# Learning Long Term Dependencies with Recurrent Neural Networks

Anton Maximilian Schäfer[1,2], Steffen Udluft[1], and Hans Georg Zimmermann[1]

[1] Information & Communications, Learning Systems
Siemens AG, Corporate Technology, 81739 Munich, Germany
{Schaefer.Anton.ext, Steffen.Udluft,
Hans_Georg.Zimmermann}@siemens.com
[2] Department Optimisation and Operations Research, University of Ulm, 89069 Ulm, Germany

**Abstract.** Recurrent neural networks (RNNs) 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 through time (BPTT) it is claimed that RNNs unfolded in time fail to learn inter-temporal influences more than ten time steps apart.

This paper provides a disproof of this often cited statement. We show that RNNs and especially normalised recurrent neural networks (NRNNs) unfolded in time are indeed very capable of learning time lags of at least a hundred time steps. We further demonstrate that the problem of a vanishing gradient does not apply to these networks.

## 1 Introduction

Recurrent neural networks (RNNs) allow the identification of dynamical systems in form of high dimensional, nonlinear state space models [1,2]. They offer an explicit modeling of time and memory and allow in principle to model any type of open dynamical system [3]. The basic concept is more than 20 years old, so e.g., unfolding in time of neural networks and related modifications of the backpropagation algorithm can already be found in [4].

Nevertheless, there is often a negative attitude towards RNNs because it has been claimed by several authors that RNNs unfolded in time are unable to identify and learn long-term dependencies of more than ten time steps [5,6,7]. To overcome the stated dilemma new forms of recurrent neural networks, e.g., long short-term memory (LSTM) networks [8], were developed, but these networks do not offer the desirable correspondence between equations and architectures as RNNs unfolded in time do.

Still, the analyses in the mentioned papers [5,6,7] were all based on a very basic architecture of RNNs and, even more important, made from a static perspective. In this paper we therefore disprove the statement that RNNs unfolded in time and trained with backpropagation through time (BPTT) are in general unable to learn long-term dependencies. We outline that RNNs and especially normalised recurrent neural networks (NRNNs) unfolded in time have no difficulty with an identification and learning of past-time information within the data which is more than ten time steps apart. Furthermore we show that by using shared weights training of these networks is not a major problem.

S. Kollias et al. (Eds.): ICANN 2006, Part I, LNCS 4131, pp. 71–80, 2006.
© Springer-Verlag Berlin Heidelberg 2006

It even helps to overcome the problem of a vanishing gradient as the networks possess a self-regularisation characteristic which adapts the error information flow.

We start with a recapitulation of the basic RNN unfolded in time (sec. 2). Here we especially emphasise the advantage of overshooting and point out that this simple extension regularises the learning with BPTT. 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 (sec. 3). In section 4 we then demonstrate that both NRNN and RNN successfully learn long-term dependencies. In doing an analysis of the backpropagated error flow we finally show that the problem of a vanishing gradient is not a relevant question for both networks. In section 5 we give a conclusion and an outlook on further research.

## 2 Recurrent Neural Networks Unfolded in Time

The basic time-delay recurrent neural network (RNN) consists of a state transition and an output equation [1,9]:

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

Here, the state transition equation $s_{t+1}$ ($t = 1, \ldots, T$ where $T$ is the number of available patterns) is a nonlinear combination of the previous state $s_t$ and external influences $u_t$ using weight matrices $A$ and $B$ of appropriate dimension and a bias $c$, which handles offsets in the input variables $u_t$. The network output $y_t$ is computed from the present state $s_t$ employing matrix $C$. It is therefore a nonlinear composition applying the transformations $A$, $B$, and $C$.

Training the RNN of equation 1 is equivalent to solving a parameter optimisation problem, i.e., minimising the error between the network output $y_t$ and the real data $y_t^d$ with respect to an arbitrary error measure, e.g.:

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

It can be solved by finite unfolding in time using shared weight matrices $A$, $B$, and $C$ [1,4]. 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 $t \in \{1, \ldots, T\}$ [1,4]. This guarantees that we have the same dynamics in every time step. By using unfolding in time the RNN can be trained with error backpropagation through time (BPTT) [1,4], which is a shared weights extension of the standard backpropagation algorithm [10]. Figure 1 depicts the resulting spatial neural network architecture [9].

We extend the autonomous part of the RNN into the future by so-called overshooting [9], i.e., we iterate matrices A and C in future direction (see fig. 1). In doing so we get a sequence of forecasts as an output. More important, overshooting forces the learning to focus on modeling 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.

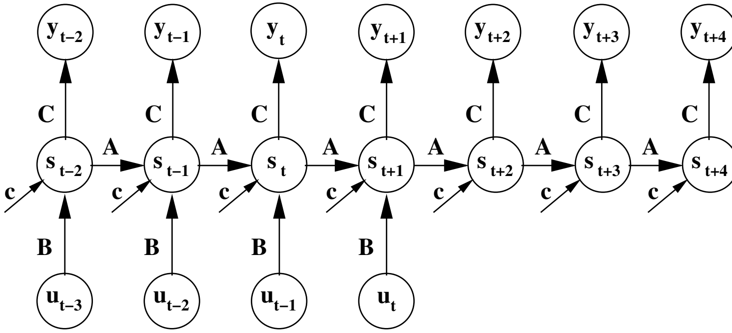


**Fig. 1.** RNN unfolded in time using overshooting

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 because 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 [9]. 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 (NRNNs) [11] 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 $s_\tau$ with $\tau \leq t$ belong to the past part and those with $\tau > t$ to the future part. The parameter $\tau$ is thereby always bounded by the length of the unfolding in time $m$ and the length of the overshooting $n$ [9], such that we have $\tau \in \{t-m, \ldots, t+n\}$ for all $t \in \{m, \ldots, T-n\}$. 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:

$$\begin{aligned} \tau \leq t: \quad & s_{\tau+1} = \tanh(As_\tau + c + Bu_\tau) \\ \tau > t: \quad & s_{\tau+1} = \tanh(As_\tau + c) \\ & y_\tau \quad = Cs_\tau \\ \sum_{t=m}^{T-n} \sum_{\tau=t-m}^{t+n} & (y_\tau - y_\tau^d)^2 \rightarrow \min_{A,B,C,c} \end{aligned} \tag{3}$$

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 BPTT, 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 the different time steps. In our experiments we found, that this problem becomes more 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, can influence them and may superpose. This implies, that there results no clear learning direction or change of weights from a certain backpropagated error [11].

NRNNs (eq. 4) avoid the stability and learning problems resulting from the concatenation of the three matrices $A$, $B$, and $C$ because they incorporate besides the bias $c$ only one connector type, a single transition matrix $A$:

$$\begin{aligned} \tau \leq t: \quad & s_\tau = \tanh\left(As_{\tau-1} + c + \begin{bmatrix} 0 \\ 0 \\ \mathrm{Id} \end{bmatrix} u_\tau\right) \\ \tau > t: \quad & s_\tau = \tanh(As_{\tau-1} + c) \\ & y_\tau = [\mathrm{Id}\ 0\ 0] s_\tau \\ & \sum_{t=m}^{T-n} \sum_{\tau=t-m}^{t+n} (y_\tau - y_\tau^d)^2 \rightarrow \min_{A,c} \end{aligned} \tag{4}$$

The corresponding architecture is depicted in figure 2.

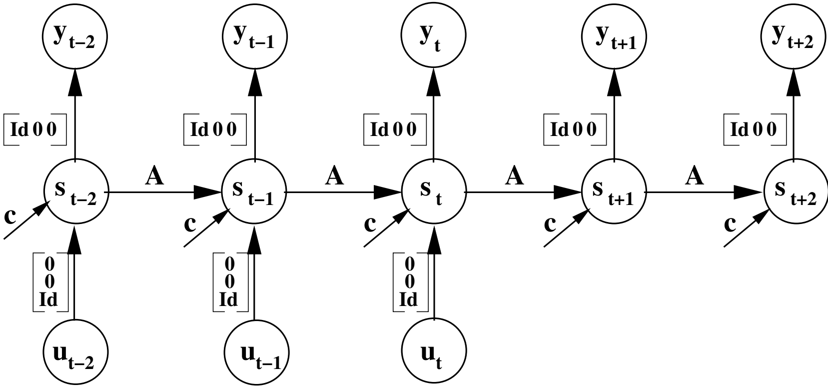


**Fig. 2.** Normalised recurrent neural network

Using NRNN modeling is solely focused on the transition matrix $A$. 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 (pre-)processing of the external inputs $u_\tau$ and the computation of the network outputs $y_\tau$. This implies that all free parameters, as they are combined in one matrix, are now treated the same way by BPTT.

At first view it seems, that in the network architecture (fig. 2) the external input $u_\tau$ is directly connected to the corresponding output $y_\tau$. This is not the case, because we enlarge the dimension of the internal state $s_\tau$, such that the input $u_\tau$ has no direct influence on the output $y_\tau$. Assuming that we have a number of $p$ outputs, $q$ computational hidden neurons and $r$ external inputs, the dimension of the internal state would be $\dim(s) = p + q + r$. With the matrix $[\text{Id } 0\ 0]$ we connect only the first $p$ neurons of the internal state $s_\tau$ to the output layer $y_\tau$. As this connector is not trained, it can be seen as a fixed identity matrix of appropriate size. Consequently, the NRNN is forced to generate its $p$ outputs at the first $p$ components of the state vector $s_\tau$. The last state neurons are used for the processing of the external inputs $u_\tau$. The connector $[0\ 0 \text{ Id}]^T$ between the externals $u_\tau$ and the internal state $s_\tau$ is an appropriately sized fixed identity matrix. More precisely, the connector is designed such that the input $u_\tau$ is connected to the last $r$ state neurons. To additionally support the internal processing and to increase the network's computational power, we add a number of $q$ hidden neurons between the first $p$ and the last $r$ state neurons. This composition ensures, that the input and output processing of the network is separated but implies that NRNNs can only be designed as large neural networks [11].

Our experiments indicate that NRNNs show, in comparison to RNNs, 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 RNNs and NRNNs. Similar problems have already been studied in [5] and [8]. In both papers the performance of RNNs trained with BPTT has been tested to be unsatisfactory and the authors concluded that RNNs are not suited for the learning of long-term dependencies.

We created time series of $10000$ values which are uniformly distributed on an interval $[-r, r]$ with $r \in \mathbb{R}$ and $0 < r < 1$. Every $d$-th value, with $d \in \mathbb{N}$ is $1$. These 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 afore in the time series data. In other words, it has to be able to learn long-term dependencies. The higher $d$ the longer memory is necessary. We used the first $5000$ data points for training and left the other half for generalisation.

### 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

$$E := \sum_{t=m}^{T-n} \sum_{\tau=t-m}^{t+n} (y_\tau - y_\tau^d)^2 \tag{5}$$

to minimise the difference between network output and target (eqs. 4 and 3). The networks were trained with BPTT in combination with pattern-by-pattern learning [12]. The learning rate $\eta$ was set to $10^{-4}$.

### 4.2 Results

Table 1 summarises our results for different time gaps $d$ and several noise ranges $r$. The error limit shows the optimal achievable error for the given problem plus a $10\%$ tolerance. It is calculated by the variance of the uniform distribution given a certain noise range $r$ and assuming no error for the time indicators in every $d$-th time step. We give the average number of epochs RNN and NRNN needed to pass this error limit, i.e., the number of learning epochs necessary to solve the problem with a maximum of a $10\%$ error tolerance.

**Table 1.** Results for different time gaps $d$ and noise ranges $r$

| time gap $d$ | range $r$ | Error limit | # Epochs RNN | # Epochs NRNN |
|---|---|---|---|---|
| 40 | 0.1 | 0.003575 | 19 | 13 |
| 40 | 0.2 | 0.0143 | 19 | 9 |
| 40 | 0.4 | 0.0572 | 50 | 28 |
| 60 | 0.1 | 0.00361 | 39 | 33 |
| 60 | 0.2 | 0.01442 | 437 | 23 |
| 60 | 0.4 | 0.05769 | 389 | 248 |
| 100 | 0.1 | 0.00363 | 65 | 106 |
| 100 | 0.2 | 0.01452 | 353 | 59 |
| 100 | 0.4 | 0.05808 | 96 | 84 |

The results demonstrate the ability of NRNNs as well as of basic RNNs 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 [7]. After only a small number of learning epochs both networks were able to solve the problem. Still, in comparison to the RNN, the NRNN

in general showed a more stable learning behaviour and needed in most cases slightly shorter to identify the data structure.

As expected, a longer gap $d$ resulted in more learning epochs, the networks needed to succeed. Also a higher noise range $r$, 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.

Using smaller dimensions for the single transition matrix $A$ increased the number of epochs necessary to learn the problem (fig. 3). This is probably 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. Downsizing the dimension forces the network to build up more complicated internal matrix structures which take more learning epochs to develop.

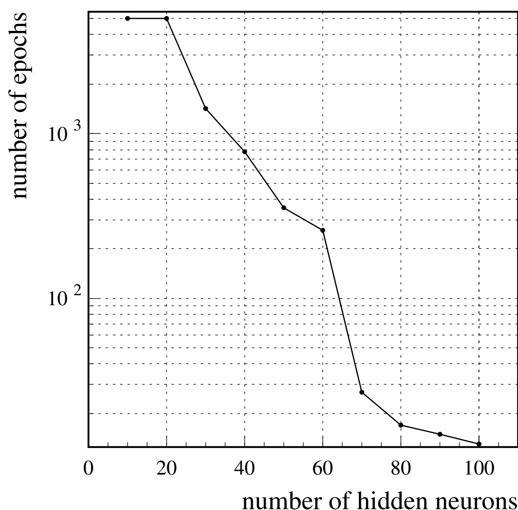


**Fig. 3.** Number of epochs needed by an NRNN to solve the problem with $d = 40$ and $r = 0.1$ using different numbers of hidden, i.e. internal state, neurons. We stopped training after 5000 epochs which implies that the network was not able to solve the problem for $dim(s) \leq 20$.

### 4.3 Analysis of the Backpropagated Error

To put the claim of a vanishing gradient in RNNs unfolded in time and trained with BPTT [7] 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 [5,7]. 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, 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 [7] we further confirmed that the occurrence of 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 our networks (sec. 4.1). 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.

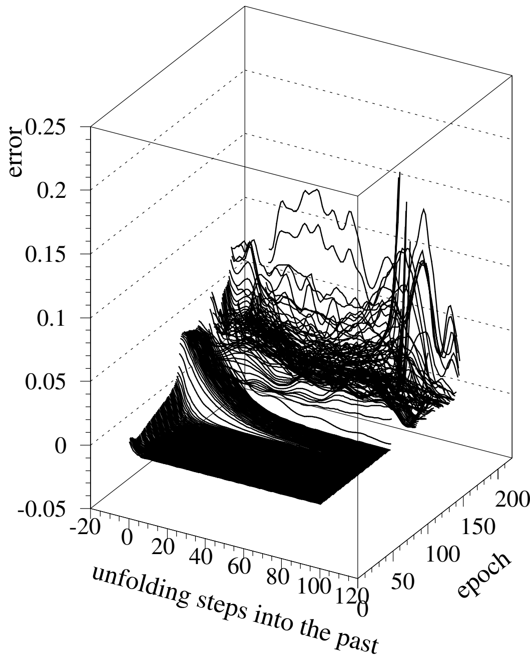


**Fig. 4.** Exemplary adaptation of the gradient error flow during the learning process of an NRNN which has been initialised with small weights: The graph shows that for a number of learning epochs smaller than approximately 100, the gradient vanishes very quickly. After that the error information distributes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

## 5 Conclusion and Outlook

In this paper we demonstrated that NRNNs as well as basic RNNs unfolded in time and trained with BPTT 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 the problem of a vanishing gradient becomes irrelevant. Our results even show 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. Consequently RNNs and especially NRNNs are valuable in time series analysis and forecasting.

Looking at the results and our general experience with recurrent neural networks we further assume that there is a conjunction between the internal state dimension and the weight values in form of an optimal expected row sum of the transition matrix $A$. The confirmation of this assumption will be part of our future research. Besides that we want to investigate in how far a theoretical analysis of the examined self-regularisation ability of recurrent neural networks is possible.

## Acknowledgment

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

## References

1. Haykin, S.: Neural Networks: A Comprehensive Foundation. Macmillan, New York (1994)
2. Kolen, J.F., Kremer, S.: A Field Guide to Dynamical Recurrent Networks. IEEE Press (2001)
3. Schaefer, A.M., Zimmermann, H.G.: Recurrent neural networks are universal approximators. In: Proceedings of the International Conference on Artificial Neural Networks (ICANN-06), Athens (2006)
4. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning internal representations by error propagation. In Rumelhart, D.E., et al., J.L.M., eds.: Parallel Distributed Processing: Explorations in The Microstructure of Cognition. Volume 1. MIT Press, Cambridge (1986) 318–362
5. Hochreiter, S.: 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. Bengio, Y., Simard, P., Frasconi, P.: Learning long-term dependencies with gradient descent is difficult. IEEE Transactions on Neural Networks **5**(2) (1994) 157–166
7. Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J.: Gradient flow in recurrent nets: The difficulty of learning long-term dependencies. In Kolen, J.F., Kremer, S., eds.: A Field Guide to Dynamical Recurrent Networks. IEEE Press (2001) 237–243
8. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Computation **9**(8) (1997) 1735–1780
9. Zimmermann, H.G., Neuneier, R.: Neural network architectures for the modeling of dynamical systems. In Kolen, J.F., Kremer, S., eds.: A Field Guide to Dynamical Recurrent Networks. IEEE Press (2001) 311–350

10. Werbos, P.J.: Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. PhD thesis, Harvard University (1974)
11. Zimmermann, H.G., Grothmann, R., Schaefer, A.M., Tietz, C.: Dynamical consistent recurrent neural networks. In Prokhorov, D., ed.: Proceedings of the International Joint Conference on Neural Networks (IJCNN), Montreal, MIT Press (2005)
12. Neuneier, R., Zimmermann, H.G.: How to train neural networks. In Orr, G.B., Mueller, K.R., eds.: Neural Networks: Tricks of the Trade. Springer Verlag, Berlin (1998) 373–423