

Learning Long-Term Dependencies with Recurrent Neural Networks

Anton Maximilian Schaefer^{a,b,*} Steffen Udluft^a
Hans-Georg Zimmermann^a

^a*Siemens AG, Corporate Technology, Information & Communications, Learning Systems
80200 Munich, Germany*

^b*University of Osnabrueck, Neuroinformatics Group, 49069 Osnabrueck, Germany*

Abstract

Recurrent neural networks (RNN) unfolded in time are in theory able to map any open dynamical system. Still, they are often blamed to be unable to identify long-term dependencies in the data. Especially when they are trained with backpropagation it is claimed that RNN unfolded in time fail to learn inter-temporal influences more than ten time steps apart. This paper refutes this often cited statement by giving counter-examples. We show that basic time-delay RNN unfolded in time and formulated as state space models are indeed capable of learning time lags of at least hundred time steps. We point out that they even possess a self-regularisation characteristic, which adapts the internal error backflow, and analyse their optimal weight initialisation. In addition, we introduce the idea of inflation for modelling of long and short-term memory and demonstrate that this technique further improves the performance of RNN.

Key words: Backpropagation, Inflation, Long Term Dependencies, Memory, Recurrent Neural Networks, State Space Model, System Identification, Vanishing Gradient

1 Introduction

Recurrent neural networks (RNN) allow the identification of dynamical systems in the form of high dimensional, nonlinear state space models [3,9]. They offer an explicit modelling of time and memory and are in principle able to model any

* Corresponding author.

Email addresses: Schaefer.Anton.ext@siemens.com (Anton Maximilian Schaefer), Steffen.Udluft@siemens.com (Steffen Udluft), Hans.Georg.Zimmermann@siemens.com (Hans-Georg Zimmermann).

open dynamical system [14]. Basic concepts like unfolding in time of RNN and related modifications of the backpropagation algorithm can already be found in [13]. Further developments are published in the books of Haykin [3,4], Medsker and Jain [10], Kolen and Kremer [9], and Soofi and Cao [15]. Different types of learning algorithms are summarised by Pearlmutter [12].

Nevertheless, over the last 20 years most time series problems have been approached with feedforward neural networks. The appeal of modelling time and memory in recurrent neural networks is opposed to the apparently better numerical tractability of a pattern-recognition approach as represented by feedforward neural networks. Besides, it has been claimed by several authors that RNN are unable to identify and learn long-term dependencies of more than ten time steps [1,5,6]. To overcome the stated dilemma new forms of recurrent networks, e.g., long short-term memory (LSTM) networks [7], were developed. Still, these networks do not offer the desirable correspondence, i.e., the mutual transferability, between equations and architectures as RNN unfolded in time do [18].

However, the analysis in the mentioned papers [1,5,6] were all based on basic RNN architectures simulating closed dynamical systems, which do not consider any external inputs. Even more important, they were made from a static perspective, which means that for the presented calculations only RNN with fixed weights were assumed whereas the effect of learning and weight adaption was not taken into account. In this paper we therefore refute the statement that RNN unfolded in time and trained with a shared weight extension of the backpropagation algorithm [13] are in general unable to learn long-term dependencies. We show that basic time-delay RNN and especially normalised recurrent neural networks unfolded in time and formulated as state space models have no difficulties with an identification and learning of past-time information within the data, which is more than ten time steps apart. In addition we point out that by using shared weights training of these networks is not a major problem. It even helps to overcome the problem of a vanishing gradient [1,5,6] as the networks possess a self-regularisation characteristic, which adapts the internal error backflow. We further analyse the effects of weight initialisation on the error flow. Here, we show that a vanishing or respectively exploding gradient [1,5,6] can be avoided by initialising the weights within an optimal interval.

Furthermore we extend the discussion about long-term dependencies to the modelling of long- and short-term memory as it is required for most real-world applications. For this, we analyse the internal network structure and show that an inflation, i.e., a combination of increasing the internal state dimension and the sparsity level, simplifies the mapping of different time scales and inter-temporal dependencies.

The paper starts with a recapitulation of the basic RNN unfolded in time (sec. 2). In section 3 we further enhance the basic RNN architecture so that it only possesses one single (high-dimensional) transition matrix. This so called normalised recurrent

neural network (NRNN) increases the stability of the learning process. In section 4 we then demonstrate that both NRNN and RNN successfully learn long-term dependencies. By analysing the backpropagated error flow we further show that the problem of a vanishing gradient is not a major question for both networks. In this context, we also investigate the influence of the weight initialisation. We extend the analysis about learning long-term dependencies to a more general discussion on modelling long and short-term memory (sec. 5). Finally, we demonstrate that inflated networks show a better performance than the smaller and fully connected ones. A conclusion and a brief outlook on future research is given in section 6.

2 Recurrent Neural Networks Unfolded in Time

Let I , J , and $N \in \mathbb{N}$ denote respectively the number of input, hidden and output neurons. For discrete time the basic time-delay recurrent neural network (RNN) consists of a state transition and an output equation [3,19]:

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

Here, the nonlinear state transition equation $s_{t+1} \in \mathbb{R}^J$ ($t = 1, \dots, T$ where $T \in \mathbb{N}$ is the number of available patterns) is a nonlinear combination of the previous state $s_t \in \mathbb{R}^J$ and external influences $x_t \in \mathbb{R}^I$ using weight matrices $A \in \mathbb{R}^{J \times J}$ and $B \in \mathbb{R}^{J \times I}$, and a bias $\theta \in \mathbb{R}^J$, which handles offsets in the input variables $x_t \in \mathbb{R}^I$. Note, that at this the hyperbolic tangent is applied component-wise. The network output $y_t \in \mathbb{R}^N$ is computed from the present state $s_t \in \mathbb{R}^J$ employing matrix $C \in \mathbb{R}^{N \times J}$. It is therefore a nonlinear composition applying the transformations A , B , and C . It has been shown that RNN, like feedforward neural networks [8], are universal approximators in the sense that they can approximate any open dynamical system of the form [3,19]

$$\begin{aligned} s_{t+1}^d &= g(s_t^d, x_t) && \text{state transition} \\ y_t^d &= h(s_t^d) && \text{output equation} \end{aligned} \tag{2}$$

where $g : \mathbb{R}^{\tilde{J}} \times \mathbb{R}^I \rightarrow \mathbb{R}^{\tilde{J}}$, with $\tilde{J} \in \mathbb{N}$, is a measurable and $h : \mathbb{R}^{\tilde{J}} \rightarrow \mathbb{R}^N$ a continuous function, $x_t \in \mathbb{R}^I$ represents the external inputs, $s_t^d \in \mathbb{R}^{\tilde{J}}$ the inner states and $y_t^d \in \mathbb{R}^N$ the output of the system [14]. Note here, that the state space of the RNN $s_t \in \mathbb{R}^J$ (eq. 1) does not in general have the same dimension as the one of the original open dynamical system $s_t^d \in \mathbb{R}^{\tilde{J}}$ (eq. 2), i.e., in most cases we have $J \neq \tilde{J}$. It basically depends on the system's complexity and the desired accuracy.

Training the RNN of equation (1) is equivalent to solving a parameter optimisation problem, i.e., minimising the error between the network output $y_t \in \mathbb{R}^N$ and the observed data $y_t^d \in \mathbb{R}^N$ with respect to an arbitrary error measure, e.g.:

$$\sum_{t=1}^T \|y_t - y_t^d\|^2 \rightarrow \min_{A,B,C,\theta} \quad (3)$$

It can be solved by finite unfolding in time using shared weight matrices A , B , and C [3,13]. 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 $\tau \in \{1, \dots, T\}$ and for every pattern t [3,13]. This guarantees that we have the same dynamics in every time step. By using unfolding in time the RNN can be trained with a shared weights extension [13] of the standard backpropagation algorithm [16]. A major advantage of RNN written in form of a state space model (eq. 1) is the explicit correspondence between equations and architecture. It is easy to see that by using unfolding in time the set of equations (1) can be directly transferred into a spatial neural network architecture (fig. 1) [3,13]. Here, the dotted connections indicate that the network can be (finitely) further unfolded into past and future.

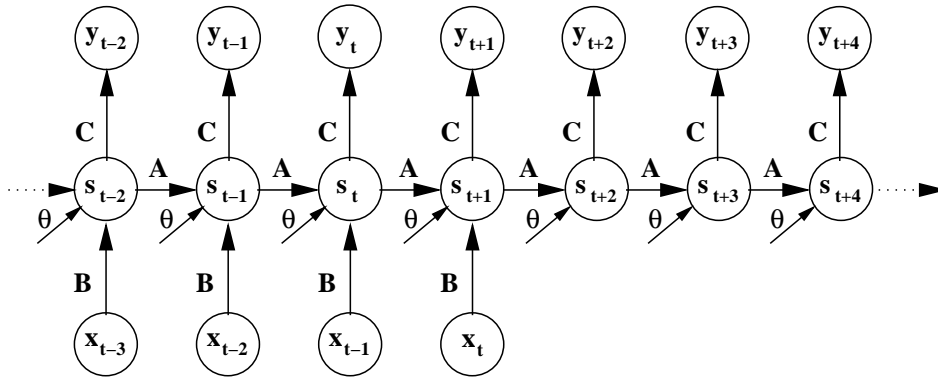


Fig. 1. RNN unfolded in time using overshooting [19].

We extend the autonomous part of the RNN into the future by so-called overshooting [19], i.e., we iterate matrices A and C in future direction (fig. 1). In doing so we get a sequence of forecasts as outputs (fig. 1). More important, overshooting forces the learning to focus on modelling the autonomous dynamics of the network, i.e., it supports the extraction of useful information from input vectors, which are more distant to the output. Consequently overshooting is a very simple remedy to the problem that the backpropagation algorithm usually tries to model the relationship between an output and its most recent inputs as the fastest adaptation takes place in the shortest path [5]. Therefore also the learning of false causalities is decreased. Hence, overshooting regularises the learning and thus improves the model's performance [19]. Note, that due to shared weights no additional parameters are used.

3 Normalised Recurrent Neural Networks

As a preparation for the development of normalised recurrent neural networks (NRNN) [18] we first separate the state equation of the basic time-delay RNN (eq. 1) 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 thereby always bounded by the length of the unfolding in time m and the length of the overshooting n [19], such that we have $\tau \in \{t - m, \dots, t + n\}$ for all $t \in \{m, \dots, T - n\}$. 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 optimisation problem:

$$\begin{aligned}
 \tau \leq t : \quad & s_{\tau+1} = \tanh(As_\tau + Bx_\tau + \theta) \\
 \tau > t : \quad & s_{\tau+1} = \tanh(As_\tau + \theta) \\
 y_\tau &= Cs_\tau
 \end{aligned} \tag{4}$$

$$\sum_{t=m}^{T-n} \sum_{\tau=t-m}^{t+n} \|y_\tau - y_\tau^d\|^2 \rightarrow \min_{A,B,C,\theta}$$

In this model, past and future iterations are consistent under the assumption of a constant future environment. Still, the difficulty with this kind of RNN is the training with backpropagation, because a sequence of different connectors has to be balanced, i.e., we do not have the same learning behaviour for the weight matrices in the different time steps. In our experiments we found that this problem becomes especially important for the training of large RNN. Even the training itself is unstable due to the concatenated matrices A , B , and C . As the training changes weights in all of these matrices, different effects or tendencies, even opposing ones, may contradict or superpose. This implies that there may not result a clear learning direction from a certain backpropagated error [18].

NRNN (eq. 5) avoid the stability and learning problems resulting from the concatenation of the three matrices A , B , and C because they incorporate besides the bias θ only one connector type, a single transition matrix $A \in \mathbb{R}^{\bar{J}}$, with $\bar{J} \in \mathbb{N}$. Generally the reduction to one single matrix implies $\bar{J} > J$, i.e., NRNN operate on a larger state space. Besides one can show, that the universal approximation capability also holds for these networks [14].

$$\begin{aligned}
\tau \leq t: \quad s_\tau &= \tanh(As_{\tau-1} + \begin{bmatrix} 0 \\ 0 \\ \mathbf{I}_I \end{bmatrix} x_\tau + \theta) \\
\tau > t: \quad s_\tau &= \tanh(As_{\tau-1} + \theta) \\
y_\tau &= [\mathbf{I}_N \ 0 \ 0] s_\tau
\end{aligned} \tag{5}$$

$$\sum_{t=m}^{T-n} \sum_{\tau=t-m}^{t+n} \|y_\tau - y_\tau^d\|^2 \rightarrow \min_{A, \theta}$$

The corresponding architecture is depicted in figure 2. Again, the dotted connections indicate that the network can be (finitely) further unfolded into past and future.

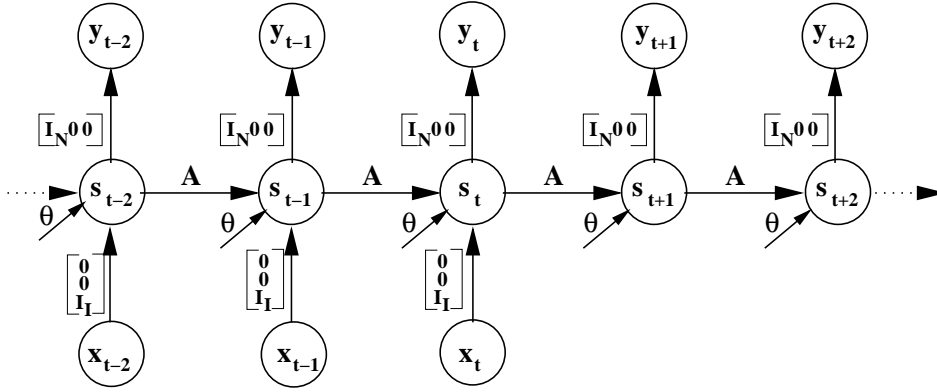


Fig. 2. Normalised recurrent neural network [18].

The matrices between input and hidden as well as hidden and output layer are fixed and therefore not changed during the training process. Consequently matrix A does not only code the autonomous and the externally driven parts of the dynamics, but also the impact of the external inputs x_τ on the internal state space and the computation of the network output y_τ . This implies that all free parameters, as they are combined in one matrix, are now treated the same way by the backpropagation algorithm.

At first view it seems that in the network architecture (fig. 2) the external input $x_\tau \in \mathbb{R}^I$ is directly connected to the corresponding output $y_\tau \in \mathbb{R}^N$. This is not the case, because we enlarge the dimension of the internal state s_τ , such that the input x_τ has no direct influence on the output y_τ . Assuming that we have a number of Q computational hidden neurons the dimension of the internal state would be $\dim(s) = \bar{J} = I + Q + N$. With the non-trained matrix $[\mathbf{I}_N \ 0 \ 0]$ we connect only the first N neurons of the internal state s_τ to the output layer y_τ , where $\mathbf{I}_N \in \mathbb{R}^{N \times N}$

is a fixed identity matrix. Consequently, the NRNN is forced to generate its N outputs at the first N components of the state vector s_τ . The last state neurons are used for the processing of the external inputs x_τ . The fixed connector $[0 \ 0 \ \mathbf{I}_I]^T$, where $\mathbf{I}_I \in \mathbb{R}^{I \times I}$, between the externals x_τ and the internal state s_τ is designed such that the input x_τ is connected to the last I state neurons. To further support the internal processing and to increase the network’s computational power, we add a number of Q hidden neurons between the first N and the last I state neurons. This composition ensures that the input and output processing of the network is separated but implies that NRNN can only be designed as large networks [18]. Note, that out of construction the output of the NRNN is bounded to $(-1, 1)$ by the hyperbolic tangent activation function. Still, this is not a real constraint as we can simply scale the data appropriately before applying it to the network.

Our experiments indicate that, in comparison to RNN, NRNN show a more stable training process, even if the dimension of the internal state is very large.

4 Learning Long-Term Dependencies

We use a very simple but well-known problem to demonstrate the ability of learning long-term dependencies of RNN (sec. 2) and NRNN (sec. 3): the prediction of periodic indicators in a time series. We therefore created time series of 10000 random values, which are uniformly distributed on an interval $[-r, r]$ with $r \in \mathbb{R}$ and $0 \leq r < 1$. Every d -th value, with $d \in \mathbb{N}$ is 1. Per construction these time indicators are the only predictable values for the network. Consequently, for a successful solution to the problem the network has to remember the occurrence of the last 1, d -time steps before in the time series data. In other words, it has to be able to learn long-term dependencies. The higher the d the longer memory is necessary. We used the first 5000 data points for training and left the other half for testing.

Similar problems have already been studied in [5] and [7]. In both papers the performance of the thereby considered recurrent networks trained with backpropagation through time [17] has been tested to be unsatisfactory and the authors concluded that RNN are not suited for the learning of long-term dependencies.

4.1 Model Description

We applied an RNN (sec. 2) and an NRNN (sec. 3) with one input neuron per time step in the past and one output neuron per time step in the future. In contrast to the descriptions in sections 2 and 3 we did not implement any outputs in the past part of the networks, as those would not help to solve the problem. This implies that the gradient information of the error function has to be propagated back from the

future outputs to all past time steps. It also avoids a superposition of the long-term gradient information with a local error flow in the past. Therefore the omission of outputs in the past also eases the analysis of the error backflow.

The networks were both unfolded a hundred time steps into the past. Whereas the NRNN was unfolded twenty time steps into future direction, we did not implement any overshooting for the RNN. In doing so we kept the RNN as simple as possible to show that even such a basic RNN is able to learn long-term dependencies. The total unfolding therefore amounts to 101 time steps for the RNN and to 120 steps for the NRNN. The dimension of the internal state matrix A is always set to 100, which is equivalent to the amount of past unfolding. We initialised the weights randomly with a uniform distribution on $[-0.2, 0.2]$. In all hidden units we implemented the hyperbolic tangent as activation function. We further used the quadratic error function

$$E := \sum_{t=m}^{T-n} \sum_{\tau=t-m}^{t+n} \|y_{\tau} - y_{\tau}^d\|^2 \quad (6)$$

to minimise the difference between network output and target (eqs. 4 and 5). The networks were trained with a shared weight extension of the backpropagation algorithm [13] in combination with pattern-by-pattern learning, a stochastic form of simple gradient descent [11], which gives us the following weight adaption rule:

$$\Delta w = \eta \cdot g_t \quad (7)$$

Here, $\Delta w \in \mathbb{R}$ represents the change of a particular weight w , which is an arbitrary element of the matrices A , B , or C . η denotes the learning rate and $g_t \in \mathbb{R}$ the gradient calculated by the backpropagation algorithm for one pattern t . The learning rate η was set to 10^{-4} , which is a good trade-off between speed and accuracy. Several other learning rules have been developed for RNN with the focus to avoid a vanishing gradient, e.g. vario-eta learning [11,19]. Still, we restricted the learning to this rather simple algorithm to strengthen the significance of our experiments. Therefore we also abstain from a further discussion on the advantages of different learning methods, algorithms and parameter settings. For those please refer to e.g. [11,12]. Besides, the used pattern-by-pattern learning rule has also shown in practice to generate very good results, especially in terms of generalisation, structure identification quality and robustness of the solution. Nevertheless, our results (sec. 4.2) could probably be improved by applying a more problem-dependent learning rule, like partial learning [11]. However, the focus of this paper is the demonstration that RNN unfolded in time are well able to learn long-term dependencies even without any further adjustment.

4.2 Results

We defined an error limit, which marks the optimal achievable error for each problem plus a 10% tolerance. For $r > 0$ it is calculated by the variance of the uniform

distribution given a certain noise range r , assuming no error for the time indicators in every d -th time step and adding 10%. For $r = 0$ it is set to 0.0001, which gives together:

$$\text{error limit} = \begin{cases} 0.0001 & \text{for } r = 0 \\ 1.1 \cdot \frac{d-1}{d} \cdot \frac{r^2}{3} & \text{for } r > 0 \end{cases} \quad (8)$$

Table 1 summarises our results for different time gaps d and several noise ranges r . It shows the mean and standard deviation (STD) of epochs RNN and NRNN needed to pass the error limit (eq. 8) on the test set, i.e., the average number of learning epochs necessary to solve the problem with a maximum of a 10% error tolerance. Hereby, not the actual value but rather the fact that the networks are able to learn the given task within a limited number of epochs is of importance. As already pointed out, the former could most likely be decreased by applying a problem-dependent learning method.

time gap d	range r	# epochs RNN		# epochs NRNN	
		Mean	STD	Mean	STD
40	0.0	25	23	39	14
40	0.1	40	19	35	18
40	0.2	23	4	11	3
40	0.4	113	70	22	22
60	0.0	160	90	158	134
60	0.1	100	98	94	87
60	0.2	382	312	40	28
60	0.4	544	228	207	189
100	0.0	45	8	302	276
100	0.1	58	47	162	70
100	0.2	298	346	79	53
100	0.4	121	206	283	400

Table 1

Average number (Mean) and standard deviation (STD) of learning epochs RNN and NRNN needed to pass the error limit (eq. 8), i.e. to solve the problem, on the test set for different time gaps d and noise ranges r .

The results demonstrate the capability of NRNN as well as of basic RNN to learn long-term dependencies of $d = 40, 60$ and even 100, which is obviously more than the often cited limit of ten time steps [6]. After only a small number of learning epochs both networks were able to solve the problem. Still, in comparison to the

RNN, the NRNN generally showed a more stable learning behaviour (lower STD) and needed in most cases, besides $r = 0$, fewer learning epochs to identify the data structure.

As expected, a larger gap d resulted in more learning epochs for the networks to succeed. Also, apart from $r = 0$, a higher noise range, i.e., a larger uniform distribution of the data, made it more challenging for the networks to identify the time indicators. Still, even in more difficult settings, RNN and NRNN captured the structure of the problem very quickly.

NRNN had most problems identifying the time series with zero noise range, $r = 0$. A zero noise range is actually more difficult than coping with a small noise level, because the networks have to identify a single existing trajectory path instead of a trajectory tube [18]. One reason for the NRNN’s difficulties might be that the applied NRNN had less free parameters than the RNN as the internal state dimension of both is identical but the NRNN has fixed weights between input and state and state and output neurons. In contrary, for larger noise levels the NRNN outperformed the RNN, which supported our theory about NRNN being more stable.

Using smaller dimensions for the single transition matrix A increased the number of epochs necessary to learn the problem (fig. 3). This is due to the fact that the network needs a certain dimension to store long-term information. So e.g., with a hundred dimensional matrix the network can easily store a time gap of $d = 100$ in form of a simple shift register (sec. 5). Downsizing the dimension forces the network to build up more complicated internal matrix structures, which take more learning epochs to develop.

4.3 Analysis of the Backpropagated Error

To put the claim of a vanishing gradient in RNN unfolded in time and trained with backpropagation [6] into perspective we analysed the backpropagated error within our networks. We noticed that under certain conditions vanishing gradients do indeed occur, but are only a problem if we put a static view on the networks like it has been done in [5,6]. Studying the development of the error flow during the learning process we observed that the networks themselves have a regularising effect, i.e., they are able to prolong their information flow and consequently solve the problem of a vanishing gradient. We see two main reasons for this self-regularisation behaviour: shared-weights and overshooting (sec. 2). Whereas shared weights constrain the networks to change weights (concurrently) in every unfolded time step according to several different error flows, overshooting forces the networks to focus on the autonomous sub-dynamics. Especially the former allows the networks to adapt the gradient information flow.

Similar to the analysis in [5] and [6] we further confirmed that the occurrence of

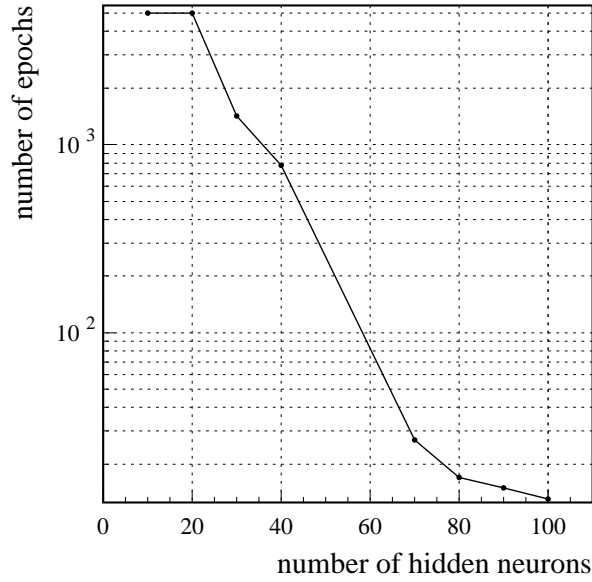


Fig. 3. Number of epochs needed by an NRNN to solve the problem with $d = 40$ and $r = 0.1$ using different numbers of hidden, i.e. internal state, neurons. We stopped training after 5000 epochs, which implies that the network was not able to solve the problem for $\dim(s) \leq 20$.

a vanishing gradient is dependent on the values of the weight matrix A . By initialising matrix A with different weight values it turned out that an initialisation with a uniform distribution in $[-0.2, 0.2]$ is a good choice for the tested networks (sec. 4.4). We never experienced any vanishing gradient in these cases. In contrary, when initialising the networks only within $[-0.1, 0.1]$, the gradient vanished in the beginning of the learning procedure. Nevertheless, during the learning process the networks themselves solved this problem by changing the weight values. Figure 4 shows an exemplary change of the gradient information flow during the learning process.

4.4 Optimal Weight Initialisation

In section 4.3 we already stated that a proper weight initialisation is of importance for the learning of the networks. We noticed that choosing the distribution interval too small can lead to a vanishing gradient in the beginning of the learning. In contrary a too large one can generate very high values in the backflow. In most cases this can be corrected during training (sec. 4.3), but it generally leads to long computational times. For extremely small or large values the network might even be unable to solve the problem.

For that reason we analysed the effects on the backpropagated error flow of different weight initialisations in the RNN and respectively NRNN. This can be done by

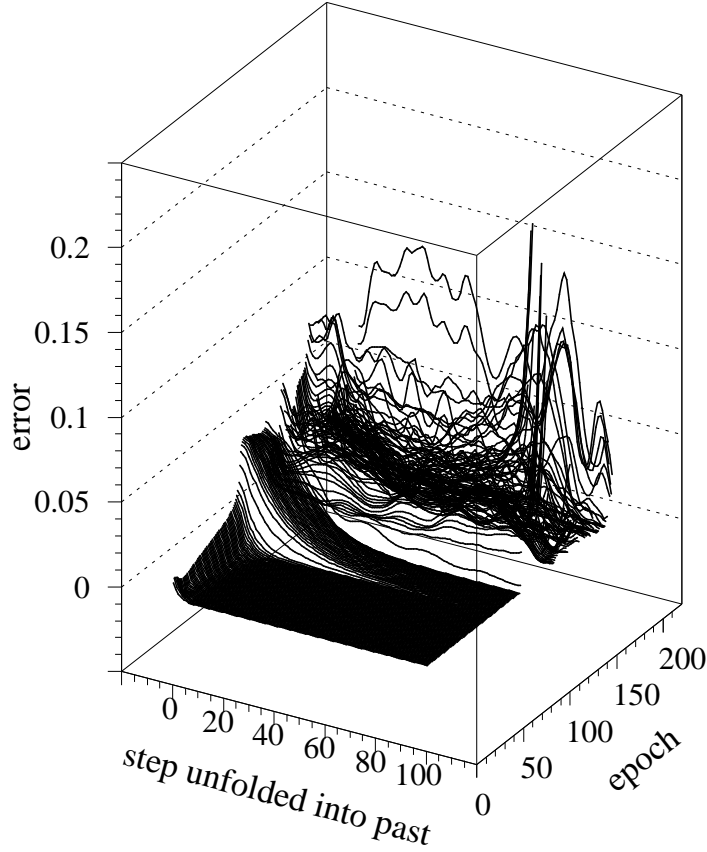


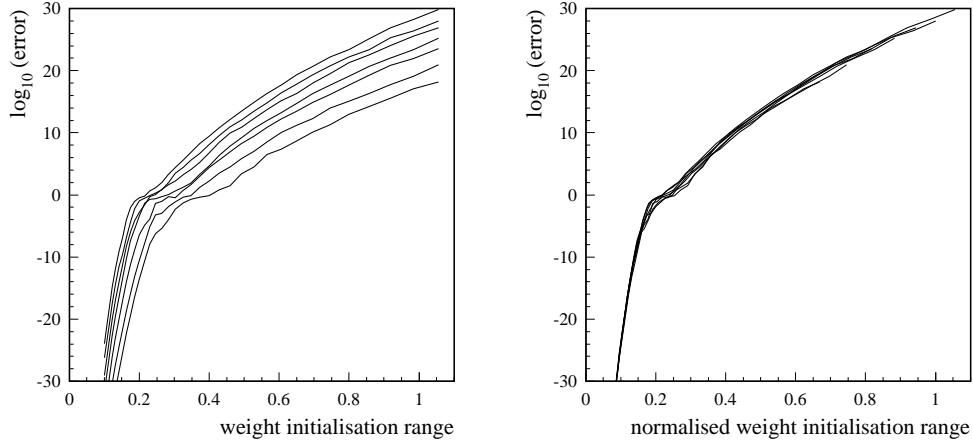
Fig. 4. Exemplary adaptation of the gradient error flow during the learning process of an NRNN, which has been initialised small weights, i.e., within $[-0.1, 0.1]$: The graph shows that for a number of learning epochs smaller than approximately 100, the gradient vanishes very quickly. After that the error information distributes more and more over the different unfolding steps, i.e., the network prolongs its memory span. Finally after about a 150 epochs the error information is almost uniformly backpropagated to the last unfolded time step 100.

measuring the error values backpropagated to the last unfolded time step.

For the experiment we took the same NRNN as in section 4.2 with internal state dimensions of $\dim(s) = 40, 50, 60, 70, 80, 90$ and 100 and calculated the error value backpropagated to the hundredth unfolded time step for different weight initialisations. The problem setting was as before $d = 40$ and $r = 0.1$. Figure 5(a) plots the measured logarithmic error values against different ranges of uniform distributed weight initialisations, each respectively averaged over ten different random initialisations. Note, that for this no learning, i.e., weight adaption, was involved, which is equivalent to $\eta := 0$ (eq. 7). In the plot the more left the curve is the larger is its internal state dimension.

The test confirms that for the 100-dimensional NRNN as used in our long-term ex-

periment (sec. 4.1) an initialisation with weights in $[-0.2, 0.2]$ generates a stable error backflow with neither a vanishing nor an exploding gradient. For those values a reasonable error E ($0.0001 < E < 1$) is propagated back to the last unfolded time step. Interestingly also for initialisations close to $[-0.2, 0.2]$ the error flow stays on a similar level whereas for higher or smaller values it increases and respectively decreases quickly. This saddle point exists for all tested state dimensions, respectively for a slightly shifted interval. It is most likely due to the transfer function used, the hyperbolic tangent, which stays in the linear range for those values.



(a) Backpropagated error information to the last unfolded time step in relation to the range of the uniform weight initialisation. From left to right the corresponding internal state dimension is 100, 90, 80, 70, 60, 50, and 40.

(b) Backpropagated error flow to the last unfolded time step in relation to the range of the uniform weight initialisation for the different regarded state dimensions normalised according to equation (9).

Fig. 5. Influence of the initial weight distribution on the backpropagated error flow in the NRNN using different internal state dimensions.

Figure 5(a) also illustrates that the smaller the internal state dimension is the higher the weights should be initialised. This is caused by the fact that with a higher dimension the probability of larger absolute values of the row sum in the transition matrix A increases, which leads to larger factors backpropagating the error through the unfolded network. Still, as the curves all run in parallel, there is obviously a connection between the optimal weight initialisation and the internal state dimension of the network.

We developed the following useful conjecture

$$\rho_2 = \rho_1 \sqrt{\frac{\dim_1}{\dim_2}} \quad (9)$$

where ρ_i stands for the range of the initialisation interval $[-\rho_i, \rho_i]$ and dim_i for the internal state dimension of two (normalised) RNN ($i = 1, 2$). It is based on our considerations about connectivity within a matrix (sec. 5) and can be easily confirmed by normalising the results of our initialisation test (fig. 5(a)). As expected after normalisation all curves coincide (fig. 5(b)), which shows that, for a given problem, there is a general development of the error flow within (normalised) RNN. Still, according to our experience, independent of the internal state dimension the optimal weight initialisation has to be determined for each problem setting and network architecture individually. However, as described, this can always be done by measuring the backpropagated error in the last unfolded time step.

Our results correspond to the heuristics for weight initialisation in feedforward neural networks [2]. Thereby a reported rule for an optimal weight initialisation is

$$\rho = \frac{3}{\sqrt{\text{dim}}}. \quad (10)$$

For the tested RNN and NRNN this would result in $\rho = 0.3$, which is slightly larger than our empirically determined value, 0.2. The deviation is probably caused by the different network structures. In contrary to feedforward networks for (normalised) RNN the number of unfolded time steps m has to be taken into account. The larger m the greater is the influence of the weight initialisation on the last unfolded time step, because the weights factorise the error flow in each unfolded time step. Note, that for sparse networks (sec. 5) the internal state dimension has to be replaced by the connectivity of the network in equations (9) as well as (10).

We finally compared the sensitivity of the weight initialisation between NRNN and RNN. Thereby both networks had an internal state dimension of 100. The problem setting was once more $d = 40$ and $r = 0.1$. We plot the curves of the RNN and NRNN at the interesting region of the saddle point around 0.2 (fig. 6). In general a flat curve is more preferable as it indicates that the network is less sensitive against the initial weight distribution. Figure 6 shows that the NRNN is slightly more robust against the weight initialisation as its curve is flatter than the one of the RNN. This underlines the theory about NRNN being more stable (sec. 3).

5 Learning Long and Short-Term Memory with RNN

In the following we extend the discussion about learning long-term dependencies with RNN to the learning of long and short-term memory for an optimal system identification. This is mainly done by an analysis of the structure of the autonomous part of the network, which is mapped by the internal state transition matrix A . So far 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} (eqs. 1 and 5). This implies that there is generally a high proportion

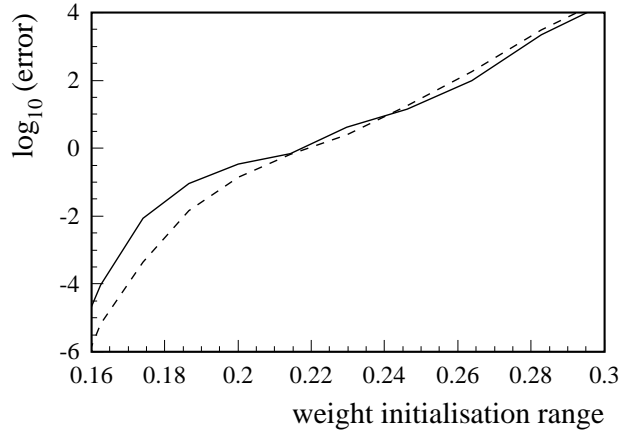


Fig. 6. Sensitivity of the backpropagated error values to different ranges of uniform weight initialisation in the RNN (dashed) and NRNN (solid). The steeper the curve the more sensitive is the network to a sub-optimal weight initialisation.

of superposition (computation) but hardly any simple conservation of information (memory) from one state to the next one (right panel of fig. 7).

For the identification of dynamical systems such memory can be essential, as information may be needed unchanged for computation in subsequent time steps. A shift register (left panel of fig. 7) is a simple example for the implementation of memory, as it only transports information within the state vector s_t . No superposition is performed in this transition matrix. We confirmed this in our basic experiment of section 4 as by setting the transition matrix A to a shift register both networks identified the structure of the simple given time series immediately.

Still, for an optimal system identification several inter-temporal dependencies over different time periods have to be learnt, i.e., the model has to be able to model the system's long-term as well as short-term memory. Therefore 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 one in a subsequent state. We call this number of necessary state transitions the path length of a neuron. In our experiments of section 4 the required path length has been equal to the distance d between two subsequent time indicators 1.

To overcome the apparent dilemma between superposition and conservation of information the transition matrix A needs a structure, which balances different effects of memory and computation. Sparseness of the transition matrix reduces the number of paths and the computational effect of the network but at the same time increases the average path length, and therefore allows for longer-lasting memory. A solution is an inflation of the (normalised) RNN, i.e., a simultaneous extension of

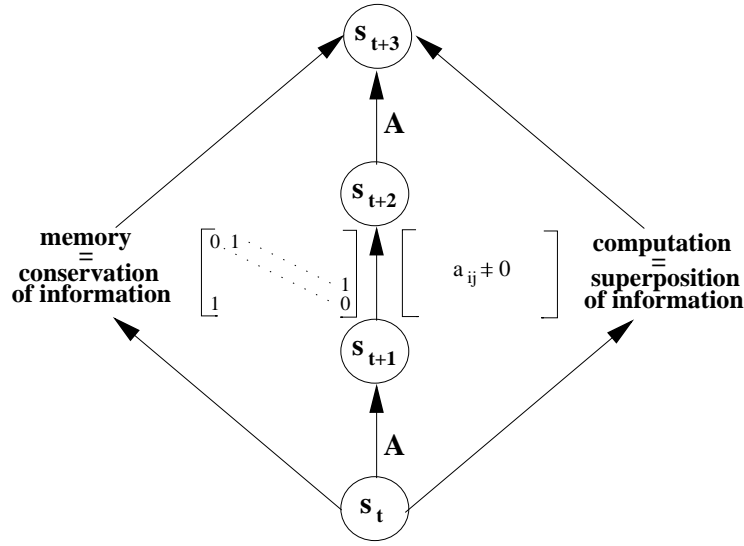


Fig. 7. Learning long and short-term memory with RNN: computation versus memory in the transition matrix A [18].

the internal state dimension and the sparsity level of the transition matrix A while keeping the connectivity constant. In other words, 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 transition matrix A we can optimise both superposition and conservation of information [18]. The sparsity structure is thereby generated randomly, which means a certain percentage of weights is initialised, whereas the remaining ones are set to zero. Proceeding this way, we replicate on average the computational effect of the fully connected network. At the same time we increase the path lengths (memory) with the sparseness level of the inflated transition matrix.

Note, that there is some similarity in the resulting structure to long short-term memory networks [7], where the so called memory cells store information over a longer period of time. In the recurrent neural network this can be achieved by particular sparsity structures of the transition matrix A .

5.1 Performance Gain Through Inflation

To demonstrate that the proposed inflation of the networks helps to learn long-term dependencies we repeat our experiments of section 4 with several different inflated networks. In particular we applied, according to our theory about inflation, RNN and NRNN with internal state dimensions of 200 and 141 and respective sparsity levels of 50% and 70.7%. Besides we tested for comparison reasons both networks with an internal state dimension of 141 neurons and a sparsity level of 50%. The latter does not keep the connectivity constant but in contrary leaves the total number

of weights unchanged. For illustration we once more give the results for the fully connected networks (tab. 1). Table 2 shows that the inflated networks identify the structure of the problem more reliable and generally quicker.

time gap d	range r	# epochs RNN / NRNN			
		dim 100 100%	dim 200 50%	dim 141 70.7%	dim 141 50%
40	0.0	25 / 39	18 / 37	45 / 33	181 / 68
40	0.1	40 / 35	41 / 10	16 / 17	18 / 15
40	0.2	23 / 11	61 / 5	86 / 6	98 / 10
40	0.4	113 / 22	28 / 12	84 / 19	143 / 23
60	0.0	160 / 158	44 / 50	130 / 50	12 / 135
60	0.1	100 / 94	44 / 46	120 / 21	56 / 30
60	0.2	382 / 40	35 / 13	77 / 72	129 / 65
60	0.4	544 / 207	205 / 22	53 / 24	83 / 57
100	0.0	45 / 302	27 / 85	26 / 108	314 / 176
100	0.1	58 / 162	19 / 96	20 / 68	17 / 30
100	0.2	298 / 79	30 / 30	17 / 32	40 / 28
100	0.4	121 / 283	16 / 80	16 / 45	21 / 62

Table 2

Average number of learning epochs the inflated RNN and NRNN with the respectively given internal state dimension and sparsity level needed to pass the error limit (eq. 8), i.e. to solve the problem, on the test set for different time gaps d and noise ranges r . For comparison reasons also the results for the fully connected networks (dim 100 100%) of section 4.2 are given. The results demonstrate the performance improvement through inflation as the maximum number of required learning epochs is lower for those networks.

6 Conclusion

In this paper we showed that RNN unfolded in time and trained with a shared weight extension of the backpropagation algorithm are, in opposition to an often stated opinion, well able to learn long-term dependencies. Using shared weights and overshooting in combination with a reasonable learning algorithm like pattern-by-pattern learning and a proper weight initialisation the problem of a vanishing gradient becomes a minor issue. Our results even indicate that due to shared weights the networks possess an internal regularisation mechanism, which keeps the error flow up and allows for an information transport over at least a hundred time steps. In addition, we described how the weights of RNN and NRNN can be optimally

initialised for a fast and stable learning. For the modelling of long and short-term memory we introduced the idea of inflation, which simplifies the identification of several inter-temporal dependencies within one network. We showed that with the inflated networks we can improve our results in the experiment on long-term dependencies. Overall, we could confirm that both RNN and NRNN are valuable in system identification and forecasting.

Future work may address a more theoretical investigation of the examined self-regularisation ability of RNN, a deeper analysis of the weight initialisation and the optimal transition matrix structure as well as a possible convergence of the weight distribution during the learning process.

Acknowledgement

Our computations were performed on our neural network modelling software SENN (*Simulation Environment for Neural Networks*), which is a product of Siemens AG.

References

- [1] Y. Bengio, P. Simard, P. Frasconi, Learning long-term dependencies with gradient descent is difficult, *IEEE Transactions on Neural Networks* 5 (2) (1994) 157–166.
- [2] K.-L. Du, M. N. S. Swamy, *Neural Networks in a softcomputing framework*, Springer, London, 2006.
- [3] S. Haykin, *Neural Networks: A Comprehensive Foundation*, Macmillan, New York, 1994.
- [4] S. Haykin, J. Principe, T. Sejnowski, J. McWhirter, *New Directions in Statistical Signal Processing: From Systems to Brain*, MIT Press, Cambridge, MA, 2006.
- [5] S. Hochreiter, The vanishing gradient problem during learning recurrent neural nets and problem solutions, *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6 (2) (1998) 107–116.
- [6] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, Gradient flow in recurrent nets: The difficulty of learning long-term dependencies, in: J. F. Kolen, S. Kremer (eds.), *A Field Guide to Dynamical Recurrent Networks*, IEEE Press, 2001, pp. 237–243.
- [7] S. Hochreiter, J. Schmidhuber, Long short-term memory, *Neural Computation* 9 (8) (1997) 1735–1780.
- [8] K. Hornik, M. Stinchcombe, H. White, Multi-layer feedforward networks are universal approximators, *Neural Networks* 2 (1989) 359–366.

- [9] J. F. Kolen, S. C. Kremer, *A Field Guide to Dynamical Recurrent Networks*, IEEE Press, 2001.
- [10] L. R. Medsker, L. C. Jain, *Recurrent neural networks: Design and application*, vol. 1 of *comp. intelligence*, CRC Press international, 1999.
- [11] R. Neuneier, H. G. Zimmermann, How to train neural networks, in: G. B. Orr, K.-R. Mueller (eds.), *Neural Networks: Tricks of the Trade*, Springer Verlag, Berlin, 1998, pp. 373–423.
- [12] B. Pearlmutter, Gradient calculations for dynamic recurrent neural networks: A survey, *IEEE Transactions on Neural Networks* 6 (5) (1995) 1212–1228.
- [13] D. E. Rumelhart, G. E. Hinton, R. J. Williams, Learning internal representations by error propagation, in: D. E. Rumelhart et al. (eds.), *Parallel Distributed Processing: Explorations in The Microstructure of Cognition*, vol. 1, MIT Press, Cambridge, MA, 1986, pp. 318–362.
- [14] A. M. Schaefer, H. G. Zimmermann, Recurrent neural networks are universal approximators, *International Journal of Neural Systems* 17 (4) (2007) 253–263.
- [15] A. Soofi, L. Cao, *Modeling and Forecasting Financial Data, Techniques of Nonlinear Dynamics*, Kluwer Academic Publishers, 2002.
- [16] P. J. Werbos, *The Roots of Backpropagation. From Ordered Derivatives to Neural Networks and Political Forecasting*, John Wiley & Sons, New York, 1994.
- [17] R. J. Williams, D. Zipser, Gradient-based learning algorithms for recurrent connectionist networks, in: Y. Chauvin, D. E. Rumelhart (eds.), *Backpropagation: Theory, Architectures, and Applications*, Erlbaum, Hillsdale, NJ, 1990.
- [18] H. G. Zimmermann, R. Grothmann, A. M. Schaefer, C. Tietz, Identification and forecasting of large dynamical systems by dynamical consistent neural networks, in: S. Haykin, J. Principe, T. Sejnowski, J. McWhirter (eds.), *New Directions in Statistical Signal Processing: From Systems to Brain*, MIT Press, 2006, pp. 203–242.
- [19] H. G. Zimmermann, R. Neuneier, Neural network architectures for the modeling of dynamical systems, in: J. F. Kolen, S. Kremer (eds.), *A Field Guide to Dynamical Recurrent Networks*, IEEE Press, 2001, pp. 311–350.