A tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the "echo state network" approach

Herbert Jaeger
Fraunhofer Institute for Autonomous Intelligent Systems (AIS)
since 2003: International University Bremen

First published: Oct. 2002
First revision: Feb. 2004
Second revision: March 2005
Third revision: April 2008
Forth revision: July 2013 Fifth revision: Dec 2013

Abstract:

This tutorial is a worked-out version of a 5-hour course originally held at AIS in September/October 2002. It has two distinct components. First, it contains a mathematically-oriented crash course on traditional training methods for recurrent neural networks, covering back-propagation through time (BPTT), real-time recurrent learning (RTRL), and extended Kalman filtering approaches (EKF). This material is covered in Sections 2 – 5. The remaining sections 1 and 6 – 9 are much more gentle, more detailed, and illustrated with simple examples. They are intended to be useful as a stand-alone tutorial for the echo state network (ESN) approach to recurrent neural network training.

The author apologizes for the poor layout of this document: it was transformed from an html file into a Word file...

This manuscript was first printed in October 2002 as

H. Jaeger (2002): **Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the "echo state network" approach.** GMD Report 159, German National Research Center for Information Technology, 2002 (48 pp.)

Revision history:
01/04/2004: several serious typos/errors in Sections 3 and 5
03/05/2004: numerous typos
21/03/2005: errors in Section 8.1, updated some URLs
16/04/2008: some more typos
19/07/2013: two typo-errors in the BPTT section (pointed out by Silviu Vlad Oprea)
17/12/2013: typo in Eq. 5.2 (pointed out by James Kirkpatrick)

Index

1. Recurrent neural networks

1.1 First impression

There are two major types of neural networks, feedforward and recurrent. In feedforward networks, activation is "piped" through the network from input units to output units (from left to right in left drawing in Fig. 1.1):

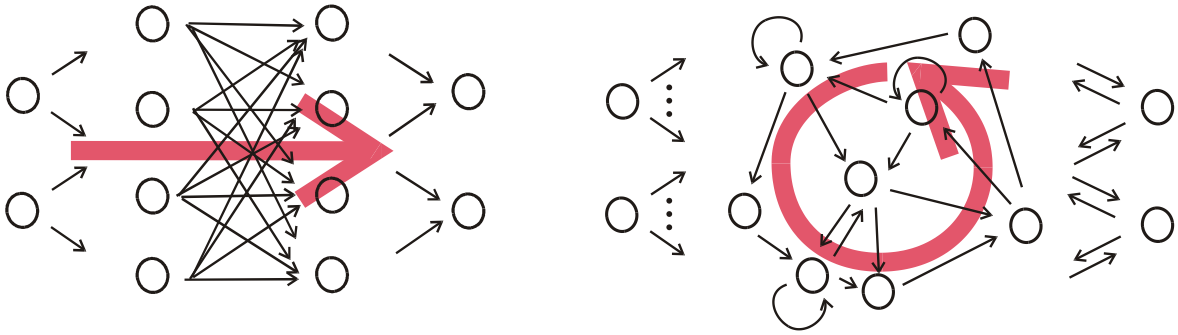

Figure 1.1: *Typical structure of a feedforward network (left) and a recurrent network (right).*

Short characterization of feedforward networks:

- typically, activation is fed forward from input to output through "hidden layers" ("Multi-Layer Perceptrons" MLP), though many other architectures exist
- mathematically, they implement static input-output mappings (functions)
- basic theoretical result: MLPs can approximate arbitrary (term needs some qualification) nonlinear maps with arbitrary precision ("universal approximation property")
- most popular supervised training algorithm: backpropagation algorithm
- huge literature, 95 % of neural network publications concern feedforward nets (my estimate)
- have proven useful in many practical applications as approximators of nonlinear functions and as pattern classificators
- are not the topic considered in this tutorial

By contrast, a recurrent neural network (RNN) has (at least one) cyclic path of synaptic connections. Basic characteristics:

- all biological neural networks are recurrent
- mathematically, RNNs implement dynamical systems
- basic theoretical result: RNNs can approximate arbitrary (term needs some qualification) dynamical systems with arbitrary precision ("universal approximation property")
- several types of training algorithms are known, no clear winner
- theoretical and practical difficulties by and large have prevented practical applications so far

- not covered in most neuroinformatics textbooks, absent from engineering textbooks
- this tutorial is all about them.

Because biological neuronal systems are recurrent, RNN models abound in the biological and biocybernetical literature. Standard types of research papers include...

- bottom-up, detailed neurosimulation:
  - compartment models of small (even single-unit) systems
  - complex biological network models (e.g. Freeman's olfactory bulb models)
- top-down, investigation of principles
  - complete mathematical study of few-unit networks (in AIS: Pasemann, Giannakopoulos)
  - universal properties of dynamical systems as "explanations" for cognitive neurodynamics, e.g. "concept ~ attractor state"; "learning ~ parameter change"; " jumps in learning and development ~ bifurcations"
  - demonstration of dynamical working principles
  - synaptic learning dynamics and conditioning
  - synfire chains

This tutorial does not enter this vast area. The tutorial is about algorithmical RNNs, intended as blackbox models for engineering and signal processing.

The general picture is given in Fig. 1.2:

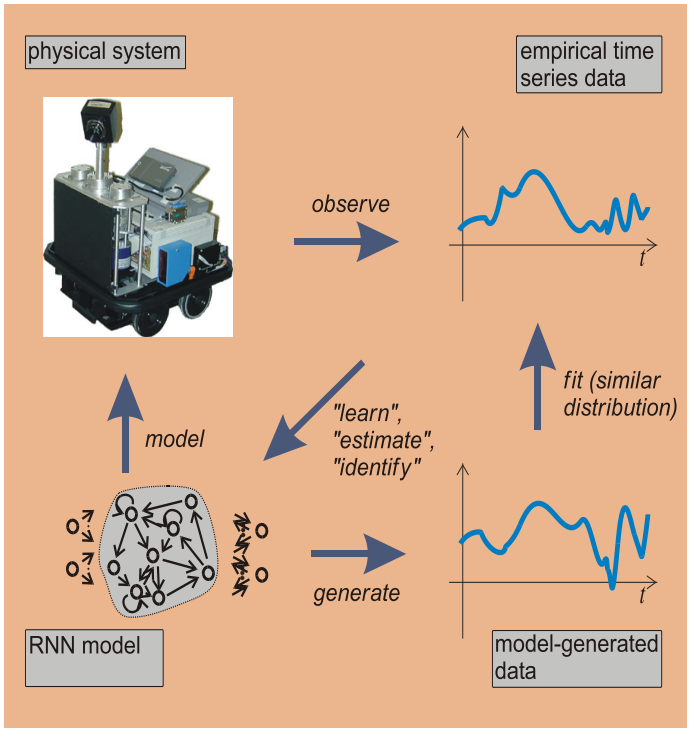

Figure 1.2: *Principal moves in the blackbox modeling game.*
Types of tasks for which RNNs can, in principle, be used:

- system identification and inverse system identification
- filtering and prediction
- pattern classification
- stochastic sequence modeling
- associative memory
- data compression

Some relevant application areas:

- telecommunication
- control of chemical plants
- control of engines and generators
- fault monitoring, biomedical diagnostics and monitoring
- speech recognition
- robotics, toys and edutainment
- video data analysis
- man-machine interfaces

State of usage in applications: RNNs are (not often) proposed in technical articles as "in principle promising" solutions for difficult tasks. Demo prototypes in simulated or clean laboratory tasks. Not economically relevant – yet. Why? supervised training of RNNs is (was) extremely difficult. This is the topic of this tutorial.

1.2 Supervised training: basic scheme

There are two basic classes of "learning": supervised and unsupervised (and unclear cases, e.g. reinforcement learning). This tutorial considers only supervised training.

In supervised training of RNNs, one starts with *teacher data* (or *training data*): empirically observed or artificially constructed input-output time series, which represent examples of the desired model behavior.

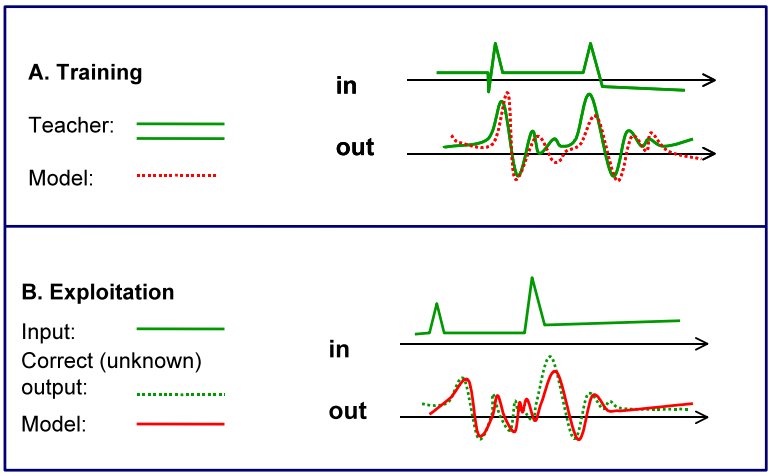

Figure 1.3: *Supervised training scheme.*

The teacher data is used to train a RNN such that it more or less precisely reproduces (*fits*) the teacher data – hoping that the RNN then *generalizes* to novel inputs. That is, when the trained RNN receives an input sequence which is somehow similiar to the training input sequence, it should generate an output which resembles the output of the original system.

A fundamental issue in supervised training is *overfitting*: if the model fits the training data too well (extreme case: model duplicates teacher data exactly), it has only "learnt the training data by heart" and will not generalize well. Particularly important with small training samples. S*tatistical learning theory* addresses this problem. For RNN training, however, this tended to be a non-issue, because known training methods have a hard time fitting training data well in the first place.

1.3 Formal description of RNNs

The elementary building blocks of a RNN are neurons (we will use the term *units*) connected by synaptic links (*connections*) whose synaptic strength is coded by a *weight*. One typically distinguishes *input units*, *internal* (or *hidden*) *units*, and *output units*. At a given time, a unit has an *activation*. We denote the activations of input units by $u(n)$, of internal units by $x(n)$, of output units by $y(n)$. Sometimes we ignore the input/internal/output distinction and then use $x(n)$ in a metonymical fashion.

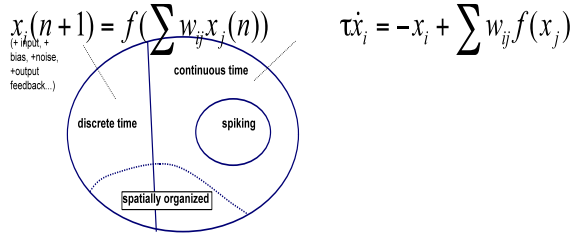

Figure 1.4: *A typology of RNN models (incomplete).*

There are many types of formal RNN models (see Fig. 1.4). Discrete-time models are mathematically cast as maps iterated over discrete time steps $n = 1, 2, 3, \ldots$. Continuous-time models are defined through differential equations whose solutions are defined over a continous time t. Especially for purposes of biological modeling, continuous dynamical models can be quite involved and describe activation signals on the level of individual action potentials (*spikes*). Often the model incorporates a specification of a spatial topology, most often of a 2D surface where units are locally connected in retina-like structures.

In this tutorial we will only consider a particular kind of discrete-time models without spatial organization. Our model consists of K input units with an activation (column) vector

(1.1) $\mathbf{u}(n) = (u_1(n), \ldots, u_K(n))^t$,

of N internal units with an activation vector

(1.2) $\mathbf{x}(n) = (x_1(n),\ldots,x_N(n))^t$,

and of L output units with an activation vector

(1.3) $\mathbf{y}(n) = (y_1(n),\ldots,y_L(n))^t$,

where ^{t} denotes transpose. The input / internal / output connection weights are collected in N x K / N x N / L x (K+N) weight matrices

(1.4) $\mathbf{W}^{in} = (w_{ij}^{in}), \quad \mathbf{W} = (w_{ij}), \quad \mathbf{W}^{out} = (w_{ij}^{out})$.

The output units may optionally project back to internal units with connections whose weights are collected in a N x L backprojection weight matrix

(1.5) $\mathbf{W}^{back} = (w_{ij}^{back})$.

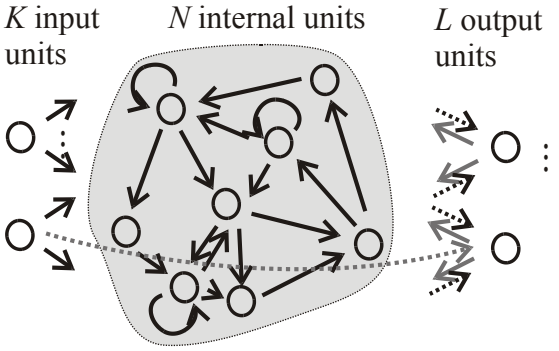

Figure 1.5. *The basic network architecture used in this tutorial. Shaded arrows indicate optional connections. Dotted arrows mark connections which are trained in the "echo state network" approach (in other approaches, all connections can be trained).*

A zero weight value can be interpreted as "no connection". Note that output units may have connections not only from internal units but also (often) from input units and (rarely) from output units.

The activation of internal units is updated according to

(1.6) $\mathbf{x}(n+1) = \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n))$,

where $\mathbf{u}(n+1)$ is the externally given input, and $\mathbf{f}$ denotes the component-wise application of the individual unit's *transfer function,* f (also known as activation function, unit output function, or squashing function). We will mostly use the sigmoid function $f = \tanh$ but sometimes also consider linear networks with $f = 1$. The output is computed according to

(1.7) $\mathbf{y}(n+1) = \mathbf{f}^{out}(\mathbf{W}^{out}(\mathbf{u}(n+1),\mathbf{x}(n+1))$,

where $(\mathbf{u}(n+1),\mathbf{x}(n+1))$ denotes the concatenated vector made from input and internal activation vectors. We will use output transfer functions $f^{out} = \tanh$ or $f^{out} = 1$; in the latter case we have linear output units.

1.4 Example: a little timer network

Consider the input-output task of *timing*. The input signal has two components. The first component $u_1(n)$ is 0 most of the time, but sometimes jumps to 1. The second input $u_2(n)$ can take values between 0.1 and 1.0 in increments of 0.1, and assumes a new (random) of these values each time $u_1(n)$ jumps to 1. The desired output is 0.5 for 10 x $u_2(n)$ time steps after $u_1(n)$ was 1, else is 0. This amounts to implementing a timer: $u_1(n)$ gives the "go" signal for the timer, $u_2(n)$ gives the desired duration.

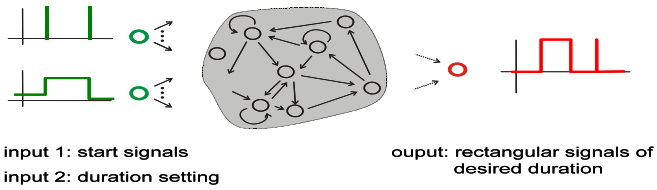

Figure 1.6: *Schema of the timer network.*

The following figure shows traces of input and output generated by a RNN trained on this task according to the ESN approach:

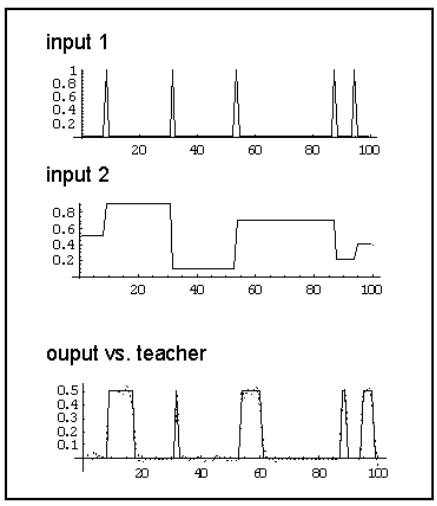

Figure 1.7: *Performance of a RNN trained on the timer task. Solid line in last graph: desired (teacher) output. Dotted line: network ouput.*

Clearly this task requires that the RNN must act as a memory: it has to retain information about the "go" signal for several time steps. This is possible because the

internal recurrent connections let the "go" signal "reverberate" in the internal units' activations. Generally, tasks requiring some form of memory are candidates for RNN modeling.

2. Standard training techniques for RNNs

During the last decade, several methods for supervised training of RNNs have been explored. In this tutorial we present the currently most important ones: backpropagation through time (BPTT), real-time recurrent learning (RTRL), and extended Kalman filtering based techniques (EKF). BPTT is probably the most widely used, RTRL is the mathematically most straightforward, and EKF is (arguably) the technique that gives best results.

2.1 Backpropagation revisited

BPTT is an adaptation of the well-known backpropagation training method known from feedforward networks. The backpropagation algorithm is the most commonly used training method for feedforward networks. We start with a recap of this method.

We consider a multi-layer perceptron (MLP) with k hidden layers. Together with the layer of input units and the layer of output units this gives $k+2$ layers of units altogether (Fig. 1.1. left shows a MLP with two hidden layers), which we number by $0, ..., k+1$. The number of input units is K, of output units L, and of units in hidden layer m is N^m. The weight of the j-th unit in layer m and the i-th unit in layer $m+1$ is denoted by w_{ij}^m. The activation of the i-th unit in layer m is x_i^m (for $m = 0$ this is an input value, for $m = k+1$ an output value).

The training data for a feedforward network training task consist of T input-output (vector-valued) data pairs

(2.1) $\mathbf{u}(n) = (x_1^0(n), \ldots, x_K^0(n))^t$, $\mathbf{d}(n) = (d_1^{k+1}(n), \ldots, d_L^{k+1}(n))^t$,

where n denotes training instance, not time. The activation of non-input units is computed according to

(2.2) $$x_i^{m+1}(n) = f\Big(\sum_{j=1,\ldots,N^m} w_{ij}^m x_j(n)\Big).$$

(Standardly one also has bias terms, which we omit here). Presented with teacher input $\mathbf{u}(t)$, the previous update equation is used to compute activations of units in subsequent hidden layers, until a network response

(2.3) $\mathbf{y}(n) = (x_1^{k+1}(n), \ldots, x_L^{k+1}(n))'$

is obtained in the output layer. The objective of training is to find a set of network weights such that the summed squared error

(2.4) $$E = \sum_{n=1,\ldots,T} \left\| \mathbf{d}(n) - \mathbf{y}(n) \right\|^2 = \sum_{n=1,\ldots,T} E(n)$$

is minimized. This is done by incrementally changing the weights along the direction of the error gradient w.r.t. weights

(2.5) $$\frac{\partial E}{\partial w_{ij}^m} = \sum_{t=1,\ldots T} \frac{\partial E(n)}{\partial w_{ij}^m}$$

using a (small) learning rate γ:

(2.6) $$new\ w_{ij}^m = w_{ij}^m - \gamma \frac{\partial E}{\partial w_{ij}^m} .$$

(Superscript m omitted for readability). This is the formula used in *batch learning mode*, where new weights are computed after presenting all training samples. One such pass through all samples is called an epoch. Before the first epoch, weights are initialized, typically to small random numbers. A variant is *incremental learning*, where weights are changed after presentation of individual training samples:

(2.7) $$new\ w_{ij}^m = w_{ij}^m - \gamma \frac{\partial E(n)}{\partial w_{ij}^m} .$$

The central subtask in this method is the computation of the error gradients $\frac{\partial E(n)}{\partial w_{ij}^m}$.

The backpropagation algorithm is a scheme to perform these computations. We give the recipe for one epoch of batch processing.

Input: current weights w_{ij}^m, training samples.

Output: new weights.

Computation steps:

1. For each sample n, compute activations of internal and output units using (2.2) ("forward pass").
2. Compute, by proceeding backward through $m = k+1,k,\ldots,1$, for each unit x_i^m the *error propagation term* $\delta_i^m(n)$

   (2.8) $$\delta_i^{k+1}(n) = (d_i(n) - y_i(n)) \frac{\partial f(u)}{\partial u}\Big|_{u=z_i^{k+1}}$$

   for the output layer and

(2.9) $$\delta_j^m(n) = \sum_{i=1}^{N^{m+1}} \delta_i^{m+1} w_{ij}^m \left.\frac{\partial f(u)}{\partial u}\right|_{u=z_j^m}$$

for the hidden layers, where

(2.10) $$z_i^m(n) = \sum_{j=1}^{N^{m-1}} x_j^{m-1}(n) w_{ij}^{m-1}$$

is the internal state (or "potential") of unit x_i^m. This is the *error backpropagation* pass. Mathematically, the error propagation term $\delta_i^m(n)$ represents the error gradient w.r.t. the potential of the unit unit x_i^m.

(2.10a) $$\left.\frac{\partial E}{\partial u}\right|_{u=z_j^m} .$$

3.

Adjust the connection weights according to

(2.11) $$new\ w_{ij}^{m-1} = w_{ij}^{m-1} + \gamma \sum_{t=1}^{T} \delta_i^m(n)\, x_j^{m-1}(n) .$$

After every such epoch, compute the error according to (2.4). Stop when the error falls below a predetermined threshold, or when the change in error falls below another predetermined threshold, or when the number of epochs exceeds a predetermined maximal number of epochs. Many (order of thousands in nontrivial tasks) such epochs may be required until a sufficiently small error is achieved.

One epoch requires $O(T M)$ multiplications and additions, where M is the total number of network connections.

The basic gradient descent approach (and its backpropagation algorithm implementation) is notorious for slow convergence, because the learning rate γ must be typically chosen small to avoid instability. Many speed-up techniques are described in the literature, e.g. dynamic learning rate adaptation schemes. Another approach to achieve faster convergence is to use second-order gradient descent techniques, which exploit curvature of the gradient but have epoch complexity $O(T M^2)$.

Like all gradient-descent techniques on error surfaces, backpropagation finds only a local error minimum. This problem can be addressed by various measures, e.g. adding noise during training (simulated annealing approaches) to avoid getting stuck in poor minima, or by repeating the entire learning from different initial weight settings, or by using task-specific prior information to start from an already plausible set of weights. Some authors claim that the local minimum problem is overrated.

Another problem is the selection of a suitable network topology (number and size of hidden layers). Again, one can use prior information, perform systematic search, or use intuition.

All in all, while basic backpropagation is transparent and easily implemented, considerable expertise and experience (or patience) is a prerequisite for good results in non-trivial tasks.

2.2. Backpropagation through time

The feedforward backpropagation algorithm cannot be directly transferred to RNNs because the error backpropagation pass presupposes that the connections between units induce a cycle-free ordering. The solution of the BPTT approach is to "unfold" the recurrent network in time, by stacking identical copies of the RNN, and redirecting connections within the network to obtain connections between subsequent copies. This gives a feedforward network, which is amenable to the backpropagation algorithm.

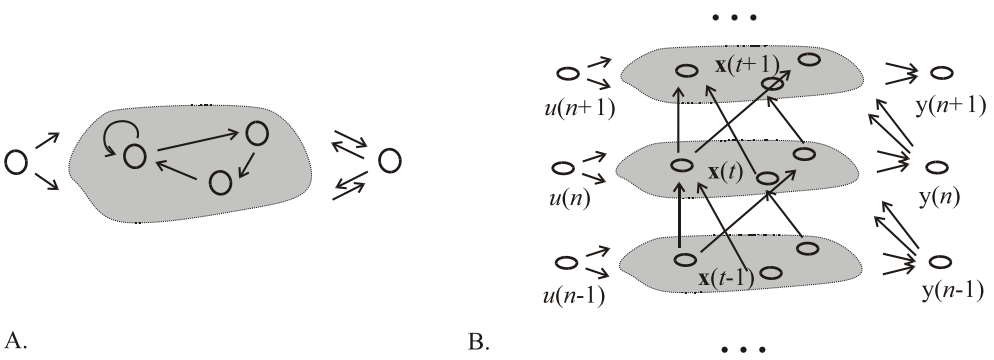

Figure 2.1: Schema of the basic idea of BPTT. A: the original RNN. B: The feedforward network obtained from it. The case of single-channel input and output is shown.

The weights w_{ij}^{in}, w_{ij}, w_{ij}^{out}, w_{ij}^{back} are identical in/between all copies. The teacher data consists now of a single input-output time series

(2.12) $\mathbf{u}(n) = (u_1(n),\ldots,u_K(n))'$, $\mathbf{d}(n) = (d_1(n),\ldots,d_L(n))'$ $\quad n = 1,\ldots T$.

The forward pass of one training epoch consists in updating the stacked network, starting from the first copy and working upwards through the stack. At each copy / time n input $\mathbf{u}(n)$ is read in, then the internal state $\mathbf{x}(n)$ is computed from $\mathbf{u}(n)$, $\mathbf{x}(n\text{-}1)$ [and from $\mathbf{y}(n\text{-}1)$ if nonzero w_{ij}^{back} exist], and finally the current copy's output $\mathbf{y}(n)$ is computed.

The error to be minimized is again (like in 2.4)

(2.13) $E = \sum_{n=1,\ldots,T} \left\| \mathbf{d}(n) - \mathbf{y}(n) \right\|^2 = \sum_{n=1,\ldots,T} E(n)$,

but the meaning of t has changed from "training instance" to "time". The algorithm is a straightforward, albeit notationally complicated, adaptation of the feedforward algorithm:

Input: current weights w_{ij}, training time series.

Output: new weights.

Computation steps:

1. Forward pass: as described above.
2. Compute, by proceeding backward through $n = T,...,1$, for each time n and unit activation $x_i(n), y_j(n)$ the error propagation term $\delta_i(n)$

(2.14) $$\delta_j(T) = (d_j(T) - y_j(T))\frac{\partial f(u)}{\partial u}\Big|_{u=z_j(T)}$$

for the output units of time layer T and

(2.15) $$\delta_i(T) = \left[\sum_{j=1}^{L}\delta_j(T)w_{ji}^{out}\right]\frac{\partial f(u)}{\partial u}\Big|_{u=z_i(n)}$$

for internal units $x_i(T)$ at time layer T and

(2.16) $$\delta_j(n) = \left[(d_j(n) - y_j(n)) + \sum_{i=1}^{N}\delta_i(n+1)w_{ij}^{back}\right]\frac{\partial f(u)}{\partial u}\Big|_{u=z_j(n)}$$

for the output units of earlier layers, and

(2.17) $$\delta_i(n) = \left[\sum_{j=1}^{N}\delta_j(n+1)w_{ji} + \sum_{j=1}^{L}\delta_j(n)w_{ji}^{out}\right]\frac{\partial f(u)}{\partial u}\Big|_{u=z_i(n)}$$

for internal units $x_i(n)$ at earlier times, where $z_i(n)$ again is the potential of the corresponding unit.

3. Adjust the connection weights according to

$$new\ w_{ij} = w_{ij} + \gamma \sum_{n=1}^{T} \delta_i(n)\, x_j(n-1) \quad [use\ x_j(n-1) = 0\ for\ n = 1]$$

$$new\ w_{ij}^{in} = w_{ij}^{in} + \gamma \sum_{n=1}^{T} \delta_i(n)\, u_j(n)$$

(2.18)
$$new\ w_{ij}^{out} = w_{ij}^{out} + \gamma \times \begin{cases} \sum_{n=1}^{T} \delta_i(n)\, u_j(n), if\ \ j\ refers\ to\ input\ unit \\ \sum_{n=1}^{T} \delta_i(n)\, x_j(n), if\ \ j\ refers\ to\ hidden\ unit \end{cases}$$

$$new\ w_{ij}^{back} = w_{ij}^{back} + \gamma \sum_{n=1}^{T} \delta_i(n)\, y_j(n-1) \quad [use\ y_j(n-1) = 0\ for\ n = 1]$$

Warning: programming errors are easily made but not easily perceived when they degrade performance only slightly.

The remarks concerning slow convergence made for standard backpropagation carry over to BPTT. The computational complexity of one epoch is $O(T N^2)$, where N is number of internal units. Several thousands of epochs are often required.

A variant of this algorithm is to use the teacher output $\mathbf{d}(n)$ in the computation of activations in layer $n+1$ in the forward pass. This is known as *teacher forcing*. Teacher forcing typically speeds up convergence, or even may be necessary to achieve convergence at all, but when the trained network is exploited, it may exhibit instability. A general rule when to use teacher forcing cannot be given.

A drawback of this "batch" BPTT is that the entire teacher time series must be used. This precludes applications where online adaptation is required. The solution is to truncate the past history and use, at time n, only a finite history

(2.19) $\mathbf{u}(n-p), \mathbf{u}(n-p+1), \ldots, \mathbf{u}(n)\ , \mathbf{d}(n-p), \mathbf{d}(n-p+1), \ldots, \mathbf{d}(n)$

as training data. Since the error backpropagation terms δ need to be computed only once for each new time slice, the complexity is $O(N^2)$ per time step. A potential drawback of such truncated BPPT (or p-BPTT) is that memory effects exceeding a duration p cannot be captured by the model. Anyway, BPTT generally has difficulties capturing long-lifed memory effects, because backpropagated error gradient information tends to "dilute" exponentially over time. A frequently stated opinion is that memory spans longer than 10 to 20 time steps are hard to achieve.

Repeated execution of training epochs shift a complex nonlinear dynamical system (the network) slowly through parameter (weight) space. Therefore, bifurcations are necessarily encountered when the starting weights induce a qualitatively different dynamical behavior than task requires. Near such bifurcations, the gradient information may become essentially useless, dramatically slowing down convergence. The error may even suddenly grow in the vicinity of such critical points, due to crossing bifurcation boundaries. Unlike feedforward backpropagation, BPTT is

not guaranteed to converge to a local error minimum. This difficulty cannot arise with feedforward networks, because they realize functions, not dynamical systems.

All in all, it is far from trivial to achieve good results with BPTT, and much experimentation (and processor time) may be required before a satisfactory result is achieved.

Because of limited processing time, BPTT is typically used with small networks sizes in the order of 3 to 20 units. Larger networks may require many hours of computation on current hardware.

3. Real-time recurrent learning

Real-time recurrent learning (RTRL) is a gradient-descent method which computes the exact error gradient at every time step. It is therefore suitable for online learning tasks. I basically quote the description of RTRL given in Doya (1995). The most often cited early description of RTRL is Williams & Zipser (1989).

The effect of weight change on the network dynamics can be seen by simply differentiating the network dynamics equations (1.6) and (1.7) by its weights. For convenience, activations of all units (whether input, internal, or output) are enumerated and denoted by v_i and all weights are denoted by w_{kl}, with $i = 1,...,N$ denoting internal units, $i = N+1,...,N+L$ denoting output units, and $i = N+L+1,...,N+L+K$ denoting input units. The derivative of an internal or output unit w.r.t. a weight w_{kl} is given by

$$\text{(3.1)} \quad \frac{\partial v_i(n+1)}{\partial w_{kl}} = f'(z_i(n))\left[\left(\sum_{j=1}^{N+L} w_{ij}\frac{\partial v_j(n)}{\partial w_{kl}}\right) + \delta_{ik} v_l(n)\right] \quad i = 1,\ldots,N+L,$$

where $k,l \le N+L+K$, $z_i(n)$ is again the unit's potential, but δ_{ik} here denotes Kronecker's delta ($\delta_{ik} = 1$ if $i = k$ and 0 otherwise). The term $\delta_{ik}\, v_l(n)$ represents an explicit effect of the weight w_{kl} onto the unit k, and the sum term represents an implicit effect onto all the units due to network dynamics.

Equation (3.1) for each internal or output unit constitutes an $N+L$-dimensional discrete-time linear dynamical system with time-varying coefficients, where

$$\text{(3.2)} \quad \left(\frac{\partial v_1}{\partial w_{kl}},\ldots,\frac{\partial v_{N+L}}{\partial w_{kl}}\right)$$

is taken as a dynamical variable. Since the initial state of the network is independent of the connection weights, we can initialize (3.1) by

$$\text{(3.3)} \quad \frac{\partial v_i(0)}{\partial w_{kl}} = 0.$$

Thus we can compute (3.2) forward in time by iterating Equation (3.1) simultaneously with the network dynamics (1.6) and (1.7). From this solution, we can calculate the error gradient (for the error given in (2.13)) as follows:

$$(3.4) \quad \frac{\partial E}{\partial w_{kl}} = 2\sum_{n=1}^{T}\sum_{i=N}^{N+L}(v_i(n) - d_i(n))\frac{\partial v_i(n)}{\partial w_{kl}}.$$

A standard batch gradient descent algorithm is to accumulate the error gradient by Equation (3.4) and update each weight after a complete epoch of presenting all training data by

$$(3.5) \quad new \;\; w_{kl} = w_{kl} - \gamma\frac{\partial E}{\partial w_{kl}},$$

where γ is is a learning rate. An alternative update scheme is the gradient descent of current output error at each time step,

$$(3.6) \quad w_{kl}(n+1) = w_{kl}(n) - \gamma\sum_{i=1}^{L}(v_i(n) - d_i(n))\frac{\partial v_i(n)}{\partial w_{kl}}.$$

Note that we assumed w_{kl} is a constant, not a dynamical variable, in deriving (3.1), so we have to keep the learning rate small enough. (3.6) is referred to as real-time recurrent learning.

RTRL is mathematically transparent and in principle suitable for online training. However, the computational cost is $O((N+L)^4)$ for each update step, because we have to solve the $(N+L)$-dimensional system (3.1) for each of the weights. This high computational cost makes RTRL useful for online adaptation only when very small networks suffice.

4. Higher-order gradient descent techniques

Just a little note: Pure gradient-descent techniques for optimization generally suffer from slow convergence when the curvature of the error surface is different in different directions. In that situation, on the one hand the learning rate must be chosen small to avoid instability in the directions of high curvature, but on the other hand, this small learning rate might lead to unacceptably slow convergence in the directions of low curvature. A general remedy is to incorporate curvature information into the gradient descent process. This requires the calculation of the second-order derivatives, for which several approximative techniques have been proposed in the context of recurrent neural networks. These calculations are expensive, but can accelerate convergence especially near an optimum where the error surface can be reasonably approximated by a quadratic function. Dos Santos & von Zuben (2000) and Schraudolph (2002) provide references, discussion, and propose approximation techniques which are faster than naive calculations.

5. Extended Kalman-filtering approaches

5.1 The extended Kalman filter

The extended Kalman filter (EKF) is a state estimation technique for nonlinear systems derived by linearizing the well-known linear-systems Kalman filter around the current state estimate. We consider a simple special case, a time-discrete system with additive input and no observation noise:

$$\begin{aligned} \mathbf{x}(n+1) &= \mathbf{f}(\mathbf{x}(n)) + \mathbf{q}(n) \\ \mathbf{d}(n) &= \mathbf{h}_n(\mathbf{x}(n)) \end{aligned} \quad , \tag{5.1}$$

where $\mathbf{x}(n)$ is the system's internal state vector, $\mathbf{f}$ is the system's state update function (linear in original Kalman filters), $\mathbf{q}(n)$ is external input to the system (an uncorrelated Gaussian white noise process, can also be considered as process noise), $\mathbf{d}(n)$ is the system's output, and $\mathbf{h}_n$ is a time-dependent observation function (also linear in the original Kalman filter). At time $n = 0$, the system state $\mathbf{x}(0)$ is guessed by a multidimensional normal distribution with mean $\hat{\mathbf{x}}(0)$ and covariance matrix $\mathbf{P}(0)$. The system is observed until time n through $\mathbf{d}(0),..., \mathbf{d}(n)$. The task addressed by the extended Kalman filter is to give an estimate $\hat{\mathbf{x}}(n+1)$ of the true state $\mathbf{x}(n+1)$, given the initial state guess and all previous output observations. This task is solved by the following two *time update* and three *measurement update* computations:

$$\begin{aligned} \hat{\mathbf{x}}^*(n) &= \mathbf{f}(\hat{\mathbf{x}}(n)) \\ \mathbf{P}^*(n) &= \mathbf{F}(n)\mathbf{P}(n-1)\mathbf{F}(n)^t + \mathbf{Q}(n) \end{aligned} \tag{5.2}$$

$$\begin{aligned} \mathbf{K}(n) &= \mathbf{P}^*(n)\mathbf{H}(n)[\mathbf{H}(n)^t \mathbf{P}^*(n)\mathbf{H}(n)]^{-1} \\ \hat{\mathbf{x}}(n+1) &= \hat{\mathbf{x}}^*(n) + \mathbf{K}(n)\xi(n) \\ \mathbf{P}(n+1) &= \mathbf{P}^*(n) - \mathbf{K}(n)\mathbf{H}(n)^t \mathbf{P}^*(n) \end{aligned} \tag{5.3}$$

where we roughly follow the notation in Singhal and Wu (1989), who first applied extended Kalman filtering to (feedforward) network weight estimation. Here $\mathbf{F}(n)$ and $\mathbf{H}(n)$ are the Jacobians

$$\mathbf{F}(n) = \left.\frac{\partial \mathbf{f}\ (\mathbf{x})}{\partial \mathbf{x}}\right|_{\mathbf{x}=\hat{\mathbf{x}}(n)}, \quad \mathbf{H}(n) = \left.\frac{\partial \mathbf{h}_n(\mathbf{x})}{\partial \mathbf{x}}\right|_{\mathbf{x}=\hat{\mathbf{x}}(n)} \tag{5.4}$$

of the components of $\mathbf{f}$, $\mathbf{h}_n$ with respect to the state variables, evaluated at the previous state estimate; $\xi(n) = \mathbf{d}(n) - \mathbf{h}_n(\hat{\mathbf{x}}(n))$ is the error (difference between observed output and output calculated from state estimate $\hat{\mathbf{x}}(n)$), $\mathbf{P}(n)$ is an estimate of the conditional error covariance matrix $\mathrm{E}[\xi\xi \mid \mathbf{d}(0),..., \mathbf{d}(n)]$; $\mathbf{Q}(n)$ is the (diagonal) covariance matrix of the process noise, and the time updates $\hat{\mathbf{x}}^*(n), \mathbf{P}^*(n)$ of state estimate and state error covariance estimate are obtained from extrapolating the previous estimates with the known dynamics $\mathbf{f}$.

The basic idea of Kalman filtering is to first update $\hat{\mathbf{x}}(n), \mathbf{P}(n)$ to preliminary guesses $\hat{\mathbf{x}}^*(n), \mathbf{P}^*(n)$ by extrapolating from their previous values, applying the known dynamics in the time update steps (5.2), and then adjusting these preliminary guesses by incorporating the information contained in $\mathbf{d}(n)$ – this information enters the measurement update in the form of $\xi(n)$, and is accumulated in the *Kalman gain* $\mathbf{K}(n)$.

In the case of classical (linear, stationary) Kalman filtering, $\mathbf{F}(n)$ and $\mathbf{H}(n)$ are constant, and the state estimates converge to the true conditional mean state $\mathrm{E}[\mathbf{x}(n) \mid \mathbf{d}(0),..., \mathbf{d}(n)]$. For nonlinear $\mathbf{f}$, $\mathbf{h}_n$, this is not generally true, and the use of extended Kalman filters leads only to locally optimal state estimates.

5.2 Applying EKF to RNN weight estimation

Assume that there exists a RNN which perfectly reproduces the input-output time series of the training data

(5.5) $\mathbf{u}(n) = (u_1(n),\ldots,u_K(n))^t,\ \mathbf{d}(n) = (d_1(n),\ldots,d_L(n))^t \qquad n = 1,\ldots T$

where the input / internal / output / backprojection connection weights are as usual collected in N x K / N x N / L x (K+N+L) / N x L weight matrices

(5.6) $\mathbf{W}^{in} = (w_{ij}^{in}),\quad \mathbf{W} = (w_{ij}),\quad \mathbf{W}^{out} = (w_{ij}^{out}),\ \mathbf{W}^{back} = (w_{ij}^{back})$.

In this subsection, we will not distinguish between all these different types of weights and refer to all of them by a weight vector $\mathbf{w}$.

In order to apply EKF to the task of estimating optimal weights of a RNN, we interpret the weights $\mathbf{w}$ of the perfect RNN as the state of a dynamical system. From a bird's eye perspective, the output $\mathbf{d}(n)$ of the RNN is a function $\mathbf{h}$ of the weights and input up to n:

(5.7) $\mathbf{d}(n) = \mathbf{h}(\mathbf{w}, \mathbf{u}(0),..., \mathbf{u}(n))$

where we assume that the transient effects of the initial network state have died out. The inputs can be integrated into the output function $\mathbf{h}$, rendering it a time-dependent function $\mathbf{h}_n$. We further assume that the network update contains some process noise, which we add to the weights (!) in the form of a Gaussian uncorrelated noise $\mathbf{q}(n)$. This gives the following version of (5.1) for the dynamics of the perfect RNN:

(5.8)
$$\begin{aligned} \mathbf{w}(n+1) &= \mathbf{w}(n) + \mathbf{q}(n) \\ \mathbf{d}(n) &= \mathbf{h}_n(\mathbf{w}(n)) \end{aligned}$$

Except for noisy shifts induced by process noise, the state "dynamics" of this system is static, and the input $\mathbf{u}(n)$ to the network is not entered in the state update equation,

but is hidden in the time-dependence of the observation function. This takes some mental effort to swallow!

The network training task now takes the form of estimating the (static, perfect) state $\mathbf{w}(n)$ from an initial guess $\hat{\mathbf{w}}(0)$ and the sequence of outputs $\mathbf{d}(0),..., \mathbf{d}(n)$. The error covariance matrix $\mathbf{P}(0)$ is initialized as a diagonal matrix with large diagonal components, e.g. 100.

The simpler form of (5.8) over (5.1) leads to some simplifications of the EKF recursions (5.2) and (5.3): because the system state (= weight!) dynamics is now trivial, the time update steps become unnecessary. The measurement updates become

$$\begin{aligned} &\mathbf{K}(n) = \mathbf{P}(n)\mathbf{H}(n)[\mathbf{H}(n)^t \mathbf{P}(n)\mathbf{H}(n)]^{-1} \\ &\hat{\mathbf{w}}(n+1) = \hat{\mathbf{w}}(n) + \mathbf{K}(n)\xi(n) \\ &\mathbf{P}(n+1) = \mathbf{P}(n) - \mathbf{K}(n)\mathbf{H}(n)^t \mathbf{P}(n) + \mathbf{Q}(n) \end{aligned} \tag{5.9}$$

A learning rate η (small at the beginning [!] of learning to compensate for initially bad estimates of $\mathbf{P}(n)$) can be introduced into the Kalman gain update equation:

$$\mathbf{K}(n) = \mathbf{P}(n)\mathbf{H}(n)[(1/\eta)\mathbf{I} + \mathbf{H}(n)^t \mathbf{P}(n)\mathbf{H}(n)]^{-1}, \tag{5.10}$$

which is essentially the formulation given in Puskorius and Feldkamp (1994).

Inserting process noise $\mathbf{q}(n)$ into EKF has been claimed to improve the algorithm's numerical stability, and to avoid getting stuck in poor local minima (Puskorius and Feldkamp 1994).

EKF is a second-order gradient descent algorithm, in that it uses curvature information of the (squared) error surface. As a consequence of exploiting curvature, for linear noise-free systems the Kalman filter can converge in a single step. We demonstrate this by a super-simple feedforward network example. Consider the single-input, single-output network which connects the input unit with the output unit by a connection with weight w, without any internal unit

$$\begin{aligned} w(n+1) &= w(n) \\ d(n) &= w\,u(n) \end{aligned} \tag{5.11}$$

We inspect the running EKF (in the version of (5.9)) at some time n, where it has reached an estimate $\hat{w}(n)$. Observing that the Jacobian $\mathbf{H}(n)$ is simply $\mathrm{d}wu(n)/\mathrm{d}w = u(n)$, the next estimated state is

$$\hat{w}(n+1) = \hat{w}(n) + \mathbf{K}(n)\xi(n) = \hat{w}(n) + \frac{1}{u(n)}(wu(n) - \hat{w}u(n)) = w\,. \tag{5.12}$$

The EKF is claimed in the literature to exhibit fast convergence, which should have become plausible from this example at least for cases where the current estimate

$\hat{\mathbf{w}}(n)$ is already close to the correct value, such that the linearisation yields a good approximation to the true system.

EKF requires the derivatives $\mathbf{H}(n)$ of the network outputs w.r.t. the weights evaluated at the current weight estimate. These derivatives can be exactly computed as in the RTRL algorithm, at cost $\mathrm{O}(N^4)$. This is too expensive but for small networks. Alternatively, one can resort to truncated BPTT, use a "stacked" version of (5.8) which describes a finite sequence of outputs instead of a single output, and obtain approximations to $\mathbf{H}(n)$ by a procedure analogous to (2.14) – (2.17). Two variants of this approach are detailed out in Feldkamp et al. (1998). The cost here is $\mathrm{O}(pN^2)$, where p is the truncation depth.

Apart from the calculation of $\mathbf{H}(n)$, the most expensive operation in EKF is the update of $\mathbf{P}(n)$, which requires $\mathrm{O}(LN^2)$ computations. By setting up the network architecture with suitably decoupled subnetworks, one can achieve a block-diagonal $\mathbf{P}(n)$, with considerable reduction in computations (Feldkamp et al. 1998).

As far as I have an overview, it seems to me that currently the best results in RNN training are achieved with EKF, using truncated BPTT for estimating $\mathbf{H}(n)$, demonstrated especially in many remarkable achievements from Lee Feldkamp's research group (see references). As with BPTT and RTRL, the eventual success and quality of EKF training depends very much on professional experience, which guides the appropriate selection of network architecture, learning rates, the subtelties of gradient calculations, presentation of input (e.g., windowing techniques), etc.

6. Echo state networks

6.1 Training echo state networks

6.1.1 First example: a sinewave generator

In this subsection I informally demonstrate the principles of echo state networks (ESN) by showing how to train a RNN to generate a sinewave.

The desired sinewave is given by $d(n) = 1/2 \sin(n/4)$. The task of generating such a signal involves no input, so we want a RNN without any input units and a single output unit which after training produces $d(n)$. The teacher signal is a 300-step sequence of $d(n)$.

We start by constructing a recurrent network with 20 units, whose internal connection weights $\mathbf{W}$ are set to random values. We will refer to this network as the "dynamical reservoir" (DR). The internal weights $\mathbf{W}$ will not be changed in the training described later in this subsection. The network's units are standard sigmoid units, as in Eq. (1.6), with a transfer function $f = \tanh$.

A randomly constructed RNN, such as our DR, might develop oscillatory or even chaotic acitivity even in the absence of external excitation. We do not want this to occur: The ESN approach needs a DR which is *damped*, in the sense that if the

network is started from an arbitrary state $\mathbf{x}(0)$, the subsequent network states converge to the zero state. This can be achieved by a proper global scaling of $\mathbf{W}$: the smaller the weights of $\mathbf{W}$, the stronger the damping. We assume that we have scaled $\mathbf{W}$ such that we have a DR with modest damping. Fig. 6.1 shows traces of the 20 units of our DR when it was started from a random initial state $\mathbf{x}(0)$. The desired damping is clearly visible.

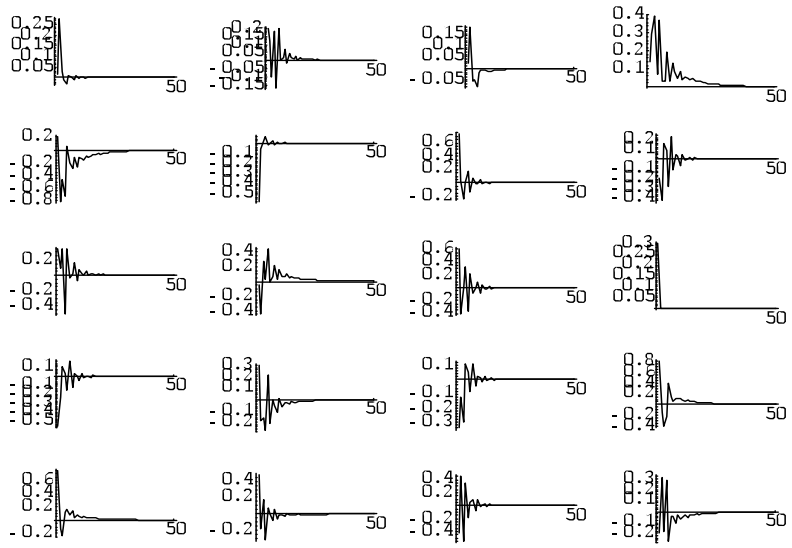

Figure 6.1. *The damped dynamics of our dynamical reservoir.*

We add a single output unit to this DR. This output unit features connections that project back into the DR. These backprojections are given random weights $\mathbf{W}^{back}$, which are also fixed and do not change during subsequent training. We use a linear output unit in this example, i.e. $f^{out} = \mathrm{id}$.

The only connections which *are* changed during learning are the weights $\mathbf{W}^{out}$ from the DR to the output unit. These weights are not defined (nor are they used) during training. Figure 6.2 shows the network prepared for training.

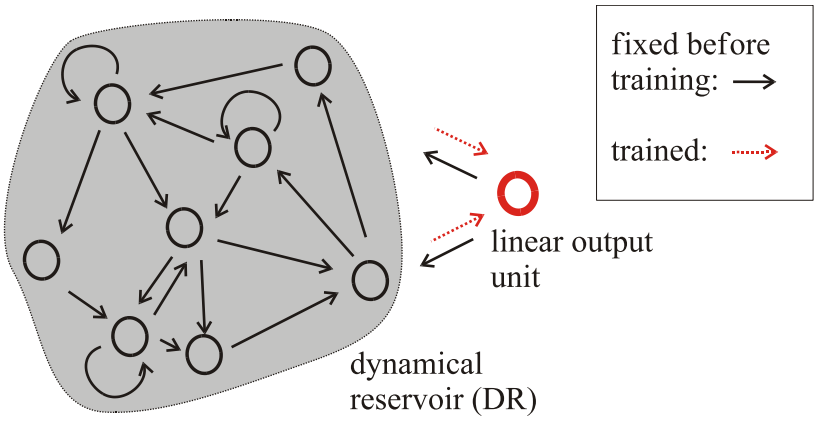

Figure 6.2: *Schematic setup of ESN for training a sinewave generator.*

The training is done in two stages, *sampling* and *weight computation*.

Sampling. During the sampling stage, the teacher signal is written into the output unit for times $n = 1,....,300$. (Writing the desired output into the output units during training is often called *teacher forcing*). The network is started at time $n = 1$ with an arbitrary starting state; we use the zero state for starting but that is just an arbitrary decision. The teacher signal $d(n)$ is pumped into the DR through the backprojection connections $\mathbf{W}^{back}$ and thereby excites an activation dynamics within the DR. Figure 6.3 shows what happens inside the DR for sampling time steps $n = 101,...,150$.

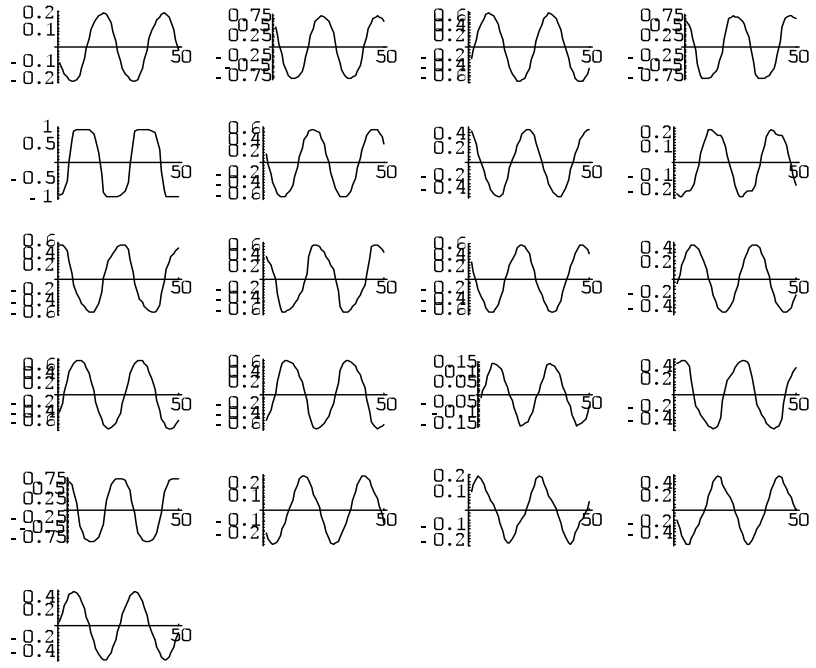

Figure 6.3. The dynamics within the DR induced by teacher-forcing the sinewave $d(n)$ in the output unit. 50-step traces of the 20 internal DR units and of the teacher signal (last plot) are shown.

We can make two important observations:

1. The activation patterns within the DR are all periodic signals of the same period length as the driving teacher $d(n)$.
2. The activation patterns within the DR are different from each other.

During the sampling period, the internal states $\mathbf{x}(n) = (x_1(n),...,x_{20}(n))$ for $n = 101, ..., 300$ are collected into the rows of a state-collecting matrix $\mathbf{M}$ of size 200x20. At the same time, the teacher outputs $d(n)$ are collected into the rows of a matrix $\mathbf{T}$ of size 200x1.

We do not collect information from times $n = 1, ..., 100$, because the network's dynamics is initially partly determined by the network's arbitrary starting state. By time $n = 100$, we can safely assume that the effects of the arbitrary starting state have died out and that the network states are a pure reflection of the teacher-forced $d(n)$, as is manifest in Fig. 6.3.

Weight computation. We now compute 20 output weights w_i^{out} for our linear output unit $y(n)$ such that the teacher time series $d(n)$ is approximated as a linear combination of the internal activation time series $x_i(n)$ by

$$d(n) \approx y(n) = \sum_{i=1}^{20} w_i^{out} x_i(n) \quad . \tag{6.1}$$

More specifically, we compute the weights w_i^{out} such that the mean squared training error

$$\mathrm{MSE}_{train} = 1/200 \sum_{n=101}^{300} (d(n) - y(n))^2 = 1/200 \sum_{n=101}^{300} (d(n) - \sum_{i=1}^{20} w_i^{out} x_i(n))^2 \tag{6.2}$$

is minimized.

From a mathematical point of view, this is a linear regression task: compute *regression weights* w_i^{out} for a regression of $d(n)$ on the network states $x_i(n)$. [$n = 101, ..., 300$].

From an intuitive-geometrical point of view, this means combining the 20 internal signals seen in Fig. 6.3 such that the resulting combination best approximates the last (teacher) signal seen in the same figure.

From an algorithmical point of view, this *offline* computation of regression weights boils down to the computation of a pseudoinverse: The desired weights which minimize MSE_{train} are obtained by multiplying the pseudoinverse of $\mathbf{M}$ with $\mathbf{T}$:

$$\mathbf{W}^{\mathrm{out}} = \mathbf{M}^{-1}\mathbf{T} \tag{6.3}$$

Computing the pseudoinverse of a matrix is a standard operation of numerical linear algebra. Ready-made functions are included in Mathematica and Matlab, for example.

In our example, the training error computed by (6.2) with optimal output weights obtained by (6.3) was found to be MSE_{train}.= 1.2e–13.

The computed output weights are implemented in the network, which is then ready for use.

Exploitation. After the learnt output weights were written into the output connections, the network was run for another 50 steps, continuing from the last training network state $\mathbf{x}(300)$, but now with teacher forcing switched off. The output $y(n)$ was now generated by the trained network all on its own [$n = 301, ..., 350$]. The *test error*

$$\mathrm{MSE}_{test} = 1/50 \sum_{n=301}^{350} (d(n) - y(n))^2 \tag{6.4}$$

was found to be MSE_{test}.= 5.6e−12. This is greater than the training error, but still very small. The network has learnt to generate the desired sinewave very precisely. An intuitive explanation of this precision would go as follows:

- The sinewave $y(n)$ at the output unit evokes periodic signals $x_i(n)$ inside the DR whose period length is *identical* to that of the output sine.
- These periodic signals make a kind of "basis" of signals from which the target $y(n)$ is combined. This "basis" is optimally "pre-adapted" to the target in the sense of identical period length. This pre-adaptation is a natural consequence of the fact that the "basis" signals $x_i(n)$ have been induced *by the target* itself, via the feedback projections.

So, in a sense, the task [to combine $y(n)$ from $x_i(n)$] is solved by means [the $x_i(n)$] which have been formed by the very task [by the backprojection of $y(n)$ into the DR]. Or said in intuitive terms, the target signal $y(n)$ is re-constituted from its own echos $x_i(n)$!

An immediate question concerns the stability of the solution. One may rightfully wonder whether the error in testing phase, small as it was in the first 50 steps, will not grow over time and finally render the network's global oscillation unstable. That is, we might suspect that the precise continuation of the sine output after the training is due to the fact that we start testing from state $\mathbf{x}(300)$, which was produced by teacher forcing. However, this is not usually the case. Most networks trained according to the prescription given here can be started from almost any arbitrary nonzero starting state and will lock into the desired sinewave. Figure 6.4 shows an example. In mathematical terms, the trained network is a dynamical system with a single attractor, and this attractor is the desired oscillation. However, the strong stability observed in this example is a pleasant side-effect of the simplicity of the sinewave generating task. When the tasks become more difficult, the stability of the trained dynamics *is* indeed a critical issue for ESN training.

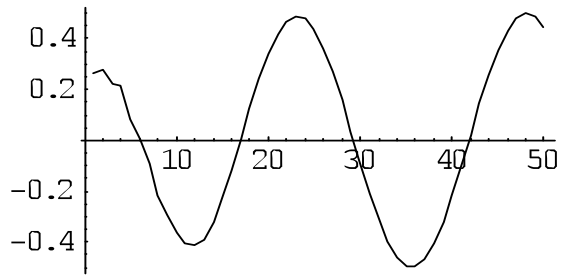

Figure 6.4: Starting the trained network from a random starting state. Plot shows first 50 outputs. The network quickly settles into the desired sinewave oscillation.

6.1.2 Second Example: a tuneable sinewave generator

We now make the sinewave generation task more difficult by demanding that the sinewave be adjustable in frequency. The training data now consists of an input signal $u(n)$, which sets the desired frequency, and an output $d(n)$, which is a sinewave whose frequency follows the input $u(n)$. Figure 6.5 shows the resulting network architecture and a short sequence of teacher input and output.

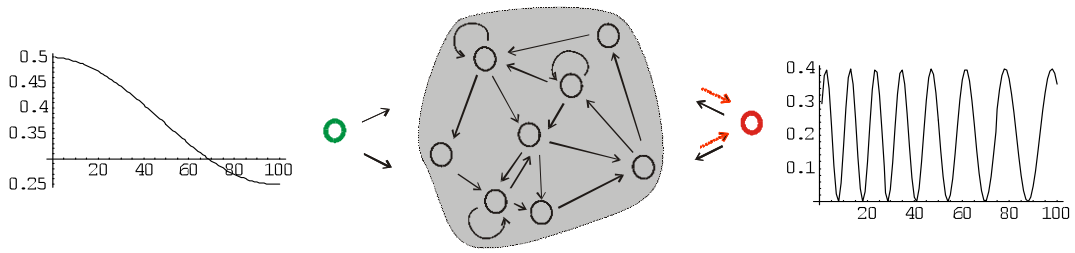

Figure 6.5: *Setup of tuneable sinewave generator task. Trainable connections appear as dotted red arrows, fixed connections as solid black arrows.*

Because the task is now more difficult, we use a larger DR with 100 units. In the sampling period, the network is driven by the teacher data. This time, this involves both inputting the slow signal $u(n)$, and teacher-forcing the desired output $d(n)$. We inspect the resulting activation patterns of internal units and find that they reflect, combine, and modify both $u(n)$ and $d(n)$ (Figure 6.6).

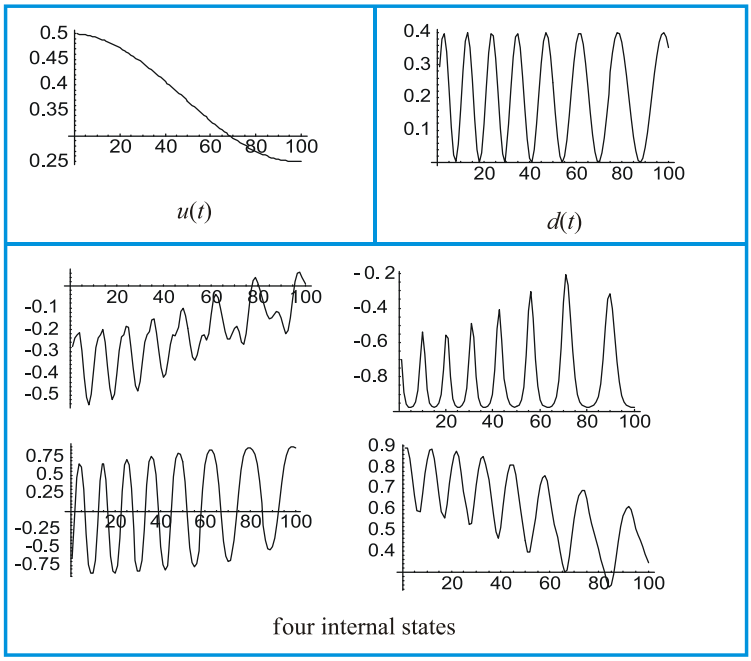

Figure 6.6: *Traces of some internal DR units during the sampling period in the tuneable frequency generation task.*

In this example we use a sigmoid output unit. In order to make that work, during sampling we collect into $\mathbf{T}$ not the raw desired output $d(n)$ but the transfer-inverted version $\tanh^{-1}(d(n))$. We also use a longer training sequence of 1200 steps of which

we discard the first 200 steps as initial transient. The training error which we minimize concerns the tanh-inverted quantities:

(6.5)

$$\mathrm{MSE}_{train} = 1/1000 \sum_{n=201}^{1000} (\tanh^{-1} d(n) - \tanh^{-1} y(n))^2 = 1/1000 \sum_{n201}^{1000} (\tanh^{-1} d(n) - \sum_{i=1}^{100} w_i^{out} x_i(n))^2$$

This is achieved, as previously, by computing $\mathbf{W}^{\mathrm{out}} = \mathbf{M}^{-1}\mathbf{T}$. The training error was found to be 8.1e-6, and the test error on the first 50 steps after inserting the computed output weights was 0.0006. Again, the trained network stably locks into the desired type of dynamics even from a random starting state, as displayed in Figure 6.7.

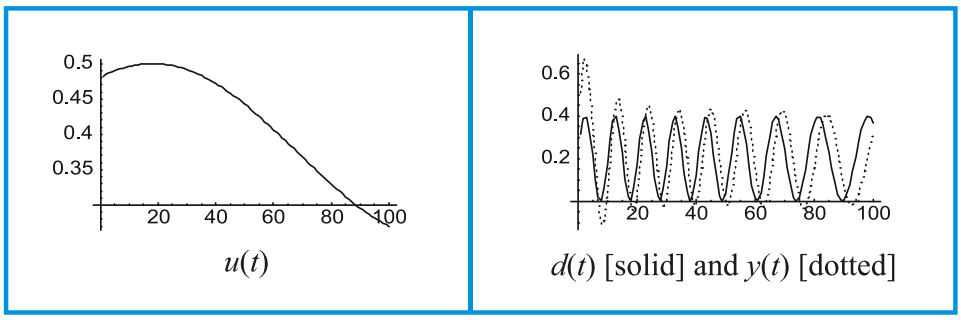

Figure 6.7 *Starting the trained generator from a random starting state.*

Stability of the trained network was not as easy to achieve here as in the previous example. In fact, a trick was used which was found empirically to foster stable solutions. The trick is to insert some noise into the network during sampling. That is, during sampling, the network was updated according to the following variant of (1.6):

(6.6) $$\mathbf{x}(n+1) = \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n) + \nu(n)),$$

where $\nu(n)$ is a small white noise term.

6.2 Training echo state networks: mathematics of echo states

In the two introductory examples, we rather vaguely said that the DR should exhibit a "damped" dynamics. We now describe in a rigorous way what kind of "damping" is required to make the ESN approach work, namely, that the DR must have *echo states*.

The key to understanding ESN training is the concept of *echo states*. Having echo states (or not having them) is a property of the network prior to training, that is, a property of the weight matrices $\mathbf{W}^{in}$, $\mathbf{W}$, and (optionally, if they exist) $\mathbf{W}^{back}$. The property is also relative to the type of training data: the same untrained network may have echo states for certain training data but not for others. We therefore require that

the training input vectors $\mathbf{u}(n)$ come from a compact interval U and the training output vectors $\mathbf{d}(n)$ from a compact interval D. We first give the mathematical definition of echo states and then provide an intuitive interpretation.

> **Definition 6.1** (echo states). Assume an untrained network with weights $\mathbf{W}^{in}$, $\mathbf{W}$, and $\mathbf{W}^{back}$ is driven by teacher input $\mathbf{u}(n)$ and teacher-forced by teacher output $\mathbf{d}(n)$ from compact intervals U and D. The network $(\mathbf{W}^{in}, \mathbf{W}, \mathbf{W}^{back})$ has *echo states* w.r.t. U and D, if for every left-infinite input/output sequence $(\mathbf{u}(n), \mathbf{d}(n\text{-}1))$, where $n = ..., -2,-1,0$, and for all state sequences $\mathbf{x}(n)$, $\mathbf{x}'(n)$ compatible with the teacher sequence, i.e. with
>
> $$\begin{aligned} \mathbf{x}(n+1) &= \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{d}(n)) \\ \mathbf{x}'(n+1) &= \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}'(n) + \mathbf{W}^{back}\mathbf{d}(n)) \end{aligned} \tag{6.7}$$
>
> it holds that $\mathbf{x}(n) = \mathbf{x}'(n)$ for all $n \leq 0$.

Intuitively, the echo state property says, "if the network has been run for a very long time [from minus infinity time in the definition], the current network state is uniquely determined by the history of the input and the (teacher-forced) output". An equivalent way of stating this is to say that for every internal signal $x_i(n)$ there exists an *echo function* e_i which maps input/output histories to the current state:

$$\begin{aligned} e_i &: (U \times D)^{-\mathbb{N}} \to \nabla \\ &(\ldots, (\mathbf{u}(-1), \mathbf{d}(-2)), (\mathbf{u}(0), \mathbf{d}(-1))) \mapsto x_i(0) \end{aligned} \tag{6.9}$$

We often say, somewhat loosely, that a (trained) network $(\mathbf{W}^{in}, \mathbf{W}, \mathbf{W}^{out}, \mathbf{W}^{back})$ is an echo state network if its untrained "core" $(\mathbf{W}^{in}, \mathbf{W}, \mathbf{W}^{back})$ has the echo state property w.r.t. input/output from *any* compact interval UxD.

Several conditions, which have been shown to be equivalent to echo states, are collected in Jaeger (2001a). We provide one for illustration.

> **Definition 6.2** (state contracting). With the same assumptions as in Def. 6.1, the network $(\mathbf{W}^{in}, \mathbf{W}, \mathbf{W}^{back})$ is *state contracting* w.r.t. U and D, if for all right-infinite input/output sequences $(\mathbf{u}(n), \mathbf{d}(n\text{-}1)) \in UxD$, where $n = 0,1,2,...$ there exists a null sequence $(\delta_n)_{n \geq 1}$, such that for all starting states $\mathbf{x}(0)$, $\mathbf{x}'(0)$ and for all $n>0$ it holds that $\| \mathbf{x}(n) - \mathbf{x}'(n) \| < \delta_n$, where $\mathbf{x}(n)$ [resp. $\mathbf{x}'(n)$] is the network state at time n obtained when the network is driven by $(\mathbf{u}(n), \mathbf{d}(n\text{-}1))$ up to time n after having been started in $\mathbf{x}(0)$, [resp. in $\mathbf{x}'(0)$].

Intuitively, the state forgetting property says that the effects on initial network state wash out over time. Note that there is some subtelty involved here in that the null sequence used in the definition depends on the the input/output sequence.

The echo state property is connected to algebraic properties of the weight matrix $\mathbf{W}$. Unfortunately, there is no known necessary and sufficient algebraic condition which allows one to decide, given $(\mathbf{W}^{in}, \mathbf{W}, \mathbf{W}^{back})$, whether the network has the echo state

property. For readers familiar with linear algebra, we quote here from Jaeger (2001) a sufficient condition for the *non*-existence of echo states.

> **Proposition 6.1** Assume an untrained network ($\mathbf{W}^{in}$, $\mathbf{W}$, $\mathbf{W}^{back}$) with state update according to (1.6) and with transfer functions $\tanh$. Let $\mathbf{W}$ have a spectral radius $|\lambda_{max}| > 1$, where $|\lambda_{max}|$ is the largest absolute value of an eigenvector of $\mathbf{W}$. Then the network has no echo states with respect to any input/output interval U x D containing the zero input/output ($\mathbf{0}$,$\mathbf{0}$).

At face value, this proposition is not helpful for finding echo state networks. However, in practice it was consistently found that when the condition noted in Proposition 6.1 is *not* satisfied, i.e. when the spectral radius of the weight matrix is smaller than unity, we *do* have an echo state network. For the mathematically adventurous, here is a conjecture which remains to be shown:

> **Conjecture 6.1** Let δ,ε be two small positive numbers. Then there exists a network size N, such that when an N –sized dynamical reservoir is randomly constructed by (1) randomly generating a weight matrix $\mathbf{W}_0$ by sampling the weights from a uniform distribution over $[-1,1]$, (2) normalizing $\mathbf{W}_0$ to a matrix $\mathbf{W}_1$ with unit spectral radius by putting $\mathbf{W}_1 = 1/|\lambda_{max}|\ \mathbf{W}_0$, where $|\lambda_{max}|$ is the spectral radius of $\mathbf{W}_0$, (3) scaling $\mathbf{W}_1$ to $\mathbf{W}_2 = (1-\delta)\ \mathbf{W}_1$, whereby $\mathbf{W}_2$ obtains a spectral radius of $(1-\delta)$, then the network ($\mathbf{W}^{in}$, $\mathbf{W}_3$, $\mathbf{W}^{back}$) is an echo state network with probability $1-\varepsilon$.

Note that both in Proposition 6.1 and Conjecture 6.1 the input and backprojection weights are not used for the claims. It seems that these weights are irrelevant for the echo state property. In practice, it is found that they can be freely chosen without affecting the echo state property. Again, a mathematical analysis of these observations remains to be done.

For practical purposes, the following procedure (also used in the conjecture) seems to guarantee echo state networks:

1. Randomly generate an internal weight matrix $\mathbf{W}_0$.
2. Normalize $\mathbf{W}_0$ to a matrix $\mathbf{W}_1$ with unit spectral radius by putting $\mathbf{W}_1 = 1/|\lambda_{max}|\ \mathbf{W}_0$, where $|\lambda_{max}|$ is the spectral radius of $\mathbf{W}_0$.
3. Scale $\mathbf{W}_1$ to $\mathbf{W} = \alpha\ \mathbf{W}_1$, where $\alpha < 1$, whereby $\mathbf{W}$ has a spectral radius of α.
4. Then, the untrained network ($\mathbf{W}^{in}$, $\mathbf{W}$, $\mathbf{W}^{back}$) is (or more precisely, has always been found to be) an echo state network, regardless of how $\mathbf{W}^{in}$, $\mathbf{W}^{back}$ are chosen.

The diligent choice of the spectral radius α of the DR weight matrix is of crucial importance for the eventual success of ESN training. This is because α is intimately connected to the intrinsic timescale of the dynamics of the DR state. Small α means that one has a fast DR, large α (i.e., close to unity) means that one has a slow DR. The intrinsic timescale of the task should match the DR timescale. For example, if one wishes to train a sine generator as in the example of Subsection 6.1.1, one should use a small α for fast sinewaves and a large α for slow sinewaves.

Note that the DR timescale seems to depend exponentially on $1-\alpha$, so e.g. settings of $\alpha = 0.99, 0.98, 0.97$ will yield an exponential speedup of DR timescale, not a linear one. However, these remarks rest only on empirical observations; a rigorous mathematical investigation remains to be carried out. An illustrative example for a fast task is given in Jaeger (2001, Section 4.2), where very fast "switching"-type of dynamics was trained with a DR whose spectral radius was set to 0.44, which is quite small considering the exponential nature of time scale dependence on α. Standard settings of α lie in a range between 0.7 and 0.98. The sinewave generator presented in Section 6.1.1 and the tuneable sinewave generator from Section 6.1.2 both used a DR with $\alpha = 0.8$.

Figure 6.8 gives a plot of the training log error $\log(\mathrm{MSE}_{train})$ of the sinewave generator training task considered in Section 6.1.1 obtained with different settings of α. It is evident that a proper setting of this parameter is crucial for the quality of the resulting generator network.

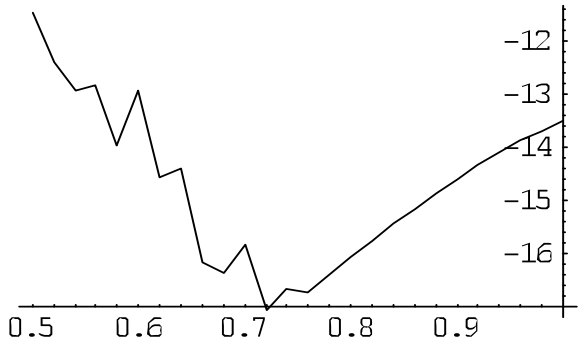

Figure 6.8: $\log_{10}(\mathrm{MSE}_{train})$ vs. spectral radius of the DR weight matrix for the sinewave generator experiment from Section 6.1.1.

6.3 Training echo state networks: algorithm

With the solid grasp on the echo state concept procured by Section 6.2, we can now give a complete description of training ESNs for a given task. In this description we assume that the output unit(s) are sigmoid units; we further assume that there are output-to-DR feedback connections. This is the most comprehensive version of the algorithm. Often one will use simpler versions, e.g. linear output units; no output-to-DR feedback connections; or even systems without input (such as the pure sinewave generator). In such cases, the algorithm presented below has to be adapted in obvious ways.

Given: A training input/output sequence $(\mathbf{u}(1), \mathbf{d}(1)), ..., (\mathbf{u}(T), \mathbf{d}(T))$.

Wanted: A trained ESN $(\mathbf{W}^{in}, \mathbf{W}, \mathbf{W}^{back}, \mathbf{W}^{out})$ whose output $\mathbf{y}(n)$ approximates the teacher output $\mathbf{d}(n)$, when the ESN is driven by the training input $\mathbf{u}(n)$.

Notes:

1. We can merely expect that the trained ESN approximates the teacher output well after initial transient dynamics have washed out, which are invoked by the (untrained, arbitrary) network starting state. Therefore, more precisely what we want is that the trained ESN approximates the teacher output for times $n = T_0, ..., T$, where $T_0 > 1$. Depending on network size and intrinsic timescale, typical ranges for T_0 are 10 (for small, fast nets) to 500 (for large, slow nets).
2. What we *actually* want is not primarily a good approximation of the teacher output, but more importantly, a good approximation of *testing* output data from independent test data sets generated by the same (unknown) system which also generated the teacher data. This leads to the question of how good *generalization* performance of a trained model can be ascertained, a far from trivial topic. This question is central to *statistical learning theory*, and we simply ignore it in this tutorial. However, if one wishes to achieve real-world problem solutions with blackbox models, it becomes mandatory to deal with this issue.

Step 1. Procure an untrained DR network ($\mathbf{W}^{in}$, $\mathbf{W}$, $\mathbf{W}^{back}$) which has the echo state property, and whose internal units exhibit mutually interestingly different dynamics when excited.

This step involves many heuristics. The way I proceed most often involves the following substeps.

1. Randomly generate an internal weight matrix $\mathbf{W}_0$.
2. Normalize $\mathbf{W}_0$ to a matrix $\mathbf{W}_1$ with unit spectral radius by putting $\mathbf{W}_1 = 1/|\lambda_{\max}| \mathbf{W}_0$, where $|\lambda_{\max}|$ is the spectral radius of $\mathbf{W}_0$. Standard mathematical packages for matrix operations all include routines to determine the eigenvalues of a matrix, so this is a straightforward thing.
3. Scale $\mathbf{W}_1$ to $\mathbf{W} = \alpha \mathbf{W}_1$, where $\alpha < 1$, whereby $\mathbf{W}$ obtains a spectral radius of α.
4. Randomly generate input weights $\mathbf{W}^{in}$ and output backpropagation weights $\mathbf{W}^{back}$. Then, the untrained network ($\mathbf{W}^{in}$, $\mathbf{W}$, $\mathbf{W}^{back}$) is (or more honestly, has always been found to be) an echo state network, regardless of how $\mathbf{W}^{in}$, $\mathbf{W}^{back}$ are chosen.

Notes:

1. The matrix $\mathbf{W}_0$ should be sparse, a simple method to encourage a rich variety of dynamics of different internal units. Furthermore, the weights should be roughly equilibrated, i.e. the mean value of weights should be about zero. I usually draw nonzero weights from a uniform distribution over $[-1, 1]$, or I set nonzero weights randomly to -1 or 1.
2. The size N of $\mathbf{W}_0$ should reflect both the length T of training data, and the difficulty of the task. As a rule of thumb, N should not exceed an order of magnitude of $T/10$ to $T/2$ (the more regular-periodic the training data, the closer to $T/2$ can N be chosen). This is a simple precaution against overfitting. Furthermore, more difficult tasks require larger N.
3. The setting of α is crucial for subsequent model performance. It should be small for fast teacher dynamics and large for slow teacher dynamics,

according to the observations made above in Section 6.2. Typically, α needs to be hand-tuned by trying out several settings.

4. The absolute size of input weights $\mathbf{W}^{in}$ is also of some importance. Large absolute $\mathbf{W}^{in}$ imply that the network is strongly driven by input, small absolute values mean that the network state is only slightly excited around the DR's resting (zero) state. In the latter case, the network units operate around the linear central part of the sigmoid, i.e. one obtains a network with an almost linear dynamics. Larger $\mathbf{W}^{in}$ drive the internal units closer to the saturation of the sigmoid, which results in a more nonlinear behavior of the resulting model. In the extreme, when $\mathbf{W}^{in}$ becomes very large, the internal units will be driven into an almost pure $-1 / +1$ valued, binary dynamics. Again, manual adjustment and repeated learning trials will often be required to find the task-appropriate scaling.
5. Similar remarks hold for the absolute size of weights in $\mathbf{W}^{back}$.

Step 2. Sample network training dynamics.

This is a mechanical step, which involves no heuristics. It involves the following operations:

1. Initialize the network state arbitrarily, e.g. to zero state $\mathbf{x}(0) = \mathbf{0}$.
2. Drive the network by the training data, for times $n = 0, ..., T$, by presenting the teacher input $\mathbf{u}(n)$, and by teacher-forcing the teacher output $\mathbf{d}(n\text{-}1)$, by computing

   (6.10) $$\mathbf{x}(n+1) = \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{d}(n))$$

3. At time $n = 0$, where $\mathbf{d}(n)$ is not defined, use $\mathbf{d}(n) = \mathbf{0}$.
4. For each time larger or equal than an initial washout time T_0, collect the concatenated input/reservoir/previous-output states $(\mathbf{u}(n)\ \mathbf{x}(n)\ \mathbf{y}(n-1))$ as a new row into a state collecting matrix $\mathbf{M}$. In the end, one has obtained a state collecting matrix of size $(T - T_0 + 1)$ x $(K + N + L)$.
5. Similarly, for each time larger or equal to T_0, collect the sigmoid-inverted teacher output $\tanh^{-1}\mathbf{d}(n)$ row-wise into a teacher collection matrix $\mathbf{T}$, to end up with a teacher collecting matrix $\mathbf{T}$ of size $(T - T_0 + 1)$ x L.

Note: Be careful to collect into $\mathbf{M}$ and $\mathbf{T}$ the vectors $\mathbf{x}(n)$ and $\tanh^{-1}\mathbf{d}(n)$, not $\mathbf{x}(n)$ and $\tanh^{-1}\mathbf{d}(n\text{-}1)$!

Step 3: Compute output weights.

1. Concretely, multiply the pseudoinverse of $\mathbf{M}$ with $\mathbf{T}$, to obtain a $(K + N + L)$ x L sized matrix $(\mathbf{W}^{out})^t$ whose i-th column contains the output weights from all network units to the i -th output unit:

   (6.11) $$(\mathbf{W}^{OUT})^t = \mathbf{M}^{-1}\mathbf{T}\ .$$

Every programming package of numerical linear algebra has optimized procedures for computing pseudoinverses.
2. Transpose $(\mathbf{W}^{out})^t$ to $\mathbf{W}^{out}$ in order to obtain the desired output weight matrix.

Step 4: Exploitation.

The network ($\mathbf{W}^{in}$, $\mathbf{W}$, $\mathbf{W}^{back}$, $\mathbf{W}^{out}$) is now ready for use. It can be driven by novel input sequences $\mathbf{u}(n)$, using the update equations (1.6) and (1.7), which we repeat here for convenience:

(1.6) $$\mathbf{x}(n+1) = \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n)),$$

(1.7) $$\mathbf{y}(n+1) = \mathbf{f}^{out}(\mathbf{W}^{out}(\mathbf{u}(n+1), \mathbf{x}(n+1), \mathbf{y}(n)).$$

Variants.

1. If stability problems are encountered when using the trained network, it very often helps to add some small noise during sampling, i.e. to use an update equation

   (6.12) $$\mathbf{x}(n+1) = \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{d}(n) + \nu(n)),$$

   where $\nu(n)$ is a small uniform white noise term (typical sizes 0.0001 to 0.01). The rationale behind this is explained in Jaeger 2001.
2. Often one does not desire to have trained output-to-output connections. In that case, collect during sampling only network states stripped off the output unit values, which will yield a $(K+N)$ x L sized matrix $\mathbf{W}^{out}$. In exploitation, instead of (1.7) use

   (1.7') $$\mathbf{y}(n+1) = \mathbf{f}^{out}(\mathbf{W}^{out}(\mathbf{u}(n+1), \mathbf{x}(n+1)).$$

3. When the system to be modeled is highly nonlinear, better models are sometimes obtained by using "augmented" network states for training and exploitation. This means that in addition to the original network states $\mathbf{x}(n)$, nonlinear transformations thereof are also included into the game. A simple case is to include the squares of the original state. During sampling, instead of collecting pure states $\mathbf{x}(n) = (x_1(n), ..., x_{K+N+L}(n))$, collect the double-sized "augmented" states $(\mathbf{x}(n), \mathbf{x}^2(n)) = (x_1(n), ..., x_{K+N+L}(n), x_1^2(n), ..., x_{K+N+L}^2(n))$ into the state collecting matrix. This will yield an output weight matrix $\mathbf{W}^{out}$ of size L x $2(K+N+L)$. During exploitation, instead of (1.7) use

   (1.7'') $$\mathbf{y}(n+1) = \mathbf{f}^{out}(\mathbf{W}^{out}(\mathbf{u}(n+1), \mathbf{x}(n+1), \mathbf{y}(n), \mathbf{u}^2(n+1), \mathbf{x}^2(n+1), \mathbf{y}^2(n)).$$

   The paper (Jaeger 2002) uses augmented states.

6.4 Why echo states?

Why must the DR have the echo state property to make the approach work?

From the perspective of systems engineering, the (unknown) system's dynamics is governed by an update equation of the form

(6.13) $$\mathbf{d}(n) = e(\mathbf{u}(n), \mathbf{u}(n-1), ..., \mathbf{d}(n-1), \mathbf{d}(n-2)),$$

where e is a (possibly highly complex, nonlinear) function of the previous inputs and system outputs. (6.13) is the most general possible way of describing a deterministic, stationary system. In engineering problems, one typically considers simpler versions, for example, where e is linear and has only finitely many arguments (i.e., the system has finite memory). Here we will however consider the fully general version (6.13).

The task of finding a black-box model for an unknown system (6.13) amounts to finding a good approximation to the system function *e*. We will assume an ESN with linear output units to facilitate notation. Then, the network output of the trained network is a linear combination of the network states, which in turn are governed by the echo functions, see (6.9). We observe the following connection between (6.13) and (6.9):

$$\begin{aligned} & e(\mathbf{u}(n), \mathbf{u}(n-1), ..., \mathbf{d}(n-1), \mathbf{d}(n-2)) = \\ & \quad = \mathbf{d}(n) \\ & \quad \approx \mathbf{y}(n) \\ & \quad = \sum w_i^{out} x_i(n) \\ & \quad = \sum w_i^{out} e_i(\mathbf{u}(n), \mathbf{u}(n-1), ..., \mathbf{y}(n-1), \mathbf{y}(n-2)) \end{aligned} \tag{6.14}$$

It becomes clear from (6.14) how the desired approximation of the system function e can be interpreted as a linear combination of echo functions e_i. This transparent interpretatation of the system approximation task directly relies on the interpretation of network states as echo states. The arguments of e and e_i are identical in nature: both are collections of previous inputs and system (or network, respectively) outputs. Without echo states, one could neither mathematically understand the relationship between network output and original system output, nor would the training algorithm work.

6. 5 Liquid state machines

An approach very similar to the ESN approach has been independently explored by Wolfgang Maass et al. at Graz Technical University. It is called the "liquid state machine" (LSM) approach. Like in ESNs, large recurrent neural networks are conceived as a reservoir (called "liquid" there) of interesting excitable dynamics, which can be tapped by trainable readout mechanisms. LSMs compare with ESNs as follows:

- LSM research focuses on modeling dynamical and representational phenomena in biological neural networks, whereas ESN research is aimed more at engineering applications.
- The "liquid" network in LSMs is typically made from biologically more adequate, spiking neuron models, whereas ESNs "reservoirs" are typically made up from simple sigmoid units.
- LSM research considers a variety of readout mechanisms, including trained feedforward networks, whereas ESNs typically make do with a single layer of readout units.

An introduction to LSMs and links to publications can be found at http://www.lsm.tugraz.at/.

7. Short term memory in ESNs

Many time-series processing tasks involve some form of *short term memory* (STM). By short-term memory we understand the property of some input-output systems, where the current output $\mathbf{y}(n)$ depends on earlier values $\mathbf{u}(n\text{-}k)$ of the input and/or earlier values $\mathbf{y}(n\text{-}k)$ of the output itself. This is obvious, for instance, in speech processing. Engineering tasks like suppressing echos in telephone channels or the control of chemical plants with attenuated chemical reactions require system models with short-term memory capabilities.

We saw in Section 6 that the DR unit's activations $x_i(n)$ can be understood in terms of echo functions e_i which maps input/output histories to the current state. We repeat the corresponding Equation (6.9) here for convenience:

$$e_i : (U \times D)^{-\mathbb{N}} \to \nabla$$
$$(\ldots, (\mathbf{u}(-1), \mathbf{d}(-2)), (\mathbf{u}(0), \mathbf{d}(-1))) \mapsto x_i(0) \tag{6.9}$$

The question which we will now investigate more closely is how many of the previous inputs/output arguments $(\mathbf{u}(n\text{-}k), \mathbf{y}(n\text{-}k\text{-}1))$ are actually relevant for the echo function? or asked in other words, how long is the effective short-term memory of an ESN?

A good intiuitive grasp on this issue is important for successful practical work with ESNs because as we will see, with a suitable setup of the DR, one can control to some extent the short-term memory characteristics of the resulting ESN model.

We will provide here only an intuitive introduction; for a more detailed treatment consult the technical report devoted to short-term memory (Jaeger 2001a) and the mathematically-oriented Master thesis (Bertschinger 2002).

7.1 First example: training an ESN as a delay line

Much insight into the STM of ESNs can be gained when we train ESNs on a pure STM task. We consider an ESN with a single input channel and many output channels. The input $u(n)$ is a white noise signal generated by sampling at each time independently from a uniform distribution over $[-0.5, 0.5]$. We consider delays $k = 1, 2, \ldots$. For each delay k, we train a separate output unit with the training signal $d_k(n) = u(n-k)$. We do not equip our network with feedback connections from the output units to the DR, so all output units can be trained simultaneously and independently from each each other. Figure 7.1 depicts the setup of the network.

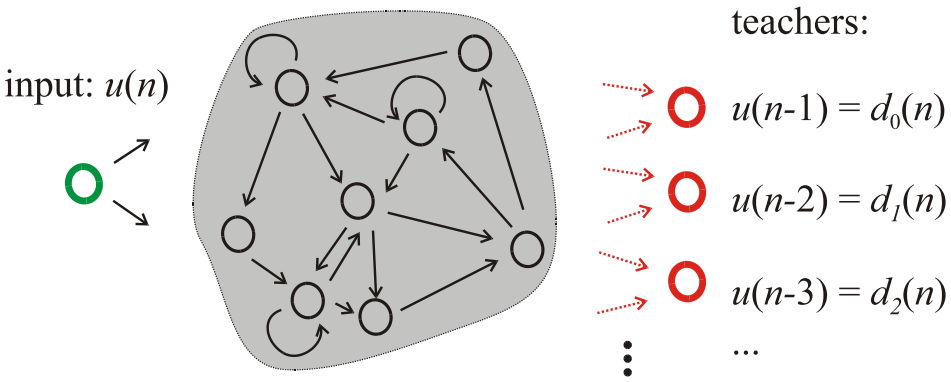

Figure 7.1: *Setup of delay learning task.*

Concretely, we use a 20-unit DR with a connectivity of 15%, that is, 15% of the entries of the weight matrix $\mathbf{W}$ are non-null. The non-null weights were sampled randomly from a uniform distribution over $[-1,1]$, and the resulting weigth matrix was rescaled to a spectral radius of $\alpha = 0.8$, as described in Section 6.2. The input weights were put to values of -0.1 or $+0.1$ with equal probability. We trained 4 output units with delays of $k = 4, 8, 16, 20$. The training was done over 300 time steps, of which the first 100 were discarded to wash out initial transients. On test data, the trained network showed testing mean square errors of $\mathrm{MSE}_{test} = 0.0000047, 0.00070, 0.040, 0.12$ for the four trained delays. Figure 7.2 (upper diagrams) shows an overlay of the correct delayed signals (solid line) with the trained network output.

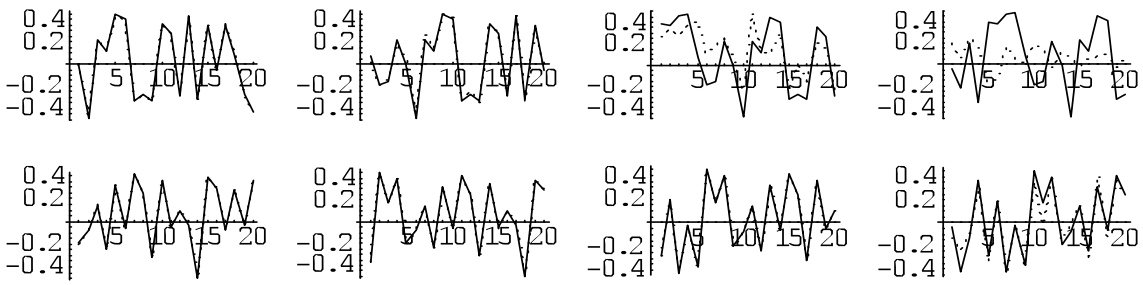

Figure 7.2: *Results of training delays $k = 4, 8, 16, 20$ with a 20-unit DR. Top row: input weights of size -0.1 or $+0.1$, bottom row: input weights sized -0.001 or $+0.001$.*

When the same experiment is redone with the same DR, but with much smaller input weights set to random values of -0.001 or $+0.001$, the performance greatly improves: testing errors $\mathrm{MSE}_{test} = 0.000035, 0.000038, 0.000034, 0.0063$ are now obtained.

Three fundamental observations can be gleaned from this simple example:

1. The network can master the delay learning task, which implies that the current network state $\mathbf{x}(n)$ retains extractable information about previous inputs $u(n-k)$.
2. The longer the delay, the poorer the delay learning performance.
3. The smaller the input weights, the better the performance.

7.2 Theoretical insights

I now report some theoretical findings (from Jaeger 2001a), which explain the observations made in the previous subsection.

First, we need a precise version of the intuitive notion of "network performance for learning the k-delay task". We consider the correlation coefficient $r(u(n-k), y_k(n))$ between the correct delayed signal $u(n-k)$ and the network ouput $y_k(n)$ of the unit trained on the delay k. It ranges between -1 and 1. By squaring it, we obtain a quantity called in statistics the *determination coefficient* $r^2(u(n-k), y_k(n))$. It ranges between 0 and 1. A value of 1 indicates perfect correlation between correct signal and network output,k a value of 0 indicates complete loss of correlation. (In statistical terms, the determination coefficient gives the proportion of variance in one signal explained by the other). Perfect recall of the k –delayed signal thus would be indicated by $r^2(u(n-k), y_k(n)) = 1$, complete failure by $r^2(u(n-k), y_k(n)) = 0$.

Next, we define the overall delay recalling performance of a network, as the sum of this coefficient over all delays. We define the *memory capacity* MC of a network by

$$MC = \sum_{k=1}^{\infty} r^2(u(n-k), y_k(n)) \tag{7.1}$$

Without proof, we cite (from Jaeger 2001a) some fundamental results concerning the memory capacity of ESNs:

Theorem 7.1. In a network whose DR has N nodes, $MC \leq N$. That is, the maximal possible memory capacity is bounded by DR size.

Theorem 7.2. In a linear network with N nodes, generically $MC = N$. That is, a linear network will generically reach maximal network capacity. Notes: (i) a linear network is a network whose internal units have a linear transfer function, i.e. $f = \mathbf{id}$. (ii) "Generically" means: if we randomly construct such a network, it will have the desired property with probability one.

Theorem 7.3. In a linear network, long delays can never be learnt better than short delays ("monotonic forgetting")

When we plot the determination coefficient against the delay, we obtain the *forgetting curves* of an ESN. Figure 7.3 shows some forgetting curves obtained from various 400-unit ESNs.

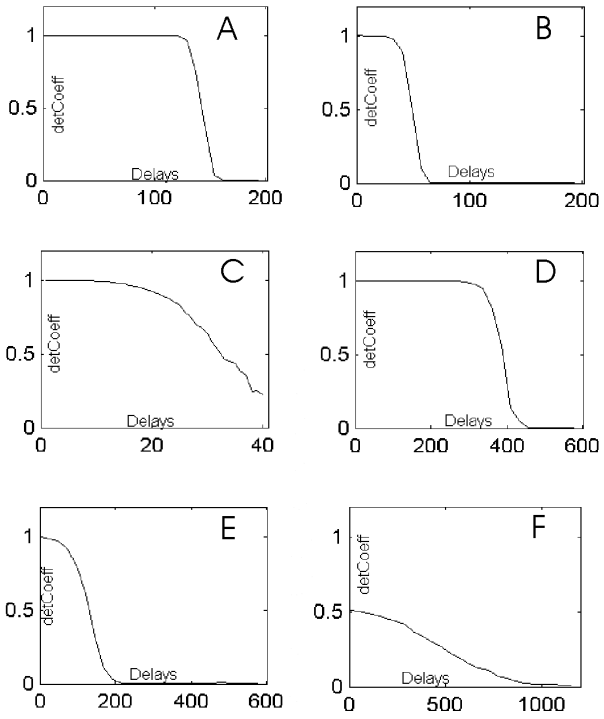

Figure 7.3: *Forgetting curves of various 400-unit networks. A: randomly created linear DR. B: randomly created sigmoid DR. C: like A, but with noisy state update of DR. D: almost unitary weight matrix, linear update. E: same as D, but with noisy state update. F: same as D, but with spectral radius* $\alpha = 0.999$*. Mind the different scalings of the x-axis!*

The forgetting curves in Figure 7.3 exhibit some interesting phenomena:

- According to theorem 7.2, the forgetting curve in curve A should reflect a memory capacity of 400 (= network size). That is, the area under the curve should be 400. However, we find an area (= memory capacity) of about 120 only. This is due to rounding errors in the network update. The longer the delay, the more severe the effect of accumulated rounding errors, which reduce the effectively achievable memory capacity.
- The curve B was generated with a DR made from the same weight matrix as in A, but this time, the standard sigmoid transfer funtion $\tanh$ was used for network update. Compared to A, we observe a drop in memory capacity. It is a general empirical observation that the more nonlinear a network, the lower its memory capacity. This also explains the finding from Section 7.1, namely, that the STM of a network is improved when the input weights are made very

small. Very small input weights make the input drive the network only minimally around its zero resting state. Around the zero state, the sigmoids are almost linear. Thus, small input weights yield an almost linear network dynamics, which is good for its memory capacity.

- Curve C was generated like A, but the (linear) network was updated with a small noise term added to the states. As can be expected, this decreases the memory capacity. What is worth mentioning is that the effect is quite strong. An introductory discussion can be found in Jaeger (2001a), and a detailed analytical treatment is given in Bertschinger (2002).
- The forgetting curve D comes close to the theoretical optimum of $MC = 400$. The trick was to use an almost unitary weight matrix, again with linear DR units. Intuitively, this means that a network state $\mathbf{x}(n)$, which carries the information about the current input, "revolves" around the state space $\mathbb{R}^N$ without interaction with succeeding states, for N update steps. There is more about this in Jaeger (2001a) and Bertschinger (2002).
- The forgetting curve E was obtained from the same linear unitary network as D, but noise was added to state update. The corruption of memory capacity is less dramatic as in curve C.
- Finally, the forgetting curve F was obtained by scaling the (linear, unitary) network from D to a spectral radius $\alpha = 0.999$. This leads to long-term "reverberations" of input. On the one hand, this yields a forgetting curve with a long tail – in fact, it extends far beyond the value of the network size, $N = 400$. On the other hand, "reverberations" of long-time past inputs still occupying the present network state lead to poor recall of even the immediately past input: the forgetting curve is only about 0.5 right at the beginning. The area under the forgetting curve, however, comes close to the theoretical optimum of 400.

For practical purposes, when one needs ESNs with long STM effects, on can resort to a combination of the following approaches:

- Use large DRs. This is the most efficient and generally applicable method, but it requires sufficiently large training data sets.
- Use small input weights, to work in the almost linear working range of the network. This might conflict with nonlinear task characteristics.
- Use linear update for the DR. Again, might conflict with nonlinear task characteristics.
- Use specially prepared DRs with almost unitary weight matrices.
- Use a spectral radius α close to 1. This would work only with "slow" tasks (for instance, it would not work if one wants to have fast oscillating dynamics with long STM effects).

8. ESNs with leaky integrator neurons

The ESN approach is not confined to standard sigmoid networks. The basic idea of a dynamical reservoir works with any kind of dynamical system which has the echo state property. Other kind of systems which one might consider are, for instance,

- neural network with biologically more adequate, spiking neuron models (as in most of the research on the related "liquid state machines" of Maass et al. http://www.lsm.tugraz.at/.),
- continuous-time neural networks described by differential equations,
- coupled map lattices,
- excitable media models and reaction-diffusion systems.

More or less for historical (random) reasons, the ESN approach has so far been worked out almost exclusively using standard sigmoid networks. However, one disadvantage of these networks is that they do not have a time constant – their dynamics cannot be "slowed down" like the dynamics of a differential equation, if one would wish to do so. It is, for instance, almost impossible with standard sigmoid networks to obtain ESN generators of very slow sinewaves. In this section, we will consider ESNs made from a continous-time neuron model, *leaky integrator neurons*, which incorporate a time constant and which can be slowed down. This model was first used in an ESN context in Jaeger 2001.

8.1 The neuron model

The continuous-time dynamics $x(t)$ of a leaky integrator neuron is described by the differential equation

$$\dot{x} = \frac{1}{\tau}\left(-ax + f(\mathbf{w}\,\mathbf{x})\right), \tag{8.1}$$

where $\mathbf{w}$ is the weight vector of connections from all units $\mathbf{x}$ which feed into neuron x, and f is the neuron's output nonlinearity (typically a sigmoid; we will use $\tanh$ again). The positive quantity τ is the *time constant* of this equation – the larger, the slower the (otherwise identical) resulting dynamics. The term $-ax$ models a decaying potential of the neuron; by virtue of this term, the neuron retains part of its previous state. The nonnegative constant a is the neuron's *decay constant*. The larger a, the faster the decay of previous state, and the larger the relative influence of input from other neurons. If an entire DR of an ESN is made from leaky integrator neurons x_i with decay constants a_i, we obtain the following DR dynamics:

$$\dot{\mathbf{x}} = \frac{1}{\tau}\left(-\mathbf{A}\mathbf{x} + \mathbf{f}(\mathbf{W}^{in}\mathbf{u} + \mathbf{W}\,\mathbf{x} + \mathbf{W}^{back}\mathbf{y})\right) \tag{8.2}$$

where $\mathbf{A}$ is a diagonal matrix with the decay constants on its diagonal. In order to sumulate (8.2) on digital machines, we must discretize it by some stepsize δ. An approximate discretized version, which gives δ-time increments of the DR state for each discrete simulation step n, reads like this:

$$\mathbf{x}(n+1) = (\mathbf{I} - \frac{\delta}{\tau}\mathbf{A})\,\mathbf{x}(n) + \frac{\delta}{\tau}\mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\,\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n)), \tag{8.3}$$

where $\mathbf{I}$ is the identity matrix. For numerical stability reasons, δ, τ, $\mathbf{A}$ must be chosen such that the matrix $\mathbf{I} - \delta/\tau\, \mathbf{A}$ has no entry of absolute value greater or equal to 1. In the following we consider only the case $\delta = \tau = 1$, whereby (8.3) simplifies to

(8.4) $$\mathbf{x}(n+1) = (\mathbf{I} - \mathbf{A})\,\mathbf{x}(n) + \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\,\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n)).$$

By substituting $1 - a_i$ with the "retainment rate" $r_i = 1 - a_i$, we arrive at the more convenient version

(8.5) $$\mathbf{x}(n+1) = \mathbf{R}\,\mathbf{x}(n) + \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\,\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n)),$$

where $\mathbf{R}$ is a diagonal matrix with the retainment rates. The retainment rates must range between $0 \le r_i < 1$. A value of zero means that the unit does not retain any information about its previous state; the unit's dynamics then degenerates to the kind of standard sigmoid units which we used in previous sections. A value close to 1 means that the unit's dynamics is dominated by retaining (most of) its previous state.

Choosing appropriate values for $\mathbf{R}$ and $\mathbf{W}$ becomes a more subtle task than choosing $\mathbf{W}$ in standard sigmoid DRs. Two effects have to be considered: the echo state property must be ensured, and the absolute values of resulting network states should be kept small enough. We consider both effects in turn, in reverse order.

Small enough network states. Because of the (leaky) integration nature of (8.4), activation states of arbitrary size can build up. These are fed into the sigmoid $\mathbf{f}$ by virtue of the second term in (8.5). $\mathbf{f}$ can easily be driven into saturation due to the integration-buildup of state values. Then, the second term in (8.5) will yield essentially a binary, $[-1, 1]$-valued contribution, and the network dynamics will tend to oscillate. A way to avoid this condition is to use a weight matrix $\mathbf{W}$ with small values. Since the tendency to grow large activation states is connected to the retainment rates (larger rates lead to stronger growth), an ad-hoc counter-measure is to "renormalize" states by multiplication with $\mathbf{I} - \mathbf{R}$ before feeding them into $\mathbf{f}$. That is, one uses

(8.6) $$\mathbf{x}(n+1) = \mathbf{R}\,\mathbf{x}(n) + \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + (\mathbf{I} - \mathbf{R})\mathbf{W}\,\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n))$$

instead of (8.5).

Echo state property. It can be shown (a simple linearization argument, see Jaeger 2001) that the echo state property does *not* hold if the matrix $(\mathbf{I} - \mathbf{R})\mathbf{W} + \mathbf{R}$ has a spectral radius greater or equal to 1 (this refers to an update according to (8.6); if (8.5) is used, consider the spectral radius of $\mathbf{W} + \mathbf{R}$ instead). Conversely, and similarly as discussed in Section 6 (discussion of Proposition 6.1), empirically it has always been found that the echo state property *does* hold when $(\mathbf{I} - \mathbf{R})\mathbf{W} + \mathbf{R}$ has a spectral radius smaller than 1.

8.2 Example: slow sinewave generator

Like in Section 6.1, we consider the task of generating a sinewave, using an ESN with a single output unit and no input. This time, however, the desired sinewave is very slow: we want $d(n) = 1/5 \sin(n/100)$. This is hard to impossible to achieve with standard sigmoid ESNs, but easy with a leaky integrator network.

We select again a 20-unit DR, whose weight matrix has sparse connectivity of 20% and which is scaled to a spectral radius of 0.2. We choose a retainment rate of 0.98 for all units. Output feedback connection weights are sampled from a uniform distribution over $[-1, 1]$. Training is run over 4000 steps, of which the first 2000 are discarded. A long initial washout is required here because the retainment rates close to 1 imply a long initial transient – after all, we are dealing with slow dynamics here (2000 steps of sampling boil down to roughly 3 cycles of the sinewave).

This task is found to easily lead to unstable solutions. Some hand-tuning is necessary to find appropriate scaling ranges for $\mathbf{R}$ and $\mathbf{W}$. Additionally, it is essential that during sampling a small noise term (of size 0.000001) is added to network states. With the settings mentioned above and this noise term, the training produced 10 stable solutions in 10 runs with different, randomly generated weight matrices. The test MSEs obtained on a 2000 step test sequence ranged between $1.8e\text{-}6$ and $3.0e\text{-}7$.

9. Tricks of the trade

The basic idea of ESNs for black-box-modeling can be condensed into the following statement:

"Use an excitable system [the DR] to give a high-dimensional dynamical representation of the task input dynamics and/or output dynamics, and extract from this reservoir of task-related dynamics a suitable combination to make up the desired target signal."

Obviously, the success of the modeling task depends crucially on the nature of the excited dynamics – it should be adapted to the task at hand. For instance, if the target signal is slow, the excited dynamics should be slow, too. If the target signal is very nonlinear, the excited dynamics should be very nonlinear, too. If the target signal involves long short-term memory, so should the excited dynamics. And so forth.

Successful application of the ESN approach, then, involves a good judgement on important characteristics of the dynamics excited inside the DR. Such judgement can only grow with the experimenter's personal experience. However, a number of general practical hints can be given which will facilitate this learning process. All hints refer to standard sigmoid networks.

Plot internal states

Since the dynamics within the DR is so essential for the task, you should always visually inspect it. Plot some of the internal units $x_i(n)$ during sampling and/or testing. These plots can be very revealing about the cause of failure. If things don't go well, you will frequently observe in these plots one or two of the following misbehaviors:

- **Fast oscillations**. In tasks where you don't want fast oscillations, this observation indicates a too large spectral radius of the weight matrix $\mathbf{W}$, and/or too large values of the output feedback weights (if they exist). Remedy: scale them down.
- **Almost saturated network states**. Sometimes you will observe that the DR units almost always take extreme values near 1 or -1. This is caused by a large impact of incoming signals (input and/or output feedback). It is only desirable when you want to achieve some almost binary, "switching" type of target dynamics. Otherwise it's harmful. Remedy: scale down the input and/or output feedback weights.

Plot output weights

You should always inspect the output weights obtained from the learning procedure. The easiest way to do this is to plot them. They should not become too large. Reasonable absolute values are not greater than, say, 50. If the learnt output weights are in the order of 1000 and larger, one should attempt to bring them down to smaller ranges. Very small values, by contrast, do not indicate anything bad.

When judging the size of output weights, however, you should put them into relation with the range of DR states. If the DR is only minimally excited (let's say, DR unit activations in the range of 0.005 – this would for instance make sense in almost linear tasks with long-term memory characteristics), and if the desired output has a range up to 0.5, then output weights have to be around 100 just in order to scale up from the internal state range to the output range.

If after factoring out the range-adaptation effect just mentioned, the output weights still seem unreasonably large, you have an indication that the DR's dynamics is somehow badly adapted to your learning task. This is because large output weights imply that the generated output signal exploits subtle differences between the DR unit's dynamics, but does not exploit the "first order" phenomena inside the DR (a more mathematical treatment of large output weights can be found in Jaeger (2001a)).

It is not easy to suggest remedies against too large output weights, because they are an indication that the DR is generally poorly matched with the task. You should consider large output values as a symptom, not as the cause of a problem. Good doctors do not cure the symptoms, but try to address the cause.

Large output values will occur only in high-precision tasks, where the training material is mathematical in origin and intrinsically very accurate. Empirical training data will mostly contain some random noise component, which will lead to reasonably scaled output weights anyway. Adding noise during training is a safe

method to scale output weights down, but is likely to impair the desired accuracy as well.

Find a good spectral radius

The single most important knob to tune an ESN is the spectral radius of the DR weight matrix. The general rule: for fast, short-memory tasks use small α, for slow, long-memory tasks use large α. Manual experimentation will be necessary in most cases. One does not have to care about finding the precise best value for α, however. The range of optimal settings is relatively broad, so if an ESN works well with $\alpha = 0.8$, it can also be expected to work well with $\alpha = 0.7$ and with $\alpha = 0.85$. The closer you get to 1, the smaller the region of optimality.

Find an appropriate model size

Generally, with larger DR one can learn more complex dynamics, or learn a given dynamics with greater accuracy. However, beware of overfitting: if the model is too powerful (i.e. the DR too large), irrelevant statistical fluctuations in the training data will be learnt by the model. That leads to poor generalization on test data. Try increasing network sizes until performance on test data deteriorates.

The problem of overfitting is particularly important when you train on empirical, noisy data. It is not theoretically quite clear (at least not to me) whether the concept of overfitting also carries over to non-statistical, deterministic, 100%-precisely defined training tasks, for example training a chaotic attractor described by a differential equation (as in Jaeger 2001). The best results I obtained in that task were achieved with a 1000-unit network trained on 2000 data points, which means that 1000 parameters were estimated from 2000 data points. For empirical, statistical tasks, this would normally lead to overfitting (a rule of thumb in statistical learning is to have at least 10 data points per estimated parameter).

Add noise during sampling

When you are training an ESN with output feedback from accurate (mathematical, noise-free) training data, stability of the trained network is often difficult to achieve. A method that works wonders is to inject noise into the DR update during sampling, as described in Section 6, Eqn. (6.6). It is not clearly understood why this works. Attempts at an explanation are made in Jaeger (2001).

In tasks with empirical, noisy training data, noise insertion does not a priori make sense. Nor is it required when there are no output feedback connections.

There is one situation, however, where noise insertion might make sense even with empirical data and without output feedback connections. That is when the learnt model overfits data, which is revealed by a small training and a large test error. In this condition, injection of extra noise works as a *regularizer* in the sense of statistical learning theory. The training error will increase, but the test error will go down. However, a more appropriate way to avoid overfitting is to use smaller networks.

Use an extra bias input

When the desired output has a mean value that departs from zero, it is a good idea to invest an extra input unit and feed it with a constant value ("bias") during training and testing. This bias input will immediately enable the training to set the trained output to the correct mean value.

A relatively large bias input will shift many internal units towards one of the extremer outer ranges of their sigmoids; this might be advisable when you want to achieve a strongly nonlinear behavior.

Sometimes you do not want to affect the DR strongly by the bias input. In such cases, use a small value for the bias input (for instance, a value of 0.01), or connect the bias only to the output (i.e., put all bias-to-DR connections to zero).

Beware of symmetric input

Standard sigmoid networks are "symmetric" devices in the sense that when an input sequence $u(n)$ gives an output sequence $y(n)$, then the input sequence $-u(n)$ will yield an output sequence $-y(n)$. For instance, you can never train an ESN to produce an output $y(n) = u(n)^2$ from an input $u(n)$ which takes negative and positive values. There are two simple methods to succeed in "asymmetric" tasks:

- **Feed in an extra constant bias input.** This will effectively render the DR an unsymmetric device.
- **Shift the input.** Instead of using the original input signal, use a shifted version which only takes positive sign.

The symmetric-input fallacy comes in many disguises and is often not easy to recognize. Generally be cautious when the input signal has a range that is roughly symmetrical around zero. It almost never harms to shift it into an asymmetrical range. Nor does a small bias input usually harm.

Shift an scale input

You are free to transform the input into any value range $[a, b]$ by scaling and/or shifting it. A rule I work with: the more nonlinear the task, the more extravagantly I shift the input range. For instance, in a difficult nonlinear system identification task (30th order NARMA system) I once got best models with an input range $[a, b] = [3, 3.5]$. The apparent reason is that shifting the input far away from zero made the DR work in a highly nonlinear range of its sigmoid units.

Blackbox modeling generals

Be aware of the fact that ESN is a blackbox modeling technique. This means that you cannot expect good results on test data which operate in a working range that was never visited during training. Or to put it the other way round, make sure that your training data visit all the working conditions that you will later meet when using the trained model.

This is sometimes not easy to satisfy with nonlinear systems. For instance, a typical approach to obtain training data is to drive the empirical system with white noise input. This approach is well-motivated with linear systems, where white noise input in training data generally reveals the most about the system to be identified. However, this may not be the case with nonlinear systems! By contrast, what typically happens is that white noise input keeps the nonlinear system in a small subregion of its working range. When the system (the original system or the trained model) is later driven with more orderly input (for instance, slowly changing input), it will be driven into a quite different working range. A black-box model trained from data with white noise input will be unable to work well in another working range.

On the other hand, one should also not cover in the training data more portions of the working range than will be met in testing / exploitation. Much of the modeling capacity of you model will then be used up to model those working regions which are later irrelevant. As a consequence, the model accuracy in the relevant working regions will be poorer.

The golden rule is: use basically the same kind of input during training as you will later encounter in testing / exploitation, but make the training input a bit more varied than you expect the input in the exploitation phase.

Acknowledgment. The author is indebted to Stéphane Beauregard for proofreading, to Danil Prokhorov for many insights into the art of RNN training, and to Wolfgang Maass for a highly valued professional collaboration.

References

➡ An up-to-date repository ESN publications and computational resources can be found at http://www.faculty.iu-bremen.de/hjaeger/esn_research.html.

➡ A repository of publications concerning the related "liquid state machine" approach can be found at http://www.lsm.tugraz.at/references.html.

Bengio, Y., Simard, P. and P. Frasconi, *Learning long-term dependencies with gradient descent is difficult.* IEEE Transactions on Neural Networks 5, 1994, 157-166.

N. Bertschinger (2002), Kurzzeitspeicher ohne stabile Zustände in rückgekoppelten neuronalen Netzen. Diplomarbeit, Informatik VII, RWTH Aachen, 2002 http://www.igi.tugraz.at/nilsb/publications/DABertschinger.pdf

Dos Santos, E.P. and von Zuben, F.J. (2000) *Efficient second-order learning algorithms for discrete-time recurrent neural networks*. In: Medsker, L.R. and Jain,

L.C. (eds), Recurrent Neural Networsk: Design and Applications, 2000, 47-75. CRC Press: Boca Raton, Florida

Doya, K. (1992), *Bifurcations in the learning of recurrent neural networks*. Proceedings of 1992 IEEE Int. Symp. on Circuits and Systems vol. 6, 1992, 2777-2780.

Doya, K. (1995) *Recurrent networks: supervised learning*. In: Arbib, M.A. (ed.), The Handbook of Brain Theory and Neural Networks, 1995, 796-800. MIT Press / Bradford Books

Feldkamp, L. A., Prokhorov, D., Eagen, C.F., and F. Yuan (1998) *Enhanced multi-stream Kalman filter training for recurrent networks.* In: XXX (ed.), Nonlinear modeling: advanced black-box techniques, 1998, 29-53. Boston: Kluwer

H. Jaeger (2001), *The "echo state" approach to analysing and training recurrent neural networks*. GMD Report 148, GMD - German National Research Institute for Computer Science, 2001, http://www.faculty.iu-bremen.de/hjaeger/pubs/EchoStatesTechRep.pdf

H. Jaeger (2001a), *Short term memory in echo state networks*. GMD Report 152, GMD - German National Research Institute for Computer Science, 2002, http://www.faculty.iu-bremen.de/hjaeger/pubs/STMEchoStatesTechRep.pdf

H. Jaeger (2002), H. Jaeger, Adaptive nonlinear system identification with echo state networks. To appear in the Proceedings of NIPS 02, 2002 (8 p.). http://www.faculty.iu-bremen.de/hjaeger/pubs/esn_NIPS02

Puskorius, G.V. and L. A. Feldkamp (1994) *Neurocontrol of nonlinear dynamical systems with Kalman filter trained recurrent networks*. IEEE Transactions on Neural Networks 5(2), 1994, 279-297

Schraudolph, N. (2002). *Fast curvature matrix-vector products for second-order gradient descent.* To appear in Neural Computation. Manuscript online at http://www.icos.ethz.ch/~schraudo/pubs/#mvp.

Singhal, S. and L. Wu (1989), *Training multilayer perceptrons with the extended Kalman algorithm*. In D.S. Touretzky (ed.), Advances in Neural Information Processing Systems 1, 1989, 133-140. San Mateo, CA: Morgan Kaufmann

Werbos, P.J. (1990) *Backpropagation through time: what it does and how to do it*. Proceedings of the IEEE 78(10), 1990, 1550-1560. *A much-cited article explaining BPTT in a detailed fashion, and introducing a notation that is often used elsewhere.*

Williams, R.J. and D. Zipser (1989) *A learning algorithm for continually running fully recurrent neural networks*. Neural Computtion 1, 1989, 270-280