Recursive Least Squares for Echo State Network Damage Compensation

Daniel Dean d.p.dean@pgr.reading.ac.uk
Prof. Slawomir Nasuto s.j.nasuto@reading.ac.uk
Prof. Kevin Warwick k.warwick@reading.ac.uk
School of Systems Engineering, University Of Reading, Reading, UK

Abstract. In this paper an Echo State Network (ESN) with a Recursive-Least-Square (RLS) output layer is developed to demonstrate RLSs potential for compensating for damage to the ESNs pool. The pool of an ESN is the Recurrent Neural Network (RNN) from which it draws its dynamics, and is normally static after the training of the ESN's output layer. RLS is used to compensate for pool damage, so that individual sub-pools of the pool can be distributed across a multi-agent system (e.g robots in a Unmanned Aerial Vehicle (UAV) swarm), which can dynamically join or drop connections. This is useful in multi-agent systems as it allows a sharing of computational resources across a swarm. The main focus of this investigation was the forgetting parameter of the RLS algorithm. We conclude here that the classical best range of this parameter is not ideal for this system, which requires a much lower value. The computation time required for RLS is also briefly investigated, to demonstrate an important consideration in the use of RLS in this situation: that of RLS being relatively slow.

1 INTRODUCTION

In the field of multi-agent systems the problem of combining computational resources in a system of distributed processors (e.g. a Unmanned Aerial Vehicle (UAV) swarm) is a complicated one. Addressing this problem, however, is important for the most efficient use of such distributed systems. A method of combining and sharing computational resources over many processing units is to distribute an Artificial Neural Network (ANN) across the processing units. This allows a larger (in number of weights and neurons) neural network than could be present on a single processing unit, which typically allows more complicated problems to be solved, or problems to be solved to a higher accuracy. Another problem to address is that the world is inherently history-dependant, that is, in majority of real-world interactions the history of the interaction is just as important as the instantaneous information available.

In this paper a novel Distributed Neural Network (DNN) is presented using a form of Recurrent Neural Networks (RNNs) known as an Echo State Network (ESN). An ESN is a technique which uses a normally static pool consisting of an RNN and is useful compared to RNNs because of its far simpler training. The use of a RNN allows for the input history to be natively processed (rather than with, say, a feed-forward network which requires extra structures in the form of delayed feedback [1]). An ESN is used as it is a computationally cheap [2] form of an RNN. An algorithm, specifically the Recursive-Least-Square (RLS) algorithm, is used to allow an ESN to gracefully deal with a changing network structure so as to compensate for network damage, for example in a UAV swarm when one agent (a sub-pool) cannot communicate. The major advantages of this form of multi-agent resource sharing are the ability to natively handle time series inputs and being able to compute an arbitrarily high number of outputs from a single network, concurrently [3].

In previous literature hierarchies of ESN pools (which would naturally lend themselves to distribution) have been described [4]. These hierarchies would perform poorly in swarms of processing units due to their inability to deal with changing topologies. Furthermore these hierarchies acted as no more than large networks of RNNs, meaning that the entire hierarchy is analogous to a single ESN (with a specific inter-pool network structure). The results this paper presents (which are for single pools) can therefore be easily applied to a dynamic hierarchy of ESNs due to this analogy.

A distributed ESN would function as a platform for multi-agent systems to share resources, specifically to pool computational power for 'difficult' problems, e.g.. ones requiring either large input dimensionality, history-dependant input, unknown (learned) input-output transform etc. Since a hierarchy of ESNs can be treated as a single pool, this means that the work presented here can be used in such a way for multi-agent systems. For an example imagine group of communicating, co-operating agents, each processing a single ESN, one finds a 'difficult' problem, with this proposed system it would simply combine its ESN with nearby agents to increase accuracy vs. a non-neural network approach.

The novel contribution this paper makes is an empirical investigation into the use of the RLS algorithm in compensating for damage to an ESN pool, with the analogy that a single pool can be treated as a hierarchical ESN, and damage represents network members (sub-pools) no longer processing.

This paper first gives a brief introduction to the two major concepts at the heart of this research, ESNs and the RLS algorithm in Section II. This is followed by an overview of the experiments undertaken in Section III, with the results presented in Section IV. Finally the conclusions of this paper are presented in Section V.

2 BACKGROUND

2.1. ESN

An Echo State Network (ESN) is a form of a RNN (i.e. a neural network with one or more recurrent connections) which uses a fixed RNN as a pool, which is perturbed by an input layer, and is sampled by an output layer, see Figure 1. ESNs, and several other similar systems, are collected under the umbrella of Reservoir Computing (RC) [5]. Typically all inter layer and intra layer weights are real. The input layer, which is one neuron per dimension of the input data, is typically used to scale incoming data, as the range of inputs

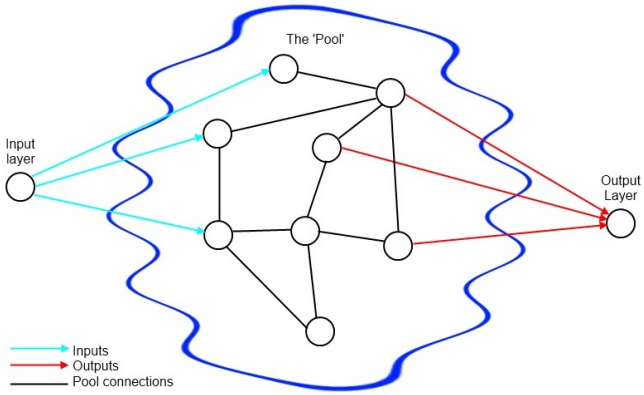

Figure 1: **An Echo State Network**

can have a significant effect on the performance. The pool consists of a set of neurons arranged in a weighted RNN. The output layer consists of a set of from-pool weights, which are the only section of the system which is trained [3]. Indeed, one of the major reasons for using an ESN is that one can have the time series handling of RNNs, with the simplicity of linear regression training, as the output becomes the combination of linear regression between all pool units and a target output.

A Typical ESN is defined by three matrices and one function, as well as the number of neurons in each layer: n_{in}, n & n_{out} in the input layer, pool and output layer respectively. The matrices are as follows: the input weights matrix, $\mathbf{W}_{in} \in \mathbb{R}^{n \times n_{in}}$, the pool weights matrix, $\mathbf{W} \in \mathbb{R}^{n \times n}$ and the output weights matrix, $\mathbf{W}_{out} \in \mathbb{R}^{n \times n_{out}}$. The pool neuron activation function, $\mathbf{f}(x)$ is typically $tanh()$, though many others have been used in ESNs [4], [6]. The state update equations governing a typical discrete-time (k) ESN are as follows:

$$\mathbf{x}(k+1) = \mathbf{f}(\mathbf{W}_{in}\mathbf{u}(k) + \mathbf{W}\mathbf{x}(k)) \quad (1)$$

$$\mathbf{y}(k) = \mathbf{f}_{out}(\mathbf{W}_{out}\mathbf{x}(k+1)) \quad (2)$$

where $\mathbf{u}(k)$ is the matrix of current inputs to the ESN, $\mathbf{y}(k)$ is the matrix of current outputs from an ESN and $\mathbf{x}(k)$ is the vector of current neuronal activations in an ESN pool. For single-dimensional inputs and outputs, called a Simple Echo State Network (SESN) [7] the input and output matrices are simple vectors. $\mathbf{f}_{out}$ is the output function, in this paper this is linear (i.e. $\mathbf{f}_{out}(x) = x$).

Several variables control the function of an ESN, which include, but are not limited to, the spectral radius of the $\mathbf{W}$ matrix, the magnitude of the input scaling, the specific pool neuron activation function used and the pools internal connectivity [8]. Many other papers explore and explain these factors (e.g. [3], [4], [6], [8]), and only typical values will be used here.

This paper concentrates on the properties of ESNs in cases of damaged pools, i.e. pools that are altered after training. Very little literature exists documenting the performance of damaged pools, so we will be demonstrating the inability of a standard ESN to cope with pool damage. ESNs suffer from an incomplete formalisation of how input information is cast into the high dimensional pool space, [3], [4] and therefore how to optimally design a pool for a given problem. This work can be seen as a stepping stone of research in how to effectively allocate pool resources to problems.

2.2. RLS

Recursive-Least-Square (RLS) is an algorithm building on the simpler (and older) Least-mean-Square (LMS) algorithm, incorporating the error history of a system into the calculation of the present error compensation. This algorithm is used, in this case, to adjust the $\mathbf{W}_{out}$ matrix during the run-time of an ESN, i.e. as an on-line learning algorithm. The primary topic of investigation of this research (in addition to question of basic viability) was that of the algorithms forgetting factor, λ. The forgetting factor determines how exponentially less important the error history is, with a value of 1 valuing all the system history (as well as the present state) exactly the same, and lower values causing an exponential reduction in the importance of the systems history. Literature states the forgetting factor in most circumstances should be very close to, but less than 1 [8], [9]. In RLS the cost function to minimise is as follows [10]:

$$E(k) = \sum_{i=1}^{k} \lambda^{k-i} e(k)^2 \quad (3)$$

with e being the error function $y_{target} - y$. After minimisation the four equations governing RLS for ESNs are as follows:

$$\mathbf{w}_{out}(k+1) = \mathbf{w}_{out}(k) + e(k)\mathbf{g}(k) \quad (4)$$

$$e(k) = y_{target}(k) - y(k) \quad (5)$$

$$\mathbf{g}(k) = \frac{\mathbf{P}(k-1)\mathbf{x}(k)}{\lambda + \mathbf{x}(k)^T\mathbf{P}(k-1)\mathbf{x}(k)} \quad (6)$$

$$\mathbf{P}(k) = \lambda^{-1}\mathbf{P}(k-1) - \mathbf{g}(k)\mathbf{x}^T(k)\lambda^{-1}\mathbf{P}(k-1) \quad (7)$$

For a single output system, $\mathbf{w}_{out}$ would be a column vector, as would $\mathbf{x}(k)$. As can be seen in this form only the P matrix is recursive, containing the history of the system. RLS is notoriously unstable [8], and the choice of forgetting factor is important in determining whether a given system will become unstable, with higher factors typically being more numerically stable (i.e. staying within expected magnitudes). Smaller λ values (causing numerically unstable P-matrices) are especially problematic on computers, where the double precision floating point standard is used, and its upper range of $\times 10^{308}$ [11] can quickly overflow, being treated as an infinity (and causing a breakdown in calculation accuracy).

3 EXPERIMENTAL METHOD

The major focus of this paper is upon determining if the RLS algorithm is suitable for use in making ESN pools dynamic. This means the main focus of experimentation is on determining if RLS can compensate for pool damage (i.e. by returning the pools performance to near pre-damage levels) and whether there are limits to this compensation. Extra experiments will then investigate what

Number of neurons	200	Spectral Radius	0.95
Internal pool connectivity	5%	Number of input neurons	1
Pool neuron activation function	$tanh()$	Number of output neurons	1
Input Scaling	0.1	Ridge Regression parameter	5×10^{-6}

Table 1: **The parameters of the ESN used in this paper's experiments**

effect the forgetting factor has upon RLS performance in this case. An ESN with the parameters described in Table 1 was used for every experiment detailed here.

Two different data sets were used to address these questions. The first dataset was a simple function approximation: $f(x) = \frac{40x^2}{3+x^4} - 3, \quad -10 \leq x \leq 10$, which is sufficiently regular to allow for easy identification of the effects of pool alterations. The second dataset, which describes a time series prediction task, was the Normalised Auto-Regressive Moving Average (NARMA) ($\tau = 30$) time series prediction dataset. This dataset is a widely used benchmark of time-series performance in the field of RC (see, e.g. [3], [6], [12]).

Firstly, a control test is performed to demonstrate the ability of a standard ESN to cope with pool alterations. The pool damage is realised in the form of adding or removing pool neurons without re-training the output layer. These control experiments involved training an ESN to solve a given problem, then modifying the pool and comparing the modified pool's performance against its original performance. A second control experiment, demonstrating the ability of a 100 and 300 neuron pool to compute the given problems is also constructed, as the pool modification occurs in sets of 100 neurons (representing a contiguous group of neurons dropping out of the network). This was performed to show that any change in performance was not due to the overall pool size, but due to the effects of pool damage.

To demonstrate the ability of the RLS algorithm to compensate for pool damage an ESN with an RLS output layer was developed. The output layer samples the pool as with a normal ESN, however after every sampling the output weights are updated according to the RLS algorithm [13].

First the output layer was trained on a given dataset, using the ridge regression method to create an initial set of output weights. Then a subset of the testing set was processed by the ESN (800 data points out of 2000), after which the pool damage, i.e. adding or removing neurons, occurred. After this damage, the remainder of the testing set was processed. Recorded was the system output plotted with the expected output, the instantaneous root-square-error between the actual output and the expected output and the Root-Mean-Square-Error (RMSE) of the entire output (excluding the initial transients, see [3]). The experiments were averaged over 50 pool setup-train-process data runs. These experiments were tested for different values of the forgetting parameter, specifically $0.6 \leq \lambda \leq 1$, in steps of 0.02.

An extra experiment was performed, showing the difference in execution time, measured in CPU time, i.e. the accumulation of CPU time the process has used, between the RLS algorithm, and the far simpler LMS algorithm on which it is based. Python's *timeit* module is used to profile the execution time as it avoids the pitfall of simply using end time - start time as the processing time, and provides a far more accurate (though not perfect) indication of the time taken to compute [14]. Both types of ESN (i.e. RLS-ESN and LMS-ESN) were set-up, trained and presented the testing dataset (the 'simple' function) 100 times, and the total and average of their execution time was recorded. Since both set-ups are as similar as possible the cause of time differences is the amount of processing time required for each on-line training algorithm.

It can be seen that the experiments performed mainly involve testing various values of the forgetting factor in a damaged ESN and comparing the outcome of these runs against non-RLS controls in the case of pool damage and no pool damage.

4 EXPERIMENTAL RESULTS

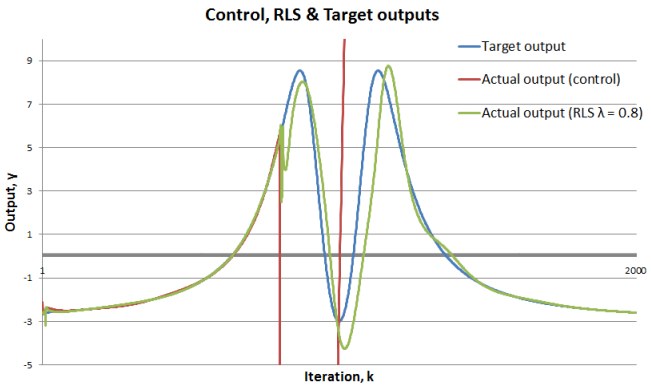

Figure 2: **Output, y for iteration k for control and example RLS systems.**

The control experiments showed that a standard ESNs performance rapidly diverges from expected when the pool is altered after initial training. See Figure 2 for the control's output against the target output and RLS output, notice the slight noise near the start ($k < 100$) is the initial transient. This RLS output is an extrema, but does clearly show many of the features observed in this research. The control experiments also showed little difference between the performance of a 100, 200 & 300 neuron pool for these problems.

In the forgetting parameter experiments (see Figure 3) we can see best performance was derived for values of the forgetting parameter far below the classical range for this parameter. Also seen is 'nan'

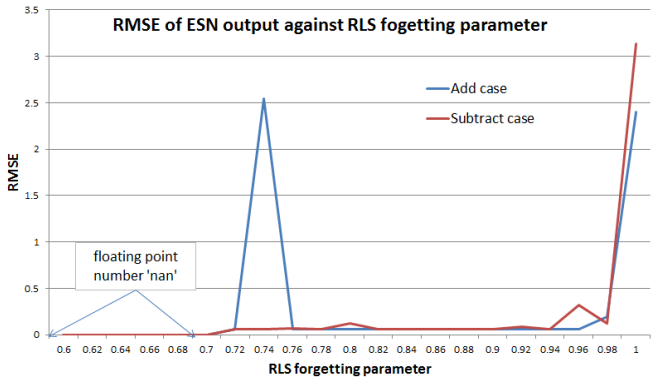

Figure 3: **Output, y for forgetting parameter λ in the RLS-ESN, for 'simple' function dataset.**

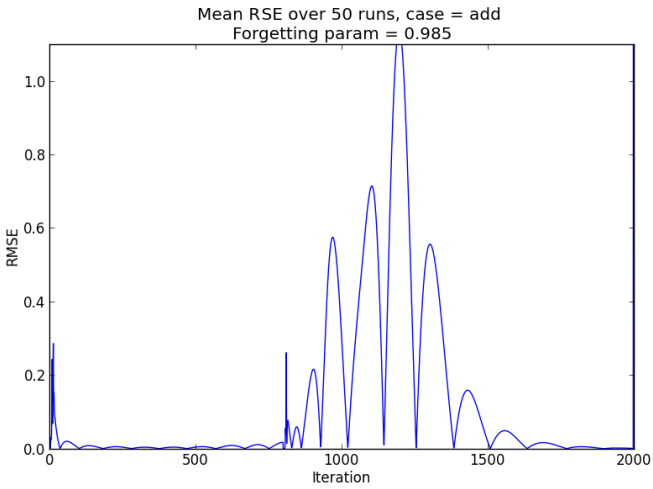

Figure 4: **The instantaneous RSE for an example set of runs**

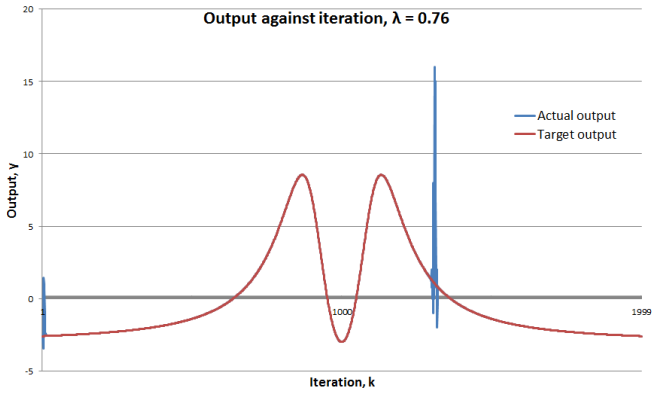

Figure 5: **Output against iteration, displaying the noise that occasionally appears.**

at lower values, representing the floating point number flag 'Not A Number' which occurs here because the RLS P-matrix overflows, being represented by infinity, which causes (in the gain, **g**, equation) an infinity to be divided by an infinity, which is represented in floating point numbers as 'nan'. Figure 4 shows the instantaneous Root-Square-Error (RSE) per output update, which shows the spike in error at around k=800, and noise that follows. Notice the output noise later in Figure 5 around iteration 1400, also note there is no 'spike' at the point of damage ($k = 800$) as the value of λ used is extremely good at damage compensation. This noise is caused by the P-matrix getting exceptionally large (e.g. on the order of $\times 10^{100}$) and double precision floating point numbers having an accuracy of 15 decimal places (i.e. accurate to $\times 10^{15}$ [11]), so that the output calculations become increasingly incorrect and noisy. This noise does not appear in all runs, and it seems that the RLS algorithm compensates even for this noise (as the noise swiftly disappears), but it does occasionally occur and degrade performance.

In general then, the lower the forgetting parameter the better the damage compensation (until the point of floating point overflow) but the higher the forgetting parameter the better the stability. The best range for maximising damage compensation and reducing noise in the 'simple' function case was found to be 0.84 to 0.9, for removing neurons, while the range 0.76 to 0.96 worked best when adding neurons. For the NARMA30 dataset there was far more noise when adding neurons, creating excessively large RMSE values, seemingly at random, and the subtractive case simply worked better (both in accuracy and noise minimisation) with lower forgetting parameters, up to the point of system breakdown (at $\lambda \approx 0.7$). To be noted, however, is that the point of RLS breakdown for this system is function magnitude specific, that is, the growth of the P matrix is dependent on λ and $\mathbf{x}$ so functions of a larger magnitude are likely to degrade quicker.

Seen in Table 2 are the results of the timing experiment, showing the RLS algorithm taking significantly longer than the simpler (though less accurate) LMS method. This would seem to suggest that a less accurate, but faster algorithm would be useful for solving this problem.

5 CONCLUSION

This work demonstrates the ability of RLS to return a damaged ESN pool to its pre-damage accuracy. The work presented here, while seemingly solving artificial problems (a graph-friendly 'simple' problem and the NARMA-30 dataset), exists as part of a

Algorithm	Time for 100 runs (s)	time per run (s)
Least Mean Squares	734.814	7.348
Recursive Least Squares	280.778	2.808

Table 2: **CPU over 100 runs for RLS-ESN and LMS-ESN**

larger project, with this larger project being intended to provide a multi-agent swarm a method to pool computational resources for solving difficult problems (by dynamically joining ESN pools). This work therefore is providing information on the selection of the most low level form of co-operation in this larger project, and investigated the viability of the RLS algorithm to work in this capacity. The problems solved are function approximation problems in the instantaneous and history-dependant cases, which are common problems in real-world interactions.

In conclusion, while RLS has demonstrated the ability to compensate for alterations to an ESNs pool structure in an on-line manner, sometimes extremely well, its numerical instability reduces its effectiveness enough to exclude it as a viable method for compensating for pool alterations. Furthermore, the point at which the instabilities simply become floating point infinities is likely dataset-specific, and is certainly platform specific (e.g. single-precision will destabilise far earlier, quad-precision later, etc.). Also of consideration is the computational complexity, and thus time to compute, of RLS, which (in this basic implementation) is very high compared to other on-line algorithms, e.g. LMS.

Future work therefore will concentrate on using methods with RMSE accuracy close to that of RLS but without the numeric instability and with much less computational complexity so that this method can be used in simple multi-agent swarms. Likely optimisers for this future work include population-based algorithms such as Particle Swarm Optimisation (PSO) and Stochastic diffusion search (SDS). Another objective is to ascertain whether other forms of problem (prediction, classification etc.) can also be solved in this way, or if this method is more suited to one class of problems over others.

6 REFERENCES

[1] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. J. Lang, "Phoneme recognition using time-delay neural networks," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 37, no. 3, pp. 328–339, 1989.

[2] M. Lukoševičius, D. Popovici, H. Jaeger, U. Siewert, and R. Park, "Time warping invariant echo state networks," Citeseer, Tech. Rep., 2006.

[3] H. Jaeger, "The "echo state" approach to analysing and training recurrent neural networks-with an erratum note'," *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, vol. 148, 2001.

[4] M. Lukoševičius, "Reservoir computing and self-organized neural hierarchies," Ph.D. dissertation, PhD thesis, Jacobs University Bremen, Bremen, Germany, 2011.

[5] "Reservoir computing," http://reservoir-computing.org/, Accessed 2013-04-27. [Online]. Available: http://reservoir-computing.org/

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

[7] G. Fette and J. Eggert, "Short term memory and pattern matching with simple echo state networks," in *Artificial Neural Networks: Biological Inspirations–ICANN 2005*. Springer, 2005, pp. 13–18.

[8] M. Lukoševičius, "A practical guide to applying echo state networks," in *Neural Networks: Tricks of the Trade*. Springer, 2012, pp. 659–686.

[9] M. H. Hayes, *Statistical Digital Signal Processing and Modeling*. John Wiley & Sons, 1996.

[10] S. Haykin, *Adaptive filter theory*. Prentice Hall, 1991.

[11] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[12] D. Verstraeten, B. Schrauwen, M. d'Haene, and D. Stroobandt, "An experimental unification of reservoir computing methods," *Neural Networks*, vol. 20, no. 3, pp. 391–403, 2007.

[13] H. Jaeger, "Adaptive nonlinear system identification with echo state networks," in *Advances in neural information processing systems*, 2002, pp. 593–600.

[14] "timeit - measure execution time of small code snippets." [Online]. Available: http://docs.python.org/2/library/timeit.html