows:

3.2.1 Definition

Let N be a two-layer feedforward network with n input neurons and m output neurons. For $1 \leq j \leq m$, let $\vec{w_j} \in \mathbb{R}^n$ be the weight vectors to be learned, initialised by $\vec{w_j}^{(0)} = \vec{0}$, and let $\Theta_j = 0$ for $1 \leq j \leq m$. Let $(\vec{x}^{(1)}, \vec{y}^{(1)}), ..., (\vec{x}^{(t)}, \vec{y}^{(t)}) \in \mathbb{R}^n \times \mathbb{R}^m$ be a set of training patterns which is presented to the network. The rule

$$\vec{w_{ij}}^{(t)} := \sum_{r=1}^{t} y_j^{(r)} x_i^{(r)}, \qquad 1 \leq i \leq n,\ 1 \leq j \leq m,$$

is called **Hebbian learning rule**. After training, the network has weights $\vec{w_j} = \vec{w_j}^{(t)}$ and threshold values $\Theta_j = 0$ for $1 \leq j \leq m$.

According to that the linear associator is defined as follows:

3.2.2 Definition

A two-layer feedforward neural network with identity transfer function T_I in the output layer and Hebbian learning rule is called **linear associator with Hebbian learning rule**.

How do we have to choose training data in order to make the net work perfectly on them? The following theorem will answer this question.

3.2.3 Theorem

Let N be a linear associator with Hebbian learning rule and n input nodes and m output nodes. Let

$$(\vec{x}^{(1)}, \vec{y}^{(1)}), ..., (\vec{x}^{(t)}, \vec{y}^{(t)}) \in \mathbb{R}^n \times \mathbb{R}^m$$

be a set of training patterns which is presented to the network. After learning, N works perfectly on these training data, if $\vec{x}^{(1)}, ..., \vec{x}^{(t)}$ is an orthonormal set of vectors, i.e. if $\vec{x}^{(r)} \cdot \vec{x}^{(s)} = 0$ for $r \neq s$ and $\left\|\vec{x}^{(r)}\right\|_2 = \sqrt{\vec{x}^{(r)} \cdot \vec{x}^{(r)}} = 1$, where $\cdot$

[10] [Hebb 1949]

denotes the scalar product.
After training we will get

$$T_I(\sum_{i=1}^{n} w_{ij}x_i^{(s)} - \Theta_j) = T_I(\sum_{i=1}^{n} w_{ij}x_i^{(s)}) = y_j^{(s)}, \qquad 1 \leq j \leq m, \quad 1 \leq s \leq t.$$

Proof

After training the network, the weights are

$$w_{ij} = \sum_{r=1}^{t} y_j^{(r)} x_i^{(r)}, \qquad 1 \leq i \leq n, \quad 1 \leq j \leq m.$$

Thus we get for an input vector $x_i^{(s)}$, $1 \leq s \leq t$:

$$T_I(\sum_{i=1}^{n} w_{ij}x_i^{(s)}) = \sum_{i=1}^{n} w_{ij}x_i^{(s)} = \sum_{i=1}^{n}\sum_{r=1}^{t} y_j^{(r)} x_i^{(r)} x_i^{(s)} = \sum_{r=1}^{t} y_j^{(r)} (\sum_{i=1}^{n} x_i^{(r)} x_i^{(s)}) = y_j^{(s)},$$

because $x_i^{(r)} x_i^{(s)} = \left\{ \begin{array}{ll} 0, & \text{if } r \neq s \\ 1, & \text{if } r = s \end{array} \right\}$ for $1 \leq j \leq m$.

□

We see that the training data must not consist of more than n pairs of vectors as a set of orthonormal vectors $\vec{x_i} \in \mathbb{R}^n$ cannot contain more than n vectors. Furthermore, we see that the network can only work perfectly on the training data, if the input vectors have length 1. Thus the capacity of the linear associator is limited.

3.3 The Perceptron with Perceptron Learning Rule

The perceptron learning rule was developed in the late 1950s by Frank Rosenblatt. Based on the Hebbian learning rule, the idea was to use even threshold values and to modify weights and threshold values only if the net does not work perfectly on training data yet.

3.3.1 Definition

Let N be a two-layer feedforward neural network with n input neurons and m output neurons. For $1 \leq j \leq m$, let $\vec{w_j} \in \mathbb{R}^n$ and $\Theta_j \in \mathbb{R}$ be the weight vectors and threshold values respectively to be learned, initialised by $\vec{w_j}^{(0)} = \vec{0}$ and $\Theta_j^{(0)} = 0$. Let $(\vec{x}^{(1)}, \vec{y}^{(1)}), ..., (\vec{x}^{(t)}, \vec{y}^{(t)}) \in \mathbb{R}^n \times \mathbb{R}^m$ be a set of training patterns, which is presented to the network. The rule

$$\begin{aligned}
&\widetilde{y_j}^{(s)} := T(\vec{w_j}^{(s-1)} \cdot \vec{x}^{(s)} - \Theta_j^{(s-1)}), \\
&\vec{w_j}^{(s)} := \vec{w_j}^{(s-1)} + (y_j^{(s)} - \widetilde{y_j}^{(s)})\vec{x}^{(s)}, \\
&\Theta_j^{(s)} := \Theta_j^{(s-1)} - (y_j^{(s)} - \widetilde{y_j}^{(s)}), \qquad 1 \leq j \leq m, \quad 1 \leq s \leq t,
\end{aligned}$$

is called **perceptron learning rule**. After training, the network has weights $\vec{w_j} = \vec{w_j}^{(t)}$ and threshold values $\Theta_j = \Theta_j^{(t)}$ for $1 \leq j \leq m$.
Here $\widetilde{y_j}^{(s)}$ is the j-th output vector the network generates from input $\vec{x}^{(s)}$ before learning, the target vector $y_j^{(s)}$ is the value we want the network to proceed for the j-th output vector. Thus, after the network has learned $s-1$ training patterns, the weights and threshold values are updated in the s-th step only if $\widetilde{y_j}^{(s)}$ and $y_j^{(s)}$ differ.
If the net does not work perfectly on the training data after one learning epoch, learning can be repeated using the weights and threshold values of the current network as initial values.

3.3.2 Definition

A two-layer feedforward neural network with step function T_1 as transfer function in the output layer on which the perceptron learning rule is applied repeatedly is called **perceptron with (iterative) perceptron learning rule**.

How should training data be chosen in order to make the net work perfectly on it after finitely many epochs? To answer this question we need the following definition.

3.3.3 Definition

Let $(\vec{x}^{(1)}, \vec{y}^{(1)}), ..., (\vec{x}^{(t)}, \vec{y}^{(t)}) \in \mathbb{R}^n \times \mathbb{R}^m$ be such that $y_j^{(s)} \in \{0,1\}$ for $1 \leq j \leq m$, $1 \leq s \leq t$, i.e. each component of the vectors $\vec{y}^{(1)}, ..., \vec{y}^{(t)}$ is either 0 or 1. $(\vec{x}^{(1)}, \vec{y}^{(1)}), ..., (\vec{x}^{(t)}, \vec{y}^{(t)})$ are called **strictly linearly separable**, if to each $j \in \{1, ..., m\}$ exist a $\delta_j > 0$, a vector $\vec{w_j}^* \in \mathbb{R}^n$, and a value $\Theta_j^* \in \mathbb{R}$ such that for $s = 1, ..., t$

$$(\vec{w_j}^*, \Theta_j^*) \cdot (\vec{x}^{(s)}, -1) = \sum_{i=1}^{n} w_{ij}^* x_i^{(s)} - \Theta_j^* \left\{ \begin{array}{ll} \geq \delta_j, & \text{if } y_j^{(s)} = 1 \\ \leq -\delta_j, & \text{if } y_j^{(s)} = 0 \end{array} \right\}.$$

Here we denote the vector $(w_1, ..., w_n, \Theta)$ by $(\vec{w}, \Theta)$ for $\vec{w} \in \mathbb{R}^n$, $\Theta \in \mathbb{R}$.

Now we can state the following theorem.

3.3.4 Theorem

Let N be a perceptron with (iterative) perceptron learning rule having n input neurons and m output neurons. Let $(\vec{x}^{(1)}, \vec{y}^{(1)}), ..., (\vec{x}^{(t)}, \vec{y}^{(t)}) \in \mathbb{R}^n \times \{0,1\}^m$ be strictly linearly separable training data which are presented to the network. Then the net works perfectly on the training data after finitely many learning iterations. After learning is completed we get

$$T_1(\sum_{i=1}^{n} w_{ij} x_i^{(s)} - \Theta_j) = y_j^{(s)}, \quad 1 \leq j \leq m, \quad 1 \leq s \leq t.$$

Proof

We write

$$\begin{array}{rcl} (\vec{x}^{(t+1)}, \vec{y}^{(t+1)}) & := & (\vec{x}^{(1)}, \vec{y}^{(1)}) \\ (\vec{x}^{(t+2)}, \vec{y}^{(t+2)}) & := & (\vec{x}^{(2)}, \vec{y}^{(2)}) \\ \vdots & \vdots & \vdots \\ (\vec{x}^{(2t)}, \vec{y}^{(2t)}) & := & (\vec{x}^{(t)}, \vec{y}^{(t)}) \\ (\vec{x}^{(2t+1)}, \vec{y}^{(2t+1)}) & := & (\vec{x}^{(1)}, \vec{y}^{(1)}) \\ \vdots & \vdots & \vdots \end{array}$$

Thus we can assume that we have infinitely many pairs of training data $(\vec{x}^{(s)}, \vec{y}^{(s)})$, $s \in \mathbb{N}$, when iterating the learning process. These pairs $(\vec{x}^{(s)}, \vec{y}^{(s)})$ are strictly linearly separable by assumption, i.e. for each $j \in \{1, 2, ..., m\}$ there exist $\delta_j > 0$, ${\vec{w}_j}^* \in \mathbb{R}^n$, and $\Theta_j^* \in \mathbb{R}$ such that

$${\vec{w}_j}^* \cdot \vec{x}^{(s)} - \Theta_j^* \left\{ \begin{array}{ll} \geq \delta_j, & \text{if } y_j^{(s)} = 1 \\ \leq -\delta_j, & \text{if } y_j^{(s)} = 0 \end{array} \right\}. \tag{1}$$

The weights and threshold values are

$${\vec{w}_j}^{(0)} = \vec{0}, \qquad \Theta_j^{(0)} = 0$$

and

$$\begin{array}{l} \widetilde{y_j}^{(s)} = T_1({\vec{w}_j}^{(s-1)} \cdot \vec{x}^{(s)} - \Theta_j^{(s-1)}), \\ {\vec{w}_j}^{(s)} = {\vec{w}_j}^{(s-1)} + (y_j^{(s)} - \widetilde{y_j}^{(s)})\vec{x}^{(s)}, \\ \Theta_j^{(s)} = \Theta_j^{(s-1)} - (y_j^{(s)} - \widetilde{y_j}^{(s)}), \qquad s \geq 1, \end{array}$$

for $1 \leq j \leq m$.

We define $M := \max\limits_{1 \leq s \leq t} \left\|\vec{x}^{(s)}\right\|_2^2 = \max\limits_{s \in \mathbb{N}} \left\|\vec{x}^{(s)}\right\|_2^2$.

We fix a j-th output neuron and consider those subsequences $(s_u)_{u \in \mathbb{N}}$ for which

$$y_j^{(s_u)} - \widetilde{y_j}^{(s_u)} = \pm 1,$$

i.e. for which the values actually are changed.
Then we have

$$y_j^{(s_u)} - \widetilde{y_j}^{(s_u)} = \left\{ \begin{array}{ll} 1, & \text{if } y_j^{(s_u)} = 1 \\ -1, & \text{if } y_j^{(s_u)} = 0 \end{array} \right\} \tag{2}$$

and

$$(y_j^{(s_u)} - \widetilde{y_j}^{(s_u)})^2 = 1\ . \tag{3}$$

From $\widetilde{y_j}^{(s)} = T_1(\vec{w_j}^{(s-1)} \cdot \vec{x}^{(s)} - \Theta_j^{(s-1)})$ follows

$$\vec{w_j}^{(s_{u-1})} \cdot \vec{x}^{(s_u)} - \Theta_j^{(s_{u-1})} \left\{ \begin{array}{ll} < 0, & \text{if } y_j^{(s_u)} = 1 \\ \geq 0, & \text{if } y_j^{(s_u)} = 0 \end{array} \right\}. \tag{4}$$

We have to show that

$$y_j^{(s_u)} - \widetilde{y_j}^{(s_u)} = \pm 1, \qquad u \in \mathbb{N},$$

is possible only for a finite number of u.

Let $u \in \mathbb{N}$, then

$$\begin{aligned}
&\left\|(\vec{w_j}^{(s_u)}, \Theta_j^{(s_u)})\right\|_2^2 = (\vec{w_j}^{(s_u)}, \Theta_j^{(s_u)}) \cdot (\vec{w_j}^{(s_u)}, \Theta_j^{(s_u)}) = \vec{w_j}^{(s_u)} \cdot \vec{w_j}^{(s_u)} + \Theta_j^{(s_u)} \cdot \Theta_j^{(s_u)} \\
&= (\vec{w_j}^{(s_{u-1})} + (y_j^{(s_u)} - \widetilde{y_j}^{(s_u)})\vec{x}^{(s_u)}) \cdot (\vec{w_j}^{(s_{u-1})} + (y_j^{(s_u)} - \widetilde{y_j}^{(s_u)})\vec{x}^{(s_u)}) + (\Theta_j^{(s_{u-1})} - (y_j^{(s_u)} - \widetilde{y_j}^{(s_u)}))^2 \\
&= \vec{w_j}^{(s_{u-1})} \cdot \vec{w_j}^{(s_{u-1})} + 2(y_j^{(s_u)} - \widetilde{y_j}^{(s_u)})\vec{w_j}^{(s_{u-1})} \cdot \vec{x}^{(s_u)} + (y_j^{(s_u)} - \widetilde{y_j}^{(s_u)})^2 \vec{x}^{(s_u)} \cdot \vec{x}^{(s_u)} \\
&\qquad + (\Theta_j^{(s_{u-1})})^2 - 2(y_j^{(s_u)} - \widetilde{y_j}^{(s_u)})\Theta_j^{(s_{u-1})} + (y_j^{(s_u)} - \widetilde{y_j}^{(s_u)})^2 \\
&\qquad = \left\|(\vec{w_j}^{(s_{u-1})}, \Theta_j^{(s_{u-1})})\right\|_2^2 \\
&+ \underbrace{2(y_j^{(s_u)} - \widetilde{y_j}^{(s_u)})(\vec{w_j}^{(s_{u-1})} \cdot \vec{x}^{(s_u)} - \Theta_j^{(s_{u-1})})}_{\leq 0 \text{ by (2) and (4)}} + \underbrace{(y_j^{(s_u)} - \widetilde{y_j}^{(s_u)})^2}_{= 1 \text{ by (3)}} (1 + \underbrace{\left\|\vec{x}^{(s_u)}\right\|_2^2}_{\leq M}) \\
&\qquad \leq \left\|(\vec{w_j}^{(s_{u-1})}, \Theta_j^{(s_{u-1})})\right\|_2^2 + (1 + M).
\end{aligned}$$

It follows

$$\begin{aligned}
\left\|(\vec{w_j}^{(s_u)}, \Theta_j^{(s_u)})\right\|_2^2 &\leq \left\|(\vec{w_j}^{(s_{u-1})}, \Theta_j^{(s_{u-1})})\right\|_2^2 + (1 + M) \\
&\leq \left\|(\vec{w_j}^{(s_{u-2})}, \Theta_j^{(s_{u-2})})\right\|_2^2 + 2(1 + M) \\
&\leq \\
&\vdots \\
&\leq \left\|(\vec{w_j}^{(s_0)}, \Theta_j^{(s_0)})\right\|_2^2 + u(1 + M) \\
&= u(1 + M) \qquad \text{as } \vec{w_j}^{(s_0)} = \vec{w_j}^{(0)} = \vec{0} \text{ and } \Theta_j^{(s_0)} = \Theta_j^{(0)} = 0 \\
&\Longrightarrow \left\|(\vec{w_j}^{(s_u)}, \Theta_j^{(s_u)})\right\|_2 \leq \sqrt{u(1 + M)}.
\end{aligned}$$

Using Cauchy-Schwarz inequality we get

$$(\vec{w_j}^{(s_u)}, \Theta_j^{(s_u)}) \cdot (\vec{w_j}^*, \Theta_j^*) \leq |(\vec{w_j}^{(s_u)}, \Theta_j^{(s_u)}) \cdot (\vec{w_j}^*, \Theta_j^*)| \leq \left\|(\vec{w_j}^{(s_u)}, \Theta_j^{(s_u)})\right\|_2 \left\|(\vec{w_j}^*, \Theta_j^*)\right\|_2$$

$$\leq \left\|(\vec{w_j}^*, \Theta_j^*)\right\|_2 \sqrt{u(1+M)}. \tag{5}$$

On the other hand we have

$$\vec{w_j}^{(s_u)} = \sum_{v=1}^{u} (y_j^{(s_v)} - \widetilde{y_j}^{(s_v)})\vec{x}^{(s_v)},$$
$$\Theta_j^{(s_u)} = -\sum_{v=1}^{u} (y_j^{(s_v)} - \widetilde{y_j}^{(s_v)}) \ .$$

From (1) and (2) follows

$$(\vec{w_j}^{(s_u)}, \Theta_j^{(s_u)}) \cdot (\vec{w_j}^*, \Theta_j^*) = \sum_{v=1}^{u} (y_j^{(s_v)} - \widetilde{y_j}^{(s_v)})(\vec{x}^{(s_v)}, -1) \cdot (\vec{w_j}^*, \Theta_j^*)$$

$$= \sum_{v=1}^{u} (y_j^{(s_v)} - \widetilde{y_j}^{(s_v)})(\vec{w_j}^* \cdot \vec{x}^{(s_v)} - \Theta_j^*) \geq u \ \delta_j$$

$$\implies (\vec{w_j}^{(s_u)}, \Theta_j^{(s_u)}) \cdot (\vec{w_j}^*, \Theta_j^*) \geq u \ \delta_j. \tag{6}$$

Combining (5) and (6) we get

$$u \ \delta_j \leq (\vec{w_j}^{(s_u)}, \Theta_j^{(s_u)}) \cdot (\vec{w_j}^*, \Theta_j^*) \leq \left\|(\vec{w_j}^*, \Theta_j^*)\right\|_2 \sqrt{u(1+M)}$$

$$\iff \sqrt{u} \leq \left\|(\vec{w_j}^*, \Theta_j^*)\right\|_2 \sqrt{1+M} \ \delta_j^{-1}.$$

This equality holds only for finitely many $u \in \mathbb{N}$, thus the j-th neuron does learn in a finite number of epochs. As the learning process for the output neurons is independent, each of the m output neurons learns in a finite number of epochs.

□

In practical use it is difficult to decide whether data are strictly linearly separable. One possibility could be to implement algorithms that analyse training data before using them. However, this demands additional calculating time. Another way could be just to use training data without testing them to see if learning terminates after finitely many epochs. If learning does not finish after a long time, no judgment can be made whether training time was not long enough or data were not strictly linearly separable. In the latter case the net would never learn to work perfectly on current training data. Therefore other solutions are needed.

4 Three-Layer Feedforward Networks

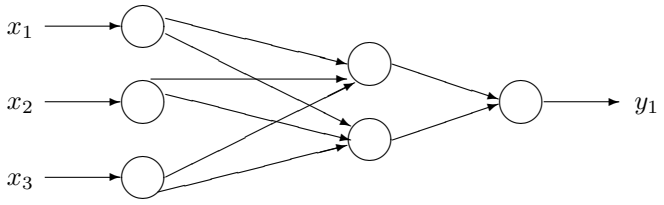
4.1 Construction

As we have seen, the two-layer feedforward networks with respective learning rules we presented in the previous chapter are limited in their functionality as they put certain claims on training data in order to be able to work perfectly on them after training is finished. Therefore we introduce three-layer feedforward networks as a more powerful instrument.

4.1.1 Definition

A three-layer feedforward network is a neural network whose nodes and vectors have the following properties:

- To each input node exists exactly one input vector.
- To each output node exists exactly one output vector.
- Each node that is neither input nor output node is called **hidden node**.
- Each input node is connected to each hidden node and each hidden node is connected to each output node. Other connections do not exist.



A three-layer feedforward network with three input nodes, two hidden nodes and one output node

The **input layer** consists of all input nodes, the **output layer** consists of all output nodes, and the **hidden layer** consists of all hidden nodes.
If the net has n input nodes, q hidden nodes, and m output nodes, it should have the following features:

- The input nodes are formal neurons with identity transfer function T_I. The i-th input node has weight vector $\vec{w_i} = 1 \in \mathbb{R}$ and threshold value $\Theta_i = 0$ for $1 \leq i \leq n$.
- The hidden nodes are formal neurons with a common transfer function T. The p-th hidden node has weight vector $\vec{w_p} = (w_{1p}, ..., w_{np}) \in \mathbb{R}^n$ and threshold value $\Theta_p \in \mathbb{R}$ for $1 \leq p \leq q$.

- The output nodes are formal neurons with identity transfer function T_I. The j-th output node has weight vector $\vec{g_j} = (g_{1j}, ..., g_{qj}) \in \mathbb{R}^q$ and threshold value $\Phi_j = 0$ for $1 \leq j \leq m$.

Thus for an input $\vec{x} = (x_1, ..., x_n) \in \mathbb{R}^n$ the network calculates the output $\vec{y} = (y_1, ..., y_m) \in \mathbb{R}^m$ by

$$y_j = \sum_{p=1}^{q} g_{pj} T(\sum_{i=1}^{n} w_{ip} x_i - \Theta_p), \qquad 1 \leq j \leq m.$$

We will now show the important result that makes neural networks such a powerful tool for calculation: Three-layer neural networks have the potential to approximate any continuous multidimensional function. This result goes historically back to Kolmogorov, who proved that it is possible to describe any multidimensional function on a compact space by linear combination and composition of one-dimensional functions.[11]

We start by showing that a three-layer neural network can approximate any continuous one-dimensional function $f : \mathbb{R} \to \mathbb{R}$.

4.1.2 Theorem

Let $[a, b] \subset \mathbb{R}$ be a closed interval and $\epsilon > 0$. Let $f : [a, b] \to \mathbb{R}$ be an arbitrary continuous function and let $T : \mathbb{R} \to \mathbb{R}$ be a continuous sigmoid transfer function, i.e. $\lim_{x \to -\infty} T(x) = 0$ and $\lim_{x \to \infty} T(x) = 1$. Then there exist $u, v \in \mathbb{N}$ such that the function $S : \mathbb{R} \to \mathbb{R}$,

$$\begin{aligned} S(x) := \quad & f(a) T(v(x - a + \tfrac{b-a}{u})) \\ & + \sum_{k=1}^{u} \Big(f(a + \tfrac{k}{u}(b-a)) - f(a + \tfrac{k-1}{u}(b-a)) \Big) T(v(x - (a + \tfrac{k}{u}(b-a)))) \end{aligned}$$

satisfies

$$|f(x) - S(x)| \leq \epsilon$$

for all $x \in [a, b]$.

Proof

Let

$$\begin{aligned} \|T\|_\infty &:= \sup_{x \in \mathbb{R}} |T(x)| < \infty \quad \text{and} \\ \|f\|_\infty &:= \sup_{x \in [a,b]} |f(x)| < \infty. \end{aligned}$$

As f is a continuous function on a closed interval, f is uniformly continuous, i.e. for

$$\widetilde{\epsilon} := \frac{\epsilon}{3 + 2\|T\|_\infty + \|f\|_\infty}$$

[11] [Lenze 2003] p. 112

there exists a $\delta > 0$ such that for all $y, z \in [a, b]$ with $|y - z| \leq \delta$, we have $|f(y) - f(z)| \leq \widetilde{\epsilon}$.

We choose a natural number $u \in \mathbb{N}$ such that

$$\frac{b-a}{u} \leq \delta$$

and $v \in \mathbb{N}$ such that

$$\begin{aligned} x \geq v\tfrac{b-a}{u} \quad &\Longrightarrow \quad |1 - T(x)| \leq \min\{\widetilde{\epsilon}, \tfrac{1}{u}\} \qquad \text{and} \\ x \leq -v\tfrac{b-a}{u} \quad &\Longrightarrow \quad\ \ |T(x)| \leq \min\{\widetilde{\epsilon}, \tfrac{1}{u}\}. \end{aligned}$$

For an arbitrary given $x \in [a, b]$ we choose $l \in \{0, 1, ..., u-1\}$ such that

$$a + \frac{l}{u}(b-a) \leq x \leq a + \frac{l+1}{u}(b-a).$$

Then it follows

$$\begin{aligned} &|f(x) - S(x)| \\ &= |f(x) - f(a)T(v(x - a + \frac{b-a}{u})) - \sum_{k=1}^{u} \Big(f(a + \frac{k}{u}(b-a)) - f(a + \frac{k-1}{u}(b-a))\Big) T(v(x - (a + \frac{k}{u}(b-a))))| \\ &\overset{(1)}{\leq} |f(x) - f(a)T(v(x - a + \frac{b-a}{u})) - \sum_{k=1}^{l} \Big(f(a + \frac{k}{u}(b-a)) - f(a + \frac{k-1}{u}(b-a))\Big) T(v(x - (a + \frac{k}{u}(b-a))))| \\ &\quad + |\sum_{k=l+1}^{u} \Big(f(a + \frac{k}{u}(b-a)) - f(a + \frac{k-1}{u}(b-a))\Big) T(v(x - (a + \frac{k}{u}(b-a))))| \\ &\overset{(2)}{\leq} |f(x) - f(a)T(v(x - a + \frac{b-a}{u})) - \sum_{k=1}^{l} \Big(f(a + \frac{k}{u}(b-a)) - f(a + \frac{k-1}{u}(b-a))\Big)| \\ &+ \sum_{k=1}^{l} |\Big(f(a + \frac{k}{u}(b-a)) - f(a + \frac{k-1}{u}(b-a))\Big)| \quad |1 - T(v(x - (a + \frac{k}{u}(b-a))))| \\ &+ \sum_{k=l+1}^{u} |\Big(f(a + \frac{k}{u}(b-a)) - f(a + \frac{k-1}{u}(b-a))\Big)| \quad |T(v(x - (a + \frac{k}{u}(b-a))))| \\ &\overset{(3)}{\leq} |f(x) - f(a + \frac{l}{u}(b-a))| + |f(a) - f(a)T(v(x - a + \frac{b-a}{u}))| \\ &\quad + \widetilde{\epsilon} \sum_{k=1}^{l} |1 - T(v(x - (a + \frac{k}{u}(b-a))))| + \widetilde{\epsilon} \sum_{k=l+1}^{u} |T(v(x - (a + \frac{k}{u}(b-a))))| \\ &\overset{(4)}{\leq} \widetilde{\epsilon} + \|f\|_\infty |1 - T(v(x - a + \frac{b-a}{u}))| \end{aligned}$$

$$+\widetilde{\epsilon}(1+\|T\|_\infty)+\widetilde{\epsilon}\sum_{k=1}^{l-1}|1-T(v(x-(a+\frac{k}{u}(b-a))))|$$

$$+\widetilde{\epsilon}\|T\|_\infty+\widetilde{\epsilon}\sum_{k=l+2}^{u}|T(v(x-(a+\frac{k}{u}(b-a))))|$$

$$\overset{(5)}{\leq}\widetilde{\epsilon}+\|f\|_\infty\widetilde{\epsilon}+\widetilde{\epsilon}(1+\|T\|_\infty)+\widetilde{\epsilon}\sum_{k=1}^{l-1}\frac{1}{u}+\widetilde{\epsilon}\|T\|_\infty+\widetilde{\epsilon}\sum_{k=l+2}^{u}\frac{1}{u}$$

$$=2\widetilde{\epsilon}+2\widetilde{\epsilon}\|T\|_\infty+\|f\|_\infty\widetilde{\epsilon}+\widetilde{\epsilon}(\frac{u-2}{u})$$

$$\leq 3\widetilde{\epsilon}+2\widetilde{\epsilon}\|T\|_\infty+\widetilde{\epsilon}\|f\|_\infty=\epsilon.$$

As x was arbitrary, the theorem follows.

□

For a better understanding we will give some explanations to the transformations that were made:

(1): We split the sum $\sum_{k=1}^{u}$ into $\sum_{k=1}^{l}$ and $\sum_{k=l+1}^{u}$ and use triangle inequality $|x+y|\leq|x|+|y|$.

(2): We add zero to the sum by adding

$$\sum_{k=1}^{l}\Big(f(a+\frac{k}{u}(b-a))-f(a+\frac{k-1}{u}(b-a))\Big)$$

and subtracting it again. Then we use triangle inequality and $|xy|\leq|x|\ |y|$ to split the sums.

(3): Most of the terms in the sum

$$-\sum_{k=1}^{l}\Big(f(a+\frac{k}{u}(b-a))-f(a+\frac{k-1}{u}(b-a))\Big)$$

cancel out each other, only

$$-f(a+\frac{l}{u}(b-a))+f(a)$$

remains. Triangle inequality is used again.
Further we know that

$$|f(a+\frac{k}{u}(b-a))-f(a+\frac{k-1}{u}(b-a))|\leq\widetilde{\epsilon}$$

because

$$|a+\frac{k}{u}(b-a)-(a+\frac{k-1}{u}(b-a))|=|\frac{b-a}{u}|\leq\delta.$$

(4): As

$$a+\frac{l}{u}(b-a)\leq x\leq a+\frac{l+1}{u}(b-a)\Leftrightarrow 0\leq x-(a+\frac{l}{u}(b-a))\leq a+\frac{l+1}{u}(b-a)-(a+\frac{l}{u}(b-a))$$

it follows from

$$|x-(a+\frac{l}{u}(b-a))|\leq|a+\frac{l+1}{u}(b-a)-a-\frac{l}{u}(b-a)|=|\frac{b-a}{u}|\leq\delta$$

that

$$|f(x)-f(a+\frac{l}{u}(b-a))|\leq\widetilde{\epsilon}.$$

Further we know that

$$|f(a)|\leq\sup_{x\in[a,b]}|f(x)|=:\|f\|_\infty.$$

Again we split up the sums $\sum_{k=1}^{l}$ and $\sum_{k=l+1}^{u}$ and use triangle inequality and the fact that

$$|T(v(x-(a+\frac{k}{u}(b-a))))|\leq\sup_{x\in\mathbb{R}}|T(x)|=:\|T\|_\infty$$

for $k=l$ and $k=l+1$.

(5): As $x\in[a,b]$ it follows from

$$x\geq a\Leftrightarrow x-a+\frac{b-a}{u}\geq\frac{b-a}{u}$$

that

$$|1-T(v(x-a+\frac{b-a}{u}))|\leq\min\{\widetilde{\epsilon},\frac{1}{u}\}\leq\widetilde{\epsilon}.$$

For $k\leq l-1$ it follows from $a+\frac{l}{u}(b-a)\leq x$ that

$$x-(a+\frac{k}{u}(b-a))\geq\frac{l-k}{u}(b-a)\geq\frac{b-a}{u}$$

so that

$$|1-T(v(x-(a+\frac{k}{u}(b-a))))|\leq\min\{\widetilde{\epsilon},\frac{1}{u}\}\leq\frac{1}{u}.$$

Analogous it follows for $k\geq l+2$ from $x\leq a+\frac{l+1}{u}(b-a)$ that

$$x-(a+\frac{k}{u}(b-a))\leq\frac{l+1-k}{u}(b-a)\leq-\frac{b-a}{u}$$

so that

$$|T(v(x-(a+\frac{k}{u}(b-a))))| \leq \min\{\widetilde{\epsilon}, \frac{1}{u}\} \leq \frac{1}{u}.$$

The function S can be interpreted as a neural network with one input neuron, $q = u+1$ hidden neurons, and one output neuron with

$w_p \in \mathbb{R}$	as the weight vector for the p-th hidden neuron,
$\Theta_p \in \mathbb{R}$	as the threshold value for the p-th hidden neuron, and
$(g_1, ..., g_q) \in \mathbb{R}^q$	as the weight vector for the output neuron

in the following way:

$$\begin{aligned} S(x) := \quad & f(a)T(v(x-a+\tfrac{b-a}{u})) \\ & + \sum_{k=1}^{u} \Big(f(a+\tfrac{k}{u}(b-a)) - f(a+\tfrac{k-1}{u}(b-a))\Big) T(v(x-(a+\tfrac{k}{u}(b-a)))) \\ = \quad & f(a)T(vx - v(a-\tfrac{b-a}{u})) \\ & + \sum_{p=2}^{u+1} \Big(f(a+\tfrac{p-1}{u}(b-a)) - f(a+\tfrac{p-2}{u}(b-a))\Big) T(vx - v(a+\tfrac{p-1}{u}(b-a))) \\ = \quad & \sum_{p=1}^{q} g_p T(w_p x - \Theta_p), \end{aligned}$$

where

$$\begin{array}{lll} g_1 = f(a), & g_p = f(a+\frac{p-1}{q-1}(b-a)) - f(a+\frac{p-2}{q-1}(b-a)) & \text{for } p = 2, ..., q, \\ w_1 = v, & w_p = v & \text{for } p = 2, ..., q, \\ \Theta_1 = v(a-\frac{b-a}{q-1}), & \Theta_p = v(a+\frac{p-1}{q-1}(b-a)) & \text{for } p = 2, ..., q, \end{array}$$

$q = u+1$.

We now consider the multidimensional case.

4.1.3 Theorem

Let $K \subset \mathbb{R}^n$, $K \neq \{\}$, be a compact subset of $\mathbb{R}^n$ and let $f : K \to \mathbb{R}^m$ be a function $f = (f_1, ..., f_m)$ which is continuous on K. Then there exist for all $\epsilon > 0$ and all continuous sigmoid transfer functions T parameters

$$\begin{array}{ll} q \in \mathbb{N}, & \\ \vec{w_p} \in \mathbb{R}^n, & 1 \leq p \leq q, \\ \Theta_p \in \mathbb{R}, & 1 \leq p \leq q, \\ \vec{g_j} \in \mathbb{R}^q, & 1 \leq j \leq m, \end{array}$$

such that for all $\vec{x} \in K$

$$|f_j(\vec{x}) - \sum_{p=1}^{q} g_{pj} T(\sum_{i=1}^{n} w_{ip} x_i - \Theta_p)| \leq \epsilon, \quad 1 \leq j \leq m.$$

The proof of this theorem uses two results from classical analysis which we will present without proving them. For proofs, see e.g. [Lenze 2003] pp. 41-43 and 91.

Tietze extension theorem states that if $K \subset \mathbb{R}^n$, $K \neq \{\}$, is a compact subset of $\mathbb{R}^n$ and if $f : K \to \mathbb{R}$ is continuous, then there exists a function $F : \mathbb{R}^n \to \mathbb{R}$ which is continuous on $\mathbb{R}^n$ and an n-dimensional interval $[-\alpha, \alpha]^n \supset K$, $\alpha > 0$, such that

$$F(\vec{x}) = f(\vec{x}) \text{ for all } \vec{x} \in K \text{ and } F(\vec{x}) = 0 \text{ for all } \vec{x} \in \mathbb{R}^n \backslash [-\alpha, \alpha]^n.$$

Weierstrass theorem for trigonometric polynomials states that given an arbitrary $\alpha > 0$ and a continuous function $f : \mathbb{R}^n \to \mathbb{R}$ such that

$$f(\vec{x}) = f(\vec{x} + 2\alpha\vec{k}), \quad \vec{x} \in \mathbb{R}^n, \vec{k} \in \mathbb{Z}^n,$$

there exists to each $\epsilon > 0$ a finite amount of nonnegative integer multi-indices $\vec{k_r} \in \mathbb{N}_0^n$ and coefficients $\gamma_r, \delta_r \in \mathbb{R}$, $1 \leq r \leq R$, such that the trigonometric polynomial $P : \mathbb{R}^n \to \mathbb{R}$,

$$P(\vec{x}) := \sum_{r=1}^{R} (\gamma_r \cos(\frac{\pi}{\alpha} \vec{k_r} \cdot \vec{x}) + \delta_r \sin(\frac{\pi}{\alpha} \vec{k_r} \cdot \vec{x})),$$

satisfies

$$|f(\vec{x}) - P(\vec{x})| < \epsilon$$

for all $\vec{x} \in \mathbb{R}^n$.

Proof of Theorem 4.1.3

We prove the theorem for a fixed j-th component function $f_j : \mathbb{R}^n \to \mathbb{R}$. This is sufficient as each component function can be described by assigning a particular set of hidden neurons to each output neuron and choosing the weights of the other hidden neurons in the output neuron to equal zero.
Let $j \in \{1, 2, ..., m\}$ be arbitrary but fixed. According to Tietze extension theorem we can assume that f_j can be extended from K to the whole $\mathbb{R}^n$ such

that it is identically zero outside a sufficiently large interval $[-\alpha,\alpha]^n,\ \alpha>0$, which contains K. We now assign to f_j its continuous 2α-periodic extension

$$\begin{aligned}&\widetilde{f_j}:\mathbb{R}^n\to\mathbb{R},\\&\widetilde{f_j}(\vec{x})=f_j(\vec{t}),\qquad \vec{t}\in[-\alpha,\alpha]^n,\quad \vec{x}\equiv\vec{t}\bmod 2\alpha\ .\end{aligned}$$

By Weierstrass theorem for trigonometric polynomials there exist to any $\epsilon>0$ a finite amount of nonnegative integer multi-indices $\vec{k_r}\in\mathbb{N}_0^n$ and coefficients $\gamma_r,\delta_r\in\mathbb{R},\ 1\le r\le R,$ such that

$$|\widetilde{f_j}(\vec{x})-\sum_{r=1}^{R}\Big(\gamma_r\cos(\frac{\pi}{\alpha}\vec{k_r}\cdot\vec{x})+\delta_r\sin(\frac{\pi}{\alpha}\vec{k_r}\cdot\vec{x})\Big)|<\frac{\epsilon}{3}\qquad\text{for all }\vec{x}\in\mathbb{R}^n.$$

For all $\vec{x}\in K\subset[-\alpha,\alpha]^n$ it follows in particular

$$|f_j(\vec{x})-\sum_{r=1}^{R}\Big(\gamma_r\cos(\frac{\pi}{\alpha}\vec{k_r}\cdot\vec{x})+\delta_r\sin(\frac{\pi}{\alpha}\vec{k_r}\cdot\vec{x})\Big)|<\frac{\epsilon}{3}.$$

We define

$$M:=\max\{|\frac{\pi}{\alpha}\vec{k_r}\cdot\vec{x}|:1\le r\le R,\quad\vec{x}\in K\}.$$

By theorem 4.1.2, the one-dimensional functions *sin* and *cos* on [-M,M] can be approximated by one-dimensional functions s_1 and s_2 respectively. As we have proved, there exist sufficiently large numbers $u,v\in\mathbb{N}$ such that

$$\begin{aligned}s_1(x):=\quad&\cos(-M)T(v(x+M+\tfrac{2M}{u}))\\&+\sum_{k=1}^{u}\Big(\cos(-M+\tfrac{2Mk}{u})-\cos(-M+\tfrac{2M(k-1)}{u})\Big)T(v(x-(-M+\tfrac{2Mk}{u})))\end{aligned}$$

and

$$\begin{aligned}s_2(x):=\quad&\sin(-M)T(v(x+M+\tfrac{2M}{u}))\\&+\sum_{k=1}^{u}\Big(\sin(-M+\tfrac{2Mk}{u})-\sin(-M+\tfrac{2M(k-1)}{u})\Big)T(v(x-(-M+\tfrac{2Mk}{u})))\end{aligned}$$

satisfy

$$|\cos(x)-s_1(x)|\le\frac{\epsilon}{3\Big(1+\sum_{r=1}^{R}|\gamma_r|\Big)},\qquad x\in[-M,M],$$

and

$$|\sin(x)-s_2(x)|\le\frac{\epsilon}{3\Big(1+\sum_{r=1}^{R}|\delta_r|\Big)},\qquad x\in[-M,M],$$

for an arbitrary sigmoid function T.

We then define $S : \mathbb{R}^n \to \mathbb{R}$,

$$S(\vec{x}) := \sum_{r=1}^{R} \Big(\gamma_r s_1(\frac{\pi}{\alpha}\vec{k_r} \cdot \vec{x}) + \delta_r s_2(\frac{\pi}{\alpha}\vec{k_r} \cdot \vec{x}) \Big).$$

It follows for all $\vec{x} \in K$ that

$$\begin{aligned} |f_j(\vec{x}) - S(\vec{x})| \quad &\leq |f_j(\vec{x}) - \sum_{r=1}^{R} \Big(\gamma_r \cos(\tfrac{\pi}{\alpha}\vec{k_r} \cdot \vec{x}) + \delta_r \sin(\tfrac{\pi}{\alpha}\vec{k_r} \cdot \vec{x}) \Big)| \\ &\quad + |\sum_{r=1}^{R} \gamma_r \Big(\cos(\tfrac{\pi}{\alpha}\vec{k_r} \cdot \vec{x}) - s_1(\tfrac{\pi}{\alpha}\vec{k_r} \cdot \vec{x}) \Big)| \\ &\quad + |\sum_{r=1}^{R} \delta_r \Big(\sin(\tfrac{\pi}{\alpha}\vec{k_r} \cdot \vec{x}) - s_2(\tfrac{\pi}{\alpha}\vec{k_r} \cdot \vec{x}) \Big)| \\ &\leq \tfrac{\epsilon}{3} + \sum_{r=1}^{R} |\gamma_r| \frac{\epsilon}{3\Big(1 + \sum_{r=1}^{R} |\gamma_r|\Big)} + \sum_{r=1}^{R} |\delta_r| \frac{\epsilon}{3\Big(1 + \sum_{r=1}^{R} |\delta_r|\Big)} \\ &< \epsilon \, . \end{aligned}$$

Similar to the one-dimensional case, the function S can be interpreted as a neural network with n input neurons, $q = (u+1)R$ hidden neurons, and one output neuron with

$(w_{1p}, ..., w_{np}) \in \mathbb{R}^n$	as the weight vector for the p-th hidden neuron,
$\Theta_p \in \mathbb{R}$	as the threshold value for the p-th hidden neuron, and
$(g_1, ..., g_q) \in \mathbb{R}^q$	as the weight vector for the output neuron

in the following way:

$$\begin{aligned} S(x) &= \sum_{r=1}^{R} \Big(\gamma_r s_1(\tfrac{\pi}{\alpha}\vec{k_r} \cdot \vec{x}) + \delta_r s_2(\tfrac{\pi}{\alpha}\vec{k_r} \cdot \vec{x}) \Big) \\ &= \sum_{r=1}^{R} \gamma_r \cos(-M) T(v(\tfrac{\pi}{\alpha}\vec{k_r} \cdot \vec{x} + M + \tfrac{2M}{u})) \\ &+ \sum_{r=1}^{R} \gamma_r \sum_{l=1}^{u} (\cos(-M + \tfrac{2Ml}{u}) - \cos(-M + \tfrac{2M(l-1)}{u})) T(v(\tfrac{\pi}{\alpha}\vec{k_r} \cdot \vec{x} - (-M + \tfrac{2Ml}{u}))) \\ &+ \sum_{r=1}^{R} \delta_r \sin(-M) T(v(\tfrac{\pi}{\alpha}\vec{k_r} \cdot \vec{x} + M + \tfrac{2M}{u})) \\ &+ \sum_{r=1}^{R} \delta_r \sum_{l=1}^{u} (\sin(-M + \tfrac{2Ml}{u}) - \sin(-M + \tfrac{2M(l-1)}{u})) T(v(\tfrac{\pi}{\alpha}\vec{k_r} \cdot \vec{x} - (-M + \tfrac{2Ml}{u}))) \end{aligned}$$

$$= \sum_{r=1}^{R} \Big(\gamma_r \cos(-M) + \delta_r \sin(-M)\Big) T(\sum_{i=1}^{n} v\tfrac{\pi}{\alpha} k_{r,i} x_i - v(-M - \tfrac{2M}{u}))$$
$$+ \sum_{r=1}^{R} \sum_{l=1}^{u} \Big(\gamma_r(\cos(-M + \tfrac{2Ml}{u}) - \cos(-M + \tfrac{2M(l-1)}{u}))$$
$$+\delta_r(\sin(-M + \tfrac{2Ml}{u}) - \sin(-M + \tfrac{2M(l-1)}{u}))\Big) T(\sum_{i=1}^{n} v\tfrac{\pi}{\alpha} k_{r,i} x_i - v(-M + \tfrac{2Ml}{u}))$$

$$= \sum_{r=1}^{q} g_{rj} T(\sum_{i=1}^{n} w_{ir} x_i - \Theta_r)$$

with

$$\begin{aligned} \Theta_r &= v(-M - \tfrac{2M}{u}), \\ g_{rj} &= \gamma_r \cos(-M) + \delta_r \sin(-M), \\ w_{ir} &= v\tfrac{\pi}{\alpha} k_{r,i} \end{aligned}$$

for $r = 1, ..., R,\ i = 1, ..., n$ and

$$\begin{aligned} \Theta_{R+(r-1)u+l} &= v(-M + \tfrac{2Ml}{u}), \\ g_{R+(r-1)u+l,j} &= \gamma_r(\cos(-M + \tfrac{2Ml}{u}) - \cos(-M + \tfrac{2M(l-1)}{u})) \\ &\quad +\delta_r(\sin(-M + \tfrac{2Ml}{u}) - \sin(-M + \tfrac{2M(l-1)}{u})), \\ w_{i,R+(r-1)u+l} &= v\tfrac{\pi}{\alpha} k_{r,i} \end{aligned}$$

for $l = 1, ..., u$, $r = 1, ..., R$, and $i = 1, ..., n$.
Here $k_{r,i}$ denotes the i-th component of the vector $\vec{k_r}$. As j was arbitrary, the theorem follows.

□

We have shown that each continuous multidimensional function on a compact space can be approximated correctly to an arbitrary small error. This is, however, primarily a proof of existence as it does not tell how in an easy and clear way a network can be constructed for a given problem.

We will now present one of the most important methods for calculating the weights and threshold values in three-layer neural networks, the backpropagation training method.

4.2 Backpropagation

Backpropagation is one optimisation method for determining the optimal weights and threshold values of a network for a given problem. The backpropagation technique involves two phases. In the first phase, called the forward phase, an input is given to and propagated forward through the network such that an output value is computed for each neuron. The total output will then be compared to the desired output, or target, and the difference, or error, will be computed. In the second phase, called the backward phase, the weights and threshold values in each successive layer are adjusted to reduce this error.

In terms of functions this means: A set of training data $(\vec{x}^{(s)}, \vec{y}^{(s)}) \in \mathbb{R}^n \times \mathbb{R}^m$, $1 \leq s \leq t$, is presented to a neural network. To each input value $\vec{x}^{(s)}$ the net computes output values

$$\widetilde{y_j}^{(s)} = \sum_{p=1}^{q} g_{pj} T(\sum_{i=1}^{n} w_{ip} x_i^{(s)} - \Theta_p), \qquad 1 \leq j \leq m,\ 1 \leq s \leq t.$$

The weights and threshold values should then be adjusted in such a way that the square error

$$\left(y_j^{(s)} - \widetilde{y_j}^{(s)}\right)^2 = \left(y_j^{(s)} - \sum_{p=1}^{q} g_{pj} T(\sum_{i=1}^{n} w_{ip} x_i^{(s)} - \Theta_p)\right)^2$$

becomes as small as possible for all $1 \leq j \leq m$ and $1 \leq s \leq t$. This can be done by a wide range of methods. The method used in backpropagation is based on the method of gradient descent.

4.2.1 Definition

Let the function $f : \mathbb{R}^n \to \mathbb{R}$ be partially differentiable such that its partial derivatives are continuous. We denote the gradient of f by

$$\text{grad} f(\vec{x}) := \left(\frac{\partial f}{\partial x_1}(\vec{x}), ..., \frac{\partial f}{\partial x_n}(\vec{x})\right) =: (f_{x_1}(\vec{x}), ..., f_{x_n}(\vec{x})).$$

Let $\lambda > 0$ be an arbitrary but fixed given value, called the step size. If f has a (local) minimum, then the algorithm for finding this minimum,

$$\begin{aligned} &\vec{x}^{(0)} \in \mathbb{R}^n, \\ &\vec{x}^{(k)} := \vec{x}^{(k-1)} - \lambda\ \text{grad} f(\vec{x}^{(k-1)}), \quad k \geq 1, \end{aligned}$$

is called the **method of gradient descent**.

If the method of gradient descent converges, it will after a sufficient number of iterations find a critical point $\vec{x}^*$ of the function f, i.e. a point $\vec{x}^*$ which satisfies $\text{grad} f(\vec{x}^*) = 0$. If further all second partial derivatives of f exist and if the Hessian matrix

$$H_f(\vec{x}) := \left(\frac{\partial^2 f}{\partial x_i \partial x_j}(\vec{x})\right)_{1 \leq i,j \leq n} \in \mathbb{R}^{n \times n}$$

is positive definite in $\vec{x}^*$, then f has a (local) minimum at $\vec{x}^*$.

Many iterations may be necessary until a critical point is found. Therefore we can stop iteration when we have reached a point $\vec{x}^{(k)}$ that is close enough to the minimum, i.e. $|\vec{x}^{(k)} - \vec{x}^*| < \epsilon$ for a given $\epsilon > 0$.

For the given problem

$$\left(y_j^{(s)} - \sum_{p=1}^{q} g_{pj} T(\sum_{i=1}^{n} w_{ip} x_i^{(s)} - \Theta_p)\right)^2 \stackrel{!}{=} \min$$

we assume that the transfer function T is continuously differentiable. To each pair of training data $(\vec{x}^{(s)}, \vec{y}^{(s)})$ we define the error function

$$F^{(s)} : \mathbb{R}^{nq} \times \mathbb{R}^{q} \times \mathbb{R}^{qm} \to \mathbb{R}, \quad 1 \leq s \leq t,$$

as

$$\begin{aligned} &F^{(s)}(w_{11}, ..., w_{n1}, ..., w_{1q}, ..., w_{nq}, \Theta_1, ..., \Theta_q, g_{11}, ..., g_{q1}, ..., g_{1m}, ..., g_{qm}) \\ &:= \sum_{j=1}^{m} \Big(y_j^{(s)} - \sum_{p=1}^{q} g_{pj} T(\sum_{i=1}^{n} w_{ip} x_i^{(s)} - \Theta_p) \Big)^2, \quad 1 \leq s \leq t. \end{aligned}$$

To minimise the error for the s-th pair of training data, we calculate the minimum of the error function $F^{(s)}$ with respect to the weights g_{pj}, w_{ip}, and the threshold values Θ_p for $1 \leq p \leq q$, $1 \leq j \leq m$, and $1 \leq i \leq n$. For one gradient step, this is done as follows:

1. For the weights $g_{pj}, \quad 1 \leq p \leq q,\ 1 \leq j \leq m$:
   The partial derivative of $F^{(s)}$ with respect to the weight g_{pj} is

   $$\begin{aligned} &F^{(s)}_{g_{pj}}(w_{11}, ..., w_{n1}, ..., w_{1q}, ..., w_{nq}, \Theta_1, ..., \Theta_q, g_{11}, ..., g_{q1}, ..., g_{1m}, ..., g_{qm}) \\ &= 2\Big(y_j^{(s)} - \sum_{\tilde{p}=1}^{q} g_{\tilde{p}j} T(\sum_{\tilde{i}=1}^{n} w_{\tilde{i}\tilde{p}} x_{\tilde{i}}^{(s)} - \Theta_{\tilde{p}}) \Big) \cdot (-1) T(\sum_{\tilde{i}=1}^{n} w_{\tilde{i}p} x_{\tilde{i}}^{(s)} - \Theta_p). \end{aligned}$$

   Thus the weight g_{pj} is modified as follows:

   $$g_{pj}^{(new)} = g_{pj} - \lambda F^{(s)}_{g_{pj}}(w_{11}, ..., w_{n1}, ..., w_{1q}, ..., w_{nq}, \Theta_1, ..., \Theta_q, g_{11}, ..., g_{q1}, ..., g_{1m}, ..., g_{qm}).$$

2. For the weights $w_{ip}, \quad 1 \leq i \leq n,\ 1 \leq p \leq q$:
   The partial derivative of $F^{(s)}$ with respect to the weight w_{ip} is

   $$\begin{aligned} &F^{(s)}_{w_{ip}}(w_{11}, ..., w_{n1}, ..., w_{1q}, ..., w_{nq}, \Theta_1, ..., \Theta_q, g_{11}, ..., g_{q1}, ..., g_{1m}, ..., g_{qm}) \\ &= 2 \sum_{\tilde{j}=1}^{m} \Big((y_{\tilde{j}}^{(s)} - \sum_{\tilde{p}=1}^{q} g_{\tilde{p}\tilde{j}} T(\sum_{\tilde{i}=1}^{n} w_{\tilde{i}\tilde{p}} x_{\tilde{i}}^{(s)} - \Theta_{\tilde{p}})) \cdot (-1) g_{p\tilde{j}} \Big) \cdot T'(\sum_{\tilde{i}=1}^{n} w_{\tilde{i}p} x_{\tilde{i}}^{(s)} - \Theta_p) \cdot x_i^{(s)}. \end{aligned}$$

   Thus the weight w_{ip} is modified as follows:

   $$w_{ip}^{(new)} = w_{ip} - \lambda F^{(s)}_{w_{ip}}(w_{11}, ..., w_{n1}, ..., w_{1q}, ..., w_{nq}, \Theta_1, ..., \Theta_q, g_{11}, ..., g_{q1}, ..., g_{1m}, ..., g_{qm}).$$

3. For the threshold value $\Theta_p, \quad 1 \leq p \leq q$:
   The partial derivative of $F^{(s)}$ with respect to the threshold value Θ_p is

   $$\begin{aligned} &F^{(s)}_{\Theta_p}(w_{11}, ..., w_{n1}, ..., w_{1q}, ..., w_{nq}, \Theta_1, ..., \Theta_q, g_{11}, ..., g_{q1}, ..., g_{1m}, ..., g_{qm}) \\ &= 2 \sum_{\tilde{j}=1}^{m} \Big((y_{\tilde{j}}^{(s)} - \sum_{\tilde{p}=1}^{q} g_{\tilde{p}\tilde{j}} T(\sum_{\tilde{i}=1}^{n} w_{\tilde{i}\tilde{p}} x_{\tilde{i}}^{(s)} - \Theta_{\tilde{p}})) \cdot (-1) g_{p\tilde{j}} \Big) \cdot T'(\sum_{\tilde{i}=1}^{n} w_{\tilde{i}p} x_{\tilde{i}}^{(s)} - \Theta_p) \cdot (-1). \end{aligned}$$

   Thus the threshold value Θ_p is modified as follows:

   $$\Theta_p^{(new)} = \Theta_p - \lambda F^{(s)}_{\Theta_p}(w_{11}, ..., w_{n1}, ..., w_{1q}, ..., w_{nq}, \Theta_1, ..., \Theta_q, g_{11}, ..., g_{q1}, ..., g_{1m}, ..., g_{qm}).$$

This updating method is successively applied to all error functions $F^{(s)}$, $1 \leq s \leq t$. The error $F^{(s)}$ is propagated back into the net in order to modify the parameters. This defines the backpropagation learning method for neural networks.

4.2.2 Definition

Let N be a three-layer feedforward neural network with n input neurons, q hidden neurons, and m output neurons. Let the transfer function T in the hidden layer be continuously differentiable. The weights $\vec{w}_p \in \mathbb{R}^n$, $1 \leq p \leq q$, the threshold values $\Theta_p \in \mathbb{R}$, $1 \leq p \leq q$, and the weights $\vec{g}_j \in \mathbb{R}^q$, $1 \leq j \leq m$, are initialised by arbitrary values $\vec{w}_p^{(0)} \in \mathbb{R}^n$, $\Theta_p^{(0)} \in \mathbb{R}$, and $\vec{g}_j^{(0)} \in \mathbb{R}^q$. Let $(\vec{x}^{(1)}, \vec{y}^{(1)}), ..., (\vec{x}^{(t)}, \vec{y}^{(t)}) \in \mathbb{R}^n \times \mathbb{R}^m$ be a set of training data and let $\lambda > 0$ be a fixed step size, then for $1 \leq s \leq t$ the algorithm

$$\widetilde{y}_j^{(s)} = \sum_{\tilde{p}=1}^{q} g_{\tilde{p}j}^{(s-1)} T(\sum_{\tilde{i}=1}^{n} w_{\tilde{i}\tilde{p}}^{(s-1)} x_{\tilde{i}}^{(s)} - \Theta_{\tilde{p}}^{(s-1)}), \quad 1 \leq j \leq m,$$

$$g_{pj}^{(s)} = g_{pj}^{(s-1)} + \lambda 2 (y_j^{(s)} - \widetilde{y}_j^{(s)}) \cdot T(\sum_{\tilde{i}=1}^{n} w_{\tilde{i}p}^{(s-1)} x_{\tilde{i}}^{(s)} - \Theta_p^{(s-1)}), \quad 1 \leq p \leq q,\ 1 \leq j \leq m,$$

$$w_{ip}^{(s)} = w_{ip}^{(s-1)} + \lambda 2 \sum_{\tilde{j}=1}^{m} \left((y_{\tilde{j}}^{(s)} - \widetilde{y}_{\tilde{j}}^{(s)}) g_{p\tilde{j}}^{(s-1)} \right) \cdot T'(\sum_{\tilde{i}=1}^{n} w_{\tilde{i}p}^{(s-1)} x_{\tilde{i}}^{(s)} - \Theta_p^{(s-1)}) \cdot x_i^{(s)},$$

$$1 \leq i \leq n,\ 1 \leq p \leq q,$$

$$\Theta_p^{(s)} = \Theta_p^{(s-1)} - \lambda 2 \sum_{\tilde{j}=1}^{m} \left((y_{\tilde{j}}^{(s)} - \widetilde{y}_{\tilde{j}}^{(s)}) g_{p\tilde{j}}^{(s-1)} \right) \cdot T'(\sum_{\tilde{i}=1}^{n} w_{\tilde{i}p}^{(s-1)} x_{\tilde{i}}^{(s)} - \Theta_p^{(s-1)}), \quad 1 \leq p \leq q,$$

is called **backpropagation learning rule**. After learning is finished, the net has weights $\vec{w}_p = \vec{w}_p^{(t)}$, $\vec{g}_j = \vec{g}_j^{(t)}$, and threshold values $\Theta_p = \Theta_p^{(t)}$ for $1 \leq p \leq q$ and $1 \leq j \leq m$.

If the net does not work sufficiently on the training data after one learning cycle, learning can be repeated, as it is done in the perceptron learning rule. For a given $\epsilon > 0$ the network is trained as long until it satisfies

$$\sum_{s=1}^{t} \sum_{j=1}^{m} \left(y_j^{(s)} - \sum_{p=1}^{q} g_{pj} T(\sum_{i=1}^{n} w_{ip} x_i^{(s)} - \Theta_p) \right)^2 < \epsilon.$$

Motivated by the method of gradient descend, we will get a network that works correctly to an ϵ-error on the training data after a sufficient number of learning cycles.

5 A Realisation of Neural Networks in Matlab

5.1 Construction and Training

Matlab provides a neural network toolbox for the realisation of neural networks. In our description we will mainly follow the MathWorks' users's guide.[12] A slightly different construction of a network is used: In contrast to the notation we used before, the neural network toolbox does not refer to the input layer as a layer because typically its only function is to buffer the input signal. Thus a two-layer network in our notation is a one-layer network in Matlab, having one output layer. In the same way, a three-layer network in our notation is a two-layer network in Matlab, having one hidden layer and one output layer. A feedforward network with $n-1$ hidden layers is created in the following way:

A network object is created by the function *newff*. This function requires four inputs and returns the network object:

$$net = newff(MINMAX,\, [L1, L2, ..., Ln],\, \{T1, T2, ..., Tn\},\, T).$$

The inputs are

- $MINMAX$: an $L0 \times 2$-matrix containing minimum and maximum values for each of the $L0$ elements of the input vector.
- $[L1, L2, ..., Ln]$: an array where Li denotes the size of the ith hidden layer for $i = 1, ..., n-1$ and Ln denotes the size of the output layer.
- $\{T1, T2, ..., Tn\}$: a cell array containing the names of the transfer functions used in each layer.
- T: the name of the training function to be used.

The command even initialises the weights and threshold values of the network.

When we have constructed a feedforward neural network *net* as described above, we can simulate it by giving an input matrix to the net. The net then calculates an output matrix in the following way:

$$a = sim(net,p).$$

p is an $L0 \times R$ matrix containing the input values, where $L0$ determines the number of input neurons and R the sample size. The output matrix a has dimension $Ln \times R$.

For training the network, we have to give a matrix of input values p and a target matrix t to the network. This is done using the command

$$net = train(net,\, p,\, t),$$

[12] [Demuth Beale 2002]

where p is an $L0 \times R$ matrix and the target matrix t has dimension $Ln \times R$. By default, the network is trained for 100 epochs or until the mean square error between output and target is less then 0.01. We can change the training parameters by assigning new values to *net.trainParam.epochs* and *net.trainParam.goal*.

We will construct a simple example of a network considering the so called XOR-problem. XOR means exclusive or: For two logical operators x and y the value x XOR y is true, if exactly one of x and y is true. The truth-table is as follows:

x	0	0	1	1
y	0	1	0	1
XOR	0	1	1	0

We will construct a feedforward network having the following structure: We choose to have one hidden layer containing $L1 = 3$ hidden neurons. The number of input neurons is $L0 = 2$, the number of output neurons is $L2 = 1$. The sample size is $R = 4$. As transfer function for the hidden layer we choose the hyperbolic tangent function denoted by *tansig*, the output neuron has the identity function as transfer function, which is denoted by *purelin*. The network will be trained with backpropagation, which is denoted by *traingd*. Thus the input matrix is

$$p = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

and the target matrix is

$$t = \begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix}.$$

The function $minmax(p)$ calculates the matrix $\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$ containing the min and max values of each row of p.

We now construct the network *net* by

$$net = newff(minmax(p), [3, 1], \{\text{'}tansig\text{'}, \text{'}purelin\text{'}\}, \text{'}traingd\text{'}).$$

We simulate the net, which is initialised by random weights and threshold values,

$$a = sim(net, p)$$

and get as output matrix

$$a = [\text{-}1.4452 \quad \text{-}1.0245 \quad \text{-}1.4735 \quad \text{-}0.8000].$$

This is not the desired output, thus we have to train the network.

We choose

$$net.trainParam.epochs = 1000$$

and

$$net.trainParam.goal = 0.01.$$

The net will be trained for 1000 epochs or until the mean square error is less than 0.01. We now train the network

$$net = train(net,\ p,\ t).$$

After 448 iterations the mean square error is 0.00997415, thus the performance goal is reached and training is stopped. If we now simulate the network with input matrix p, the output is

$$a = [0.0989 \quad 0.9143 \quad 0.8577 \quad 0.0503],$$

which is close to the desired output.

As we see, it takes many epochs to train the network. This is a quite common problem to occur when backpropagation is used because the method of gradient descent is a method which converges rather slowly.

One possibility to get faster convergence is to use an adaptive step size, or learning rate, λ, as the optimal learning rate changes during the training process. The learning rate is adapted according to the error between output and target in each step. If the new error exceeds the old one by more than a certain predefined ratio, the learning rate is decreased. If the new error is less than the old one, the learning rate is increased. Backpropagation with adaptive learning rule is denoted by *traingda*. Applying this method to our problem, a mean square error of 0.00818517 is reached after only 74 epochs.

Another possibility is to use standard numerical optimisation techniques such as Conjugate Gradient, Quasi-Newton, or Levenberg-Marquardt.

In the method of **Conjugate Gradient** search is no longer performed along the direction in which the function decreases most but along conjugate directions. On the first iteration, the direction of steepest gradient is chosen. The next search direction is chosen in such a way that it is conjugate to the previous search directions. The direction is determined by combining the new steepest descent direction with the previous search direction. The different ways how this can be done result in different algorithms, e.g. the Fletcher-Reeves algorithm *traincgf* or the Polak-Ribire algorithm, denoted by *traincgp*.[13]
Applying the method of conjugate gradient to our problem, a mean square error of 0.00567906 is reached after four epochs using *traincgf*. Using *traincgp* training is stopped after three iterations when a mean square error of 0.00111612 is reached.

[13] for a more detailed description, see [Demuth Beale 2002] pp. 5-17 - 5-21.

Another method which converges fast is Newton's Method, which besides the gradient even considers the Hessian matrix when choosing search direction. Computing and inverting the Hessian matrix is, however, an expensive operation. Therefore it can be more effective to use methods which only approximate the Hessian matrix, called **Quasi-Newton Methods**. One such algorithm has been developed by Broyden, Fletcher, Goldfarb, and Shanno and is denoted by *trainbfg*.[14] Applied to our problem, a mean square error of 0.000554043 is reached after four iterations.

Even in the **Levenberg-Marquardt Method** the Hessian matrix is approximated. The method uses the Jacobian matrix, which contains the first derivatives of the network errors with respect to the weights and threshold values.[15] It is denoted by *trainlm*. In our problem the network needs only two training epochs to reach a mean square error of 0.0043434.

An algorithm which combines the basic idea of Levenberg-Marquardt with the conjugate gradient approach is the method of **Scaled Conjugate Gradient**, denoted by *trainscg*.[16] In our problem it takes 14 iterations to reach a mean square error of 0.00246505.

The number of epochs needed can vary depending on the initial network parameters. Anyhow, we can see a trend that some methods converge faster than others. Which method is the most suitable one depends on the given problem and has to be tested in each case.

How should training data be chosen to train a network as effectively as possible? The set of training data should be large enough to provide an adequate fit. On the other hand, when the set of training data is too large, so called overfitting can occur. The network then memorises the training examples but does not learn to generalise new data. When overfitting occurs, the training error decreases monotonically during training. When new cross-validation data are applied to the network after training, this error can be large. When monitored simultaneously, it can be noticed that the error of cross-validation data reaches a minimum early in training and then increases again, while the training error still decreases. Thus the training error is not always a reliable indicator of how good a network has learned to generalise. To find the point at which a network's generalisation ability is best, both training and cross-validation error should be considered.
In Matlab's neural network toolbox this is done in a method called *early stopping*. In this method data are divided into three subsets: a training set, a validation set and a test set. While the network is trained with the training set, the validation error is monitored, and when it increases for a specified number

[14] for a more detailed description, see [Demuth Beale 2002] pp. 5-26 - 5-27.
[15] ibid. pp. 5-28 - 5-31.
[16] ibid. pp. 5-22 - 5-23.

of iterations, training is stopped. The test set is used to evaluate the division of data. When the test set error and the validation error reach a minimum at a significantly different number of iterations, this can indicate a bad division of data.
Early stopping should be used with training algorithms that do not converge too rapidly. An algorithm which works well with early stopping is the scaled conjugate gradient method.

One question that remains is how to choose an optimal structure of a network for a given problem. The number of input and output neurons is mostly determined by the problem, the number of hidden neurons, however, is not a priori fixed. It is even possible to have more than one hidden layer, although it is very rare that a neural network has more than two hidden layers.[17]
In choosing the number of hidden neurons, the same problem as in choosing the size of training data can occur: If too few hidden neurons are used, the network may not be able to detect signals in a complicated data set. Using too many hidden neurons, however, can cause overfitting.
There are no clear rules how to optimise a network structure. In nearly all cases the optimal structure of a particular network has to be determined by experimentation. For finding the optimal number of hidden neurons there exist some rules of thumb, for example the Baum-Haussler rule:[18]

- The number of hidden neurons NH should be less than $NT * E/(NP + NO)$, where NT is the number of training examples, E is the error tolerance, NP is the number of input neurons, and NO is the number of output neurons.

This rule generally ensures that a neural network generalises rather than memorises.[19]

Searching for the best structure for a network in a given problem, we can proceed in the following way:
We start with a minimal number of hidden neurons, typically two. The resulting network is then trained and tested. In each step the number of hidden neurons is increased and training and testing of the network is repeated until the results of training and testing are not improved anymore.

5.2 Data Preprocessing

5.2.1 Normalisation

The XOR-problem we presented is a rather small example, both in sample size and in dimension of the input vector. In real problems the size of input data is often large. In this case it can be useful to preprocess data before applying

[17] [Heaton 2005] p. 129
[18] [Lin 1995] p. 3
[19] [Lin 1995] p. 3

them to the network to make training more effective.
If sample size is large, the inputs and targets can be scaled before training the network. This can be done in different ways. One approach is to normalise the means and standard deviations. This is implemented in the function *prestd* in the following way:

$$[pn,\ meanp,\ stdp,\ tn,\ meant,\ stdt] = prestd(p,t).$$

p and t are the matrices containing the original input and target values. Their means and standard deviations are given in the vectors *meanp*, *stdp*, *meant* and *stdt* respectively. *pn* and *tn* contain the normalised inputs and targets, having zero means and unity standard deviations.
When a network has been trained with normalised data and new inputs are applied to it, these should be preprocessed using the means and standard deviations that were computed for the training set. This is done with the function *trastd* in the following way:

$$dn = trastd(d,\ meanp,\ stdp).$$

The network will now produce a normalised output. To convert it back to its original units, we can use *poststd*:

$$an = sim(net,\ dn)$$
$$a = poststd(an,\ meant,\ stdt).$$

5.2.2 Principal Component Analysis

If the dimension of the input vector is large, it can be useful to reduce the dimension because the components can be highly correlated. This can be done using principal component analysis. In principal component analysis, data are performed in the following way: The components of the vectors are orthogonalised, so that they become uncorrelated with each other. They are then ordered by their size of variation. The components with least variation are eliminated, as they contribute least to the total variation of the data set.
In Matlab principal component analysis is realised in the function *precpa*. It requires a matrix of normalised input vectors and a fraction which denotes the minimum variance for the components to keep as arguments and returns the transformed data set and the transformation matrix:

$$[ptrans,\ transMat] = prepca(pn,\ 0.01).$$

When a network has been trained with normalised data that has been preprocessed with *precpa*, new inputs should, after normalisation, be transformed with the transformation matrix of the training set. This is done in the function *trapca*:

$$dntrans = trapca(dn,\ transMat).$$

5.3 Training Evaluation

After having trained a network, its performance can be analysed. For a first evaluation, the training, validation, and test set errors can be regarded. For a more detailed investigation, Matlab provides a function for performing regression analysis between the network response and its corresponding targets. The function *postreg* takes the output and the target vector, if the network has one output neuron, or one element of the network output, if there are more than one output neurons, and performs a linear regression. It returns the slope m and the y-intercept b of the linear regression as well as the correlation coefficient r between the outputs and the targets:

$$[m,b,r]=postreg(a,t).$$

The closer m is to 1 and b to 0 , the better is the fit. The correlation coefficient is a value between -1 and 1 and measures the linear association between the outputs and targets. A correlation value close to 1 indicates a strong linear relationship, which implies a good network performance.

6 An Application of Neural Networks

6.1 The Project NIVA^{B}

The German business enterprise TROUT GmbH develops a device for non-invasive measurement of blood glucose value, the Glucose Monitor, in cooperation with the Saint-Petersburg Polytechnic University. The aim is to be able to determine the blood glucose level by means of high and low frequency impedances and skin temperature instead of using conventional methods that require access to blood. The coherence between the glucose level and the conductivity of tissue regarding the temperature makes it possible to measure the blood glucose value continuously under a period of 24 hours after a one-time calibration with an invasively measured glucose value. In this way the pain and trouble in adequately monitoring the glucose level can be reduced to a minimum.

Calculating the glucose level from impedances and temperature is a problem of mapping correlated values where the explicit function is unknown. But this is exactly what neural networks are able to do. In fact, they have given the best results when different calculation methods were tested for the project NIVA^{B}.[20]

In the following we will shortly describe the system and the calculation program, which is implemented in Matlab. Based on this we will then develop a simplified version of a neural network program for calculating the glucose value to demonstrate the process of constructing, training, and simulating a neural network.

How does the system work?

The Glucose Monitor measures high and low frequency impedances and skin temperature ten times per minute and saves it in binary code. Data are then converted into txt format and finally average values for each minute are saved in an excel file. For training and calibration purposes even invasive glucose values are measured at some points of time and have to be entered. Furthermore ingestion events can be reported, as well as insulin ingestion for diabetic patients. The Matlab program reads data from the excel file. First of all the data are preprocessed by different statistical methods to reduce the influence of measurement errors and make data more smooth. Then neural networks are created. Optionally feedforward networks with Levenberg-Marquardt training method or recurrent Elman networks can be chosen. Both networks have the same structure of two hidden layers containing five and three neurons respectively and an output layer containing one neuron. The hidden layers have hyperbolic tangent transfer function, the output layer has identity transfer function. We can choose if one single network or several networks are to be used. In the latter case nine identical networks are generated and trained. This is done because the quality of training results even depends on the net's initial weights and threshold values,

[20] [Voelkel 2007] p. 17

which are generated randomly.
As input data high and low frequency impedances, skin temperature, and optionally food, drink, and insulin events are used both as actual and delayed values. The resulting total of up to 23 input neurons is reduced with principal component analysis to between four and seven neurons.
The invasive glucose values serve as target data. To be able to use these values over the whole period of time interim values are calculated by means of spline interpolation.
Input and target data are normalised so that they have zero means and unity standard deviations. Training is then carried out and the training results are saved. After training, target and output values are postprocessed and compared to determine the quality of the resulting networks.

The networks can now be used for calculation. New input data are preprocessed using the precalculated means and standard deviations as well as the previously computed principal component transformation matrix. The seven networks with best training results are then simulated and output data are postprocessed. Their mean is calculated as resulting output value.

6.2 Construction and Testing of a Simplified Network Based on NIVA^{B}

We will now demonstrate the process of finding an optimal network for the given problem by implementing a simplified version of the Matlab program used for NIVA^{B}.[21] Based on the NIVA^{B} code we will construct an own program, which follows the rough structure of some parts of the NIVA^{B} program, and add some analysing tools for finding a network that works best for our purpose. The code of our simplified program can be found in the appendix.
In our network we will only use three input neurons determined by high and low frequency impedances and skin temperature. As target data invasive glucose values will be used.
We start by importing data from an excel file which contains data that were recorded within a clinical test for Saint-Petersburg University.

Data are then transformed to remove anomalies resulting from measurement errors. First we investigate input data for extreme values. We calculate the differences of consecutive data and define for each sample a threshold value as the maximum of the absolute differences divided by ten. We then investigate the differences for extreme values. If a value increases (respectively decreases) by more than the threshold value in one timestep and decreases (respectively increases) by more than 0.8 times the threshold value in one of the following ten timesteps, we interpret this as a measurement error and adjust these values in such a way that they do not differ by more than the threshold value.

[21] [NIVAB]

We then apply a median filter of order five to the input values for smoothing data. The median filter replaces each value by the median of the current value and the two previous and consecutive values.

Invasive glucose data are available only for some points of time. To receive target data for the entire interval between the first and the last invasive glucose value, we use natural cubic spline interpolation.

Input and target data are normalised so that they have means of zero and unity standard deviations. Finally, principal component analysis is carried out on input data. All three components remain; we thus have a network with three input and one output neurons. We can now create a feedforward network and train it with the preprocessed data.

First of all we want to determine an optimal architecture for our network. In each step we will create nine identical networks for a better evaluation. We will proceed as follows: We start with a minimal neural network structure having one hidden layer with two hidden neurons and successively increase the number of hidden neurons until we have reached a maximum of ten hidden neurons. To reduce the risk of overfitting, we will use early stopping when training the networks, as training method we will use the scaled conjugate gradient method as it works best with early stopping. After training we will perform regression analysis for each network to evaluate how good the network works. The nine networks will be ranked by their regression coefficients. We then consider the average of all nine regression coefficients and mean square errors between target and output values to compare the quality of the network architectures.
Then networks with two hidden layers will be tested. We will start with a minimal number of neurons in both layers and successively increase the number of neurons in the second hidden layer, until we have reached a maximum of totally ten hidden neurons. Then we will increase the number of neurons in the first hidden layer and iterate the process.
Finally we will choose the network architecture with the best regression coefficient.

For the network with the best architecture we will test different training methods. After each training we will analyse the network fit considering the average regression coefficient and mean square error to choose the training method with best fit.

All networks will be trained for a maximum of 1000 epochs and a goal of 0.1. We will use the hyperbolic tangent transfer function for the hidden layers and the identity transfer function for the output layer.

From the training results, which are shown in Table 1, we can see that we get the best fit for the networks that have two hidden layers with six and three neurons respectively. For these nets we will test the different training methods.

Table 1: Network structure

hidden neurons			
layer 1	layer 2	mean MSE	mean r
2	-	0.1583	0.9149
3	-	0.1405	0.9199
4	-	0.1213	0.9359
5	-	0.1036	0.9441
6	-	0.1085	0.9426
7	-	0.0992	0.9470
8	-	0.1008	0.9466
9	-	0.0995	0.9473
10	-	0.0991	0.9474
2	2	0.1489	0.9217
2	3	0.1785	0.9053
2	4	0.1513	0.9140
2	5	0.1525	0.9171
2	6	0.1467	0.9246
2	7	0.1591	0.9182
2	8	0.1660	0.9114
3	2	0.1315	0.9357
3	3	0.1286	0.9296
3	4	0.1333	0.9276
3	5	0.1074	0.9444
3	6	0.1108	0.9438
3	7	0.1106	0.9415
4	2	0.1269	0.9384
4	3	0.1139	0.9421
4	4	0.1193	0.9356
4	5	0.1107	0.9440
4	6	0.1058	0.9444
5	2	0.1177	0.9409
5	3	0.1080	0.9423
5	4	0.1069	0.9435
5	5	0.0993	0.9477
6	2	0.1101	0.9438
6	3	0.0994	0.9479
6	4	0.1105	0.9418
7	2	0.1220	0.9326
7	3	0.1020	0.9464
8	2	0.0992	0.9476

Table 2: Training methods

training method	mean MSE	mean r
traingd - backpropagation	0.2893	0.8679
traingda - backpropagation with adaptive learning rule	0.1707	0.9067
traincgf - method of conjugate gradient, Fletcher-Reeves algorithm	0.1207	0.9352
traincgp - method of conjugate gradient, Polak-Ribire algorithm	0.1142	0.9403
trainbgf - quasi-Newton method	0.1020	0.9449
trainlm - Levenberg-Marquardt method	0.0978	0.9472
trainscg - method of scaled conjugate gradient	0.1035	0.9454

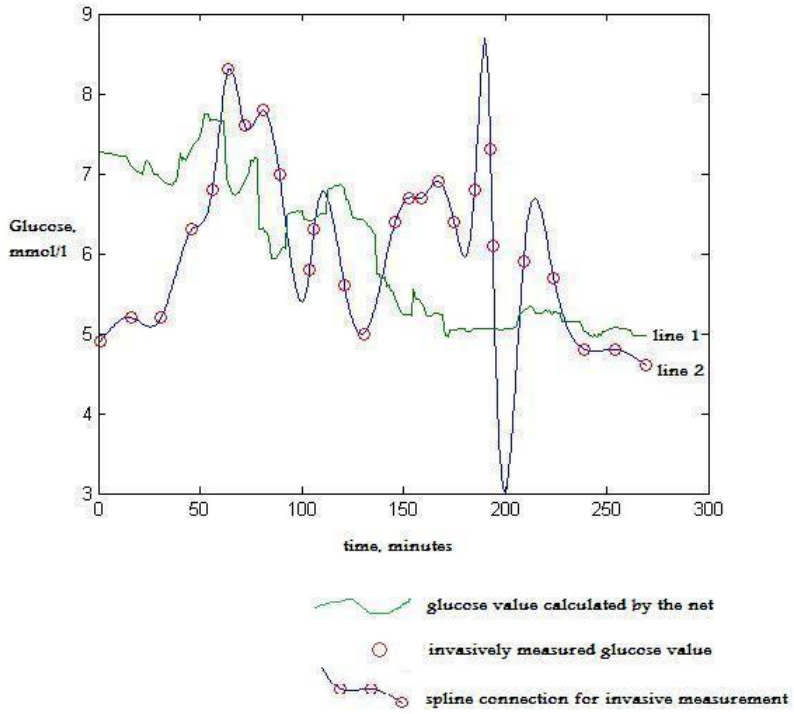
The results in Table 2 show that the Levenberg-Marquardt training method works best.

Analysing the training results we see that our optimal network structure is quite similar to the network structure in the NIVA^{B} project. Our simplified network has two hidden layers with six and three neurons respectively and is trained with the Levenberg-Marquardt method. Even the NIVA^{B} net is trained with the Levenberg-Marquardt method and contains five and three hidden neurons respectively in its two hidden layers. The slightly different number of neurons can be explained by the more extensive preprocessing of data in NIVA^{B}.

We will now test if our network can carry out adequate calculations from new input values. For this purpose, data from another excel file recorded for the same clinical test are imported. The data are preprocessed in the same way as before to remove anomalies resulting from measurement errors. Using the precalculated means, standard deviations, and transformation matrix, data are normalised and principal component analysis is carried out. The four nets with best training results are simulated with these data. The outputs are unnormalised and their average is calculated as the resulting glucose level. In Figure 1 the result is visualised. Line 1 shows the glucose level calculated by the neural network. The dots show the actual glucose values in mmol/l measured invasively. The connecting line 2 is created by interpolating the invasive glucose values.

We see that the glucose values calculated by the neural network approximate the invasive glucose values to a certain degree. The output does not completely coincide with the target values, however, the trend of the glucose level is largely similar.

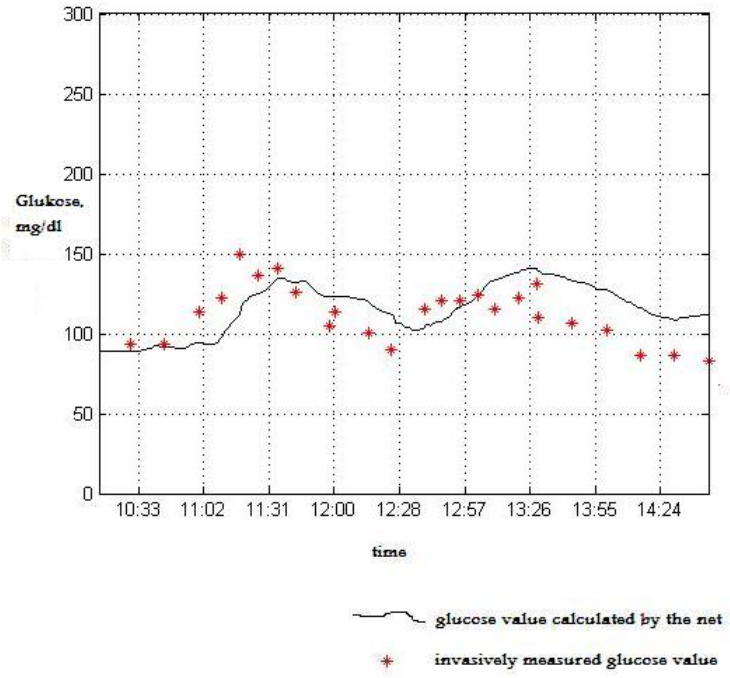
Figure 1: Simulation of the created network



In the NIVA^{B} network the approximation is clearly better as we can see in Figure 2.[22] The line shows the glucose values calculated by the network, the dots show the invasively measured glucose values in mg/dl, where 18.0182 mg/dl correspond to 1 mmol/l. The line does not totally coincide with the dots but the fit is much better than in our simplified network and the trend can clearly be seen.

[22]The figure is taken from the original NIVA^{B} program [NIVAB 2008]

Figure 2: Simulation of the NIVA^{B} network



6.3 Conclusion

Neither the network we constructed nor the original network in the project NIVA^{B} calculate the blood glucose level exactly, even though the NIVA^{B} network gives much better results. However, the trend of the glucose level can be approximated.
The inaccuracy can result from measurement errors and other external interferences. It can even be considered if the impedances and skin temperature are not sufficient to explain 100 percent of the blood glucose value. On the other hand, neither the invasively measured values give the exact blood glucose level as they are measured with test strips and hence only approximate the actual values.
The Glucose Monitor is under development and will be further improved to make non-invasive measurement of blood glucose more reliable. At this point of time it could already be applied as a tool for monitoring the blood glucose level in order to detect sudden and significant changes in the blood glucose level.
Another application could be to use it for the detection of diabetes mellitus type 2, where glucose cannot be reduced sufficiently within a certain amount of time. In this case, continuous measurement showing the trend of the blood glucose level can be a helpful tool.

A Matlab Program Code

NIVAnet.m

```
clear all;

%------ import data from excel file ---------
M = xlsread('T4-B A-2002-10-11-(04)-G0.xls');
%-------------------------------------------


%------ data preprocessing ------------------
[P,T,x0,y0] = transformData(M);
%-------------------------------------------


%------ network training --------------------
NTrain(P,T);
%-------------------------------------------
```

NTransformData.m

```
function [P,T,x0,y0] = NTransformData(M)

M(find(isnan(M(:)))) = 0;   % replace all empty entries by 0

[Mrow,Mcol] = size(M);   % size of matrix M
k = 0;
for i = 1:Mrow
    if(M(i,3) ~= 0)   % find invasive glucose entries
       k = k+1;
       x0(k) = i;   % save the line number in x0,
       y0(k) = M(i,3);   % save the respective glucose value in y0
    end;
end;

% x0(1) is the number of the first, x0(k) the number of the last line
% containing invasive glucose values
```

```
M1(1:(x0(k)-x0(1)+1),:) = M(x0(1):x0(k),:);   % save lines x0(1) to
% x0(k) of matrix M in M1


%------ input data --------------------------
M_in(:,1) = M1(:,2);
M_in(:,2) = M1(:,11);
M_in(:,3) = M1(:,12);
%-------------------------------------------


% ----- remove extreme values ---------------
for i = 1:3
    M_in(:,i) = NRemoveExtreme(M_in(:,i));
end;
%-------------------------------------------


%------ median filter -----------------------
for i = 1:3
    M_in(:,i) = medfilt1((M_in(:,i)), 5);
end;
%-------------------------------------------


%------ input vector ------------------------
P = M_in';
%-------------------------------------------


M1_out(:,1) = M1(:,3);
T0 = M1_out(:,1);   % T0 contains the invasively measured glucose
% values.  Where no value was measured, the entry is 0.


%------ target vector -----------------------
X = x0(1):x0(k);
pc = 0.999;
Y = csaps(x0,y0,pc,X);   % natural cubic spline
T = Y;   % T contains glucose values, approximated by a natural cubic
% spline
%-------------------------------------------
```

NRemoveExtreme.m

```
function Pt = NRemoveExtreme(P)

Pt = P;
Px = diff(P);   % Px contains the differences between adjacent elements
% of P
[nx,mx] = size(Px);

window = 10;
threshold = max(abs(Px)) * 0.1;   % the threshold value is the maximum
% absolute value of the differences divided by ten

for i = 1:nx
    isExtreme = 0;
    if (Px(i) > threshold)   % if any difference exceeds the threshold
% value ...
       beg = i;
       if (nx - i) < window
          kend = nx;
       else
          kend = i + window;
       end;
       for k = i:kend
          if (Px(k) < -threshold * 0.8)   % ... and if any of the following
% ten differences falls below -0.8 times the threshold value ...
             endi = k;
             isExtreme = 1;   % ... there is an extreme value due to
% measurement error
          end;
       end;
       if (isExtreme == 1)   % if there are extreme values ...
          if(beg > 1) beg = beg-1; end;
          if(endi < nx) endi = endi + 1; end;
          for j = beg:endi   % ... the differences that exceed (fall
% below) the threshold value in this window are replaced by the (negative)
% threshold value
             if(Px(j) > threshold)
                Px(j) = threshold;
             end;
             if(Px(j) < -threshold)
                Px(j) = -threshold;
             end;
          end;
          Pt(beg) = P(beg);
          for l = beg:endi
```

```
            Pt(l+1) = Pt(l) + Px(l);   % the original values are updated
% with the adapted differences
         end;
      end;
   end;
end;

for i=1:nx
   isExtreme = 0;
   if (Px(i) < -threshold)   % if any difference falls below the negative
% threshold value ...
      beg = i;
      if (nx - i) < window
         kend = nx;
      else
         kend = i + window;
      end;
      for k = i:kend
         if (Px(k) > threshold * 0.8)   % ... and if any of the following
% ten differences exceeds 0.8 times the threshold value ...
            endi = k;
            isExtreme = 1;   % ... there is an extreme value due to
% measurement error
         end;
      end;
      if (isExtreme == 1)   % if there are extreme values ...
         if(beg > 1) beg = beg - 1; end;
         if(endi < nx) endi = endi + 1; end;
         for j = beg:endi   % ... the differences that exceed (fall
% below) the threshold value in this window are replaced by the (negative)
% threshold value
            if(Px(j) > threshold)
               Px(j) = threshold;
            end;
            if(Px(j) < -threshold)
               Px(j) = -threshold;
            end;
         end;
         Pt(beg) = P(beg);
         for l = beg:endi
            Pt(l+1) = Pt(l) + Px(l);   % the original values are updated
% with the adapted differences
         end;
      end;
   end;
end;
```

NTrain.m

```
function res = NTrain(P,T)

res = 0;

%----- net and training parameter -----------
netsize = [6,3,1];
transfer = {'tansig','tansig','purelin'};
trainmethod = 'trainlm';
goal = 0.1;
epochs = 1000;
%--------------------------------------------


%------- normalisation ----------------------
[PN,meanP,stdP,TN,meanT,stdT] = prestd(P,T);
%--------------------------------------------


%------ pca --------------------------------
[PNC,transMat] = prepca(PN,0.01);   % principal component analysis
% on input data
%--------------------------------------------


[R,Q] = size(PNC);   % R is the number of input neurons, Q is the sample
% size

% data are divided into three subsets: training data, cross validation
% data, and test data
iitst = 2:4:Q;
iival = 4:4:Q;
iitr = [1:4:Q,3:4:Q];
val.P = PN(:,iival); val.T = TN(:,iival);
test.P = PN(:,iitst); test.T = TN(:,iitst);
ptr = PN(:,iitr); ttr = TN(:,iitr);

% nine nets are created and trained with early stopping
% the output of net i is saved in A(i,:)
```

```
%------ net 1 -------------------------------
GlucNet1 = newff(minmax(PNC),netsize,transfer,trainmethod);
GlucNet1.trainParam.epochs = epochs;
GlucNet1.trainParam.goal = goal;
GlucNet1 = train(GlucNet1,ptr,ttr,[],[],val,test);
AN1 = sim(GlucNet1,PN);
A(1,:) = poststd(AN1,meanT,stdT);
[GlucNet1,y,e,pf] = adapt(GlucNet1,ptr,ttr);
delta(1) = mse(e);
%-----------------------------------------


%------ net 2 -------------------------------
GlucNet2 = newff(minmax(PNC),netsize,transfer,trainmethod);
GlucNet2.trainParam.epochs = epochs;
GlucNet2.trainParam.goal = goal;
GlucNet2 = train(GlucNet2,ptr,ttr,[],[],val,test);
AN2 = sim(GlucNet2,PN);
A(2,:) = poststd(AN2,meanT,stdT);
[GlucNet2,y,e,pf] = adapt(GlucNet2,ptr,ttr);
delta(2) = mse(e);
%-----------------------------------------


%------ net 3 -------------------------------
GlucNet3 = newff(minmax(PNC),netsize,transfer,trainmethod);
GlucNet3.trainParam.epochs = epochs;
GlucNet3.trainParam.goal = goal;
GlucNet3 = train(GlucNet3,ptr,ttr,[],[],val,test);
AN3 = sim(GlucNet3,PN);
A(3,:) = poststd(AN3,meanT,stdT);
[GlucNet3,y,e,pf] = adapt(GlucNet3,ptr,ttr);
delta(3) = mse(e);
%-----------------------------------------


%------ net 4 -------------------------------
GlucNet4 = newff(minmax(PNC),netsize,transfer,trainmethod);
GlucNet4.trainParam.epochs = epochs;
GlucNet4.trainParam.goal = goal;
GlucNet4 = train(GlucNet4,ptr,ttr,[],[],val,test);
AN4 = sim(GlucNet4,PN);
A(4,:) = poststd(AN4,meanT,stdT);
[GlucNet4,y,e,pf] = adapt(GlucNet4,ptr,ttr);
delta(4) = mse(e);
```

```
%-----------------------------------------


%------ net 5 -------------------------------
GlucNet5 = newff(minmax(PNC),netsize,transfer,trainmethod);
GlucNet5.trainParam.epochs = epochs;
GlucNet5.trainParam.goal = goal;
GlucNet5 = train(GlucNet5,ptr,ttr,[],[],val,test);
AN5 = sim(GlucNet5,PN);
A(5,:) = poststd(AN5,meanT,stdT);
[GlucNet5,y,e,pf] = adapt(GlucNet5,ptr,ttr);
delta(5) = mse(e);
%-----------------------------------------


%----- net 6 -------------------------------
GlucNet6 = newff(minmax(PNC),netsize,transfer,trainmethod);
GlucNet6.trainParam.epochs = epochs;
GlucNet6.trainParam.goal = goal;
GlucNet6 = train(GlucNet6,ptr,ttr,[],[],val,test);
AN6 = sim(GlucNet6,PN);
A(6,:) = poststd(AN6,meanT,stdT);
[GlucNet6,y,e,pf] = adapt(GlucNet6,ptr,ttr);
delta(6) = mse(e);
%-----------------------------------------


%------ net 7 -------------------------------
GlucNet7 = newff(minmax(PNC),netsize,transfer,trainmethod);
GlucNet7.trainParam.epochs = epochs;
GlucNet7.trainParam.goal = goal;
GlucNet7 = train(GlucNet7,ptr,ttr,[],[],val,test);
AN7 = sim(GlucNet7,PN);
A(7,:) = poststd(AN7,meanT,stdT);
[GlucNet7,y,e,pf] = adapt(GlucNet7,ptr,ttr);
delta(7) = mse(e);
%-----------------------------------------


%------ net 8 -------------------------------
GlucNet8 = newff(minmax(PNC),netsize,transfer,trainmethod);
GlucNet8.trainParam.epochs = epochs;
GlucNet8.trainParam.goal = goal;
GlucNet8 = train(GlucNet8,ptr,ttr,[],[],val,test);
AN8 = sim(GlucNet8,PN);
A(8,:) = poststd(AN7,meanT,stdT);
```

```
[GlucNet8,y,e,pf] = adapt(GlucNet8,ptr,ttr);
delta(8) = mse(e);
%-------------------------------------------


%------ net 9 -------------------------------
GlucNet9 = newff(minmax(PNC),netsize,transfer,trainmethod);
GlucNet9.trainParam.epochs = epochs;
GlucNet9.trainParam.goal = goal;
GlucNet9 = train(GlucNet9,ptr,ttr,[],[],val,test);
AN9 = sim(GlucNet9,PN);
A(9,:) = poststd(AN9,meanT,stdT);
[GlucNet9,y,e,pf] = adapt(GlucNet9,ptr,ttr);
delta(9) = mse(e);
%-------------------------------------------


%----- regression analysis ------------------
for i=1:9
    [m(i),b(i),r(i)] = postreg(A(i,:),T);
end;
%-------------------------------------------


% the correlation coefficient between the outputs and targets of net
% i is saved as row i of matrix r

AverageMES = mean(delta)   % the avarage mean square error of the nine
% networks in shown
AverageR = mean(r)   % the avarage regression coefficient of the nine
% networks in shown


%----- the nets are ranked by correlation ---
r1 = r; r1 = sort(r); dop = r1(5);
k = 0;
for j = 1:9
    if(r(j) < dop)
       k = k + 1;
       NumNet(k) = j;
    end;
end;
%-------------------------------------------


save NumNet NumNet   % the numbers of the four best nets are saved
```

```
%------ save all nets ------------------------
save GlucNet1 GlucNet1;
save GlucNet2 GlucNet2;
save GlucNet3 GlucNet3;
save GlucNet4 GlucNet4;
save GlucNet5 GlucNet5;
save GlucNet6 GlucNet6;
save GlucNet7 GlucNet7;
save GlucNet8 GlucNet8;
save GlucNet9 GlucNet9;
%-----------------------------------------


%------ save preprocessing data -------------
save meanP meanP;
save stdP stdP;
save meanT meanT;
save stdT stdT;
save transMat transMat;
%-----------------------------------------


res = 1;
```

References

[Astion 1983] M. L. Astion et al., *Overtraining in Neural Networks That Interpret Clinical Data*, Clinical Chemistry, Vol. 39 No. 9, 1983.

[Berns Kolb 1994] K. Berns, T. Kolb, *Neuronale Netze für technische Anwendungen*, Springer-Verlag, 1994.

[Demuth Beale 2002] H. Demuth, M. Beale, *Neural Network Toolbox User's Guide*, TheMathWorks, 2002.

[Grauel 1992] A. Grauel, *Neuronale Netze*, B.I. Wissenschaftsverlag, 1992.

[Grimaldi 2004] R. P. Grimaldi, *Discrete and Combinatorial Mathematics*, Pearson Education, 2004.

[Heaton 2005] J. Heaton, *Introduction to Neural Networks with Java*, Heaton Research, 2005.

[Hebb 1949] D. O. Hebb, *The Organization of Behavior*, Wiley, 1949.

[Hopfield 1982] J. J. Hopfield, *Neural Networks and Physical Systems with Emergent Collective Computational Abilities*, Proceedings National Academy of Science 79, pp. 2554-2558, 1982.

[Koehle 1990] M. Koehle, *Neurale Netze*, Springer-Verlag, 1990.

[Lenze 2003] B. Lenze, *Einführung in die Mathematik neuronaler Netze*, Logos Verlag Berlin, 2003.

[Lin 1995] F. Lin et al., *Time Series Forecasting with Neural Networks*, Complexity International, Volume 2, 1995.

[Lundgren Roennqvist Vaerbrand 2003] J. Lundgren, M. Rönnqvist, P. Värbrand, *Optimeringslära*, Studentlitteratur, 2003.

[Malychina 2008] G. Malychina, *NeuralNetwork Based Algorithm for the Blood Glucose Level Measurement with the Group Parameter Estimation and Binary Glucose-Significant Events*, Saint-Petersburg Polytechnic University, 2008.

[McCulloch Pitts 1943] W. S. McCulloch, W. Pitts, *A logical calculus of the ideas immanent in nervous activity*, Bulletin of Mathematical Biophysics 5, pp. 115-137, 1943.

[Nelson Illingworth 1991] M. McCord Nelson, W. T. Illingworth, *A Practical Guide to Neural Nets*, Addison Wesley, 1991.

[NIVAB 2008] Program Code of the Project NIVA^{B}, Version 8.0, G. Malychina, Saint-Petersburg Polytechnic University, 2008.

[Rosenblatt 1958] F. Rosenblatt, *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain*, Psychological Review 65, pp. 386-408, 1958.

[Rumelhart Hinton Williams 1986] D. E. Rumelhart, G. E. Hinton, R. J. Williams, *Learning Internal Representations by Error Propagation*, Parallel Distributed Processing 1, pp. 318-362, 1986.

[Sanchez-Sinencio Lau 1992] E. Sanchez-Sinencio, C. Lau, eds, *Artificial Neural Networks*, IEEE Press, 1992.

[Voelkel 2007] A. Völkel, *Beschreibung NIVA^B*, Version 06, TROUT GmbH, 2007.

[Werbos 1974] P. J. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, Ph.D. Thesis, Harvard University, Committee on Applied Mathematics, 1974.