JACOBS
UNIVERSITY

Herbert Jaeger

Long Short-Term Memory in Echo State Networks: Details of a Simulation Study

Technical Report No. 27
February 2012

School of Engineering and Science

Long Short-Term Memory in Echo State Networks: Details of a Simulation Study

Herbert Jaeger

Jacobs University Bremen
School of Engineering and Science
Campus Ring
28759 Bremen
Germany

E-Mail: h.jaeger@jacobs-university.de
http://minds.jacobs-university.de

Abstract

Echo State Networks (ESNs) is an approach to design and train recurrent neural networks in supervised learning tasks. An important objective in many such tasks is to learn to exploit long-time dependencies in the processed signals ("long short-term memory" performance). Here we expose ESNs to a series of synthetic benchmark tasks that have been used in the literature to study the learnability of long-range temporal dependencies. This report provides all the detail necessary to replicate these experiments. It is intended to serve as the technical companion to a journal submission paper where the findings are analysed and compared to results obtained elsewhere with other learning paradigms.

1 Introduction

Many tasks in signal processing, pattern recognition and control require dynamical systems models whose output at some time is influenced by previous input. An intrinsic property of recurrent neural networks (RNNs) is that they have memory. The current network state is influenced by inputs which can date back considerable time. RNNs are thus a natural choice for modeling dynamical systems with memory. However, it is not easy to achieve long memory spans when RNNs are trained in a supervised fashion. There are currently three main approaches to obtain this objective:

- The *Long short-term memory* (LSTM) networks introduced by Hochreiter and Schmidhuber [4] achieve long memory spans through the use of a special kind of processing units whose activation decay can be controlled by trainable gates. Training is usually done by gradient descent using the back-propagation through time (BPTT) scheme [13].

- Echo State Networks (ESNs), introduced by Jaeger [5], can achieve long memory spans by using large networks [6, 14, 2, 1] and/or by using versions of leaky integrator neurons with long time constants [8, 15]. Training is done in the *reservoir computing* paradigm [10, 9], i.e. only the weights of output units are trained while the RNN itself is randomly created and not changed by training.

- By very much refined second-order optimization methods, Martens and Sutskever [11] have recently overcome the mathematical and algorithmic hurdles which previously prevented heads-on, gradient-based optimization methods from functioning in RNNs. Long memory spans are realized by generic sigmoid unit networks which are "simply" trained on targets that incorporate long-term memory effects.

In all of this work, the realized memory mechanisms are based on transient (albeit slow) dynamics and decay with time. This should not be confounded with other lines of research in RNN-based working memory mechanisms which are based on the formation of attractors (e.g. [7, 12]). In this report I am exclusively concerned with transient short-term memory.

In their original paper [4], Hochreiter and Schmidhuber designed a suite of synthetic tasks which challenge several aspects of short-term memory in adjustable and controllable ways. Most of these tasks have been repeated by Martens and Sutskever in their article [11], albeit sometimes with minor modifications. Here I give these tasks yet another go, using ESNs. The task versions employed by Martens and Sutskever are used; where their paper does not supply enough detail to reproduce the exact setup I asked the authors for clarifications.

The purpose of my investigation is not only to clarify what levels of performance can be reached with ESNs, but also with what effort. It is a common observation in machine learning that given a task and a basic algorithmic approach, the quality of the results depend very much on the experience and effort of the engineer. Therefore I "simulated" three levels of effort/expertise, and carried out learning experiments pretending to be operating within the options afforded by the respective levels. Concretely, I simulated the following levels:

Blind: Out-of-the-box ESN learning, investing absolutely no insight into problem or data. Specifically, the input signals of a task are not individually optimized.

Basic: Still out-of-the-box ESN learning, but being aware that different input channels of the task have different functional roles and should be individually optimized.

Smart: The structure of the RNN is optimized in a problem-specific way. This requires experimentation and/or insight.

For each of the Hochreiter/Schmidhuber tasks that have been re-addressed by Martens/Sutskever, this report provides (i) a detailed specification of the task, (ii) a detailed description of how ESN training was configured to solve the respective task, and (iii) a documentation of the results. In companionship with the original Hochreiter/Schmidhuber and Martens/Sutskever papers, the material presented here thus provides a basis for comparing essential aspects of the three approaches that address training short-term memory in RNNs. This report however only documents technical material. A comparative discussion is delegated to a separate journal submission.

All experiments except the *blind* version of the "20-bit memory" task have been carried out using Matlab (release R2008b for Macintosh) on a 2GHz Intel Macintosh notebook computer with 2 GB RAM. The *blind* version of the "20-bit memory" task, while just feasible on my notebook, led to time-consuming paging, and was run on a more high-end 2.9 GHz PC with 8 GB RAM. The Matlab code for all experiments is available at the author's publications page `http://minds.jacobs-university.de/pubs`.

The report is structured along the task suite in the same order as in [11]. After a summary of the generic ESN training setup (Section 2), each task is documented in a separate section.

2 Setting up ESNs for the Benchmark Tasks

First a note on terminology: task vs. experiment vs. trial vs. run. This report documents ESN training and testing for a number of benchmark *tasks*, each of which specifies a particular type of inputs and desired outputs. For each of these tasks, I describe a learning *experiment*, which consists in a particular setup of an ESN learner, repeated learning *trials*, and performance statistics across the trials. Each trial in turn is specified by a fresh set of training and testing data and a fresh random initialization of the ESN, which is then trained and tested on these data. A *run* refers to a single presentation of an input sequence to the ESN, either in training or testing.

I assume some familiarity with the principles of reservoir computing. For all experiments reported here I use the same basic setup. The reservoir network employs a version of leaky-integrator neurons and has the update equation

$$\mathbf{x}(n+1) = (1-\alpha)\,\mathbf{x}(n+1) + \alpha\, tanh(\mathbf{W}\,\mathbf{x}(n) + \mathbf{W}^{\mathsf{in}}\,\mathbf{u}(n+1)), \qquad (1)$$

where $\mathbf{x}(n) \in \mathbb{R}^K$ is the K-dimensional state vector at time n, α is the leaking rate, $\mathbf{W}$ is the $K \times K$ matrix of reservoir connection weights, $\mathbf{u}(n)$ is the input at time n, and for L-dimensional input $\mathbf{W}^{\mathsf{in}}$ is the $K \times L$ sized input weight matrix. The *tanh* nonlinearity is applied element-wise.

Since all tasks considered here process finite time series, time always starts at $n = 0$, and the network state is initialized with a starting state $\mathbf{x}(0)$ which is the same across all runs within one trial. In all experiments I used the all-zero state for $\mathbf{x}(0)$.

The M-dimensional outputs are computed at times $n > 0$ by

$$\mathbf{y}(n) = \mathbf{W}^{\mathsf{out}}\,\mathbf{x}(n), \tag{2}$$

where $\mathbf{W}^{\mathsf{out}}$ is the $M \times K$ dimensional output weight matrix. Thus, the output is a linear readout from the network state. According to the reservoir computing paradigm, only the readout weights are trained. The input weights and the reservoir weights are created randomly and are not changed by training.

In each trial, the training data consist of N^{train} many input time series $\bar{\mathbf{u}}_1, \ldots, \bar{\mathbf{u}}_{N^{\mathsf{train}}}$, which have lengthes $T_1, \ldots, T_{N^{\mathsf{train}}}$. In most tasks the length is the same for all input sequences. If that case, I refer to the unique sequence length as T. Each input series $\bar{\mathbf{u}}_i$ is paired with a target output series $\hat{\bar{\mathbf{y}}}_i$. According to the nature of the task, there are two variants:

- The task requires an output at every timestep. The length of a target output series is then identical to the length of its associated input series. Since this case occurs only with datasets where the inputs have a uniform length T, the length T^{out} of the training output series is then uniformly $T^{\mathsf{out}} = T$.
- The task requires an output only at the last timestep. The length of the target series is then $T^{\mathsf{out}} = 1$.

A trial unfolds in the following steps.

1. **Data generation.** A set of N^{train} input time series $\bar{\mathbf{u}}_i$ and its associated target outputs $\hat{\bar{\mathbf{y}}}_i$ is synthesized, as well as a set of N^{test} many input-output pairings for testing.
2. **ESN generation.** A reservoir weight matrix $\mathbf{W}$ and input weights $\mathbf{W}^{\mathsf{in}}$ are randomly created. In the "blind" and "basic" experiments, $\mathbf{W}$ is a sparse matrix with on average 10 nonzero elements per row, at randomly chosen positions. Its nonzero entries are first uniformly sampled from $[-1, 1]$, then the matrix is scaled to attain a spectral radius ϱ which is specific to an experiment. In the "smart" condition, $\mathbf{W}$ is intelligently designed with the goal to support "rich" dynamics. The input weights are always full matrices which are created by first uniformly sampling entries from $[-1, 1]$. Then the columns of this raw input matrix are individually scaled with factors $\sigma_1, \ldots, \sigma_M$ specific to each experiment to obtain the final $\mathbf{W}^{\mathsf{in}}$.

3. **State and target harvesting.** For each training input sequence $\bar{\mathbf{u}}_i$, the reservoir is driven with it by using (1), which yields a state sequence $(\mathbf{x}_i(1), \ldots, \mathbf{x}_i(T_i))$ (the initial state is not included). From these states, the T^{out} many ones that are paired in time with target outputs are taken and appended to the right as columns to a state collection matrix $\mathbf{S}$. When all training input sequences have been processed in this way, $\mathbf{S}$ has grown to size $K \times T^{\mathsf{out}}\, N^{\mathsf{train}}$. Similarly, the associated target outputs of all runs are collected in a target collection matrix $\mathbf{T}$ of size $M \times T^{\mathsf{out}}\, N^{\mathsf{train}}$.

4. **Compute output weights.** The output weights are now computed as the linear regresssion weights of the targets on the states, by

$$\mathbf{W}^{\mathsf{out}} = ((\mathbf{S}^{\mathsf{T}})^{\dagger}\, \mathbf{T}^{\mathsf{T}})^{\mathsf{T}}, \tag{3}$$

where $\cdot^{\mathsf{T}}$ denotes matrix transpose and $\cdot^{\dagger}$ denotes the pseudo-inverse. In one problem (the "blind" setup for the 20bit-memory task), very large reservoirs became necessary. The pseudo-inverse algorithm then would exceed the Matlab RAM capacity allocated on my notebook. In this case I resorted to the numerically inferior, but more memory-economic regularized Wiener-Hopf solution to compute the linear regression by

$$\mathbf{W}^{\mathsf{out}} = (\mathbf{S}\,\mathbf{S}^{\mathsf{T}} + r^2\mathbf{I})^{-1}\, \mathbf{S}\,\mathbf{T}^{\mathsf{T}}. \tag{4}$$

The regularizer r was set as small as possible while still warranting numerical stability.

5. **Testing.** The reservoir is then driven by the test sequences in a similar way as it was during state harvesting. For each test run i, the outputs $\mathbf{y}_i(n)$ at the task-relevant output times were computed by (2). Depending on the task, the relevant output times were either all times of the entire run, or only the last timestep. The trained outputs $\mathbf{y}_i(n)$ of the ith run were then compared to the target outputs $\hat{\mathbf{y}}_i(n)$ by considering the absolute error $\mathrm{abs}(\hat{\mathbf{y}}_i(n) - \mathbf{y}_i(n))$.

   The benchmark tasks considered in this report come in two flavors. Some of the tasks have real-valued output targets. For these tasks, the reference paper [11] calls a run successful if the absolute error never exceeds 0.04 (for all relevant times and output channels). Other tasks have binary output. Here the success criterium for a run used in [11] is to call the run successful if, per timestep, the maximal network output is obtained on the channel where the target is 1. Out of programming laziness I adopted a slightly more demanding criterium, by calling a run successful if the maximal absolute error over all relevant times and output channels is smaller than 0.5.

   Like the reference authors Martens and Sutskever I call a trial successful when the failed runs make up for at most 1 percent.

Per experiment, always hundred trials were carried out, each comprising 1000 runs on independently generated data (exception: the 5bit memory task, in which there are only 32 possible different test sequences).

The setup of a trial is based on a number of global parameters:

1. the reservoir size K,
2. the spectral radius ϱ,
3. the input weight scalings $\sigma_1, \ldots, \sigma_M$, and
4. the leaking rate α.

These parameters are identical across all trials of an experiment. They are optimized by manual experimentation in explorative trials which I carried out before settling on the final version of the experiments. My strategy here is to use reduced-sized datasets and small reservoirs for this exploration. This leads to fast processing times per trial (typcially less than 10 seconds), which enables a high turnover rate. This manual exploration phase altogether typically took less than half an hour per task (in the *blind* and *basic* conditions). The settings of these globals are quite robust, without careful fine-tuning required for good performance. An alternative is to employ automated search methods for this optimization. The currently most high-end reservoir computing toolbox OGER ("OrGanic Environment for Reservoir computing", implemented in Python with interfaces to many other neural toolboxes and Matlab, obtainable at `organic.elis.ugent.be/organic/engine`) has a choice of sophisticated search methods implemented. I personally prefer manual experimentation because for me it works faster and yields insight into the dynamical characteristics of a task.

In the three "use-case type" conditions *blind*, *basic*, and *smart* I invested different degrees of effort in the optimization:

Blind: All input scalings were optimized together by a global scaling, that is, input scalings were set to $(\sigma_1, \ldots, \sigma_M) = \sigma\,(1, \ldots, 1)$, and only the uniform scaling σ was optimized, along with K, ϱ, and α. The optimization search space is thus of dimension 4.

Basic: All parameters $\sigma_1, \ldots, \sigma_M, K, \varrho, \alpha$ were optimized, giving different inputs the chance to impact on the reservoir by individual degrees. For high-dimensional input this however quickly becomes infeasible both in manual or automated search. Therefore, the inputs were grouped into subsets of similar functional role, and the input scalings were optimized per group by a group-uniform scaling. The *basic* condition poses higher requirements on the user than the *blind* conditions in two ways. First, this grouping of inputs requires an elementary insight into the nature of the task. Second, the number of optimization parameters increases, which makes the search for good

values more demanding. In the experiments reported here, the inputs were grouped in 2 or 3 groups depending on the task, leading to optimization search spaces of dimensions 5 or 6.

Smart: Global parameters were treated as in the *basic* simulations. In addition, the reservoir matrix $\mathbf{W}$ was structurally pre-configured with the aim to endow it with a "rich" excitable dynamics.

3 The Addition Task

3.1 Task: Synopsis

In this task there are two input channels ($L = 2$). The first channel receives a stream $u_1(n)$ of random reals sampled uniformly from $[0, 1]$. The second channel receives zero input $u_2(n) = 0$ at all times except at two timesteps $n_1 < n_2$ when $u_2(n_1) = u_2(n_2) = 1$. The objective is that at the end of a run (much later than n_1 or n_2), the network should output the normalized sum $(u_1(n_1) + u_1(n_2))/2$. An additional difficulty is that the length of input sequences varies randomly.

3.2 Task: Detail

In the following let $i[a, b]$ denote a random variable which uniformly samples from the integers contained in the real interval $[a, b]$.

Data. First a minimal length T_0 for input sequences is fixed. To generate an input sequence $\bar{\mathbf{u}}_i$ for training or testing, determine a random length $T_i = i[T_0, 1.1\,T_0]$ which may be up to 10% longer than T_0. Sample the two critical input times $n_1 = i[1, 0.1\,T_0], n_2 = i[0.1\,T + 1, 0.5\,T]$. Then generate a 2-dim input series $\bar{\mathbf{u}}_i = (\mathbf{u}_i(1), \ldots, \mathbf{u}_i(T_i))$, where $\mathbf{u}_i(n) = [u_1(n), u_2(n)]$ by filling the first components u_1 with uniformly random reals from $[0, 1]$, and setting all $u_2 = 0$ except at times n_1, n_2 where it is set to 1. The single-channel target output at the last timestep is $\hat{\mathbf{y}}(T_i) = (u_1(n_1) + u_1(n_2))/2$.

Success criterium. A single run of a trained network is called successful if the network output at time T_i does not absolutely differ from the target output by more than 0.04. A trained network is called successful if the success rate of its individual runs on random test sequences is at least 99%.

3.3 Experiment

This task turned out to be particularly easy, so I only ran the *blind* version. I chose a minimal sequence length of $T_0 = 10,000$. Each trial was based on $N^{\mathsf{train}} = 300$ training sequences (the number of test sequences is always $N^{\mathsf{test}} = 1000$ throughout all experiments documented in this report). Hundred trials were performed (again, this is the same for all reported experiments).

Addition: parameters	
	blind, $T = 10,000$
K	100
ϱ	3
σ	0.01
α	0.001
N^{train}	300
Addition: results	
Mean CPU time (sec)	98
Mean abs output weight	1.6e+9
Mean abs train error	0.00016
Mean abs test error	0.00025
Max abs test error	0.0034
Nr failed trials	0

Table 1. Parameters and results of the addition task. For explanation see text.

The networks in this task were parametrized as shown in Table 3.3. A zero starting vector $\mathbf{x}(0)$ was used. The manual optimization was done in a quick fashion until zero error performance was achieved; further manual or automated optimization would likely yet improve results. The results are summarized in Table 3.3.

The entries in Table 3.3 are to be understood as follows. *Mean CPU time*: refers to the time used for steps 3 & 4 (cf. Section 2), i.e. state harvesting and computing output weights by linear regression. *Mean abs output weight*: the mean over all trials of the absolute values of the obtained output weights. *Mean abs train error*: The mean over all runs and trials of the absolute output error of the trained network, run on the training data. *Mean abs test error*: similar for the testing errors. *Max abs test error*: the maximum over all trials and runs of absolute test output errors. *Nr failed trials*: how many learning trials failed according to the trial success criterium "more than 1 percent of runs had absolute error greater than 0.04". In this experiment every single test run in every trial was successful (because the *Max abs test error* was below 0.04).

Figure 1 shows the development of states in a typical run, and the maximal (within a trial) absolute test errors encountered, plotted over all trials.

4 The Multiplication Task

4.1 Task

The multiplication task is defined entirely like the addition task, except that the desired output at the last timestep is $\hat{\mathbf{y}}(T_i) = u_1(n_1)\, u_1(n_2)$. The success criterium for runs and trials is the same as in the addition task.

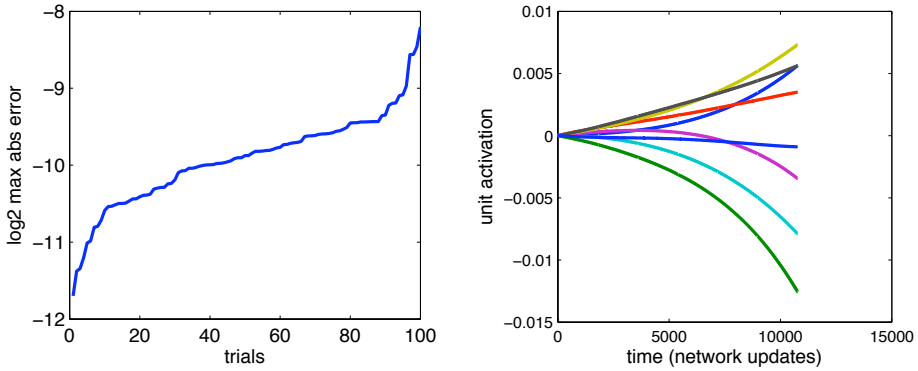

Figure 1: The addition task. Left: log2 of the maximum absolute test errors obtained in the 100 trials, sorted. Right: traces of 8 reservoir units in a typical run.

4.2 Experiments

Multiplication is "more nonlinear" than addition, and training became a bit more challenging. I carried out four experiments, two under condition *blind* and two under condition *basic*. In both conditions, one of the two experiments used a minimal sequence length $T_0 = 200$, which is the maximal length considered in [11]. Given this sequence length, the reservoir size K was increased until very good performance was achieved. Not surprisingly it turned out that the *blind* condition led to a larger reservoir than the *basic* one. The other experiment under the *blind* condition used $T_0 = 500$, which turned out to require a reservoir size of $K = 1000$. This complete experiment (100 trials, including testing) kept my notebook busy for one night (runtime per trial about 6 minutes), which was the maximum resource that I was willing to spend. Under the *basic* condition, a sequence length $T_0 = 1000$ was mastered with a reservoir size of $K = 300$.

The training data sizes were adjusted in the four experiments to warrant good testing performance. When reservoirs are larger, larger training sets are necessary to prevent overfitting. Since the results obtained were fully satisfactory, no further *smart* condition experiments were undertaken.

Table 4.2 gives an overview of the parameters used, and the results obtained. The input scaling parameters are given in this table as σ_1, σ_2, where the former refers to the numerical payload input (multiplication candidates) and the latter to the indicator input. In the *blind* condition both parameters were uniformly optimized as $(\sigma_1, \sigma_2) = \sigma\,(1, 1)$, while in the *basic* condition they were individually optimized. Figure 2 highlights the max abs error distribution across trials and presents examplary reservoir state sequences.

Multiplication: parameters				
	blind		*basic*	
	$T_0 = 200$	$T_0 = 500$	$T_0 = 200$	$T_0 = 1000$
K	300	1000	100	300
ϱ	10	12	12	12
σ_1, σ_2	0.3, 0.3	0.1, 0.1	0.01, 1	0.01, 1
α	0.0001	0.0003	0.00005	0.00002
N^{train}	1000	2000	1000	1000
Multiplication: results				
Mean CPU time (sec)	14	180	6.7	65
Mean abs output weight	1.1e+9	9.7e8	2.4e+11	2.0e11
Mean abs train error	9.2e-5	0.0015	2.1e-4	1.9e-4
Mean abs test error	1.3e-4	0.0024	2.3e-4	2.3e-4
Max abs test error	0.0020	0.046	0.0034	0.0042
Nr failed trials	0	0	0	0

Table 2. Parameters and results of the multiplication task.

5 The XOR Task

5.1 Task

The XOR task is defined in close analogy to the addition and multiplication tasks, except that the input signal u_1 is 0-1-binary (uniform random), and the target output at the last timestep is $\hat{\mathbf{y}}(T_i) = \mathsf{XOR}(u_1(n_1)\, u_1(n_2))$. The success criterium for runs is adapted to the binary nature of the task. A run is deemed successful if the absolute output error at the last timestep is less than 0.5. Notice that the model is simply trained with linear regression on real-valued targets 0 and 1. Binary outputs could be then obtained by thresholding at 0.5, which is what this success criterium amounts to. A trial is called a success if no more than 1 percent of its test runs fail.

5.2 Experiment

This task again turned out quite easy, so only a single *blind* condition experiment with $T_0 = 1000$ was executed. Table 5.2 and Figure 3 present the settings and findings.

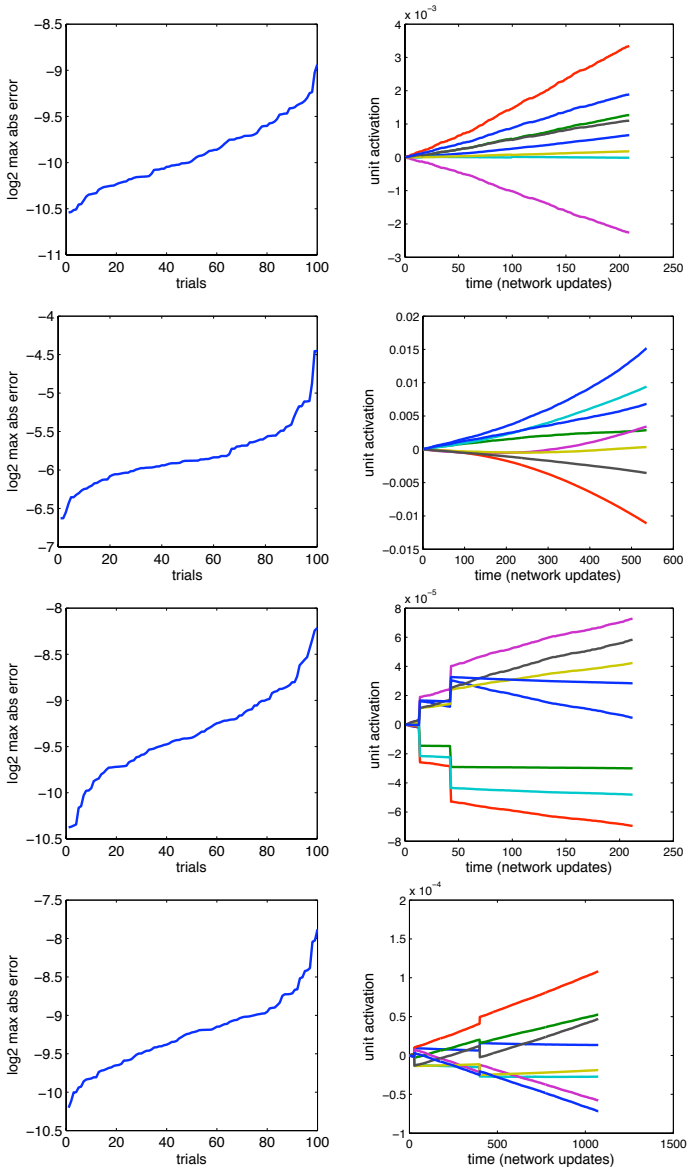

Figure 2: The multiplication task. Left: log2 of the maximum absolute test errors obtained in the 100 trials, sorted. Right: traces of 8 reservoir units in a typical run. From top to bottom: (i) condition *blind*, $T_0 = 200$; (ii) *blind*, $T_0 = 500$; (iii) *basic*, $T_0 = 200$; (iv) *basic*, $T_0 = 1000$.

6 The 2-Symbol Temporal Order Task

6.1 Task

Synopsis. The reservoir is driven with a random distractor signal most of the time, except at two random times T_1, T_2 (where $T_1 < T_2$), when the input is a non-distractor event A or B. Which of the two events A or B is fed at time T_1 or

XOR: parameters	
	blind, $T = 1000$
K	100
ϱ	2
σ	3
α	0.00003
N^{train}	200
XOR: results	
Mean CPU time (sec)	7.1
Mean abs output weight	3.3e8
Mean abs train error	7.8e-5
Mean abs test error	8.5e-5
Max abs test error	0.00097
Nr failed trials	0

Table 3. Parameters and results of the XOR task.

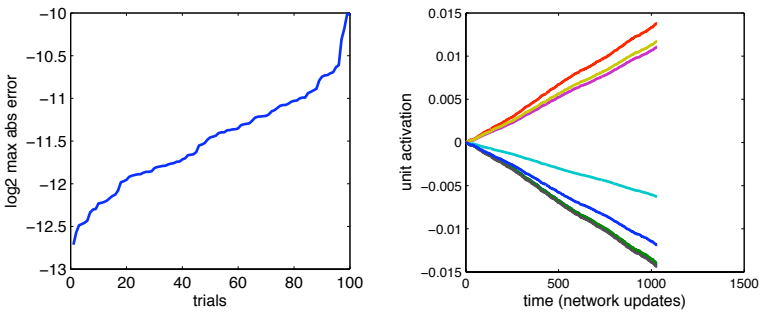

Figure 3: The XOR task. Left: log2 of the maximum absolute test errors obtained in the 100 trials, sorted. Right: traces of 8 reservoir units in a typical run.

T_2 is random, so there are four possible event orderings A-A, A-B, B-A, and B-B. At the last timestep the output has to classify this order of occurrence, i.e. the target output is a one-out-of-four choice.

Data, detail. The input in this task has six channels, the first four for the distractor inputs and the last two for the critical events A and B. All runs have the same duration T. The critical event times T_1, T_2 are sampled from $i[0.1T, 0.2T]$ and $i[0.5T, 0.6T]$. To generate a task run input, at all timesteps except T_1, T_2, exactly one of the four distractor inputs is set to 1. At T_1, channel 5 or channel 6 is randomly set to one; the same is done for time T_2. There are four outputs. Only at the last timestep $n = T$ the network output is evaluated. The target output at that time is one of the four possible indicator outputs $(1, 0, 0, 0), \ldots, (0, 0, 0, 1)$, in agreement with the four possibilities of what the input at times T_1, T_2 was.

Success criteria. This is a binary-value task. A test run of the trained network is classified as success if none of the four outputs at time $n = T$ differs from the

2-symbol temporal order: parameters		
	blind, $T = 200$	*basic*, $T = 1000$
K	100	50
ϱ	0.1	0.8
$\sigma_1, \sigma 2$	0.15 0.15	0.001 1.0
α	0.0001	0.001
N^{train}	500	200
2-symbol temporal order: results		
Mean CPU time (sec)	3.7	5.2
Mean abs output weight	3.5e10	3.6e6
Mean abs train error	0.00028	0.00029
Mean abs test error	0.00036	0.00039
Max abs test error	0.011	0.0066
Nr failed trials	0	0

Table 6.2 Parameters and results of the 2-symbol temporal order task. Notice that in the *blind* conditions some runs failed (as can be seen from the max test error) but their percentage was below 1% so the trials were counted as successes.

binary target value by more than 0.5. A trial is successful if no more than 1 percent of the test runs fails. In fact, in all trials all runs were successful.

6.2 Experiments

This task proved to be relatively easy, so I only ran a *blind* experiment for $T = 200$ and a *basic* one for $T = 1000$. Quite small reservoirs of sizes 100 and 50, respectively, were enough for perfect performance. In the *basic* condition, the four distractors and the two "critical" channels were grouped for input scalings σ_1, σ_2. Table 6.2 and Figure 4 summarize the parameters and outcomes.

7 The 3-Symbol Temporal Order Task

7.1 Task

The task is completely analogous to the 2-symbol temporal order task, except that now there are three critical times (sampled from $i[0.1T, 0.2T]$, $i[0.3T, 0.4T]$ and $i[0.6T, 0.7T]$) when one of the input channels 5 or 6 gets a randomly channel-assigned 1 input. This makes for 8 possible orderings of events, which have to be classified by the 8-channel output of the trained system at the last timestep.

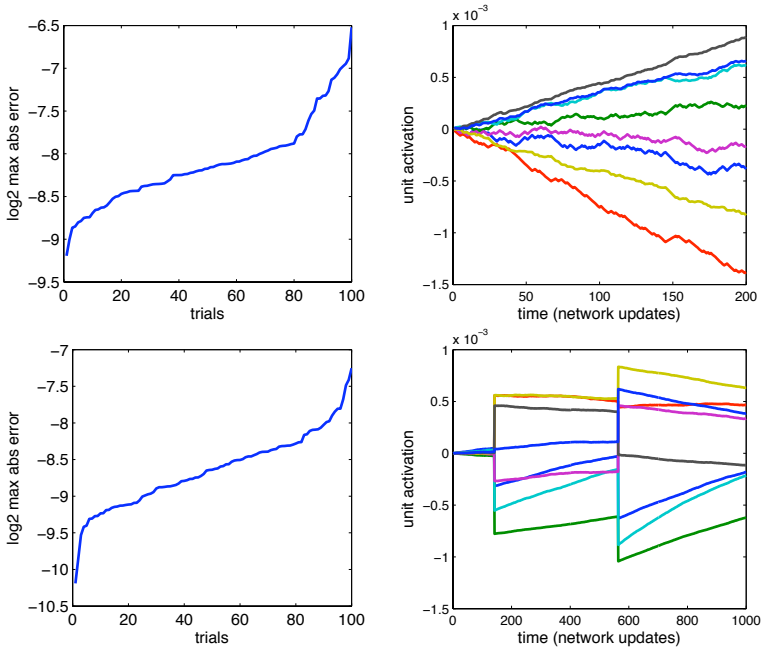

Figure 4: The 2-symbol temporal order task. Left: log2 of the maximum absolute test errors obtained in the 100 trials, sorted. Right: traces of 8 reservoir units in a typical run. Top: Condition *blind*, $T = 200$; bottom: *basic*, $T = 1000$

3-symbol temporal order: parameters		
	blind, $T = 200$	*basic*, $T = 1000$
K	500	100
ϱ	1.3	0.8
$\sigma_1, \sigma 2$	0.008 0.008	0.001 1.0
α	0.003	0.001
N^{train}	2000	200
3-symbol temporal order: results		
Mean CPU time (sec)	47	6.5
Mean abs output weight	7.2e10	9.4e8
Mean abs train error	0.089	0.012
Mean abs test error	0.12	0.024
Max abs test error	1.26	0.32
Nr failed trials	0	0

Table 7.2 Parameters and results of the 3-symbol temporal order task.

7.2 Experiments

Again this proved rather easy, so I only carried out a single *blind* and a single *basic* type experiment. Findings are given in Table 7.2 and Figure 5.

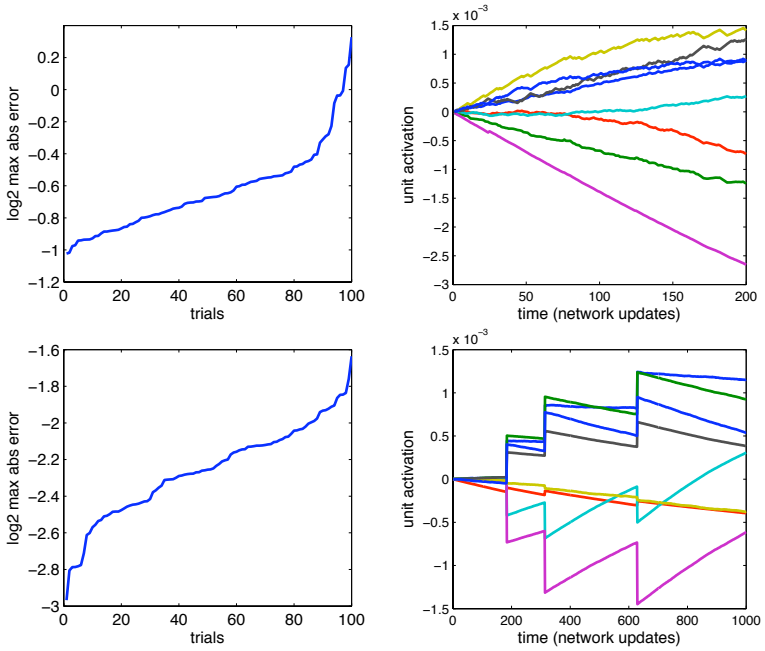

Figure 5: The 3-symbol temporal order task. Left: log2 of the maximum absolute test errors obtained in the 100 trials, sorted. Right: traces of 8 reservoir units in a typical run. Top: Condition *blind*, $T = 200$; bottom: *basic*, $T = 1000$

8 The Random Permutation Task

8.1 Task

The input in this task is 100-dimensional and binary. At each timestep, exactly one of the 100 channels gets a 1 value, the others are zero; i.e. we have a 100-symbol place coding. A run has length T. At time 1, one of the first two channels is randomly assigned to be 1. For the remaining time $n = 2, \ldots, T$, at each timestep one of the 98 remaining channels is set to 1 in a random fashion.

The only relevant output is at the last timestep T. At this last timestep, the target output is a 100-dimensional binary vector which replicates the input vector from time $n = 1$.

A run is successful if the output vector at $n = T$ differs from the target vector by no more than 0.5 in any component. A trial is successful if no more than 1 percent of its test runs fail.

In this task specification I follow Martens and Sutskever [11]. The original task (Nr. 2b) in Hochreiter and Schmidhuber [4] differs in some non-crucial aspects from the Martens and Sutskever version. Specifically, in the original version from Hochreiter and Schmidhuber the runlength T is equal to the input dimension, and the 98 distractor inputs which are then fed are a permutation of the possible 98 symbols (hence the name, "random permutation" task). When variable runlengthes are investigated (as in [11]) while the input dimension is kept constant,

Random permutation: parameters	
	blind, $T = 1000$
K	150
ϱ	0.0 (!)
σ	1
α	0.000001
N^{train}	500
Random permutation: results	
Mean CPU time (sec)	24
Mean abs output weight	270
Mean abs train error	3.2e-16
Mean abs test error	3.6e-16
Max abs test error	1.2e-13
Nr failed trials	0

Table 8.2 Parameters and results of the random permutation task.

the permutation constraint must be replaced by a random presentation of the distractors. Furthermore, Hochreiter and Schmidhuber call a run a success if the absolute difference between outputs and targets is nowhere larger than 0.25.

8.2 Experiment

This proved a very simple task, even trivial in a revealing fashion. Thus I only present a single *blind* experiment with $T = 1000$, but as will become clear, essentially any runlength would be achievable. Findings are given in Table 8.2 and Figure 6.

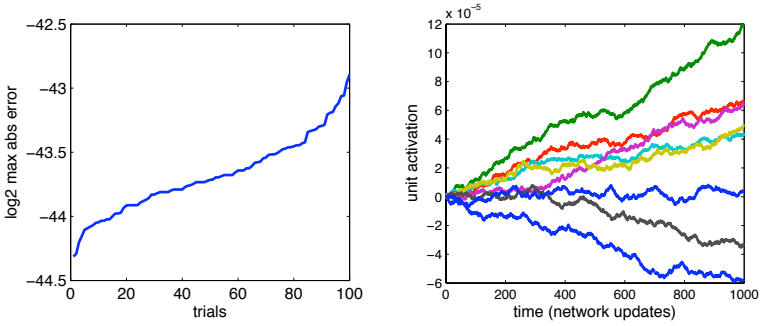

Figure 6: The random permutation task, *blind*, $T = 1000$. Left: log2 of the maximum absolute test errors obtained in the 100 trials, sorted. Right: traces of 8 reservoir units in a typical run.

8.3 Discussion

An essentially zero test error was obtained with a spectral radius of 0.0, i.e. a "reservoir" with all-zero connections, and a very small leaking rate, i.e. an extreme degree of smoothing. This extreme configuration functions as follows. At each timestep, the input connection weight vector corresponding to the input channel that was 1 at that timestep is added to the network state. Due to the low leaking rate, this additive contribution essentially remains there until the end of the run. Thus, at the end of the run the network state $\mathbf{x}(T)$ is essentially the sum of all the input weight vectors that have been activated by corresponding inputs encountered during the run. The weight vectors for the 100 input channels have been constructed as random vectors. The critical input in channels 1 or 2 given at start time thus adds one of two unique random vectors to $\mathbf{x}(T)$. If the network size K is larger than the input dimension (such as here where $K = 150 > 100 =$ input dimension), generically these two critical weight vectors will be linearly independent from each other and all the other 98 distractor weight vectors that may appear as summands in $\mathbf{x}(T)$. The linear regression computation of the output weights thus just has to pick the projective component from the critical weight vectors which is independent from all others, and transform it into a 1 value in the associated critical output channel.

Notably, this state of affairs was not transparent to me when I started optimizing the global control parameters. I was however quickly guided to vanishingly small values of the spectral radius, which then led me to the insight reported here of why this task is essentially trivial.

9 The 5-Bit Memory Task

9.1 Task: Synopsis

At the beginning of each run, a 2-dimensional, 5-timestep memory pattern is presented to the network. Then, for a period of duration $T_0 - 1$, a constant distractor input is given. After this, i.e. at time $5 + T_0$, a cue signal is given as a spike input, after which in the final 5 time steps, the memory pattern has to be reproduced in the output units. In all preceding timesteps before this last 5-step recall period, the output has to be a constant on anonther output channel, signalling something like "still waiting for the cue for recall".

9.2 Task: Detail

Data. The total length of a single run is $10 + T_0$. There are four binary input channels and four binary output channels. The first two channels of each of the inputs or outputs carries the memory pattern. The third channel in the input feeds the distractor input. The fourth input channel carries the cue. In the output, the

third output channel should signal the "waiting for recall cue" condition, and the fourth is unused and should always be zero (this channel could just as well be dropped but it is included in the original task specs, so I keep it).

Specificially, an input sequence is constructed as follows. First, all $4 \times (10+T_0)$ inputs are initialized to zero. Then, for the first 5 timesteps, one of the first two input channels is randomly set to 1. Note that there are altogether $2^5 = 32$ possible input patterns, hence this task is named the *5-bit* memory task. Then, for timesteps 6 through T_0+4, the third input channel is set to 1 (distractor signal). At time T_0+5, the fourth input channel is set to 1 (cue). For the remaining times from T_0+6 to T_0+10, the input is again set to 1 on the 3rd channel. Thus, at every timestep exactly one of the four inputs is 1.

The target output is always zero on all channels, except for times 1 to T_0+5, where it is 1 on the 3rd channel, and for times T_0+6 to the end where the input memory pattern is repeated in the first two channels.

Notice that there are only 32 different sequences possible (given T_0).

Success criteria. A run is successful if at all timesteps and in all of the four output channels, the absolute difference between the network output and the target output is less than 0.5. A trial is successful if less than 1 percent of its runs is successful. Since there are only 32 different sequences, this is equivalent to requiring that all possible 32 test runs be successful.

9.3 Experiments

This task turned out more challenging than any of the preceding ones, so all conditions *blind*, *basic*, and *smart* were demonstrated, in order to demonstrate typical reservoir computing strategems when tackling a problem.

A peculiarity of this task is that there are only 32 different sequences available. I trained and "tested" each model on all of these 32 sequences. Since everything is deterministic, the training error equals the testing error, so a separate testing phase was omitted. In the reference study [11], RNNs are trained iteratively with randomly chosen input sequences. In the $T_0 = 200$ version of this task (the maximum length explored in that work), about 10 Mio randomly chosen sequences were presented to the learning network. It is thus fair to assume that all possible sequences were seen during training in almost exactly equal shares. In this sense, my training setup which uses exactly these 32 sequences (and each of them once) is comparable with the work in [11].

The following experiments were done:

1. Condition *blind*, $T_0 = 200$. The blind setup rendered this task quite unwieldy, thus a quite large reservoir of size $K = 2000$ was needed to reach perfect performance.

2. Condition *basic*, $T_0 = 200$. Allowing the inputs to be scaled differently made it possible to achieve reasonable performance with a much smaller reservoir

5-bit Memory: parameters					
	blind $T_0 = 200$	*basic* $T_0 = 200$	*basic* $T_0 = 200$	*smart* $T_0 = 200$	*smart* $T_0 = 1000$
K	2000	200	500	200	500
ϱ	1	1	1	1	1
$\sigma_1, \sigma_2, \sigma_3$	.01 .01 .01	2e-4 2e-6 2e-1	*same*	*same*	*same*
α	1	1	1	1	1
N^{train}	32	32	32	32	32
5-bit Memory: results					
Mean CPU time (sec)	117	1.2	3.6	1.1	17.3
Mean abs outweight	8.0e4	1.32e8	2.2e7	2.1e5	2.3e4
Mean abs error	.0050	.0013	9.6e-5	7.7e-5	8.7e-7
Max abs error	.31	.80	.80	.039	.00044
Nr failed trials	0	13	1	0	0

Table 4. Parameters and results of the 5-bit memory task.

of $K = 200$.

3. Condition *basic*, still $T_0 = 200$, but using a larger reservoir of $K = 500$ led to almost perfect performance.

4. Condition *smart*, $T_0 = 200$: perfect performance was possible with a network size $K = 200$.

5. Condition *smart*, $T_0 = 1000$: perfect performance with $K = 500$.

Table 9.3 gives an overview of parameters and results of these five experiments. In the *basic* and *smart* conditions, the four inputs were grouped in three groups, with the two "memory payload" inputs lumped in one group and the remaining two inputs in two further singleton groups. This led to three scaling parameters $\sigma_1 \cdot (u_1, u_2), \sigma_2 \cdot u_3, \sigma_3 \cdot u_4$, – these three parameters are reported in the table. For all *basic* and *smart* experiments, the same scalings were used.

9.4 Discussion

A number of phenomena that occurred in this interesting task deserve a little more discussion.

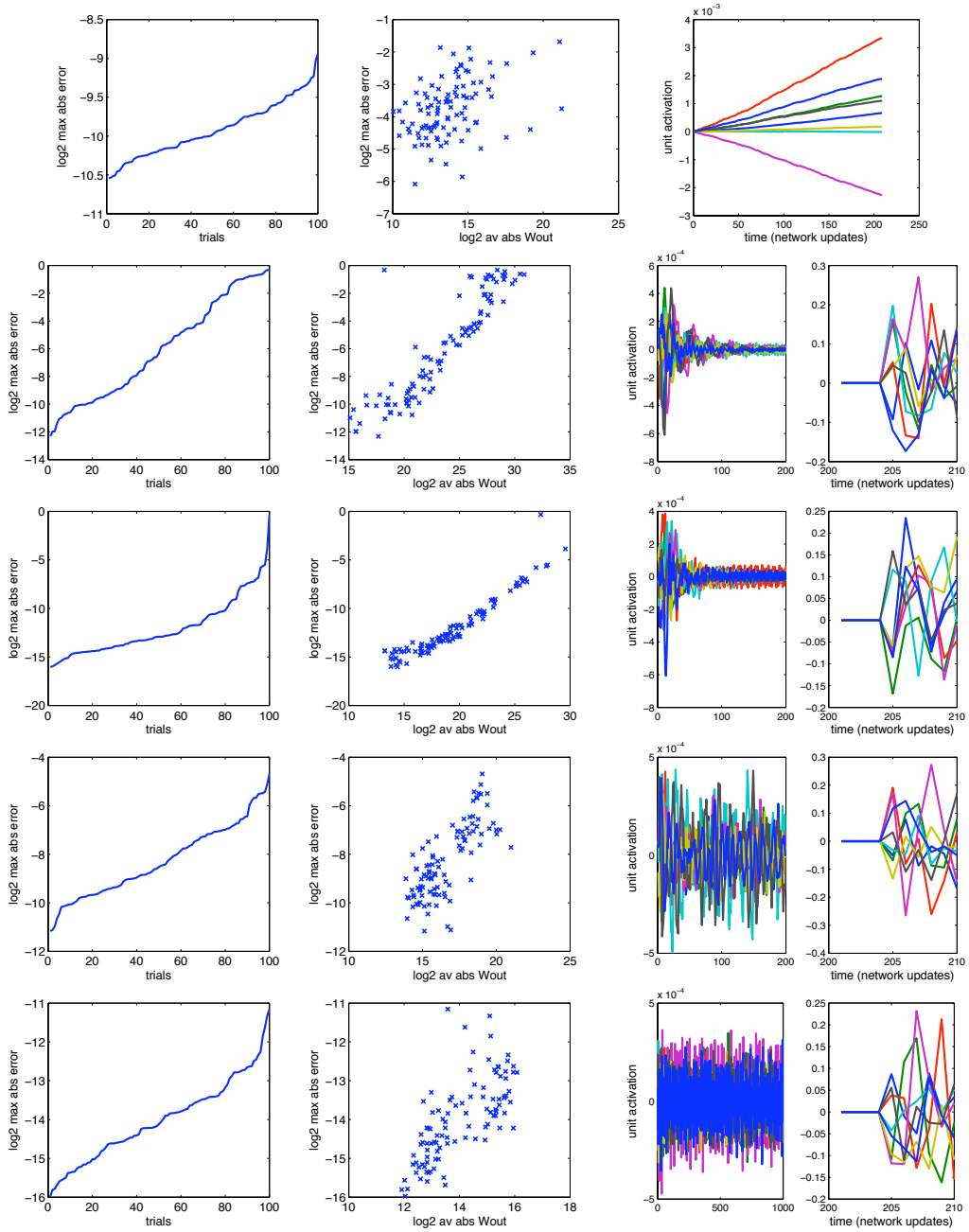

Figure 7: The 5-bit memory task. Left: log2 of the maximum absolute test errors obtained in the 100 trials, sorted. Center: max absolute test error vs. average absolute output weight size. Right: traces of 8 reservoir units in a typical run. For the *basic* and *smart* conditions in rows 2–5, the state sequence is shown in two parts in different scalings. From top to bottom: (i) condition *blind*, $T_0 = 200$; (ii) *basic*, $T_0 = 200$, $K = 200$; (iii) *basic*, $T_0 = 200$, $K = 500$; (iv) *smart*, $T_0 = 200$; (v) *smart*, $T_0 = 1000$.

Peak performance at unit spectral radius and unit leaking rate. The manual optimization quickly revealed that performance in this task sharply peaks when the spectral radius ϱ and the leaking rate α are both very close or identical to 1. A leaking rate $\alpha = 1$ means that the update equation (1) degenerates to a standard sigmoid-unit reservoir. With $\varrho = \alpha = 1$, reservoirs driven by low-amplitude constant input will converge to a resting state extremely slowly, and on this transient the state dynamics will exhibit a mixture of decaying oscillations determined by the spectrum of complex eigenvalues of $\mathbf{W}$. This is clearly visible in the state sequences plotted in Figure 7 for the *basic* and *smart* conditions, where the external input in the long interim distractor phase was scaled to a very low value of 2e-6. In the *blind* condition (top row in this figure), the oscillatory component of the state dynamics is overshadowed by the relatively large-amplitude distractor input which pulls the states toward a nonzero fixed point attractor (which would be converged to only long after the allotted runlength).

After witnessing this phenomenon in the *basic* condition, I was led to the intuitive (but admittedly vague) understanding that the dynamical mechanism which here "transports" the initial memory pattern information through the distractor period might be based on phase and amplitude relationships of the components of a frequency mix. I hypothesize that during the first five timesteps, when the payload pattern is fed to the reservoir, the indigeneous oscillation components of the reservoir (as determined by the complex eigenvalues of the weight matrix) are started off with a particular profile of phases and amplitudes which is specific to the memory pattern. This profile would then be carried through the distractor period by the undisturbed and almost non-decaying oscillations of the $\varrho = \alpha = 1$ dynamics, to be decoded during the last five timesteps. It is interesting to note that the RNNs trained in the reference work [11] likewise develop a mixture of high frequency oscillations in successful training runs (as can be seen in the video available at `http://www.cs.utoronto.ca/~ilya/pubs/`). If this hypothesis of coding the memory pattern in oscillation relationships is true, it would explain the peak in performance at $\varrho = \alpha = 1$. A value $\varrho < 1$ would lead to a quick attenuation of oscillations; a value $\varrho > 1$ would lead to a quick expansion of oscillation amplitudes, which would enter the nonlinear parts of the tanh nonlinearity and lead to nonlinear confounding interactions between the oscillations; a value $\alpha < 1$ would lead to a slowdown of oscillations on the one hand (with likely detrimental effects on the decoding precision because we need radically different decodings at each timestep), and furthermore $\alpha < 1$ would attenuate oscillations if $\varrho = 1$.

Informed design of reservoirs. Based on these intuitions, I created specially designed reservoir matrices for the *smart* conditions, as follows. The rationale is to create reservoirs with a rich native mixture of oscillations. To

this end, I first randomly chose a set of N^{osc} period lengthes within a range $[P_{\mathsf{min}}, P_{\mathsf{max}}]$. Concretely, this interval was first partitioned into N^{osc} subintervals, from each of which one period length was uniformly sampled. This resulted in a collection $(p_1, \ldots, p_{N^{\mathsf{osc}}})$ of desired periods for the oscillation mix. Each of these periods was imprinted on a linear oscillator matrix

$$\mathbf{P}_i = \begin{pmatrix} \cos\,\phi_i & -\sin\,\phi_i \\ \sin\,\phi_i & \cos\,\phi_i \end{pmatrix},$$

where the rotation angle $\phi_i = p_i/2\,\pi$. To create $\mathbf{W}$ of size $K \times K$ (where $K >> 2\,N^{\mathsf{osc}}$), these 2×2 rotation matrices were placed on the beginning of the main diagonal of $\mathbf{W}$. The remaining "bulk" matrix area (of size $K - 2\,N^{\mathsf{osc}} \times K$) below this upper block-diagonal submatrix assigned to the oscillators was then sparsely filled with weights uniformly sampled from $[-1, 1]$. The sparsity was set to a value that on average every row received 10 nonzero entries. Finally, this raw matrix was scaled to the desired unit spectral radius. When a reservoir based on a matrix of this type is run with a small initial excitation, the oscillator submatrices will generate autonomous oscillations which decay only very slowly (since we are in the almost linear range of the tanh), and feed their oscillations forward into the remaining "bulk" of the reservoir, yielding a rich mixture (in phase and amplitude) of oscillations in the "bulk" units.

On a side note, having the oscillator-driven "bulk" part of the reservoir was important. Reservoir matrices that were built entirely from oscillators on the diagonal performed very poorly in this task (not documented here), much worse than plain random reservoirs.

For the $T_0 = 200, K = 200$ experiment, I used $N^{\mathsf{osc}} = 20$ and $[P_{\mathsf{min}}, P_{\mathsf{max}}] = [2, 8]$. For the $T_0 = 1000, K = 500$ experiment, I used $N^{\mathsf{osc}} = 30$ and $[P_{\mathsf{min}}, P_{\mathsf{max}}] = [2, 10]$. These were ad hoc settings with no optimization attempted.

Effects of smart reservoir design. The benefits of the smart reservoir preconditioning become apparent when one compares the performance of the $T_0 = 200, K = 200$ setup in the *basic* vs. the *smart* conditions (see Table 9.3 and Figure 7). The error distribution plots (left panels in the figure) reveal that across the 100 trials, in the *basic* condition the log2 maximal absolute test errors roughly range in $[-12, -0.3]$, while with *smart* reservoirs of the same size the range is $[-11 - 5]$. The high-error band from the *basic* condition is avoided in the *smart* experiment. Likewise instructive is an inspection of the range of the log2 of learnt output weights, which is about $[15, 30]$ vs. $[14, 21]$ in the two conditions (middle panels in the figure). Very large output weights, such as the ones obtained in the *basic* case, indicate that the solution-relevant components in the reservoir dynamics are strongly

correlated, or stated the other way round, the lower output weights found in the *smart* condition mean that the solution-relevant signal components are better separated from each other (in the sense of being less correlated). The middle panels in the figure also demonstrate that the model accuracy is closely connected to the smallness of learnt ouput weights. The *smart* reservoirs are superior to the *basic* ones in that they lead to a better directional separation of solution-relevant reservoir response signal components. This is one of the many things that the reservoir parlance of "rich reservoir dynamics" can mean.

Effects of very large reservoirs. The *blind* approach to this task was successful with a rather large reservoir of $K = 2000$. One effect of largeness is that random reservoirs will have less "individuality" than smaller ones; repeated trials with fresh reservoirs will yield more similar outcomes than when small reservoirs are used. This becomes apparent in the top row of Figure 7, which illustrates that the max error range and also the output weight sizes are more narrowly distributed than in the other 5-bit experiments that used smaller reservoirs. I refer the reader to recent theoretical work [3] which unveils how with growing network size, reservoir systems converge to infinite-dimensional recurrent kernel representations; in that work it is also empirically demonstrated that with reservoir sizes of about 20,000, the differences to infinite-dimensional "reservoirs" (which would all behave identically) almost vanishes.

Outliers. In the *basic* condition with $T_0 = 200, K = 500$, all but one trials were successful, and indeed were so with log2 maximal absolute errors of at most -4. However, there was one outlier reservoir with an extremely poor log2 error of about -0.3. This failing reservoir was parametrized in some "unlucky" way which exactly made it unsuitable for this task. This can always happen in reservoir computing (unless one adequately pre-configures reservoirs, as in the *smart* conditions). On a related note, almost all sorted max error distribution plots shown for the various experiments throughout this report exhibit a mirrored-S curve shape, with an initial quick rise, followed by a long and almost steady slope, followed by a terminal quicker rise. The quick rises at the good and poor performance ends mean that with random reservoir creation one sometimes – relatively rarely – stumbles across particularly good or exceptionally poor reservoirs. It would be interesting to inspect similar performance distributions for gradient-trained RNNs, where the training starts from different random parameter initializations. I would expect to see similar phenomena of relatively rare cases of particularly good or poor performance, and a similar weakening of this effect with growing network size.

Linear vs. nonlinear behavior. In this task, the *basic* condition manual optimization quickly and clearly led to very small input scalings except for the

cue signal. The small input scalings for the memory payload and distractor channels mean that the reservoir is essentially operating as a linear system until the cue spike appears, when it is driven into the nonlinear range of the tanh. The nonlinear behavior during the recall period appears to be important. When experiments were run with linear reservoirs (not reported here), performance was thoroughly poor. This task is dynamically interesting in that it apparently benefits from linearity in its memorizing phase – which is intuitively plausible – and nonlinearity in the decoding phase. Accordingly, the best uniform input scaling found in the *blind* setup strikes a compromise between linearity and nonlinearity.

Failing vs. successful trials. In the *basic* experiment with $T_0 = 200, K = 200$ there were 13/100 failing trials. An inspection of the reservoir dynamics dynamics revealed that a characteristic of failing trials was the absence of oscillations in the distractor phase. Figure 8 contrasts the evolution of a run in a failing trial with a run in a successful trial. This observation was one of the reasons why I went for a reservoir pre-construction with built-in oscillations in the *smart* setting.

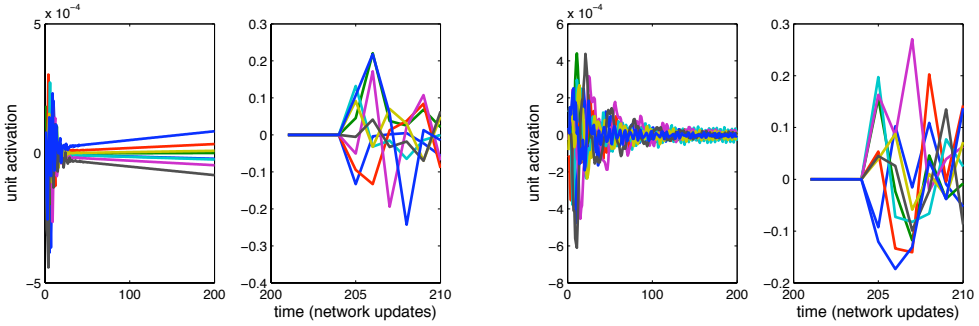

Figure 8: A typical failing run (left) contrasted with a typical successful run (right) in the 5-bit memory task.

10 The 20-Bit Memory Task

10.1 Task

This task is structurally identical to the 5-bit memory task, but now the memory pattern is more complex. Instead of 2 binary "payload" channels there are 5, and the length of the memory pattern is increased from 5 to 10. Again, at each of the ten pattern timesteps, exactly one of the 5 channels is randomly set to 1. This yields a set of 5^{10} different possible patterns, which is a bit more than 20 bit information per pattern and has given this task its accustomed name.

Again, in the interim period between the initial pattern presentation and the recall, on channel 6 a distractor input is constantly set to 1. Then, at the last

timestep before the desired recall, on channel 7 a single-timestep indicator signal is given. During the following 10 steps of recall the input again is set to the distractor. Thus there are 7 input channels. The length of the interim period is T_0, which leads to a total length of a run of $T_0 + 20$.

There are 7 output channels. The required output is a 1 on channel 6 for times $1 \leq n \leq T_0 + 10$, followed by a repetition of the memory pattern on channels 1–5 in the last 10 timesteps. Channel 7 is again without function and must always be zero.

The success criteria for runs and trials are analogous to the 5-bit memory task. A run is successful if at all timesteps and in all of the seven output channels, the absolute difference between the network output and the target output is less than 0.5. A trial is successful if less than 1 percent of its test runs is successful.

10.2 Experiments

This task turned out to be the most difficult of all by far. The following experiments were done:

1. Condition *blind*, $T_0 = 200$. A very large reservoir of size 8,000 was found necessary to give a reasonable performance. Since the RAM requirements of the pseudoinverse computation exceeded the available resources, the linear regression for the output weights was computed by the regularized Wiener-Hopf solution (4), with a regularizer $r^2 = 1e-20$ which was just big enough to render the matrix inversion numerically reliable.

2. Condition *basic*, $T_0 = 200$. A 2000-sized reservoir was now enough for very good performance, using the pseudoinverse method for the linear regression.

3. Condition *smart*, $T_0 = 300$. In order to demonstrate that reservoirs can deliver memory spans beyond 200, another 2000-sized, "smartly" preconfigured oscillator reservoir of similar makeup as in the 5-bit experiments was employed. It was equipped with $N^{\mathsf{osc}} = 50$ oscillators whose periods ranged in $[2, 10]$.

Given the slim RAM and time resources that I wanted to stay within (i.e. the limits of a standalone notebook and at most minutes per trial), I could not venture beyond $T_0 = 300$.

Due to the large number of possible patterns, training and testing sequences were randomly generated (different from the 5-bit task where *all* sequences could be used for training = testing). I used 500 training sequences for all 20-bit experiments.

In this task an output is required for every time step, that is, $T^{\mathsf{out}} = T_0 + 20$, and hence, the state collection matrix $\mathbf{S}$ (see Section 2) would grow to a very large size, rendering the linear regression expensive. Note that this was not a problem in the 5-bit case due to the small number of training sequences and smaller sizes of

20-bit Memory: parameters			
	blind $T_0 = 200$	*basic* $T_0 = 200$	*smart* $T_0 = 300$
K	8000	2000	2000
ϱ	1	1	1
$\sigma_1, \sigma_2, \sigma_3$	1e-7 1e-7 1e-7	1e-5 1e-6 1.0	*same*
α	1	1	1
N^{train}	500	500	500
20-bit Memory: results			
Mean CPU time (sec)	1700	313	121
Mean abs outweight	9.9e7	5.5e6	1.3e6
Mean abs train error	not computed	0.0010	0.00014
Mean abs test error	0.0080	0.00070	0.00010
Max abs test error	0.45	1.13	0.038
Nr failed trials	0	3	0

Table 5. Parameters and results of the 20-bit memory task. The CPU time for the *blind* condition refers to a 2.9 GHz, 8 GB RAM PC.

reservoirs. In order to economize, from each training run I only kept 30 states from the distractor period, discarding the remaining ones from the linear regression. These 30 states were picked from a time interval that was cyclically shifted over the interim period from run to run.

Table 10.2 and Figure 9 document the parameters and results.

References

[1] M. Hermans and B. Schrauwen. Memory in linear neural networks in continuous time. *Neural Networks*, 23(3):341–355, 2010.

[2] M. Hermans and B. Schrauwen. Memory in reservoirs for high dimensional input. In *Proc. WCCI 2010 (IEEE World Congress on Computational Intelligence)*, pages 2662–2668, 2010.

[3] M. Hermans and B. Schrauwen. Recurrent kernel machines: Computing with infinite echo state networks. *Neural Computation*, 24(1):104–133, 2012.

[4] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[5] H. Jaeger. 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.jacobs-university.de/hjaeger/pubs/EchoStatesTechRep.pdf.

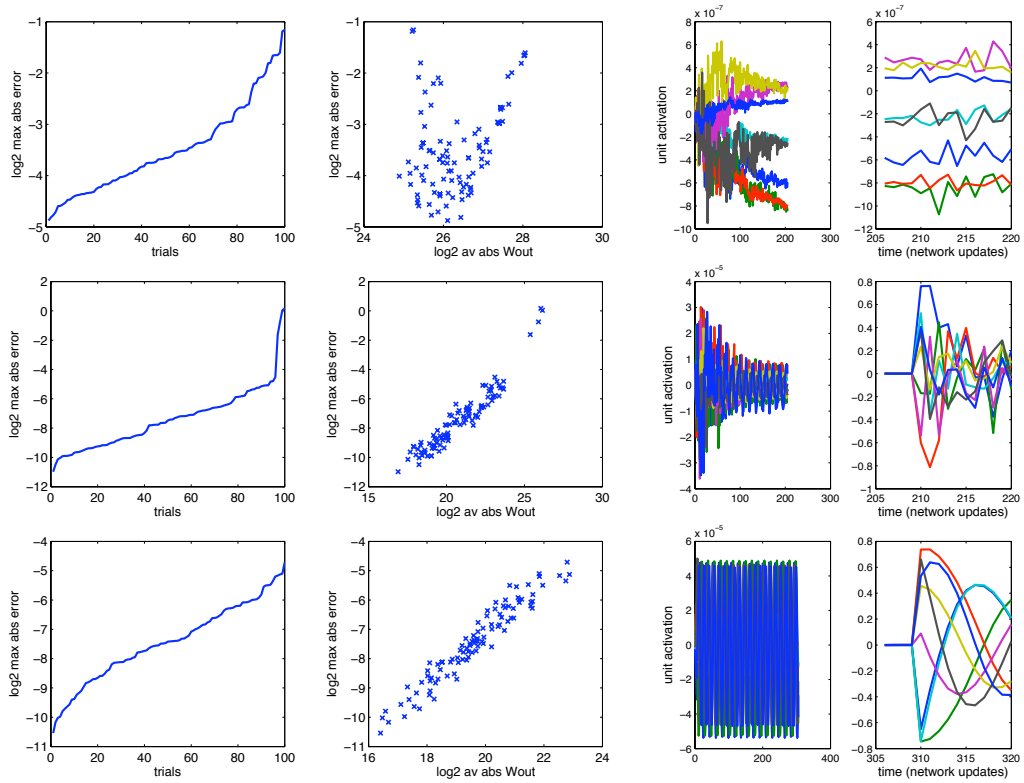

Figure 9: The 20-bit memory task. Left: log2 of the maximum absolute test errors obtained in the 100 trials, sorted. Center: max absolute test error vs. average absolute output weight size. Right: traces of 8 reservoir units in a typical run. From top to bottom: (i) condition *blind*, $T_0 = 200$, $K = 8000$; (ii) *basic*, $T_0 = 200$, $K = 2000$; (iii) *smart*, $T_0 = 300$, $K = 2000$.

[6] H. Jaeger. Short term memory in echo state networks. GMD-Report 152, GMD - German National Research Institute for Computer Science, 2002.

[7] H. Jaeger and D. Eck. Can't get you out of my head: A connectionist model of cyclic rehearsal. In I. Wachsmuth and G. Knoblich, editors, *Modeling Communication for Robots and Virtual Humans*, volume 4930 of *Lecture Notes in Artificial Intelligence*, pages 310–335. Springer Verlag, 2008.

[8] H. Jaeger, M. Lukosevicius, D. Popovici, and U. Siewert. Optimization and applications of echo state networks with leaky integrator neurons. *Neural Networks*, 20(3):335–352, 2007.

[9] Mantas Lukoševičius and Herbert Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, 2009.

[10] W. Maass, T. Natschläger, and H. Markram. Real-time computing without stable states: A new framework for neural computation

based on perturbations. *Neural Computation*, 14(11):2531–2560, 2002. http://www.cis.tugraz.at/igi/maass/psfiles/LSM-v106.pdf.

[11] J. Martens and I. Sutskever. Learning recurrent neural networks with Hessian-free optimization. In *Proc. 28th Int. Conf. on Machine Learning*, 2011.

[12] R. Pascanu and H. Jaeger. A neurodynamical model for working memory. *Neural Networks*, 24(2):199–207, 2011. DOI: 10.1016/j.neunet.2010.10.003.

[13] P.J. Werbos. Backpropagation through time: what it does and how to do it. *Proc. of the IEEE, October*, 78(10, October):1550–1560, 1990.

[14] O. L. White, D. D. Lee, and H. S. Sompolinsky. Short-term memory in orthogonal neural networks. *Phys. Rev. Lett.*, 92(14):148102, 2004.

[15] F. wyffels, B. Schrauwen, D. Verstraeten, and D. Stroobandt. Band-pass reservoir computing. In *Proc. IEEE Int. Joint Conf. on Neural Networks 2008*, pages 3204 – 3209, 2008.