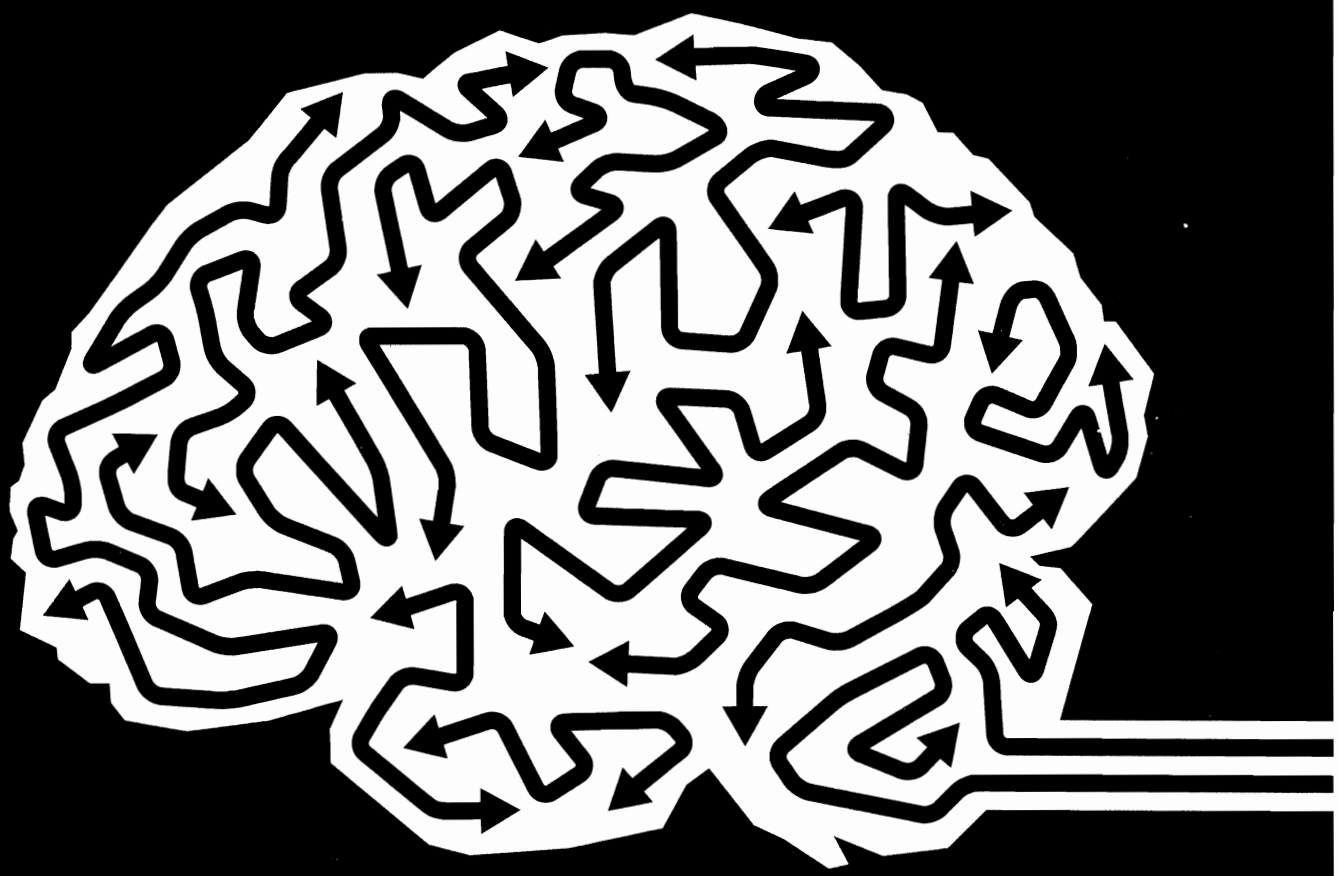
New Directions in Statistical Signal Processing

FROM SYSTEMS TO BRAINS

edited by Simon Haykin, José C. Príncipe, Terrence J. Sejnowski, and John McWhirter



Printed and bound in the United States of America

Library of Congress Cataloging-in-Publication Data

New directions in statistical signal processing: from systems to brain / edited by Simon Haykin ... [et al.].
p. cm. (Neural information processing series)
Includes bibliographical references and index.
ISBN10: 0-262-08348-5 (alk. paper)
ISBN13: 978-0-262-08348-5
1. Neural networks (Neurobiology) 2. Neural networks (Computer science) 3. Signal processing Statistical methods. 4. Neural computers. I. Haykin, Simon S., 1931 II. Series.
QP363.3.N52 2006
612.8'2 dc22 2005056210

10 9 8 7 6 5 4 3 2 1

8 Modeling Large Dynamical Systems with Dynamical Consistent Neural Networks

Hans-Georg Zimmermann, Ralph Grothmann, Anton Maximilian Schäfer, and Christoph Tietz

Recurrent neural networks are typically considered to be relatively simple architectures, which come along with complicated learning algorithms. Most researchers focus on improving these algorithms. Our approach is different: Rather than focusing on learning and optimization algorithms, we concentrate on the network architecture. Unfolding in time is a well-known example of this modeling philosophy. Here, a temporal algorithm is transferred into an architectural framework such that the learning can be done using an extension of standard error backpropagation.

As we will show, many difficulties in the modeling of dynamical systems can be solved with neural network architectures. We exemplify architectural solutions for the modeling of open systems and the problem of unknown external influences.

Another research area is the modeling of high-dimensional systems with large neural networks. Instead of modeling, e.g., a financial market as small sets of time series, we try to integrate the information from several markets into an integrated model. Standard neural networks tend to overfit, like other statistical learning systems. We will introduce a new recurrent neural network architecture in which overfitting and the associated loss of generalization abilities is not a major problem. In this context we will point to different sources of uncertainty which have to be handled when dealing with recurrent neural networks. Furthermore, we will show that sparseness of the network's transition matrix is not only important to dampen overfitting but also provides new features such as an optimal memory design.

8.1 Introduction

Recurrent neural networks (RNNs) allow the identification of dynamical systems in the form of high-dimensional, nonlinear state space models. They offer an explicit modeling of time and memory and allow us, in principle, to model any type of dy-

namical systems (Elman, 1990; Haykin, 1994; Kolen and Kremer, 2001; Medsker and Jain, 1999). The basic concept is as old as the theory of artificial neural networks, so, e.g., unfolding in time of neural networks and related modifications of the backpropagation algorithm can be found in Werbos (1974) and Rumelhart et al. (1986). Different types of learning algorithms are summarized by Pearlmutter (1995). Nevertheless, over the last 15 years most time series problems have been approached with feedforward neural networks. The appeal of modeling time and memory in recurrent networks is opposed to the apparently better numerical tractability of a pattern-recognition approach as represented by feedforward neural networks. Still, some researchers did enhance the theory of recurrent neural networks. Recent developments are summarized in the books of Haykin (1994), Kolen and Kremer (2001), Soofi and Cao (2002), and Medsker and Jain (1999).

Our approach differs from the outlined research directions in a significant but, at first sight nonobvious, way. Instead of focusing on algorithms, we put network architectures in the foreground. We show that a network architecture automatically implies using an adjoint solution algorithm for the parameter identification problem. This correspondence between architecture and equations holds for simple as well as complex network architectures. The underlying assumption is that the associated parameter optimization problem is solved by error backpropagation through time, i.e., a shared weights extension of the standard error backpropagation algorithm.

recurrent neural networks

In technical and economical applications virtually all systems of interest are open dynamical systems (see section 8.2). This means that the dynamics of the system is determined partly by an autonomous development and partly by external drivers of the system environment. The measured data always reflect a superposition of both parts. If we are interested in forecasting the development of the system, extracting the autonomous subsystem is the most relevant task. It is the only part of the open system that can be predicted (see subsec. 8.2.3). A related question is the sequence length of the unfolding in time which is necessary to approximate the recurrent system (see subsec. 8.2.2).

error correction neural networks

The outlined concepts are only applicable if we have a perfectly specified open dynamical system, where all external drivers are known. Unfortunately, this assumption is virtually never fulfilled in real-world applications. Even if we knew all the external system drivers, it would be questionable whether an appropriate amount of training data would be available. As a consequence, the task of identifying the open system is misspecified right from the beginning. On this problem, we introduce error correction neural networks (ECNN) (Zimmermann et al., 2002b) (see section 8.2.4).

dynamical consistent neural networks

Another weakness of our modeling framework is the implicit assumption that we only have to analyze a small number of time series. This is also uncommon in real-world applications. For instance, in economics we face coherent markets and not a single interest or foreign exchange rate. A market or a complex technical plant is intrinsically high dimensional. Now the major problem is that all our neural networks tend to overfit if we increase the model dimensionality in order to approach the true high-dimensional system dynamics. We therefore present recurrent network

architectures, which work even for very large state spaces (see section 8.3). These networks also combine different operations of small neural networks (e.g., processing of input information) into one shared state transition matrix. Our experiments indicate that this stabilizes the model behavior to a large extent (see section 8.3.1).

If one iterates an open system into the future, the standard assumption is that the system environment remains constant. As this is not true for most real-world applications, we introduce dynamical consistent recurrent neural networks, which try to forecast also the external influences (see section 8.3.2). We then combine the concepts of large networks and dynamic consistency with error correction. We show that ECNNs can be extended in a slightly different way than basic recurrent networks (see section 8.3.3).

We also demonstrate that some types of dynamical systems can more easily be analyzed with a (dynamical consistent) recurrent network, while others are more appropriate for ECNNs. Our intention is to merge the different aspects of the competing network architectures within a single recurrent neural network. We call it DCNN, for *dynamical consistent neural network*. We found that DCNNs allow us to model even small deviations in the dynamics without losing the generalization abilities of the model. We point out that the networks presented so far create state trajectories of the dynamics that are close to the observed ones, whereas the DCNN evolves exactly on the observed trajectories (see section 8.3.4). Finally, we introduce a DCNN architecture for partially known observables to generalize our models from a differentiation between past and future to the (time-independent) availability of information (see section 8.3.5).

uncertainties

The identification and forecasting of dynamical systems has to cope with a number of uncertainties in the underlying data as well as in the development of the dynamics (see section 8.4). Cleaning noise is a technique which allows the model itself—within the training process—to correct corrupted or noisy data (see section 8.4.1). Working with finite unfolding in time brings up the problem of initializing the internal state at the first time step. We present different approaches to achieve a desensitization of the model's behavior from the initial state and simultaneously improve the generalization abilities (see section 8.4.2). To stabilize the network against uncertainties of the environment's future development we further apply noise to the inputs in the future part of the network (see section 8.4.3).

function & structure

Working with (high-dimensional) recurrent networks raises the question of how a desired network function can be supported by a certain structure of the transition matrix (see section 8.5). As we will point out, sparseness alone is not sufficient to optimize the network functions regarding conservation and superposition of information. Only with an inflation of the internal dimension of the recurrent neural network can we implement an optimal balance between memory and computation effects (see sections 8.5.1 and 8.5.2). In this context we work out that sparseness of the transition matrix is actually a necessary condition for large neural networks (see section 8.5.3). Furthermore we analyze the information flow in sparse networks and present an architectural solution which speeds up the distribution of information (see section 8.5.4).

Finally, section 8.6 summarizes our contributions to the research field of recurrent neural networks.

8.2 Recurrent Neural Networks (RNN)

Figure 8.1) illustrates a dynamical system (Zimmermann and Neuneier, 2001, p. 321).

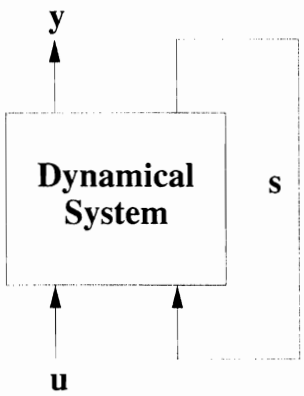


Figure 8.1 Identification of a dynamical system using a discrete time description: input u, hidden states s, and output y.

The dynamical system (fig. 8.1) can be described for discrete time grids as a set of equations (eq. 8.1), consisting of a state transition and an output equation (Haykin, 1994; Kolen, 2001):

$$\begin{aligned} s_{t+1} &= f(s_t, u_t) \quad \text{state transition} \\ y_t &= g(s_t) \quad \text{output equation} \end{aligned} \tag{8.1}$$

The state transition is a mapping from the present internal hidden state of the system s_t and the influence of external inputs u_t to the new state s_{t+1}. The output equation computes the observable output y_t.

The system can be viewed as a partially observable autoregressive dynamic state transition $s_t \rightarrow s_{t+1}$ which is also driven by external forces u_t. Without the external inputs the system is called an autonomous system (Haykin, 1994; Mandic and Chambers, 2001). However, in reality most systems are driven by a superposition of an autonomous development and external influences.

system identification

The task of identifying the dynamical system of equation 8.1 can be stated as the problem of finding (parameterized) functions f and g such that a distance measurement (eq. 8.2) between the observed data y_t^d and the computed data y_t of

the model is minimal:[1]

$$\sum_{t=1}^{T} \left(y_t - y_t^d\right)^2 \rightarrow \min_{f,g} \tag{8.2}$$

If we assume that the state transition does not depend on s_t, i.e., $y_t = g(s_t) = g(f(u_{t-1}))$, we are back in the framework of feedforward neural networks (Neuneier and Zimmermann, 1998). However, the inclusion of the internal hidden dynamics makes the modeling task much harder, because it allows varying intertemporal dependencies. Theoretically, in the recurrent framework an event s_{t+1} is explained by a superposition of external inputs $u_t, u_{t-1}, \ldots$ from all previous time steps (Haykin, 1994; Mandic and Chambers, 2001).

8.2.1 Representing Dynamic Systems by Recurrent Neural Networks

basic RNN

The identification task of equations 8.1 and 8.2 can be easily modeled by a *recurrent neural network* (Haykin, 1994; Zimmermann and Neuneier, 2001)

$$\begin{aligned} s_{t+1} &= \tanh(As_t + c + Bu_t) && \text{state transition} \\ y_t &= Cs_t && \text{output equation} \end{aligned} \tag{8.3}$$

where A, B, and C are weight matrices of appropriate dimensions and c is a bias, which handles offsets in the input variables u_t.

Note that the output equation $y_t = Cs_t$ is implemented as a linear function. It is straightforward to show that this is not a functional restriction by using an augmented inner state vector (Zimmermann and Neuneier, 2001, pp. 322–323).

By specifying the functions f and g as a neural network with weight matrices A, B and C and a bias vector c, we have transformed the system identification task of equation 8.2 into a parameter optimization problem:

$$\sum_{t=1}^{T} \left(y_t - y_t^d\right)^2 \rightarrow \min_{A,B,C,c} \tag{8.4}$$

As Hornik et al. (1992) proved for feedforward neural networks, it can be shown that recurrent neural networks (eq. 8.3) are universal approximators, as they can approximate any arbitrary dynamical system (eq. 8.1) with a continuous output function g.

8.2.2 Finite Unfolding in Time

In this section we discuss an architectural representation of recurrent neural networks that enables us to solve the parameter optimization problem of equation 8.4 by an extended version of standard backpropagation (Haykin, 1994; Rumelhart et al., 1986).[2] Figure 8.2 unfolds the network of equation 8.3 (fig. 8.2, left) over time using *shared weight matrices* A, B, and C (fig. 8.2, right). Shared weights

share the same memory for storing their weights, i.e., the weight values are the same at each time step of the unfolding and for every pattern $t \in \{1, \ldots, T\}$ (Haykin, 1994; Rumelhart et al., 1986). This guarantees that we have in every time step the same dynamics.

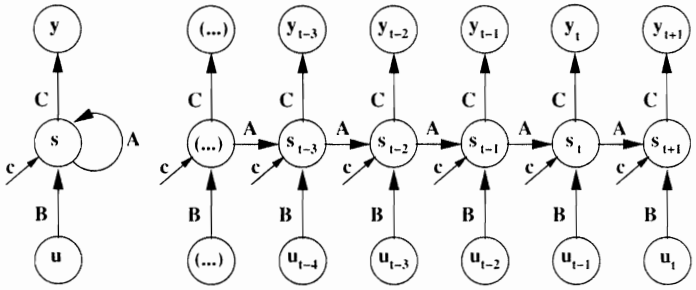


Figure 8.2 Finite unfolding using shared weight matrices A, B, and C.

We approximate the recurrence of the system with a finite unfolding which truncates after a certain number of time steps $m \in \mathrm{N}$. The important question to solve is the determination of the correct amount of past information needed to predict y_{t+1}. Since the outputs are explained by more and more external information, the error of the outputs is decreasing with each additional time step from left to right until a minimum error is achieved. This saturation level indicates the maximum number of time steps m which contribute relevant information for modeling the present time state. A more detailed description is given in Zimmermann and Neuneier (2001).

backpropagation through time

We train the unfolded recurrent neural network shown in fig. 8.2 (right) with error backpropagation through time, which is a shared weights extension of standard backpropagation (Haykin, 1994; Rumelhart et al., 1986). Error backpropagation is an efficient way of calculating the partial derivatives of the network error function. Thus, all parts of the network are provided with error information.

advantages of the RNN

In contrast to typical feedforward neural networks, RNNs are able to explicitly model memory. This allows the identification of intertemporal dependencies. Furthermore, recurrent networks contain less free parameters. In a feedforward neural network an expansion of the delay structure automatically increases the number of weights (left panel of fig. 8.3). In the recurrent formulation, the shared matrices A, B, and C are reused when more delayed input information from the past is needed (right panel of fig. 8.3).

Additionally, if weights are shared more often, more gradient information is available for learning. As a consequence, potential overfitting is not as dangerous in recurrent as in feedforward networks. Due to the inclusion of temporal structure in the network architecture, our approach is applicable to tasks where only a small training set is available (Zimmermann and Neuneier, 2001, p. 325).

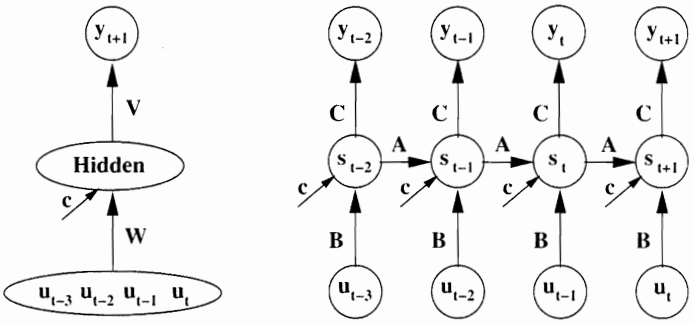


Figure 8.3 An additional time step leads in the feedforward framework (left) with $y_{t+1} = V \tanh(Wu + c)$ to a higher dimension of the input vector u, whereas the number of free parameters remains constant in recurrent networks (right), due to the use of shared weights.

8.2.3 Overshooting

An obvious generalization of the network in fig. 8.2 is the extension of the autonomous recurrence (matrix A) in future direction $t+2, t+3, \ldots$ (see fig. 8.4) (Zimmermann and Neuneier, 2001, pp. 326–327). If this so-called *overshooting* leads to good predictions, we get a whole sequence of forecasts as an output. This is especially interesting for decision support systems. The number of autonomous iterations into the future, which we define with $n \in \mathrm{N}$, most often depends on the required forecast horizon of the application. Note that overshooting does not add new parameters, since the shared weight matrices A and C are reused.

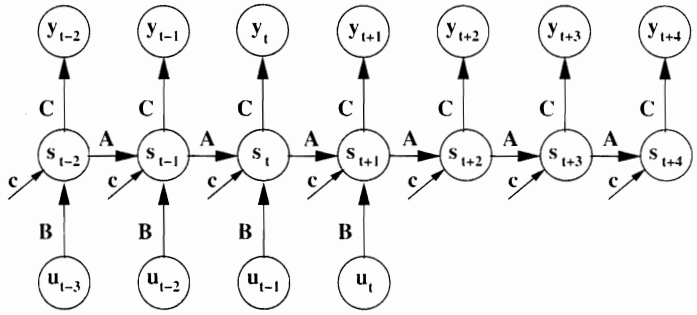


Figure 8.4 Overshooting extends the autonomous part of the dynamics.

The most important property of the overshooting network (fig. 8.4) is the concatenation of an input-driven system and an autonomous system. One may argue that the unfolding-in-time network (fig. 8.2) already consists of recurrent functions, and that this recurrent structure has the same modeling characteristics as the overshooting network. This is definitely not true, because the learning algorithm

leads to different models for each of the architectures. Backpropagation learning usually tries to model the relationship between the most recent inputs and the output because the fastest adaptation takes place in the shortest path between input and output. Thus, learning mainly focuses on u_t. Only later in the training process may learning also extract useful information from input vectors u_τ $(t-m \leq \tau < t)$ which are more distant from the output. As a consequence, the unfolding-in-time network (fig. 8.2, right) tries to rely as much as possible on the part of the dynamics which is driven by the most recent inputs $u_t, \ldots, u_{t-k}$ with $k < m$. In contrast, the overshooting network (fig. 8.4) forces the learning through additional future outputs $y_{t+2}, \ldots, y_{t+n}$ to focus on modeling an internal autonomous dynamics (Zimmermann and Neuneier, 2001).

In summary, overshooting generates additional valuable forecast information about the analyzed dynamical system and stabilizes learning.

8.2.4 Error Correction Neural Networks (ECNN)

If we have a complete description of all external influences, recurrent neural networks (eq. 8.3) allow us to identify the intertemporal relationships (Haykin, 1994). Unfortunately, our knowledge about the external forces is typically incomplete and our observations might be noisy. Under such conditions, learning with finite data sets leads to the construction of incorrect causalities due to learning by heart (overfitting). The generalization properties of such a model are questionable (Neuneier and Zimmermann, 1998).

If we are unable to identify the underlying system dynamics due to insufficient input information or unknown influences, we can refer to the actual model error $y_t - y_t^d$, which can be interpreted as an indicator that our model is misleading. Handling this error information as an additional input, we extend equation 8.1, obtaining:

$$\begin{aligned} s_{t+1} &= f(s_t, u_t, y_t - y_t^d), \\ y_t &= g(s_t). \end{aligned} \tag{8.5}$$

The state transition s_{t+1} is a mapping from the previous state s_t, external influences u_t, and a comparison between model output y_t and observed data y_t^d. If the model error $(y_t - y_t^d)$ is zero, we have a perfect description of the dynamics. However, due to unknown external influences or noise, our knowledge about the dynamics is often incomplete. Under such conditions, the model error $(y_t - y_t^d)$ quantifies the model's misfit and serves as an indicator of short-term effects or external shocks (Zimmermann et al., 2002b).

error correction network

Using weight matrices A, B, C and D of appropriate dimensions corresponding to s_t, u_t, and $(y_t - y_t^d)$ and a bias c, a neural network approach to 8.5 can be written

as

$$\begin{aligned} s_{t+1} &= \tanh(As_t + c + Bu_t + D\tanh(Cs_t - y_t^d)), \\ y_t &= Cs_t. \end{aligned} \tag{8.6}$$

In 8.6 the output y_t is computed by Cs_t and compared to the observation y_t^d. The matrix D adjusts a possible difference in the dimension between the error correction term and s_t. The system identification is now a parameter optimization task of appropriately sized weight matrices A, B, C, D, and the bias c (Zimmermann et al., 2002b):

system identification

$$\sum_{t=1}^{T} \left(y_t - y_t^d\right)^2 \rightarrow \min_{A,B,C,D,c} \tag{8.7}$$

finite unfolding

We solve the system identification task of 8.7 by finite unfolding in time using shared weights (see section 8.2.2). Figure 8.5 depicts the resulting neural network solution of 8.6.

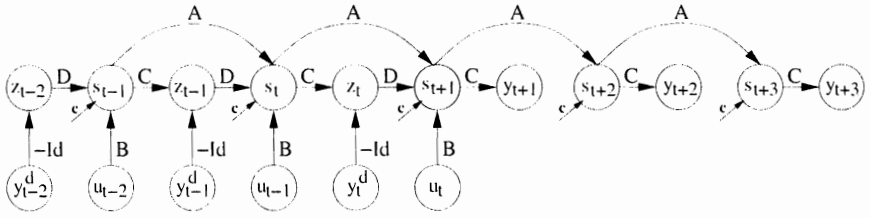


Figure 8.5 Error correction neural network (ECNN) using unfolding in time and overshooting. Note that $-Id$ is the fixed negative of an appropriate-sized identity matrix, while z_τ with $t - m \leq \tau \leq t$ are output clusters with target values of zero in order to optimize the error correction mechanism.

The ECNN (eq. 8.6) is best understood by analyzing the dependencies of s_t, u_t, $z_t = Cs_t - y_t^d$, and s_{t+1}. The ECNN has two different inputs: the externals u_t directly influencing the state transition; and the targets y_t^d. Only the difference between y_t and y_t^d has an impact on s_{t+1} (Zimmermann et al., 2002b). At all future time steps $t < \tau \leq t + n$, we have no compensation of the internal expectations y_τ, and thus the system offers forecasts $y_\tau = Cs_\tau$.

overshooting

The autonomous part of the ECNN is analogous to the RNN case (see section 8.2.3) – extended into the future by overshooting. Besides all advantages described in section 8.2.3, overshooting influences the learning of the ECNN in an extended way. A forecast provided by the ECNN is in general, based on a modeling of the recursive structure of a dynamical system (coded in the matrix A) and on the error correction mechanism which acts as an external input (coded in C and D). Now, the overshooting enforces an autoregressive substructure allowing long-term forecasts. Of course, we have to supply target values for the additional output

clusters y_τ, $t < \tau \leq t + n$. Due to the shared weights, there is no change in the number of model parameters (Zimmermann et al., 2002b).

8.3 Dynamical Consistent Neural Networks (DCNN)

The neural networks described in section 8.2 not only learn from data, but also integrate prior knowledge and first principles into the modeling in the form of architectural concepts.

market dynamics

However, the question arises if the outlined neural networks are a sufficient framework for the modeling of complex nonlinear dynamical systems, which can only be understood by analyzing the interrelationship of different subdynamics. Consider the following economic example: The dynamics of the US dollar euro foreign exchange market is clearly influenced by the development of other major foreign exchange, stock or commodity markets (Murphy, 1999). In other words, movements of the US dollar euro foreign exchange rate can only be comprehended by a combined analysis of the behavior of other coherent markets. This means that a model of the US dollar and euro foreign exchange market must also learn the dynamics of related markets and intermarket dependencies. Now it is important to note that, due to their computational power (in the sense of modeling high-dimensional nonlinear dynamics), the described medium-sized recurrent neural networks are only capable of modeling a single market's dynamics. From this point of view an integrated approach of market modeling is hardly possible within the framework of those networks. Hence, we need *large* neural networks.

A simple scaling up of the presented neural networks would be misleading. Our experiments indicate that scaling up the networks by increasing the dimension of the internal state results in overfitting due to the large number of free parameters.

overfitting

Overfitting is a critical issue, because the neural network does not only learn the underlying dynamics, but also the noise included in the data. Especially in economic applications, overfitting poses a serious problem.

In this section we deal with architectures which are feasible for large recurrent neural networks. These architectures are based on a redesign of the recurrent neural networks introduced in section 8.2. Most of the resulting networks cannot even be designed with a low-dimensional internal state (see section 8.3.1). In addition, we focus on a consistency problem of traditional statistical modeling: Typically one assumes that the environment of the system remains unchanged when the dynamics is iterated into the future. We show that this is a questionable statistical assumption, and solve the problem with a dynamical consistent recurrent neural network (see section 8.3.2). Thereafter, we deal with large error correction networks and integrate dynamical consistency into this framework (see section 8.3.3). Finally, we point out that large RNNs and large ECNNs are appropriate for different types of dynamical systems. Our intention is to merge the different characteristics of the two models in a unified neural network architecture. We call it DCNN for *dynamical consistent neural network* (see section 8.3.4). Finally we discuss the problem of

partially known observables (see section 8.3.5).

8.3.1 Normalization of Recurrent Networks

Let us revisit the basic time-delay recurrent neural network of 8.3. The state transition equation s_t is a nonlinear combination of the previous state s_{t-1} and external influences u_t using matrices A and B. The network output y_t is computed from the present state s_t employing matrix C. The network output is therefore a nonlinear composition applying the transformations A, B, and C.

In preparation for the development of large networks we first separate the state equation of the recurrent network (eq. 8.3) into a past and a future part. In this framework s_t is always regarded as the present time state. That means that for this pattern t all states s_τ with $\tau \leq t$ belong to the past part and those with $\tau > t$ to the future part. The parameter τ is hereby always bounded by the length of the unfolding m and the length of the overshooting n (see sections 8.2.2 and 8.2.3), such that we have $\tau \in \{t-m, \ldots, t+n\}$ for all $t \in \{m, \ldots, T-n\}$ with T as the available number of data patterns. The present time ($\tau = t$) is included in the past part, as these state transitions share the same characteristics. We get the following representation of the optimization problem:

$$\begin{aligned} \tau \leq t: \quad s_{\tau+1} &= \tanh(As_\tau + c + Bu_\tau) \\ \tau > t: \quad s_{\tau+1} &= \tanh(As_\tau + c) \\ y_\tau &= Cs_\tau, \qquad \sum_{t=m}^{T-n} \sum_{\tau=t-m}^{t+n} (y_\tau - y_\tau^d)^2 \rightarrow \min_{A,B,C,c} \end{aligned} \tag{8.8}$$

As shown in section 8.2, these equations can be easily transformed into a neural network architecture (see fig. 8.4).

In this model, past and future iterations are consistent under the assumption of a constant future environment. The difficulty with this kind of recurrent neural network is the training with backpropagation through time, because a sequence of different connectors has to be balanced. The gradient computation is not regular, i.e., we do not have the same learning behavior for the weight matrices in different time steps. In our experiments we found that this problem becomes more important for training large recurrent neural networks. Even the training itself is unstable due to the concatenated matrices A, B, and C. As training changes weights in all of these matrices, different effects or tendencies – even opposing ones – can influence them and may superpose. This implies that no clear learning direction or weight changes result from a certain backpropagated error.

The question arises of how to redesign the basic recurrent architecture (eq. 8.8) to improve learning behavior and stability especially for large networks.

normalized recurrent networks

As a solution, we propose the neural network of 8.9, which incorporates besides the bias c only one connector, the matrix A. The corresponding architecture is depicted in fig. 8.6. Note that from now on we change the formulation of the system

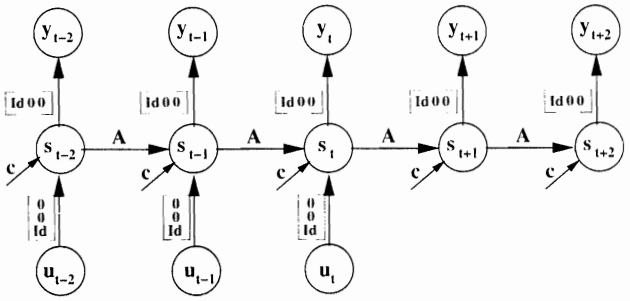


Figure 8.6 Normalized recurrent neural network.

equations (e.g., eq. 8.8) from a forward ($s_{t+1} = f(s_t, u_t)$) to a backward formulation ($s_t = f(s_{t-1}, u_t)$). As we will see, the backward formulation is internally equivalent to a forward model.

$$\begin{aligned} \tau \leq t: \quad s_\tau &= \tanh\left(As_{\tau-1} + c + \begin{bmatrix} 0 \\ 0 \\ Id \end{bmatrix} u_\tau \right) \\ \tau > t: \quad s_\tau &= \tanh(As_{\tau-1} + c) \\ y_\tau &= [Id\ 0\ 0]s_\tau, \qquad \sum_{t=m}^{T-n} \sum_{\tau=t-m}^{t+n} (y_\tau - y_\tau^d)^2 \to \min_{A,c} \end{aligned} \tag{8.9}$$

We call this model a normalized recurrent neural network (NRNN). It avoids the stability and learning problems resulting from the concatenation of the three matrices A, B, and C. The modeling is now focused solely on the transition matrix A. The matrices between input and hidden as well as between hidden and output layers are fixed and therefore not learned during the training process. This implies that all free parameters – as they are combined in one matrix – are now treated the same way by backpropagation.

It is important to note that the normalization or concentration on only one single matrix is paid for with an oversized (high-dimensional) internal state. At first view it seems that in this network architecture (fig. 8.6) the external input u_τ is directly connected to the corresponding output y_τ. This is not the case, though, because we increase the dimension of the internal state s_τ, such that the input u_τ has no direct influence on the output y_τ. Assuming that we have a number p of network outputs, q computational hidden neurons, and r external inputs, the dimension of the internal state would be $\dim(s) \geq p + q + r$.

With the matrix $[Id\ 0\ 0]$ we connect only the first p neurons of the internal state s_τ to the output layer y_τ. This connector is a fixed identity matrix of appropriate size. Consequently, the neural network is forced to generate the p outputs of the neural network at the first p components of the state vector s_τ.

Let us now focus on the last r state neurons, which are used for the processing

of the external inputs u_τ. The connector $[0\ 0\ Id]^T$ between the externals u_τ and the internal state s_τ is an appropriately sized fixed identity matrix. More precisely, the connector is designed such that the input u_τ is connected to the last state neurons. Recalling that the network outputs are located at the first p internal states, this composition avoids a direct connection between input and output. It delays the impact of the externals u_τ on the outputs y_τ by at least one time step.

To additionally support the internal processing and to increase the network's computational power, we add a number q of hidden neurons between the first p and the last r state neurons. This composition ensures that input and output processing of the network are separate.

Besides the bias vector c the state transition matrix A holds the only tunable parameters of the system. Matrix A does not only code the autonomous and the externally driven parts of the dynamics, but also the processing of the external inputs u_τ and the computation of the network outputs y_τ. The bias added to the internal state handles offsets in the input variables u_τ.

large networks

Remarkably, the normalized recurrent network of 8.9 can only be designed as a large neural network. If the internal network state is too small, the inputs and outputs cannot be separated, as the external inputs would at least partially cover the internal states at which the outputs are read out. Thus, the identification of the network outputs at the first p internal states would become impossible.

Our experiments indicate that recurrent neural networks in which the only tunable parameters are located in a single state transition matrix (e.g., eq. 8.9) show a more stable training process, even if the dimension of the internal state is very large. Having trained the large network to convergence, many weights of the state transition matrix will be dispensable without derogating the functioning of the network. Unneeded weights can be singled out by using a weight decay penalty and standard pruning techniques (Haykin, 1994; Neuneier and Zimmermann, 1998).

modeling observables

In the normalized recurrent neural network (eq. 8.9) we consider inputs and outputs independently. This distinction between externals u_τ and the network output y_τ is arbitrary and mainly depends on the application or the view of the model builder instead of the real underlying dynamical system. Therefore, for the following model we take a different point of view. We merge inputs and targets into one group of variables, which we call *observables*. So we now look at the model as a high-dimensional dynamical system where input and output represent the observable variables of the environment. The hidden units stand for the unobservable part of the environment, which nevertheless can be reconstructed from the observations. This is an integrated view of the dynamical system.

We implement this approach by replacing the externals u_τ with the (observable) targets y_τ^d in the normalized recurrent network. Consequently, the output y_τ

and the external input y_τ^d have now identical dimensions.

$$\begin{aligned} \tau \leq t: \quad s_\tau &= \tanh\left(As_{\tau-1} + c + \begin{bmatrix} 0 \\ 0 \\ Id \end{bmatrix} y_\tau^d\right) \\ \tau > t: \quad s_\tau &= \tanh(As_{\tau-1} + c) \\ y_\tau &= [Id\ 0\ 0]s_\tau, \qquad \sum_{t=m}^{T-n} \sum_{\tau=t-m}^{t+n} (y_\tau - y_\tau^d)^2 \to \min_{A,c} \end{aligned} \tag{8.10}$$

The corresponding model architecture is shown in fig. 8.7.

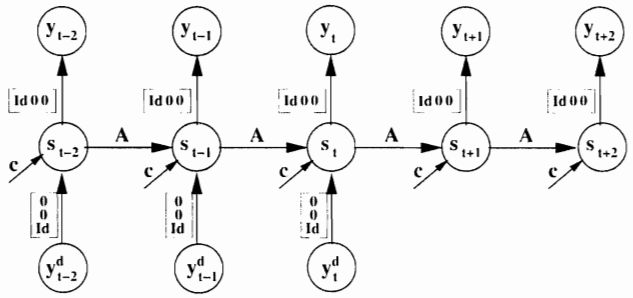


Figure 8.7 Normalized recurrent net modeling the dynamics of observables y_τ^d.

Note that, because of the one-step time delay between input and output, y_τ^d and y_τ are not directly connected. Furthermore, it is important to understand that we now take a totally different view of the dynamical system. In contrast to 8.9, this network (eq. 8.10) not only generates forecasts for the dynamics of interest but for all external observables y_τ^d. Consequently, the first r state neurons are used for the identification of the network outputs. They are followed by q computational hidden neurons, and r state neurons that read in the external inputs.

8.3.2 Dynamical Consistent Recurrent Neural Networks (DCRNN)

The models presented so far are all statistical but not dynamical consistent, as we assume that the environment stays constant for the future part of the network. In the following we improve our models with dynamical consistency.

An open dynamical system is partially driven by an autonomous development and partially by external influences. When the dynamics is iterated into the future, the development of the system environment is unknown. Now, one of the standard statistical paradigms is to assume that the external influences are not significantly changing in the future part. This means that the expected value of a shift in an external input y_τ^d with $\tau > t$ is 0 by definition. For that reason we have so far

neglected the external inputs y_τ^d in the normalized recurrent neural network at all future unfolding time steps, $\tau > t$ (see eq. 8.10).

Especially when we consider fast-changing external variables with a high impact on the dynamics of interest, the above assumption is very questionable. In relation to 8.10 it even poses a contradiction, as the observables are assumed to be constant on the input and variable on the output side. Even in case of a slowly changing environment, long-term forecasts become doubtful. The longer the forecast horizon is, the more the statistical assumption is violated. A statistical model is therefore not consistent from a dynamical point of view. For a dynamical consistent approach, one has to integrate assumptions about the future development of the environment into the modeling of the dynamics.

For that reason we propose a network that uses its own predictions as replacements for the unknown future observables. This is expressed by an additional fixed matrix in the state equation. The resulting DCRNN is:

$$\begin{aligned}
\tau \le t: \quad s_\tau &= \begin{bmatrix} Id & 0 & 0 \\ 0 & Id & 0 \\ 0 & 0 & 0 \end{bmatrix} \tanh(As_{\tau-1} + c) + \begin{bmatrix} 0 \\ 0 \\ Id \end{bmatrix} y_\tau^d \\
\tau > t: \quad s_\tau &= \begin{bmatrix} Id & 0 & 0 \\ 0 & Id & 0 \\ Id & 0 & 0 \end{bmatrix} \tanh(As_{\tau-1} + c) \\
y_\tau &= [Id\ 0\ 0] \quad s_\tau, \qquad \sum_{t=m}^{T-n} \sum_{\tau=t-m}^{t+n} (y_\tau - y_\tau^d)^2 \to \min_{A,c}
\end{aligned} \tag{8.11}$$

state vector

Similarly to the end of section 8.3.1, we look at the state vector s_τ in a very structured way. The recursion of the state equations (eq. 8.11) acts in the past ($\tau \le t$) and future ($\tau > t$) always on the same partitioning of that vector. For all $\tau \in \{t-m, \ldots, t+n\}$, s_τ can be described as

$$s_\tau = \begin{bmatrix} y_\tau \\ h_\tau \\ \left\{ \begin{array}{l} \tau \le t: \ y_\tau^d \\ \tau > t: \ y_\tau \end{array} \right\} \end{bmatrix} = \begin{bmatrix} \text{expectations} \\ \text{hidden states} \\ \left\{ \begin{array}{l} \tau \le t: \ \text{observations} \\ \tau > t: \ \text{expectations} \end{array} \right\} \end{bmatrix}. \tag{8.12}$$

This means that in the first r components of the state vector we have the expectations y_τ, i.e., the predictions of the model. The q components in the middle of the vector represent the hidden units h_τ. They are actually responsible for the development of the dynamics. In the last r components of the vector we find in the past ($\tau \le t$) the observables y_τ^d, which the model receives as external input. In the future ($\tau > t$) the model replaces these unknown future observables by its own expectations y_τ. This replacement is modeled with two consistency matrices:

consistency matrices

$$C_{\leq} = \begin{bmatrix} Id & 0 & 0 \\ 0 & Id & 0 \\ 0 & 0 & 0 \end{bmatrix} \text{ and } C_{>} = \begin{bmatrix} Id & 0 & 0 \\ 0 & Id & 0 \\ Id & 0 & 0 \end{bmatrix}. \tag{8.13}$$

Let us explain one recursion of the state equation (eq. 8.11) in detail: In the past ($\tau \leq t$) we start with a state vector $s_{\tau-1}$, which has the structure of 8.12. This vector is first multiplied with the transition matrix A. After adding the bias c, the vector is sent through the nonlinearity tanh. The consistency matrix then keeps the first $r+q$ components (expectations and hidden states) of the state vector but deletes (by multiplication with zero) the last r ones. These are finally replaced by the observables y_τ^d, such that s_τ again has the partitioning of 8.12. Note that in contrast to the normalized recurrent neural network (eq. 8.10) the observables are now added to the state vector after the nonlinearity. This is important for the consistency structure of the model.

The recursion in the future state transition ($\tau > t$) differs from the one in the past in terms of the structure of the consistency matrix and the missing external input. The latter is now replaced with an additional identity block in the future consistency matrix $C_>$, which maps the first r components of the state vector, the expectations y_τ, to its last r components. Thus we get the desired partitioning of s_τ (eq. 8.12) and the model becomes dynamical consistent.

Figure 8.8 illustrates this architecture. Note that the nonlinearity and the final calculation of the state vector are separate and hence modeled in two different layers. This follows from the dynamical consistent state equation (eq. 8.11), in which the observables are added separately from the nonlinear component.

transition matrix

Regarding the single transition matrix A, we want to point out that in a statistical consistent recurrent network (eq. 8.10) the matrix has to model the state transformation over time and the merging of the input information. However, the

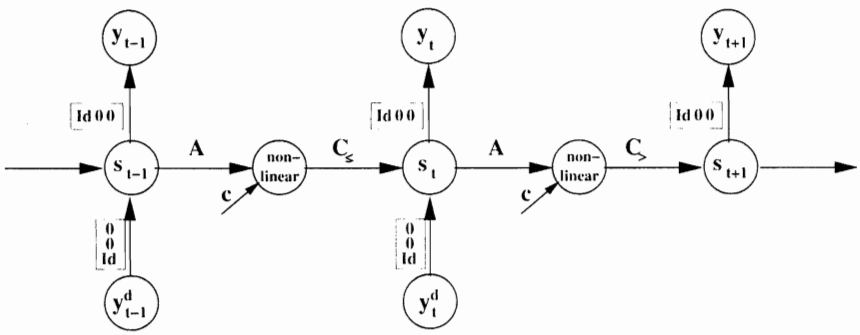


Figure 8.8 Dynamical consistent recurrent neural network (DCRNN). At all future time steps of the unfolding the network uses its own forecasts as substitutes for the unknown development of the environment.

network is only triggered by the external drivers up to the present time step t. In a dynamical consistent network we have forecasts of the external influences, which can be used as future inputs. Thus, the transition matrix A is always dedicated to the same task: modeling the dynamics.

8.3.3 Dynamical Consistent Error Correction NNs (DCECNN)

The ECNN is a nonlinear state space model employing the shared weight matrices A, B, C, and D (eq. 8.6). Matrix A computes the state transformation over time, B processes the external input information, C derives the network output, and D is responsible for the error correction mechanism. The latter composition of nonlinear transformations A, B, C, and D is difficult to handle when the network's internal state is high dimensional.

Therefore we developed a dynamical consistent error correction neural network (DCECNN) of the form of 8.14. It is an analogous approach to the DCRNN (eq. 8.11) and consequently the equations are very similar. The only changes concern the two consistency matrices $C_{\leq}$ and $C_{>}$.

$$\begin{aligned}
\tau \leq t: \quad s_\tau &= \begin{bmatrix} Id & 0 & 0 \\ 0 & Id & 0 \\ -Id & 0 & 0 \end{bmatrix} \tanh(As_{\tau-1} + c) + \begin{bmatrix} 0 \\ 0 \\ Id \end{bmatrix} y_\tau^d \\
\tau > t: \quad s_\tau &= \begin{bmatrix} Id & 0 & 0 \\ 0 & Id & 0 \\ 0 & 0 & 0 \end{bmatrix} \tanh(As_{\tau-1} + c) \\
y_\tau &= [Id \ 0 \ 0] \ s_\tau, \qquad \sum_{t=m}^{T-n} \sum_{\tau=t-m}^{t+n} (y_\tau - y_\tau^d)^2 \to \min_{A,c}.
\end{aligned} \tag{8.14}$$

state vector

Due to the error correction, the definition or partitioning of the state vector differs in the last r components. We now have for all $\tau \in \{t-m, \ldots, t+n\}$

$$s_\tau = \begin{bmatrix} y_\tau \\ h_\tau \\ \left\{ \begin{array}{ll} \tau \leq t: & e_\tau \\ \tau > t: & 0 \end{array} \right\} \end{bmatrix} = \begin{bmatrix} \text{expectations} \\ \text{hidden states} \\ \left\{ \begin{array}{ll} \tau \leq t: & \text{error correction} \\ \tau > t: & 0 \end{array} \right\} \end{bmatrix}. \tag{8.15}$$

In the past part ($\tau \leq t$) we get the error correction term in the state vector by subtracting the expectations y_τ from the observations y_τ^d. This is performed by the negative identity matrix $-Id$ within the consistency matrix $C_{\leq}$. In the future part ($\tau > t$) we expect that our model is correct. Therefore we replace the error correction by zero. The future consistency matrix $C_{>}$ simply overwrites the last r

components of the state vector with zero. Analogous to the DCRNN, the internal transition matrix A is only used for the modeling of the dynamics over time.

The graphical illustration of a dynamical consistent error correction neural network is identical to the recurrent one (fig. 8.8), but note, that the consistency matrices $C_{\leq}$ and $C_{>}$ have changed their structure.

8.3.4 Dynamical Consistent Neural Networks (DCNN)

Dynamical consistent neural networks(see section 8.3.2) are most appropriate if the observed dynamics is not hidden by noise and evolves smoothly over time, e.g., modeling of a sine curve. However, modeling can only be successful if we know all external drivers of the system and the dynamics is not influenced by external shocks. In many real-world applications, e.g. trading (see Zimmermann et al. (2002a)), this is simply not true. The dynamics of interest is often covered with noise. External shocks or unknown external influences disturb the system dynamics. In this case, one should apply DCECNNs (see section 8.3.3), which describe the dynamics with an internal expectation and its deviation from the observables. Now the question arises of whether and how we can merge the different model characteristics within a single dynamical consistent neural network (DCNN).

There are two different ways to set up this combination. In our first approach (eq. 8.18) we keep the framework of the DCRNN (eq. 8.11), whereas the second one (eq. 8.22) is based on the DCECNN (eq. 8.14).

The first approach is based on the DCRNN. Consequently, the state vector s_τ has, in the past ($\tau \leq t$) and future ($\tau \leq t$) for all $\tau \in \{t-m, \ldots, t+n\}$, the partitioning of 8.16 (see also eq. 8.12).

$$s_\tau = \begin{bmatrix} y_\tau \\ h_\tau \\ \left\{ \begin{array}{l} \tau \leq t : y_\tau^d \\ \tau > t : y_\tau \end{array} \right\} \end{bmatrix} = \begin{bmatrix} \text{expectations} \\ \text{hidden states} \\ \left\{ \begin{array}{l} \tau \leq t : \text{observations} \\ \tau > t : \text{expectations} \end{array} \right\} \end{bmatrix} \tag{8.16}$$

In comparison to the DCRNN (eq. 8.11) the recursion of the new model (eq. 8.18) is extended by an additional consistency matrix

$$C = \begin{bmatrix} 0 & 0 & Id \\ 0 & Id & 0 \\ -Id & 0 & Id \end{bmatrix} \tag{8.17}$$

between the state vector and the transition matrix A. As we will see, this matrix ensures that the model is supplied with the information of the observables y_τ^d as well as the error corrections e_τ. We call this approach DCNN1 (eq. 8.18). The

corresponding network architecture is depicted in fig. 8.9.

$$
\begin{aligned}
\tau \leq t: \; s_\tau &= \begin{bmatrix} Id & 0 & 0 \\ 0 & Id & 0 \\ 0 & 0 & 0 \end{bmatrix} \tanh\left(A \begin{bmatrix} 0 & 0 & Id \\ 0 & Id & 0 \\ -Id & 0 & Id \end{bmatrix} s_{\tau-1} + c \right) + \begin{bmatrix} 0 \\ 0 \\ Id \end{bmatrix} y_\tau^d \\
\tau > t: \; s_\tau &= \begin{bmatrix} Id & 0 & 0 \\ 0 & Id & 0 \\ Id & 0 & 0 \end{bmatrix} \tanh\left(A \begin{bmatrix} 0 & 0 & Id \\ 0 & Id & 0 \\ -Id & 0 & Id \end{bmatrix} s_{\tau-1} + c \right) \\
y_\tau &= \begin{bmatrix} Id & 0 & 0 \end{bmatrix} s_\tau, \qquad \sum_{t=m}^{T-n} \sum_{\tau=t-m}^{t+n} (y_\tau - y_\tau^d)^2 \rightarrow \min_{A,c}
\end{aligned}
\tag{8.18}
$$

inner state vector

To describe how the model evolves, we explain the state equations step by step: We start with a state vector $s_{\tau-1}$ which has the structure of 8.16. Through the multiplication with the additional consistency matrix C the state vector is transformed into a vector with the partitioning

$$
\tilde{s}_\tau = \begin{bmatrix} y_\tau^d \\ h_\tau \\ e_\tau \end{bmatrix} = \begin{bmatrix} \text{observations} \\ \text{hidden states} \\ \text{error correction} \end{bmatrix} \tag{8.19}
$$

for all $\tau \in \{t-m, \ldots, t+n\}$. This inner state vector $\tilde{s}_\tau$ contains the observables and the error correction and combines the ideas of DCRNN and DCECNN. The rest of the recursion is identical with the DCRNN (eq. 8.11). As before, the only learnable parameters of the network are located in matrix A and the bias c.

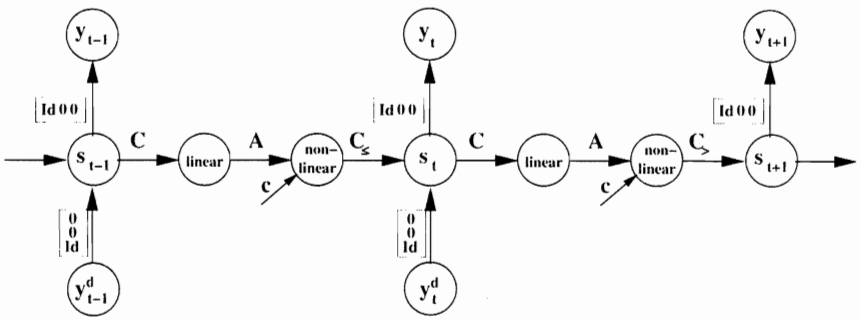


Figure 8.9 Dynamical consistent neural network (DCNN).

DCNN2

As already mentioned, the second approach to a dynamical consistent neural network (DCNN2) is based on the DCECNN model (eq. 8.14). The state vector s_τ

assumes the corresponding structure (see eq. 8.15):

$$s_\tau = \begin{bmatrix} y_\tau \\ h_\tau \\ \left\{ \begin{array}{ll} \tau \leq t: & e_\tau \\ \tau > t: & 0 \end{array} \right\} \end{bmatrix} = \begin{bmatrix} \text{expectations} \\ \text{hidden states} \\ \left\{ \begin{array}{ll} \tau \leq t: & \text{error correction} \\ \tau > t: & 0 \end{array} \right\} \end{bmatrix}. \tag{8.20}$$

Analogous to the development of the DCNN1 (eq. 8.18) the DCECNN equation is extended by an additional consistency matrix C, which now has the structure

$$C = \begin{bmatrix} Id & 0 & Id \\ 0 & Id & 0 \\ 0 & 0 & Id \end{bmatrix}. \tag{8.21}$$

The resulting DCNN2 can be described with the following set of equations:

$$\begin{aligned} \tau \leq t: \; s_\tau &= \begin{bmatrix} Id & 0 & 0 \\ 0 & Id & 0 \\ -Id & 0 & 0 \end{bmatrix} \tanh \left(A \begin{bmatrix} Id & 0 & Id \\ 0 & Id & 0 \\ 0 & 0 & Id \end{bmatrix} s_{\tau-1} + c \right) + \begin{bmatrix} 0 \\ 0 \\ Id \end{bmatrix} y_\tau^d \\ \tau > t: \; s_\tau &= \begin{bmatrix} Id & 0 & 0 \\ 0 & Id & 0 \\ 0 & 0 & 0 \end{bmatrix} \tanh \left(A \begin{bmatrix} Id & 0 & Id \\ 0 & Id & 0 \\ 0 & 0 & Id \end{bmatrix} s_{\tau-1} + c \right) \\ y_\tau &= [Id \; 0 \; 0] \; s_\tau, \qquad \sum_{t=m}^{T-n} \sum_{\tau=t-m}^{t+n} (y_\tau - y_\tau^d)^2 \to \min_{A,c} \end{aligned} \tag{8.22}$$

Looking at the multiplication $C \cdot s_{\tau-1}$ we can easily confirm that – supposing that $s_{\tau-1}$ is structured as in 8.20 – we once again get an inner state vector $\tilde{s}_\tau$ partitioned as in 8.19.

This implies that the transition matrix A is applied in both models to the same inner state vector $\tilde{s}_\tau$. Consequently, although the two models look quite different, they share an identical modeling of the dynamics. It may depend on additional modeling tools or a particular application which approach is preferable.

The network architecture for the alternative approach, DCNN2, is identical to DCNN1 (fig. 8.8), but note that the consistency matrices C, $C_\leq$, and $C_>$ differ.

advantages of the DCNN

Opposite to the DCRNN (eq. 8.11) and DCECNN (eq. 8.14) the two approaches to the DCNN (eqs. 8.18 and 8.22) compute the state trajectory of the dynamics in the past exactly on the observed path. This follows from the partitioning of the inner state vector $\tilde{s}_\tau$ (eq. 8.19), which is responsible for the calculation of the dynamics. It contains the observables in the first r components, which are directly used to determine the prediction y_τ. The error corrections, which are now located in the last r components, act as additional inputs. Furthermore, the DCNN offers an

interesting new insight into the observation of dynamical systems: Typically, small movements of the dynamics are treated as noise, and thus the modeling focuses on larger shifts in the dynamics. Our view is different. We believe that small system changes characterize the autonomous part of our open system, while the large swings originate at least partially from the external forces. If we neglect small system changes, we also suppress valuable substructure in our observations. We found that DCNNs allow us to model even small changes in the dynamics without losing the generalization abilities of the model. This introduces a new perspective on the structure/noise dilemma in modeling dynamical systems.

8.3.5 Partially Known Observables

So far our models have always distinguished between a past and a future development of the state equation. We assumed that in the past part ($\tau \leq t$) all the identified observables are available. In the future part ($\tau > t$) we accepted that we do not know anything about the observables and hence we replaced them by the model's own expectations.

In many practical applications we have observables which are not available for all time steps in the past. In contrast, one might have observables which are also available in the future, e.g., calendar data. In the following we therefore switch from a model differentiating between past and future to a modeling structure which distinguishes between available and missing external inputs.

The DCNN with partially known observables merges the two state equations of the DCNN (e.g., eq. 8.22) into one single equation that allows us to differentiate between available and unavailable observables. Consequently, it is a reformulation of the normal DCNN providing an easier and more general structure. The simplification in one equation makes the model also more tractable for further discussions (see sections 8.4 and 8.5). The following model (eq. 8.23) is based on the DCNN2 (eq. 8.22), but an analogous model can also easily be created for DCNN1 (eq. 8.18). For all $\tau \in \{t-m, \ldots, t+n\}$, we have

$$\begin{aligned} s_\tau &= \begin{bmatrix} Id & 0 & 0 \\ 0 & Id & 0 \\ E & 0 & 0 \end{bmatrix} \tanh\left(A \begin{bmatrix} Id & 0 & Id \\ 0 & Id & 0 \\ 0 & 0 & Id \end{bmatrix} s_{\tau-1} + c \right) + \begin{bmatrix} 0 \\ 0 \\ Id \end{bmatrix} y_\tau^E \\ y_\tau &= [Id \ 0 \ 0] \ s_\tau, \qquad \sum_{t=m}^{T-n} \sum_{\tau=t-m}^{t+n} (y_\tau - y_\tau^d)^2 \to \min_{A,c}. \end{aligned} \tag{8.23}$$

In this model the external inputs y_τ^E and the included matrix E are defined as follows:

$$y_\tau^E := \left\{ \begin{array}{ll} 0 & \text{if input missing} \\ y_\tau^d & \text{input available} \end{array} \right\} \tag{8.24}$$

and

$$E := \left\{ \begin{array}{c} 0 \\ -1 \end{array} \text{ if } \begin{array}{l} \text{input missing} \\ \text{input available} \end{array} \right\}. \tag{8.25}$$

It is important to note that the inner consistency matrix C is independent of the input availability. We only adapt the consistency matrix

$$C_E = \begin{bmatrix} Id & 0 & 0 \\ 0 & Id & 0 \\ E & 0 & 0 \end{bmatrix}. \tag{8.26}$$

The structure guarantees that an error correction is calculated in the last r state components if external input is available. Thus, we have a time-independent combination of the former two state equations (eq. 8.22). The corresponding model architecture (fig. 8.10) does not change significantly in comparison to the former (time-oriented) DCNN (fig. 8.9).

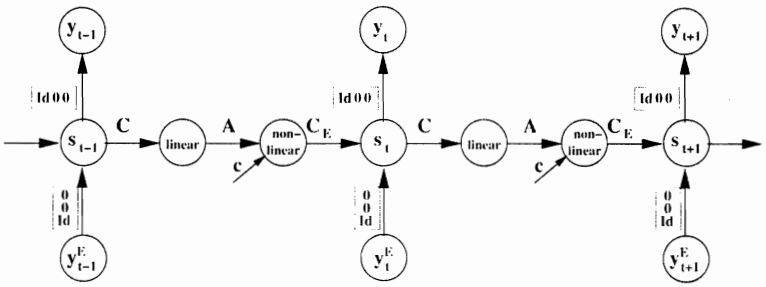


Figure 8.10 DCNN with partially known observables.

The DCNN with partially known observables is more general in the sense of observable availability and hence better applicable to real-world problems. The following discussions are mainly based on this model.

8.4 Handling Uncertainty

In practical applications our models have to cope with several forms of uncertainty. So far we have neglected their possible influence on generalization performance. Uncertainty can disturb the development of the internal dynamics and seriously harm the quality of our forecasts. In this section we present several methods which reduce the model's dependency on uncertain data.

There are actually three major sources of uncertainty. First, the input data itself might be corrupted or noisy. We deal with that problem in section 8.4.1. In the framework of finitely unfolded in time recurrent neural networks we also have

the uncertainty of the initial state. We present different approaches to overcome that uncertainty and achieve a desensitization of the model from the unknown initialization (see section 8.4.2). Finally, we discuss the uncertainty of the future inputs and question once more the assumption of a constant environment (see section 8.4.3).

8.4.1 Handling Data Noise

So far we have always assumed our input data to be correct. In most practical applications this is not true. In the following we present an approach which tries to minimize input uncertainty.

cleaning noise

Cleaning noise is a method which improves the model's learning behavior by correcting corrupted or noisy input data. The method is an enhancement of the *cleaning* technique which is described in detail in (Neuneier and Zimmermann, 1998). In short, cleaning considers the inputs as corrupted and adds corrections to the inputs if necessary. However, we want to keep the cleaning correction as small as possible. This leads to an extended error function

$$E_t^{y,x} = \frac{1}{2}[(y_t - y_t^d)^2 + (x_t - x_t^d)^2] = E_t^y + E_t^x \quad \rightarrow \min_{x_t, w} \quad . \tag{8.27}$$

Note that this new error function does not change the usual weight adaption rule

$$w^+ = w - \eta \frac{\partial E^y}{\partial w} \quad , \tag{8.28}$$

where $\eta > 0$ is the so-called learning rate and w^+ stands for the adapted weight. To calculate the cleaned input

$$x_t = x_t^d + \rho_t \tag{8.29}$$

we need the correction vectors ρ_t for all input data of the training set. The update rule for these corrections, initialized with $\rho_t = 0$, can be derived from typical adaption sequences:

$$x_t^+ = x_t - \eta \frac{\partial E^{y,x}}{\partial x}, \tag{8.30}$$

leading to

$$\rho_t^+ = (1 - \eta)\rho_t - \eta \frac{\partial E^y}{\partial x} \quad . \tag{8.31}$$

This is a nonlinear version of the error-in-variables concept from statistics.

We derive all the information needed, especially the residual error $\frac{\partial E^{y,x}}{\partial x}$, from training the network with backpropagation (fig. 8.18), which makes the computational effort negligible. It is important to note that in this way the corrections are performed by the model itself and not by applying external knowledge (see "observer-observation dilemma" in Neuneier and Zimmermann, 1998).

We now assume that the data is not only corrupted but also noisy. For that

reason we add an extra noise vector, $-\rho_\tau$, to the cleaned value:

$$x_t = x_t^d + \rho_t - \rho_\tau \quad . \tag{8.32}$$

The noise vector ρ_τ is a randomly chosen row vector $\{\rho_{i\tau}\}_{i=1,\ldots,r}$ of the cleaning matrix

$$C_{Cleaning} := \begin{bmatrix} \rho_{11} & \cdots & \cdots & \cdots & \rho_{r1} \\ \rho_{12} & \ddots & & & \rho_{r2} \\ \vdots & & \rho_{it} & & \vdots \\ \vdots & & & \ddots & \vdots \\ \rho_{1T} & \cdots & \cdots & \cdots & \rho_{rT} \end{bmatrix} .$$

which stores the input error corrections of all data patterns. The matrix has the same size as the pattern matrix, as the number of rows equals the number of patterns T and the number of columns equals the number of inputs r.

One might wonder why disturb the cleaned input $x_t = x_t^d + \rho_t$ with an additional noise-term $-\rho_\tau$. The reason for this is, that we want to benefit from representing the whole input distribution to the network instead of only using one particular realization (Zimmermann and Neuneier, 1998).

local cleaning noise

A variation on the Cleaning Noise method is called *local cleaning noise*. Cleaning noise adds to every training pattern the same noise term $-\rho_\tau$ and therefore assumes that the noise of the different inputs is correlated. Especially in high-dimensional models it is improbable that all the components of the input vector follow an identical or at least correlated noise distribution. For these cases we propose a method which is able to differentiate component-wise:

$$x_{it} = x_t^d + \rho_t - \rho_{i\tau}. \tag{8.33}$$

In contrast to the normal cleaning technique, the local version is correcting each component of the input vector x_{it} individually by a cleaning correction and a randomly taken entry $\rho_{i\tau}$ of the corresponding column $\{\rho_{it}\}_{t=1,\ldots,T}$ of the cleaning matrix $C_{Cleaning}$.

A further advantage of the local cleaning technique is that —with the increased number of (local) correction terms $(T \cdot r)$ we can cover higher dimensions. In contrast, with the normal cleaning technique the dimension is bounded by the number of training patterns T, which can be insufficient for high-dimensional problems.

8.4.2 Handling the Uncertainty of the Initial State

One of the difficulties with finite unfolding in time is to find a proper initialization for the first state vector of the recurrent neural network. An obvious solution is to set the first state s_0 to zero. We then implicitly assume that the unfolding includes

enough (past) time steps such that the misspecification of the initialization phase is compensated along the state transitions. In other words, the network accumulates information over time, and thus can eliminate the impact of the arbitrary initial state on the network outputs.

state initialization

The model can be improved if we make the unfolded recurrent network less sensitive to the unknown initial state s_0. For this purpose we look for an initialization, for which the interpretation of the state recursion is consistent over time. Since the initialization procedure is identical for all types of DCNNs, we demonstrate the approach on the DCNN with partially known observables (eq. 8.23):

$$\begin{aligned} s_\tau &= C_E \tanh(A \cdot C \cdot s_{\tau-1} + c) + \begin{bmatrix} 0 \\ 0 \\ Id \end{bmatrix} y_\tau^E \\ y_\tau &= [Id\ 0\ 0] s_\tau. \end{aligned} \qquad \sum_{t=m}^{T-n} \sum_{\tau=t-m}^{t+n} (y_\tau - y_\tau^d)^2 \to \min_{A,c}. \tag{8.34}$$

In a first step we explicitly integrate a first state vector s_0 (see fig. 8.11). This is no longer set to zero but receives the target information y_{t-1}^{target} of the first output vector y_{t-1}. The vector is then multiplied by the consistency matrix C_E, such that the first r components of the first state vector s_{t-1} coincide with the first expected output. This avoids the generation of an excessively large error for the first output.

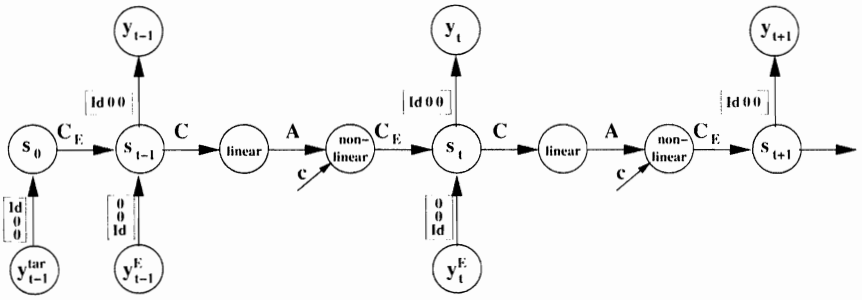


Figure 8.11 Time consistent initialization of a DCNN with an additional initial state s_0.

The hidden states of this model are arbitrarily initialized with zero. In a second step we add a noise term ε to the first state vector s_0 to stiffen the model against the uncertainty of the unknown initial state. A fixed noise term ε that is drawn from a predetermined noise distribution is clearly inadequate to handle the uncertainty of the initial state. Instead we apply according to the cleaning noise method—an adaptive noise term, which fits best the volatility of the unknown initial state s_0. As explained in section 8.4.1, the characteristics of the adaptive noise term are automatically determined as a by-product of the error backpropagation algorithm.

residual error

The basic idea is as follows: The residual error ρ as measured at the initial state s_0 can be interpreted as the uncertainty stemming from missing information about the true initial state vector. If we disturb s_0 with a noise term which follows the distribution of the residual error of the network, we diminish the uncertainty about the unknown initial state during system identification. In addition, this allows a better fitting of the target values over the training set. A corresponding network architecture is depicted in fig. 8.12.

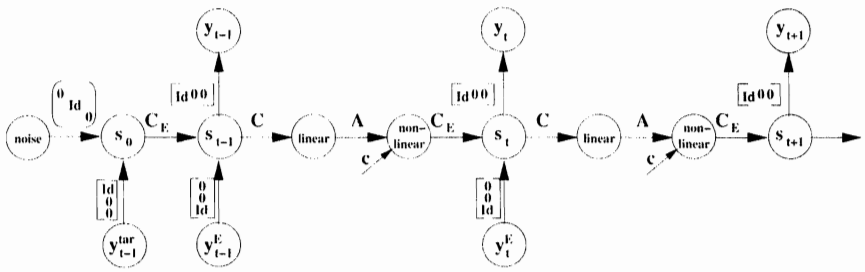


Figure 8.12 Desensitization of a DCNN from the unknown initial state s_0.

Technically, *noise* is introduced into the model via an additional input layer. The dimension of *noise* is equal to that of the internal state. The input values are fixed at zero over time. Due to the incomplete identity matrix between *noise* and the initial state the noise is only applied to the hidden values of the initial state, where no input information is available. The desensitization of the network from the initial state vector s_0 can therefore be seen as a self-scaling stabilizer of the modeling. Note that the noise term ρ is drawn randomly from the observed residual errors, without any prior assumption on the underlying noise distribution.

In general, a discrete-time state trajectory forms a sequence of points over time. Such a trajectory is comparable to a thread in the internal state space. The trajectory is very sensitive to the initial state vector s_0. If we apply noise to s_0, the space of all possible trajectories becomes a tube in the internal state space (fig. 8.13). Due to the characteristics of the adaptive noise term, which decreases over time, the tube contracts. This enforces the identification of a stable dynamical system. Consequently, the finite volume trajectories act as a regularization and stabilization of the dynamics.

initialization techniques

The question arises, what may be the best method to create an appropriate noise level. Table 8.1 gives an overview of several initialization techniques we have developed and examined so far. Remember that in all cases the corrections are only applied to the hidden variables of the initial state s_0.

We already explained the first three methods in section 8.4.1. The idea behind the initialization with *start noise* is, that we do not need a cleaning correction but solely focus on the noise term. *double start noise* tries to achieve a nearly symmetrical noise distribution, which is also double in comparison to normal *start*

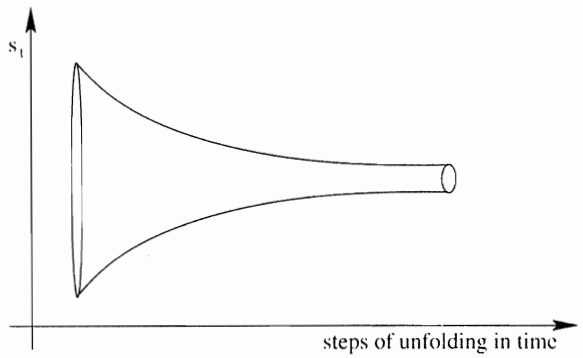


Figure 8.13 Creating a tube in the internal state space by applying noise to the initial state.

Table 8.1 Overview of Initialization Techniques

Cleaning:	$s_0 = 0 + \rho_t$
Cleaning noise:	$s_0 = 0 + \rho_t - \rho_\tau$
Local Cleaning noise:	$s_{0_i} = 0 + \rho_t - \rho_{\tau_i}$
Start noise:	$s_0 = 0 + \rho_\tau$
Local start noise:	$s_{0_i} = 0 + \rho_{\tau_i}$
Double start noise:	$s_0 = 0 + (\rho_\tau^1 - \rho_\tau^2)$
Double local start noise:	$s_{0_i} = 0 + (\rho_{\tau_i}^1 - \rho_{\tau_i}^2)$

noise. In all cases *local* always corresponds to the individual application of a noise term to each component of the initial state s_0 (see *local cleaning noise* in section 8.4.1).

From top to bottom the methods listed in table 8.1 use less and less information about the training set. Hence *double start noise* emphasizes more the generalization abilities of the model. This is also confirmed by our experiments. Furthermore we could confirm that the local initialization techniques lead to better performance in high-dimensional models (see section 8.4.1).

8.4.3 Handling the Uncertainty of Unknown Future Inputs

In the past part of the network, the influence of the unknown externals is reflected in the error corrections as calculated by the backpropagation algorithm. In the future part we do not have any information about the correctness of our inputs. As explained in section 8.3, we either use our own forecasts as future inputs, or simply assume that the inputs in the future stay constant. The underlying assumption is that the observables evolve in the future like they did in the past. We cannot verify if this is correct. Anyway, for most practical applications it is a very questionable assumption.

To stabilize our model against these uncertainties of the future inputs we apply a Gaussian noise term $\varepsilon_{t+\tau}$ to the last r components of each future state vector $s_{t+\tau}$. The corresponding architecture is depicted in fig. 8.14.

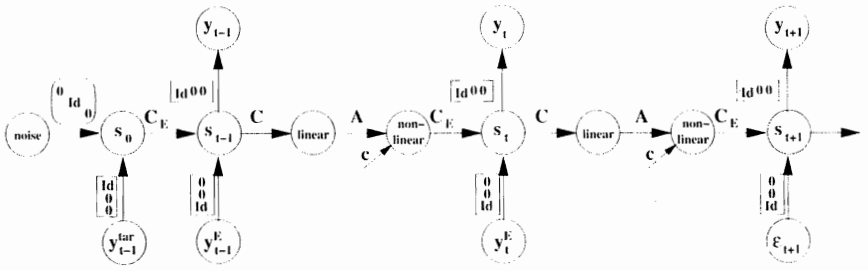


Figure 8.14 Handling the uncertainty of the future inputs by adding a noise term $\varepsilon_{t+\tau}$ to each future state vector $s_{t+\tau}$.

The additional noise is used during the training of the model to achieve a more stable output. For the actual deterministic forecast we either skip the application of noise to avoid a disturbance of the predictions or average our results over a sufficient number of different forecasts (Monte Carlo approach).

8.5 Function and Structure in Recurrent Neural Networks

Our discussion about function and structure in recurrent neural networks is focused on the autonomous part of the model, which is mapped by the internal state transition matrix A. So far the transition matrix A has always been assumed to be fully connected. In a fully connected matrix the information of a state vector s_t is processed using the weights in A to compute s_{t+1}. This implies that there is a high proportion of superposition (computation) but hardly any conservation of information (memory) from one state to a succeeding one (see the right panel of fig. 8.15).

For the identification of dynamical systems such memory can be essential, as information may be needed for computation in subsequent time steps. A shift register (see the left panel of fig. 8.15) is a simple example for the implementation of memory, as it only transports information within the state vector s. No superposition is performed in this transition matrix.

superposition and conservation of information

At first view we have two contradicting functions: superposition and conservation of information. Superposition of information is necessary to generate or adapt changes of the dynamics. In contrast, conservation of information causes memory effects by transporting information more or less unmodified to a subsequent state neuron. In this context, memory can be defined as the average number of state transitions necessary to transmit information from one state neuron to any other

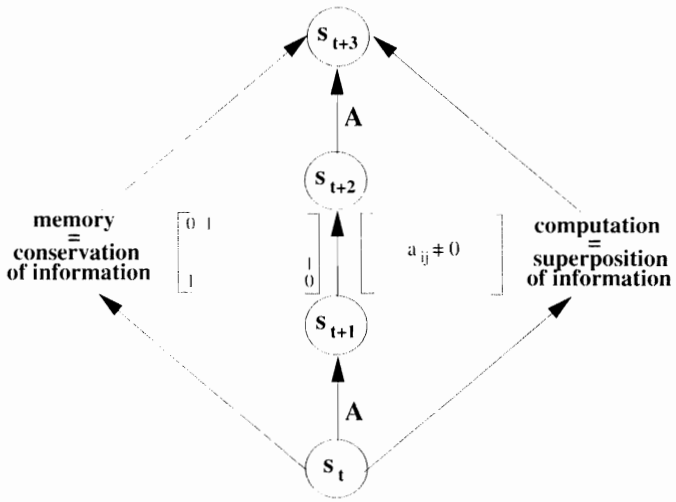


Figure 8.15 Function and structure in dynamical systems: computation versus memory in the transition matrix A.

one in a subsequent state. We call this number of necessary state transitions the *path length* of a neuron. To overcome the apparent dilemma between superposition and conservation of information the transition matrix A needs a structure which balances memory and computation effects. Sparseness of the transition matrix reduces the number of paths and the computation effect of the network but at the same time increases the average path length, and therefore allows for longer-lasting memory. A possible solution is an inflation of the recurrent network, i.e., of the transition matrix A.

We show that with such an inflation an optimal balance between memory and computation can be achieved (section 8.5.1). In this context we present conjectures about the optimal level of sparseness and the required minimum dimension. An experiment with artificial data underlines our results (section 8.5.2). In section 8.5.3 we conclude that sparseness is actually an essential condition for high-dimensional neural networks. Finally, we discuss in section 8.5.4 the information flow in sparse networks.

8.5.1 Inflation of Recurrent Neural Networks

Based on the length of the past unfolding m (see section 8.2.2) and the optimal state dimension $\dim(s)$ of a fully connected recurrent network, we can define a procedure for an optimal design of the neural network structure, which solves the dilemma between memory and computation.

The idea is to inflate the network to a higher dimensionality, while maintaining the computational complexity of the former lower-dimensional and fully connected network, and at the same time allowing for memory effects. With an inflated transi-

tion matrix A we can optimize both superposition and conservation of information.

optimal inflation

To determine the optimal dimension and the level of sparseness, we propose two conjectures, which we will empirically investigate in section 8.5.2. In a first step we calculate the new dimension of the internal state s by

$$\dim(s_{new}) := m \cdot \dim(s). \tag{8.35}$$

As the former dimension of s was supposed to be optimal, we have to ensure that the higher-dimensional network has the same superposition of information as the original one. This can be achieved by keeping the number of active weights constant. On average we want to have the same number of nonzero elements as in the former lower-dimensional network. Thus, the sparseness level of the new matrix A_{new} is given by

$$\text{initialize } A_{new} \text{ with } \quad Random\left(\frac{\dim(s)}{\dim(s_{new})}\right) = Random\left(\frac{1}{m}\right). \tag{8.36}$$

Hereby $Random(\cdot)$ represents the percentage of randomly initialized weights, whereas the remaining weights are set to zero. Proceeding this way, we replicate on average the computation effect of the former network. At the same time we increase the path lengths (memory) with the sparseness level of the new transition matrix A_{new}. Note that the sparseness level only depends on the length of the past unfolding m.

The conjecture (eq. 8.36) implies that the sparseness of A_{new} is generated randomly. In section 8.5.3 we present techniques which try to optimize the sparse structure and consequently the memory and computation abilities of the network.

training procedure for RNNs

Based on our conjectures about inflation, a proper training procedure for recurrent neural networks should consist of four steps: First, one has to set up an appropriate network architecture (e.g., DCNN, eq. 8.23). Second, the length of the past unfolding m and the optimal internal state dimension $\dim(s)$ of the system have to be estimated by analyzing the network errors along the time steps of the unfolding (see section 8.2.2). Third, we use the estimated parameters m and $\dim(s)$ to determine the optimal dimensionality and sparseness (eqs. 8.35 and 8.36). Fourth, the inflated network is trained until convergence by backpropagation through time using, e.g., the *vario-eta* learning rule (Neuneier and Zimmermann, 1998).

8.5.2 Experiments: Testing Conjectures About Inflation

In the following experiments we want to evaluate our conjectures about optimal inflation of recurrent networks. To ensure a straight analysis of our proposed equations (eqs. 8.35 and 8.36), we modeled an artificial network which consists of an

autonomous development only. We applied the network to forecast the development of the following artificial data generation process

$$s_t = \tanh(A \cdot s_{t-m}) + \epsilon_t \ , \tag{8.37}$$

where $\dim(s) = 5$, A is randomly initialized, $m = 3$, and ϵ is white noise with $\sigma_\epsilon = 0.2$, one time step ahead. The unfolding in time of the recurrent network includes five time steps from $t-3$ to $t+1$. As the data generation process is a closed dynamical system, there are no inputs, but time-delayed states s_{t-k} $(k = 1, \ldots, m)$ are used as external influences.

Each of the following experiments is based on 100 Monte Carlo simulation runs. For each run we generated 1000 observations, 25% for training and 75% for testing purposes. The network was trained until convergence with error backpropagation through time using *vario-eta* learning (Neuneier and Zimmermann, 1998).

First we evaluated our conjecture about the sparse random initialization of the transition matrix A_{new}. For this purpose we randomly initialized matrix A_{new} with different levels of sparseness (100 test runs per sparseness degree). The dimension of the internal state was fixed according to 8.35 at $\dim(s_{new}) = 3 \cdot 5 = 15$ for all test runs. The mean square error of the network measured on the test set was used as a performance criterion.

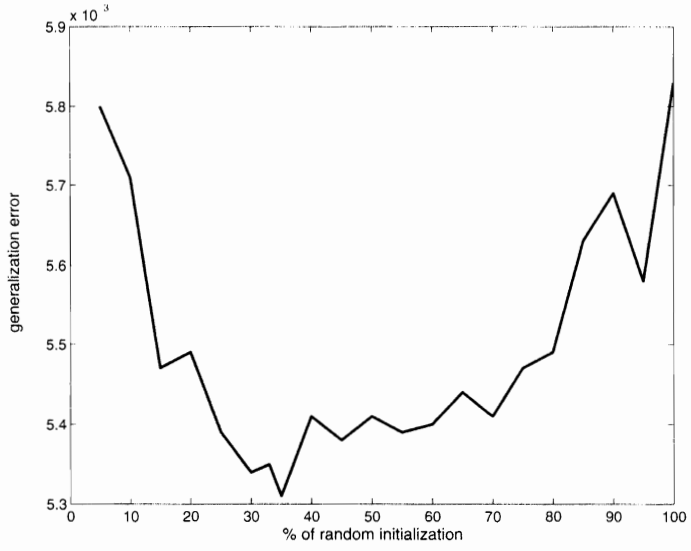


Figure 8.16 Effects of different degrees of sparseness on matrix A_{new}.

The results of the experiment (Figure 8.16) confirm our conjecture about an optimal sparseness level: If we initialize matrix A_{new} randomly 35% sparse, we observe the best performance (i.e., lowest average error on the test set). This corresponds to equation 8.36, where an optimal sparseness level computes to 33%.

The second series of experiments was connected with the optimal internal state dimension. During these experiments we kept the sparseness level constant at 33% (see eq. 8.36), whereas the dimension of the internal state was variable. We performed 100 runs for each dimension of the internal state with different random initializations of matrix A_{new}. Again we used the network error as an indicator of the model performance. The results are shown in fig. 8.17.

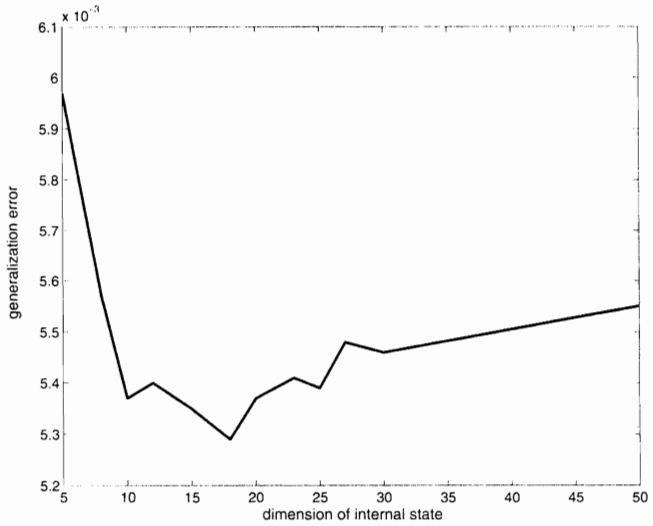


Figure 8.17 Impacts of different internal state dimensions $\dim(s_{\text{new}})$.

It turns out that the best performance is achieved if the dimension of the internal state is equal to $\dim(s_{new}) = 18$. Our conjecture of $\dim(s_{new}) = 3 \cdot 5 = 15$ (eq. 8.35) slightly underestimates the empirically measured optimal dimensionality. However, because of the noise term ϵ_t, we suppose that the optimal dimension of the system is larger than 5. This indicates that our conjecture in 8.35 is a helpful estimate of an optimal level of sparseness.

Both experiments show that mismatches between dimensionality and sparseness cause problems in the function (superposition and conservation) of the transition matrix. In other words, an unbalanced parameterization of the inflation leads to lower generalization performance of the network.

8.5.3 Sparseness as a Necessary Condition for Large Systems

One might come up with the idea of initializing a model with a fully connected transition matrix A, and then pruning it during the learning process until a desired degree of sparseness is reached. This approach is misleading, as sparseness is an essential condition for the performance of the backpropagation algorithm in large networks.

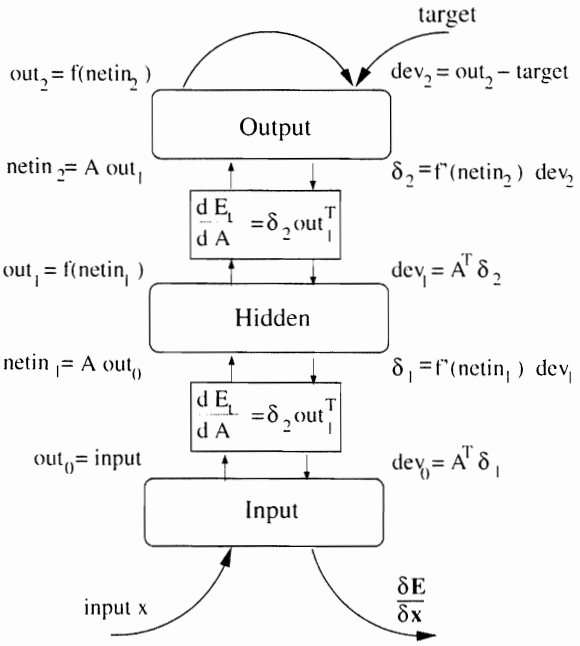


Figure 8.18 Forward and backward information flow in the backpropagation algorithm.

backpropagation

Figure 8.18 shows the forward and backward information flow in the backpropagation algorithm. [3] When we look at the calculations, it becomes obvious that if the transition matrix A is fully connected and $dim(s)$ is increasing, we get a growing number as well as a lengthening of the sums in the matrix times vector operations. Due to the law of large numbers the probability for large sum values also increases. This does not pose any problems in the forward flow of the algorithm. The hyperbolic tangent as the nonlinear activation function guarantees that the calculated values stay numerically tractable. In contrast, backward information flow is linear. In this part of the algorithm large values are spread all over a fully connected matrix A. They quickly sum up to values which cause numerical instabilities and may destroy the whole learning process. This can be avoided if we use a sparse transition matrix A. The number of summands is then smaller and therefore the probability of large sums is low.

sparse initialization of A

In the remainder of this section we want to discuss the question of how to choose a sparse transition matrix A that is still trainable to a stable model.

One intuitive answer is to initialize the model several times and then compare the different results. We performed 100 test runs of the neural network using different random initializations of matrix A. The prestructuring of the network followed our conjectures about inflation (eqs. 8.35 and 8.36).

An obvious approach to overcome the uncertainty of the random initialization is to pick the best-performing network out of the 100 test runs. However, it is

not clear if 100 test runs are effectively required to find an appropriate model. To study how many test runs are needed to find a good solution with a minimum of computational effort, we picked all possible subsets of $k = 1, 2, \ldots, 100$ solutions out of the 100 test runs. From each subset we chose the solution with the lowest error on the test set and computed the average performance:

$$E_k = \frac{1}{\binom{100}{k}} \sum_{i_1 \leq \ldots \leq i_k} \min(e_{i_1}, e_{i_2}, \ldots, e_{i_k}). \tag{8.38}$$

The resulting average error curve for $k = 1, 2, \ldots, 100$ is depicted in fig. 8.19 (solid line).

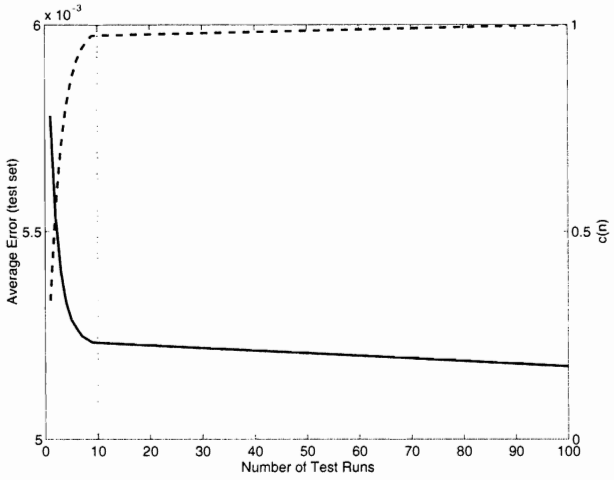


Figure 8.19 Estimating the number of random initializations for matrix A.

As can be seen from the error curve in fig. 8.19, an appropriate solution can be obtained on the average by choosing the best model out of a subset of 10 networks (vertical dotted line). Of course, the performance is worse than picking the best model out of 100 solutions. However, the additional computational effort does not justify the small improvement of performance.

coverage percentage

As an apparently easier guideline to determine the number of required test runs, we choose the number k such that the so-called coverage percentage,

$$c(k) = 1 - \left(1 - \frac{1}{m}\right)^k, \tag{8.39}$$

is close to 1. The idea behind the coverage percentage $c(k)$ in 8.39 is that the first inflated network covers $c(1) = 1/m$ active elements in the internal transition matrix A_{new}. Assuming that we have $c(k)$, the next initialization covers another percentage of the weights in the transition matrix, resulting in $c(k+1) = c(k) + (1/m) \cdot (1 - c(k))$. The coverage percentage $c(k)$ for the different numbers of initializations is also

reported in fig. 8.19 (dashed line). A number of $k = 10$ random initializations already leads to a coverage of $c(10) \approx 0.983$.

pruning & re-creation

To further reduce the computational effort, we developed a more sophisticated approach, a process we call *pruning and re-creation of weights*. As described in section 8.5.1, we initialize matrix A with a sparseness level of $Random(1/m)$ (eq. 8.36). The idea is now to optimize the initial sparse structure by alternating weight pruning and re-creation. Using this method, matrix A is always sparse and the number of active weights stays constant. The network still gets the opportunity to replace active weights by initially inactive ones that it considers more important for the identification of the dynamics.

For the first step, the weight pruning, we use a test criterium similar to *optimal brain damage* (OBD) (LeCun et al., 1990):

$$test_w(w \neq 0) = \frac{\partial^2 E}{\partial w^2} w^2. \tag{8.40}$$

We prune a certain percentage (e.g., 5%) of the lowest values, as these weights w are assumed to be less important for the identification of the dynamics. To simplify our calculations we use

$$\frac{\partial^2 E}{\partial w^2} \approx \frac{1}{T} \sum_t g_t^2, \tag{8.41}$$

with $g_t := \frac{\partial E_t}{\partial w}$, as an approximation for the second derivative. Our simulations showed that this equivalence holds for a 95% level.

In the second step, the re-creation of inactive weights, we use the following test:

$$test_w(w = 0) \sim \frac{1}{T} \left| \sum_t g_t \right|. \tag{8.42}$$

We reactivate the weights w with the highest test values. This implies that we recover weights whose average of the absolute gradient information is high and which are therefore considered important for the identification of the dynamics. Note that we always re-create the same amount of weights we pruned in the first step to keep the sparseness level of the transition matrix A constant.

Our experiments showed that we can even prune and re-create weights simultaneously without losing modeling ability.

8.5.4 Information Flow in Sparse Recurrent Networks

In small networks with a full transition matrix A the information of a state neuron can reach every other one within one time step. This is different in (large) sparse networks, where state neurons have a longer path length on average.

As matrix A is sparse there is in most cases no direct connection between different state neurons. Hence, it can take several state transitions to transport information from one state neuron to another. As information might not reach a

desired neuron in a limited number of time steps, this can be disadvantageous for the modeling ability of the network. The resulting question is, how we can speed up the information flow, i.e., shorten the path length?

undershooting

In a simple recurrent network (e.g., eq. 8.9) the transition matrix A is applied once in every state transition. The idea is now to reduce the average path length with at least one additional undershooting step (Zimmermann and Neuneier, 2001). Undershooting means that we implement intermediate states $s_{\tau \pm \frac{1}{2}}$ which improve the computation of the network (fig. 8.20). These intermediate states have no external inputs. Like future unfolding time steps, they are only responsible for the development of the dynamics and therefore also improve numerical stability.

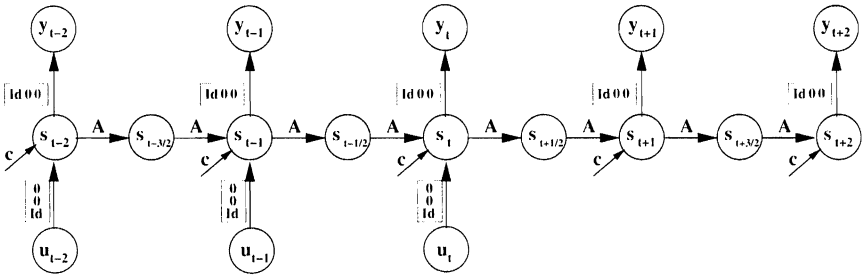


Figure 8.20 Undershooting improves the computation of a sparse matrix A and the numerical stability of the model.

The following formula gives a rough approximation of how many undershooting steps k are needed (eq. 8.43). As the state transition matrix A in the inflated network has, per definition (eq. 8.36), a sparseness of $1/m$, in each time step every state neuron only gets the information of approximately $1/m$ others. The equation now determines the number of undershooting steps which are needed to achieve a desired average path length (information flow) between all state neurons. The kth power of the product of the sparseness factor $1/m$ and the dimension of the state vector $\dim(s)$ must be higher than the number of state neurons ($\widehat{=}\dim(s)$):

$$\begin{aligned} & \left(\frac{1}{m} dim(s)\right)^k \geq dim(s) \\ \Rightarrow \quad & k \geq \frac{1}{1 - \frac{\log(m)}{\log(dim(s))}} \\ \Rightarrow \quad & k \geq 1 + \frac{\log(m)}{\log(dim(s))}. \end{aligned} \tag{8.43}$$

undershooting with DCNN

Let us reconsider the equations of the DCNN with partially known observables

(eq. 8.23):

$$\begin{aligned} s_\tau &= C_E \tanh\left(A \cdot C \cdot s_{\tau-1} + c\right) + \begin{bmatrix} 0 \\ 0 \\ Id \end{bmatrix} y_\tau^E \\ y_\tau &= [Id\ 0\ 0] s_\tau, \qquad \sum_{t,\tau} (y_\tau - y_\tau^d)^2 \to \min_{A,c}. \end{aligned} \tag{8.44}$$

Following the principle of undershooting, we add a state $s_{\tau-\frac{1}{2}}$ between the states $s_{\tau-1}$ and s_τ (fig. 8.21). Consequently, the matrix A is now applied twice between two consecutive time steps, which implies that the information flow is doubled. The consistency matrix C_E handles the lack of external inputs, such that the network stays dynamical consistent.

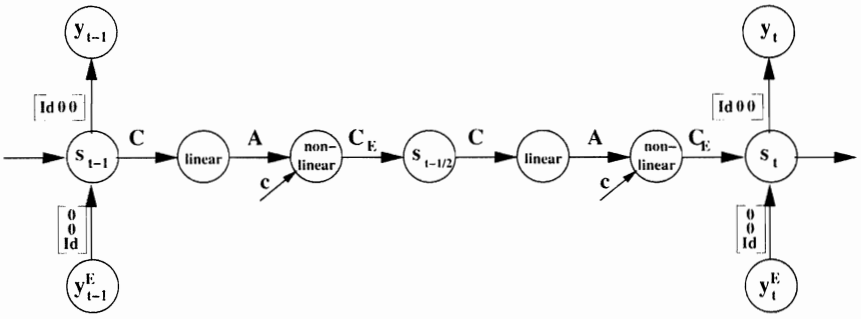


Figure 8.21 Undershooting doubles the information flow between two successive states by applying the transition matrix twice.

It is important to note that the solution is different from just decreasing the sparseness of the matrix A. The latter would not only cause numerical problems in the backpropagation algorithm but also disturb the balance between memory and computation.

8.6 Conclusion

In this chapter we focused on dynamical consistent neural networks (DCNN) for the modeling of open dynamical systems. After a short description of small recurrent networks, including error correction neural networks, we presented a new kind of dynamical consistent neural networks. These networks allow an integrated view on particular modeling problems and consequently show better generalization abilities. We concentrated the modeling of the dynamics on one single-transition matrix and also enhanced the model from a simple statistical to a dynamical consistent handling of missing input information in the future part. The networks are now able to inap

integrated system dynamics (e.g., financial markets) instead of only a small set of time series. The final DCNN combines the advantages of the former RNN and the ECNN.

Besides the new, more powerful architectures, the modeling involves a paradigm shift in the analysis of open systems (see fig. 8.1). In the beginning we looked at the description of a dynamical system from an *exterior* point of view. This means that we observed the information flow into and out of an open system and tried to reconstruct the interior. The long-term predictability of the model finally depended on the quality of the extracted autonomous subsystem.

world model

In our new approach we describe dynamical systems from an *interior* viewpoint. Conceptually we start with a *world model* (fig. 8.22). Without loss of generalization, we assume that our variables of interest are all organized as the first elements in a large state vector. Identifying this first section of the state vector as our observables (y_t), we can reconstruct some more unobservable states (h_t) by their indirect influence on the observables. Nevertheless, there are an infinite number of variables which are unobservable and even unidentifiable. Their nearly infinite influence can be shrunk to a finite dimensional section in the state vector: the error correction part (e_t). If the error correction is equal to zero, knowledge about the unidentifiable variables is not necessary. Otherwise, it dispenses us from having to know the details of the unknown part of the world.

Clearly, the concept of a world model is a closed one from the beginning. As a consequence of dynamical consistency the closure concept even holds for finite-dimensional subsections of it. Therefore it models the dynamics as a closed system and is still able to keep model evolution exactly on the observed state trajectory (see DCNN2, eq. 8.22).

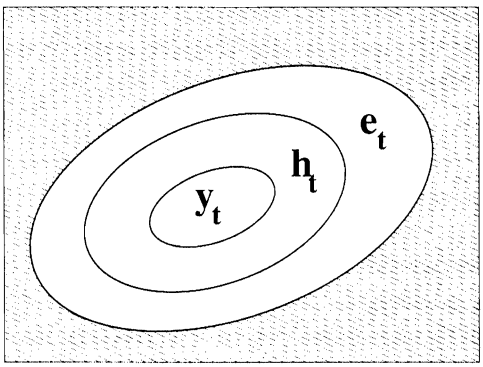


Figure 8.22 Variable space of the world model. y_t stands for the observables, h_t for the hidden variables which can be explained by the observables, and e_t for the error corrections, which close the gap between the observable and the unobservable part of the system.

We augmented the model-building process by incorporating prior knowledge. Learning from data is only one part of this process. The recurrent ECNN and the DCNN are two examples of this model-building philosophy. Remarkably, such a joint model-building framework does not only provide superior forecasts, but also a deeper understanding of the underlying dynamical system. On this basis it is also possible to analyze and to quantify the uncertainty of the predictions. This is especially important for the development of decision support systems.

Currently we test our models in several industrial applications. Further research is conducted concerning the optimal sparseness of the transition matrix A as well as the optimal initialization method for the state vector.

Acknowledgment

We thank J. Zwierz for the calculations and tests during our experiments in section 8.5. The extensive work performed by M. Pellegrino in proof reading this chapter is gratefully acknowledged. The computations were performed on our neural network modeling software SENN (Simulation Environment for Neural Networks), which is a product of Siemens AG.

Notes

[1] For other cost functions see Neuneier and Zimmermann (1998).
[2] For an overview of algorithmic methods see Pearlmutter (2001) and Medsker and Jain (1999).
[3] For further details, the reader is referred to Haykin (1994) and Bishop (1995).