A Practical Guide to Applying Echo State Networks

Mantas Lukoševičius

Jacobs University Bremen, Campus Ring 1,
28759 Bremen, Germany
m.lukosevicius@jacobs-university.de

Abstract. Reservoir computing has emerged in the last decade as an alternative to gradient descent methods for training recurrent neural networks. Echo State Network (ESN) is one of the key reservoir computing "flavors". While being practical, conceptually simple, and easy to implement, ESNs require some experience and insight to achieve the hailed good performance in many tasks. Here we present practical techniques and recommendations for successfully applying ESNs, as well as some more advanced application-specific modifications.

To appear in *Neural Networks: Tricks of the Trade, Reloaded.*
G. Montavon, G. B. Orr, and K.-R. Müller, editors, Springer, 2012.

1 Introduction

Training Recurrent Neural Networks (RNNs) is inherently difficult. This (de-) motivates many to avoid them altogether. RNNs, however, represent a very powerful generic tool, integrating both large dynamical memory and highly adaptable computational capabilities. They are the Machine Learning (ML) model most closely resembling biological brains, the substrate of natural intelligence.

Error backpropagation (BP) [1] is to this date one of the most important achievements in artificial neural network training. It has become the standard method to train especially Feed-Forward Neural Networks (FFNNs). Many useful practical aspects of BP are discussed in other chapters of this book and in its previous edition, e.g., [2]. BP methods have also been extended to RNNs [3,4], but only with a partial success. One of the conceptual limitations of BP methods for RNNs is that bifurcations can make training non-converging [5]. Even when they do converge, this convergence is slow, computationally expensive, and can lead to poor local minima.

Ten years ago an alternative trend of understanding, training, and using RNNs has been proposed with Echo State Networks (ESNs) [6,7] in ML, and Liquid State Machines (LSMs) [8] in computational neuroscience. It was shown that RNNs often work well enough even without full adaptation of all network weights. In the classical ESN approach the RNN (called *reservoir*) is generated randomly, and only the readout from the reservoir is trained. It should be noted that this basic idea was first clearly spelled out in a neuroscientific model of the corticostriatal processing loop [9]. Perhaps surprisingly this approach yielded excellent performance in many benchmark tasks, e.g., [6,10,11,12,13,14].

The trend started by ESNs and LSMs became lately known as Reservoir Computing (RC) [15]. RC is currently a prolific research area, giving important insights into RNNs, procuring practical machine learning tools, as well as enabling computation with non-conventional hardware [16]. RC today subsumes a number of related methods and extensions of the original idea [17], but the original ESN approach still holds its ground for its simplicity and power.

The latest developments in BP for RNNs, second-order gradient descent methods called Hessian-free optimization, presented in [18] and discussed in a chapter [19] of this book, alleviate some of the mentioned shortcomings. In particular, they perform better on problems which require long memory [18]. These are known to be hard for BP RNN training [20], unless networks are specifically designed to deal with them [21]. Structural damping of Hessian-free optimization [18], an online adaptation of the learning process which penalizes big changes in RNN activations, likely tends to drive the learning process away from passing through many bifurcations (that are exactly big changes in activations

and can probably be anticipated and avoided to some extent). On a benchmark suite designed to challenge long short-term memory acquisition, ESNs however still outperform Hessian-free trained RNNs [22].

ESNs from their beginning proved to be a highly practical approach to RNN training. It is conceptually simple and computationally inexpensive. It reinvigorated interest in RNNs, by making them accessible to wider audiences. However, the apparent simplicity of ESNs can sometimes be deceptive. Successfully applying ESNs needs some experience. There is a number of things that can be done wrong. In particular, the initial generation of the raw reservoir network is influenced by a handful of global parameters, and these have to be set judiciously. The same, however, can be said about virtually every ML technique. Techniques and recommendations on successfully applying ESNs will be addressed in this work.

We will try to organize the "best practices" of ESNs into a logical order despite the fact that they are often non-sequentially interconnected. We will start with defining the ESN model and the basic learning procedure in Section 2. Then we will detail out guidelines on producing good reservoirs in Section 3, various aspects of training different types of readouts in Section 4, and dealing with output feedback in Section 5. We will end with a short summary in Section 6.

2 The Basic Model

ESNs are applied to supervised temporal ML tasks where for a given training input signal $\mathbf{u}(n) \in \mathbb{R}^{N_\mathrm{u}}$ a desired target output signal $\mathbf{y}^{\mathrm{target}}(n) \in \mathbb{R}^{N_\mathrm{y}}$ is known. Here $n = 1, \ldots, T$ is the discrete time and T is the number of data points in the training dataset. In practice the dataset can consist of multiple sequences of varying lengths, but this does not change the principles. The task is to learn a model with output $\mathbf{y}(n) \in \mathbb{R}^{N_\mathrm{y}}$, where $\mathbf{y}(n)$ matches $\mathbf{y}^{\mathrm{target}}(n)$ as well as possible, minimizing an error measure $E(\mathbf{y}, \mathbf{y}^{\mathrm{target}})$, and, more importantly, generalizes well to unseen data. The error measure E is typically a Mean-Square Error (MSE), for example Root-Mean-Square Error (RMSE)

$$E(\mathbf{y}, \mathbf{y}^{\mathrm{target}}) = \frac{1}{N_\mathrm{y}} \sum_{i=1}^{N_\mathrm{y}} \sqrt{\frac{1}{T} \sum_{n=1}^{T} \left(y_i(n) - y_i^{\mathrm{target}}(n)\right)^2}, \tag{1}$$

which is also averaged over the N_y dimensions i of the output here.

The RMSE can also be dimension-wise normalized (divided) by the variance of the target $\mathbf{y}^{\mathrm{target}}(n)$, producing a Normalized Root-Mean-Square Error (NRMSE). The NRMSE has an absolute interpretation: it does not depend on the arbitrary scaling of the target $\mathbf{y}^{\mathrm{target}}(n)$ and the value of 1 can be achieved with a simple constant output $\mathbf{y}(n)$ set to the mean value of $\mathbf{y}^{\mathrm{target}}(n)$. This suggests that a reasonable model of a stationary process should achieve the NRMSE accuracy between zero and one.

The normalization and the square root parts are more for human interpretability, as the optimal output $\mathbf{y}^{\mathrm{target}}$ minimizing any MSE is equivalent to the one minimizing (1), as long as no additional penalties or weighting are introduced into the equation, such as discussed in Sections 4.2 and 4.6.

ESNs use an RNN type with leaky-integrated discrete-time continuous-value units. The typical update equations are

$$\tilde{\mathbf{x}}(n) = \tanh\left(\mathbf{W}^{\mathrm{in}}[1; \mathbf{u}(n)] + \mathbf{W}\mathbf{x}(n-1)\right), \tag{2}$$

$$\mathbf{x}(n) = (1-\alpha)\mathbf{x}(n-1) + \alpha\tilde{\mathbf{x}}(n), \tag{3}$$

where $\mathbf{x}(n) \in \mathbb{R}^{N_\mathrm{x}}$ is a vector of reservoir neuron activations and $\tilde{\mathbf{x}}(n) \in \mathbb{R}^{N_\mathrm{x}}$ is its update, all at time step n, $\tanh(\cdot)$ is applied element-wise, $[\cdot;\cdot]$ stands for a vertical vector (or matrix) concatenation, $\mathbf{W}^{\mathrm{in}} \in \mathbb{R}^{N_\mathrm{x} \times (1+N_\mathrm{u})}$ and $\mathbf{W} \in \mathbb{R}^{N_\mathrm{x} \times N_\mathrm{x}}$ are the input and recurrent weight matrices respectively, and $\alpha \in (0, 1]$ is the leaking rate. Other sigmoid wrappers can be used besides the tanh, which however is the most common choice. The model is also sometimes used without the leaky integration, which is a special case of $\alpha = 1$ and thus $\tilde{\mathbf{x}}(n) \equiv \mathbf{x}(n)$.

The linear readout layer is defined as

$$\mathbf{y}(n) = \mathbf{W}^{\mathrm{out}}[1; \mathbf{u}(n); \mathbf{x}(n)], \tag{4}$$

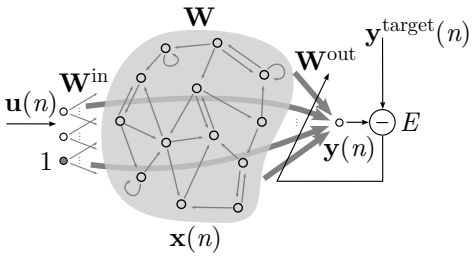


Fig. 1: An echo state network.

where $\mathbf{y}(n) \in \mathbb{R}^{N_\text{y}}$ is network output, $\mathbf{W}^\text{out} \in \mathbb{R}^{N_\text{y} \times (1+N_\text{u}+N_\text{x})}$ the output weight matrix, and $[\cdot;\cdot;\cdot]$ again stands for a vertical vector (or matrix) concatenation. An additional nonlinearity can be applied to $\mathbf{y}(n)$ in (4), as well as feedback connections $\mathbf{W}^\text{fb}$ from $\mathbf{y}(n-1)$ to $\tilde{\mathbf{x}}(n)$ in (2). A graphical representation of an ESN illustrating our notation and the idea for training is depicted in Figure 1.

The original method of RC introduced with ESNs [6] was to:

1. generate a large random reservoir RNN $(\mathbf{W}^\text{in}, \mathbf{W}, \alpha)$;
2. run it using the training input $\mathbf{u}(n)$ and collect the corresponding reservoir activation states $\mathbf{x}(n)$;
3. compute the linear readout weights $\mathbf{W}^\text{out}$ from the reservoir using linear regression, minimizing the MSE between $\mathbf{y}(n)$ and $\mathbf{y}^\text{target}(n)$;
4. use the trained network on new input data $\mathbf{u}(n)$ computing $\mathbf{y}(n)$ by employing the trained output weights $\mathbf{W}^\text{out}$.

In subsequent sections we will delve deeper into the hidden intricacies of this procedure which appears so simple on the surface, and spell out practical hints for the concrete design choices that wait on the way. More specifically, Step 1 is elaborated on in Section 3; Step 2 is done by Equations (2) and (3), with initialization discussed in Section 4.5; Step 3 is formally defined and options explained in Section 4 with additional options for some particular applications in Section 5; and Step 3 is again performed by Equations (2), (3), and (4).

3 Producing a Reservoir

For producing a good reservoir it is important to understand what function it is serving.

3.1 Function of the Reservoir

In practice it is important to keep in mind that the reservoir acts **(i)** as a nonlinear expansion and **(ii)** as a memory of input $\mathbf{u}(n)$ at the same time.

There is a parallel between RC and kernel methods in ML. The reservoir can be seen as (i) a nonlinear high-dimensional expansion $\mathbf{x}(n)$ of the input signal $\mathbf{u}(n)$. For classification tasks, input data $\mathbf{u}(n)$ which are not linearly separable in the original space $\mathbb{R}^{N_\text{u}}$, often become so in the expanded space $\mathbb{R}^{N_\text{x}}$ of $\mathbf{x}(n)$, where they are separated by $\mathbf{W}^\text{out}$. In fact, employing the "kernel trick" to integrate over all possible reservoirs is also possible in the context of RC, even though not really practical [23].

At the same time, (ii) the reservoir serves as a memory, providing temporal context. This is a crucial reason for using RNNs in the first place. In the tasks where memory is not necessary, non-temporal ML techniques implementing functional mappings from current input to current output should be used.

Both aspects (i) and (ii) combined, the reservoir, being an input-driven dynamical system, should provide a rich and relevant enough signal space in $\mathbf{x}(n)$, such that the desired $\mathbf{y}^\text{target}(n)$ could be obtained by linear combination from it. There is however some trade-off between (i) and (ii) when setting the parameters of the reservoir [24], which we will explain in more detail.

3.2 Global parameters of the Reservoir

Given the RNN model (2),(3), the reservoir is defined by the tuple $(\mathbf{W}^{\text{in}}, \mathbf{W}, \alpha)$. The input and recurrent connection matrices $\mathbf{W}^{\text{in}}$ and $\mathbf{W}$ are generated randomly according to some parameters discussed later and the leaking rate α is selected as a free parameter itself.

In analogy to other ML, and especially NN, approaches, what we call "parameters" here could as well be called "meta-parameters" or "hyper-parameters", as they are not concrete connection weights but parameters governing their distributions. We will call them "global parameters" to better reflect their nature, or simply "parameters" for brevity.

The defining global parameters of the reservoir are: the size N_{x}, sparsity, distribution of nonzero elements, and spectral radius of $\mathbf{W}$; scaling(-s) of $\mathbf{W}^{\text{in}}$; and the leaking rate α. We will now proceed in this order to give more details on each of these design choices and intuitions on how to make them. Then, in Section 3.3, we will summarize by advising how to prioritize these global parameters and tune the really important, or rather task-specific, ones in a principled way.

3.2.1 Size of the Reservoir One obviously crucial parameter of the model (2)(3) is N_{x}, the number of units in the reservoir.

The general wisdom is that the bigger the reservoir, the better the obtainable performance, *provided appropriate regularization measures are taken against overfitting* (see Section 4.1). Since training and running an ESN is computationally cheap compared to other RNN approaches, reservoir sizes of order 10^4 are not uncommon [13]. The bigger the space of reservoir signals $\mathbf{x}(n)$, the easier it is to find a linear combination of the signals to approximate $\mathbf{y}^{\text{target}}(n)$. In our experience the reservoir can be too big only when the task is trivial and there is not enough data available $T < 1 + N_{\text{u}} + N_{\text{x}}$.

> For challenging tasks use as big a reservoir as you can afford computationally.

That being said, computational trade-offs are important. In academic settings, when comparing different approaches instead of going for the best possible performance, authors often limit their reservoir sizes for convenience and compatibility of results. Even when going for the best performance, starting with the biggest possible reservoir from the beginning is cumbersome.

> Select global parameters with smaller reservoirs, then scale to bigger ones.

The tuning of global parameters (described below) often needs multiple trials, thus each should not consume too much time. Good parameters are usually transferable to bigger reservoirs, but some trials with big reservoirs can also be done to confirm this.

A lower bound for the reservoir size N_{x} can roughly be estimated by considering the number of independent real values that the reservoir must remember from the input to successfully accomplish the task. The maximal number of stored values, called memory capacity, in ESN can not exceed N_{x} [25].

> N_{x} should be at least equal to the estimate of independent real values the reservoir has to remember from the input to solve its task.

For i.i.d. inputs $\mathbf{u}(n)$, this estimate is N_{u} times a rough estimate of how many time steps the inputs should be remembered to solve the task. While the result in [25] is precise for i.i.d. inputs, in practice there are often temporal and inter-channel correlations in $\mathbf{u}(n)$, that make it somewhat "compressible". Also, the shapes of the "forgetting curves" of the reservoirs are typically not rectangular (depend on other parameters), i.e., the forgetting is not instantaneous but gradual. As a result, reservoir can often make do with smaller sizes.

3.2.2 Sparsity of the Reservoir In the original ESN publications it is recommended to make the reservoir connections sparse, i.e., make most of elements in $\mathbf{W}^{\text{in}}$ equal to zero. In our practical experience also sparse connections tend to give a slightly better performance. In general, sparsity of the reservoir does not affect the performance much and this parameter has a low priority to be optimized. However, sparsity enables fast reservoir updates if sparse matrix representations are used.

> Connect each reservoir node to a small fixed number of other nodes (e.g., 10) on average, irrespective of the reservoir size. Exploit this reservoir sparsity to speedup computation.

If regardless of reservoir size, a fixed fanout number is chosen, the computational cost of network state updates grows only linearly with the network size instead of quadratically. This greatly reduces the cost of running big reservoirs. The computational savings require virtually no additional effort when the programming environment supports efficient representation and operations with sparse matrices, which many do.

3.2.3 Distribution of Nonzero Elements The matrix $\mathbf{W}$ is typically generated sparse, with nonzero elements having an either a symmetrical uniform, discrete bi-valued, or normal distribution centered around zero. Different authors prefer different distributions. We usually prefer a uniform distribution for its continuity of values and boundedness. Gaussian distributions are also popular. Both distributions give virtually the same performance which depends on the other parameters discussed here. The discrete bi-valued distribution tends to give a slightly less rich signal space (there is a non-zero probability of identical neurons), but might make analysis of what is happening in the reservoir easier. The width of the distributions does not matter, as it is reset in a way explained in the next section.

The input matrix $\mathbf{W}^{\text{in}}$ is usually generated according to the same type of distribution as $\mathbf{W}$, but typically dense.

3.2.4 Spectral Radius One of the most central global parameters of an ESN is spectral radius of the reservoir connection matrix $\mathbf{W}$, i.e., the maximal absolute eigenvalue of this matrix. It scales the matrix $\mathbf{W}$, or viewed alternatively, scales the width of the distribution of its nonzero elements.

Typically a random sparse $\mathbf{W}$ is generated; its spectral radius $\rho(\mathbf{W})$ is computed; then $\mathbf{W}$ is divided by $\rho(\mathbf{W})$ to yield a matrix with a unit spectral radius; this is then conveniently scaled with the ultimate spectral radius to be determined in a tuning procedure.

For the ESN approach to work, the reservoir should satisfy the so-called echo state property: the state of the reservoir $\mathbf{x}(n)$ should be uniquely defined by the fading history of the input $\mathbf{u}(n)$ [6]. In other words, for a long enough input $\mathbf{u}(n)$, the reservoir state $\mathbf{x}(n)$ should not depend on the initial conditions that were before the input.

Large $\rho(\mathbf{W})$ values can lead to reservoirs hosting multiple fixed point, periodic, or even chaotic (when sufficient nonlinearity in the reservoir is reached) spontaneous attractor modes, violating the echo state property.

> $\rho(\mathbf{W}) < 1$ ensures echo state property in most situations.

Even though it is possible to violate the echo state property even with $\rho(\mathbf{W}) < 1$, this is unlikely to happen in practice. More importantly, the echo state property often holds for $\rho(\mathbf{W}) \geq 1$ for nonzero inputs $\mathbf{u}(n)$. This can be explained by the strong $\mathbf{u}(n)$ pushing activations of the neurons away from 0 where their tanh() nonlinearities have a unitary slope to regions where this slope is smaller, thus reducing the gains of the neurons and the effective strength of feedback connections. Intuitively speaking, due to activation-squashing nonlinearities, strong inputs "squeeze out" the autonomous activity from the reservoir activations. This means, that for nonzero $\mathbf{u}(n)$ $\rho(\mathbf{W}) < 1$ is not a necessary condition for the echo state property and optimal $\rho(\mathbf{W})$ values can sometimes be significantly greater than 1.

In practice $\rho(\mathbf{W})$ should be selected to maximize the performance, with the value 1 serving as an initial reference point.

As a guiding principle, $\rho(\mathbf{W})$ should be set greater for tasks where a more extensive history of the input is required to perform it, and smaller for tasks where the current output $\mathbf{y}(n)$ depends more on the recent history of $\mathbf{u}(n)$. The spectral radius determines how fast the influence of an input dies out in a reservoir with time, and how stable the reservoir activations are [24].

> The spectral radius should be greater in tasks requiring longer memory of the input.

3.2.5 Input Scaling The scaling of the input weight matrix $\mathbf{W}^{\text{in}}$ is another key parameter to optimize in an ESN. For uniformly distributed $\mathbf{W}^{\text{in}}$ we usually define the input scaling a as the range of the interval $[-a; a]$ from which values of $\mathbf{W}^{\text{in}}$ are sampled; for normal distributed input weights one may take the standard deviation as a scaling measure.

In order to have a small number of freely adjustable parameters, often all the columns of $\mathbf{W}^{\text{in}}$ are scaled together using a single scaling value. However, the scaling of the first column of $\mathbf{W}^{\text{in}}$ corresponding to the bias input to the reservoir units in (2) can be optimized separately from the rest. If the remaining "active" input channels contribute to the task in very different ways, it is also advised to optimize their scalings separately.

> Scale the whole $\mathbf{W}^{\text{in}}$ uniformly to have few global parameters in ESN. However, to increase the performance:
>
> – scale the first column of $\mathbf{W}^{\text{in}}$ (i.e., the bias inputs) separately;
> – scale other columns of $\mathbf{W}^{\text{in}}$ separately if channels of $\mathbf{u}(n)$ contribute differently to the task.

This varies the number of free global parameters to set for $\mathbf{W}^{\text{in}}$ from 1 up to $N_{\text{u}} + 1$.

It was suggested in the original ESN publications to scale and shift the input data, optimizing the magnitude of both. But the same effect can be achieved by scaling the input weights of the active inputs and the bias, respectively.

Still, input data normalization is advisable for ESNs just as for any other ML approach. This puts each learning task into a more standardized setting. It may be helpful to have the range of the input data values bounded. For example, apply the $\tanh(\cdot)$ squashing to $\mathbf{u}(n)$ if its distribution is unbounded. Otherwise the outliers can throw the reservoir state $\mathbf{x}(n)$ into some "unfamiliar" regions not well covered by the usual working trajectories of $\mathbf{x}(n)$ for which the global parameters have been optimized or the outputs learned. This can lead to a virtual loss of useful memory (due to saturations in the activation nonlinearities) or unpredictable outputs at these points, respectively.

> It is advisable to normalize the data and may help to keep the inputs $\mathbf{u}(n)$ bounded avoiding outliers (e.g., apply $\tanh(\cdot)$ to $\mathbf{u}(n)$ if it is unbounded).

Input scaling determines how nonlinear the reservoir responses are. For very linear tasks $\mathbf{W}^{\text{in}}$ should be small, letting units operate around the 0 point where their activation $\tanh(\cdot)$ is virtually linear. For large $\mathbf{W}^{\text{in}}$, the units will get easily saturated close to their 1 and -1 values, acting in a more nonlinear, binary switching manner. While $\rho(\mathbf{W})$ also affects the nonlinearity, the reservoir activations become unstable when increasing $\rho(\mathbf{W})$, as explained in Section 3.2.4, before it can make the reservoir highly nonlinear.

The amount of nonlinearity the task requires is not easy to judge. Finding a proper setting benefits from experience and intuitive insight into nonlinear dynamics. But also the masters of RC (if there are such) use trial and error to tune this characteristic.

Looking at (2), it is clear that the scaling of $\mathbf{W}^{\text{in}}$, together with the scaling of $\mathbf{W}$ (i.e., $\rho(\mathbf{W})$) determines the proportion of how much the current state $\mathbf{x}(n)$ depends on the current input $\mathbf{u}(n)$ and how much on the previous state $\mathbf{x}(n-1)$, respectively. The respective sizes N_{u} and N_{x} should also be taken into account.

The input scaling regulates:

- the amount of nonlinearity of the reservoir representation $\mathbf{x}(n)$ (also increasing with $\rho(\mathbf{W})$);
- the relative effect of the current input on $\mathbf{x}(n)$ as opposed to the history (in proportion to $\rho(\mathbf{W})$).

It has been empirically observed that the representation of the different principle components of $\mathbf{u}(n)$ in $\mathbf{x}(n)$ is roughly proportional to the square root of their magnitudes in $\mathbf{u}(n)$ [26]. In other words, the reservoir tends to flatten the spectrum of principal components of $\mathbf{u}(n)$ in $\mathbf{x}(n)$ – something to keep in mind when choosing the right representation or preprocessing of the data. For example, if smaller principal components carry no useful information it might be helpful to remove them from the data by Principal Component Analysis (PCA) before feeding them to a reservoir, otherwise they will get relatively amplified there.

3.2.6 Leaking Rate The leaking rate α of the reservoir nodes in (3) can be regarded as the speed of the reservoir update dynamics discretized in time. We can describe the reservoir update dynamics in continuous time as an Ordinary Differential Equation (ODE)

$$\dot{\mathbf{x}} = -\mathbf{x} + \tanh\left(\mathbf{W}^{\text{in}}[1;\mathbf{u}] + \mathbf{W}\mathbf{x}\right). \tag{5}$$

If we make an Euler's discretization of this ODE (5) in time, taking

$$\frac{\Delta\mathbf{x}}{\Delta t} = \frac{\mathbf{x}(n+1) - \mathbf{x}(n)}{\Delta t} \approx \dot{\mathbf{x}}, \tag{6}$$

we arrive at exactly (up to some time indexing conventions) the discrete time equations (2)(3) with α taking the place of the sampling interval Δt. Thus α can be regarded as the time interval in the continuous world between two consecutive time steps in the discrete realization. Also, empirically the effect of setting α is comparable to that of re-sampling $\mathbf{u}(n)$ and $\mathbf{y}^{\text{target}}(n)$ when the signals are slow [27,28]. The leaking rate α can even be adapted online to deal with time wrapping of the signals [27,12]. Equivalently, α can be introduced as a time constant in (5), if keeping $\Delta t \equiv 1$.

While there are some slight variations alternative to those in (3), of how to do leaky integration (e.g., [12]), the version (3) has emerged as preferred, because it guarantees that $\mathbf{x}(n)$ never goes outside the $(-1, 1)$ interval.

Set the leaking rate α in (3) to match the speed of the dynamics of $\mathbf{u}(n)$ and/or $\mathbf{y}^{\text{target}}(n)$.

This can, again, be difficult and subjective to determine in some cases. Especially when the timescales of $\mathbf{u}(n)$ and $\mathbf{y}^{\text{target}}(n)$ are quite different. This is one more of the global parameters to be tuned by trial and error.

When the task requires modeling the time series producing dynamical system on multiple time scales, it might be useful to set different leaking rates to different units (making α a vector $\alpha \in \mathbb{R}^{N_{\text{x}}}$) [29], with a possible downside of having more parameters to optimize.

Alternatively, the leaky integration (3) can be seen as a simple digital low-pass filter, also known as exponential smoothing, applied to every node. Some contributions even suggest applying more powerful filters for this purpose [30,31].

In some cases setting a small α, and thus inducing slow dynamics of $\mathbf{x}(n)$, can dramatically increase the duration of the short-term memory in ESN [22].

3.3 Practical Approach to Reservoir Production

3.3.1 Prioritizing Parameters While all the ESN reservoir parameters discussed in Section 3.2 have their guiding intuitions in setting them, fixing some of them is more straightforward than others.

> The main three parameters to optimize in an ESN reservoir are:
>
> – input scaling(-s);
> – spectral radius;
> – leaking rate(-s).

These three parameters, discussed in Sections 3.2.5, 3.2.4, and 3.2.6 respectively, are very important for a good performance and are quite task-specific.

The reservoir size N_x almost comes as an external restriction (Section 3.2.1), and the rest of the parameters can be set to reasonable default values: reservoir sparseness (Section 3.2.2), weight distribution (Section 3.2.3), or details of the model (2)(3). It is still worth investigating several options for them, as a lower priority.

As explained before, the performance can also be additionally improved in many cases by "splitting" a single parameter into several. Setting different scalings to the columns of $\mathbf{W}^{\text{in}}$ (corresponding to the bias input and possibly to different dimensions of input if they are of different nature) can go a long way. Also, setting leaking rates α differently for different units (e.g., by splitting them to several sub-populations with constant value) can help a lot in multi-timescale tasks.

3.3.2 Setup for Parameter Selection One of the main advantages of ESNs is that learning the outputs is fast. This should be exploited in evaluating how good a reservoir generated by a particular set of parameters is.

> The most pragmatic way to evaluate a reservoir is to train the output (4) and measure its error.

Either validation or training error can be used. Validation is, of course, preferred if there is a danger of overfitting. Training error has the advantage of using less data and in some cases no need to rerun a trained network with it. If a validation data set is necessary for the output training (as explained in Section 4), the error on it might be utilized with no additional cost, as a compromise between the training and a yet separate second validation set.

If training of the output and validation is not fast enough, smaller initial reservoirs (as stated in Section 3.2.1), or a reduced representative data set can be used. For the same reason it is often an overkill to use a k-fold cross-validation in global parameter optimization, at least in initial stages, unless the data are really scarce.

It is important to keep in mind that the randomly generated reservoirs even with the same parameters vary slightly in their performance. This variance is ever present but is typically more pronounced with smaller reservoirs than with bigger ones; the random variations inside of a big reservoir tend to "average out". It is nonetheless important to keep this random fluctuation of performance separate from the one caused by different parameter values.

> To eliminate the random fluctuation of performance, keep the random seed fixed and/or average over several reservoir samples.

Fixing a random seed in the programming environment before generating the reservoirs makes the random aspect of the reservoirs identical across trials and thus the experiments deterministically repeatable. Using a single reservoir is faster, but with an obvious danger of below-average performance and/or overfitting the parameters to a particular instance of a randomly generated reservoir: good parameters might not carry over well to a different software implementation, or, e.g., different size of a reservoir.

3.3.3 Manual Parameter Selection Manual selection of parameters is unavoidable to some extent in virtually all ML approaches. Even when parameters are learned or selected through automated search, it is typically necessary to set meta-parameters (or rather "meta-meta-parameters") for these procedures.

> When manually tuning the reservoir parameters, change one parameter at a time.

Changes in several parameters at once often have opposing effects on performance, but it is impossible to tell which contributed what. A reasonable approach is to set a single parameter to a well enough value before starting changing another one, and repeating this until the performance is satisfactory.

It is also advisable to take notes or log the performance automatically for extended optimizations, in order not to "go in circles" when repeating the same parameter values.

An empirical direction of a gradient can be estimated for a parameter, making a small change to it and observing the change in performance. However, the error landscapes are often non-convex and trying distant values of the parameters can sometimes lead to dramatic improvements.

> Always plot samples of reservoir activation signals $\mathbf{x}(n)$ to have a feeling of what is happening inside the reservoir.

This may reveal that $\mathbf{x}(n)$ are over-saturated, under-activated, exhibiting autonomous cyclic or chaotic behavior, etc. Overall, plotting information additional to the error rate helps a lot in gaining more insight into how the parameters should be changed.

Typically, good average performance is not found in a very narrow parameter range, thus a very detailed fine-tuning of parameters does not give a significant improvement and is not necessary.

3.3.4 Automated Parameter Selection Since manual parameter optimization might quickly get tedious, automated approaches are often preferred.

Since ESNs have only a few parameters requiring more careful tuning, *grid search* is probably the most straightforward option. It is easy enough to implement with a few nested loops, and high-level ML programming libraries, such as Oger (mentioned in Section 6) in the case of RC, often have ready-made routines for this.

A reasonable approach is to do a coarser grid search over wider parameter intervals to identify promising regions and then do a finer search (smaller steps) in these regions. As mentioned, typically the grid must not be very dense to achieve a good performance.

The best performance being on a boundary of a covered grid is a good indication that the optimal performance might be outside the grid.

In general, meta-parameter or hyper-parameter optimization is a very common topic in many branches of ML and beyond. There are numerous generic optimization methods applicable to this task described in the literature. They are often coping with much larger search spaces than a grid search is effectively capable of, such as random search, or more sophisticated methods trying to model the error landscape (see, e.g., [32]). They are in principle just as well applicable to ESNs with a possibility of also including in the optimization the parameters of second importance.

There is also a way to optimize the global parameters of the reservoir through a gradient descent [12]. It has, however, not been widely applied in the literature.

3.4 Pointers to Reservoir Extensions

There are also alternative ways of generating and adapting reservoirs suggested in the literature, including deterministic, e.g., [33], and data-specific, e.g., [34], ones. With a variety of such methods the modern field of RC has evolved from using the initial paradigm of a fixed reservoir and only training a readout from it, to also adapting the reservoir but differently from the readout, using generic, unsupervised, or even supervised methods. In some cases a hardware system is used as a

reservoir and thus is predetermined by its specific features. See [17] and updated in Chapter 2 of [35] for a classification and overview. The classical ESN approach described here, however, still holds its ground for its simplicity and performance.

4 Training Readouts

4.1 Ridge Regression

Since readouts from an ESN are typically linear and feed-forward, the Equation (4) can be written in a matrix notation as

$$\mathbf{Y} = \mathbf{W}^{\text{out}}\mathbf{X}, \tag{7}$$

where $\mathbf{Y} \in \mathbb{R}^{N_\text{y} \times T}$ are all $\mathbf{y}(n)$ and $\mathbf{X} \in \mathbb{R}^{(1+N_\text{u}+N_\text{x}) \times T}$ are all $[1; \mathbf{u}(n); \mathbf{x}(n)]$ produced by presenting the reservoir with $\mathbf{u}(n)$, both collected into respective matrices by concatenating the column-vectors horizontally over the training period $n = 1, \ldots, T$. We use here a single $\mathbf{X}$ instead of $[\mathbf{1}; \mathbf{U}; \mathbf{X}]$ for notational brevity.

Finding the optimal weights $\mathbf{W}^{\text{out}}$ that minimize the squared error between $\mathbf{y}(n)$ and $\mathbf{y}^{\text{target}}(n)$ amounts to solving a typically overdetermined system of linear equations

$$\mathbf{Y}^{\text{target}} = \mathbf{W}^{\text{out}}\mathbf{X}, \tag{8}$$

where $\mathbf{Y}^{\text{target}} \in \mathbb{R}^{N_\text{y} \times T}$ are all $\mathbf{y}(n)$, with respect to $\mathbf{W}^{\text{out}}$ in a least-square sense – i.e., a case of linear regression. In this context $\mathbf{X}$ can be called the *design matrix*. The system is overdetermined, because typically $T \gg 1 + N_\text{u} + N_\text{x}$.

There are standard well-known ways to solve (8), we will discuss a couple of good choices here.

Probably the most universal and stable solution to (8) in this context is ridge regression, also known as regression with Tikhonov regularization:

$$\mathbf{W}^{\text{out}} = \mathbf{Y}^{\text{target}}\mathbf{X}^\text{T}\left(\mathbf{X}\mathbf{X}^\text{T} + \beta\mathbf{I}\right)^{-1}, \tag{9}$$

where β is a regularization coefficient explained in Section 4.2, and $\mathbf{I}$ is the identity matrix.

> The most generally recommended way to learn linear output weights from an ESN is ridge regression (9)

We start with this method because it should be the first choice, even though it is not the most trivial one. We will explain different aspects of this method in the coming sections together with reasons for why it should be preferred and alternatives that in some cases can be advantageous.

4.2 Regularization

To assess the quality of the solution produced by training, it is advisable to monitor the actual obtained output weights $\mathbf{W}^{\text{out}}$. Large weights indicate that $\mathbf{W}^{\text{out}}$ exploits and amplifies tiny differences among the dimensions of $\mathbf{x}(n)$, and can be very sensitive to deviations from the exact conditions in which the network has been trained. This is a big problem in the setups where the network receives its output as the next input. The slight deviation of the output from the expected value quickly escalates in subsequent time steps. Ways of dealing with such setup are explained in Section 5.

> Extremely large $\mathbf{W}^{\text{out}}$ values may be an indication of a very sensitive and unstable solution.

To counteract this effect is exactly what the regularization part $\beta\mathbf{I}$ in the ridge regression (9) is for. Instead of just minimizing RMSE (1), ridge regression (9) solves

$$\mathbf{W}^{\text{out}} = \arg\min_{\mathbf{W}^{\text{out}}} \frac{1}{N_{\text{y}}} \sum_{i=1}^{N_{\text{y}}} \left(\sum_{n=1}^{T} \left(y_i(n) - y_i^{\text{target}}(n) \right)^2 + \beta \left\| \mathbf{w}_i^{\text{out}} \right\|^2 \right), \tag{10}$$

where $\mathbf{w}_i^{\text{out}}$ is the ith row of $\mathbf{W}^{\text{out}}$ and $\|\cdot\|$ stands for the Euclidean norm. The objective function in (10) adds a regularization, or weight decay, term $\beta \left\| \mathbf{w}_i^{\text{out}} \right\|^2$ penalizing large sizes of $\mathbf{W}^{\text{out}}$ to the square error between $\mathbf{y}(n)$ and $\mathbf{y}^{\text{target}}(n)$. This is a sum of two objectives, a compromise between having a small training error and small output weights. The relative "importance" between these two objectives is controlled by the regularization parameter β.

> Use regularization (e.g., (9)) whenever there is a danger of overfitting or feedback instability.

In (9) the optimal regularization coefficient β depends on the concrete instantiation of the ESN. It should be selected individually for a concrete reservoir based on validation data.

> Select β for a concrete ESN using validation, without rerunning the reservoir through the training data.

There is no need to rerun the model through the data with every value β, because none of the other variables in (9) are affected by its changes. Memory permitting, there is also no need to rerun the model with the (small) validation dataset, if you can store $\mathbf{X}$ for it and compute the validation output by (7). This makes testing β values computationally much less expensive than testing the reservoir parameters explained in Section 3.

The optimal values of β can vary by many magnitudes of size, depending on the exact instance of the reservoir and length of the training data. If doing a simple exhaustive search, it is advisable to search on a logarithmic grid.

Setting β to zero removes the regularization: the objective function in (10) becomes equivalent to RMSE (1), making the ridge regression a generalization of a regular linear regression. The solution (9) with $\beta = 0$ becomes

$$\mathbf{W}^{\text{out}} = \mathbf{Y}^{\text{target}} \mathbf{X}^{\text{T}} \left(\mathbf{X}\mathbf{X}^{\text{T}} \right)^{-1}, \tag{11}$$

known as normal equations method for solving linear regression (8). In practice, however, setting $\beta = 0$ often leads to numerical instabilities when inverting $(\mathbf{X}\mathbf{X}^{\text{T}})$ in (11). This too recommends using a logarithmic scale for selecting β where it never goes to zero. The problem can in some cases also be alleviated by using a pseudoinverse instead of the real inverse in (11).

A Gaussian process interpretation of the linear readout gives an alternative criterion for setting β directly [36].

A similar regularization effect to Tikhonov (9) can be achieved by adding scaled white noise to $\mathbf{x}(n)$ in (3) – a method that predates ridge regression in ESNs [6]. Like in ridge regression, i.i.d. noise emphasizes the diagonal of $(\mathbf{X}\mathbf{X}^{\text{T}})$. The advantage is that it is also propagated through $\mathbf{W}$ in (2), modeling better the effects of noisy signals in the reservoir. The output learns to recover from perturbed signals, making the model more stable with feedback loops (Section 5). The downside of this noise immunization is that the model needs to be rerun with each value of the noise scaling.

4.3 Large Datasets

Ridge regression (9) (or the Wiener-Hopf solution (11) as a special case) also allows a one-shot training with virtually unlimited amounts of data.

Notice that the dimensions of the matrices $(\mathbf{Y}^{\text{target}}\mathbf{X}^{\text{T}}) \in \mathbb{R}^{N_{\text{y}} \times N_{\text{x}}}$ and $(\mathbf{X}\mathbf{X}^{\text{T}}) \in \mathbb{R}^{N_{\text{x}} \times N_{\text{x}}}$ do not depend on the length T of the training sequence in their sizes. The two matrices can be updated by simply adding the corresponding results from the newly incoming data. This one-shot training approach in principle works with an unlimited amount of data – neither complexity of working memory, nor time of the training procedure (9) itself depend on the length of data T.

With large datasets collect the matrices $\left(\mathbf{Y}^{\text{target}}\mathbf{X}^{\text{T}}\right)$ and $\left(\mathbf{X}\mathbf{X}^{\text{T}}\right)$ incrementally for (9).

The eventual limitation of the straightforward summation comes from the finite precision of floating point numbers – adding large (like in the so-far accumulated matrix) and small (like the next update) numbers becomes inaccurate. A better summation scheme, such as a hierarchical multi-stage summation where the two added values are always of similar magnitude (e.g., coming from the same amount of time steps), or Kahan summation [37] that compensates for the accumulating errors, should be used instead.

With very large datasets, a more accurate summation scheme should be used for accumulating $\left(\mathbf{Y}^{\text{target}}\mathbf{X}^{\text{T}}\right)$ and $\left(\mathbf{X}\mathbf{X}^{\text{T}}\right)$.

Using extended precision numbers here could also help, as well as in other calculations.

4.4 Direct Pseudoinverse Solution

A straightforward solution to (8) is

$$\mathbf{W}^{\text{out}} = \mathbf{Y}^{\text{target}}\mathbf{X}^{+}, \tag{12}$$

where $\mathbf{X}^{+}$ is the Moore-Penrose pseudoinverse of $\mathbf{X}$. If $(\mathbf{X}\mathbf{X}^{\text{T}})$ is invertible the (12) in essence becomes equivalent to (11), but works even when it is not. The direct pseudoinverse calculation typically exhibits high numerical stability. As a downside, it is expensive memory-wise for large design matrices $\mathbf{X}$, thereby limiting the size of the reservoir N_{x} and/or the number of training samples T. Since there is virtually no regularization, the system of linear equations (8) should be well overdetermined, i.e., $1 + N_{\text{u}} + N_{\text{x}} \ll T$. In other words, the task should be difficult relatively to the capacity of the reservoir so that overfitting does not happen.

Use direct pseudoinverse (12) to train ESNs with high precision and little regularization when memory and run time permit.

Many modern programming libraries dealing with linear algebra have implementations of a matrix pseudoinverse, which can be used "off the shelf". However, implementations vary in their precision, computational efficiency, and numerical stability.

For high precision tasks, check whether the regression $(\mathbf{Y}^{\text{target}} - \mathbf{W}^{\text{out}}\mathbf{X})\mathbf{X}^{+}$ on the error $\mathbf{Y}^{\text{target}} - \mathbf{W}^{\text{out}}\mathbf{X}$ is actually all $= \mathbf{0}$, and add it to $\mathbf{W}^{\text{out}}$ if it is not.

This computational trick should not work in theory (the regression on the error should be equal to zero), but sometimes does work in practice in Matlab [38], possibly because of some internal optimizations.

Some high level linear algebra libraries have ready-made subroutines for doing regression, i.e., solving linear least-squares as in (8), where the exact methods are not made explicit and can internally be chosen depending on the conditioning of the problem. The use of them has an obvious disadvantage of lacking the control on the issues discussed here.

A powerful extension of the basic ESN approach is training (very) many (very small) ESNs in parallel and averaging their outputs, which in some cases has drastically improved performance [11,12]. This might be not true for tasks requiring large memory, where one bigger reservoir may still be better than several smaller ones.

Averaging outputs from multiple reservoirs increases the performance.

4.5 Initial Transient

Usually $\mathbf{x}(n)$ data from the beginning of the training run are discarded (i.e., not used for learning $\mathbf{W}^{\text{out}}$) since they are contaminated by initial transients. To keep notation simple let us assume they come before $n = 1$.

> For long sequences discard the initial time steps of activations $\mathbf{x}(n)$ for training that are affected by initial transient.

The initial transient is in essence a result of an arbitrary setting of $\mathbf{x}(0)$, which is typically $\mathbf{x}(0) = \mathbf{0}$. This introduces an unnatural starting state which is not normally visited once the network has "warmed up" to the task. The amount of time steps to discard depends on the memory of the network (which in turn depends on reservoir parameters), and typically are in the order of tens or hundreds.

However, if the data consists of multiple short separate sequences (like in sequence classification), "the initial transient" might be the usual working mode of the ESN. In this case discarding the precious (possibly all!) data might be disadvantageous. See Section 4.7 for more on this. Note, that you would want to reset the state to some initial (the same) value before each sequence to make the classification of sequences independent.

With multiple sequences of data the time steps on which the learning should be performed should be concatenated in $\mathbf{X}$ and $\mathbf{Y}^{\text{target}}$, the same way as there would only be a single long sequence.

A generalization of discarding data is presented in the next Section 4.6.

4.6 Regression Weighting

In the regression learning of ESNs it is easy to make some time steps count more than others by weighting the minimized square error differently. For this, the time steps of the square error are weighted with a weight vector $s(n) \in \mathbb{R}$:

$$E(\mathbf{y}, \mathbf{y}^{\text{target}}) = \frac{1}{N_{\text{y}}} \sum_{i=1}^{N_{\text{y}}} \sum_{n=1}^{T} s(n) \left(y_i(n) - y_i^{\text{target}}(n)\right)^2. \tag{13}$$

This error is minimized with the same learning algorithms, but at each time step n the vectors $[1; \mathbf{u}(n); \mathbf{x}(n)]$ and $\mathbf{y}^{\text{target}}(n)$ are element-wise multiplied with $\sqrt{s(n)}$ before collecting them into $\mathbf{X}$ and $\mathbf{Y}^{\text{target}}$. Higher values of $s(n)$ put more emphasis on minimizing the error between $\mathbf{y}(n)$ and $\mathbf{y}^{\text{target}}(n)$. Putting a weight $s(n)$ on a time step n has the same effect as if $[1; \mathbf{u}(n); \mathbf{x}(n)]$ and $\mathbf{y}^{\text{target}}(n)$ have appeared $s(n)$ times in the training with a regular weight $s(n) = 1$. Setting $s(n) = 0$ is equivalent to discarding the time steps from training altogether.

> Use weighting to assign different importance to different time steps when training.

This weighted least squares scheme can be useful in different situations like discarding or weighting down the signals affected by the initial transients, corrupted or missing data, emphasizing the ending of the signal [39], etc. It is fully compatible with ridge regression (9) where the weighting (13) is applied to the square error part of the objective function in (10).

For an even more refined version different channels of $\mathbf{y}(n)$ can be trained separately and with different $s(n)$. The weighting can for example be used to counter an imbalance between positive vs. negative samples in a classification or detection task.

4.7 Readouts for Classification

When the task is to classify separate short time series, the training is typically set up such that the output $\mathbf{y}(n)$ has a dimension for every class and $\mathbf{y}^{\text{target}}(n)$ is equal to one in the dimension

corresponding to the correct class and zero everywhere else. The model is trained to approximate $\mathbf{y}^{\text{target}}(n)$ and the class for single sequence $\mathbf{u}(n)$ is very often decided by

$$\text{class}\left(\mathbf{u}(n)\right) = \arg\max_k \left(\frac{1}{|\tau|} \sum_{n \in \tau} y_k(n) \right) = \arg\max_k \left((\Sigma \mathbf{y})_k \right), \tag{14}$$

where $y_k(n)$ is the kth dimension of $\mathbf{y}(n)$ produced by ESN from $\mathbf{u}(n)$, τ is some integration interval (can be the length of the whole sequence $\mathbf{u}(n)$), and $\Sigma\mathbf{y}$ stands for a shorthand notation of $\mathbf{y}(n)$ time-averaged over τ.

There is a better way to do this. Notice, that in this case

$$\Sigma\mathbf{y} = \frac{1}{|\tau|} \sum_{n \in \tau} \mathbf{y}(n) = \frac{1}{|\tau|} \sum_{n \in \tau} \mathbf{W}^{\text{out}}[1; \mathbf{u}(n); \mathbf{x}(n)] = \tag{15}$$

$$= \mathbf{W}^{\text{out}} \frac{1}{|\tau|} \sum_{n \in \tau} [1; \mathbf{u}(n); \mathbf{x}(n)] = \mathbf{W}^{\text{out}} \Sigma\mathbf{x}, \tag{16}$$

where $\Sigma\mathbf{x}$ is a shorthand for $[1; \mathbf{u}(n); \mathbf{x}(n)]$ time-averaged over τ. The form (16) is a more efficient way to compute $\Sigma\mathbf{y}$, since there is only one multiplication with $\mathbf{W}^{\text{out}}$.

More importantly, (16) can be used to make training more efficient and powerful. For a given short sequence $\mathbf{u}(n)$, instead of finding $\mathbf{W}^{\text{out}}$ that minimizes $E\left(\mathbf{y}^{\text{target}}(n), \mathbf{y}(n)\right)$ for every $n \in \tau$, it is better to find the one that minimizes the error between the time-averaged values $E(\mathbf{y}^{\text{target}}, \Sigma\mathbf{y})$. In this case $\mathbf{y}(n)$ (which is not actually explicitly computed) is allowed to deviate from $\mathbf{y}^{\text{target}}(n)$ as long as the time-averaged $\Sigma\mathbf{y}$ is close to $\mathbf{y}^{\text{target}}$. Here $\mathbf{y}^{\text{target}} \equiv \Sigma\mathbf{y}^{\text{target}} = \mathbf{y}^{\text{target}}(n) = \text{const}$ for a single short sequence.

> To classify sequences, train and use readouts from time-averaged activations $\Sigma\mathbf{x}$ (16), instead of $\mathbf{x}(n)$.

Note that weighting is still possible both among the short sequences and inside each sequence over the intervals τ, using weighted average instead of a simple one. Actually, weighting over τ is often recommendable, emphasizing the ending of the sequence where the whole information to make the classification decision has been fed into the reservoir.

To retain information from different times in the short sequence, weighted averages $\Sigma_1\mathbf{x}, \ldots, \Sigma_k\mathbf{x}$ over several time intervals $\tau_1, \ldots, \tau_k$ during the short sequence can be computed and concatenated into an extended state $\Sigma_*\mathbf{x} = [\Sigma_1\mathbf{x}; \ldots; \Sigma_k\mathbf{x}]$. This extended state $\Sigma_*\mathbf{x}$ can be used instead of $\Sigma\mathbf{x}$ for an even more powerful classification. In this case $\mathbf{W}^{\text{out}}$ is also extended to $\mathbf{W}^{\text{out}}_* \in \mathbb{R}^{N_\text{y} \times k \cdot (1+N_\text{u}+N_\text{x})}$.

> Concatenate weighted time-averages over different intervals to read out from for an even more powerful classification.

Since the short sequences are typically of different lengths (the advantage of using temporal classification methods), the intervals $\tau_1, \ldots, \tau_k$ should be scaled to match the length of each sequence.

The techniques so far described in this section effectively reduce the time series classification to a static data classification problem by reducing the variable-length inputs $\mathbf{u}(n)$ to fixed-size feature vectors $\Sigma_*\mathbf{x} \in \mathbb{R}^{k \cdot (1+N_\text{u}+N_\text{x})}$. There are many powerful machine learning methods available to solve the static classification problem that can be employed at this point, such as logistic regression or maximum margin classifiers. See, e.g., [40] for different options. These methods define the error function differently and offer different, mostly iterative, optimization algorithms.

> Different powerful classification methods for static data can be employed as the readout from the time-averaged activations $\Sigma_*\mathbf{x}$.

Among others, the same regression methods as for temporal data can be used to train a linear readout $\mathbf{y} = \mathbf{W}_*^{\text{out}} \Sigma_* \mathbf{x}$ and decide the class by maximum as in (14). In this case, for every short sequence $\mathbf{u}(n)$ only one pair of vectors $\mathbf{y}^{\text{target}}$ and $\Sigma_* \mathbf{x}$ is collected into $\mathbf{Y}^{\text{target}}$ and $\mathbf{X}$ respectively for training by (9) or (12). Since this reduces training data points from the total number of time steps to the number of short sequences, precautions against overfitting should be taken. Such regression training for classification has an advantage of the single-shot closed form solution, however, it is not optimal because it does not directly optimize the correct classification rates.

For temporal pattern recognition tasks in a long sequence (i.e., detection plus classification), $\mathbf{y}^{\text{target}}(n)$ should be designed cleverly. The shapes, durations, and delays of the signals in $\mathbf{y}^{\text{target}}(n)$ indicating patterns in $\mathbf{u}(n)$ are also parameters that have to be optimized; as well as algorithms producing the final recognition (in the form of discrete symbol sequences or annotations) from the continuous signals $\mathbf{y}(n)$. But this goes beyond the scope of this paper. Alternatively, dynamic programming methods (such as Viterbi algorithm) can be used for trainable recognizers at the output layer, see [41].

4.8 Online Learning

Some applications require online model adaptation, e.g., [11]. In such cases the process generating the data is often not assumed to be stationary and is tracked by the constantly adapting model. $\mathbf{W}^{\text{out}}$ here acts as an adaptive linear combiner.

The simplest way to train $\mathbf{W}^{\text{out}}$ is the method known as the *Least Mean Squares* (LMS) algorithm [42], it has many extensions and modifications. It is a stochastic gradient descent algorithm which at every time step n changes $\mathbf{W}^{\text{out}}$ in the direction of minimizing the instantaneous squared error $\|\mathbf{y}^{\text{target}}(n) - \mathbf{y}(n)\|^2$. LMS is a first-order gradient descent method, locally approximating the error surface with a hyperplane. This approximation is poor then curvature of the error surface is very different in different directions, which is signified by large eigenvalue spreads of $\mathbf{X}\mathbf{X}^{\text{T}}$. In such a situation the convergence performance of LMS is unfortunately severely impaired.

An alternative linear readout learning to LMS, known in linear signal processing as the *Recursive Least Squares* (RLS) algorithm, is insensitive to the detrimental effects of eigenvalue spread and boasts a much faster convergence. It explicitly at each time step n minimizes a square error that is exponentially discounted going back in time:

$$E(\mathbf{y}, \mathbf{y}^{\text{target}}, n) = \frac{1}{N_{\text{y}}} \sum_{i=1}^{N_{\text{y}}} \sum_{j=1}^{n} \lambda^{n-j} \left(y_i(j) - y_i^{\text{target}}(j) \right)^2, \tag{17}$$

where $0 < \lambda \leq 1$ is the error "forgetting" parameter. This weighting is not unlike the one discussed in Section 4.6 where $s^{(n)}(j) = \lambda^{n-j}$ at time step n. RLS can be seen as a method for minimizing (17) at each time step n similar to Wiener-Hopf (11), but optimized by keeping and updating the estimate of $(\mathbf{X}\mathbf{X}^{\text{T}})^{-1}$ from time $n-1$ instead of recomputing it from scratch. The downside of RLS is it being computationally more expensive (quadratic in number of weights instead of linear like LMS) and notorious for numerical stability issues. Demonstrations of RLS for ESNs are presented in [11,43]. A careful and comprehensive comparison of variants of RLS as ESN readouts is carried out in a Master's thesis [44], which may be helpful for practitioners.

The BackPropagation-DeCorrelation (BPDC) [45] and FORCE [46] learning algorithms discussed in Section 5.3 are two other powerful methods for online training of single-layer readouts with feedback connections from the reservoirs.

4.9 Pointers to Readouts Extensions

There are also alternative ways of training outputs from the reservoirs suggested in the literature, e.g., Gaussian process [36], copula [47], or Support Vector Machine [48] style outputs. See [17] and updated in Chapter 2 of [35] for an overview.

5 Dealing with Output Feedbacks

5.1 Output Feedbacks

Even if the reservoir is kept fixed, for some tasks the trained readouts are fed back to the reservoir and thus the training process changes its dynamics. In other words, a recurrence exists between the reservoir and the trained readout. Pattern generation is a typical example of such task. This can be realized in two ways. Either by feedback connections $\mathbf{W}^{\text{fb}} \in \mathbb{R}^{N_\text{x} \times N_\text{y}}$ from the output to the reservoir, replacing (2) with

$$\tilde{\mathbf{x}}(n) = \tanh\left(\mathbf{W}^{\text{in}}[1; \mathbf{u}(n)] + \mathbf{W}\mathbf{x}(n-1) + \mathbf{W}^{\text{fb}}\mathbf{y}(n-1)\right), \tag{18}$$

or by looping the output $\mathbf{y}(n-1)$ as an input $\mathbf{u}(n)$ for the next update step n in (2), in effect turning a trained one step predictor ESN into a pattern generator. Note that these two options are equivalent and are just a matter of notation: $\mathbf{u}(n)$ and $\mathbf{W}^{\text{in}}$ instead of $\mathbf{y}(n-1)$ and $\mathbf{W}^{\text{fb}}$, respectively. The same principles thus apply to producing $\mathbf{W}^{\text{fb}}$ as to $\mathbf{W}^{\text{in}}$. In some cases, however, both external input and output feedback can be present.

This extends the power of RC, because it no longer relies on fixed random input-driven dynamics to construct the output, but the dynamics are adapted to the task. This power has its price, because stability issues arise here.

> Use output feedbacks to the reservoir only if they are necessary for the task.

This may include tasks that simply cannot be learned well enough without feedbacks. Feedbacks enable reservoirs to achieve universal computational capabilities [49] and can in practice be beneficial even where they are not an integral part of the task [38].

In order to avoid falling prey to the same difficulties as with full RNN training algorithms, two strategies are used in RC when learning outputs with feedbacks:

- Breaking the feedback loop during the training, Section 5.2;
- Adapting $\mathbf{W}^{\text{out}}$ online with specialized algorithms in the presence of real feedbacks, Section 5.3.

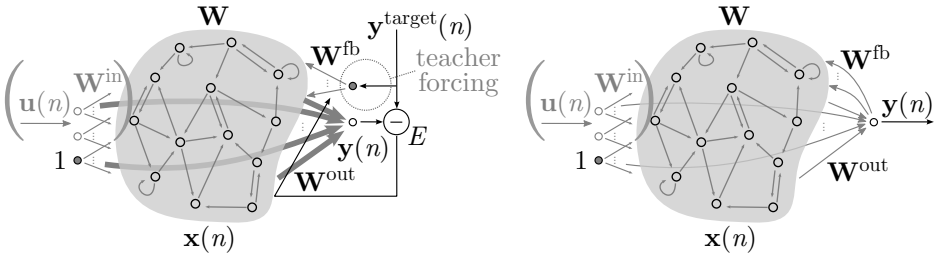
5.2 Teacher Forcing

The first strategy is to disengage the recurrent relationship between the reservoir and the readout using *teacher forcing* and treat output learning as a feedforward task. This is done by feeding the desired output $\mathbf{y}^{\text{target}}(n-1)$ through the feedback connections $\mathbf{W}^{\text{fb}}$ in (18) instead of the real output $\mathbf{y}(n-1)$ while learning (Figure 2a). The target signal $\mathbf{y}^{\text{target}}(n)$ "bootstraps" the learning process and if the output is learned with high precision (i.e., $\mathbf{y}(n) \approx \mathbf{y}^{\text{target}}(n)$), the recurrent system runs much in the same way with the real $\mathbf{y}(n)$ in feedbacks after training as it did with $\mathbf{y}^{\text{target}}(n)$ during training (Figure 2b). In a pure pattern generator setup with no additional inputs, $\mathbf{u}(n)$ and $\mathbf{W}^{\text{in}}$ may not be present at all – after training the ESN is run to autonomously generate a pattern, using teacher forcing initially to start the pattern. As noted before, this is equivalent to training a one time step predictor without feedbacks $\mathbf{W}^{\text{fb}}$ and looping its output $\mathbf{y}(n-1)$ as input $\mathbf{u}(n)$ through $\mathbf{W}^{\text{in}}$.

> For simple tasks, feed $\mathbf{y}^{\text{target}}(n)$ instead of $\mathbf{y}(n)$ in (18) while learning to break the recurrence.

This way $\mathbf{W}^{\text{out}}$ can be learned in the same efficient batch mode by linear regression, as explained before.

Teacher forcing applied to speedup error backpropagation RNN training is also discussed in chapter [50] of this book.



(a) Training with teacher forcing (b) Running in a generative mode

Fig. 2: An ESN with output feedbacks trained with teacher forcing.

There are some caveats here. The approach works very well if the output can be learned precisely [6]. However, if this is not the case, the distorted feedback leads to an even more distorted output and feedback at the next time step, and so on, with the actual generated output $\mathbf{y}(n)$ quickly diverging from the desired $\mathbf{y}^{\text{target}}(n)$. Even with well-learned outputs the dynamical stability of the autonomous running system is often an issue.

5.2.1 Stability with Feedbacks In both cases regularization of the output by ridge regression or/and noise immunization, as explained in Section 4.2, is the key to success.

> Regularization by ridge regression or noise is crucial to make teacher-forced feedbacks stable.

Some additional options to the ones in Section 4.2 are available here. One is adding scaled noise to the forced teacher signal $\mathbf{y}^{\text{target}}(n)$, emulating an imperfectly learned $\mathbf{y}^{\text{target}}(n)$ by $\mathbf{y}(n)$ and making the network robust to this. In fact, a readout can be trained to ignore some inputs or feedbacks altogether by feeding strong noise into them during training [51].

Another is doing training in several iterations and feeding back the signals that are in between the perfect $\mathbf{y}^{\text{target}}(n)$ and the actual $\mathbf{y}(n)$ obtained from the previous iteration. For example, a one time step prediction of the signal by the ESN (as opposed to running with real feedbacks) can be used as teacher forcer for the next iteration of training [11]. This way the model learns to recover from the directions of deviations from the correct signal that it actually produces, not from just random ones as in the case with noise; while at the same time the teacher signal does not diverge from the target too far.

Another recently proposed option is to also regularize the recurrent connections $\mathbf{W}$ themselves. A one-shot relearning of $\mathbf{W}$ with regularization (similar to ridge regression (9) for $\mathbf{W}^{\text{out}}$) to produce the same $\mathbf{x}(n)$, as the one from the initially randomly generated $\mathbf{W}$, reduces the recurrent connection strengths and helps making the ESN generator more stable [52,53].

5.3 Online Learning with Real Feedbacks

The second strategy to deal with output feedbacks in ESNs is using online (instead of one-shot) learning algorithms to train the outputs $\mathbf{W}^{\text{out}}$ while the feedbacks are enabled and feed back the (yet imperfectly) learned output, not the teacher signal. This way the model learns to stabilize itself in the real generative setting.

General purpose online learning algorithms, such as discussed in Section 4.8, can be used for this. However, there exist a couple of online RC learning algorithms that are specialized in training outputs with feedbacks, and in fact would not work without them.

BackPropagation-DeCorrelation (BPDC) [45] is such a highly optimized RC online learning algorithm which runs with a linear time complexity in the number of connections. The algorithm is said to be insensitive to reservoir settings and capable of tracking quickly changing signals. As a downside of the latter feature, the trained network forgets the previously seen data and is highly biased by the recent data. Some remedies for reducing this effect are reported in [54].

A recent RC approach named *FORCE* learning uses the RLS (Section 4.8) online learning algorithm to vigorously adapt $\mathbf{W}^{\text{out}}$ in the presence of the real feedbacks [46]. By the initial fast and strong adaptation of $\mathbf{W}^{\text{out}}$ the feedbacks $\mathbf{y}(n)$ are kept close to the desired $\mathbf{y}^{\text{target}}(n)$ already from the beginning of the learning process, similar to teacher forcing. The algorithm benefits from initial spontaneous chaotic activations inside the reservoir which are then subdued by the feedbacks. It appears that FORCE learning is well suited to yield very stable and accurate neural pattern generators.

6 Summary and implementations

We have presented many practical aspects for successfully applying ESNs. Some of them are not universal and should be filtered depending on a particular task. They are also not the only possible approaches, and can most likely be improved upon. They collect, however, the best practices accumulated in the field over the ten years from its start, and should serve well as guiding principles for ESN researchers and practitioners.

Implementing an ESN is relatively straightforward – minimalistic one-page self-contained code examples in several programming languages are available through http://reservoir-computing.org/software/minimal. There is also a number of ready-made and expandable software libraries available which incorporate many of the techniques described here. A collection of open source RC toolboxes in different programming languages and varying degrees of sophistication can be found at http://reservoir-computing.org/software. The most comprehensive of them is the Oger toolbox in Python http://reservoir-computing.org/oger.

The http://reservoir-computing.org website is an overall good hub of reservoir computing related resources, where new people can also register and contribute.

Acknowledgments

The author was supported through funds from the European FP7 projects ORGANIC and AMARSi. He would like to specially thank Herbert Jaeger for proof-reading this chapter and valuable suggestions, as well as the anonymous reviewers, and the whole reservoir computing community on whose collective wisdom this chapter is to some extent based.

References

1. David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. In *Neurocomputing: Foundations of research*, pages 673–695. MIT Press, Cambridge, MA, USA, 1988.
2. Yann LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Effiicient backprop. In *Neural Networks: Tricks of the Trade*, pages 9–50, 1996.
3. Paul J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.
4. Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1:270–280, 1989.
5. Kenji Doya. Bifurcations in the learning of recurrent neural networks. In *Proceedings of IEEE International Symposium on Circuits and Systems 1992*, volume 6, pages 2777–2780, 1992.
6. Herbert Jaeger. The "echo state" approach to analysing and training recurrent neural networks. Technical Report GMD Report 148, German National Research Center for Information Technology, 2001.
7. Herbert Jaeger. Echo state network. *Scholarpedia*, 2(9):2330, 2007.
8. Wolfgang Maass, Thomas Natschläger, and Henry Markram. Real-time computing without stable states: a new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531–2560, 2002.
9. Peter F. Dominey and Franck Ramus. Neural network processing of natural language: I. sensitivity to serial, temporal and abstract structure of language in the infant. *Language and Cognitive Processes*, 15(1):87–127, 2000.

10. Iulian Ilies, Herbert Jaeger, Olegas Kosuchinas, Monserrat Rincon, Vytenis Šakėnas, and Narūnas Vaškevičius. Stepping forward through echoes of the past: forecasting with echo state networks. Short report on the winning entry to the NN3 financial forecasting competition, available online at http://www.neural-forecasting-competition.com/downloads/NN3/methods/27-NN3_Herbert_Jaeger_report.pdf, 2007.
11. Herbert Jaeger and Harald Haas. Harnessing nonlinearity: predicting chaotic systems and saving energy in wireless communication. *Science*, 304(5667):78–80, 2004.
12. Herbert Jaeger, Mantas Lukoševičius, Dan Popovici, and Udo Siewert. Optimization and applications of echo state networks with leaky-integrator neurons. *Neural Networks*, 20(3):335–352, 2007.
13. Fabian Triefenbach, Azarakhsh Jalalvand, Benjamin Schrauwen, and Jean-Pierre Martens. Phoneme recognition with large hierarchical reservoirs. In *Advances in Neural Information Processing Systems 23 (NIPS 2010)*, pages 2307–2315. MIT Press, Cambridge, MA, 2011.
14. David Verstraeten, Benjamin Schrauwen, and Dirk Stroobandt. Reservoir-based techniques for speech recognition. In *Proceedings of the IEEE International Joint Conference on Neural Networks, 2006 (IJCNN 2006)*, pages 1050 – 1053, 2006.
15. David Verstraeten, Benjamin Schrauwen, Michiel D'Haene, and Dirk Stroobandt. An experimental unification of reservoir computing methods. *Neural Networks*, 20(3):391–403, 2007.
16. Mantas Lukoševičius, Herbert Jaeger, and Benjamin Schrauwen. Reservoir computing trends. *KI - Künstliche Intelligenz*, pages 1–7, 2012.
17. Mantas Lukoševičius and Herbert Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, August 2009.
18. James Martens and Ilya Sutskever. Learning recurrent neural networks with Hessian-free optimization. In *Proc. 28th Int. Conf. on Machine Learning*, 2011.
19. James Martens and Ilya Sutskever. Training deep and recurrent networks with Hessian-free optimization. In Klaus-Robert Müller, Grégoire Montavon, and Geneviève B. Orr, editors, *Neural Networks Tricks of the Trade, Reloaded*. Springer, 2012.
20. Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.
21. Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.
22. Herbert Jaeger. Long short-term memory in echo state networks: Details of a simulation study. Technical Report No. 27, Jacobs University Bremen, 2012.
23. Michiel Hermans and Benjamin Schrauwen. Recurrent kernel machines: Computing with infinite echo state networks. *Neural Computation*, 24(1):104–133, 2012.
24. David Verstraeten, Joni Dambre, Xavier Dutoit, and Benjamin Schrauwen. Memory versus non-linearity in reservoirs. In *Proc. Int Neural Networks (IJCNN) Joint Conf*, pages 1–8, 2010.
25. Herbert Jaeger. Short term memory in echo state networks. Technical Report GMD Report 152, German National Research Center for Information Technology, 2002.
26. Michiel Hermans and Benjamin Schrauwen. Memory in reservoirs for high dimensional input. In *Proceedings of the IEEE International Joint Conference on Neural Networks, 2010 (IJCNN 2010)*, pages 1–7, 2010.
27. Mantas Lukoševičius, Dan Popovici, Herbert Jaeger, and Udo Siewert. Time warping invariant echo state networks. Technical Report No. 2, Jacobs University Bremen, May 2006.
28. Benjamin Schrauwen, Jeroen Defour, David Verstraeten, and Jan M. Van Campenhout. The introduction of time-scales in reservoir computing, applied to isolated digits recognition. In *Proceedings of the 17th International Conference on Artificial Neural Networks (ICANN 2007)*, volume 4668 of *LNCS*, pages 471–479. Springer, 2007.
29. Udo Siewert and Welf Wustlich. Echo-state networks with band-pass neurons: towards generic time-scale-independent reservoir structures. Internal status report, PLANET intelligent systems GmbH, 2007. Available online at http://reslab.elis.ugent.be/node/112.
30. Francis wyffels, Benjamin Schrauwen, David Verstraeten, and Dirk Stroobandt. Band-pass reservoir computing. In Z. Hou and N. Zhang, editors, *Proceedings of the IEEE International Joint Conference on Neural Networks, 2008 (IJCNN 2008)*, pages 3204–3209, Hong Kong, 2008.
31. Gregor Holzmann and Helmut Hauser. Echo state networks with filter neurons and a delay and sum readout. *Neural Networks*, 23(2):244–256, 2010.
32. James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R.S. Zemel, P. Bartlett, F.C.N. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 23 (NIPS 2010)*, pages 2546–2554. 2011.
33. Ali Rodan and Peter Tiňo. Minimum complexity echo state network. *IEEE Transactions on Neural Networks*, 22(1):131–144, 2011.

34. Mustafa C. Ozturk, Dongming Xu, and José C. Príncipe. Analysis and design of echo state networks. *Neural Computation*, 19(1):111–138, 2007.
35. Mantas Lukoševičius. *Reservoir Computing and Self-Organized Neural Hierarchies*. PhD thesis, Jacobs University Bremen, Bremen, Germany, 2011.
36. Sotirios P. Chatzis and Yiannis Demiris. Echo state Gaussian process. *Neural Networks, IEEE Transactions on*, 22(9):1435 –1445, sept. 2011.
37. W. Kahan. Pracniques: further remarks on reducing truncation errors. *Communications of the ACM*, 8(1):40, January 1965.
38. Mantas Lukoševičius. Echo state networks with trained feedbacks. Technical Report No. 4, Jacobs University Bremen, February 2007.
39. Saulius Daukantas, Mantas Lukoševičius, Vaidotas Marozas, and Arūnas Lukoševičius. Comparison of "black box" and "gray box" methods for lost data reconstruction in multichannel signals. In *Proceedings of the 14th International Conference "Biomedical Engineering"*, pages 135–138, Kaunas, 2010.
40. Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
41. Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*. PhD thesis, Technical University Munich, Munich, Germany, 2008.
42. Behrouz Farhang-Boroujeny. *Adaptive Filters: Theory and Applications*. Wiley, 1998.
43. Herbert Jaeger. Adaptive nonlinear system identification with echo state networks. In *Advances in Neural Information Processing Systems 15 (NIPS 2002)*, pages 593–600. MIT Press, Cambridge, MA, 2003.
44. Ali U. Küçükemre. Echo state networks for adaptive filtering. Master's thesis, University of Applied Sciences Bohn-Rhein-Sieg, Germany, 2006. Available online at `http://reservoir-computing.org/publications/2006-echo-state-networks-adaptive-filtering`.
45. Jochen J. Steil. Backpropagation-decorrelation: recurrent learning with O(N) complexity. In *Proceedings of the IEEE International Joint Conference on Neural Networks, 2004 (IJCNN 2004)*, volume 2, pages 843–848, 2004.
46. David Sussillo and Larry F. Abbott. Generating coherent patterns of activity from chaotic neural networks. *Neuron*, 63(4):544–557, 2009.
47. Sotirios P. Chatzis and Yiannis Demiris. The copula echo state network. *Pattern Recognition*, 45(1):570 – 577, 2012.
48. Zhinwei Shi and Min Han. Support vector echo-state machine for chaotic time-series prediction. *IEEE Transactions on Neural Networks*, 18(2):359–72, 2007.
49. Wolfgang Maass, Prashant Joshi, and Eduardo D. Sontag. Principles of real-time computing with feedback applied to cortical microcircuit models. In *Advances in Neural Information Processing Systems 18 (NIPS 2005)*, pages 835–842. MIT Press, Cambridge, MA, 2006.
50. Hans-Georg Zimmermann, Ralph Grothmann, and Christoph Tietz. Forecasting with recurrent neural networks: 12 tricks. In Klaus-Robert Müller, Grégoire Montavon, and Geneviève B. Orr, editors, *Neural Networks Tricks of the Trade, Reloaded*. Springer, 2012.
51. Herbert Jaeger. Generating exponentially many periodic attractors with linearly growing echo state networks. Technical Report No. 3, Jacobs University Bremen, 2006.
52. Felix R. Reinhart and Jochen J. Steil. Reservoir regularization stabilizes learning of echo state networks with output feedback. In *Proceedings of the 19th European Symposium on Artificial Neural Networks (ESANN 2011)*, 2011. In Press.
53. Felix R. Reinhart and Jochen J. Steil. A constrained regularization approach for input-driven recurrent neural networks. *Differential Equations and Dynamical Systems*, 19:27 – 46, 2011.
54. Jochen J. Steil. Memory in backpropagation-decorrelation O(N) efficient online recurrent learning. In *Proceedings of the 15th International Conference on Artificial Neural Networks (ICANN 2005)*, volume 3697 of *LNCS*, pages 649–654. Springer, 2005.