A Modified Neural Gas Network Learns Online Unevenly Distributed Data from an Accumulator without Training Sets

Daniel Kray, Dirk Uwe Sauer
Fraunhofer Institute for Solar Energy Systems
Oltmannsstr. 5, D-79100 Freiburg Germany
Phone: +49 (0) 7 61 45 88 21 9, Fax +49 (0) 7 61 45 88 21 7
e-mail: sauer@ise.fhg.de

ABSTRACT: We present a software for data analysis that uses modified neural gas networks. Initially designed to provide a stable and easy-to-use tool for energy management systems, the software can be used for multidimensional approximations of unknown functional relationships. In this paper, the modified neural gas algorithm and a validation of the software with mathematical and real world data is presented. The practical application, the state-of-charge estimation of a lead-acid accumulator in a grid-independent photovoltaic installation, is especially challenging because of very inaccurate data.

KEYWORDS: neural networks, neural gas, unsupervised learning, renewable energies, real world applications, modeling of multidimensional functional relationships

1 Neural Networks for Data Analysis

In numerous applications there are huge sets of multidimensional data that must be analysed to be able to make the right decisions. Often the functional relationship between input and output data is highly non-linear or can even not be represented analytically. Artificial neural networks can be used to model such relationships in order to provide look-up-tables for the given application.

Therefore, one needs a problem-independent toolbox that can approximate functional relationships between m-dimensional input vectors and n-dimensional output vectors. There should also be a mechanism that takes into consideration measurement errors and inaccurate input vectors. Because we want our model to work with sequential data rather than with training sets, self-organization is required and the user is not forced to store large amounts of training data. In some cases, i.e. microprocessor applications, there is also not enough memory to store recent data, so our approach is even appliable to problems with very restricted hardware ressources.

At Fraunhofer ISE, a suitable software *NeuroToolBox* [Kray (1999)] has been developed to cope with such problems. One can treat problems of the kind

$$\mathbb{R}^m \longrightarrow \mathbb{R}^n, \quad \mathbf{x} \stackrel{?}{\longrightarrow} \mathbf{y}$$

by easy-to-use ANSI-C-functions. This functional relationship may explicitly vary with time by ageing effects e.g. We will first present the modified neural gas network type before we will discuss applications on mathematical and real world problems.

2 The Modified Neural Gas Network

The network model used in the *NeuroToolBox* is based on the original algorithm presented by Martinetz *et al.* [Martinetz (1991), Walter (1993)]. We use additionally modifications of Demartines [Demartines (1994)] and finally three ideas of the authors.

2.1 The Original Algorithm

The neural gas model prevents the known disadvantages of Kohonen's self-organizing maps where a shape of the Kohonen layer must be fixed a priori: No structure at all must be fixed at the beginning and the neighbourhood

relationships of the neurons are calculated anew at each time step according to the input vector ξ. The neurons are sorted in a linear way according to the euclidian distances δ_k of their weight vectors $\mathbf{x_k}$ to the input vector.

$$\delta_0 < \delta_1 < \ldots < \delta_k < \delta_{k+1} < \ldots < \delta_{N-1}$$

with

$$\delta_k = \|\xi - \mathbf{x_{i(k)}}\|$$

Each neuron $\mathbf{i}$ is attributed a unique *rang* $\mathbf{k(i)} \in I\!N$ where the *winner*, i.e. the nearest neuron, gets rang 0 and the *loser* gets rang $N-1$ if the network consists of N neurons. $i(k)$ denotes the index of the neuron with rang k, where $k(i)$ means the rang of neuron i.

Once the rangs are calculated, the *learning rule* adapts the weight vectors:

$$\mathbf{\Delta x_i} = \alpha(t) \cdot \exp\left(-\frac{k(i)}{\lambda(t)}\right) \cdot (\xi - \mathbf{x_i}) \tag{1}$$

As one can see in formula (1), the winner makes the largest adaption step but all other neurons are adapted too, so we speak of a *winner-takes-most* algorithm.

The learning parameters α (*learning rate*, $0 < \alpha \leq 1$) and λ (*"neighbourhood radius"*, $\lambda > 0$) manage the adaption process: High values of α (near 1.0) make the winner move very 'fast' towards the learning vector whereas high values of λ (greater 30-50) allow very 'global' movements of the neurons weight vectors.

Because the components of all weight vectors are randomly initialized it is clear that at the beginning of the learning process one wants to allow large adaptions steps. On the other hand, only slight changes in the network should be made after a considerable number of adaption steps. This is why Martinetz *et al.* realized an evolution of α and λ according to the number of training steps t:

$$r(t) = r_i \cdot \left(\frac{r_f}{r_i}\right)^{t/t_{max}}$$

In this formula, r stands for α or λ and the indices i and f mean initial and final values for the variables. So, one musts fix initial and final values for α and λ that evolute continuously from r_i at $t = 0$ to r_f at $t = t_{max}$.

Recalling formula (1), a point that should be stressed is that the rangs are natural numbers that are different for each two neurons. Therefore the adaption steps $\mathbf{\Delta x_i}$ are different for each two neurons at each time step. So the problem of 'glued' neurons can efficiently be avoided: Neurons i and j with the same weight vectors at $t = t_0$ (a possible situation) are imperatively separated at the next time step since α, λ, ξ, $\mathbf{x_i}$ and $\mathbf{x_j}$ are the same for both, but their rangs are different and so are the adaption steps calculated by (1).

2.2 Modifications of Demartines

In his dissertation, Demartines [Demartines (1994)] proposed several modifications of the original algorithm to cope with different problems:

- To avoid the occurence of socalled *dead units*, never activated neurons outside the learning areas, Demartines introduced a *fatigue mechanism* that faciliates the activation of neurons 'out-of-area'.
- Calculation time can be cut considerably if the number of adapted neurons is restricted. So Demartines proposed a time-dependent limit K of neurons to be adapted at each time step.
- In the case one doesn't know much about the considered problem, the parameter t_{max} can eventually not be properly set. Thus the flexibility of the network is reduced too fast or too slow what results in poor approximation quality. Demartines introduced an *automatic parameter adaption scheme* that detects if the learning vector distribution change and increases the parameters α and λ therefore.

We will analyse these modifications in the following.

2.2.1 Fatigue Mechanism

In biological neural networks, neurons have a dead time that describes the time interval after an activation in which incoming signals cannot be processed. The units must 'relax' before another activation is accepted.

Demartines contributed therefore a *potential* $p_i \in [0,1]$ to each neuron which is randomly intialized and adapted at each time step following

$$p_{i,new} = p_{i,old} + \frac{1}{\tau}\left(G(i) - p_{i,old}\right)$$

Here $G(i) = \exp(-k(i)/\lambda(t))$ means the gain from the learning rule and τ is a time constant which manages the convergence speed of the potentials. As one can see, the potential is not changed when it is equal to this gain, the second term on the right side is zero then. So the potential of each neuron converges towards its mean gain: neurons that are often activated have relatively high mean gains and therefore high potentials whereas the dead units will have relatively small potentials.

The rang distribution process is now changed according to this fact: Instead of sorting the euclidian distances δ_k of the neurons weight vectors to the input vector, we multiply these with the potentials and sort afterwards:

$$q_0 < q_1 < \ldots < q_k < q_{k+1} < \ldots < q_{N-1}$$

with

$$q_k = p_{i(k)} \|\xi - \mathbf{x_{i(k)}}\|.$$

So seldomly activated units are favoured in the process and no dead units are observed anymore.

2.2.2 Reduction of Adapted Neurons

From the learning rule (1) we can derive that neurons with rangs $k(i) \gg \lambda$ are adapted only very slightly. To speed up calculation, Demartines proposed to adapt only the neurons with rangs $< 3 \cdot \lambda + 1 = K$.

If one uses the wide-spread *quicksort* algorithm which has a complexity of $O(N \log N)$, the necessary calculation steps are reduced to $O(K \log N)$. This is particulary interesting for small values of K.

2.2.3 Automatic Parameter Adaption

The irreversible time evolution of α and λ from their initial values at $t = 0$ to their final values at $t = t_{max}$ requires thorough optimization for t_{max}. If this parameter is not properly set, the network might 'freeze' before the organization process has finished. On the other hand it might as well stay flexible too long and concentrate always on small areas of the learning space.

Demartines derived therefore from his potential values an efficient alternative for the evolution rule of α and λ. The base for this alternative is the consideration that the goal for the network is an equilibrated distribution of the neurons. In such a state, all neurons have the same potential since they are all activated with the same probability. If this organization process is not completed yet, the potentials vary in a wide range. As an indicator for the 'order state' of the network, Demartines used

$$P = \frac{\min(p_i)}{\max(p_i)}$$

If one single neuron has a relatively low potential, then the value for P will be low, too. This is why this term has been favoured over the standard deviation of the potentials.

With the indicator P we can realize a parameter evolution for α and λ:

$$r(t) = r_i \cdot \left(\frac{r_f}{r_i}\right)^{P/P_0}$$

P_0 is a kind of 'order yield' ($0 < P_0 \leq 1$) which is the target value for P. While the current value of P is below P_0, the network parameters are high and large adaption steps are allowed. Once P becomes similar to P_0, α and λ converge to their final values and the network becomes rigid – the organization process is nearly completed.

The most important advantage of this parameter adaption is the reaction of the network when the learning set changes fundamentally: Since one single neuron can change the value for P, the network flexibility returns dramatically fast when the first new input vectors are presented. This is a feature that cannot be realized with the original algorithm and which is very useful in most applications – the painful optimization of t_{max} is omitted.

2.3 Additional Modifications

Because the *NeuroToolBox* was designed for real world applications, we use two additional modifications to provide a possibility to introduce expert knowledge into the learning process. We also use another term taken from the *competetitive learning with regularization* (CLR) model from Demartines [Demartines (1994)].

2.3.1 Regularization

In many neural network applications a kind of 'border effect' occurs: When one works with a circular training set with equally distributed training data, the neurons of the trained network tend towards the inside of the circle so that the border is the worst-approximated area.

To reduce this problem, we introduced the socalled *regularization* in our network model which is part of the CLR-model by Demartines. The idea of the regularization is to minimize the variance of the distances of neighbouring neurons to the winner *without changing the mean distance*. Demartines used his *dynamical links* (that we will not discuss here) to determine what neurons are 'in the neighbourhood' of the winner, see figure 1. The reduction of the border effect can clearly be seen.

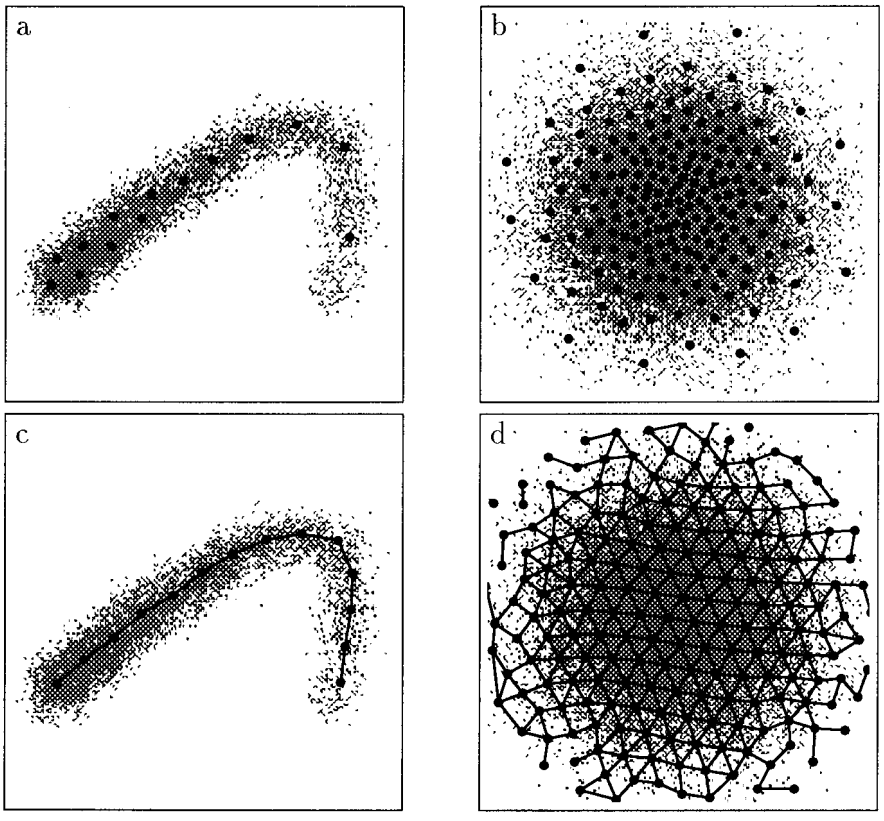

Figure 1: Comparison between the CLR model with deactivated (a and b) and activated (c and d) regularization. The regularization parameter has been fixed to $\gamma = 0.5$. Left side: The density of learning vectors varies exponentially along the arc. From [Demartines (1994)].

These links are not yet implemented into our software, so we try to organize the neurons on the corners of hypertetraeders by using the $W + 1$ nearest neurons to the winner where W is the intrinsic dimension of the problem that must be specified. Figure 2 explains the goal of the regularization.

Mathematically, we introduce an additional term into our learning rule. If V_i denotes the set of involved neurons

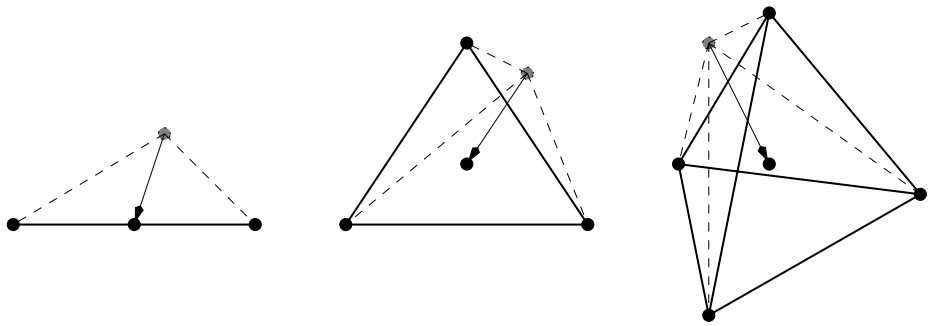

Figure 2: The goal of the regularization depending on the intrinsic dimension W

(i.e. the $W+1$ nearest neurons to the winner) and $N_i = \mathrm{card}(V_i) = W+1$ then the variance of the distances of these neurons to the winner can be written as

$$\begin{aligned} \mathrm{Var}_{j\in V_i}(\|\mathbf{x_i}-\mathbf{x_j}\|) &= \mathrm{E}_{j\in V_i}\left[(\|\mathbf{x_i}-\mathbf{x_j}\|-\mu_i)^2\right] \\ &= \frac{1}{N_i}\sum_{j\in V_i}(\|\mathbf{x_i}-\mathbf{x_j}\|-\mu_i)^2 \end{aligned} \tag{2}$$

Here i denotes the index of the winner neuron and μ_i the mean value for the distances $\|\mathbf{x_i}-\mathbf{x_j}\|$. The extra adaption vector $\mathbf{r_i}$ for the winner should now be the negative gradient of the variance in order to minimize this value

$$\begin{aligned} \mathbf{r_i} = -\frac{\partial}{\partial \mathbf{x_i}}\mathrm{Var}_{j\in V_i}(\|\mathbf{x_i}-\mathbf{x_j}\|) &= -\frac{\partial}{\partial \mathbf{x_i}}\frac{1}{N_i}\sum_{j\in V_i}(\|\mathbf{x_i}-\mathbf{x_j}\|-\mu_i)^2 \\ &= \frac{2}{N_i}\sum_{j\in V_i}(\|\mathbf{x_i}-\mathbf{x_j}\|-\mu_i)\frac{\mathbf{x_j}-\mathbf{x_i}}{\|\mathbf{x_j}-\mathbf{x_i}\|} \end{aligned} \tag{3}$$

In this calculation, μ_i has been supposed to be only slowly changing within the process so that it has not to be derived.

Finally, Demartines introduced a regularization parameter γ $(0 \le \gamma \le 0.5)$ that fixes the fraction of $\mathbf{r_i}$ to be added to the winner's weight vector in addition to the standard learning rule:

$$\begin{aligned} \mathbf{\Delta x_i} &= \alpha(t)\cdot\exp\left(-\frac{k(i)}{\lambda(t)}\right)\cdot(\xi-\mathbf{x_i})+\gamma(t)\cdot\mathbf{r_i} \\ &= \alpha(t)\cdot\exp\left(-\frac{k(i)}{\lambda(t)}\right)\cdot(\xi-\mathbf{x_i})+\gamma(t)\cdot\frac{2}{N_i}\sum_{j\in V_i}(\|\mathbf{x_i}-\mathbf{x_j}\|-\mu_i)\frac{\mathbf{x_j}-\mathbf{x_i}}{\|\mathbf{x_j}-\mathbf{x_i}\|} \end{aligned} \tag{4}$$

γ can change with the time but can also be held constant as in the *NeuroToolBox*.

2.3.2 Input Accuracies

In order to represent inaccurate measurements from real installations less than accurate data, we provide a possibility to contribute 'accuracy' values to the input vectors. In many applications experts know their systems so well that they can tell for each measured data if it's more or less accurate.

Especially in the accumulator application that we will discuss later, the accuracy of the training data is highly uneven and should absolutely be taken into account. We realize this by multiplying the network parameters α, λ and K by the input accuracy $0\% \le acc_{in} \le 100\%$ *for the single training step.* So less accurate data cause only weak and local adaption processes of the network and do not have large impact on the approximation quality.

If one fixes acc_{in} to 100%, the feature is switched off.

2.3.3 Individual Learning Factors

Often the operation of a system consists of 90% 'business-as-usual' and 10% extraordinary system states. Perhaps you want just these states to be represented exactly by the network. This is the aim of the *learning factor* that we implemented. Each input vector gets besides its accuracy a learning factor LF greater 0 that is an indicator of its importance.

In our network model, the parameters α, λ and K are additionally multiplied by LF before the learning rule is applied. Naturally, α is limited to 1 and the multiplication is restricted to the single adaption step.

By this mecanism, expert knowledge of the system can easily be introduced in the learning process. A constant value of $LF = 1$ brings back the original algorithm if the distinction is not wanted or needed.

In the accumulator application, intensive efforts must be done to calculate the appropriate learning factor for each training data. But this is the key for successfull state-of-charge approximation, the equal use of all measured data leads to very poor approximations.

3 The NeuroToolBox Software

The software that we want to present here realizes the modified neural gas network model which we discussed in the preceding section within a lean ANSI-C code that can be compiled on many hardware platforms. We summarize the learning steps and adaption terms:

1. Presentation of *learning vector* ξ to the network.
2. Calculation of the *rangs* $\{k(i)|i = 0, \ldots, N-1\}$ by sorting
$$q_k = p_{i(k)} \|\xi - x_{i(k)}\|$$
3. *Accuracies* acc_{in} and *learning factor* LF_{in} of the input vector change the network parameters for the single training step:
$$(\alpha,\ \lambda,\ K) \longrightarrow acc_{in} \cdot LF_{in} \cdot (\alpha,\ \lambda,\ K)$$
4. Application of the *learning rule*:
$$x_i \longrightarrow x_i + \alpha(t) \cdot \exp\left(-\frac{k(i)}{\lambda(t)}\right) \cdot (\xi - x_i)$$
5. *Regularization* of the winner
$$x_i \longrightarrow x_i + \gamma(t) \cdot \frac{2}{N_i} \sum_{j \in V_i} (\|x_i - x_j\| - \mu_i) \frac{x_j - x_i}{\|x_j - x_i\|}$$
with constant γ.
6. Evolution of α and λ:
$$r(t) = r_i \cdot \left(\frac{r_f}{r_i}\right)^{t/t_{max}} \quad \text{(Martinetz in [Martinetz (1991)])} \quad \text{or}$$
$$r(t) = r_i \cdot \left(\frac{r_f}{r_i}\right)^{P/P_0} \quad \text{(Demartines in [Demartines (1994)])}$$
$$\text{with } P = \frac{\min(p_i)}{\max(p_i)}$$
7. Evolution of the *potentials* of the neurons
$$p_{i,neu} = p_{i,alt} + \frac{1}{\tau}(G(i) - p_{i,alt})$$

3.1 Capabilities of the *NeuroToolBox*

The software has been developed to approximate unknown functional relationships between m input variables and n output variables.

$$f: \ \mathbb{R}^m \longrightarrow \mathbb{R}^n, \ \mathbf{x} \longrightarrow \mathbf{y}$$

We have therefore a m-dimensional *input space*, a n-dimensional *output space* and a $(m+n)$-dimensional *weight space* for the weight vectors of the neurons.

Training data is presented to the network by the means of complete weight vectors in $\mathbb{R}^{m+n}$ consisting of input and output components. When the user requests an approximation, he specifies an input vector in $\mathbb{R}^m$ and gets an output vector in $\mathbb{R}^n$ from the network. The two functions of the *NeuroToolBox* are summarized in figure 3.

- *Learning* of weight vectors, $\mathbf{w} = (\mathbf{x}, \mathbf{y}) \in \mathbb{R}^{m+n}$.
- *Estimation* of output vectors $\tilde{\mathbf{y}} \in \mathbb{R}^n$ for input vectors $\mathbf{x} \in \mathbb{R}^m$.

Figure 3: Functions of the *NeuroToolBox*

To speed up the optimization work and to enlarge the application possibilities, the user can work with *pools* of networks. These structures allow the work with many networks simultaneously for parallel parameter testing or for different focusing areas in the learning set. This feature can be very useful especially when large amount of learning vectors must be trained.

Since the network provides only a cloud of neurons that should 'enclose' the graph of the unknown function, extrapolation algorithms are required to give continuous estimations. In the *NeuroToolBox* there are two different methods: hyperplane extrapolation and estimation by weighted mean values.

In regions with low neuron density, not enough neurons may be present to calculate an extrapolating hyperplane, thus weighted mean values must be calculated. The second method is also implemented because one can restrict the range of linear extrapolation in very 'lively' functions.

3.1.1 A simple application

To demonstrate the approximation possiblities of the *NeuroToolBox*, we tried to model the following spiral function for $0 \leq x \leq 2\pi$.

$$f(x) = \begin{pmatrix} \sin x \\ \cos x \end{pmatrix}$$

For this purpose, we must write a little C-program (here approx. 15 relevant lines) which sets up the parameters of the networks and calls the appropriate functions. We trained (in this first approach) 10.000 learning vectors with random x values and visualized afterwards the *NeuroToolBox* estimation for $0 \leq x \leq 2\pi$, see figure 5. The calculation time was approximately 50 seconds on an Apple™ Macintosh™ G3.

To demonstrate the usefulness of Demartines parameter evolution scheme, we used two networks with identical parameter sets but with the two different parameter evolutions. As can be seen in figure 4, the weight vectors of the first network (with Demartines scheme) are perfectly placed on the spiral whereas the second network still needs a serious optimization. It is the parameter t_{max} which has not yet been optimized but whose optimal value will surely be found after some time. As t_{max} describes the amount of training steps until the network gets rigid it is obvious that the chosen value is too low – the network froze before perfect organization has been achieved.

It must be stressed, that these results were produced using typical parameter settings without any optimization within a very short time. This underlines the easy-to-use aspect of the software that provides several predefined networks for standard applications.

We will now show how the *NeuroToolBox* can be included in complex real world systems.

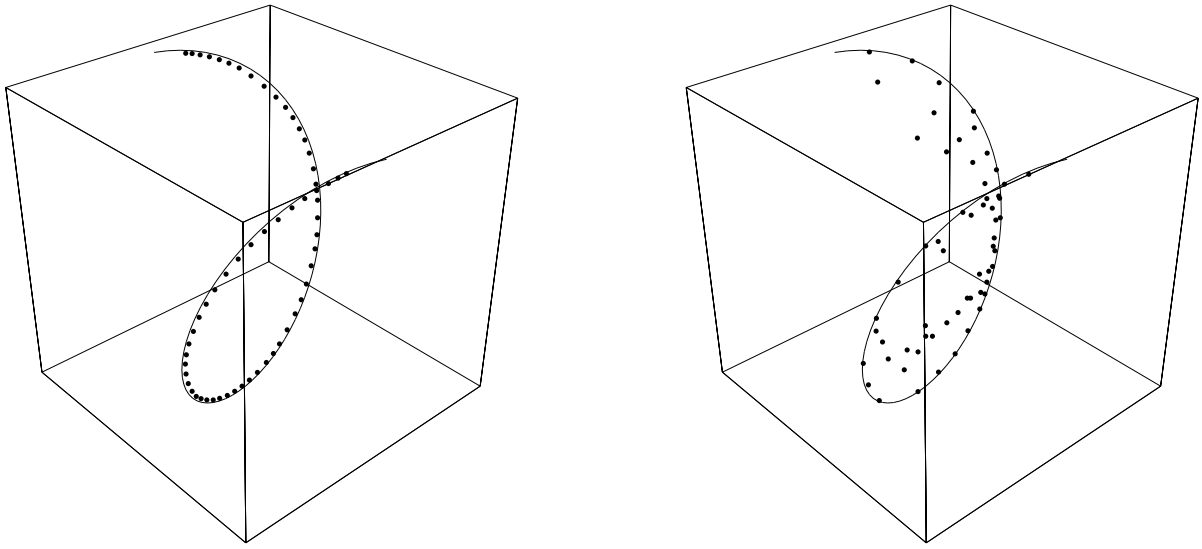

Figure 4: Weight vectors of the first (left side) and seond network (right side), the exact function is displayed as thin solid line.

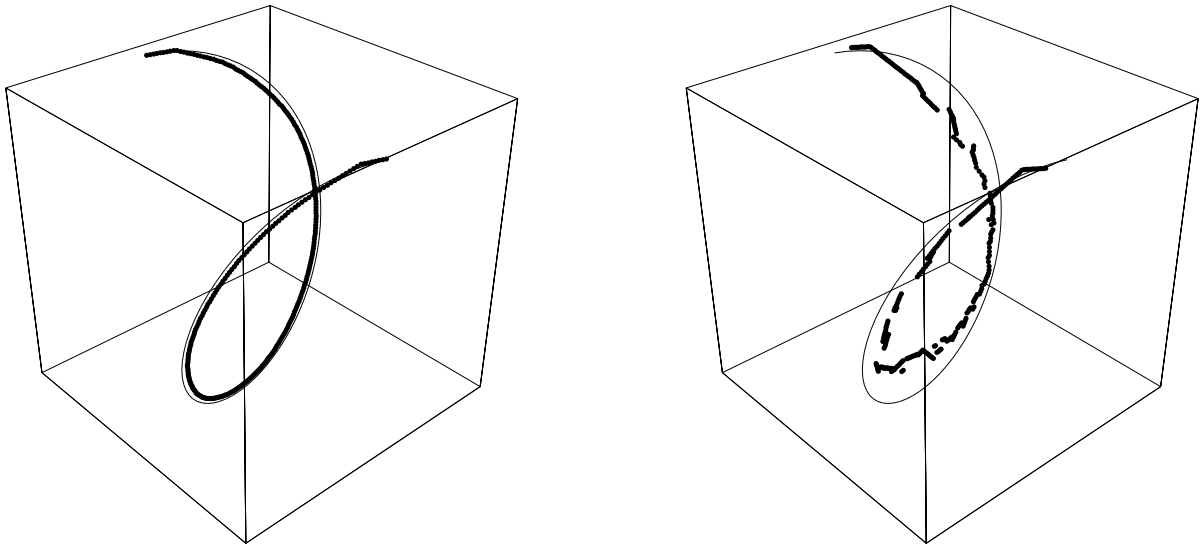

Figure 5: Estimations of the two networks, the exact function is displayed as thin solid line. Left side: Automatic parameter evolution [Demartines (1994)], right side: Standard parameter evolution [Martinetz (1991)] (without optimization yet).

4 State of charge estimation for lead-acid accumulators

In all remote electric power supplies without grid-connection a possibility for energy-storage is needed. The most widespread because less expensive solution is a lead-acid accumulator. To ensure energy supply and to prevent unadapted battery management, it is most important to know the state of charge (SOC) of the accumulator. This value is a function of the measured voltage and current (and temperature) and can be determined quite exactly by conventional algorithms if the measurements are of good quality. But the exactness of the measurement means expensive hardware which increases the costs of remote area photovoltaic installations significantly.

This is why at Fraunhofer ISE, a new system of state of charge estimator *BRAINSOC* [Kray (1999)] has been developed which allows the physical measurement of voltage and current to be less accurate because of the support of a modified neural gas network.

In a lead-acid accumulator, the electrodes made of lead resp. lead-oxide are transformed to lead sulfate and vice versa to store or produce electric energy. The space between the two electrodes is filled with sulfuric acid that participates in the chemical reaction. By measuring the mean acid concentration after several hours of zero current, the SOC can be very accurately determined.

But because acid concentration measurement devices are very expensive and in some batterie systems the acid itself is not accessible, alternative methods are needed. These methods mostly rely on simplified physical models of the accumulator and estimate the SOC depending on the batterie voltage and current.

Hence a functional relationship SOC=f(U,I) can be established and an example for simulation data is given in figure 6. The value of SOC ranges always betwenn 0% and 100% resp. 0 and 1. If one manages to train a neural

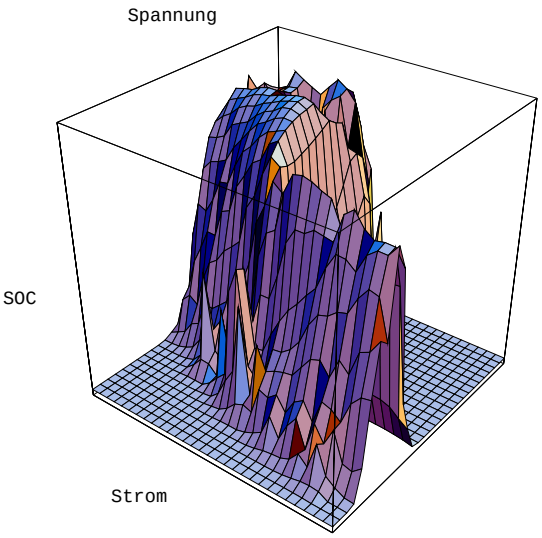

Figure 6: An example of the functional relationship between voltage (Spannung), current (Strom) and SOC of a lead-acid accumulator based on simulation data

network to reproduce this graph one has a look-up-table to detect significant offset of conventional methods. This is the idea of *BRAINSOC*, see figure 7.

There are three different conventional methods (AdBiLa, CCE, Jossen) that give each a SOC estimation. From these values, a weighted mean SOC is calculated and used to train a neural gas network. When the training has progressed sufficiently, an additional SOC estimation can be retrieved from the network. This estimation is compared to the current conventional value and if an offset is detected, the output SOC is corrected.

The neural gas network is embedded in the *NeuroToolBox*.

4.1 Difficulties of the training data

The task of reproducing the functional relationship seems quite easy but there are a few problem-inherent diffficulties that makes the SOC estimation so challenging.

First, *there is no training set.* This is because of the low memory space of microcontrollers that are used

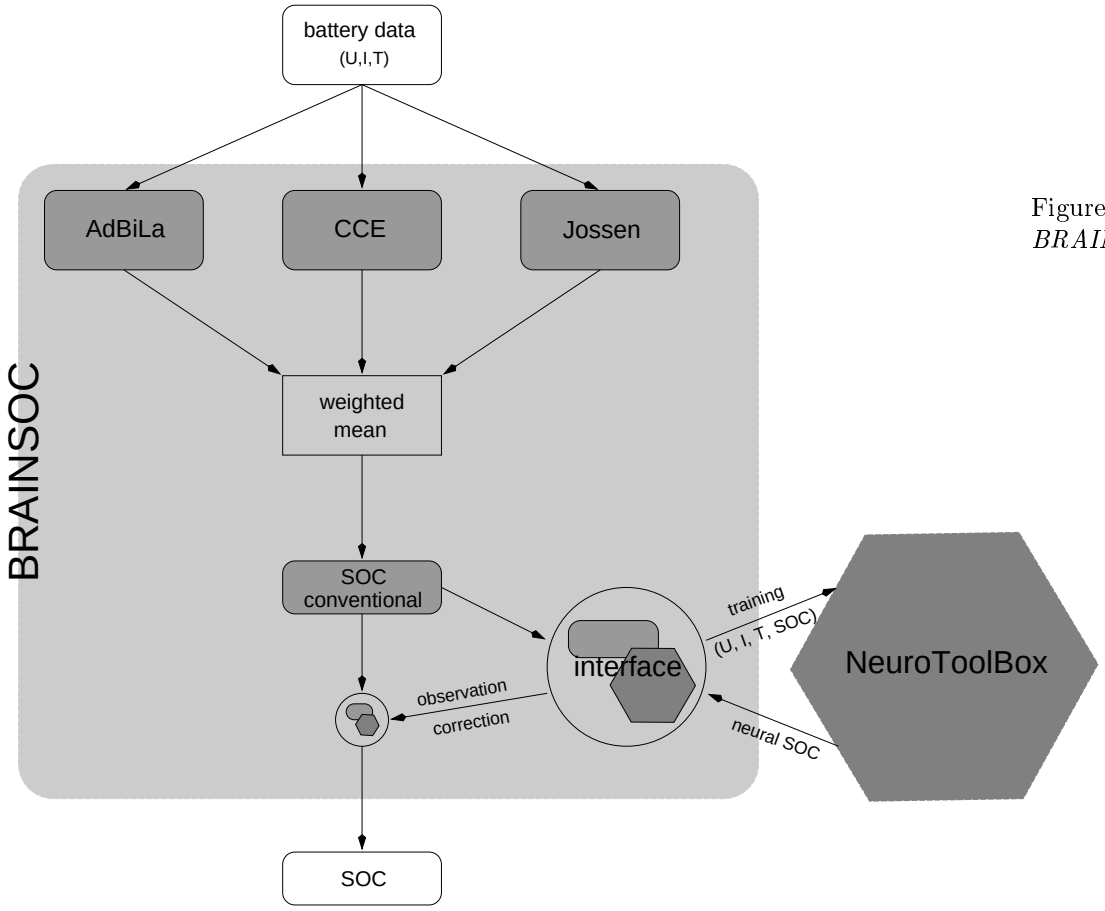

Figure 7: The *BRAINSOC* method

in typical hardware. The learning must be effected by the means of single pairs of voltage and current, no history can be saved. This would be of no particular use in the present application because ageing effects of the accumulator change slowly the functional relationship which must be continuously adapted to these drifts.

Second, the *distribution of current and voltage values is highly uneven.* In the summer there may be long periods of high SOC and high voltage where in the winter, long-term deep-discharge situations may occur. For typical distributions of SOC in a grid-independent hybrid photovoltaic installation see figure 8. This fact requires an intelligent filtering of input data to equilibrate the learning vector distribution which is an extremely difficult task for the sequential data from an accumulator.

Finally, the *functional relationship itself is not properly given.* When one analyses the range of SOC values for a pair of current and voltage values of about $60\%_{abs}$ can be found. This means that a battery at I=0 A and U=2 V/cell can be at SOC=20% or at SOC=80% for example[1]. The mean range lies at approximately 20% so one can be satisfied if the overall estimation error of the neural networks does not exceed 15% or so what we realised so far.

4.2 First Results

The base for our analysis are data measured at a grid-independent photovoltaic installation. A conventional algorithm calculated a reference SOC *a posteriori* using the complete training set. Afterwards we ran the *BRAINSOC* algorithm using sequential data from the set and compared the results for the conventional and the corrected neural estimator.

Since the conventional algorithms give quite good estimations when accurate measurement data is available we simulated inaccurate (and therefore cheaper) measurement devices by perturbating the data of voltage and

[1]This is because of long term diffusion and acid stratification processes which make the voltage values change even at constant current conditions for many hours, respective days.

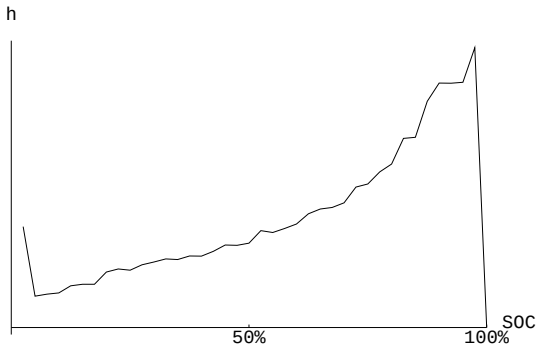

Figure 8: Highly unevenly distributed training data measured at a remote hybrid photovoltaic installation. There are remarkable favourite values for the SOC which are caracteristic for this kind of application.

current.

Since the validation of the complete *BRAINSOC* method is still in process, we present results which are based on networks that are trained with unperturbed reference data within the first 400-days period of the test data. After the 400 days, the learning phase is stopped and the fixed network is used to correct the BRAINSOC value while working now with perturbated input data[2].

As we can see in figure 11, the BRAINSOC estimation is considerably more accurate than the conventional one, cf. table 1. Even with exact measurement, the conventional SOC estimators can be improved by the *BRAINSOC* algorithm. These error values may vary slightly for longer observation intervals because of more statistical data: The fact that *BRAINSOC* has lower error values with perturbated measurement data does sureley not indicate that the algorithm profits of inaccurate input data.

Algorithm	**Mean quadratic error** [$\%_{\mathbf{abs}}$]	
	Exact measurement	With perturbation
SOC conventional	9,7	20,9
BRAINSOC	8,1	7,9

Table 1: Mean quadratic errors of SOC conventional and BRAINSOC

Even though we did not yet learn using the SOC conventional values, we were able to show that our filter for learning vectors works well and the *NeuroToolBox* is able to model the highly perturbed functional relationship SOC=f(U,I). In figures 9 and 10 we see how the 60 neurons weight vectors are organized after the training phase (i.e. 1.685 training vectors) and what overall estimation is derived from this cloud of neurons.

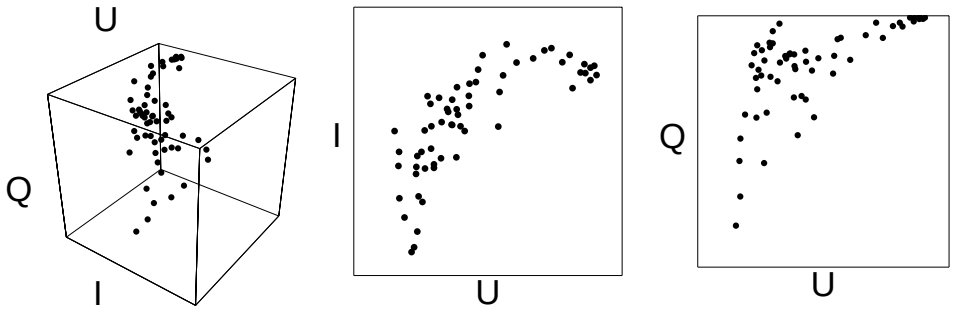

Figure 9: Final distribution of the weight vectors (after 1.685 learning vectors). Middle and right side: Projections on different planes. (Q stands for SOC)

[2]Both voltage and current were evenly perturbated by ±0.35% and had an offset of -0.14%. The percentages rely on the maximum values of the variables.

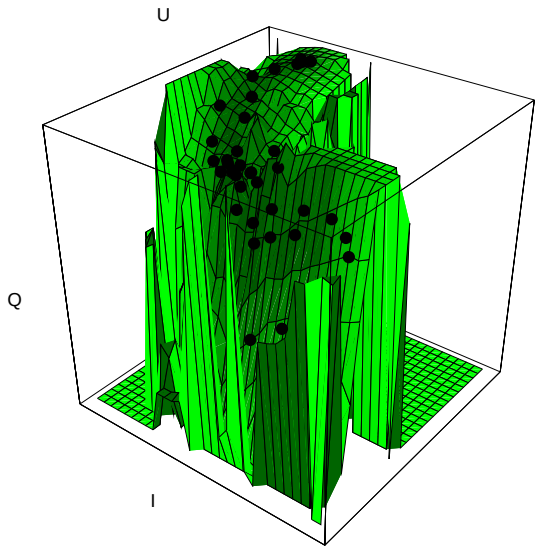

Figure 10: The final overall estimation of the *NeuroToolBox* based on the neuron distribution of figure 9.

5 Conclusion

The software *NeuroToolBox* provides an easy-to-use tool for a variety of applications. Multidimensional functional relationships with any industrial or scientific background can be analysed and modeled for further examination. It is highly adaptable for different purposes and can cope also with inherent difficult problems as we have seen for the example of the lead-acid accumulator.

Our approach faciliates the introduction of expert knowledge in the learning process and is therefore well adapted for real world applications.

References

[Demartines (1994)] Demartines, Pierre, "Analyse de données par réseaux de neurones auto-organisés", Dissertation, Laboratoire de Traitement d'Images et de Reconnaissance des Formes TIRF, Grenoble, France, 1994.

[Kray (1999)] Kray, Daniel. "Neuronale Netze zur Approximation unbekannter funktionaler Zusammenhänge — Anwendung auf die Ladezustandsschätzung beim solartypisch zyklisierten Bleiakkumulator", Diploma Thesis, University of Freiburg, Germany, March 1999.

[Martinetz (1991)] Martinetz, T., Schulten, K., "A neural gas network learns topologies", In Kohonen, T. et al., editor, IEEE International Conference on Artificial Neural Networks, Espoo, Finnland, vol. 1, pp. 397-407. Elsevier, 1991.

[Walter (1993)] Walter, J. A., Schulten, K.J., "Implementation of self-organizing neural networks for visuo-motor control of an industrial robot", IEEE Transactions on Neural Networks, 4(1): 86-95, 1993.

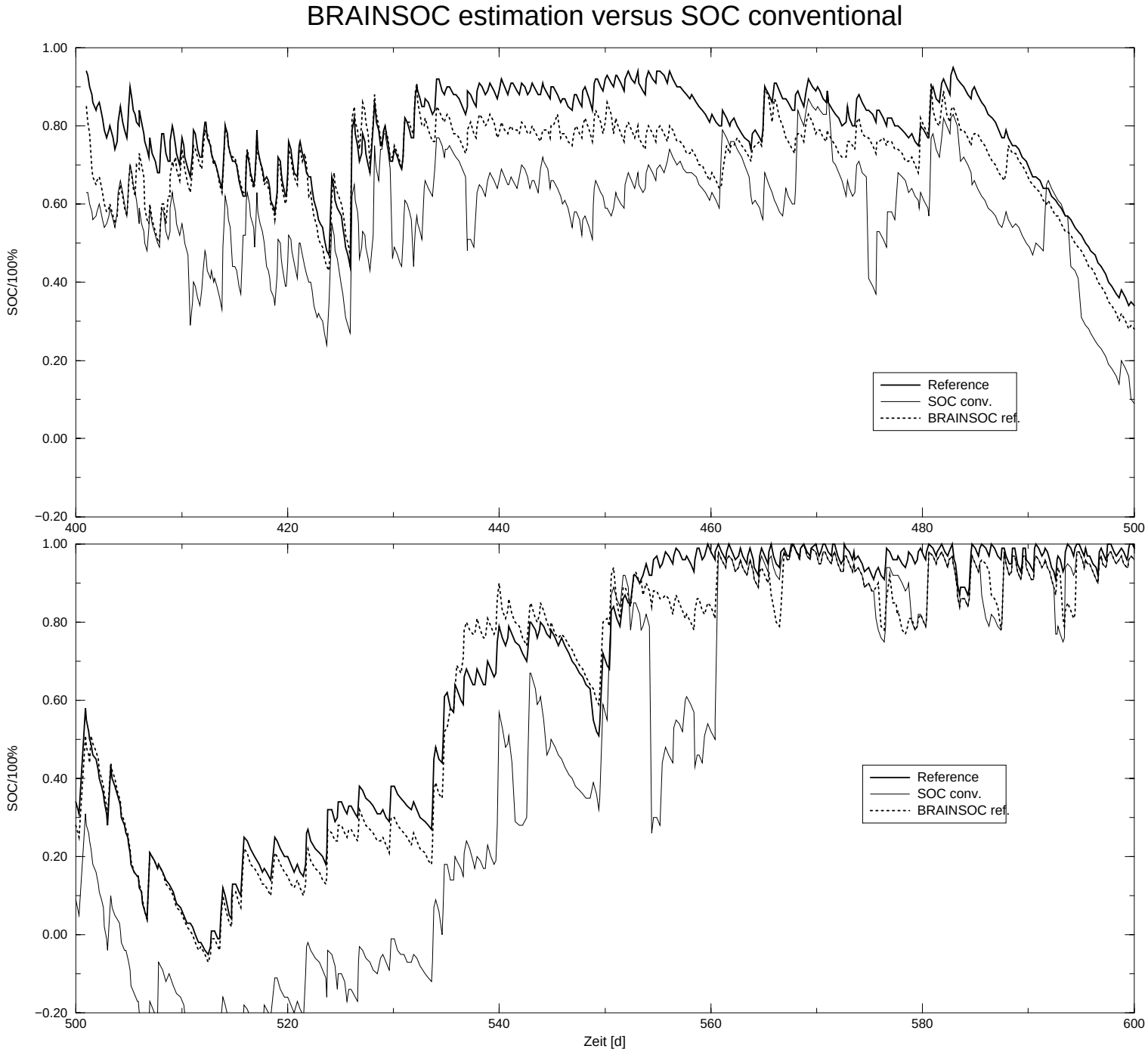

Figure 11: BRAINSOC estimation versus the conventional methods. Reference data were used for training.