Exploration of combining ESN learning with gradient-descent RNN learning techniques
A Bachelor of Science thesis

by

Dumitru Erhan

d.erhan@iu-bremen.de

A thesis submitted in partial satisfaction of the

requirements for the degree of

Bachelor of Science (BSc.)

in the

School of Engineering & Science

of the

INTERNATIONAL UNIVERSITY BREMEN

Supervisor:
Herbert Jaeger
Spring Semester 2004

Executive Summary

Modeling dynamical systems via neural networks has become a well-established field of research in Computer Science. However, for more complex dynamical systems the current algorithms are often impractical or not accurate enough. Recurrent Neural Networks (RNNs), when coupled with the Echo-State Network (ESN) approach, offer an efficient way of solving such kind of problems – by concentrating on the behavior that can be observed from the outside and modeling it, while ignoring the underlying processes. This biologically inspired method produces very good results for a variety of problems and, in many cases, out-beats other algorithms' results by orders of magnitude. However, for certain scenarios, the ESN algorithm requires large networks in order to operate at its full potential. This is not desirable in those situations where the physical or computational requirements limit the size of network to be used (such as the case of micro-controllers and embedded devices). The aim of this project was to provide a remedy to this problem by employing a "post-processing" technique on the solutions generated by the ESN algorithm. This technique takes the form of a traditional algorithm for learning with RNNs. While these algorithms are normally slow and tend to produce worse results, they require small networks in order to operate properly. This way we found a compromise solution that helped us show that it is possible to employ the ESN algorithm even on small networks and still obtain good results by means of "post-processing" them with other, more traditional, algorithms.

Contents

1 Summary Description and Motivation 4
2 Theoretical Setup 5
2.1 Echo-State Network Algorithm 7
2.2 Gradient-Descent Algorithms and RTRL 9
2.3 Interconnection of ESN and RTRL 11
3 Experimental Setup 13
3.1 The Dataset 13
3.2 The Network 14
3.3 Interconnection Details 15
4 Experimental Results 15
4.1 Combination of ESN and RTRL 16
4.2 Verification Runs 17
4.3 Further Issues 17
5 Conclusions and Discussion 18
6 Future Research 19
1 Appendix 19
1.1 Results of First Set of Experiments 19
1.2 Results of Second Set of Experiments 19

Bibliography22

1 Summary Description and Motivation

Training Recurrent Neural Networks (RNNs) is usually done via the so-called "gradient-descent techniques". These algorithms change the synaptic weights of the entire network in an incremental fashion until the algorithm converges to some, usually local, minimum on the error surface. The Echo-State-Network approach differs notably in this respect – in that only the synaptic weights that go towards the output units are modified by the algorithm; the others remain fixed. The algorithm assumes that the output of the network can be expressed by a linear combination of the states on the internal units. In practice, this means that the actual learning is, from a mathematical point of view, just a linear regression task – a drastic reduction in complexity as compared to the other techniques.

Such a reduction in the computation time required for training and learning with RNNs comes, however, at a certain cost. In order for the ESN algorithm for to operate successfully, a network must satisfy certain conditions:

- It has to possess the so-called "echo-states". This means that the activations of the internal units have to "echo" the input (and the output) to the network (more about it in Section 2.1 or in [1, 2]).
- The network has to be large and inhomogeneous enough in order to be able to provide for a large (so-called) "Dynamical Reservoir" of "echo-states".

In applications where a large network is impractical or undesirable, our only choice for training RNNs would be to employ a standard gradient-descent algorithm. A small network cannot normally provide a rich enough Dynamical Reservoir for the ESN algorithm to operate at its best. But, in practice, we have discovered that the ESN algorithm does give satisfactory results, even when employed on networks that are significantly smaller than its usual range (tens to hundreds of units, for more complex learning tasks). Therefore, our idea was to simply re-train the network trained with the ESN algorithm with the Real-Time Recurrent Learning algorithm (RTRL), which is one of the standard gradient-descent algorithms for training RNNs.

The motivation for this entire endeavor is now clear: we seek a combination of good results and small networks, and that seems to be feasible with the aforementioned idea. In practical applications where even the exploitation of the network is costly – because of its complexity of $\mathcal{O}(n^2)$ (n is the number of units) – this combination of ESN and RTRL would provide better results than by training the network either only with the ESN approach or only with

the RTRL algorithm, while, at the same time, keeping down the size of the network.

Our assumption is that the ESN algorithm computes a set of synaptic weights that is already relatively close to a minimum on the error surface. Feeding those synaptic weights as the initial state of the RTRL algorithm and re-training the network with the same training dataset should, in principle, lead us to a much better solution than if we simply start our algorithm from a random weight matrix. Our goal was to verify these assumptions and to provide an experimental setup which would assert their validity. We wanted to investigate the gains in our results and the consistency with which we gained more by applying this combination of ESN and RTRL. Lastly, we compared the results of this combination with the results obtained by applying solely the ESN algorithm, but on larger networks.

Our relatively preliminary results suggest that all of our assumptions were correct and that they can be verified experimentally for the case of learning a Non-linear Auto-Regressive Moving Average (NARMA) system. We have managed to construct a scenario in which the combination of ESN and RTRL performs consistently better than either ESN or RTRL alone, for a relatively small network and number of epochs of RTRL. We have also shown that in order for the ESN algorithm to match the performance of the combination of ESN and RTRL, one would need to employ the ESN algorithm on a network of a significantly larger size.

The structure of this paper is the following: Section 2 presents the theoretical setup of our research. It shortly describes the model of the RNN that we are employing and the necessary nomenclature. We then briefly describe the functioning of the ESN algorithm and of Real-Time Recurrent Learning, as well as the details of interconnecting the two. Section 3 describes our experimental setup – the network and its preparation, the dataset and the implementation details of the combination of ESN and RTRL. Section 4 gives an overview of our results, Section 5 provides some concluding remarks and, finally, Section 6 states several ideas for future research on the topic.

2 Theoretical Setup

Recurrent Neural Networks (RNNs) are the foundations of our theoretical and experimental setup. These biologically inspired networks are of special interest because they possess capabilities not found in feed-forward networks – such as attractor dynamics or the ability to "store" information that can be used at a later stage. They can be employed for solving a very large class of problems involving time-varying input and output. However, in order to actually make use of such capabilities, one needs to add nonlinearity into the networks – this

makes it hard to study RNNs from a formal, mathematical point of view.

In general, RNNs can be represented in a graphical manner as shown in Figure 2. The units are the "neurons" of the network, whereas the connections between them are the equivalent of the "synaptic links", and the weights associated with these connections are equivalent to "synaptic strengths" (to keep the parallel with neuroscience).

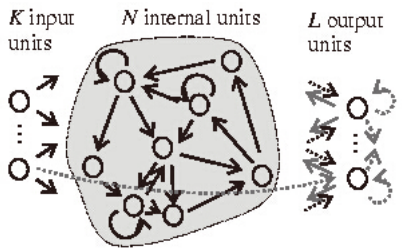


Figure 2.1: Basic Recurrent Neural Network structure (figure taken from [1])

There are many formal models describing RNNs: in this paper we only deal with a specific formal model of the RNNs – namely, discrete time models that lack spatial organization. Mathematically, RNNs can be best understood as (we are following closely the notation from [3]) a set of 3 vectors (n being the given time-step):

1. $\mathbf{u}(n) = (u_1(n), \ldots, u_K(n))^t$ – the input units,
2. $\mathbf{x}(n) = (x_1(n), \ldots, x_N(n))^t$ – the internal (hidden) units,
3. and $\mathbf{y}(n) = (y_1(n), \ldots, y_L(n))^t$ – the output units.

and 4 associated weight matrices:

1. $\mathbf{W}^{in} = (\omega_{ij}^{in})$ – of size $N \times K$ – which contains the weights associated with the connections from the input units to the internal units,
2. $\mathbf{W} = (\omega_{ij})$ – of size $N \times N$ – which contains the weights associated with the connections among the internal units,
3. $\mathbf{W}^{out} = (\omega_{ij}^{out})$ – of size $K \times N \times L$ – which contains the weights associated with the connections from the input units to the output units, from the internal units to the output units and from the output units to themselves,

4. and (optionally) $\mathbf{W}^{back} = (\omega_{ij}^{back})$ – of size $N \times L$ – the so-called back-projection matrix – which contains the weights associated with the connections from the output units to the internal units (hence the name).

The activation of the internal units is computed via

$$\mathbf{x}(n+1) = \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n)) \tag{2.1}$$

where f is the transfer function, usually a sigmoid – $\tanh(x)$ or $\frac{1}{1+e^{-x}}$ – or, sometimes, a linear function. The output can be thus calculated as follows:

$$\mathbf{y}(n+1) = \mathbf{f}^{out}(\mathbf{W}^{out}(\mathbf{u}(n+1), \mathbf{x}(n+1), \mathbf{y}(n))) \tag{2.2}$$

where $(\mathbf{u}(n+1), \mathbf{x}(n+1), \mathbf{y}(n))$ is the concatenation of the input, internal and output units and f^{out} is the output unit's output functions.

In this paper we will be dealing with *supervised* learning techniques – i.e. scenarios in which we have *teacher* data, which is the model that the RNN has to *fit*, and which is generated by some mathematical function or physical system. The main purpose of supervised learning is to train a RNN in such a way that, when its input resembles the training input data, it should output something that resembles the actual output of the original function or system.

2.1 Echo-State Network Algorithm

The Echo-State Network (ESN) algorithm, developed by Herbert Jaeger and introduced in [1, 3], is a novel technique for learning with RNNs that overcomes several of the drawbacks of the classical algorithms. It is a constructive algorithm for supervised training of RNNs, and the experiments so far have shown that it is a powerful and efficient method for learning complex dynamics (amongst others). This section provides a brief introduction to the technique and some of the theoretical insights related to it.

Given a network that possesses the so-called "echo-states" (the formal treatment of which is provided [1]), one can observe that the activation vector $\mathbf{x}(n)$ can be represented as a function of the past input to the network $(\mathbf{u}(n), \mathbf{u}(n-1), \ldots)$. On an intuitive level, this means that the activations of the internal units will simply be "echos" of the input. If the network is sufficiently inhomogeneous, these "echos" will differ substantially from each other and will thus offer a "reservoir" into which the output units can "tap".

Building up upon this idea, one can note that, in the most general case, the output of a Recurrent Neural Network will not only depend on the past input, but also on the past

output. This is the so-called "Nonlinear Auto-Regressive Moving Average" (NARMA) model of a dynamical system, and can be formalized via the following equation:

$$\mathbf{y}(n+1) = \mathbf{G}(\ldots, \mathbf{u}(n), \mathbf{u}(n+1); \ldots, \mathbf{y}(n-1), \mathbf{y}(n)) \tag{2.3}$$

where $\mathbf{G}$ is a (nonlinear) function that computes the next system output, given the previous inputs and outputs. The Echo-State Network approach is to generalize the idea presented above, and consider the output units as "input units", when viewed from the "perspective" of the internal units. Hence, a typical training procedure would imply enabling the back-projection weights and using the so-called "teacher-forcing" (writing the actual teacher output to the output units). This way, our "echo states" will not only incorporate the past inputs, but also the past (teacher) outputs.

"Tapping" from this (as Herbert Jaeger puts it) Dynamical Reservoir of "echo states" is done simply by combining the "echos" (more precisely, the function $\mathbf{G}$ which is associated to them) in a mean square way. This means that the ESN approach simply modifies the output weights and leaves the internal weights intact during the entire training process. As was mentioned before, this is nothing but "a task of computing the regression weights w_i^{out} for a regression of $d(n)$ on the network states $x_i(n)$"[1] (where $d(n)$ is the training data). Since this is the only major computation that has to be done by the ESN algorithm, its runtime is very small, and so it is considerably faster than other techniques.

The main steps of the algorithm can be thus summarized as follows:

- **Prepare the network**
  One has to build a RNN such that it has the echo-state property. If we consider a network, in which the activation function of the units is a sigmoid, one rule of thumb in doing so, according to [1], is to generate a weight matrix $\mathbf{W}$ such that its $|\lambda_{max}| < 1$ (but close to 1).

- **Run the network**
  As explained above, in order to benefit from the "echos" of the outputs, one has to apply "teacher forcing" during training. Given a sequence of time steps $n_0, \ldots, n_{max}$, we update the state of the network as follows:

  $$\mathbf{x}(n+1) = \mathbf{f}(\mathbf{W}^{in}(\mathbf{u}_{teach}(n+1)) + \mathbf{W}^{back}\mathbf{y}_{teach}(n) + \mathbf{W}\mathbf{x}(n))) \tag{2.4}$$

  where $(\mathbf{u}(n_0), \mathbf{y}(n_0), \ldots, (\mathbf{u}(n_{max}), \mathbf{y}(n_{max})$ is our training dataset. Because, in the beginning, the network output may exhibit traits of the initial random state of the network, we would also have to dismiss a long enough transient up to n_{min}.

- **Compute the output weights**
  Given $\mathbf{y}_{teach}(n) = (y_{teach,1}(n), \ldots, y_{teach,L}(n))$, $g'_j(n) = (f_j^{out})^{-1} y_{teach,j}(n)$, we have to compute for each $j = 1, \ldots, L$ the output weights ω_{ij}^{out} such that the value

$$MSE_{train,j} = \frac{1}{n_{max} - n_{min}} \sum_{n=n_{min}}^{n_{max}} \left(g'_j(n) - \sum_{i=1}^{N} w_{ji}^{out} x_i(n) \right)^2 \tag{2.5}$$

  is minimized. This, as we noted above, can be done via linear regression. With the computed output, the network can now be exploited.

2.2 Gradient-Descent Algorithms and RTRL

The other techniques for training RNNs rely on an frequently used method for finding minima – the method of gradient descent. The most commonly used technique is the so-called Back-Propagation Through Time (described in [4]) – it works similarly with the classical Back-Propagation for feed-forward networks but it has several drawbacks that make it less useful in practice: error-free implementation of the algorithm is hard and one has to do lots of experimentation ("tuning") in order to make it work. Another algorithm is the "Real-Time Recurrent Learning", developed independently by several scientists at the end of the 1980s. It is a mathematically very transparent technique, it was relatively easy to implement and, hence, was the one that we chose for our experiments. The most often cited paper on RTRL is the one by Williams and Zipser [5]. In presenting the algorithm, we will follow closely the notation from that paper.

For convenience, we redefine our network setup such that we now consider m input units and n internal units. The output units that we previously dealt with will simply be incorporated into the internal units, as, technically, there is no difference between the two (both can have connections from the input units, from the internal units and from/to themselves). Denoting by $\mathbf{y}(t)$ the values of the activation function at time t, and by $\mathbf{x}(t)$ the external input at time t, we concatenate the two vectors into $\mathbf{z}(t)$. Letting U be the set of internal units and I the set of input units, we can define the following value:

$$z_k(t) = \begin{cases} x_k(t), & if\ k \in U \\ y_k(t), & if\ k \in I \end{cases} \tag{2.6}$$

In the same fashion, we redefine our weight matrix $\mathbf{W}$ to contain all the weights between the units of the network. Now, we define

$$s_k(t) = \sum_{l \in U \sqcup I} w_{kl} z_l(t) \tag{2.7}$$

as the net input to the kth unit at time t. For $k \in U$, also define

$$y_k(t+1) = f_k(s_k(t)) \tag{2.8}$$

the output at time t, with f_k being the unit's activation function (a sigmoid, just as above). Given the set of output units T (which, as a reminder, is a subset of U), we can define the error function:

$$e_k(t) = d_k(t) - y_k(t) \tag{2.9}$$

(where $d_k(t)$ is the target value for the respective output) which is zero if $k \notin T$.

With these definitions in hand, we can proceed on to describing the actual algorithm. The objective of the algorithm is to minimize the total network error $J_{total}(t_0, t_1)$ over the trajectory from t_0 to t_1. This total network error is calculated as follows:

$$J_{total}(t_0, t_1) = \sum_{t=t_0}^{t_0+t_1} J(t) \tag{2.10}$$

with $J(t)$ being simply

$$J(t) = \frac{1}{2} \sum_{k \in U} [e_k(t)]^2 \tag{2.11}$$

This minimization is done by a gradient descent method, namely by changing the weight matrix $\mathbf{W}$ along $-\nabla \mathbf{W} J_{total}(t_0, t+1)$.

One way to do this is to accumulate the values of $\nabla \mathbf{W} J(t)$ for each time step from t_0 to $t+1$. As shown in [5], one can write the overall change in the weight matrix as

$$\Delta\omega_{ij} = \sum_{t=t0+1}^{t1} \Delta\omega_{ij}(t) \tag{2.12}$$

where one computes the change in the weights at time step t via the classical equation

$$\Delta\omega_{ij}(t) = -\alpha \frac{\partial J(t)}{\partial \omega_{ij}} \tag{2.13}$$

α being some learning rate (usually set to a small positive real number). To compute the partial derivative on the right side of equation 2.13, one simply calculates

$$-\frac{\partial J(t)}{\partial \omega_{ij}} = \sum_{k \in U} e_k(t) \frac{\partial y_k(t)}{\partial \omega_{ij}} \tag{2.14}$$

The right-hand side is computed by differentiating equations 2.7 and 2.8, and we obtain:

$$\frac{\partial y_k(t+1)}{\partial \omega_{ij}} = f'_k(s_k(t)) \left[\sum_{l \in U} \omega_{kl} \frac{\partial y_(t)}{\partial \omega_{ij}} + \delta_{ik} z_j(t) \right] \tag{2.15}$$

Setting the initial condition ($\frac{\partial y_k(t_0)}{\partial \omega_{ij}}$) to zero and replacing $\frac{\partial y_k(t)}{\partial \omega_{ij}}$ with $p_{ij}^k(t)$, we can now compute

$$\Delta\omega_{ij}(t) = \alpha \sum_{k \in U} e_k(t) p_{ij}^k(t) \tag{2.16}$$

and update the weights using this and equation 2.13.

In order to harness the full power of RTRL one has to run the above algorithm many times, until it reaches a local minimum on the error surface (one cannot normally hope for a gradient-descent algorithm to reach a global minimum on the error surface). One such iteration of the algorithm is called an "epoch" – to successfully train a RNN using RTRL one would normally consider having hundreds or even thousands of epochs.

One simplification of RTRL – which we used in our implementation – is that of updating the weights right after a training sample is presented (i.e. applying equation 2.16 directly, and not summing them up), which is a case of the so-called "online-learning". This has the drawback of not computing the actual exact negative gradient, but, in practice, according to Williams and Zipser [5], it does not make much of a difference as long as the learning rate is small enough. One practical advantage of this "direct updating" scheme is its potential for usage in applications where online learning provides more robust results – this is often the case in tasks related to filtering in wireless communications, for instance. The algorithm name itself – Real-Time Recurrent Learning – reflects the fact that one its main usages is online learning.

One very large drawback of the algorithm is its computational complexity – it is $\mathcal{O}(n^4)$ for each epoch – and so, normally, RTRL is of little practical interest, except in those cases when the network size is small enough. As noted in Section 3.2, the networks that we consider are suitable for running RTRL for several hundreds of epochs, with relatively low run-times (in the order of 10 minutes).

2.3 Interconnection of ESN and RTRL

Given the above descriptions of the ESN algorithm and of the RTRL algorithm, it is now easy to see how to interconnect the two so that one obtains the desired effect of re-training. Mathematically this is trivial and transparent: one simply needs to create a new, enlarged, weight matrix $\mathbf{W}^{all}$, that would encompass all the 4 matrices presented in Section 2 – $\mathbf{W}^{in}, \mathbf{W}^{out}, \mathbf{W}^{back}$ and $\mathbf{W}$. This new matrix has to contain the final weight matrices that were produced during the training phase of the ESN algorithm. With this matrix $\mathbf{W}^{all}$ in hand, it is easy to accommodate for the notation in Section 2.2 and to implement the algo-

rithm directly from the formulas. (A side note: the arrangement of the 4 matrices in $\mathbf{W}^{all}$ is not important as such, as long as the indices of ω_{ij}^{all} behave as expected – i.e. represent the synaptic weight of the connection from unit j to unit i.)

One has to take care when performing this last step – of creating $\mathbf{W}^{all}$ – because in our theoretical model we do not have connections between the output units and so we would need to set the respective region in $\mathbf{W}^{all}$ to zero; however, the RTRL algorithm might modify these values while trying to finding the steepest gradient along the error surface.

Obviously, we need to train both of the algorithms using the same network setup, dataset, activation function, etc. The experimental details on how this was done follow in the next section. That section also shows that, in practice, it is much harder to implement this interconnection – many details need to be taken care of and, unfortunately, these details are very problem-specific. Therefore, we are unable to provide a theoretical framework that would give a universal recipe for re-training with the RTRL algorithm networks that were already trained with the ESN algorithm. Things like the learning rate of RTRL or the network size – which would not overfit the given training dataset for both of the algorithms – can, in this case, only be set via trial-and-error.

Steps of the interconnection

Given the above, we can still describe a *simplified* model of our interconnection as follows:

1. Pick a network size that should be small enough for RTRL to work in a reasonable time and not overfit, but large enough for the ESN algorithm to produce satisfactory results.
2. "Tune" the ESN algorithm for our problem.
3. Run the ESN algorithm and train the network weights.
4. Assemble the weights $\mathbf{W}^{in}, \mathbf{W}^{out}, \mathbf{W}^{back}$ and $\mathbf{W}$ into a weight matrix $\mathbf{W}^{all}$.
5. Set to zero the weights among the output units.
6. "Tune" the RTRL algorithm for our problem.
7. Run the RTRL algorithm with the initial weight matrix begin equal to $\mathbf{W}^{all}$

3 Experimental Setup

In the following we will describe the experimental setup of our project. Section 3.1 contains a short description of the dataset that we used. Several details about selecting the network size and preparing it will be revealed in Section 3.2. Finally, in Section 3.3 we will discuss the specific details of interconnecting the two algorithms to be able to produce meaningful results.

3.1 The Dataset

As our test-bed, we have chosen a Nonlinear Auto-Regressive Moving Average (NARMA) system. It is a 2-input-2-output system, that can be described by:

$$\begin{aligned} d(1,n) &= u(2,n-5)\cdot u(2,n-10) + u(2,n-2)\cdot d(2,n-2) \\ d(2,n) &= u(2,n-1)\cdot u(2,n-3) + u(2,n-2)\cdot d(1,n-2) \end{aligned}$$

where $d(1,n)$ represents the first output at time-step n, $d(2,n)$ represents the second output at time-step n, $u(1,i) = 1$ is the bias input and $u(2,i)$ (sampled from a uniform distribution on the interval $(0,1)$) is the second input. Figure 3.1 shows the output of the system for n ranging from 1 to 500.

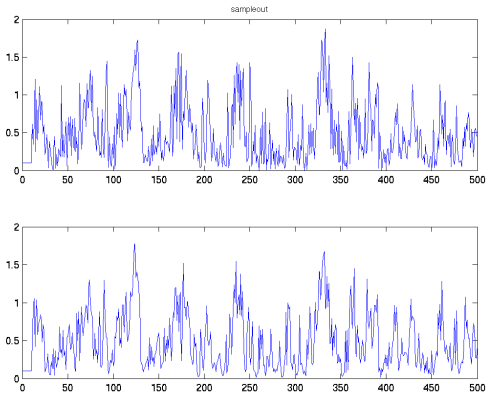


Figure 3.1: The output of the NARMA system

This system has all the necessary features that make it suitable for our experiments:

1. Modeling it is quite hard, given its non-linear nature.

2. Its output depends on both the past input and the past output, which can be exploited by the ESN algorithm.

3. This dependency is only of 10 steps away, which means we can still apply RTRL successfully (gradient-descent algorithms are renowned for being poor at learning long-term temporal dependencies, as shown in [6]).

We use 300 samples for training and 50 for testing. This was found to be – empirically – an optimal size of the dataset, given the computations and their complexity. For the purposes of learning with ESN, an additional 100 samples were used to "wash out" the "initial transient". This is done because of the partially arbitrary initial state, which reflects itself into the further dynamics of the system (a more thorough explanation can be found in [1]).

3.2 The Network

Initially, we have tried to find a network size for which the ESN algorithm performs well and does not overfit the data (the 300 samples from our NARMA system) at the same time. This has proven to be around 70 units – which is simply too large of a number if we want to apply RTRL. This is because it we would either run into time problems or would simply overfit the data. We have decreased the network size progressively until we reached a network that RTRL could handle and which would still be large enough to provide a Dynamical Reservoir for the ESN algorithm to work properly. Thus, after many experiments, the network size was chosen to be equal to 10.

The connectivity in the network is typically set to a value such that on average a reservoir unit is connected with 10 other units. In this case, it was set to 0.1 in order to satisfy this requirement. Afterwards, the weight matrix $\mathbf{W}$ is generated, such that it has the echo-state property. Empirically, this was shown in [1] to be equivalent to having a spectral radius $|\lambda_{max}| < 1$. The other matrices – $\mathbf{W}^{in}$ and $\mathbf{W}^{back}$ – are initialized with random values, since experience has shown that this has no effect on the echo-state property. In our case, the spectral radius of $\mathbf{W}$ was set to 0.8.

The activation function of the units is the usual sigmoid:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.1}$$

We have however obtained better results when using linear readout functions for the output units.

3.3 Interconnection Details

In the following, we will discuss several of the minor issues that arise during the interconnection phase of our "combination algorithm":

- A common way to improve the results obtained with the ESN algorithm is to shift and/or scale the input and/or output. For tips and tricks on how to do that, the reader is referred to Herbert Jaeger's tutorial on training RNNs with ESNs [1]. When using only the ESN algorithm for training our RNN to learn the given NARMA system, this technique indeed improves the results. However, such a scaled and shifted dataset destabilizes our RTRL algorithm. Unfortunately, we had to give up on these modifications of the training data simply for the sake of making both of our algorithms run properly.

- As a measurement of error for both the training and the testing phase, we chose the Normalized Root Mean-Square Error, defined as:

$$NRMSE = \sqrt{\frac{E[(y(n) - d(n))^2]}{\sigma^2(d)}} \tag{3.2}$$

  where $d(n)$ is the teacher output at time n, $y(n)$ is the network output at time n, and $\sigma^2(d)$ is the variance of the teacher output.

- The learning rate α of RTRL was set, after numerous trials, to 0.03. There is no theoretical motive behind this setting it to this value – it simply worked out to be a value which is not too large for RTRL to "diverge" from its path to the minimum, and not too small for RTRL to converge too slowly to the minimum of the error surface.

4 Experimental Results

Several experiments have been carried out in order to verify our assumptions related to the effectiveness of the combination of the ESN algorithm with the RTRL algorithm. We have first tried to substantiate the claim that this combination performs better than the ESN algorithm by itself, given the same dataset. Section 4.1 describes these experiments. Further, simply for the verification of our algorithm, we show that the combination of ESN and RTRL perform better than RTRL by itself, when started off with some random initial state (Section 4.2 details on this). Finally, in Section 4.3, we proceeded on to finding a network, which we train solely with the ESN algorithm, and whose size would be large enough to match the

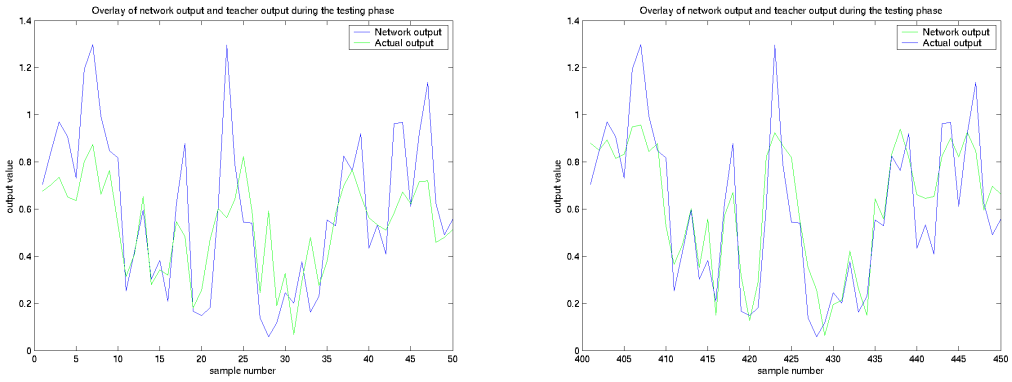
performance of the combination of ESN and RTRL (applied to our smaller network of size 10).

4.1 Combination of ESN and RTRL

The first set of experiments that we performed was concerned with the study of the performance of the ESN algorithm as compared to the combination of RTRL + ESN, given different initial starting weights for the ESN algorithm. Ten such experiments have been performed, during which the RTRL algorithm was run for 1000 iterations/epochs. These experiments have shown that the combination that we investigated throughout this paper performs better in every test run. On average, the *training* error for output nr. 1 decreases by 0.12 (18%), whereas for output nr. 2 by 0.09 (16%).

More importantly, the data from appendix 1.1 shows that the *test* NRMSE decreases dramatically – on average by 0.25 (28%) for output 1 and 0.27 (34%) for output 2 – a good sign that the model learnt generalizes better than the model learnt with the ESN algorithm.

Figure 4.1 provide a visual comparison of the two techniques – they depict one of the outputs of the network overlayed and the actual teacher output. It is easy to see that "post-processing" has had its effects – the picture on the right shows the output modeling the actual data much more closely. This only confirms the numbers presented above.



(a) Network trained only with ESN

(b) Network trained with ESN and RTRL

Figure 4.1: Overlay of the network output and the testing data

The next step was to investigate the behaviour of our system when the other random com-

ponent in our experiments – the dataset was changed (as a reminder, we note that the input to the NARMA system is a random number on the interval from 0 to 1). We have, naturally, repeated the same experiments as above, this time, though, at each run, we changed the data set that was used.

In the light of the previous findings, it was not suprising that the results were simply confirming the ones just presented. On average, the *training* error for output nr. 1 decreases by 0.03 (4%), whereas for output nr. 2 by 0.10 (17%). The *test* NRMSE decreases dramatically, too – on average by 0.42 (37%) for output 1 and 0.53 (50%) for output 2. Appendix 1.2 contains the raw figures for this series of experiments.

During one of the experiments, the training error did not improve, but stayed at roughly the same level. This suggests that the learning rate can still be fine-tuned – our assumption is that the RTRL algorithm has, in this case, simply landed in a part of the error surface from where (given the α considered) it couldn't progress any further.

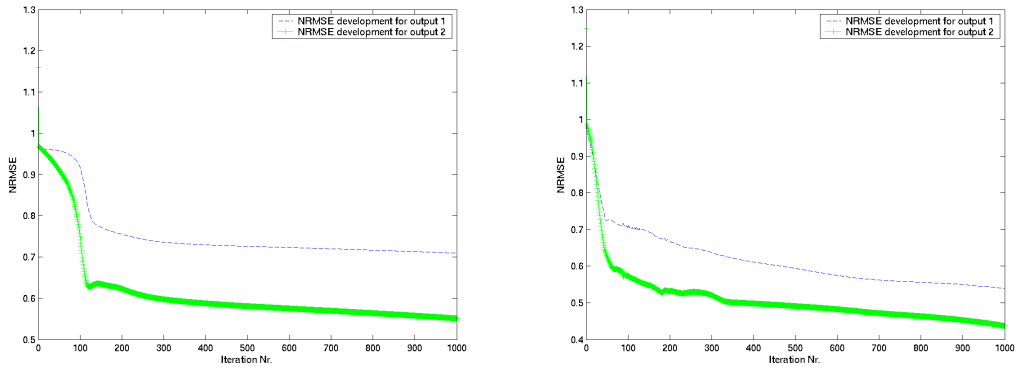
On the whole, we can safely conclude (as safely as one can make any conclusions from such a small number of test runs) that our assumption holds in this case: the combination ESN + RTRL does improve upon using only ESN, and this is especially true when comparing the results for the testing cases.

4.2 Verification Runs

In order to substantiate even more our claims about the effectiveness of combining ESN with RTRL, we have run several tests that would compared this combination with the RTRL aglorithm that is initialized with a random weight matrix. Naturally, this latter scenario turned out to be less effective in learning our NARMA system. Figure 4.2 exemplifies this on a test-run of 1000 epochs, using the same dataset as before. If we compare the two pictures, it is easy to see that there is no time-step at which the RTRL algorithm that started off with random weights is better than the "optimized" RTRL. This, again, shows the advantage of "pre-processing" the weights by running the ESN algorithm.

4.3 Further Issues

In the end, we set ourselves to find a network whose size would be enough for the ESN algorithm to perform just as well as the combination of ESN and RTRL that we have been studying above. This search was done by an incremental trial-and-error – we found the network size of 30 to be the one that qualifies. Its average performance during the training part



(a) Random initial weights

(b) Initial weights taken from the run of ESN

Figure 4.2: NRMSE for both of the outputs, given two different scenarios in which RTRL is employed (note the difference in the scale of the y-axes)

is $(0.481, 0.483)$ and, for the testing part, $(0.592, 0.439)$, for each of the outputs, respectively. The reader should note that this is a significant increase – our previous scenario considered networks of size 10!

5 Conclusions and Discussion

We considered the problem of training Recurrent Neural Networks (RNNs) with the help of the Echo-State Network (ESN) approach. We then developed a technique for improving the results obtained with the ESN algorithm by applying a "post-processing" technique – a gradient-descent based learning algorithm for training RNNs. The networks that we considered are of relatively small size, which means that the ESN algorithm was not be able to perform according to its full potential. However, our findings suggest that the results thus obtained can be further improved by applying Real-Time Recurrent Learning. This paper shows that, with the help of such a setup (i.e. the combination of ESN and RTRL), we can train small RNNs that perform as well as the networks of significantly larger size, which were trained only with the ESN algorithm. This latter finding is especially relevant for practical applications where network size is critical and is one of our motivations for having tackled this research question in the first place.

However, our findings are of limited statistical value – unfortunately, the run-time of the

RTRL algorithm imposes a certain limit on the number of experiments that can be performed in a reasonable amount of time. A more comprehensive study of the phenomena described above would need to consider, at the very least, more network sizes and more test runs. In addition to that, our experimental setup is very limited – we cannot generalize our results to other datasets or network setups. Therefore, the reader should take our findings only as an *indication* that the combination of ESN and RTRL performs better than ESN by itself – and not as conclusive results.

6 Future Research

As mentioned in the previous section, future research should concentrate on investigating the statistical validity of our results. Of special interest would be the testing of the combined algorithm on a benchmark dataset – such as the Mackey-Glass system – or in a real-life scenario (such as adaptive filtering in cellular communication).

As an alternative to RTRL, more efficient algorithms, such as Back-Propagation through time or the Extended Kalman Filter [7] should be considered. EKF is especially interesting in this context, given that it is a second-order gradient-descent algorithm that could exploit the special curvature of the error surface in the vicinity of the minimum. Ultimately, a real-life application – such as an embedded controller that operates a trained RNN – is the goal to be pursued in confirming and applying the results obtained in this paper.

1 Appendix

1.1 Results of First Set of Experiments

Here are NRMSE values for the set of experiments in which, with each experiment, a new *network* was generated. The second columns of the tables represent the values of the first output, the third columns – for the second output.

1.2 Results of Second Set of Experiments

Here are NRMSE values for the set of experiments in which, with each experiment, a new *dataset* was generated. The second columns of the tables represent the values of the first output, the third columns – for the second output.

0.709440 0.652051 0.948723 1.128996 ESN + RTRL: 0.705556 0.515002 0.684429 0.563980

Nr.	*ESN*	*ESN + RTRL*	*ESN*	*ESN + RTRL*
1	0.701	0.465	0.506	0.410
2	0.642	0.634	0.505	0.463
3	0.654	0.446	0.524	0.361
4	0.661	0.548	0.535	0.407
5	0.682	0.568	0.539	0.437
6	0.623	0.476	0.529	0.422
7	0.648	0.574	0.493	0.440
8	0.685	0.620	0.499	0.468
9	0.626	0.618	0.571	0.484
10	0.707	0.478	0.543	0.429
Avg	*0.663*	*0.543*	*0.524*	*0.432*

Table 1.1: Comparison of the *train* NMRSE

Nr.	*ESN*	*ESN + RTRL*	*ESN*	*ESN + RTRL*
1	0.856	0.603	0.745	0.483
2	0.871	0.733	0.767	0.572
3	0.919	0.497	0.817	0.433
4	0.883	0.548	0.891	0.550
5	0.904	0.697	0.830	0.528
6	0.934	0.580	0.846	0.499
7	0.823	0.782	0.714	0.578
8	0.877	0.677	0.739	0.536
9	0.965	0.736	0.857	0.581
10	0.875	0.536	0.795	0.502
Avg	*0.891*	*0.639*	*0.800*	*0.526*

Table 1.2: Comparison of the *test* NMRSE

Nr.	*ESN*	*ESN + RTRL*	*ESN*	*ESN + RTRL*
1	0.646	0.642	0.643	0.576
2	0.553	0.536	0.541	0.535
3	0.675	0.662	0.683	0.510
4	0.631	0.567	0.608	0.388
5	0.658	0.644	0.669	0.668
6	0.666	0.570	0.688	0.455
7	0.701	0.596	0.655	0.460
8	0.541	0.604	0.569	0.596
9	0.626	0.618	0.571	0.484
10	0.709	0.705	0.652	0.515
Avg	*0.640*	*0.614*	*0.627*	*0.518*

Table 1.3: Comparison of the *train* NMRSE

Nr.	*ESN*	*ESN + RTRL*	*ESN*	*ESN + RTRL*
1	1.106	0.871	1.044	0.630
2	1.025	0.899	1.021	0.709
3	1.110	0.664	1.314	0.432
4	1.567	0.657	1.802	0.392
5	1.179	0.722	1.213	0.631
6	1.285	0.589	1.512	0.429
7	1.189	0.596	1.436	0.496
8	0.949	0.704	0.997	0.582
9	0.965	0.736	0.857	0.581
10	0.948	0.684	1.128	0.563
Avg	*1.132*	*0.712*	*1.083*	*0.545*

Table 1.4: Comparison of the *test* NMRSE

Bibliography

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

[2] Herbert Jaeger. Short term memory in echo state networks. Technical Report 152, German National Research Center for Information Technology, 2001.

[3] Herbert Jaeger. Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the "echo state network" approach. Technical Report 159, German National Research Center for Information Technology, 2002.

[4] Paul J. Werbos. Backpropagation through time; what it does and how to do it. In *Proceedings of the IEEE*, volume 78, pages 1550–1560, 1990.

[5] Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1:270–280, 1989.

[6] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient-descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, March 1994.

[7] Gintaras V. Puskorius and Lee A. Feldkamp. Neurocontrol of nonlinear dynamical systems with kalman filter trained recurrent networks. *IEEE Transactions on Neural Networks*, 5(2):279–297, March 1994.