Taming the Reservoir: Feedforward Training for Recurrent Neural Networks

Oliver Obst
Commonwealth Scientific and Industrial Research Organisation
ICT Centre, Adaptive Systems
Marsfield, NSW 2122, Sydney, Australia
Email: oliver.obst@csiro.au

Martin Riedmiller
University of Freiburg, Department of Computer Science
Machine Learning Lab
79110 Freiburg, Germany
Email: riedmiller@informatik.uni-freiburg.de

***Abstract*—Recurrent neural networks are successfully used for tasks like time series processing and system identification. Many of the approaches to train these networks, however, are often regarded as too slow, too complicated, or both. Reservoir computing methods like echo state networks or liquid state machines are an alternative to the more traditional approaches. Echo state networks have the appeal that they are simple to train, and that they have shown to be able to produce excellent results for a number of benchmarks and other tasks. One disadvantage of echo state networks, however, is the high variability in their performance due to a randomly connected hidden layer. Ideally, an efficient and more deterministic way to create connections in the hidden layer could be found, with a performance better than randomly connected hidden layers but without excessively iterating over the same training data many times. We present an approach – tamed reservoirs – that makes use of efficient feedforward training methods, and performs better than echo state networks for some time series prediction tasks. Moreover, our approach reduces some of the variability since all recurrent connections in the network are trained.**

I. Introduction

Recurrent neural networks (RNN) are successfully applied to tasks like time series prediction, filtering and system identification. Methods that are used to train RNN, like Backpropagation Through Time (BPTT) [1], involve many iterations over the training data and possibly also larger amounts of memory.

Echo state networks (ESN) [2], on the other hand, are a type of RNN that can be trained with relatively simple methods. A key element of ESN is the randomly constructed, fixed hidden layer, also called reservoir. In reservoir computing approaches like ESN or liquid state machines [3] typically only connections to output units are trained. Appeal and challenges of ESN [4] are the simple one-shot training methods that can be used, and their impressive performance for some tasks (see, e.g., [2]) on the one hand, but significant variation in performance [4]–[6] on the other.

To address this issue, self-organised approaches that help to improve randomly constructed reservoirs have been used [7]–[9]. These approaches are based on the idea to either maximise available information at each internal unit, or to maximise information transfer between units by changing the behaviour of individual units. Another option to improve randomly created reservoirs is to apply BBTT for one time step [6].

The approach presented here is different in the sense that it does not try to improve a randomly connected hidden layer, but instead in an intermediate step generates a feedforward network that uses input provided through a tapped delay line (also called a focussed time-delay neural network (FTDNN)). In a second step, we sample and store activations of the FTDNN hidden units. The third step uses these activations to train a recurrent layer to approximate the dynamics of the FTDNN. Even if the feedforward network is trained only for a fixed number of steps, activations in the hidden layer should be able to inform the design of reservoirs that perform better than randomly created ones. The final step uses the "tamed reservoir" obtained in that way to train the output layer similar to regular ESN, for example by using the Moore-Penrose pseudoinverse over the history of reservoir states multiplied by the desired output of the network.

In the following section, we start with some preliminaries and a high-level overview of our method. In Sect. III we give details about every step in the proposed approach. To demonstrate the performance of our method, we present results for some benchmarks, and conclude with a discussion.

II. Feedforward training with a tapped delay line

In theory, feedforward neural networks can learn to model relationships according to any mathematical function [10]. To model time series or dynamical systems however, some form of memory is needed. One way to provide this memory is by using recurrent connections. A more explicit method for providing memory to a feedforward network is to use a tapped delay line for the input time series. Dependent on the task, the delay line needs to be quite long in order to achieve acceptable performance. With increasing number of input units (i.e., longer delay lines), training times may grow thanks to the larger amount of data required. It is also necessary to explicitly choose the length of the delay line, in contrast to RNNs where the memory is provided implicitly through internal units. The idea behind our approach is that the internal dynamics of a (possibly partially) trained feedforward network with tapped delay line can be used to train the hidden layer of a recurrent network that is treated as an ESN otherwise. Before we give an high level overview of our approach, we

start with some preliminary notation and definitions in the following subsection.

A. Preliminaries

Our RNN model consists of K input units, N units in the hidden layer, and L output units. The respective activations at time step n are:

$$\mathbf{u}(n) = (u_1(n), ..., u_K(n)),$$
$$\mathbf{x}(n) = (x_1(n), ..., x_N(n)),$$
$$\mathbf{y}(n) = (y_1(n), ..., y_L(n)),$$

For the experiments in this paper, we used univariate input, with an extra input unit used for a constant bias (i.e., $K = 2$), however, multivariate input is also possible. It is also possible to attach several output units, each for a different task, to the network (in the same way this is possible for ESN), however, we present our approach to train the hidden layer for a single task.

Three connection matrices are used: $\mathbf{W}^{\text{in}}$ is a $N \times K$ matrix from input to hidden layer, $\mathbf{W}$ the recurrent $N \times N$ weight matrix, and $\mathbf{W}^{\text{out}}$ the $L \times N$ output weight matrix (i.e., we use no connections from input units to output, and no feedback from the output back into the network).

The hidden layer is then updated according to

$$\mathbf{x}(n+1) = \tanh(\mathbf{W}^{\text{in}}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n)). \quad (1)$$

The output uses linear units and is computed as

$$\mathbf{y}(n+1) = \mathbf{W}^{\text{out}}\mathbf{x}(n+1). \quad (2)$$

Assuming univariate input, the feedforward network for the intermediate learning step consists of $\hat{K} = 1 + \ell$ input units, where ℓ is the length of the tapped delay line. The extra input unit is used for a bias, all other input units contain data from the current and the past $\ell - 1$ steps. Activation of input units at time step n are then

$$\hat{\mathbf{u}}(n) = (u(n-\ell+1), ..., u(n), 1). \quad (3)$$

We use $\hat{\mathbf{u}}^*(n) = (u(n-\ell+1), ..., u(n))$ to denote the tapped delay line activations without the extra bias.

With N the number of hidden units in the RNN to be trained, and tapped delay line length ℓ, we use $\hat{N} = N - \ell$ hidden units in the feedforward network. The number of output units L in the feedforward is the same as in the RNN.

The input weight matrix $\hat{\mathbf{W}}^{\text{in}}$ consists of $\hat{N} \times \hat{K}$ elements; output weight matrix $\hat{\mathbf{W}}^{\text{out}}$ has the size $L \times \hat{N}$. The update equations for the hidden layer are

$$\hat{\mathbf{x}}(n) = \tanh(\hat{\mathbf{W}}^{\text{in}}\ \hat{\mathbf{u}}(n)), \quad (4)$$

and for the output

$$\hat{\mathbf{y}}(n) = \hat{\mathbf{W}}^{\text{out}}\hat{\mathbf{x}}(n), \quad (5)$$

respectively.

B. Overview of our training method

The following is an overview of all steps in our approach. Each step is then expanded in a subsection of the next section.

1) Training input $\mathbf{u}^{\text{train}}$ together with the desired output $\mathbf{d}^{\text{train}}$ is used to train a FTDNN. In our approach we stop the training after a fixed number of epochs, i.e., the feedforward network may not be fully trained after this step.
2) The training input $\mathbf{u}^{\text{train}}$ is used to drive the trained FTDNN for T steps. At each step, state information from the $\hat{N}$ units is collected in two $(T-1) \times \hat{N}$ matrices $\mathbf{S}_x$ and $\mathbf{S}_y$, with $\mathbf{S}_x(n)$ the state of the units at step n before passing it through the nonlinearity, and $\mathbf{S}_y(n)$ after the nonlinearity at the same step. As we will describe later, it is beneficial to extend the training input data in this step by a larger, randomly generated time series.
3) Together with the training input $\mathbf{u}^{\text{train}}$, matrices $\mathbf{S}_x$ and $\mathbf{S}_y$ are used to train $\mathbf{W}^{\text{in}}$ and $\mathbf{W}$ using linear regression.
4) Training input $\mathbf{u}^{\text{train}}$, and desired output $\mathbf{d}^{\text{train}}$ are used to train the output weights $\mathbf{W}^{\text{out}}$ of the RNN with input weights $\mathbf{W}^{\text{in}}$ and recurrent weights $\mathbf{W}$. At this point, the RNN is treated as an ESN, so that $\mathbf{W}^{\text{out}}$ can be trained using linear regression like any other ESN. $\mathbf{W}^{\text{in}}$ and $\mathbf{W}$ remain unchanged from the last step.

All training methods used in the steps above a either simple regression or backpropagation as used in feedforward networks. The backpropagation is not necessarily fully completed but only run for a few steps.

III. From Tapped Delay Line to Recurrent Connections

In this section, we detail each of the four steps of our method.

A. Step 1: Training the intermediate feedforward network

A first step of our approach consists of creating a feedforward neural network with a tapped delay line as an input. Parameters to choose are the length of the tapped delay line ℓ, and the number of epochs for the training. The number of hidden units is set to the number of hidden units of the recurrent reservoir N. With an additional bias unit, the network has $\ell + 1$ input units, and the number of output units L is determined by the training data (in our experiments, we used one output unit). In the hidden layer, we did not use any bias unit to match the structure of the feedforward network with the recurrent network that we would like to create (Alternatively, using a bias unit in the hidden layers of both the RNN as well as in the FTDNN is also possible). The hidden layer units are nonlinear ($\tanh$) units, and for the output we are using linear units.

Assuming we have univariate input data, one point at a time is fed into the delay line. Multivariate input requires a separate delay line for each dimension. The network is trained using resilient backpropagation (RPROP) [11] in batch mode, for a fixed number of epochs. RPROP is an improvement to standard back propagation, and converges faster thanks to

adaptive update values. In RPROP, the sign of the derivative is used to indicate the direction of the weight update with an adaptive update value $\Delta_{ij}^{(n)}$:

$$\Delta w_{ij}^{(n)} = \begin{cases} -\Delta_{ij}^{(n)} & , \text{ if } \frac{\delta E^{(n)}}{\delta w_{ij}} > 0 \\ +\Delta_{ij}^{(n)} & , \text{ if } \frac{\delta E^{(n)}}{\delta w_{ij}} < 0 \\ 0 & , \text{ else.} \end{cases} \quad (6)$$

In batch mode, $\frac{\delta E^{(n)}}{\delta w_{ij}}$ denotes the summed gradient information over all patterns of the training set. The update value $\Delta_{ij}^{(n)}$ is then determined in a second step:

$$\Delta_{ij}^{(n)} = \begin{cases} \eta^+ * \Delta_{ij}^{(n-1)} & , \text{ if } \frac{\delta E^{(n-1)}}{\delta w_{ij}} * \frac{\delta E^{(n)}}{\delta w_{ij}} > 0 \\ \eta^- * \Delta_{ij}^{(n-1)} & , \text{ if } \frac{\delta E^{(n-1)}}{\delta w_{ij}} * \frac{\delta E^{(n)}}{\delta w_{ij}} < 0 \\ \Delta_{ij}^{(n-1)} & , \text{ else,} \end{cases} \quad (7)$$

with $0 < \eta^- < 1 < \eta^+$. For further details, we refer to [11].

In our experiments, we used a fixed number of epochs to train the feedforward network, i.e., the network is not necessarily fully trained. Dependent on the problem, longer training times may further increase performance of our approach. The initial weights for the FTDNN were drawn uniformly at random, $\hat{w}_{i,j}^{\text{in}} \in (-0.1, 0.1)$, and $\hat{w}_{i,j}^{\text{out}} \in (-0.2, 0.2)$. The learned input weights are the essential output of this step, the quality of the output weights is not important since they will not be used in the subsequent steps.

B. Step 2: Collect state information from the feedforward network

After stopping the training of the feedforward network, the same training data $\mathbf{u}^{\text{train}}$ is used once again to drive the network and to collect information on internal network states in $(T-1) \times \hat{N}$ state collection matrices $\mathbf{S}_x$ and $\mathbf{S}_y$. For each step $n = 2, ..., T$, the total input to each of the hidden units is collected in $\mathbf{S}_x$, i.e., $\mathbf{S}_x(n) = \hat{\mathbf{W}}^{\text{in}} \hat{\mathbf{u}}(n)$. For each step $n = 1, ..., T-1$, we collect the states after they passed through the nonlinearity in $\mathbf{S}_y$, i.e., $\mathbf{S}_y(n) = \tanh(\hat{\mathbf{W}}^{\text{in}} \hat{\mathbf{u}}(n))$ from each step). We use these matrices containing the collected states to train input and recurrent weights in step 3.

If the time series available for training is short, random data can be used in this step to obtain larger state collection matrices. These random data may help to fully explore the state space of the FTDNN, and also serve regularisation. In our experiments, the use of additional random data helped to both improve individual results, as well as to prevent unfavourably large values in the computed recurrent weight matrices that lead to unstable solutions.

C. Step 3: Train RNN input and recurrent weight matrices

In this step, $\mathbf{S}_x$ and $\mathbf{S}_y$ are used to train the input and the recurrent connections of the new network. More concretely, we use extended matrices $\hat{\mathbf{S}}_y$ and $\hat{\mathbf{S}}_x$, with $\hat{\mathbf{S}}_y(n) = (\mathbf{S}_y(n),\, \hat{\mathbf{u}}(n),\, u(n))$ as input to a one step prediction of the (known) successor state before the nonlinearity $\hat{\mathbf{S}}_x$, with $\hat{\mathbf{S}}_x(n) = (\mathbf{S}_x(n),\, \hat{\mathbf{u}}^*(n))$. Since the history of all states is known, it can be used, together with input activations and bias, to compute (estimate) the successor state of the network, i.e., the input to all hidden units at the next time step $\hat{\mathbf{S}}_x(n+1)$. We can use the collected state information $\hat{\mathbf{S}}_x$ and $\hat{\mathbf{S}}_y$ over a number of steps to compute a set of weights $\mathbf{W}^{\text{all}}$ that approximate this update by regression, e.g., by using the Moore-Penrose pseudoinverse:

$$\mathbf{W}^{\text{all}} = (\hat{\mathbf{S}}_y^{-1}\, \hat{\mathbf{S}}_x)^t. \quad (8)$$

From $\mathbf{W}^{\text{all}}$, we can now extract the recurrent weights matrix $\mathbf{W}$, and the input weight matrix $\mathbf{W}^{\text{in}}$ for our RNN:

$$\mathbf{W} = \mathbf{W}^{\text{all}}_{1..n,1..n} \quad (9)$$

$$\mathbf{W}^{\text{in}} = \mathbf{W}^{\text{all}}_{1..n,(n+1)..(n+2)} \quad (10)$$

Extending the state matrices obtained from the FTDNN by the states of the tapped delay line leads to an embedding of the delay line in the hidden layer of the resulting tamed reservoir. It is also possible to *only* use internal states of the FTDNN and the current input into the network. In this case, the size of the FTDNN hidden layer needs to match the desired size of the hidden layer in the tamed reservoir, and the extended state matrices $\hat{\mathbf{S}}_y$ and $\hat{\mathbf{S}}_x$ are $\hat{\mathbf{S}}_y(n) = (\mathbf{S}_y(n),\, 1,\, u(n))$ and $\hat{\mathbf{S}}_x(n) = \mathbf{S}_x(n)$, respectively. In this case, we aim to predict the next network state before the nonlinearity $\mathbf{S}_x(n+1)$, using the previous network state after the nonlinearity, the bias unit, and the current input value $u(n)$ (but not the delay line $\hat{\mathbf{u}}$).

Different methods for regression and regularisation may also be used to enforce particular constraints on the resulting weight matrices.

D. Step 4: Train RNN output weight matrix

In the final step of our approach, we now use the training data $\mathbf{u}$, and the weight matrices $\mathbf{W}$ and $\mathbf{W}^{\text{in}}$ to collect state information from the pre-trained reservoir. For T training steps, we obtain a $T \times N$ state matrix $\mathbf{M}$ (commonly, a number of initial states are discarded to wash out initial transients). Like in the previous step to compute the recurrent weights, we can use the pseudoinverse method to compute our output weight matrix:

$$\mathbf{W}^{\text{out}} = \mathbf{M}^{-1}\mathbf{d}. \quad (11)$$

With this step, all weight matrices of the RNN have been computed, and the network is fully trained.

IV. Experimental results

To evaluate our approach, we have conducted two different kinds of experiments:

1) With three different time series, we perform a one-step-ahead prediction with both tamed reservoirs and ESN. Three data sets have been used for each task: a training set, a validation set to adjust hyper-parameters as well as to select the best performing network from a set, and finally a test set for the actual evaluation of such selected networks.
2) Our second experiment is a comparison of both approaches using data from arbitrary feedforward networks with a delay line.

Even though the number of units used varies between the tasks, ESN and tamed reservoirs always use the same number of units when directly compared against each other. For the intermediate FTDNN, the number of units in the hidden layer is reduced by the number of units in the tapped delay line. For all tasks, we measure the performance using the normalised root mean squared error

$$\text{NRMSE} = \frac{\sqrt{\langle(\tilde{y}(n) - y(n))^2\rangle_n}}{\langle(y(n) - \langle y(n)\rangle_n)^2\rangle_n}, \tag{12}$$

where $\tilde{y}(n)$ is the sampled output and $y(n)$ is the desired output.

A. Selection of hyper-parameters

A number of hyper-parameters have to be selected specific to each of the task in the following subsection. Our first step consists in a search for the ESN reservoir size N, and for the spectral radius of the connectivity matrix $\rho(\mathbf{W})$ for which, on average, 10 randomly generated ESN perform best on a validation set. This search is performed over the following configuration space:

- hidden layer size $N \in [40..130]$ in steps of 10,
- spectral radius $\rho(\mathbf{W}) \in [0.1, 1.0]$, in steps of 0.1.

ESN input weights are fully connected to the reservoir and randomly initialised, $w^{\text{in}}_{i,j} \in (-1, 1)$. All weights are drawn from a uniform random distribution, and the reservoirs are densely connected.

In a second step, the remaining hyper-parameters for the tamed reservoir have to be selected. The number of units available is already determined by the ESN reservoir size; hyper-parameters left to choose are the length of the delay line for the intermediate FTDNN, as well as the RPROP parameters η^+ and η^-. We use random search (see also [12]) over the following configuration space:

- delay line length $\ell \in [1..30]$,
- $\eta^- \in (0, 1)$, and $\eta^+ \in (1, 2)$.

In our search we evaluate 500 randomly generated configurations. Initial weights for the FTDNN are drawn from a uniform random distribution ($\hat{w}^{\text{in}}_{i,j} \in (-0.1, 0.1)$, $\hat{w}^{\text{out}}_{i,j} \in (-0.2, 0.2)$) — a source of variation in performance. For each configuration, we create 10 networks, use them for training with RPROP, and select the configuration that performs best on average on our validation set.

A hyper-parameter that we did not explore in our search is the amount of additional random data to drive the FTDNN in step 2 of our approach. In addition to the respective training set, we use time series with 100000 values drawn at random from a normal distribution $\mathcal{N}(0, 2)$ in all experiments on the one-step-ahead prediction tasks.

B. One-step-ahead prediction

For each of the one-step-ahead prediction tasks, we train 10 newly created tamed reservoirs as well as 10 newly created ESN, using the hyper-parameters as determined in the previous step. The tamed reservoir and the ESN that perform best on our validation set are selected for comparison against each other, using the test set. This test is repeated 10 times, with results for each task reported in Table I.

TABLE I
TIME SERIES PREDICTION RESULTS

NRMSE (σ)	Electrical	Laser	AR(3)
Average best			
- Tamed RNN	**0.3573** (0.0068)	**0.0623** (0.0019)	**0.1716** (0.0006)
- ESN	0.4261 (0.0108)	0.1470 (0.0025)	0.1724 (0.0006)
- FTDNN	0.8283 (0.0982)	0.5548 (0.0208)	0.2138 (0.0212)
Absolute best			
- Tamed RNN	**0.3479**	**0.0591**	**0.1706**
- ESN	0.4120	0.1426	0.1705
- FTDNN	0.7211	0.5310	0.1815

a) *Task 1:* Electrical load prediction

The data used for this task represent the daily electrical consumption in Poland [13]. We use 700 values of the series for testing (see also Fig. 1), and two series of 350 values each as a validation and test set, respectively. The hyper-parameters found to work best for ESN are a reservoir size of 90 units, and a spectral radius of 1.0. Parameters found for the FTDNN are a tapped delay line length of 23 units, resulting in 67 remaining hidden units, and RPROP parameters $\eta^+ = 1.0083$, and $\eta^- = 0.3428$.

For one trial, we train 10 tamed reservoirs as well as 10 ESN, and select the ones that perform best on our validation set. We record results of the best performing networks from 10 different trials. This results in an average best NRMSE for tamed reservoirs of 0.3573, with a standard deviation $\sigma = 0.0068$. The absolute best NRMSE is 0.3479. The average best NRMSE for ESN is 0.4261, with $\sigma = 0.0108$. Absolute best NRMSE of the tested ESN is 0.4120. The difference in performance is statistically highly significant ($p \approx 3 \cdot 10^{-11}$). The performance of the intermediate average best FTDNN on the test set is considerably worse than both recurrent networks, with an NRMSE of 0.8283 ($\sigma = 0.0982$).

b) *Task 2:* Santa Fe Laser Data

From the Santa Fe Laser Data set (set A.cont), we create a training test series (4500 values), and validation and test series of 2250 values each (see Fig. 2 for a sample interval of the data, along with a small interval of predictions from both a tamed reservoir, and an ESN). Based on our hyper-parameter search, we use 130 reservoir units and a spectral radius of 0.4 for the ESN.

For our tamed reservoirs, we train FTDNN with a delay line of length 10, with 120 hidden units, and RPROP parameters $\eta^+ = 1.0937$, and $\eta^- = 0.5859$, respectively.

From 10 trained tamed reservoirs, we select the best performing on the validation set. Using the test series, we measure performance of the network and repeat this procedure 10 times. The average best performing tamed reservoir has a test error of 0.0623, with standard deviation $\sigma = 0.0019$. The NRMSE

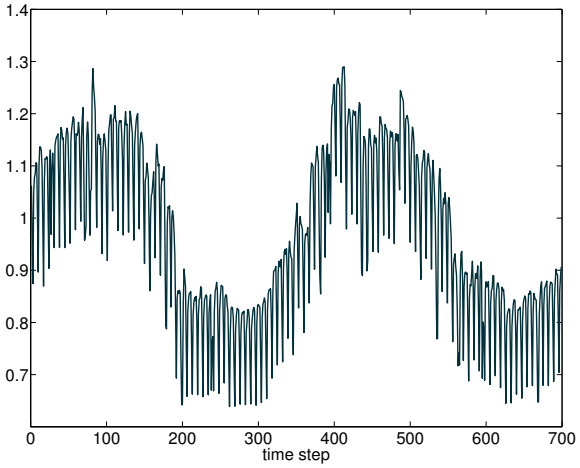

Fig. 1. Training interval of the electrical load prediction data. 700 values of the data set have been used for training, 350 for validation, and 350 for testing.

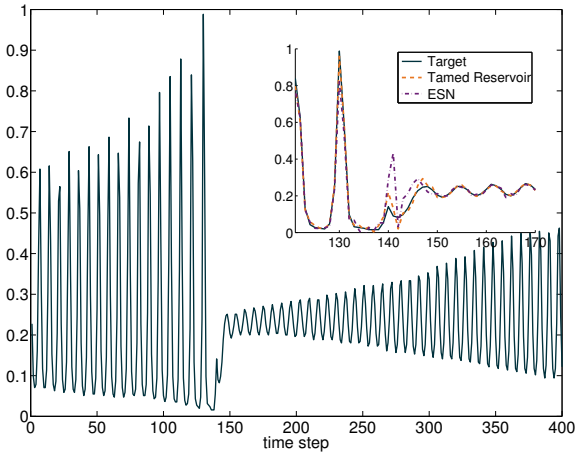

Fig. 2. Sample interval of the Santa Fe laser data. 4500 values from the data set have been used for training, and 2250 for testing. Inset: a small interval of the target, with approximations from both a tamed reservoir and an ESN.

of the absolute best performing tamed reservoir is 0.0591.

The same procedure for ESN yields an NRMSE of 0.1470, with $\sigma = 0.0025$ for the average best ESN, for the absolute best ESN the NRMSE is 0.1426. The difference between these two results is statistically highly significant ($p \approx 3 \cdot 10^{-23}$).

For comparison, the test NRMSE of the average best feedforward network used in the intermediate step was 0.5548, $\sigma \approx 0.02$ after 50 epochs of RPROP.

c) Task 3: Autoregressive Model ar(3)

Prediction of a time-series created using the following autoregressive model of order 3 is a simple task:

$$y(n) = a_1\, y(n-1) + a_2\, y(n-2) + a_3\, y(n-3) + \epsilon, \quad (13)$$

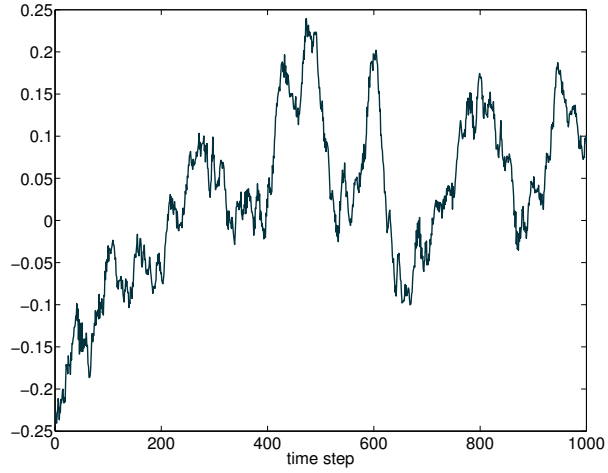

Fig. 3. The AR(3) training data used for task 3.

with $a_1 = 0.8$, $a_2 = 0.6$, $a_3 = -0.41$, and $\epsilon \sim \mathcal{N}(0, 0.01)$. It is nevertheless interesting to use this series, because for a comparison the optimal prediction can be computed using (13) without the noise term. We create a training set, a validation set, and a test set with 1000 values each (see Fig. 3 for a sample interval of the data). The error of the optimal prediction on our test set computes to 0.1688. Hyper-parameters as a result of our search are: reservoir size of 40 units, a spectral radius for the ESN of 0.3, RPROP parameters $\eta^+ = 1.3456$, and $\eta^- = 0.1347$, respectively, as well as FTDNN with 37 hidden units and, perhaps unsurprisingly, 3 units for the tapped delay line.

The average best results for both tamed reservoirs and ESN are close to the optimal prediction. With NRMSE of 0.1716 and 0.1724, respectively, and $\sigma = 6 \cdot 10^{-4}$ for both, the difference between the two approaches is statistically significant ($p < 0.01$), but practically hardly relevant. NRMSE of the absolute best performing networks are 0.1706 for tamed reservoirs, 0.1715 for ESN, and 0.1815 for the intermediate FTDNN.

C. Comparison for various random time series

To compare the approach independently of the specific feedforward training, we use randomly created feedforward networks with tapped delay lines, and random input to create training data for our method. The task for the recurrent networks is then to approximate the output of the feedforward network as closely as possible. The test procedure is as follows:

1) Choose a tapped delay line of length ℓ.
2) Choose a number of internal units N.
3) Create a randomly connected feedforward neural network with ℓ input units, N internal units, and 1 output unit.

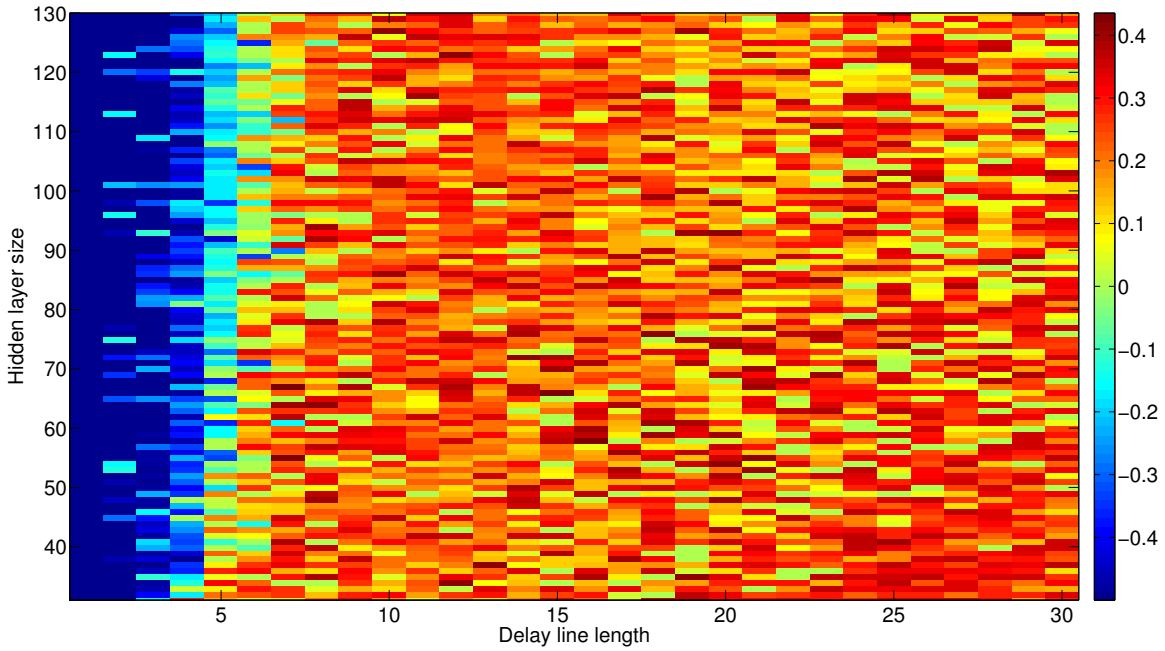

Fig. 4. Comparison of tamed reservoirs against ESN. The task was to approximate the output of a FTDNN with a tapped delay line of lengths 1...30 (delay line length on the x-axis). The number of hidden units in the reservoirs was chosen between 31 and 130 (on the y-axis). Colours in the plot represent the difference in performance (NRMSE). Negative values indicate an advantage for ESN. For this comparison, all NRMSE above 0.5 have been considered to be equal. For each comparison, 10 ESN have been created from which the best one was used for evaluation.

4) Create a random "training" sequence $\mathbf{u}^{\text{train}}$, and a random test sequence $\mathbf{u}^{\text{test}}$, uniformly from the interval $0..1$.
5) Drive the network using $\mathbf{u}^{\text{train}}$ to obtain $\mathbf{d}^{\text{train}}$, and using $\mathbf{u}^{\text{test}}$ to obtain $\mathbf{d}^{\text{test}}$.
6) Use the set $\mathbf{u}^{\text{train}}$ and $\mathbf{d}^{\text{train}}$ for training of a RNN with our approach, as well as for training of 10 ESN for comparison. For the tamed reservoir training, no additional random data is used in sampling FTDNN states. The ESN are initialised with random spectral radii $\in (0, 1)$.
7) The tamed reservoirs as well as the ESN are then evaluated using the test set. We compare the tamed reservoir against the best NRMSE out of the 10 generated ESN.

Figure 4 shows an overview of the error differences between tamed reservoirs and ESN. From this graph, it can be seen that for a large majority of the trials the trained RNN performs better than ESN. In total, we performed 3000 comparisons (with 10 ESN trained for each), in which our approach performs better in 2430 of them (approx. 81%). One may also argue that for cases where NRMSE are larger than 0.5 for both approaches, neither of them manages to approximate the data well enough to be considered. In 155 cases, the NRMSE was equal to or larger than 0.5 for both approaches. The tamed reservoirs performed better than the best of 10 ESN in 2336 out of the remaining 2845 cases ($\approx 82\%$). Independently of the reservoir size, ESN perform better when we used short delay lines to generate the training data.

In Fig. 5, we plot the NRMSE of the tamed reservoir for each of the trials (capped to 0.5, i.e., all NRMSE greater than 0.5 were considered equally unsuccessful).

V. Conclusion

We presented tamed reservoirs, a novel approach for training the internal layer of a recurrent neural network. Our training procedure makes use of efficient feedforward training methods, and uses simple regression for training of all recurrent weights. For all presented example time series, our approach significantly improved results over the best performing ESN. In the same way the RNN training is also an improvement over the FTDNN networks, with only a few extra steps from where the feedforward training stopped. Moreover, since all weights in the recurrent network are trained, our approach removes also some of the randomness that comes with reservoir computing approaches, shown by the slightly smaller variance of tamed reservoirs compared to ESN. However, with the random initialisation of the intermediate feedforward network used during training, and possible local minima found during its training, a certain amount of non-determinism remains.

Our results seem surprising from several point of views: for the first two tasks of the one-step-ahead prediction experiment, performance of the intermediate FTDNN is not even close to performance of the ESN, but nevertheless approximating the dynamics of their hidden layer in our tamed reservoirs helps to substantially improve performance over ESN. One point to note is that the performance of the FTDNN is not necessarily

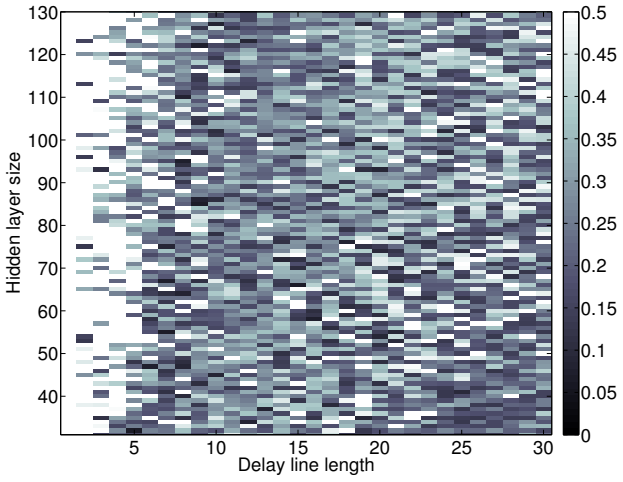

Fig. 5. NRMSE of tamed reservoirs over synthetical data generated from a FTDNN with tapped delay lines of varying lengths (NRMSE capped to 0.5). Darker colours represent better performance.

correlated with performance of the derived tamed reservoirs: we do not make any use of the feedforward output weights in our training, but these weights would have substantial impact on the FTDNN results. It may also surprise that a finite, fixed-size history of the inputs improves the prediction performance. The partial training of the feedforward input weights may direct the dynamics (and the resulting tamed reservoir) into the "right" direction, so that it becomes easier to learn output weights, i.e., the use of the FTDNN helps to learn features of the recent past. The subsequent steps in training can then make use of these features. It remains to be tested if this technique also helps for longer term predictions, or completely different tasks. For one-step-ahead predictions, the infinite, decaying memory provided by recurrence appears to contribute to a lesser extent to the prediction.

It should also be noted that some applications require larger hidden layers than the 40 – 130 unit reservoirs that we tested. Figure 5 seems to suggest that performance of the tamed reservoirs is particularly good for smaller reservoirs when the task requires long short-term memory. On the other hand, from Fig. 4 we can see that tamed reservoirs also perform well compared to ESN towards the larger reservoir sizes, when the task requires a short-term memory of more than 10 units.

These considerations suggest an investigation of different time series, and along with that also effects of varying delay line lengths and reservoir sizes. Dependent on the time series, different types of delay lines may turn out to be more useful, for example, the gamma model of de Vries and Principe [14].

Another possibly useful extension of the approach may be to pre-train only a part of the recurrent weights, and have a number of weights randomly connected. In particular for very small reservoirs, or for shorter delay lines used during training, forcing all weights so that all units approximate specific time series becomes very difficult if the constraints are too rigid. Some extra units that do not have to be trained may help in removing some of the rigidity (but also re-introduce more variance).

ACKNOWLEDGMENT

The authors would like to thank the High Performance Computing and Communications Centre (http://www.hpccc.gov.au/) for the use of their super-computer cluster in performing some of the experiments for this paper. MR gratefully acknowledges a grant from the CSIRO OCE Distinguished Visiting Scientist Scheme.

REFERENCES

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning internal representations by error propagation.* Cambridge, MA, USA: MIT Press, 1986, pp. 318–362.

[2] H. Jaeger and H. Haas, "Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication," *Science*, vol. 304, no. 5667, pp. 78–80, 2004. [Online]. Available: http://www.sciencemag.org/cgi/content/abstract/304/5667/78

[3] W. Maass, T. Natschläger, and H. Markram, "Real-time computing without stable states: A new framework for neural computation based on perturbations," *Neural Computation*, vol. 14, no. 11, pp. 2531–2560, 2002.

[4] D. Prokhorov, "Echo state networks: appeal and challenges," in *IEEE International Joint Conference on Neural Networks (IJCNN '05)*, vol. 3, 2005, pp. 1463–1466.

[5] J. Boedecker, O. Obst, N. M. Mayer, and M. Asada, "Initialization and self-organized optimization of recurrent neural network connectivity," *HFSP Journal*, vol. 3, no. 5, pp. 340–349, Oct. 2009. [Online]. Available: http://dx.doi.org/10.2976/1.3240502

[6] M. Hermans and B. Schrauwen, "One step backpropagation through time for learning input mapping in reservoir computing applied to speech recognition," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, Jun. 2010, pp. 521–524.

[7] J. Triesch, "A gradient rule for the plasticity of a neuron's intrinsic excitability," in *Proceedings of the International Conference on Artificial Neural Networks (ICANN 2005)*, ser. Lecture Notes in Computer Science, W. Duch, J. Kacprzyk, E. Oja, and S. Zadrozny, Eds. Springer, 2005, pp. 65–70.

[8] J. J. Steil, "Online reservoir adaptation by intrinsic plasticity for backpropagation-decorrelation and echo state learning," *Neural Networks*, vol. 20, no. 3, pp. 353–364, Apr. 2007.

[9] O. Obst, J. Boedecker, and M. Asada, "Improving recurrent neural network performance using transfer entropy," in *Neural Information Processing. Models and Applications*, ser. Lecture Notes in Computer Science, K. W. Wong, B. S. U. Mendis, and A. Bouzerdoum, Eds. Springer, 2010, vol. 6444, pp. 193–200.

[10] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organisation in the brain," *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958.

[11] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: the RPROP algorithm," in *IEEE International Conference on Neural Networks*, vol. 1, 1993, pp. 586–591.

[12] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, pp. 281–305, 2012.

[13] M. Cottrell, B. Girard, and P. Rousset, "Forecasting of curves using a kohonen classification," *Journal of Forecasting*, vol. 17, no. 5-6, pp. 429–439, 1998.

[14] B. de Vries and J. C. Principe, "The gamma model – a new neural model for temporal processing," *Neural Networks*, vol. 5, no. 4, pp. 565–576, 1992.