IUB International University Bremen

International University Bremen • Jacobs University Bremen • 2007

Mantas Lukoševičius

Echo State Networks with Trained Feedbacks

Technical Report No. 4
February 2007

School of Engineering and Science

Echo State Networks with Trained Feedbacks

Mantas Lukoševičius

Jacobs University Bremen
School of Engineering and Science
28759 Bremen
Germany

E-Mail: m.lukosevicius@iu-bremen.de
http://www.iu-bremen.de/

Summary

Echo State Networks (ESNs) is an approach to the recurrent neural network (RNN) training, based on generating a big random network (reservoir) of sparsely interconnected neurons and learning only a single layer of output weights from the reservoir as the target function. Despite many advantages of ESNs over gradient based RNN training techniques, they lack the power of learning some complex functions. New findings in dynamical systems theory state, that fixed neural circuits can obtain universal computational qualities if suitable feedbacks (or intermediate units) can be trained. Unfortunately the theory gives no hint on how this can be done. In this report we explore possible directions in which the theoretical findings could be applied to increase the computational power of ESNs. More specifically, we discuss possible options for defining training targets for the feedbacks, present and discuss some empirical results (positive as well as negative) testing the ideas in practice and analyze some problems pointing out to some intrinsic limitations of ESNs. Another contribution of this report is a discussion of many practical issues of training ESNs in particular the ones having feedback connections. We also propose a modification of ESNs called Layered ESNs. This technical report is based on the author's Master Thesis named "Improving Echo State Networks by Training Intermediate Units".

Contents

1 Introduction

Artificial neural networks is perhaps the most prominent black-box modeling technique in Machine Learning. Since feedforward neural networks (FFNNs) [Bishop, 1995] have been shown to have the ability to approximate with desired accuracy any given smooth function [Irie and Miyake, 1988], discrete-time recurrent neural networks (RNNs) can approximate with desired accuracy any given dynamics with continuous transition function on compact sets and in a finite time horizon [Hammer and Steil, 2002]. Since most of the real world systems are dynamical, RNNs have a large application potential. All biological neural networks (i.e. brains) are also recurrent. Despite the large potential, RNNs penetrate real world applications quite slowly. Because of their complexity, RNNs are difficult to train and analyze, thus less popular or even accessible among many engineers and developers. Training of the networks is also computationally expensive which makes using only a small number of neurons practical. Echo State Networks (ESNs) [Jaeger and Haas, 2004] is a new simple and powerful approach to RNN training, which is currently becoming more and more popular. Even though ESNs are shown to perform very well in many tasks, there are some complex problems where they perform poorly.

A recent development in the theory of dynamic systems has showed, that trained feedbacks can endow fixed neural circuits with universal computational capabilities [Maass et al., 2006]. This theory has direct implications to ESNs since they largely constitute of a fixed neural circuits (*reservoirs*). In this report we explore different ideas of how the power of ESNs could be improved along the lines of this new theory, namely by teaching ESNs intermediate feedbacks and training their other weights, while keeping the *reservoir* fixed. If we could devise a good technique for doing this, we would be able to improve the power of ESNs without sacrificing much of its simplicity and computational efficiency.

The purpose of this report is not to break any performance records in the accuracy of learning some standard tasks, but rather explore different directions of modifying conventional ESNs in the light of the new theory and testing in practice which of them would lead to improving the power of ESNs and which would not.

We will introduce the basic concepts of echo state networks in Section 2. In Section 3.1 we will briefly present the new development in the theory of dynamical systems and relate it to ESNs in Section 3.2. The theory presented in Section 3 is the main motivating conjecture for the directions of extending ESNs that are explored in this report. Unfortunately the theory is not constructive: it tells that fixed neural circuits can have universal computational capabilities if some external fed-back functions are “right”, but it does not tell how to construct them. We continue with a discussion of what could be the possible ways of defining the training targets for these functions for some large classes of problems in Section 4. In Section 5 we will discuss some specifics of actually doing the training for the targets. In Section 6 we will introduce the datasets that we used in our numerical simulations, ranging from such on which ESNs are famous for their good performance (in Section 6.1), to such where ESNs perform very poorly (in Section 6.4). In Section 7 we present some empirical results testing the so far discussed ideas on the datasets presented in Section 7, demonstrating the improvement which can be achieved in some cases. We also make an analysis of (a bad) performance of ESNs on a particular dataset in Section 7.3, pointing out some fundamental incompatibilities of the problem with the ESN approach. Along the lines of this analysis, we propose a possible modification of ESNs in Section 8, which though not really helps to solve this particular problem, proves itself to be an improvement in solving other problems. Finally in Section 9 we summarize our findings and the directions still left unexplored.

2 Echo State Networks

2.1 Definition of ESNs

Echo state networks (ESNs) [Jaeger and Haas, 2004] is a recent approach to recurrent neural network supervised training, which overcomes some obstacles encountered in many other approaches to training RNNs, mentioned above. In the ESN approach a large (order of several tens to several thousand neurons), randomly connected RNN is used as a “reservoir” of dynamics which can be excited by suitably presented input and/or fed-back output. The connection weights of this reservoir network are not changed by training. In order to compute a desired output dynamics, only the weights of connections from the reservoir to the output units are calculated. In the conventional case of ESNs this is a simple linear combination of the reservoir activations (i.e. a single layer of weights). Because there are no cyclic dependencies between the trained readout connections, training an ESN becomes a simple linear regression task, for which numerous batch or adaptive on-line algorithms are available [Jaeger and Haas, 2004]. In a more general case the readout can be done via a multi-layered feedforward neural network, often referred to by the name of *Multi-Layer Perceptron* (MLP) [Bishop, 1995]. This case is explained in more detail in Section 2.2. The main idea of ESNs has been independently investigated in a more biologically oriented setting under the name of “liquid state networks” [Maass et al., 2002].

The difference between ESN and gradient based RNN training is illustrated in Figure 1. The bold gray connections denote the weights that are being trained. In gradient based training they are updated iteratively, whereas in ESN the exact values of the output weights W_{out} are calculated in a single iteration.

The echo state networks are usually discrete-time networks of sigmoid units with the following state update equation:

$$x(n) = f(W_{\text{in}}u(n) + Wx(n-1) + W_{\text{ofb}}y(n-1)), \quad n = 1, \cdots, T, \tag{1}$$

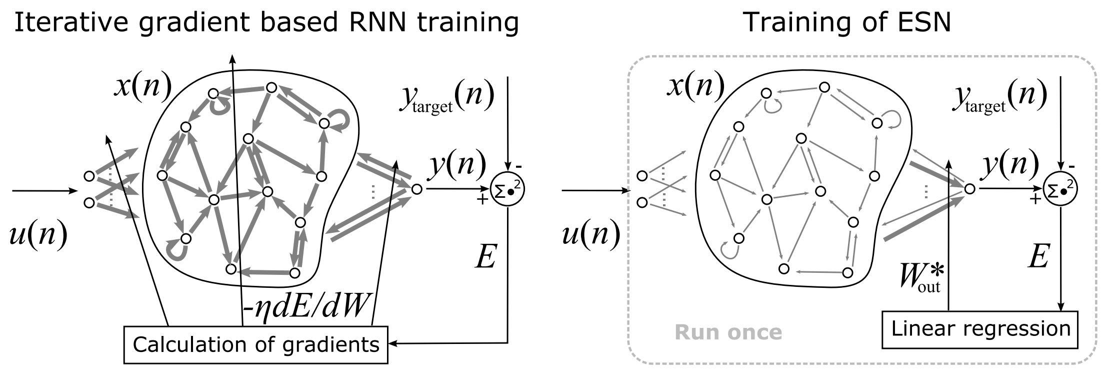

Figure 1: The difference between gradient based and ESN training of RNN.

where $x(n) \in \mathbb{R}^N$ is a vector of reservoir neuron activations at a time step n, $f(\cdot)$ is the neuron activation function (usually the $\tanh(\cdot)$ sigmoid) applied element-wise, $W_{\text{in}} \in \mathbb{R}^{N \times N_u}$ is the input weight matrix, $u(n) \in \mathbb{R}^{N_u}$ is the input vector, $W \in \mathbb{R}^{N \times N}$ is a randomly generated sparse weight matrix of internal reservoir connections, $W_{\text{ofb}} \in \mathbb{R}^{N \times N_y}$ is an optional output feedback weight matrix and $y(n) \in \mathbb{R}^{N_y}$ is the output of the network. Weight matrices W_{in} and W_{ofb} are usually dense with each element randomly chosen from some interval (depending on the scaling). The network is usually started with the initial state $x(0) = 0$ and $y(0) = 0$. Each unit in the reservoir has also a bias value, which is omitted in the equations for simplicity. The bias can be easily implemented, adding an additional input, which has a constant value of 1 (and a corresponding randomly generated column in W_{in}).

In the traditional linear readout case, the output of the system $y(n)$ is defined by the equation

$$y(n) = f_{\text{out}}(W_{\text{out}}[u(n)|x(n)]), \quad n = 1, \cdots, T, \tag{2}$$

where W_{out} is the (learned) output weight matrix, and $f_{\text{out}}(\cdot)$ is the output neuron activation function (usually $\tanh(\cdot)$ sigmoid or the identity) applied component-wise, and $\cdot|\cdot$ stands for a vertical concatenation of vectors (or matrices). The standard batch supervised training of ESN proceeds by driving them with the training input sequence $u(n)$ once, collecting the internal states over the whole training period, and then computing the output weights W_{out} as the linear regression weights of the teacher output $y_{\text{target}}(n)$ on the internal states. Different standard methods can be applied here. They are further discussed in Section 5.2.

The *echo state property* is essential for making the ESN learning method work. Let us define it following [Jaeger and Haas, 2004] closely. Intuitively, a RNN which is driven by an external signal $u(n)$ has the echo state property if the activations $x(n)$ of the RNN neurons are systematic variations of the driver signal $u(n)$. More formally, this means that for each internal unit x_i there exists an "echo function" e_i, such that, if the network has been run for an indefinitely long time in the past, the current state can be written as $x_i(n) = e_i(u(n), u(n-1), u(n-2), \cdots)$. For discrete-time ESNs there are several nontrivial alternative definitions of this condition and algebraic characterizations of which network weight matrices W lead to networks having the echo state property [Jaeger, 2001]. For practical purposes it suffices to fix the spectral radius $\rho(W)$ of W to a value less than unity to ensure the echo state property. It is worth mentioning, that even though $\rho(W) > 1$ cause self-induced internal dynamics in the reservoir, such network can still be trained for some tasks, as they become stable with the presence of input [Ozturk and Principe, 2005].

It is also important that the dynamics of the reservoir neurons should be richly varied. This is ensured by a sparse interconnectivity (of 1-20%) within the reservoir. The condition lets the network decompose into many loosely coupled subsystems, establishing a richly structured reservoir of excitable dynamics [Jaeger and Haas, 2004]. There has been some research done to find alternative models for generating the random reservoirs (e.g. [Liebald, 2004]), but so far there has been no other topology discovered, which would perform significantly better than the sparse randomly connected graph mentioned before.

2.2 ESNs with MLP Readouts

In the case of a multi-layer readout, $y(n)$ is calculated as the output of a MLP taking $[x(n)|u(n)]$ as its input, thus generalizing (2) into

$$y(n) = f_{\text{out}}(W_{\text{out}m} f(\cdots W_{\text{out}2} f(W_{\text{out}1}[u(n)|x(n)]) \cdots)), \tag{3}$$

where m is the number of layers in the readout MLP and $W_{\text{out}1}, W_{\text{out}2}, \cdots, W_{\text{out}m}$ are connection weight matrices for the inputs of the corresponding layers each having respectively $N_{\text{out}1}, N_{\text{out}2}, \cdots, (N_{\text{out}n} = N_y)$ units. Each weight matrix $W_{\text{out}i} \in \mathbb{R}^{N_{\text{out}i-1} \times N_{\text{out}i}}$, for $i = 1, \cdots, m$ and $N_{\text{out}0} := N_u + N$. Alternatively, $y(n+1)$ can be calculated as a concatenation of outputs from several distinct MLPs having the same input. The biases for the units in the MLP are also omitted here as in (1). They can be similarly implemented by adding a pseudo-unit in each layer with a constant activation of 1.

When training the MLP(s), the activation states of the reservoir are collected once as in the case with a single readout layer, but then the MLP(s) need to be trained iteratively, using e.g. error back-propagation and a stochastic gradient descent, as discussed more throughout in Section 5.3. This is illustrated by Figure 2 as a comparison to a recurrent gradient-based training of RNNs and a classical training of ESNs with linear readouts depicted in Figure 1.

All the same rules apply to the reservoir of the ESN in the MLP readout case, as in the linear readout case. Note that in contrast to the reservoir, the information propagates through all the m layers of the readout MLP in a single time step. If disregarding the training method for finding W_{out}, the linear readout (2) can be seen as a special case of (3) with $m = 1$. When talking about ESNs we will have ESNs with linear readouts (2) in mind, if no MLPs are mentioned explicitly.

Training of ESN with a MLP readout

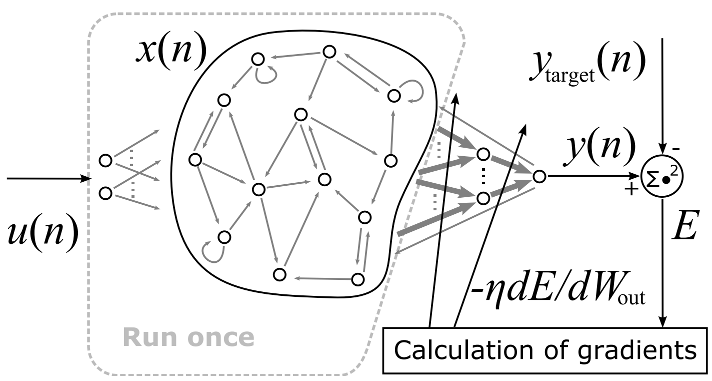

Figure 2: Training of ESN with MLP readouts.

3 Trained Feedbacks for Fixed Neural Circuits

3.1 Recent Developments in the Theory of Dynamical Systems

There is a new theoretical result, showing that trained feedbacks can endow fixed neural circuits and other dynamical systems with universal capabilities for analog computing [Maass et al., 2006]. In particular, feedback supports computations where on-line input streams are processed in diverse ways according to the internal state of the dynamical system. The authors of [Maass et al., 2006] have stated and proved that bounded noise (e.g. due to the fact that feedback functions are not learned perfectly) stays bounded, i.e. does not get amplified through feedback. Their theorem claims that even though such systems can not simulate a Turing machine, they can still simulate any finite state machine, which is equivalent to a Turing machine with a finite memory (the best we can realistically do in any case).

More precisely, the authors of [Maass et al., 2006] state, that a given fixed dynamical system [1] of the form

$$\dot{x}(t) = f(x(t)) + g(x(t)) \cdot v(t), \tag{4}$$

where $f : \mathbb{R}^N \to \mathbb{R}^N$, $f = (f_1, \cdots, f_N)$ and $g : \mathbb{R}^N \to \mathbb{R}^N$, $g = (g_1, \cdots, g_N)$ are fixed component-wise functions, $x = (x_1, \cdots, x_N)$ is the vector of the system state (e.g. activations of neurons) and $v(t)$ is input, can approximate any N'th order differential equation of the form

$$p^{(N)}(t) = G(p(t), \dot{p}(t), \ddot{p}(t), \cdots, p^{(N-1)}(t)) + u(t) \tag{5}$$

(for arbitrary smooth functions $G : \mathbb{R}^N \to \mathbb{R}$), because there exists a (memory-free) feedback function $K : \mathbb{R}^N \times \mathbb{R} \to \mathbb{R}$ and a memory-free readout function

[1]Belonging to a certain class S_n, see [Maass et al., 2006] for details.

$h : \mathbb{R}^N \to \mathbb{R}$ (which can both be chosen to be smooth, in particular continuous) such that, for every external input $u(t), t \geq 0$, and each solution $p(t)$ of the forced system (5) there is an input $u_0(t)$ with $u_0(t) \equiv 0$ for all $t \geq 1$, so that the solution

$$\dot{x}(t) = f(x(t)) + g(x(t))K(x(t), u(t) + u_0(t)), \quad x(0) = 0 \tag{6}$$

satisfies

$$h(x(t)) = p(t) \quad \text{for all } t \geq 1. \tag{7}$$

3.2 Application of the Theory to Echo State Networks

If we assume, that the classical ESN state update equation (1) is equivalent to a discrete time case of the dynamical system (4) (i.e. fixed weights W and activation function f in (1) correspond to the fixed functions f and g in (4)), and further assume that the equivalent of the feedback function K can be well approximated by a linear or a MLP readout from the reservoir of the ESN, then the classical ESNs can obtain the above mentioned universal computational capabilities, if we properly train auxiliary outputs that are fed back into the reservoir. For such ESN with auxiliary feedbacks we can modify equation (1) into

$$x(n) = f(W_{\text{in}}u(n) + Wx(n-1) + W_{\text{afb}}z(n-1) + W_{\text{ofb}}y(n-1)), \tag{8}$$

where W_{afb} are auxiliary feedback weights produced similarly to W_{ofb} and $z(n) \in \mathbb{R}^{N_z}$ is the vector of auxiliary outputs, defined similarly to (3) by

$$z(n) = f_{\text{aux}}(W_{\text{aux}m}f(\cdots W_{\text{aux}2}f(W_{\text{aux}1}[u(n)|x(n)])\cdots)), \tag{9}$$

where $W_{\text{aux}i}$, $i = 1, \cdots, m$ is the (learned) auxiliary output readout MLP and $f_{\text{aux}}(\cdot)$ is the output neuron activation function (usually $\tanh(\cdot)$ sigmoid or the identity) applied component-wise. More precisely, we assume here that training W_{aux} and including $z(n)$ in (8) can have the same effect as including the function K in (7). We can see that auxiliary outputs $z(n)$ in (9) technically do not differ from additional dimensions of the final output $y(n)$ in (3) and for simplicity in some equations will be included as part of $y(n)$. The difference here is more semantical. One point is that the feedback connections W_{ofb} from the final output might be not used, while the feedback connections W_{afb} from $z(n)$ are essential. Another difference is that from the formulation of the problem at hand we know what we would like to have as the output $y(n)$, but not $z(n)$ (i.e. we have a target signal $y_{\text{target}}(n)$), but not $z_{\text{target}}(n)$).

It is easy to show that if we use a simple linear readout for $z(n)$ and the dimension N_z of $z(n)$ is equal to the number of units in the reservoir ($N_z = N$ – a rather extreme case), we can transform the fixed reservoir into any desirable recurrent neural network without changing its connections W. More specifically, by a simple linear readout we mean a special case of (9):

$$z(n) = W_{\text{aux}}[u(n)|x(n)] = W_{\text{aux}u}u(n) + W_{\text{aux}x}x(n) = W_{\text{aux}x}x(n), \tag{10}$$

where the part $W_{\text{aux}u}$ of the matrix W_{aux} connecting the input $u(n)$ directly to $z(n)$ is set to 0. Then substituting (10) into (8) we get

$$\begin{aligned} x(n+1) &= f(W_{\text{in}}u(n+1) + Wx(n) + W_{\text{afb}}W_{\text{aux}x}x(n) + W_{\text{ofb}}y(n)) \\ &= f(W_{\text{in}}u(n+1) + (W + W_{\text{afb}}W_{\text{aux}x})x(n) + W_{\text{ofb}}y(n)) \\ &= f(W_{\text{in}}u(n+1) + W^*x(n) + W_{\text{ofb}}y(n)), \end{aligned} \tag{11}$$

where $W^* := (W + W_{\text{afb}}W_{\text{aux}x})$. We can obtain any W^* by setting

$$W_{\text{aux}x} = W_{\text{afb}}{}^{+}(W^* - W). \tag{12}$$

Note that (11) is equivalent to the update equation of the classical ESN (1), but instead of the fixed W in (1), we can have any desirable W^* in (11), thus any recurrent neural network as the reservoir. The Equation (12) is solvable, because W_{afb} is a randomly generated square matrix, thus generally invertible. Having $N_z < N$ restricts achievable W^*s, by restricting $\text{rank}(W_{\text{afb}}W_{\text{aux}x}) = \text{rank}(z(n)) \leq N_z$, but the reservoir dynamics can still be influenced significantly.

The rest of the trained feedback weights $W_{\text{aux}u}$ has a similar effect on W_{in} as $W_{\text{aux}x}$ on W. It would have exactly the same effect if we would modify (8), by replacing $u(n)$ with $u(n-1)$, which could be argued to be just a matter of convention.

4 Options for Feedback Targets

Unfortunately the theorem in [Maass et al., 2006] is not constructive, i.e. it does not mention what the function K and consequently the auxiliary outputs $z(n)$ should be or how to obtain them. We need some *trainer* mechanism for the ESN to produce the targets $z_{\text{target}}(n)$ which the ESN should learn as $z(n)$ (Figure 3). Intuitively speaking, they should push the dynamics of the reservoir into the "right direction" towards the activations $x(n)$ which could be linearly combined into the target signal $y_{\text{target}}(n)$. The auxiliary targets should be like hints or partial solutions – something which the network must find out in order to solve the given problem. For example, if the task of the network is to map a robot sensor input into control signals for collision-free movement, a plausible intermediate problem would be to identify the obstacles using the sensor data before moving to avoid them. Thus we could use obstacle recognition as a subproblem on which we could train the auxiliary outputs and feed them back to the reservoir. The metaphor of the trainer is quite deep here: like in sports, the trainer should make one work on the necessary elements of the final task to improve the results.

Since learning of the final output $y_{\text{target}}(n)$ depends on the fed-back results $z(n)$ of learning $z_{\text{target}}(n)$, common sense suggests that the auxiliary feedbacks $z(n)$ should be trained before training the final output $y(n)$. This means that ESN should be trained in several epochs. Separate dimensions of $z(n)$ can also be trained in separate epochs if they have similar one-directional dependences as

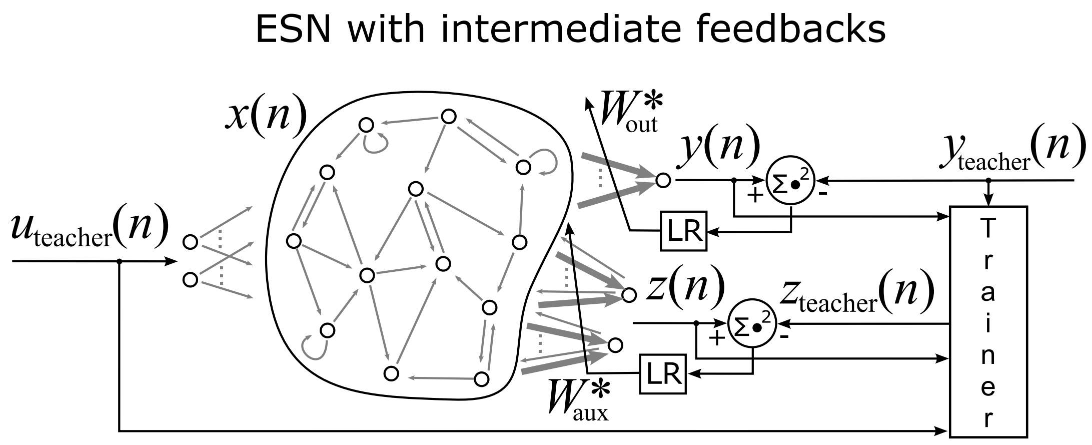

Figure 3: ESN with auxiliary feedbacks $z(n)$ and its trainer.

between $z(n)$ and $y(n)$. Note, that the trainer in Figure 3 does not necessarily need to produce $z_{\text{target}}(n)$ from the $u(n)$, $y_{\text{target}}(n)$, $z(n)$ and/or $y(n)$ but can use the signals from different than n or even several time steps, including steps ahead in time (at least in off-line mode of training).

Defining $y_{\text{target}}(n)$ is in essence biasing the ESN as a modeling system into particular kind of models. There exists a quite general "no free lunch" principle in supervised machine learning [Wolpert, 2001] (as well as in optimization), which in essence claims that there can be no bias of this kind which would universally improve the accuracy of the model for all possible problems it is solving. In other words we can not define *universally* good $z_{\text{target}}(n)$. We need to have some kind of insight into the problem at hand to do this successfully. In this section we will discuss some potential options for defining $z_{\text{target}}(n)$ for some large classes of problems.

4.1 Context Identifying Feedbacks

In many real world applications parameters of the function that we are trying to learn change in time in a random (out of scope, not realistically predictable) way. Such changes are often referred to as changes of the *context* or changes of the *generator mode* (in the case of time series). They could alternatively be considered as changes of some hidden, not directly observable input parameters of the system we are trying to model. A classical example of such a situation is changes of the physical environment (source of echoes and noises) in which wireless communication is performed. The standard technique to deal with this is an adaptive online training of the (usually feedforward) neural network. For this training we need to somehow know the correct output signal (the online teacher), which e.g. is available with some delay (in time series prediction) or at certain points in time (in mobile communications).

In many real world tasks we do not know or are not able to encode the context even for the training data, let alone for the working environment of the model. However, one credible assumption which we can in many cases make is that context changes on a slower time scale than the signal itself (otherwise the system would not be predictable at all). Thus one possible way to generate auxiliary targets could be using *slow feature analysis* (SFA) [Wiskott and Sejnowski, 2002] in the *trainer* (Figure 3). SFA is a way of extracting the most slowly varying statistically uncorrelated features from the input signal. The speed of varying of a feature a_i is defined in terms of its average squared derivative $\left\langle \dot{a_i}^2 \right\rangle$. The extracted features are normalized so that $\langle a_i \rangle = 0$ and $\left\langle (a_i - \langle a_i \rangle)^2 \right\rangle = \langle a_i^2 \rangle = 1$, thus avoiding trivial solutions where $a = \text{const}$. By statistical uncorrelatedness we mean $\forall j < i : \langle a_j a_i \rangle = 0$.

In SFA the features a_i are constrained to be linear combinations of a finite set of non-linear functions of the input signal. For this purpose the normalized input signal is expanded using the non-linear functions and then after one more normalization the features are extracted using principal component analysis. It is thus quite possible, that the idea of SFA can be exploited in an ESN directly, using the reservoir as the nonlinear expansion $x(n)$ of the input $u(n)$ and calculating W_{aux} to directly obtain $z(n)$ as the slow features, instead of calculating them externally in the trainer and using as $z_{\text{target}}(n)$ to find W_{aux} minimizing mean square error. One apparent difference is that in SFA the non-linear expansion is memoryless. We have not explored the direction of combining ESNs with SFA, but this could be an attractive direction for future research especially when working with datasets where slow features are intrinsically relevant.

4.2 Predictive Feedbacks

If we are not able to teach an ESN the final target $y_{\text{target}}(n)$ (i.e. with sufficient accuracy) directly, an intuitive approach would be to choose the intermediate targets $z_{\text{target}}(n)$ such, that they can be directly learned by ESN with higher accuracy and then in their turn make learning of the final target $y_{\text{target}}(n)$ easier (i.e. improve accuracy). Following this approach intermediate targets are defined in such a way, that each of them is a necessary (or helpful) but not sufficient "component" of the final output. However, having intermediate targets less "ambitious", i.e. easier to learn but only contributing partially to the final target y_{target}, could be not the only possible option. Alternatively, we might also try having them equally or even more difficult to learn than the final target. Even though they would not be learned properly (otherwise the final target as easily could be learned directly too), they could still move the internal dynamics of the reservoir states so that learning of the final target becomes easier. In this context perhaps calling the targets $z_{\text{target}}(n)$ *auxiliary* would be more intuitive then calling them intermediate, since they are not reached before the final target is reached, but on the other hand they are still intermediate in the sequence of trainings.

A concrete example of the latter approach in a time series prediction task could be trying to predict the time series two time steps ahead in the auxiliary target, i.e. $z_{\text{target}}(n) = y_{\text{target}}(n+1)$. While it is generally not easier than learning to predict one time step ahead (the final target), the auxiliary target provides maximal information needed for the final output: the output $y(n)$ at time step n makes use of the auxiliary feedback $z(n-1)$ generated at time step $n-1$. The same way we can define k auxiliary feedbacks as parts of $z_{\text{target}}(n)$, where the k'th feedback at time step n is trained on $y_{\text{target}}(n+k)$. This way the i'th feedback helps to learn $(i-1)$'th feedback. Instead of predicting the output itself we can predict some of its features. Since prediction of the next time step usually depends on the several previous time steps, it could also make sense to train auxiliary outputs $z(n)$ to "post-predict" the values, i.e. refine the previous prediction result $y(n-k)$ if the correct value $y(n-k)$ is not (yet) available to the network. This should help to improve the prediction accuracy.

Even if the task is not time series prediction, in some quite general cases it could be beneficial (and possible) to predict the input $u(n+k)$ in the auxiliary outputs $z(n)$, and in some tasks it could be beneficial to preserve the old input $u(n-k)$ in $z(n)$ (note, that learning $u(n-1)$ precisely is trivially done, by setting $W_{\text{aux}u}$ to identity matrix and $W_{\text{aux}x}$ to zero). In general, predicting and/or improving previous input and/or output as auxiliary feedbacks could teach the ESN to represent the hidden process better for quite large classes of problems. In other words, modeling the temporal context of the problem at hand might provide some additional valuable information (dynamics) for the ESN on how to solve the problem.

4.3 Error Predicting Feedbacks

Another information-centric option could be auxiliary targets trying to predict the error of the subsequent final output:

$$z_{\text{target}}(n) = y_{\text{target}}(n+1) - y(n+1). \tag{13}$$

Such auxiliary feedbacks would aim at creating exactly those dynamics in the reservoir, that can not be learned using the final readout alone. This approach could be implemented by the following algorithm:

1. Train $y(n)$, having no feedback $z(n)$ to the reservoir;
2. Define $z_{\text{target}}(n)$ as in (13);
3. Train $z(n)$ on $z_{\text{target}}(n)$;
4. Retrain $y(n)$, having a trained feedback $z(n)$ to the reservoir.

The algorithm can be recursively extended to several stages of error prediction, by embedding it again inside the Step 3, i.e. by recursively predicting the error of $z_i(n)$, $i = 1, \cdots, N_y$ in other dimensions of the auxiliary feedback $z_j(n)$, $j > N_y$.

Learning to predict the error (13) might be very difficult. For example, learning $z_{\text{target}}(n) = y_{\text{target}}(n) - y(n)$ is not possible by definition, if $y(n)$ is optimally trained and we use the same linear readout mechanism for $z(n)$ as for $y(n)$ (and not putting much hope into the effect of the teacher-forced feedback $z_{\text{target}}(n)$ when learning $z(n)$, because of the reasons discussed in Section 5.4). Trying to predict this difference one time step ahead most probably would not make things easier either.

Instead of learning to approximate the difference by minimizing the mean squared error we could maximize the correlation of $z(n)$ with $z_{\text{target}}(n)$ from (13) in a way similar to the Cascade Correlation Learning [Fahlman and Lebiere, 1990], [Campbell, 1997]. Such a performance measure is not well suited for finding a single vector of linear readouts by regression, but works well with feed-forward neural network readouts. So instead of minimizing the sum-of-square error

$$E_z = \sum_{n=1}^{T} (z(n) - z_{\text{target}}(n))^2, \tag{14}$$

we could maximize the correlation

$$C_z = \sum_{n=1}^{T} (z(n) - \langle z(n) \rangle)(z_{\text{target}}(n) - \langle z_{\text{target}}(n) \rangle), \tag{15}$$

where $z_{\text{target}}(n)$ in both (14) and (15) is from (13).

4.4 On Rational Separation Between ESN and its Trainer

Another thing worth mentioning is that from an applications point of view it makes little sense to produce auxiliary targets using only inputs $z_{\text{target}}(n) = \phi(u(n))$, because if this could be done, we might use the target producing mechanism ϕ as a preprocessor $u'(n) = (u(n)|\phi[u(n)])$ of the original inputs and include the targets as additional inputs for ESN, instead of teaching ESN to mimic the auxiliary target producer. Note that this would be equivalent to teacher forcing the auxiliary feedbacks all the time, which would improve precision (this, however, does not apply to the input prediction, which is discussed in more detail in a moment). Using auxiliary targets produced solely from the inputs would only make sense if we could benefit in terms of computational cost, better generalization (trading off some precision), or from some other side-products of this approximation inside the reservoir.

Similarly, using a bijective function ψ of the output (also holds if including input) as auxiliary teachers $z_{\text{target}}(n) = \psi(y_{\text{target}}(n + k))$ means that the function ψ is reversible into function ψ^{-1} which can be used as a postprocessor. In such case if the auxiliary outputs $z_{\text{target}}(n)$ are well learned, they could become the final outputs of ESN and the postprocessor $y(n + k) := \psi^{-1}(z(n))$ would do the rest of the job. Thus, if we know ψ^{-1} precisely we would save the ESN from learning it (and again gain in terms of precision). On the other hand, if the auxiliary outputs

$z_{\text{target}}(n)$ are not learned well, then the ESN might in fact produce a better final output by making additional use of the components $x(n)$ from which the $z(n)$ was constructed, than applying the exact ψ^{-1}.

As mentioned above, these considerations come more from the applications point of view, where achieving best performance is the main concern. In such a setup the question of which tasks should be dedicated to ESN and which could be done better by the trainer or mechanisms derived from it should be answered. However, if investigating the potential of ESNs and/or being inclined to biologically plausible systems, leaving as much of the job to the ESN as possible could also be a valuable approach.

5 Specifics of Training ESN Readouts

5.1 A Typical Setup for Training ESN Readouts

As mentioned in Section 2.1, ESNs are trained by collecting input $u(n)$ and activation states $x(n)$, then calculating W_{out} using a linear regression for the linear readouts, or an iterative training technique, described in Sections 5.2, respectively 5.3. First the ESN is run for the *initial period* (typically a 1000 time steps in our simulations), by providing the input and teacher-forcing signals (if applicable) to "wash out" the transiences that occur because of the initialization of the activations to $x(0) = 0$. Then, continuing running the ESN in the same manner for the *training period*, inputs $u(n)$ and activation states $x(n)$ over the whole period are collected into one single big matrix. Then the output weights W_{out} are computed using this matrix and the target $y_{\text{target}}(n)$, $n = 1, \cdots, T$, in a way described in more details bellow, which concludes the training. For simplicity of the notation, we will denote the training period as $n = 1, \cdots, T$, i.e. start counting time steps n from 1, ignoring the initial period.

The trained ESN is then usually rerun with no teacher-forcing for the initial period, training period and a succeeding *testing* (or *validation*) *period*. Outputs $y(n)$ collected over the training and testing periods are used to estimate training, respectively testing performance of the ESN. For this in our experiments we use the normalized root-mean-square (NRMS) error [2] of the form

$$E = \sqrt{\frac{\left\langle \|y(n) - y_{\text{target}}(n)\|^2 \right\rangle}{\left\langle \|y_{\text{target}}(n) - \langle y_{\text{target}}(n)\rangle\|^2 \right\rangle}}, \tag{16}$$

where $\|\cdot\|$ stands for the Euclidean distance (or norm), to evaluate the performance. The error has two obvious properties, namely $E = 0$ if, and only if $y(n) = y_{\text{target}}(n)$, and $E = 1$ for an optimal constant solution $y(n) = \text{const} = \langle y_{\text{target}}(n)\rangle$. These two properties also hold for the error measure E^2, which many authors tend

[2]Similar to RMS error defined in [Bishop, 1995] p.197, where it is, contrary to what the name suggests, not square-rooted for some reason.

to use. In the interval $(0, 1)$, where these errors typically reside, measure E^2 gives a smaller number (twice the negative order for the small errors near the value 0).

5.2 Training the Linear Readouts

The standard way of training ESNs is to find the linear output weights W_{out} by solving a *linear regression*, or a *linear least squares* problem, which in essence is a mathematical optimization technique for finding an approximate solution for a system of linear equations that has no exact solution.

More specifically, let $U \in \mathbb{R}^{N_u \times T}$ denote a matrix collecting all the inputs $u(n) \in \mathbb{R}^{N_u}$ over the training period, $X \in \mathbb{R}^{N \times T}$ – the matrix of states $x(n) \in \mathbb{R}^{N}$, and $Y \in \mathbb{R}^{N_y \times T}$ – the matrix of outputs $y(n) \in \mathbb{R}^{N_y}$. Let us assume here for simplicity, that $z(n)$ is part of $y(n)$. Using this notation we can rewrite the ESN readout equation (2) for the entire training period $n = 1, \cdots, T$ in a single equation as

$$Y = f_{\text{out}}(W_{\text{out}}[U|X]). \tag{17}$$

When training ESN, the output weights $W_{\text{out}} \in \mathbb{R}^{u_y \times (N_u+N)}$ are computed directly from (17) as

$$W_{\text{out}} = {f_{\text{out}}}^{-1}(Y_{\text{target}})(U|X)^{+}, \tag{18}$$

where ${f_{\text{out}}}^{-1}(\cdot)$ is the inverse function of output activation and $(U|X)^{+}$ stands for a pseudoinverse of the matrix $(U|X) \in \mathbb{R}^{(N_u+N)\times T}$ (or an alternative method of solving the matrix equation as discussed below).

Computationally refined pseudoinverse-based linear regression. As a standard method for solving a matrix equation $A = BC$, where B is unknown, we use the matrix pseudoinverse $B = AC^{+}$. Strangely enough, the precision of this method in Matlab can be in many cases improved, by adjusting the calculated B into $B' = B + (A - BC)C^{+}$. While in theory $B = B'$ should hold, in practice, for high-precision tasks like predicting chaotic time series where the NRMS error is in the order of 10^{-8}, this trick can decrease the training error more than twice. This fix virtually does not increase the computational cost of the operation, since it works well with reusing the same (i.e. only once calculated) pseudoinverse C^{+}. I came up with this fix, when found out by accident that the remainder (or error of learning B) $A - BC$ can still be partially learned using the same regression, i.e. $(A - BC)C^{+} \neq 0$. My guess for explanation of this would be some optimizations inside the Matlab's operations on (big) matrices, that are not quite lossless. Further iterative applications of this fix to B did not yield any further tangible improvement.

The matrix pseudoinverse is a very computationally stable, but a bit expensive method for the ESN training, since we need to calculate a pseudoinverse of a large matrix $(U|X)$ (which corresponds to C). As another alternative, in our simulations we also use the standard Matlab matrix division operator $B = A/C$, which in effect uses a QR (orthogonal-triangular) factorization via Householder reflections. This method is usually much faster, and in some cases gives a smaller error, but is also computationally less stable.

As an even faster (and even less computationally stable) alternative we also make use of Wiener-Hopf equations and calculate $B = (AC^{\mathrm{T}})(CC^{\mathrm{T}})^{-1}$. Since the $(U|X)$ matrix has typically much more columns than rows $N_u + N \ll T$, we get a much smaller autocovariance matrix $\big([U|X][U|X]^{\mathrm{T}}\big) \in \mathbb{R}^{(N_u+N)\times(N_u+N)}$ whose inverse we need to calculate. In most cases (namely, when the condition number of CC^{T} has a reasonably small size) this method gives similar results to the ones calculated by the QR factorization.

When training only a subset $S \subset \{1, \cdots, N_y\}$ of the output (dimensions) $y_S(n)$ at a time, only the corresponding rows of Y_{target} are used and thus only the corresponding rows $W_{\mathrm{out}S} = {f_{\mathrm{out}}}^{-1}(Y_{\mathrm{target}S})[U|X]^{+}$ of the output weights are calculated.

5.3 Training of the MLP Readouts

As discussed in Section 2.2, ESNs can have multi-layered readouts in the form of MLPs. Learning of the MLP weights typically can not be done in a single iteration, thus after running the ESN once and collecting $(U|X)$ we use it as an input to train the readout MLP iteratively. This scheme is depicted in Figure 2.

While MLPs have a much bigger potential of expressiveness than a simple linear combination, they are much harder and computationally expensive to train. It has been shown, that for tasks where classical ESNs with linear outputs perform well, achieving a similar level of performance using MLPs is very hard (if realistic at all) [Jain, 2004]. Thus MLPs are practical to use only where linear readouts fail.

We use MLPs with two or three layers of units m (3) and train it using error back propagation with a stochastic gradient descent [Bishop, 1995]. The MLPs are trained with slowly decreasing learning rate and having a constant momentum value. The training data is randomly shuffled once before training and presented to the training algorithm in cycles as a long (order of 50000) sequence of data points.

Stochastic gradient descent proved itself to be a much faster method than the batch version (which is also claimed in the literature, e.g. [Bishop, 1995]). Since the number T of training data points is typically large and the data is typically redundant, using the batch version of gradient descent for ESN readout training is not practical. A compromise method with data grouped into smaller batches was also tried. Doing a rough manual optimization of all the other parameters for several batch sizes, the compromise method showed that the smaller is the batch size, the faster the learning rate decreases, which in effect led back to the stochastic version of the algorithm (batch size equal to one).

We have also done experiments with an online adaptation of the learning rate for the batch training, following the the "bold driver" approach [Vogl et al., 1988] together with a momentum term. In this approach during the training one increases the learning rate slightly each time a weight update results in decrease of error, and undoes the update and decreases the learning rate sharply in the opposite case. The method guarantees non-increase of the error and finds its way through difficult areas of the error surface, but is very slow if comparing to stochastic gradient descent, as mentioned above. The "bold driver" approach only works well in a pure batch mode of gradient descent, as having smaller batches it gets completely confused by the stochastic nature of the learning and the learning rate keeps rapidly shrinking (or bloating, depending on the parameters).

For two-layered MLPs we used a weight initialization proposed in [Nguyen and Widrow, 1990]. The weights were initialized with random values uniformly distributed over $[-0.5, 0.5]$ and the ones of the first layer $W_{\text{out}1}$ scaled so that

$$\left\|W_{\text{out}1j}\right\| = 0.7 \cdot N_{\text{out}1}^{\frac{1}{N_u+N}}, \quad j = 1, \cdots, N_{\text{out}1}, \tag{19}$$

where $W_{\text{out}1j}$ is the jth row of the input weight matrix $W_{\text{out}1} \in \mathbb{R}^{N_{\text{out}1} \times (N_u+N)}$ of the first (hidden) layer of the readout MLP (3). Typically $N_{\text{out}1} \ll N_u + N$ and thus $\left\|W_{\text{out}1j}\right\| \approx 0.7$. Here the notion of $W_{\text{out}1j}$ excludes the bias value of the unit which we will denote as $W_{\text{out}1j,0}$ and set separately to a uniformly distributed random value from the interval $\left[-\left\|W_{\text{out}1j}\right\|, \left\|W_{\text{out}1j}\right\|\right]$ (19).

For a three-layered MLP, only the input layer is initialized by (19).

We have also experimented with combining gradient based training of MLPs with a linear regression on their output weights $W_{\text{out}m}$ in the spirit of ESNs. Indeed, we can consider the rest of the MLP $W_{\text{out}i}$, $i = 1, \cdots, m-1$ as part of the reservoir (see Figure 2) and find $W_{\text{out}m}$ by solving the linear the regression as in the classical training of ESNs. This trick when applied increases accuracy significantly (for some problems in the order of 5 times), but can not be easily mixed with gradient descent methods. After regressing $W_{\text{out}m}$, further gradient based training becomes difficult. Unless the learning rate is set to an extremely small value, the learning error quickly gets even much bigger than before the regression. This holds even if the gradient based training is not allowed to change the output weights $W_{\text{out}m}$. In practice we have only succeeded to apply the batch "bold driver" to further decrease the error after the regression, which creeps then with extreme slowness.

As a practical solution, we settled on using the stochastic gradient descent with a regression on $W_{\text{out}m}$ after the training, but this issue deserves a separate investigation, which is out of the scope of this work.

5.4 Teacher Forcing of Feedbacks

So far we have only discussed the options for intermediate targets. In this section we will discuss how the training using them could be implemented. A difficulty of training ESNs with feedback connections (common to all RNNs) is, that once the trained outputs are fed back into the reservoir, they change the dynamics of all the internal units $x(n)$ and thus the outputs too. In conventional training of ESN with feedback connections from the output, we use a so called *teacher forcing* technique for breaking this recurrence. Teacher forcing means that target values are fed back to the reservoir, as if they were already successfully learned. This enables us to learn outputs in one iteration and is a valid assumption, if in the end the outputs are learned well (i.e. the feedbacks are similar to the one which we assumed while training). The results can also be improved by producing feedbacks that are somewhat in between the exact teacher signal and the signal which would be produced by the ESN, as explained in the supporting online material of [Jaeger and Haas, 2004].

If the feedbacks are not learned well, this assumption is not valid and the distorted feedbacks may further distort the outputs. This is not a big concern for classical ESNs, since if we are not able to learn the output then we have already failed. For auxiliary outputs of ESNs (8)(9) this is not necessarily the case. Especially if the targets are constructed in the ways described in Sections 4.2 and 4.3, while learning the final output $y(n+1)$ we can not assume that $z(n)$ was learned with enough precision. Thus it would make more sense first to train $z(n)$ for the auxiliary target $z_{\text{target}}(n)$ and for training $y(n+1)$ feed the actual value of $z(n)$ back to the reservoir instead of unrealistic $z_{\text{target}}(n)$. This, as already discussed, training of ESNs with auxiliary outputs should be done in stages, where a subset of auxiliary or final outputs is trained at a time. There is still an open question, however, what signals should be fed back from the outputs that have not been trained. Possible answers to this question could be:

- Optimistically use the teacher signal;
- Use pure noise, making the learning process avoid influence of the feedback;
- Use a mixture of teacher and noise, which could imitate an imperfectly learned target (the optimal proportion of the two would also be an open question);
- Zero feedback signals.

Which option is best to use, depends on each concrete case.

6 Datasets Used for Numerical Simulations

Let us at this point briefly introduce the time series which we used to run simulations and empirically test the ideas discussed in this report. The four time series described here are roughly in the order of increasing difficulty to learn for the classical ESNs.

6.1 Lorenz Attractor

We used a Lorenz chaotic attractor [Lorenz, 1963] as a representative of a chaotic time series prediction task in our simulations. ESNs are famous for their record-breaking performance on this type of tasks [Jaeger and Haas, 2004].

The series is governed by a three-dimensional differential equation

$$\begin{aligned} \dot{x} &= \sigma(y - x), \\ \dot{y} &= x(\rho - z) - y, \\ \dot{z} &= xy - \beta z, \end{aligned} \tag{20}$$

with the original Lorenz' parameters $\sigma = 10$, $\beta = 8/3$, and $\rho = 28$. In our experiments we only use the x coordinate as a one-dimensional ($N_u = 1$) time series. We generated the time series $x(n)$, by solving the differential equation (20) discretized with a time step size 0.01, and discarding every second time step, which in effect gave an approximate solution with a discretization rate 0.02.

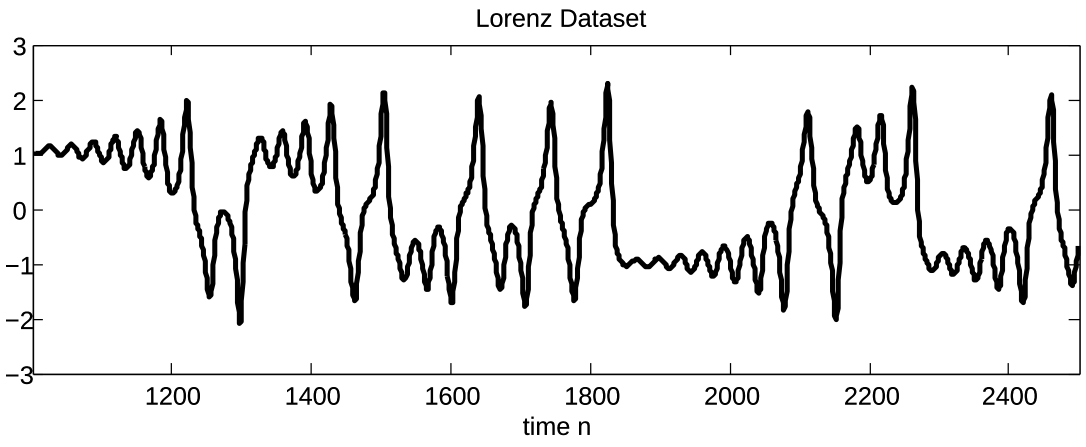

Figure 4: A fragment of the Lorenz data series.

Using this dataset ESNs where trained to predict it one time step ahead, i.e. $u(n) = x(n)$ and $y_{\text{target}}(n) = x(n+1)$, where x is from (20). The length of the generated time series was 5000, with 1000 time steps dedicated to the initial run, 2800 for training, and 1200 for testing. The data set was normalized to have zero mean and standard deviation equal to one before using it.

A fragment of the Lorenz dataset is presented in Figure 4.

6.2 Laser Dataset

As a representative of a real world chaotic time series we used a continuation of the Laser data series, which was originally presented in the 1994 Santa Fe time series prediction competition [Weigend and Gershenfeld, 1994]. The data was recorded

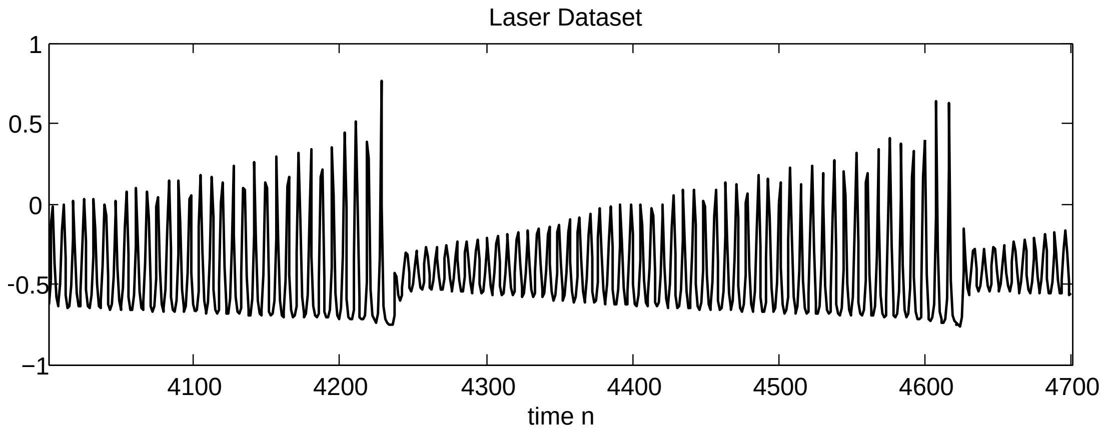

Figure 5: A fragment of the Laser data series.

in a physics laboratory experiment, using an oscilloscope to measure a chaotically pulsating intensity of a laser.

Predicting this time series is a difficult task for several reasons [Jaeger and Haas, 2004]. The most important one is that the time series has to bee predicted on two different time scales. One is the local oscillation prediction, and the other is predicting the breakdown events of this oscillation. The hardest part of this prediction is catching the correct phase of the oscillations after a breakdown [Jaeger and Haas, 2004], which is hard even using the continuation of the original task (i.e. having a longer training series with several breakdown events).

Conventional ESNs can perform well on this task [Jaeger and Haas, 2004], but finding suiting parameters for the training is highly involving [Jaeger, 2007].

As with the Lorenz data set, we used it for training on one time step prediction task, and again used the first 5000 time steps of the continuation of the series with 1000 time steps dedicated to the initial run, 2800 for training, and 1200 for testing. The data set was normalized to have zero mean and standard deviation equal to one before using it.

A fragment of the Laser dataset is presented in Figure 5.

6.3 Synthetic Temporal Pattern Recognition Dataset

As a representative of a temporal pattern recognition task we used a synthetically generated dataset as the one described in [Lukoševičius et al., 2006], but having no time warping. We started out from data that combined a red noise (a $[-0.5, 0.5]$ uniformly distributed white noise with filtered-out 60% of its higher frequencies) background signal $g(t)$ with smoothly embedded random short target sequences $p(t)$ with a similar frequency makeup. Smooth continuous-time signals of this kind were produced, and then discrete-time samples were drawn (the above mentioned frequency makeup corresponds to the discrete-time signals). We did a recognition of only one pattern (i.e. $N_y = 1$), as recognition of multiple patterns would in essence be done independently.

More specifically, first a short (length T_p) target sequence $p(t)$, $p : [0, T_p] \to \mathbb{R}^{N_u}$ was generated in the same (above described) way as $g(t)$, $g : \mathbb{R}^+ \to \mathbb{R}^{N_u}$ (all N_u dimensions were generated independently). Then, in order to smoothly embed $p(t)$ into $g(t)$, a windowing signal $w(t)$ was created, where $t \in [0, T_p]$, $w(0) = w(T_p) = 0$, and $w(t)$ gently rises to 1 after $t = 0$ and smoothly falls again to zero level at $t = T_p$. $w(t)$ was made by filtering a trapezoidal window signal with the same low-pass filter which was used to produce $g(t)$ and $p(t)$, so that $w(t)$ would not introduce any new (high) frequencies. Then the input signal $u(t)$ was produced by smoothly embedding $p(t)$ into $g(t)$ at random positions t_i, where $i \in \mathbb{N}$, and $(t_{i+1} - t_i) \in (T_p, 3T_p)$ is a uniformly distributed random variable with a mean value $2T_p$. At each t_i the embedding of $p(t)$ was $u(t_i + t) = (1 - w(t))g(t_i + t) + w(t)p(t)$, where $t \in [0, T_p]$. The (1-dimensional) desired output signal $y_{\text{target}}(t)$ was constructed by placing Gaussian bumps centered at the time points $t_i + T_b$ on a background zero signal. The height of the bumps is 1 and the width roughly corresponds to the average width of the main lobe of the autocorrelation of $p(t)$. The reason for choosing the form of the bumps in this way is that both pattern and background signals are smooth and smoothly embedded into each other, moreover they are decimated later on, which makes identifying a sharp position with a sharp 0-1 identification signal hard and not practical. The position of appearance of the bump T_b was chosen after some experimentation, to make it as easy to learn for ESNs as possible. This is important since we are going to use this series for training predicting feedbacks (discussed in Section 4.2) and bumps being easier to learn in other than their original locations could distort the evaluation of usefulness of the method in general.

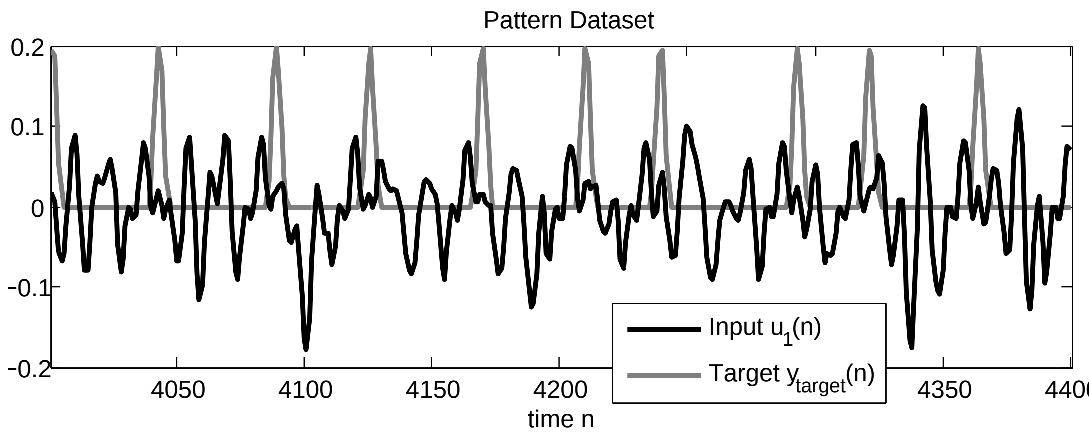

Figure 6: A fragment of the Pattern data series.

The above described continuous time signals were modeled in Matlab by a cubic interpolation of the signals having a six times higher discretization rate, and in some sense stood for the underlying generating process $u(t)$, where $t \in \mathbb{R}^+$.

Discrete time observations $u(n)$ of the process $u(t)$ were drawn as $u(n) = u(\tau(n))$, where $\tau : \mathbb{N} \to \mathbb{R}^+$ is a discretization function. In the obtained signals $u(n)$ a time interval T_p corresponds to 20 time steps, placement of the teacher bumps T_b corresponds to 17 time steps, and $u(n)$ has (as mentioned before) on average 40% of its lower frequencies present. Smooth embedding, and discretization mean that different instances of $p(t)$ may differ considerably in $u(n)$. This, together with $p(t)$ being short and having similar statistics and spectrum as $u(t)$, make this a hard recognition task.

A fragment of the Pattern dataset with only one dimension of the input and $z_{\text{target}}(n)$ scaled down by 0.2 is presented in Figure 6.

Since the shape of the pattern $p(t)$ has good chances to be almost exactly repeated in the background signal $g(t)$, the level of difficulty of the task depends on the number of inputs N_u. The more inputs we have the less the chances are, that the background $g(t)$ will resemble the pattern $p(t)$ in all the N_u dimensions at the same time. We chose $N_u = 4$ for most our experiments as an intermediate level of difficulty.

We used the same length of the signal as for the other two datasets already discussed: 1000 time steps for initialization, 2800 for training and 1200 for testing.

6.4 An Example of a Particularly Hard Task for ESNs

Despite the above mentioned advantages of the ESN training scheme, for some hard problems learning weights of only one output layer is simply not enough. Even though it is possible to use reservoirs of a very big size, they are still finite, and randomly wired pools of neurons might not provide rich enough dynamics of the input echoes so that they could be linearly combined into good approximation of the target signal.

Recently some attention has been payed to an alternative approach to online training, called *Fixed Weight* Neural Networks (FWNNs) (e.g. [Prokhorov et al., 2002] and [Santiago, 2004]). FWNNs are recurrent neural networks (RNNs) which, after training, have the ability to adapt to the changes of the generating process without further change of connection weights. The idea was first presented in 1990 by N. Cotter and R. Conwell [Cotter and Conwell, 1990]. The authors proposed and proved the Fixed Weight Learning Theorem (FWLT) which describes how a fixed-weight recurrent neural networks can approximate with arbitrary precision the dynamics of a feedforward neural network being trained with an adaptive weight learning algorithm, taken both FFNN and the algorithm as one dynamical system. The theorem applies to most networks and learning algorithms in use. They concluded from the theorem that a system which exhibits learning behavior may exhibit no synaptic weight modifications and demonstrated this idea by transforming an online error backpropagating feedforward network into a fixed weight RNN system. The idea of fixed weight learning is also sometimes referred to in the literature by sounding names of "learning to learn" or "metalearning".

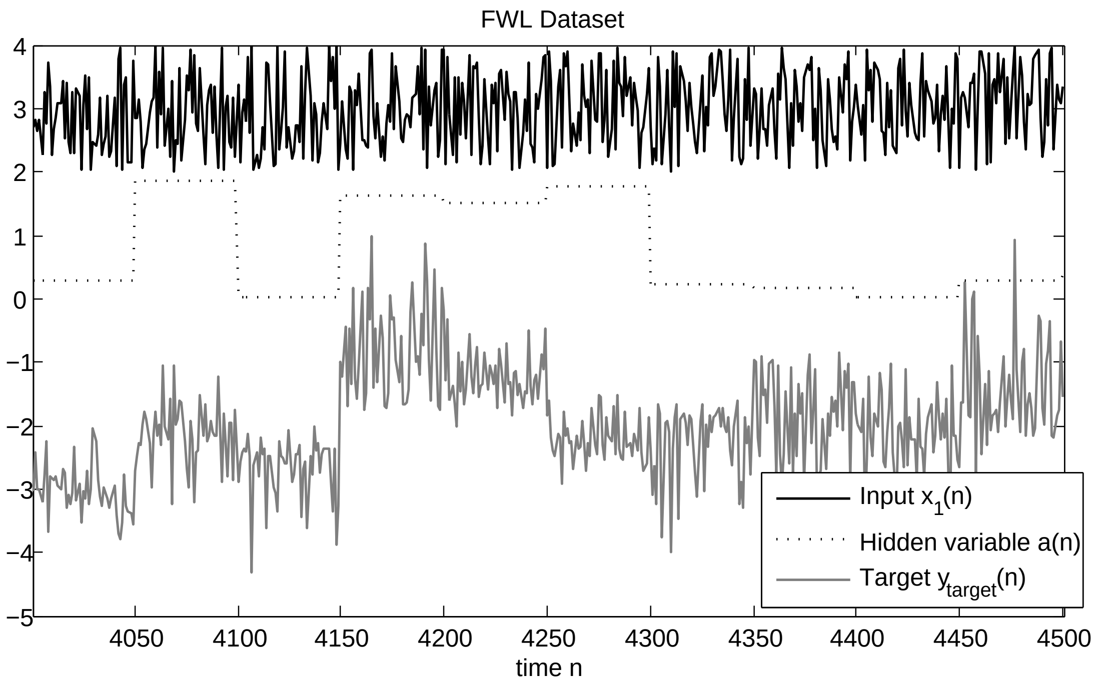

Figure 7: A fragment of the FWL data series.

One concrete formulation of a synthetic fixed weight learning problem which is quite popular in the literature [Prokhorov et al., 2002] [Santiago, 2004] is learning to approximate the parametrized quadratic equation

$$y_{\text{target}}(n) = ax_1(n)^2 + bx_2(n)^2 + cx_1(n)x_2(n) + dx_1(n) + ex_2(n) + f, \qquad (21)$$

where $a, b, c, d, e, f, x_1(n), x_2(n) \in [-1, 1]$; the inputs $x_1(n)$ and $x_2(n)$ are randomly chosen over a uniform distribution at each time step n, and the parameters a, b, c, d, e, f are chosen from the same distribution but change only every T_h time steps. The task at each time step n is to predict the value of $y_{\text{target}}(n)$ from the given input $u(n) = \{x_1(n), x_2(n), y_{\text{target}}(n-1)\}$. Classical ESNs perform poorly in this task [Jaeger, 2007], thus it is interesting to have more details on why it is so, and how the performance could be improved.

A fragment of the fixed weight learning dataset with only $x_1(n)$ and $a(n)$ representing the input is presented in Figure 6. The signals are spread apart here for more visual clarity, in reality they all have zero mean.

As with other datasets we used 1000 time steps for initialization, 2800 for training and 1200 for testing. The period of changing the hidden parameters T_h was set to 50. Each first 25 time steps after changing the hidden parameters were discarded from training to improve the performance, since it is not possible to find out the new parameters instantly (we need *at least* 7 time steps to deduce a, b, c, d, e, f from $x_1(n)$, $x_2(n)$, and $y(n-1)$).

This data set is the only one at hand, for which the ideas of context identifying feedbacks discussed in Section 4.1 are relevant. See Section 7.3 for the analysis of ESN performance with this dataset.

7 Some Empirical Results

In this section we will present some empirical results testing out some of the options for feedback targets discussed in Section 4 on the datasets presented in Section 6, using the techniques for training discussed in Section 5.

7.1 Predictive Feedbacks

Figure 8 presents testing and training NRMS errors and their standard deviations of learning Pattern recognition with ESNs having a different number N_z of predictive feedbacks $z(n) \in \mathbb{R}^{N_z}$ trained on $z_{\text{target}_i}(n) = y_{\text{target}}(n+i)$, $i = 1, \cdots, N_z$. The ESN had $N = 200$ units, spectral radius $\rho(W) = 0.8$, reservoir connectivity 0.1, W_{in} chosen randomly from $[-2, 2]$, W_{ofb} and W_{afb} chosen randomly from $[-0.25, 0.25]$. Both $y(n)$ and $z(n)$ are linear outputs. The outputs were trained one by one in N_z+1 iterations, in a sequence $z_{N_z}(n), \cdots, z_1(n), y(n)$. As a feedback for each already trained output the real signal was used, and for each not yet trained output its teacher signal with added $[-0.0025, 0.0025]$ noise was teacher-forced. The figure presents NRMS errors for training and testing data with their mean values and standard deviations computed over 20 different random initializations of the ESNs. Each of the 20 randomly generated reservoirs was reused with all the numbers N_z of auxiliary feedbacks ranging from 0 to 15.

The results show, that predictive feedbacks can improve the performance of ESNs in this type of tasks considerably and consistently. Up to a certain point ($N_z = 8$), the more predictive feedbacks we use, the better the result gets, both in training and testing. Later the performance starts degrading, which could be attributed to the imprecisions of the too-early predictions.

Figure 9 presents testing and training NRMS errors and their standard deviations of intermediate $z(n)$ and final $y(n)$ outputs of ESNs having $N_z = 15$ predictive feedbacks from the same simulations as in Figure 8. They are again averaged over 20 different ESNs. This graph not surprisingly shows that the more time steps ahead a feedback is predicting, the less precision it has. This can be attributed to two factors. One is the fact that long-distance prediction is more difficult due to the smaller amount of relevant information available, in fact $z_{N_z}(n) = z_{15}(n)$ is trying to predict the appearance of the trainer bump 16 time steps ahead in time, while the center of the bump is only $T_b = 17$ time steps after the pattern is starting to smoothly appear in $u(n)$. No wonder its NRMS error is almost 1. The second factor is the sequence in which the outputs were trained. Outputs trained more recently have less feedbacks teacher-forced and more real feedback signals, thus a smaller uncertainty in the reservoir dynamics during the

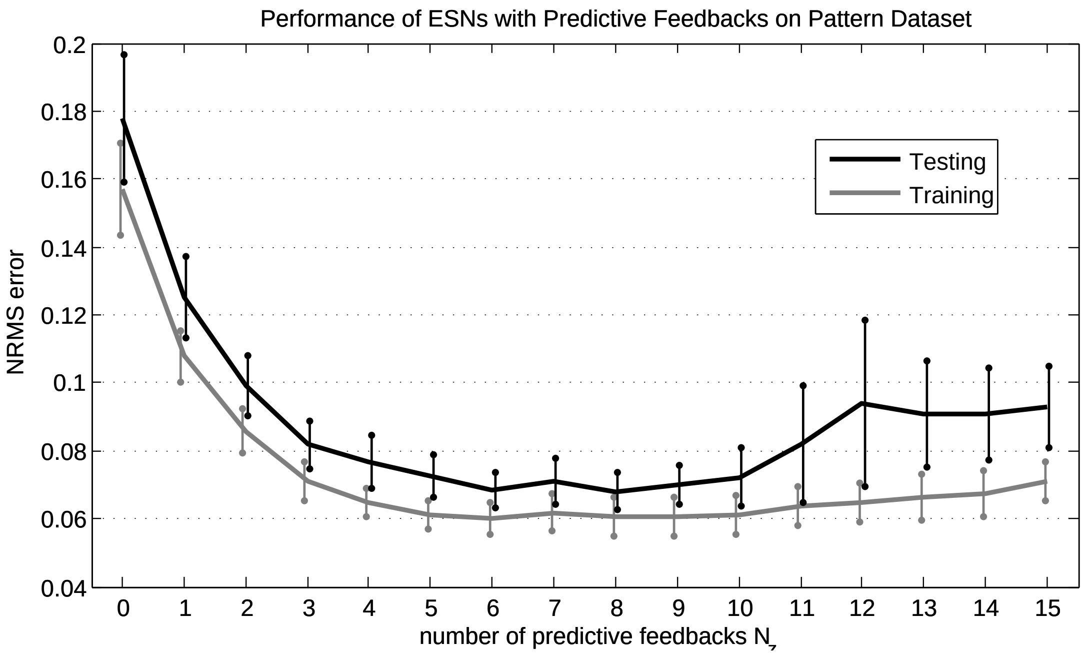

Figure 8: The effect of output predicting feedbacks on performance with Pattern dataset.

process of training. Very low accuracy of the longest-distance predicting feedbacks could explain the degrade of performance in Figure 8, when N_z is too big.

The parameters of the ESN are not optimized here using any automated technique. Due to the stochastic nature of ESNs, and the number of parameters that could be adjusted, we would need to run massive numerical simulations to ensure that they are optimal. Finding good parameters for such performance is not very trivial and requires some manual exploration. A common problem is that the dynamics of the reservoir become unstable, i.e. go to extremes if due to amplification through the feedbacks. One needs to find a good combination of scaling the feedback weights (W_{ofb} and W_{afb}), and noise in the teacher and the units of the reservoir (which has an effect of increasing the stability of the trained dynamical system). On the other hand, applying such measures as noise usually hurts the performance. A good balance here is not always easy to find and as a matter of fact in quite some cases we fail to come up with a setup which is stable and outperforms the conventional ESNs.

We have also tried predicting input as the auxiliary targets for the Pattern dataset $z_{\mathrm{target}}(n) = u(n + 1)$, but this does not seem to be an easy task to learn.

Experiments with predicting the Laser dataset in the same pre-predicting setup as with the Pattern data above led to a bit less impressive but still good performance, which is presented in Figure 10. The results are averaged over 20 different random reservoirs of ESN, reusing every reservoir for all the the different N_z. The ESN had $N = 500$ units, spectral radius $\rho(W) = 0.8$, each unit was on

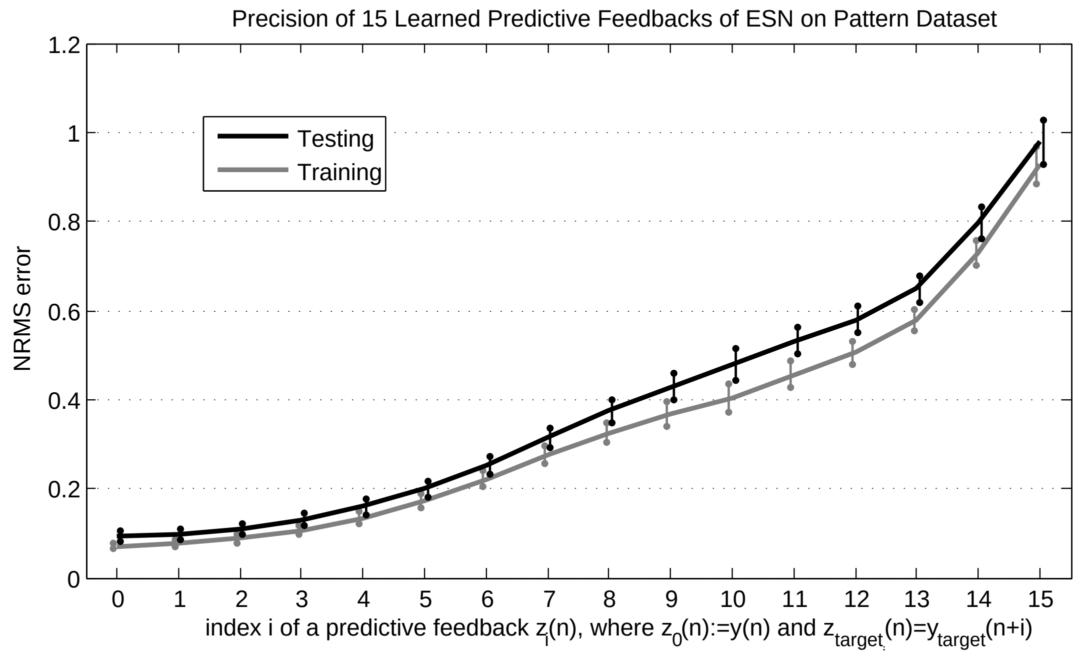

Figure 9: Accuracy of the learned predictive feedbacks with Pattern dataset.

average connected to 48 others, W_{in} chosen randomly from $[-0.8, 0.8]$, W_{ofb} and W_{afb} chosen randomly from $[-0.003, 0.003]$. Both $y(n)$ and $z(n)$ are linear outputs. The outputs were trained one by one in $N_z + 1$ iterations, in a sequence $z_{N_z}(n), \cdots, z_1(n), y(n)$. As a feedback for each already trained output the real signal was used, and for each not yet trained output its teacher signal with added $[-0.225, 0.225]$ noise was teacher-forced.

As it can already be guessed, achieving a stable behavior with the Laser dataset was significantly harder than with the Pattern dataset, so we had to scale down W_{ofb} and W_{afb} and to introduce quite heavy noise in the teacher forcing. This also has its negative effect in the performance: we can observe bigger deviations and also the gain in performance with increasing N_z is sooner surpassed by increasing negative effects of the feedbacks. A bigger testing and training error spread here is due to the nature and the length of the dataset. The model does not learn well how to predict the breakdowns in the testing data, as discussed in Section 6.2.

Figure 11 presents testing and training NRMS errors and their standard deviations of intermediate $z(n)$ and final $y(n)$ outputs of ESNs having $N_z = 15$ predictive feedbacks from the same simulations as in Figure 10. They are again averaged over 20 different ESNs. The error E_{z_i} is again increasing with i, as in the Figure 9 for the similar reasons. The waviness of the graph can most probably be attributed to the oscillations in the original signal, since they both have the same period equal to approximately 7.5 time steps. It is apparently slightly easier to predict the signal this number of time steps ahead.

Figure 12 presents testing and training NRMS errors and their standard devia-

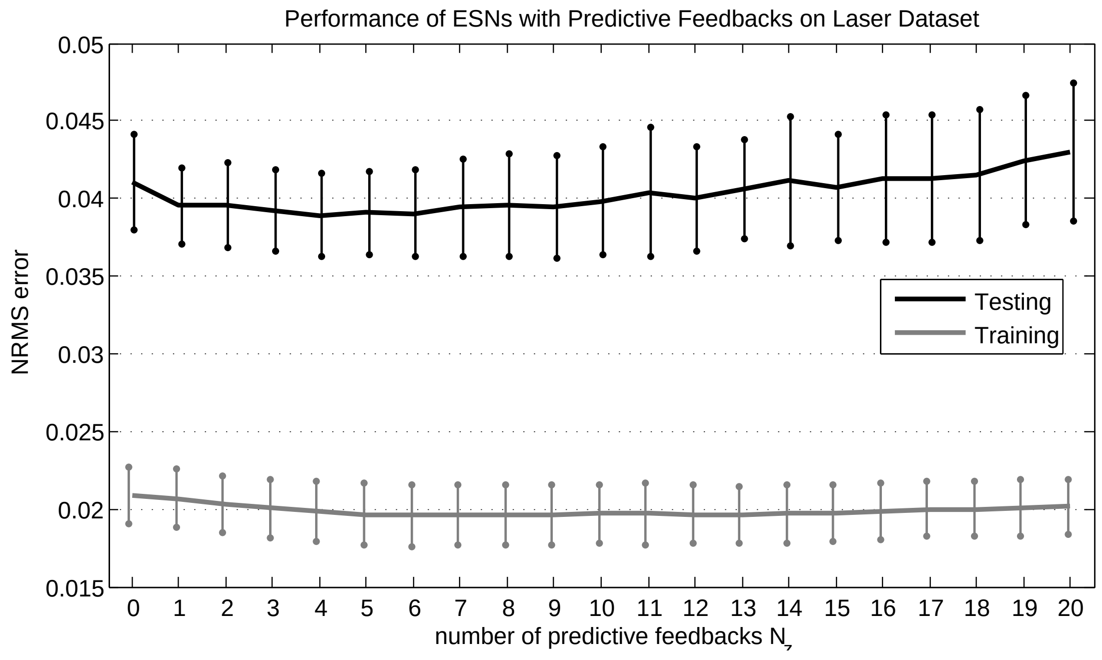

Figure 10: The effect of pre-predicting feedbacks on performance with Laser dataset.

tions of learning the Lorenz attractor prediction task by ESNs having no predictive feedbacks and having a single one trained in $z_{\mathrm{target}}(n) = y_{\mathrm{target}}(n+1)$. The ESN had $N = 500$ units, spectral radius $\rho(W) = 0.7$, each unit was on average connected to 10 others, W_{in} chosen randomly from $[-1, 1]$, W_{afb} chosen randomly from $[-0.4, 0.4]$, $W_{\mathrm{ofb}} = 0$, biases of the reservoir units randomly chosen from $[-1.1, 1.1]$, a white noise from the interval $[-5 \cdot 10^{-9}; 5 \cdot 10^{-9}]$ added to the reservoir activations $x(n)$ during the training run, and white noise from the interval $[-1.25 \cdot 10^{-7}; 1.25 \cdot 10^{-7}]$ added to the teacher forcing signal $z_{\mathrm{target}}(n)$. Both $y(n)$ and $z(n)$ are linear outputs, that were trained using the refined version of the matrix pseudoinverse based regression, presented in Page 16. As in previous cases the results are averaged over 20 different ESNs.

As one can see from the parameters used, it took a very fine tuning of noise levels to achieve the stability of the ESN with auxiliary feedbacks, not sacrificing much of its fine accuracy on this task. This stability, however, was achieved for only one auxiliary feedback. Extensive simulations were also run with several predictive feedbacks trying to fine-tune noise levels in the teacher-force signals individually for each feedback, but in all simulations the average performance degrades drastically (NRMS error $\gg 1$) as some of the feedbacks start working in the generative mode.

We did not apply the output or error predicting feedbacks to the fixed weight learning task, because they are impossible to predict without knowing the next input.

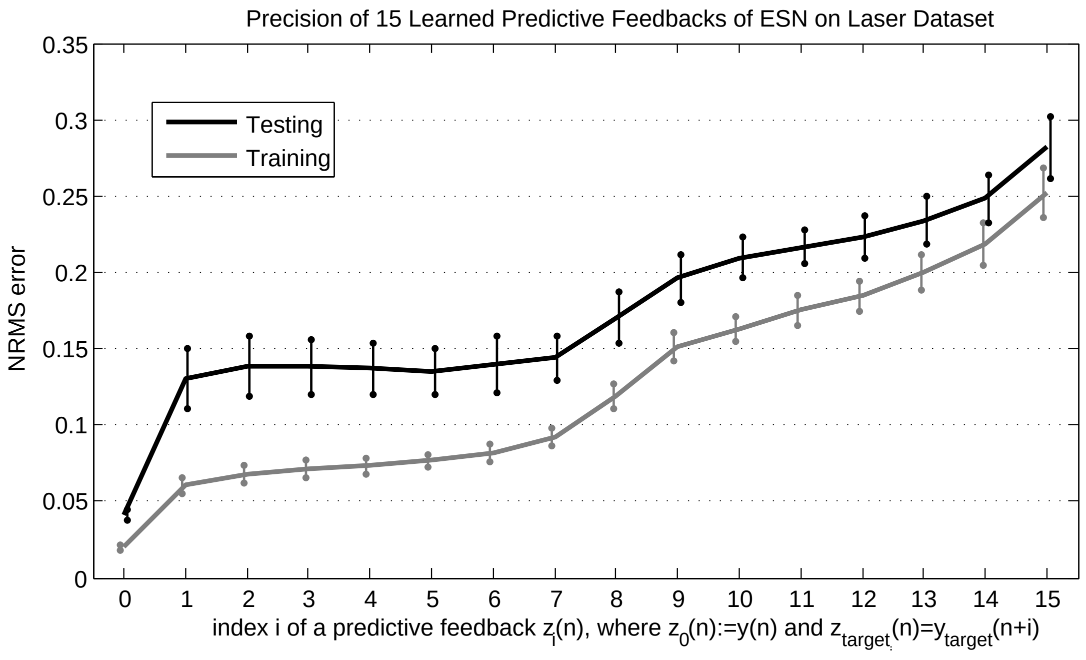

Figure 11: Accuracy of the learned predictive feedbacks with Laser dataset.

7.2 Error Predicting Feedbacks

We have conducted numerous simulations with the Pattern, Laser and Lorenz data sets using the error predicting feedbacks as discussed in Section 4.3. We have tried out a wide range of parameter settings and used linear readouts $z(n)$ predicting error of linear $y(n)$ readouts, MLP readouts $z(n)$ (with different number of layers m and different numbers of units per layers) predicting the error of a linear readouts $y(n)$, and also MLP readouts $z(n)$ predicting error of linear readouts $y(n)$. We used both regular MLP training by stochastic gradient descent and the same one, but doing a linear regression on MLP's output weights, as discussed in Section 5.3. We experimented with different scalings of the error signals (as $z_{\text{target}}(n)$ targets) and feedback weights W_{afb}, tried recursive "error prediction of error prediction" schemes. None of the combinations however exhibited consistently (i.e. averaged over many runs) good results. The feedbacks did not produce a clearly observable improvement and in addition caused frequent outlier divergent behavior.

We also implemented MLP training of $z(n)$ on maximizing the correlation between the error (of $y(n)$) $z_{\text{target}}(n)$ and $z(n)$ as in (15), but this method is hard to tame since it tends to produce outputs of hardly predictable amplitude. Applying the original cascade correlation networks [Fahlman and Lebiere, 1990] as the readout mechanism from the ESN reservoir would, however, be an interesting approach which we have not tried (as it is a bit too far from the scope of the thesis).

One of the reasons why this does not work is, as already pointed out in Section 4.3, that the errors are indeed hard to learn. Even if the task is not hard for

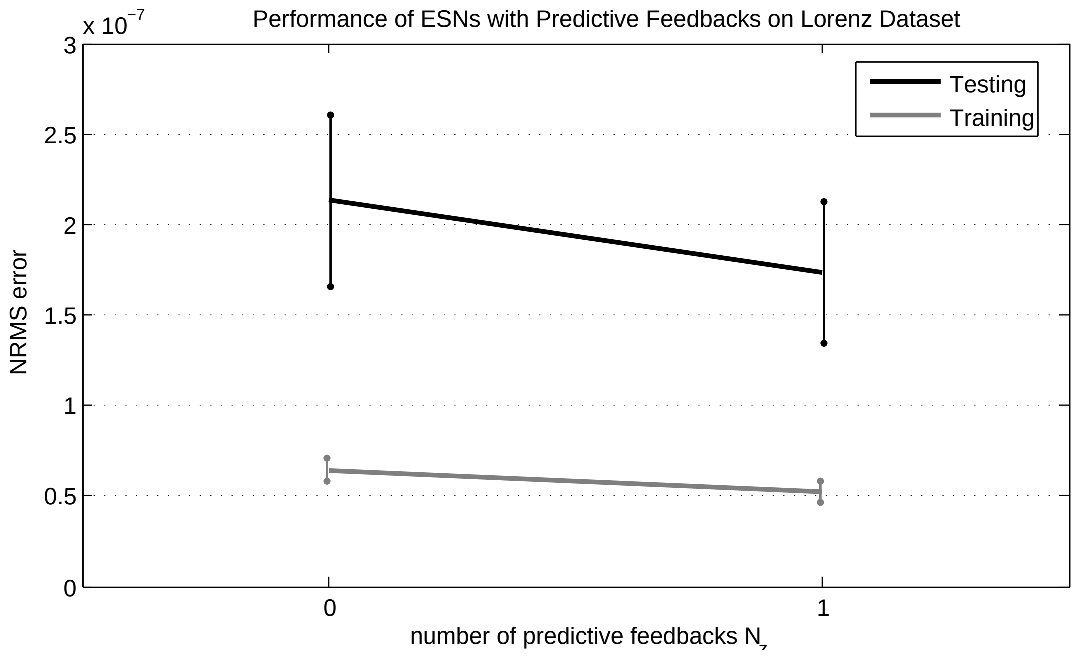

Figure 12: The effect of pre-predicting feedbacks on performance with Lorenz dataset.

regular ESNs, $y(n)$ will model most of this task and the error $y_{\text{target}}(n) - y(n)$ will still become very hard to model, even if $z(n)$ is using a different kind of readout. This way the ESN is "stepping on its own toes", so to say.

Another major difficulty is teacher forcing. Not only it is problematic because the targets $z_{\text{target}}(n)$ get badly learned, but also because of the negative cross-dependencies of the signals: training one feedback immediately invalidates all the assumptions (in the form of teacher-forcings) that other feedbacks had about it. If we successfully model the error, it disappears, thus invalidating our assumptions about its existence and consequently making our modeling unsuccessful again. These negative cross-dependencies can not be eliminated by increasing the noise in teacher forcing signals (as it is possible in some cases), because this noise will propagate to the errors $z_{\text{target}}(n)$, resulting in feedbacks $z(n)$ trained on it resulting even more noise.

All in all the error predicting feedbacks turned out to be not a viable approach in practice.

7.3 Analysis of ESNs Applied to the Fixed Weight Learning Task

In this chapter we will present an analysis of the reasons why ESNs perform badly in the fixed weight learning (FWL) task as presented in Section 6.4. We will provide some insight on why this problem is in particular hard for ESNs, and by that characterize a whole class of problems for which ESNs are not well suited.

The fixed weight learning task is a good example of a generating process with a changing mode. As discussed in Section 4.1 and shown in [Santiago, 2004], learning some kind of representation of the six hidden parameters a, b, c, d, e, f of (21) is an essential sub-task of FWL. Thus a natural candidate for intermediate target $z_{\text{target}}(n)$, when trying to solve this task using ESNs, would be finding out the context of the system, i.e. some kind of a representation of the hidden variables. The network can hopefully infer them from the given sample input and teacher output values, the way that online learning does implicitly. It is interesting how (and if) learning intermediate feedbacks could improve the performance. In this case SFA discussed in Section 4.1 would not be a good choice for extracting the context represented by the parameters a, b, c, d, e, f, since they don't change slowly, but rather infrequently and sharply. We don't know a good method which could do this without directly employing the exact knowledge of the process (21) (the latter is done in [Santiago, 2004]), but we can try training auxiliary feedbacks with $z_{\text{target}}(n) = (a, b, c, d, e, f)^{\text{T}}$ directly. If going for a benchmark of the method, this would amount to cheating, since the parameters should be unknown, but in our case it is interesting to test the benefit of intermediate feedbacks for ESNs.

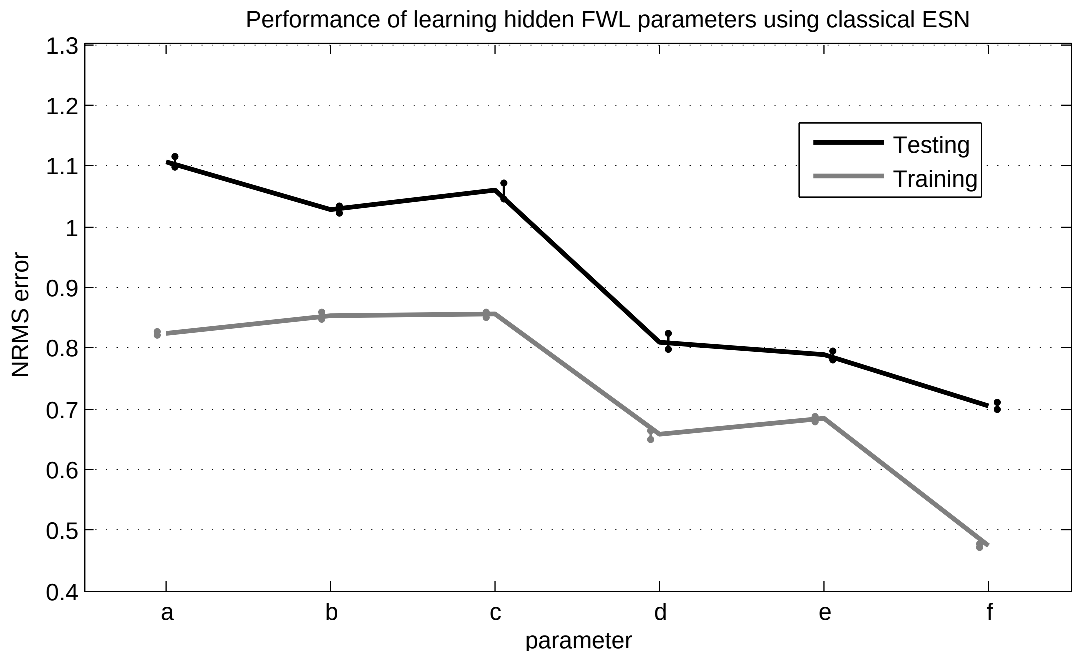

Figure 13: Performance of learning hidden FWL parameters using classical ESN.

Our simulations show (Figures 13 and 14), however, that even learning $\{a, b, c, d, e, f\}$ as $z(n)$ having input $u(n) = \{x_1(n), x_2(n), y_{\text{target}}(n-1)\}$ from (21) is a difficult task for ESNs. On the other hand, even having $\{a, b, c, d, e, f\}$ as part of the input $u(n) = \{x_1(n), x_2(n), a, b, c, d, e, f\}$ (which is analogous to assuming that they are perfectly learned as $z(n-1)$), learning $y_{\text{target}}(n)$ proved to be a very hard task for

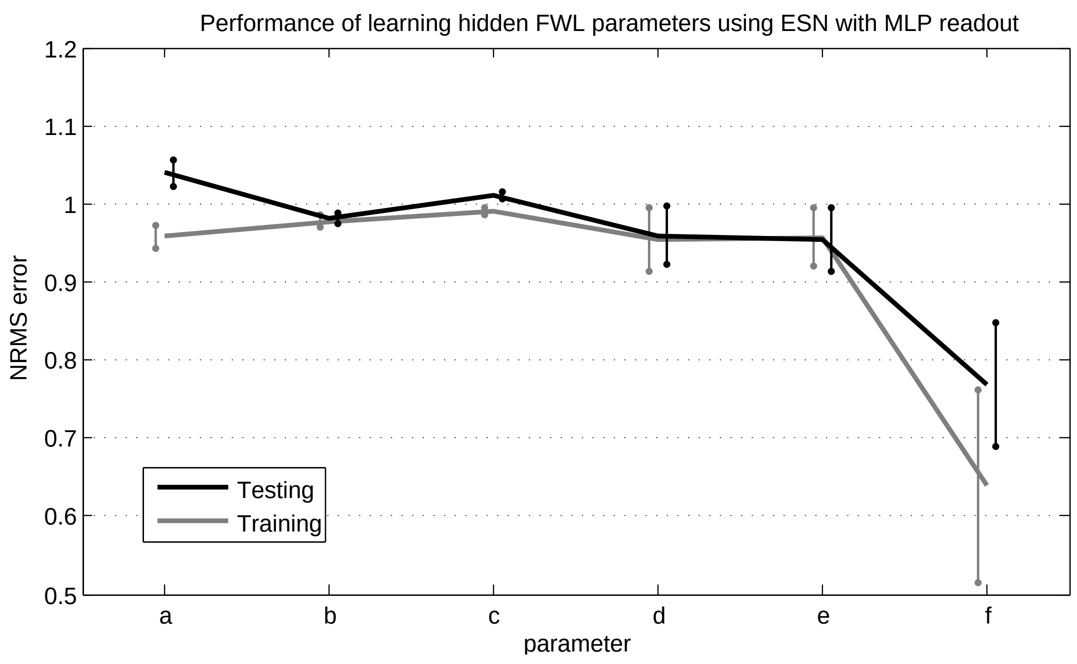

Figure 14: Performance of learning hidden FWL parameters using ESN with MLP readouts.

ESNs.

Let us take a closer look at the latter task. Here $u(n)$ provides the complete information needed to calculate $y_{\text{target}}(n)$ as in (21). Thus this is in essence no longer a temporal problem and the learning model does not need to have any memory. Having memory in fact is harmful here, because the inputs $x_1(n)$, respectively $x_2(n)$ are statistically completely independent from $x_1(n-k)$, respectively $x_2(n-k)$ for any k, thus any form of memory of these inputs from any previous time step $(n-k)$ can only disturb the calculation of current $y(n)$. A similar observation is also true for the rest of the input $\{a, b, c, d, e, f\}$, because it changes randomly as well on transition from one data interval to another (thus memory is again harmful), and remembering any form of them within an interval does not provide any additional useful information. In addition to having no memory, the learning model should also be able to approximate a highly nonlinear function (21).

A good candidate for such a model would be a FFNN (or MLP as a special case). ESN in this context possess just about the opposite properties. It intrinsically has memory (i) and can not learn to approximate a difficult nonlinear function within a single time step (ii). The property (ii) comes from the fact, that if we want the information from the input of ESN to propagate through a series of k reservoir units (1) before reaching the output (which is needed for a difficult nonlinear mapping and in some sense corresponds to a k-layered perceptron), we need to wait for k time steps for this to happen. However, if we postpone the expected output by k time steps (i.e. $u(n)$ corresponds to $y_{\text{target}}(n+k)$), we get an even worse result due to the increased interference between time steps (i.e. memory, as discussed above). An optimal ESN for this task with respect to (i) should have no connections within the reservoir at all, i.e. $\rho(W) = 0$ – no memory. Thus we would in essence no longer have an ESN, but a FFNN with one randomly generated hidden layer ("reservoir") and one learned output layer. Our empirical simulations (not reported here) confirmed that such a setup gives the best performance.

One viable approach of adapting ESNs to this task might be to let it run for several time steps with each input data point $u(n)$. Another approach which we have investigated is presented in Section 8.

8 Layered ESNs

In the context of adapting ESNs to FWL discussed in Section 7.3, we tried out a modification of ESNs, which we call *layered* ESNs (LESNs). The idea of LESNs is to let the information propagate through more than one "layer" of units in the reservoir during one time step. For this we randomly divide the reservoir into k roughly equal subsets, which we call *layers*, L_i, $i = 1, \cdots, k$ ($N_{L_i} = |L_i| \approx \frac{N}{k}$) and update one layer after another within a single time step. So in contrast to updating the whole reservoir at once as in (1), we update it layer by layer:

$$x_{L_i}(n+1) = f(W_{\mathrm{in}L_i}u(n+1) + W_{L_i}x(n,i) + W_{\mathrm{ofb}L_i}y(n)), \quad i = 1, \cdots, k, \quad (22)$$

where $W_{L_i} \in \mathbb{R}^{N_{L_i} \times N}$ are the weights of connections from all the reservoir to the units of layer i, and $x(n,i)$ is the activation state at the time step n and the moment of updating layer i ($i = 1, \cdots, k$ can be seen as minor time steps within a major time step n), defined by

$$x(n,i) = [x_{L_1}(n+1)| \cdots |x_{L_{i-1}}(n+1)|x_{L_i}(n)| \cdots |x_{L_k}(n)], \quad (23)$$

i.e. when layers $L_1, \cdots, L_{i-1}$ have already been updated. In this way information propagates through the reservoir connections going from L_i to L_j, where $i < j$ within a single time step. This is illustrated in the Figure 15. The LESN on the right hand side is produced by dividing the reservoir of the classical ESN on the left into tree equal-sized layers. The thin dark arrows indicate connections through which previous activations $x(n-1)$ are propagated at a time step n, and bold arrows show connections, which use current (time step n) signals. The bold arrows on the right appear as the result of the division of the reservoir into layers (23) and graphically are the connections that cross any of the two boundaries between layers going from left to right. In this section we will refer to the connections as *bold* if they propagate information from activations $x(n)$ to $x(n)$ (in contrast to other connections that propagate information from activations $x(n-1)$ to $x(n)$).

Because of the fact that the layers are equally-sized, the number of connections between any two layers is on average the same. Thus it is not hard to show that

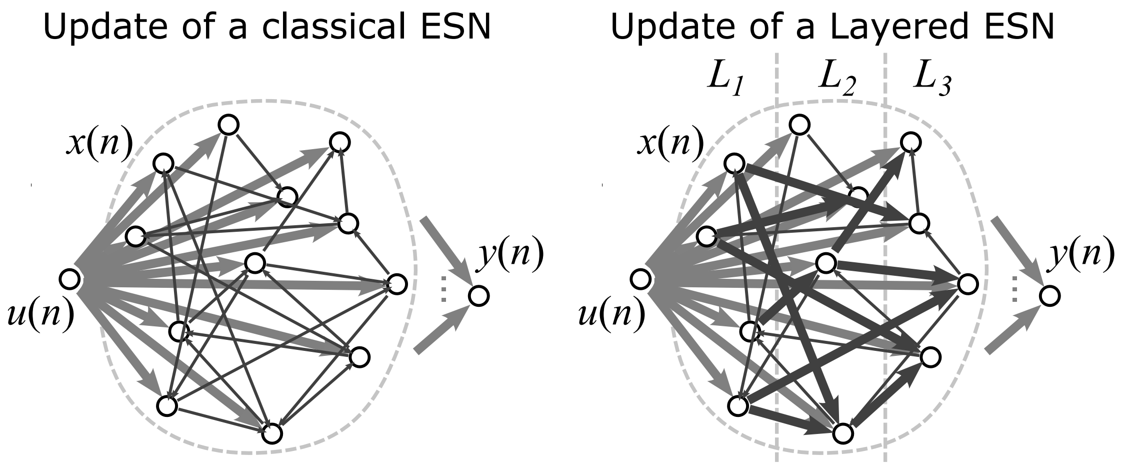

Figure 15: The difference between updating a classical ESN and a Layered ESN (LESN).

having k layers on average $M\frac{k\frac{k-1}{2}}{k^2} = M\frac{k-1}{2k}$ connections out of M will become bold. In the limiting case where the number of layers k approaches the size of the reservoir N, on average nearly half of all the connections will become bold.

Classical ESNs can be seen as a special case of LESNs having only a single layer $k = 1$. Readout (and feedback) mechanisms for LESNs can be chosen from all the same options as for classical ESNs. It is obvious, that bold connections can only form a graph without cycles.

The effect of increasing the number of layers k (and thus the number of bold connections) is that at a time step n less information from the time step $n-1$ is preserved (a smaller amount of memory) and we get a more nonlinear and rich random representation ("echoes") of the current input $u(n)$ in $x(n)$. Thus LESNs could perform better than ESNs in the tasks where these properties are beneficial.

Testing this approach on the original FWL task (having "no knowledge" about the hidden parameters), however did not exhibit an improvement. Figure 16 presents testing and training NRMS errors and their standard deviations when applying ESNs with different number of layers k on the Lorenz dataset, averaged over 20 randomly generated reservoirs. The LESN (no auxiliary feedbacks) had $N = 500$ units, spectral radius $\rho(W) = 0.92$, each unit was on average connected to 10 others, W_{in} chosen randomly from $[-1, 1]$, W_{ofb} set to 0 and $y(n)$ being a linear output. These parameters were manually tunned to give a best observed performance on the data set with a classical ESN.

The reason why layers of the reservoir did not improve the performance is most probably that not *any* nonlinear representations of the input are needed, but but very specific ones. Thus a multilayer RNN with random weight connections between the layers is not providing a good basis for solving this problem by a linear combination.

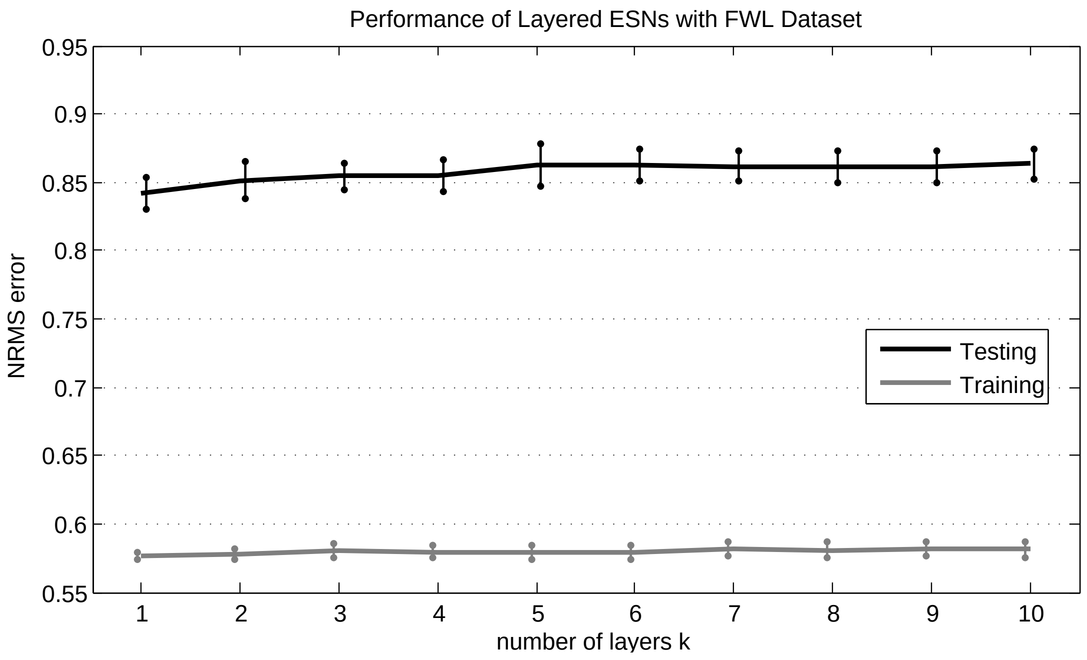

Figure 16: The effect of ESN layers on performance with the Fixed Weight Learning dataset.

LESNs, nevertheless, as a way of biasing the input representations in the reservoir to the properties mentioned above proved to be a viable method for some problems. Figure 17 presents testing and training NRMS errors and their standard deviations when applying ESNs with different number of layers k on the Laser dataset, averaged over 100 randomly generated reservoirs. The LESN (no auxiliary feedbacks) had $N = 500$ units, spectral radius $\rho(W) = 0.7$, each unit was on average connected to 12 others, W_{in} chosen randomly from $[-0.02, 0.02]$, units had a bias values randomly ranging in $[-1.6, 1.6]$ (in contrast to normal, and thus omitted, $[-1, 1]$ in other cases), W_{ofb} set to 0, having a single layer output with $f_{\text{out}} = \tanh$ and (thus) the training signals scaled and shifted to $[-0.76, 0.76]$. These parameters were manually tunned to give a best observed performance on the data set with a classical ESN.

In this case having five layers in the reservoir proved to be a good bias leading to a better generalization of the ESN, i.e. modeling the testing data better even if the training error increases.

9 Final Thoughts and Future Work

We have explored a wide range of ideas along the lines of improving performance of ESNs by training their feedbacks. Some of them were more successful than others. We could improve the results for all relevant datasets using output prepredicting feedbacks, which seem to be an almost universally viable approach for

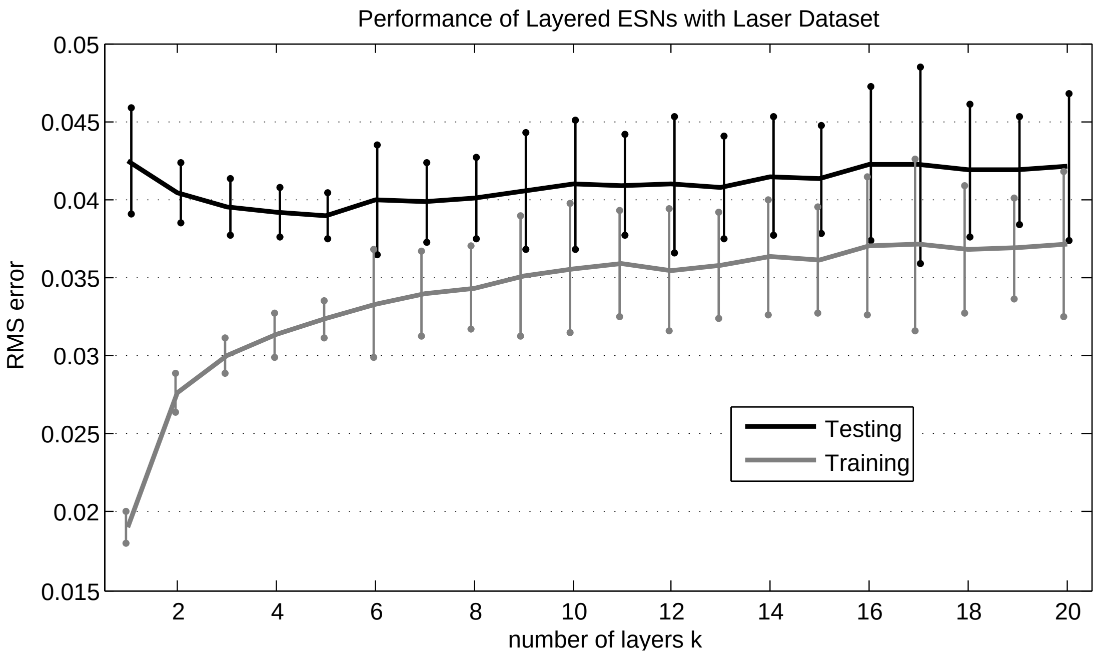

Figure 17: The effect of ESN layers on performance with Laser dataset.

time series prediction tasks. On the other hand, feedbacks predicting errors did not work stably for any of our data sets, neither using feedbacks trained as linear combinations, nor MLPs. In any case, the potential of the trained feedbacks for empowering ESNs is undisputable and, as shown, can in effect replace training of the internal weights of the reservoir. However, which way it can be best exploited and what are the ultimate trade-offs (e.g. in terms of training complexity) of gaining this power are questions that are largely yet unanswered.

In addition we have shown that for some tasks a more expressive instantaneous mapping is required than classical ESNs can provide. It can not be implemented by just adding trained feedbacks, as they come into effect only in subsequent time steps. The proposed Layered ESNs is an attempt toward this direction, however the space of possible functions that the *multilayer* network can implement is too vast to be sufficiently “covered” by the random weights. Thus some other methods of reservoir pre-adaptation should be used here.

Acknowledgments

The author would like to thank Prof. Herbert Jaeger who supervised the Master Thesis on which this report is based, and Planet GmbH (Raben Steinfeld, Germany) for partial financial support of our research in ESNs.

References

[Bishop, 1995] Bishop, C. M. (1995). *Neural Networks for Pattern Recognition.* Oxford University Press, Oxford, UK.

[Campbell, 1997] Campbell, C. (1997). Constructive learning techniques for designing neural network systems. In *Neural Network Systems Technologies and Applications.* Academic Press, San Diego, CA.

[Cotter and Conwell, 1990] Cotter, N. E. and Conwell, P. R. (1990). Fixed-weight networks can learn. In *Proceedings of International Joint Conference on Neural Networks (IJCNN1990)*, pages 553–559 vol. 3.

[Fahlman and Lebiere, 1990] Fahlman, S. E. and Lebiere, C. (1990). The cascade-correlation learning architecture. In *Advances in neural information processing systems 2 (NIPS)*, pages 524–532, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Hammer and Steil, 2002] Hammer, B. and Steil, J. J. (2002). Tutorial: Perspectives on learning with rnns. In *Proceedings of European Symposium on Artificial Neural Networks (ESANN'2002)*, pages 357–368, Bruges (Belgium).

[Irie and Miyake, 1988] Irie, B. and Miyake, S. (1988). Capabilities of three-layered perceptrons. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 641–648 vol. 1, San Diego, CA.

[Jaeger, 2001] Jaeger, H. (2001). The "echo state" approach to analysing and training recurrent neural networks. Technical Report GMD Report 148, German National Research Center for Information Technology.

[Jaeger, 2007] Jaeger, H. (2005-2007). Personal communications.

[Jaeger and Haas, 2004] Jaeger, H. and Haas, H. (2004). Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, pages 78–80.

[Jain, 2004] Jain, A. (2004). Improve on chaotic time series prediction using MLPs for output training. Bachelor's thesis, International University Bremen.

[Liebald, 2004] Liebald, B. (2004). Exploration of effects of different network topologies on the esn signal crosscorrelation matrix spectrum. Bachelor's thesis, International University Bremen.

[Lorenz, 1963] Lorenz, E. (1963). Deterministic nonperiodic flow. *Journal of Atmospheric Science*, 20:130–141.

[Lukoševičius et al., 2006] Lukoševičius, M., Popovici, D., Jaeger, H., and Siewert, U. (2006). Time warping invariant echo state networks. Technical Report No. 2, International University Bremen.

[Maass et al., 2006] Maass, W., Joshi, P., and Sontag, E. D. (2006). Principles of real-time computing with feedback applied to cortical microcircuit models. In *Advances in Neural Information Processing Systems (NIPS).* to appear.

[Maass et al., 2002] Maass, W., Natschläger, T., and Markram, H. (2002). Real-time computing without stable states: a new framework for neural computation based on perturbations. *Neural Comput.*, 14(11):2531–2560.

[Nguyen and Widrow, 1990] Nguyen, D. and Widrow, B. (July 1990). Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. In *Proceedings of the International Joint Conference on Neural Networks*, pages 21–26 vol. 3, San Diego, CA.

[Ozturk and Principe, 2005] Ozturk, M. C. and Principe, J. C. (2005). Computing with transiently stable states. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN2005).*

[Prokhorov et al., 2002] Prokhorov, D. V., Feldkamp, L. A., and Tyukin, I. Y. (2002). Adaptive behavior with fixed weights in rnn: an overview. In *Proceedings of International Joint Conference on Neural Networks (IJCNN2002)*, pages 2018–2023.

[Santiago, 2004] Santiago, R. A. (2004). Context discerning multifunction networks: Reformulating fixed weight neural networks. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN2004).*

[Vogl et al., 1988] Vogl, T., Mangis, J., Rigler, A., Zink, W., and Alkon, D. (1988). Accelerating the convergence of the back-propagation method. *Biological Cybernetics*, 59:257–263.

[Weigend and Gershenfeld, 1994] Weigend, A. S. and Gershenfeld, N. A., editors (1994). *Time Series Prediction: Forecasting the Future and Understanding the Past.* Addison-Wesley, Reading, MA.

[Wiskott and Sejnowski, 2002] Wiskott, L. and Sejnowski, T. (2002). Slow feature analysis: Unsupervised learning of invariances. *Neural Computation*, 14(4):715–770.

[Wolpert, 2001] Wolpert, D. H. (2001). The supervised learning no-free-lunch theorems. In *Proc. 6th Online World Conf. on Soft Computing in Industrial Applications.*