

Reservoir Computing with Stochastic Bitstream Neurons

David Verstraeten, Benjamin Schrauwen and Dirk Stroobandt
Department of Electronics and Information Systems (ELIS), Ugent

{david.verstraeten, benjamin.schrauwen, dirk.stroobandt}@ugent.be

Abstract—Reservoir Computing (RC) [6], [5], [9] is a computational framework with powerful properties and several interesting advantages compared to conventional techniques for pattern recognition. It consists essentially of two parts: a recurrently connected network of simple interacting nodes (the reservoir), and a readout function that observes the reservoir and computes the actual output of the system. The choice of the nodes that form the reservoir is very broad: spiking neurons [6], threshold logic gates [7] and sigmoidal neurons [5], [9] have been used. For this article, we will use analogue neurons to build an RC-system on a *Field Programmable Gate Array* (FPGA), which is a chip that can be reconfigured. A traditional neuron calculates a weighted sum of its inputs, which is then fed through a non-linearity (like a threshold or sigmoid function). This is not hardware efficient due to the extensive use of multiplications. In [2], a type of neuron is introduced that communicates using stochastic bitstreams instead of fixed-point values. This drastically simplifies the hardware implementation of arithmetic operations such as addition, the nonlinearity and multiplication. We have built an implementation of RC on FPGA, using these stochastic neurons.

In this article we present some results regarding the performance results of this implementation. However, the use of stochastic bitstreams requires some additional precautions. We address some of these challenges and present techniques to overcome them.

Index Terms—Reservoir Computing, FPGA, neural networks

I. INTRODUCTION: RESERVOIR COMPUTING

Neural Networks (NN) [3] are tried and true solutions for many complex pattern recognition problems. They have been used for a wide range of applications, such as speech and character recognition, data mining, classification and clustering. Neural networks consist of simple computational nodes called *neurons*. These neurons are interconnected using weighted connections called *synapses* to form a network. Generally, a number of nodes are designated to be inputs – these nodes receive external signals from the environment. Likewise, some of the neurons are interpreted as outputs: these neurons give

the response of the network. The network’s behaviour is determined by the value of the connections and their weights. In order to train the network to perform a certain task, the weights are adapted according to a learning rule, for instance the well known backpropagation rule [8].

The majority of the networks currently used are interconnected in a feedforward manner: the neurons are arranged in layers, and connections only exist from one layer to the next. This means that information flows in one direction through the network. These feedforward networks are well understood and analyzed. However, they suffer from one major drawback: they are in principle only suited to implement functions without a temporal character. The network can formulate a response to a single input vector, but information contained in *time series* of inputs is completely ignored. This is because of the feedforward structure of the network: the network is not able to integrate information from different time steps since the information flows only forward.

The standard solution to this problem is to use windowing to transform the time-dependent inputs into stationary input vectors (these networks are called Time Delay NN’s (TDNN)). However, these networks always impose a bias for a certain timescale and introduce extra parameters into the model, which is undesirable. Another solution is to avoid the layered feedforward structure and to permit recurrent connections with delay inside the network. This way, information can flow back into the network and be incorporated into calculations of future timesteps. These types of networks are called Recurrent Neural Networks (RNN’s). These networks have an inherent time processing capability: they are well suited to solve temporal problems. However, the recurrent connections also make these RNN’s far more difficult to train and control. Indeed: these networks can be interpreted as a very complex, time dependent filter with feedback, and it is well known from system theory that these types of systems are prone to chaotic behaviour and are difficult to analyze. Most learning rules that exist for training RNN’s suffer from problems with convergence or slow learning rates.

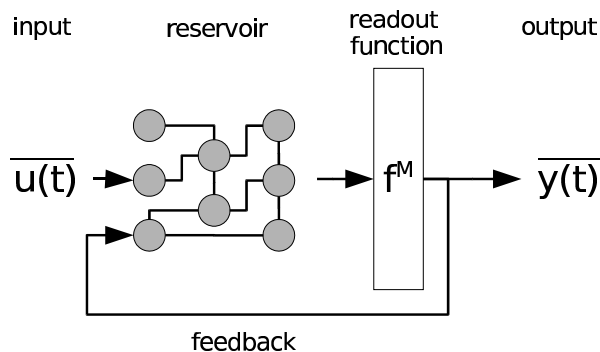


Fig. 1. Structure of Reservoir Computing.

Recently, an elegant solution to this problem was proposed independently by three different researchers [6], [5], [9]. These three solutions are very similar and can be grouped under the general term *Reservoir Computing* (RC). The principle of RC is as follows: a RNN is constructed in a more or less random fashion¹, and is interpreted as a *reservoir* into which the input is fed. After the network is constructed, it is left untouched: the reservoir itself is not trained. Instead, a so-called *readout function* is trained (see Figure 1). This readout function can be any type of pattern classification or regression algorithm. The readout function looks at the state of the reservoir in reaction to a certain series of inputs, and computes the output of the system. Generally, the readout function is a very simple algorithm, requiring far less time and computational effort to train. This effectively solves the learning problem of RNN's, while preserving the interesting temporal processing capabilities of these networks.

RC is a very time-efficient way to train a system to perform a certain task, but the reservoir itself still needs to be simulated. The precise computational effort required depends on the model for the neurons used in the network and the interconnection topology, but even for simple models it is generally rather slow – especially for larger networks – since a sequential processor needs to calculate every node separately to simulate a single time step of the network. Since NN's are parallel by construction, a logical step would be to use specifically designed hardware to parallelize the calculations and thus obtain a considerable speedup. However, this transition from a processor to dedicated hardware is far from trivial, and forms the main topic of this article.

In Section II we will outline the model of the computational node we used to construct our networks. In Section III we discuss the implementation details of out

¹The construction of these networks is one of the active research topics.

setup. In Section IV we outline the experimental setup and results we performed for this research, and finally in Section V we formulate some conclusions about our work.

II. STOCHASTIC BITSTREAM NEURONS

We already mentioned the computational cost associated with the simulation of large neural networks on a sequential instruction processor. A rather obvious solution to this problem is to implement the network in hardware: this way the simulation can exploit the parallelism present in the network fully and compute each neuron in parallel.

Many different types of neural models exist, but the majority of the analogue² models compute a weighted sum of their input values and then run this sum through some form of non-linear function – usually this is a sigmoid-type non-linearity such as a $\tanh()$ function. These sigmoidal neurons are not very hardware-friendly: they require a large amount of multipliers and adders, and the non-linearity is usually not very space-efficient either. Many hardware adaptations of neural models have been proposed and successfully implemented. However, in this article, we will be focusing on a specific type of hardware neuron – one that is specifically designed with an implementation on digital hardware in mind: the stochastic bitstream neuron [4].

Stochastic neurons are nothing more than traditional sigmoid-type neurons. However, instead of communicating through analogue (e.g. fixed-point) values, they use stochastic arithmetic. In stochastic arithmetic, a value is represented by the statistical characteristics of a bitstream. In practice, two different ways of converting values to bitstreams are used: for the unipolar representation, a value $w \in [0, 1]$ is represented by a stochastic bitstream W where the probability of having a 1 is $P[W_i = 1] = w$, for the bipolar representation a value $w \in [-1, 1]$ is represented by a stochastic bitstream W where the probability of having a 1 is $P[W_i = 1] = w/2 + 0.5$. It is clear that this representation introduces an error that decreases as the bitstreams under consideration get longer, but this disadvantage is in our case balanced by a spectacular simplification of the basic arithmetic operations. Indeed: if two bitstreams are statistically independent, the product of their values is simply the bitwise AND for unipolar representation and a bitwise XNOR for bipolar representation. Likewise, the scaled sum of a number of bitstreams can be implemented as

²So called *spiking* neurons also exist that communicate through isolated pulses. We do not regard these models in this article.

I	A_2	A_1	A_0	O_{uni}	O_{bip}
0	0	0	0	0	$\overline{w_7}$
0	0	0	1	0	$\overline{w_6}$
...
0	1	1	1	0	$\overline{w_0}$
1	0	0	0	w_7	w_7
1	0	0	1	w_6	w_6
...
1	1	1	1	w_0	w_0

TABLE I

LUT-RAM SYNAPSE IN THE CASE OF UNIPOLAR (O_{uni}) AND BIPOLAR (O_{bip}) CODING.

selected at timestep t by the three address lines is

$$P(\text{bit } j \text{ selected at time } t) = P\left(\sum_{i=0}^2 A_i^t 2^{i+1} = j\right)$$

If we choose the values a_i so that $P(\text{bit } j \text{ selected}) = 2^{-j}$, we maximize the precision of the output bitstream and we can directly write the binary form of the weight value in the LUT-RAM, i.e. bit $j = w_j$. This gives :

$$\begin{aligned} a_2 a_1 a_0 &= 2^{-1} \\ a_2 a_1 (1 - a_0) &= 2^{-2} \\ &\dots \\ (1 - a_2)(1 - a_1)(1 - a_0) &= 2^{-8} \end{aligned}$$

In [2] the least-squares solution for this overdetermined system of equations is used. We made a slight adjustment to find the desired probability values for a_i : since the influence of an error on the more significant weight bits on the result are greater, we should like the error for these bits to be relatively smaller. We therefore weighed each equation above with a factor equal to the right-hand side when computing the square-error. In effect, this decreases the errors on the most-significant bits. The solution to this system of equations is : $a_2 = 0.9385$, $a_1 = 0.7993$ and $a_0 = 0.6665$. If we then use the neuron input bitstream I as fourth address line, and fill the remaining 8 bits of the LUT-RAM with zeros in case I uses unipolar representation (which corresponds to an AND operation), and with $\overline{w_j}$ in case I uses bipolar representation (which corresponds to an XNOR operation), the output bitstream O represents exactly the product of the I and W .

- **Bs-gen functionality** The LUT-RAM can also be used in a similar manner to generate a bitstream encoding the value in its memory. It suffices to replace the input bitstream I with a fourth address

bitstream a_3 , and to recalculate the four address values according to the following system of equations:

$$\begin{aligned} a_3 a_2 a_1 a_0 &= 2^{-1} \\ a_3 a_2 a_1 (1 - a_0) &= 2^{-2} \\ &\dots \\ (1 - a_3)(1 - a_2)(1 - a_1)(1 - a_0) &= 2^{-16} \end{aligned}$$

This results in the values $a_3 = 0.9907$, $a_2 = 0.9468$, $a_1 = 0.7998$ and $a_0 = 0.6665$. Due to the fact that an extra address line can be used, the precision of the encoded value has now doubled to 16 bits. The contents of the LUT-RAM in this case are represented in Table II.

A_3	A_2	A_1	A_0	O
0	0	0	0	p_{15}
0	0	0	1	p_{14}
...
0	1	1	1	p_8
1	0	0	0	p_7
1	0	0	1	p_6
...
1	1	1	1	p_0

TABLE II

LUT-RAM USED AS A PROBABILITY ENCODER.

The readout function can be any type of classification or regression algorithm, that takes information from the reservoir as input and calculates the output. However, because the reservoir effectively projects the input into a high-dimensional space, the problem becomes easier to solve, and a simple linear discriminant or regressor suffices: a weighted sum of the neuron states is calculated, and in the case of the discriminant winner-take-all is applied. This simple type of readout permits us to again use the LUT-RAM-based multipliers, in this case to calculate the output of the readout function. This allows us to implement the whole RC-framework on-chip in a very space efficient way.

IV. EXPERIMENTS

We have implemented the ideas described above in VHDL and performed some simple experiments as a proof of concept. All experiments that will be described are simulated in ModelSim. They all use a reservoir consisting of 25 neurons, connected in a sparse, *small-world* [1] topology (these types of reservoirs are called *liquids*). Each time, a single sine wave was input into the network, and the readout function was trained to perform a certain task. The windowlength used when converting values to and from bitstreams is 2^{16} bits.

The first task was to reproduce the input sine. The output of the RC system for this task are depicted in the middle graph of Figure 3. It shows that the readout function is able to extract the original sine input from the reservoir very well.

The next task was, given a sine as input, to output a phase-shifted version of the signal. This experiment is a good indication of the memory-function of the network: it has to remember the input for a period of time to be able to reproduce it. We trained different readout functions to reproduce sinewaves with increasing phase-shifts, and computed the Normalized Root Mean Square Error (NRMSE) for phase-shifts $\phi \in [0, 2\pi]$. The results are shown in Figure 4. The figure shows that the RC-system generally performs very well at this task. It also appears to be most difficult to generate sines with phase-shifts of $\phi = \pm\pi/2$, which was to be expected: these signals are least present in the reservoir states, as can be seen in figure 3. There, a plot of the reservoir states is shown (top graph) along with the desired output (bottom graph).

V. CONCLUSIONS AND FUTURE WORK

In this article we have presented an implementation of the RC framework using stochastic bitstream neurons. These types of neurons allow a very FPGA-friendly implementation, thus permitting larger networks to be constructed. As a proof of concept, we have constructed a small reservoir and trained a readout function to perform some simple tasks. We were able to show that temporal information is retained in the reservoir state and can be used by the readout function for computations.

The next step in our research is to go from simulations over to actual hardware. This will allow us to validate our expectations about the area-efficiency of these networks, and also to apply them to more challenging real world tasks like speech recognition [10].

REFERENCES

[1] L. Adamic. The small world web. <http://www.hpl.hp.com/research/idl/papers/smallworld/smallworldpaper.html>, 2004.

[2] S.L. Bade and B.L. Hutchings. FPGA-based stochastic neural networks-implementation. *IEEE Workshop on FPGAs for Custom Computing Machines*, 1994.

[3] C. M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, 1995.

[4] Bradley D. Brown and Howard C. Card. Stochastic Neural Computation I : Computational Elements. *IEEE Transactions on Computers*, 50:891–905, 2001.

[5] Herbert Jaeger and Harald Haas. Harnessing nonlinearity: predicting chaotic systems and saving energy in wireless telecommunication. *Science*, 308:78–80, April 2 2004.

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

[7] T. Natschläger, N. Bertschinger, and R. Legenstein. At the edge of chaos: Real-time computations and self-organized criticality in recurrent neural networks. In *Proceedings of NIPS '04*, pages 145–152, 2004.

[8] D. Rumelhart, G. Hinton, and R. Williams. *Learning internal representations by error propagation*. MIT Press, Cambridge, MA, 1986.

[9] J. J. Steil. Backpropagation-Decorrelation: Online recurrent learning with $O(N)$ complexity. In *Proceedings of IJCNN '04*, volume 1, pages 843–848, 2004.

[10] D. Verstraeten, B. Schrauwen, D. Stroobandt, and J. Van Campenhout. Isolated word recognition with the liquid state machine: a case study. *Information Processing Letters*, 95(6):521–528, 2005.

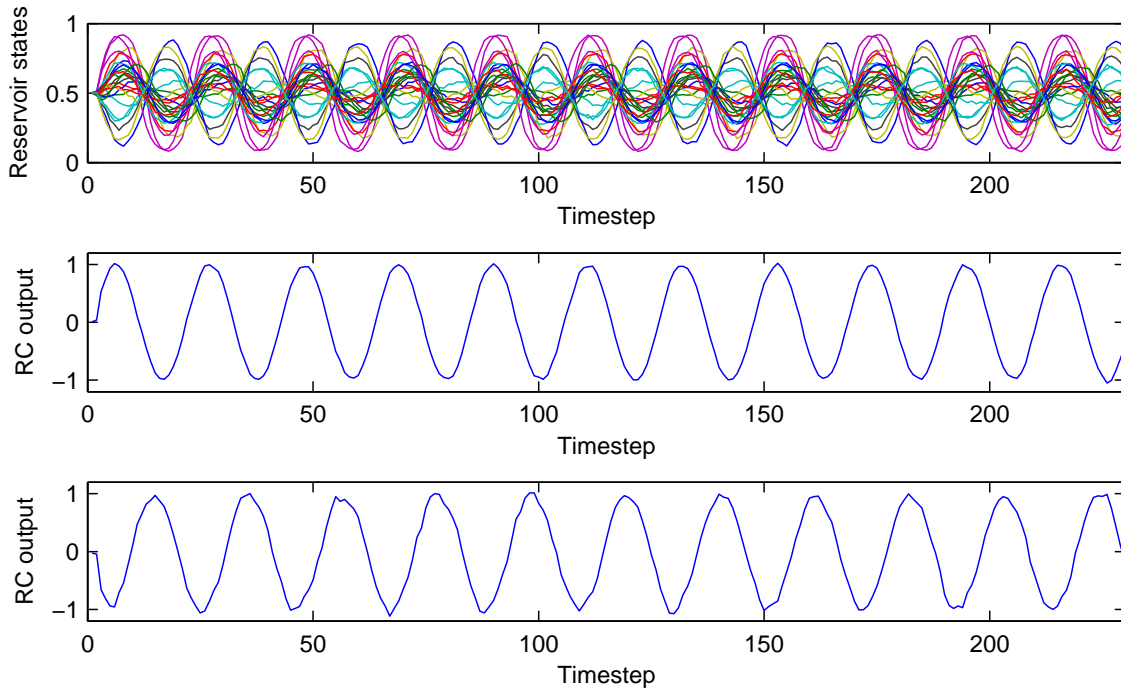


Fig. 3. The top plot shows the reservoir states, the middle plot shows the reproduction of the input sine by the readout function, and the bottom plot shows the phase-shifted sine ($\phi = -\pi/2$) calculated by the readout function.

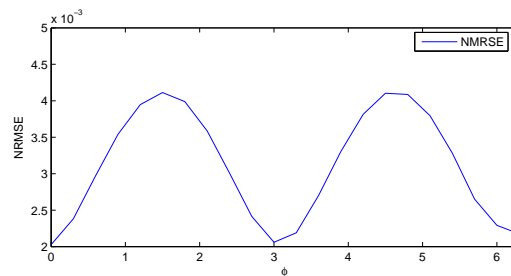


Fig. 4. The Normalized Root Mean Square Error (NRMSE) for increasing phase-shifts of the target sine.