FPGA Implementation of Reservoir Computing with Online Learning

Piotr Antonik
Laboratoire d'Information Quantique
Université Libre de Bruxelles
Avenue F. D. Roosevelt 50, CP 225
B-1050 Bruxelles, Belgium
pantonik@ulb.ac.be

Anteo Smerieri
Service OPERA-Photonique
Université Libre de Bruxelles
Avenue F. D. Roosevelt 50, CP 194/5
B-1050 Bruxelles, Belgium

François Duport
Service OPERA-Photonique
Université Libre de Bruxelles
Avenue F. D. Roosevelt 50, CP 194/5
B-1050 Bruxelles, Belgium

Marc Haelterman
Service OPERA-Photonique
Université Libre de Bruxelles
Avenue F. D. Roosevelt 50, CP 194/5
B-1050 Bruxelles, Belgium

Serge Massar
Laboratoire d'Information Quantique
Université Libre de Bruxelles
Avenue F. D. Roosevelt 50, CP 225
B-1050 Bruxelles, Belgium

Abstract

Reservoir Computing is a bio-inspired computing paradigm for processing time dependent signals. The performance of its analogue implementation are comparable to, and sometimes surpass, other state of the art algorithms for tasks such as speech recognition or chaotic time series prediction. However, these implementation present several issues, which we address here by using programmable dedicated electronics in place of a personal computer. We demonstrate a standalone reservoir computer programmed onto a FPGA board and apply it to the real-world task of equalisation of a nonlinear communication channel. The training of the RC is carried out online, as this learning method, on top of being simple to implement, allows the RC to adapt itself to a changing environment, as we show here by equalising a variable communication channel.

1. Introduction

Despite the breathtaking advances in traditional computing technologies and algorithms, scientists keep on developing alternative bio-inspired computational methods. While the former focus on fast and efficient algorithms exhibiting centralised control, the latter place more emphasis on robustness and adaptability, based on interaction of many coupled units. These alternative computational methods often work well even on a poorly defined task and are able to adapt to unforeseen variations of the environment, see e.g. [5].

Reservoir Computing (RC) is a set of methods for designing and training artificial neural networks [12, 20]. The RC inherits core features from recurrent neural networks, while drastically simplifying them by fixing many of the parameters. Specifically, the coupling of the input to the network and the interconnections between the nodes are fixed (typically at random), while only a linear readout layer is trained to achieve the best performance. This greatly simplifies the training process, rendering the RC approach intrinsically faster than alternative recurrent neural networks methods, as training of the output layer only (generally a simple matrix inversion) is much simpler than the common error backpropagation algorithm.

The RC algorithm equals, and sometimes evens beats, other algorithms [18, 9, 19, 17]. In 2007, it won an international competition on prediction of future evolution of financial time series [2]. It has been applied to speech and phoneme recognition, equalling or outperforming other approaches [30, 14, 28]. Reservoir computing is also remarkably well suited to analog implementations, as recently

demonstrated in [3, 15, 22, 8, 6] where experimental implementations were shown to exhibit performance comparable to state-of-the-art digital implementations for a series of benchmark tasks. We note two points that have to be addressed for further development of software and hardware implementations of reservoir computing.

Firstly, real-time applications require large amounts of data (training sets exceeding thousands of inputs) to be processed in a fraction of a second. Such speeds can be achieved by the use of programmable dedicated electronics, such as Field Programmable Gate Arrays (FPGAs) or Application Specific Integrated Circuits (ASICs). A reservoir computer implementation using spiking neural networks on a simulated FPGA board was reported in [29], and another RC was implemented on a FPGA chip for the speech recognition task [25], while in [11], individual logic elements on a FPGA board were used to produce dynamics suitable for RC.

Secondly, the training of the reservoir is of key importance for a good performance. While the traditional offline training methods offer good results [13, 17], their limitations can be critical for real-time applications. Indeed they require the data on the states of the reservoir and target outputs to be collected, stored and then transferred to the post-processing computer. This implies long dead times, which may be detrimental in some applications. An alternative approach is online training in which the readout weights are adjusted in real time using algorithms such as gradient descent, recursive least squares or reward-modulated Hebbian approach, as discussed in [7]. Online training requires minimal data storage. Moreover, these methods allow the reservoir to adapt in real time to changes in the task. Online training is particularly crucial to physically implemented RC working in real time at high data rates [15, 22, 8, 6]: storing and processing the data digitally in such systems constitute a major bottleneck. Note however that online training methods can require more data points than offline training.

We address these issues in the present report through the study of a FPGA implementation of a reservoir computer. We exploit the ring topology, proposed in [24], whose speed is remarkably well adapted to real-time applications [3, 22]. In this architecture, only the first neighbour nodes are connected, resulting in a very sparse interconnection matrix. As also demonstrated experimentally in [3, 15, 22, 8, 6], this simplified architecture provides performance comparable to those obtained with complex interconnection matrices. We illustrate the performance of the implementation on a specific real-world task, namely, the equalisation of a nonlinear communication channel. Training is performed online using a simple gradient descent algorithm [4]. A key issue of implementations on programmable devices is the limited chip resources. In this regard online training is highly advantageous as the exploited gradient descent algorithm is very simple. Furthermore, online training is suited for environment dependent tasks such as the channel equalisation task considered here. We illustrate this by changing the task in real time and verifying that the algorithm converges to a new solution.

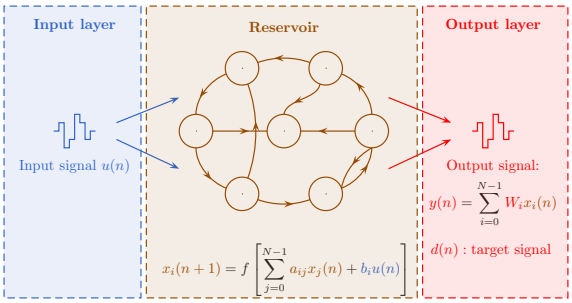

Figure 1. Schematic representation of a reservoir computer. The time multiplexed input signal $u(n)$ is injected into a dynamical system, composed of a large number of nodes $x_i(n)$. The dynamics of the system is defined by the nonlinear function f and the coefficients a_{ij} and b_i. A linear combination of the reservoir states $x_i(n)$ and the readout weights $w_i(n)$ gives the output $y(n)$ that should be as close as possible to a target signal $d(n)$.

The longer term goal of our project is to combine the computational power of a FPGA with the information processing speed of the experimental reservoir computers developed in our group [22, 8]. The present report is a big step towards this goal, since the programming of the FPGA board is definitely the most challenging task of the project. A detailed overview of the practical implications of our implementation to the experimental reservoir computers is presented in the conclusion.

2. Basic principles of reservoir computing

A reservoir computer (schematised in figure 1) is a nonlinear recurrent dynamical system containing a large number N of nodes $x_i(n)$ evolving in discrete time $n \in \mathbb{Z}$. Their evolution is perturbed by some external signal $u(n)$, and their states are given by:

$$x_i(n+1) = f\left(\sum_{j=0}^{N-1} a_{ij} x_j(n) + b_i u(n)\right), \qquad (1)$$

where f is a nonlinear function. The dynamics of the reservoir is determined by the time-independent coefficients a_{ij} and b_i, drawn from some random distribution with zero mean. The variances of these distributions are adjusted to obtain good performance on the tasks considered.

In our implementation, we use a sine nonlinearity $f = \sin(x)$, introduced in [15, 22], and a ring-like reservoir topology, introduced in [24, 3, 22], to simplify the intercon-

nection matrix a_{ij}. The resulting evolution equations are:

$$x_0(n+1) = \sin\left(\alpha x_N(n-1) + \beta b_0 u(n)\right), \quad (2a)$$
$$x_i(n+1) = \sin\left(\alpha x_{i-1}(n) + \beta b_i u(n)\right), \quad (2b)$$

$i = 1, \ldots, N-1$, where α and β are feedback and input gain parameters, and the input mask b_i is drawn from a uniform distribution over the the interval $[-1, +1]$, as in [24, 22, 8].

The output $y(n)$ of the reservoir computer is a linear combination of the states of the variables:

$$y(n) = \sum_{i=0}^{N-1} w_i x_i(n), \quad (3)$$

where w_i are the readout weights. The aim is to get the output signal as close as possible to some target signal $d(n)$. For that purpose, the readout weights are trained either offline (using standard linear regression methods), or online, as described in section 4.

3. Channel equalisation

We focus on a single task to demonstrate the implementation of an online training algorithm on a FPGA board. We chose to work on a channel equalisation task introduced in [21]. This is a popular choice in the reservoir computing community, as it doesn't require the use of large reservoirs to obtain state-of-the-art performance, and it was used in e.g. [13, 24, 22, 8].

The aim is to reconstruct the input of a noisy nonlinear wireless communication channel from its output $u(n)$. The input $d(n) \in \{-3, -1, 1, 3\}$ is subject to symbol interference:

$$\begin{aligned} q(n) = {} & 0.08d(n+2) - 0.12d(n+1) + d(n) \\ & + 0.18d(n-1) - 0.1d(n-2) + 0.091d(n-3) \\ & - 0.05d(n-4) + 0.04d(n-5) + 0.03d(n-6) \\ & + 0.01d(n-7), \end{aligned} \quad (4)$$

and nonlinear distortion:

$$u(n) = q(n) + 0.036q^2(n) - 0.011q^3(n) + \nu(n), \quad (5)$$

where $\nu(n) = A \cdot r(n)$ is the noise with amplitude A and $r(n)$ are independent random numbers drawn (for ease of implementation on the FPGA board) from a uniform distribution over the interval $[-1, +1]$. In our experiments, we considered three levels of noise: low, medium and high, with amplitudes $A_{low} = 0.1$, $A_{medium} = 0.5$ and $A_{high} = 1.0$. These values were chosen for sake of comparison with 20 dB, 13 dB and 10 dB signal-to-noise-ratios reported in [22, 8] (with Gaussian noise).

The aim of the channel equaliser is to reconstruct $d(n)$ from $u(n)$. The performance of the equaliser is measured using the Symbol Error Rate (SER), that is the fraction of symbols that are wrongly reconstructed. For comparison, a 50-neuron reservoir computer with a sine nonlinearity can achieve a symbol error rate as low as 1.6×10^{-4} for a channel with very low Gaussian noise level (SNR = 32 dB), see [22].

In wireless communications, the environment has a great impact on the quality of the signal. Given its highly variable nature, the properties of the channel may be subject to important changes in real time. To demonstrate the ability of the online training method to adapt to these variations, we studied the situation where the channel changes in real time. For this application we set the noise to zero, and considered that the channel is slightly modified, with equation (5) replaced by three slightly different channels:

$$u_1(n) = 1.00q(n) + 0.036q^2(n) - 0.011q^3(n), \quad (6a)$$
$$u_2(n) = 0.85q(n) + 0.036q^2(n) - 0.011q^3(n), \quad (6b)$$
$$u_3(n) = 0.70q(n) + 0.036q^2(n) - 0.011q^3(n). \quad (6c)$$

We regularly switched from one channel to another. The results of this experiment are presented in section 7.2.

4. Gradient descent algorithm

Taking into account the linearity of the readout layer, finding the optimal readout weights w_i is very simple [13]. After having collected all the reservoir states $x_i(n)$, the problem is reduced to solving a system of N linear equations. However, this so-called offline training method requires the use of a large set of inputs and reservoir states. For state-of-the-art experimental performance, one needs to generate thousands of symbols and thus hundreds of thousands of reservoir states. In our experiments [22, 8], the internal memory size of arbitrary waveform generators, used for this task, can become the limiting factor for this method.

Here we use an entirely different approach. Instead of collecting and post-processing data, we generate data and train the reservoir in real time. That is, the inputs are generated one by one, and for every reservoir output a small correction is applied to the readout weights, proportional to the reservoir error from the target output. Our approach falls within the online training category and we use the gradient descent algorithm to compute the corrections to the readout weights (already used in [7]).

The gradient or steepest descent method is a way to find a local minimum of a function using its gradient [4].

For the channel equalisation task, this methods provides the following rule for updating the readout weights:

$$w_i(n+1) = w_i(n) + \lambda \left(d(n) - y(n)\right) x_i(n). \quad (7)$$

The step size λ should be small enough not to overshoot the minimum at every step, but big enough for reasonable convergence time. In practise, we start with a high value λ_0, and then gradually decrease it during the training phase until a minimum value λ_{min} is reached, according to the equation:

$$\lambda(m+1) = \lambda_{min} + \gamma\left(\lambda(m) - \lambda_{min}\right), \qquad (8)$$

with $\lambda(0) = \lambda_0$ and $m = \lfloor n/k \rfloor$, where $\gamma < 1$ is the decay rate and k is the update rate for the parameter λ.

Note that the gradient descent algorithm requires the knowledge of the target values $d(n)$ during the training phase. This is realistic in the case of channel equalisation where standard sequences are transmitted at regular intervals to check the quality of the channel and tune the equaliser if necessary. Here for simplicity we shall suppose that the input $d(n)$ of the channel is known at all times, as the aim is to demonstrate the implementation of the gradient descent algorithm.

The gradient descent algorithm converges relatively slowly to the local minimum, and it has no memory, as the gradient is evaluated each time, disregarding previous states. But its simplicity makes it easy to implement and it offers a certain flexibility, as the parameters λ and γ can be tuned for faster or more precise convergence. It is a reasonable choice for a first implementation. In future work it will be interesting to investigate other online training algorithms, such as recursive least squares [10] (a more sophisticated method that converges faster) or unsupervised learning [16] (which doesn't require exact knowledge of the target output, but only an estimation of the reservoir performance).

5. Simulations

Before implementing the algorithm on the FPGA board, we tested it in software. A Matlab script was written to measure the performance of the training process. The neuron states and the reservoir outputs were produced by the optoelectronic reservoir computer simulation reported in [22], with low noise level (SER $= 20$ dB). A 18-neuron reservoir was used with feedback gain α and input gain β parameters set to optimal values to ensure the best performance of the reservoir.

We ran the script on a notebook equipped with a third generation Intel Core i7-3537U CPU, 8 Gb of RAM and a 256 Gb Samsung PM830 SSD. The R2012a 64-bit Matlab version was running under GNU Linux. With a training dataset containing $100k$ outputs we obtained an average SER of 8.3×10^{-3}. The offline training approach produced a SER of 3.3×10^{-3} on the same data, with a training set of $3k$ outputs. The computer processes $20k$ samples in approximately 6 seconds, thus requiring 300 µs per output.

Note that we didn't devote any effort to performance optimisation and these results are presented only for the sake of comparison. Our goal was to implement a simple online training algorithm on an embedded system, with no intention of competing against a regular CPU in terms of execution speed.

6. FPGA design

The use of a field-programmable gate array (FPGA) offers greater flexibility and better performance compared to a regular microprocessor. Although relatively slow in terms of clock speeds (the best FPGAs clock at around 600 MHz), the FPGA allows to perform independent calculations simultaneously (in parallel), thus making the most of every clock cycle.

For our implementation, we use the Xilinx Virtex-6 chip. The design is written in the standard IEEE 1076-1993 VHDL language [1, 23] and synthesised and placed using tools provided by the board manufacturer (ISE Design Suite 14.7). Xilinx ChipScope Pro Analyser was used to monitor signals on the board and to record the performance in real time. The board has a soldered 200 MHz differential oscillator and a single-ended clock socket populated with a 66.00 MHz oscillator.

Figure 2 presents the simplified schematics of our design. Separate entities (Sim, RC and Train) are shown in different colours. Smaller dashed boxes depict the processes contained in those entities. The `IP Cores` box stands for Xilinx proprietary cores (namely, `icon` and `ila`) used to record signals on the board and transfer the data to the computer via USB.

The simulation entity `Sim` generates data for the reservoir computer. It outputs a time sequence containing reservoir states $x_i(n)$, as shown in figure 1, as well as targets outputs $d(n)$. The entity plays the role of the nonlinear channel and the experimental reservoir computer, and it will be replaced later on by an analog-to-digital converter collecting real data from a physical experiment. A pseudorandom linear congruential generator creates the inputs $d(n) \in \{-3, -1, 1, 3\}$ for the nonlinear channel using equations (4) and (5 or 6). The output $u(n)$ of the channel is used to compute the reservoir states $x_i(n)$ using equations (2a) and (2b). The nonlinear function $f(x) = \sin(x)$ is implemented using the Xilinx CORDIC v5.0 LogiCORE module. The signals $d(n)$ and $x_i(n)$ are fed to the following entities, as if the data came from a real experiment.

The reservoir computing entity `RC` contains two separate processes, which discretise the reservoir states and calculate the reservoir output. The N last reservoir states (i.e. one reservoir) are kept in memory and fed to the `Update` process for the calculations of the new weights, using equation (7). The `Out` process calculates the linear combination of reservoir states using the current readout weights and pro-

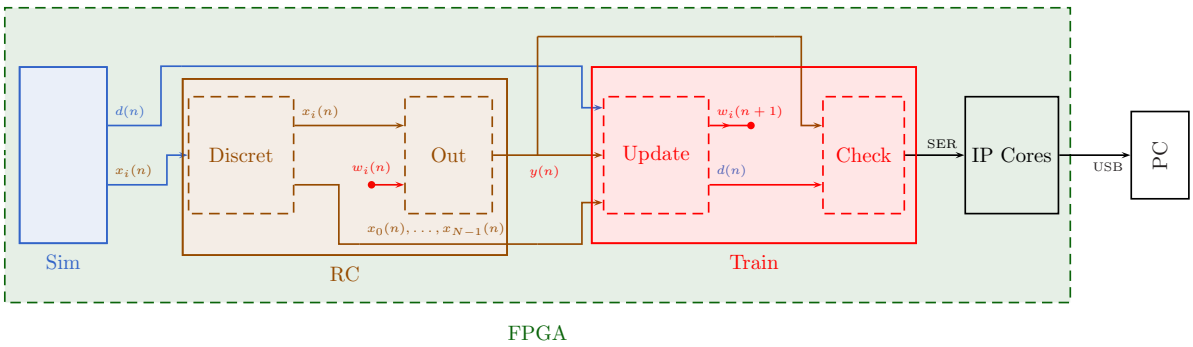

Figure 2. Simplified schematics of our design. The `Sim` entity simulates the nonlinear channel and the reservoir. It generates reservoir states $x_i(n)$ and target outputs $d(n)$. The `RC` entity produces the reservoir output $y(n)$ after calculating the linear combination of discretised reservoir states and readout weights $w_i(n)$. The `Train` entity implements the gradient descent algorithm to update the readout weights $w_i(n+1)$ and measures the performance of the reservoir in terms of symbol error rate (SER). The `IP Cores` box contains proprietary code for transferring data from the board to the computer.

vides the resulting reservoir output.

The training entity `Train` contains the implementation of the gradient descent algorithm. The `Update` process calculates the new readout weights and decreases the value of λ parameter during the training. The `Check` process simply checks whether the reservoir output is correct, that is, if rounding up the reservoir output to the closest channel symbol $y(n) \rightsquigarrow \{-3, -1, 1, 3\}$ gives the correct symbol $d(n)$. The number of errors is counted per set of 1000 symbols and output to the `IP Cores`, which transfers the data to a personal computer for further analysis.

To take into account the possible variations of the communications channel (as described in section 3), the `Update` process monitors the performance of the reservoir computer. If it detects an increase of the symbol error rate above a certain threshold (we used $\mathrm{SER}_{th} = 3 \times 10^{-2}$) after the training is complete (that is, when $\lambda = \lambda_{min}$), the λ parameter is reset to λ_0 and the training starts over again.

Precision and real numbers representation is of key importance when programming arithmetic operations on a bit-logic device. In our design, we implemented a fixed-point representation with 16-bit precision (1 bit for the sign, 4 bits for the integer part and 11 for the decimal part).

Synthesis reports that our design uses only 7% of the board's slice LUTs for logic and arithmetic operations. The block RAM usage is much higher, at 77%, as it is used to store data from the board before transferring through the slow USB port. The current design is driven by a 33 MHz clock, with little effort devoted to optimisation and critical paths analysis. In the future timing performance will be improved by rewriting the design to take maximum advantage of the FPGA's concurrent logic.

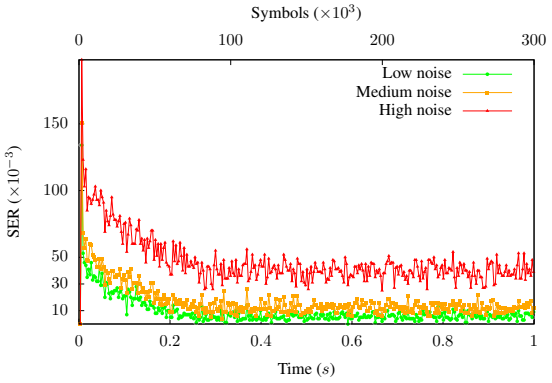

Figure 3. Training results for various noise levels, expressed in terms of symbol error rate. Horizontal axes show both time since the beginning of the experiment (lower axis) and the number of symbols n processed by the FPGA (upper axis).

7. Results

7.1. Constant channel

Figure 3 shows the symbol error rates captured from the FPGA board for different noise levels. Error rates are plotted against time, as well as the number of reservoir computer outputs. The following parameter values were used for the gradient descent algorithm:

$$\lambda_0 = 0.4, \qquad \lambda_{min} = 0, \qquad \gamma = 0.999, \qquad (9)$$

and the λ parameter was updated every 100 outputs (i.e. $k = 100$ in equation (8)). The asymptotic error rate is reached at, approximately, $t = 0.4$ s. The noise level in the channel has no impact on the training time, but increases the final symbol error rate. For a low-noise channel (with $A_n = 0.1$), the symbol error rate is as low as 5.8×10^{-3}, that is, 5 to 6 errors per one thousand symbols. For medium

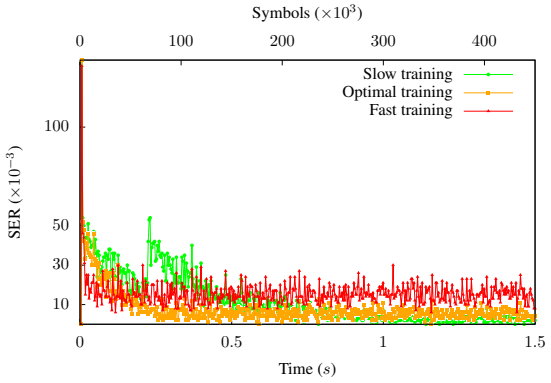

Figure 4. Various training speeds depending on the update rate k. A fast training, with $k = 10$, gives a high SER of 1.55×10^{-2}, while a slow training, with $k = 1000$, produces a very low SER of 2.6×10^{-3} at cost of a longer convergence time. The optimal training with $k = 100$ is a compromise between the performance and speed.

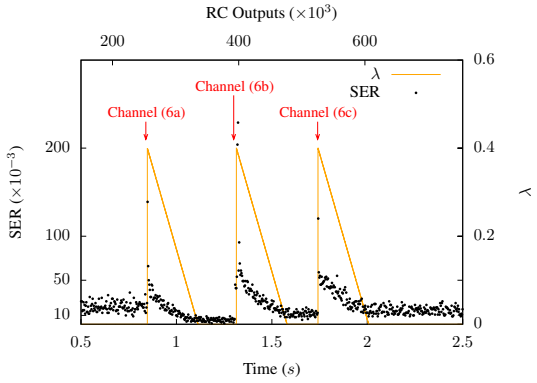

Figure 5. Symbol error rate produced by the FPGA in case of a variable communications channel. The channels are switched at preprogrammed times. The change in channel is followed immediately by a steep increase of the SER. The λ parameter is automatically reset to $\lambda_0 = 0.4$ every time a performance degradation is detected, and then returns to its minimum value, as the equaliser adjusts to the new channel, bringing down the SER to its asymptotic value. The final value of the SER depends on the channel: the more non-linear the channel, the higher the resulting SER.

and high-noise channels (with $A_n = 0.5$ and $A_n = 1.0$), we obtain SERs of, respectively, 1.25×10^{-2} and 4.00×10^{-2}. Similar results were obtained in [21] with a nonlinear adaptive filter. Our experimental results are comparable to those obtained in simulations (section 5), which demonstrates that the simple online training algorithm we implemented performs as well as a standard offline training method.

The overall training speed can be increased by decreasing the λ parameter update rate k, but at the cost of a higher asymptotic SER. On the other hand, increasing k gives a lower asymptotic SER, but slows down the training process. As shown in figure 4, a high update rate $k = 10$ gives a very fast convergence ($t < 0.1$ s) towards a relatively high asymptotic SER of 1.55×10^{-2}, whereas a low update rate $k = 1000$ results in a slow convergence ($t > 1.5$ s) towards a low SER of 2.6×10^{-3}. Our choice of $k = 100$ is a compromise between the final performance (SER $= 5.2 \times 10^{-3}$) and the convergence rate ($t < 1$ s). Note that the decrease of the SER is not always monotonous (see the glitch when using the slow training rate, due to an accidental overshoot of the target output).

FPGA chips are intrinsically faster than standard microprocessors for a specific task. We used this advantage to increase significantly the speed of the training process. As mentioned in section 5, a high-end laptop with a 3 GHz CPU processes $20k$ inputs in approximately 6 seconds, requiring 300 µs per input. The FPGA, clocking at 33 MHz, needs only 0.065 seconds for this task, at a rate of 3.24 µs per input. This speed gain is crucial for future analogue implementations, as discussed in the concluding section.

7.2. Variable channel

Figure 5 shows the performance of our design in case of a noiseless variable communications channel. Every channel switch produces an increase of the symbol error rate, which is detected by the algorithm, causing the λ parameter to be reset to λ_0. The training phase lasts less than half of a second. For each channel configuration, the readout weights are adapted to achieve the best performance possible. Such flexibility would be impossible with an offline training method, as it requires the use of the same static channel during both training and testing phases. Our implementation of this new feature is an important step towards practical applications of reservoir computing to real-life tasks with variable parameters.

Conclusion

We presented what is, to the best of our knowledge, the first FPGA-based reservoir computer with an online learning algorithm. We implemented a basic reservoir computer simulator and a simple gradient descent algorithm on a high-end FPGA board. The standalone design requires no external hardware except a computer to set the parameters and read the results. We tested the reservoir on a nonlinear channel equalisation task and we obtained performance as good as in simulations. We also showed that the gradient descent algorithm could be applied to a task which varies in time.

Our implementation is an important step towards applications of FPGA chips to physical reservoir computers.

First of all, the chip is fast enough to interface in real time with experiments such as the optoelectronic reservoir computer reported in [22]. It will also allow much shorter experiment run time by removing the slow process of data transfer from the experiment to the computer. Moreover, it will solve the memory limitation issue, thus allowing the reservoir to be trained or tested over an arbitrary long input sequence. Lastly, interfacing a physical reservoir computer with a FPGA chip in real time will make it possible to feed the output of the reservoir back into itself, thus enriching its dynamics, and allowing new functionalities such as pattern generation [27].

The online learning algorithm will be particularly useful to train analog output layers, such as the ones reported in [26], where the main difficulty encountered was the necessity of a very precise model of the output layer. Using an FPGA chip to implement online training based on gradient descent algorithm, such a modelling will no longer be required.

On top of these results, the use of a FPGA chip paves the way towards fast reservoir computers, as all aforementioned experimental implementations are limited in speed by the output layer, implemented offline on a relatively slow computer. We presented here an implementation that is two orders of magnitude faster than a high-end personal computer. Note that the speed of our design can be further increased by thorough optimisation, that should allow to raise the clock rate up to 200 MHz. Once the maximum computational power achievable on a FPGA board is reached, an upgrade to an ASIC chip would offer the opportunity to advance towards GHz rates. The present work thus demonstrates the vast computational power that a FPGA chip can offer to the reservoir computing field.

Acknowledgements

We would like to thank Benoît Denègre and Ali Chichebor for helpful discussions. We acknowledge financial support by Interuniversity Attraction Poles program of the Belgian Science Policy Office, under grant IAP P7-35 photonics@be and by the Fonds de la Recherche Scientifique FRS-FNRS.

References

[1] IEEE Standard VHDL Language Reference Manual. *ANSI/IEEE Std 1076-1993*, 1994.

[2] The 2006/07 forecasting competition for neural networks & computational intelligence. `http://www.neural-forecasting-competition.com/NN3/`, 2006. (Date of access: 21.02.2014).

[3] L. Appeltant, M. C. Soriano, G. Van der Sande, J. Danckaert, S. Massar, J. Dambre, B. Schrauwen, C. R. Mirasso, and I. Fischer. Information processing using a single dynamical node as complex system. *Nat. Commun.*, 2:468, 2011.

[4] G. B. Arfken. *Mathematical methods for physicists*. Orlando FL: Academic Press, 1985.

[5] J. Bongard. Biologically inspired computing. *IEEE Comp.*, 42:95–98, 2009.

[6] D. Brunner, M. C. Soriano, C. R. Mirasso, and I. Fischer. Parallel photonic information processing at gigabyte per second data rates using transient states. *Nat. Commun.*, 4:1364, 2012.

[7] K. Caluwaerts, M. D'Haene, D. Verstraeten, and B. Schrauwen. Locomotion without a brain: physical reservoir computing in tensegrity structures. *Artificial life*, 19(1):35–66, 2013.

[8] F. Duport, B. Schneider, A. Smerieri, M. Haelterman, and S. Massar. All-optical reservoir computing. *Opt. Express*, 20:22783–22795, 2012.

[9] B. Hammer, B. Schrauwen, and J. J. Steil. Recent advances in efficient learning of recurrent networks. In *Proceedings of the European Symposium on Artificial Neural Networks*, pages 213–216, Bruges (Belgium), April 2009.

[10] S. Haykin. *Adaptive filter theory*. Prentice-Hall, Upper Saddle River, New Jersey, 2000.

[11] N. D. Haynes, M. C. Soriano, D. P. Rosin, I. Fischer, and D. J. Gauthier. Reservoir computing with a single time-delay autonomous Boolean node. *arXiv preprint arXiv:1411.1398*, Nov. 2014.

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

[13] H. Jaeger and H. Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304:78–80, 2004.

[14] H. Jaeger, M. Lukoševičius, D. Popovici, and U. Siewert. Optimization and applications of echo state networks with leaky-integrator neurons. *Neural Netw.*, 20:335–352, 2007.

[15] L. Larger, M. Soriano, D. Brunner, L. Appeltant, J. M. Gutiérrez, L. Pesquera, C. R. Mirasso, and I. Fischer. Photonic information processing beyond Turing: an optoelectronic implementation of reservoir computing. *Opt. Express*, 20:3241–3249, 2012.

[16] R. Legenstein, S. M. Chase, A. B. Schwartz, and W. Maass. A reward-modulated hebbian learning rule can explain experimentally observed network reorganization in a brain control task. *J. Neurosci.*, 30:8400–8410, 2010.

[17] M. Lukoševičius. A practical guide to applying echo state networks. In *Neural Networks: Tricks of the Trade*, pages 659–686. Springer, 2012.

[18] M. Lukoševičius and H. Jaeger. Survey: Reservoir computing approaches to recurrent neural network training. *Comp. Sci. Rev.*, 3:127–149, 2009.

[19] M. Lukoševičius, H. Jaeger, and B. Schrauwen. Reservoir computing trends. *Künst. Intell.*, 26:365–371, 2012.

[20] W. Maass, T. Natschläger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural comput.*, 14:2531–2560, 2002.

[21] V. J. Mathews and J. Lee. Adaptive algorithms for bilinear filtering. In *SPIE's 1994 International Symposium on Optics, Imaging, and Instrumentation*, pages 317–327. International Society for Optics and Photonics, 1994.

[22] Y. Paquot, F. Duport, A. Smerieri, J. Dambre, B. Schrauwen, M. Haelterman, and S. Massar. Optoelectronic reservoir computing. *Sci. Rep.*, 2:287, 2012.

[23] V. Pedroni. *Circuit Design with VHDL.* MIT Press, 2004.

[24] A. Rodan and P. Tino. Minimum complexity echo state network. *IEEE Trans. Neural Netw.*, 22:131–144, 2011.

[25] B. Schrauwen, M. D'Haene, D. Verstraeten, and J. V. Campenhout. Compact hardware liquid state machines on FPGA for real-time speech recognition. *Neural Netw.*, 21:511–523, 2008.

[26] A. Smerieri, F. Duport, Y. Paquot, B. Schrauwen, M. Haelterman, and S. Massar. Analog readout for optical reservoir computers. In *Advances in Neural Information Processing Systems 25*, pages 944–952. Curran Associates, Inc., 2012.

[27] D. Sussillo and L. Abbott. Generating coherent patterns of activity from chaotic neural networks. *Neuron*, 63(4):544 – 557, 2009.

[28] F. Triefenbach, A. Jalalvand, B. Schrauwen, and J.-P. Martens. Phoneme recognition with large hierarchical reservoirs. *Adv. Neural Inf. Process. Syst.*, 23:2307–2315, 2010.

[29] D. Verstraeten, B. Schrauwen, and D. Stroobandt. Reservoir computing with stochastic bitstream neurons. In *Proceedings of the 16th annual Prorisc workshop*, pages 454–459, 2005.

[30] D. Verstraeten, B. Schrauwen, and D. Stroobandt. Reservoir-based techniques for speech recognition. In *IJCNN'06. International Joint Conference on Neural Networks*, pages 1050–1053, Vancouver, BC, July 2006.