Reservoir Computing with Cellular Automata and Spiking Neural Networks

Øyvind Robertsen
Department of Computer and Information Science
Norwegian University of Science and Technology
Sem Sælands Vei 9, 7491 Trondheim, Norway
oyvinrob@stud.ntnu.no

***Abstract*—Reservoir Computing (RC) is a new and interesting approach to machine learning in which temporal input is imposed as perturbations on a dynamic reservoir and output is read out by performing a linear classification of reservoir state some time after the initial perturbation. While Recurrent Neural Networks are a common choice of reservoir, any dynamical system exhibiting the ability to let perturbations "echo" through the system over time and to respond distinctly to different inputs could be used. In this paper we are interested in systems where the output is spiking in nature, such as biological neurons and cellular automata (CA). Implementations of RC-systems with spiking reservoirs typically convert spiking data to analog values before performing classification. This conversion can introduce unwanted biases to the system, as well as being potentinally expensive to implement in hardware. In this paper, we introduce a linear classifier operating in the spiking domain, the Simplified Spiking Neural Network (SSNN), that avoids this conversion. To show that this model is viable, we implement a software simulation of an RC-system consisting of a one-dimensional, uniform cellular automata and a small SSNN and train the network so that is able to distinguish between the dynamics emerging in the reservoir from different initial states.**

We also implement a novel RC-system on an FPGA, using a Dynamical System with Dynamical Structure $(DS)^2$ as a reservoir and an SSNN as a readout layer. $(DS)^2$ systems are interesting in the context of reservoir computing because they have the ability to self-regulate and adapt their own dynamics based on their environment. This allows for agents that are capable learning on their own, as opposed to being trained. We implement the $(DS)^2$ reservoir as a non-uniform CA in which the state transition function in each cell is subject to development over time.

I. Introduction

In recent years, research into computation using non-traditional physical mediums and paradigms, so called unconventional computing, has seen increased interest. With challenges currently facing traditional architectures, such as the von Neumann bottleneck and ensuring continued scalability and reliability, unconventional computation presents possible solutions from a new perspective. Instead of designing architectures top-down, by composing complex units and orchestrating their interaction, a bottom-up approach is employed, where complex behaviour emerges from local interactions between simple units. Cellular computing, introduced by Sipper in [1], is an example of one such paradigm. These same principles are also used to explore computational capabilities in unconventional materials, as shown in [2].

Reservoir computing (RC) is a novel approach to machine learning and intelligent systems, in which input data is imposed as perturbations on a dynamic system (the reservoir) and output is generated by performing a linear classification of the reservoir state [3]. With RC as a field originating from the study of Recurrent Neural Networks (RNNs), these are a common choice of reservoir in RC implementations [3]. The focus of this paper however, is the use of developmental systems [4] as a reservoir and how to perform readout from such a system. Specifically, we are interested in developmental systems based on non-uniform Cellular Automata (CA) [5]. Where traditional CAs have fixed transition functions throughout a simulation, determining how the state of a cell should update based on the state of its neighbors and itself, these functions are subject to change in developmental systems. Based on growth rules and environment feedback, developmental systems change their dynamical behaviour. This ability is particularily interesting in the context of unconventional computation, as it allows for architectures with self-organizing, self-repairing and self-scaling properties. A major advantage of CA-based reservoirs over ones based on RNNs, is that they allow for efficient implementation in hardware, since no expensive floating-point operations are required.

In RC systems where the reservoir output is spiking in nature, such as (developmental) cellular automata and biological neurons, spiketrains from the reservoir is typically converted to analog values before being processed by a linear readout layer. This is usually done by resampling spiketrains and applying exponential filtering before classifying the output with linear/ridge regression [3]. Converting to, and performing classification in the analog domain, introduces expensive floating-point operations to the RC pipeline. In the interest of achieving efficient end-to-end simulation of a CA-based RC system in hardware, a readout layer that can process spiking data without need for conversion is needed.

In this paper we propose a simplified spiking neural network model (SSNN) suitable for use as a readout layer in an RC system. This model is capable of classifying reservoir dynamics based on unconverted spiketrain data. As a proof of principles, we implement a software simulation of an RC system consisting of a one-dimensional, homogenous CA and a SSNN, and show that the system can compute XOR.

We also implement a RC-machine based on a cellular

developmental system on reconfigurable hardware, a field-programmable gate array (FPGA), where the readout layer is implemented using the above mentioned SSNN model.

This paper is organized as follows: Section II gives relevant background and Section III describes the Simplified Spiking Neural Network model. In Section IV the cellular, developmental RC-machine implemented in hardware is presented. Section V describes the software and methodology used to carry out the experiment described in Section VI. Finally, Section VII offers a short conclusion, followed by an overview of future work in Section VIII.

II. Background

A. Reservoir Computing

Artificial Neural Networks (ANNs) are a commonly used computational model in machine learning and bio-inspired computing. Simple, feed forward ANNs lend themselves well to problems were data can be spatially correlated, such as classification. Many real world problems however, are temporal in nature. Recurrent neural networks (RNNs) have been shown to be powerful tools for solving temporal problems such as stock market prediction [6], learning context free/sensitive languages [7] and speech synthesis [8]. Training RNNs is computationally expensive and often requires application specific adaptions of generalized training algorithms in order to reliably converge [9]. Several techniques have been proposed that circumvent problems related to training, such as Echo State Networks [10] (ESNs), Liquid State Machines [11] (LSMs) and Backpropagation Decorrelation learning [12]. These all share the common feature of only training weights of the output layer of the network, while leaving the hidden layers of the network untrained or simply subject to weight scaling. In [13], Verstraeten et al. propose that systems based on this idea should be unified under the term reservoir computing (RC).

In general, reservoir computing as a term describes any computational system where a dynamic reservoir is excited by input data and output is generated by performing classification/regression over reservoir state. Figure 1 shows the basic architecture of any reservoir computing system. With its origins in research on various types of recurrent neural networks and training thereof, the reservoir in RC systems is often represented as an RNN [13]. However, any dynamic system capable of eventually forgetting past perturbations and of responding distinctly to different perturbations, can in principle be used. Snyder et al. [14] investigate using Random Boolean Networks, Yilmaz uses Cellular Automata [15] and Fernando et al. use a bucket of water [16].

B. Cellular Automata and EvoDevo Systems

John von Neumann introduced cellular automata (CA) as a discrete computational model based on local interaction of cells on a grid of finite dimensionality [17]. At any timestep t during the simulation, each cell in the grid is in one of a finite number of states. The state of any cell at time $t+1$ is computed as a function of the cells and its neighboring cells current

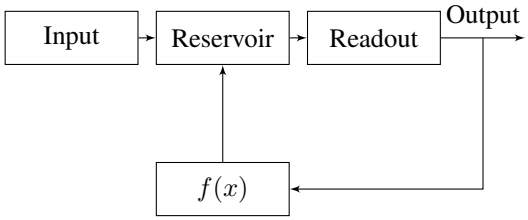

Fig. 1. Basic overview of an RC system.

states. Under this simple scheme, von Neumann showed that advanced properties such as autonomous self-reproduction is possible. In later years, much research has gone into both qualitative and quantitative analysis of the capabilities of CAs as an abstract model of computation [18] [19].

Artificial evolution is an important tool in biologically inspired computing. Inspired by Darwin's theories of evolution, artificial evolution is often used to solve problems by encoding potential solutions as individuals in a population and applying mechanisms such as reproduction, recombination, mutation and selection on said population to breed better solutions with regard to some fitness function [20]. Of particular interest with regards to the work being presented in this paper, is the use of evolution to explore the rule-space of CAs [21]. In [22], Tufte describes how artificial evolution and developmental techniques can be combined to grow multicellular systems for which both structure and behaviour are emerging properties, so called Dynamical Systems with Dynamical Structures ($(DS)^2$) [23]. Where dynamical systems, such as Random Boolean Networks and CAs, have fixed structural topology and state transition functions, $(DS)^2$ systems allow these properties to undergo a developmental process. Through this process, these systems can self-regulate and adapt their dynamics and change their possible trajectories through state space. In this paper and in future work, we are interested in exploring the capabilities of these kinds of systems when used as reservoir.

C. Spiking Neural Networks

Artificial neural networks can be grouped into three generations, based on the characteristics of their base computational unit, the neuron. The first generation, based on McCulloch-Pitts neurons [24], simple threshold gates, allows for universal computation on digital input/output values. In the second generation, neurons apply a non-linear, continuous activation function on the weighted sum of their inputs.

The third generation of networks bases itself on spiking neurons, which model the interaction between biological neurons more closely. In this model, a neuron v fires when its potential P_v exceeds a threshold θ_v. The potential is, at any time, the sum of the postsynaptic potentials, resulting from firing of presynaptic neurons. The contribution of a spike from presynaptic neuron u at time s to the potential P_v of postsynaptic neuron v is given by $w_{u,v} \cdot \varepsilon_{u,v}(t-s)$, where $w_{u,v}$ is a weight representing the strength of the synapse connecting u and v, and $\epsilon_{u,v}(t-s)$ models the response of the

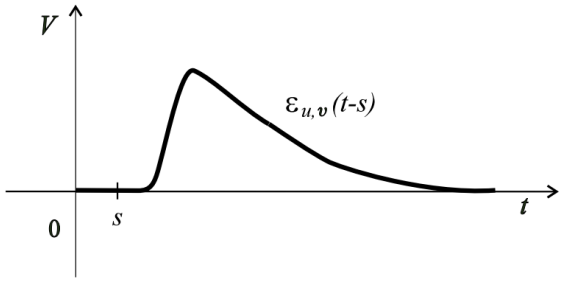

Fig. 2. Common spike response function shape, figure taken from [25].

spike as a function of time passed since the spike occurred. A synapse can be both excitatory and inhibitory, meaning that its contribution to the total potential P_v can be both positive and negative. A biologically plausible response function is shown in figure 2. From a machine learning perspective, the trainable part of a spiking neural network, is the weight $w_{u,v}$, determining to what degree spikes from a neuron u influences the potential of neuron v.

In [25], Maass shows that spiking neurons are at least computationally equal to the models used in generation one and two, and that they can also be more efficient in terms of neurons required to compute a function. SNNs also have the required attributes to be used as a reservoir in an RC system [26]. In order to be able to create an efficient implementation of a RC-machine based on a cellular developmental system as described in IV, being able to perform classification in the spiking domain is essential.

III. Simplified Spiking Neural Network

In this section, we present a simple neural model inspired by the SNN model outlined in Section II-C. The basic architecture of each neuron is shown in Figure 3.

Similarily to their biological counterparts, the potential for a neuron to fire increases as more spikes come in. For each incoming edge/synapse, a neuron has a separate counter/weight-pair. Counters are incremented every time a spike comes in via the corresponding synapse. Weights act as thresholds. For each synapse, they represent the minimal number of incoming spikes required for the neuron to fire. An incoming spike to a counter that is already equal to its weight, will not cause the counter to increment. When all counters are equal to their weights, the neuron fires, and the counters reset.

Counters are also susceptible to decay over time. If at any timestep a spike is not occurring for some incoming edge, the corresponding counter is decremented.

IV. $(DS)^2$ RC-Machine

An FPGA-prototype of a $(DS)^2$ RC-machine with a SSNN readout layer was implemented as a part of this project. In this section, we provide an overview of the functionality and implementation of the reservoir and readout components of the prototype, as well as numbers on resource usage and scalability.

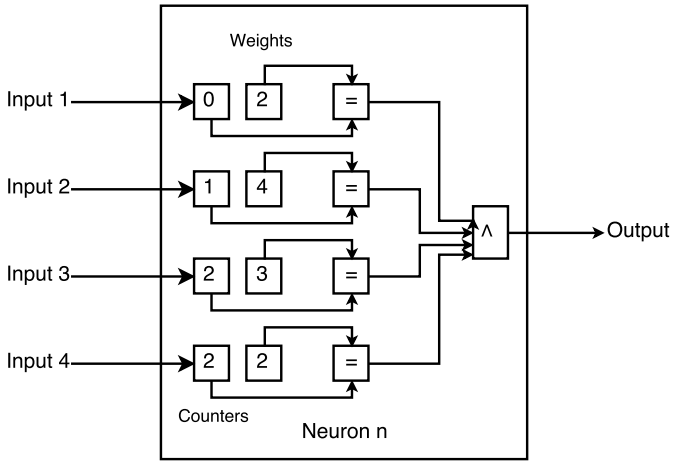

Fig. 3. Block diagram for the base computation occuring in the simplified spiking neurons.

A. Developmental Reservoir

The implemented reservoir is a developmental system based on the work presented by Tufte in [27]. In this model, the functional behavior of each cell on a regular grid of finite dimensionality is decided by its type. To model emergence of both structure and behaviour, the rules governing growth and structural development, the genome, is applied to all cells every so called Development Step (DS). These rules are expressed as a condition and a result, with the condition being dependent on both the current type and state of the cells in the von Neumann neighborhood [1] of the cell being developed. Growth is in other words encoded in the genome in much the same way as transition functions are in traditional CAs. It is important to note that the genome is not a blueprint describing exactly how the final result should look, but rather a description of how to build a system.

Each DS consists of a set number of State Steps (SS). Every state step, the state of each cell is updated. The type of the cell determines its behavior, or which transition function to use to evaluate cell states every state step. As in traditional CAs, the state of cells in the von Neumann neighborhood is used as input to the transition function.

A system operating under the regime described in this section, will essentially be a heterogeneous CA where the transition function used in each cell changes over time. Since both the type and state (environment) is taken into account during development, the system can adapt to and self-regulate its own behavioral dynamics. This property is of particular interest in the context of machine learning, as it allows for agents that are capable of learning, as opposed to the current norm, agents that can be trained.

Figure 5 shows the development of a reservoir starting from a single green cell, under the growth rule from Figure 6. For each growth rule, the label underneath the neighborhood

[1]The cells directly north, east, south and west of a cell, as well as the cell itself.

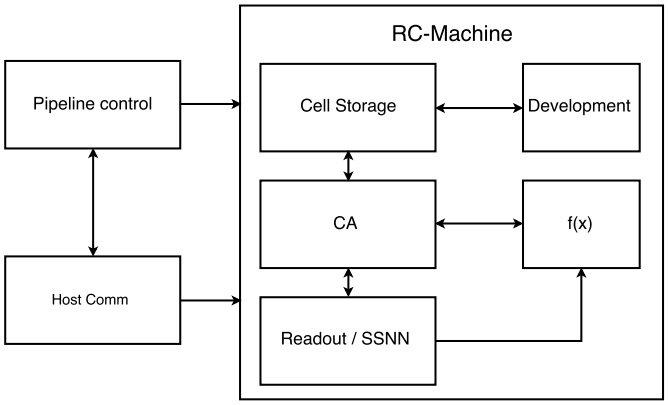

Fig. 4. High-level system architecture for the $(DS)^2$ RC-machine

indicates the direction from which the cell should grow. For instance; Gw, Grow west, indicates that the rightmost cell in the neighborhood should be copied into the center cell. Analogous rules apply for Grow north and east, as well as a special case for Grow south. A cell with a green northern neighbor will turn into a red cell, while one with a red northern neighbor will turn green. The functional difference between the red and green cell type is that they use different state transition functions. For the neighborhoods not enumerated here, the center cell remains unchanged. A subset of statesteps for DS 1 and DS 2 is shown to illustrate how cell type affects behavior. For the sake of simplicity, this rule does not take the state of neighboring cells into account when performing a development step, only their type.

B. FPGA implementation

An FPGA-based implementation of a developmental system as described here was completed and verified by Lundal in [28]. Figure 4 shows a high-level overview of the system architecture. The system is implemented as a three-stage interlocked pipeline, consisting of Fetch, Decode and Execute stages. Cell states and types is stored in On-chip Memory, so called Block RAM (BRAM) in the Cell Storage module. At each execute stage, one of two modules, either the Development or the CA module, operates on the data. Development steps happen in the development module, where growth rules are stored in BRAM. Whenever a DS instruction occurs, all cells are fetched from the Cell Storage and tested against all rules. Similarily for state steps, cells are moved to the CA module, a two- or three-dimensional grid of Sblocks [29], allowing for fully parallel execution of the behavior of the system. Each sblock is connected the sblocks in the von Neumann-neighborhood around it, receiving their states as input. The type of a cell determines the functionality configured in the sblock LUT.

New hardware, equipped with Convey Wolverine WX-2000 coprocessors (see micron.com) based on the Xilinx Virtex-7 XC7V2000T FPGA (see xilinx.com), has since been aquired and the system has been ported to work on this new platform. In an effort to further modernize the codebase, the system has been partially ported from VHDL to Chisel, a Hardware Description Domain Specific Language implemented in Scala (see chisel.eecs.berkeley.edu).

The architecture of the developmental system as implemented by Lundal remains largely unchanged. To allow the readout layer to process data in real time, without the need for an intermediate instruction between state steps, support has been added for marking sblocks as output blocks. The state from all output blocks is wired directly to the readout layer. This happens at synthesis time, as the amount of FPGA resources required for making it possible to reconfigure which blocks are used as output blocks at runtime is immense. For large reservoirs, the routing of state-signals from sblocks to the readout layer will require a considerable amount of resources in itself, making it necessary to limit the number of output sblocks. While this will limit the readout layer's "view" into the dynamics of the reservoir, the hope is that the reservoir will adapt around this, based on the feedback it receives from the readout layer.

C. Readout

Based on the basic description of each neurons computational functionality in Section III, a parameterized FPGA-implementation was created to perform realtime classification of the readout from the developmental reservoir described in the previous section. Given a description of a network topology (i.e. the number of neurons in each layer and the number of output cells in the reservoir), and weights for all neurons, hardware is generated accordingly, as shown in Figure 7. For each layer, a module encapsulating the neurons in that layer is synthesised, with each neuron being modeled as a separate entity within the layer. Neurons are implemented very similar to how they are depicted in Figure 3. Incoming spikes are counted in a set of registers, one for each neuron in the preceeding layer, which are used to determine wether or not the neuron should fire by comparing register values with statically configured weight values. In addition, circuitry for resetting the counters after a neuron fires has been added.

Layers in the SSNN form a pipeline, through which data propagates in sync with the state steps of the reservoir. For two layers in the SSNN, A and B, where B directly follows A in the pipeline, the output from neurons in layer A at time t is used as input for the neurons in layer B at time $t+1$. As mentioned, all neurons are synthesised with one counter per neuron in the preceeding layer. This means that all networks are implemented fully connected. That is, all neurons in one layer are connected to all neurons in the next. Connections can be pruned by setting a weight value of zero.

D. Resource Usage

Table I shows resource usage for different configurations of the developmental system in the RC-machine. The percentage utilization is based on the total amount of resources available on the Virtex-7 XC7V2000T FPGA. We see that for the configuration with $96 * 96 = 9216$ cells, $\sim 20\%$ of LUTs

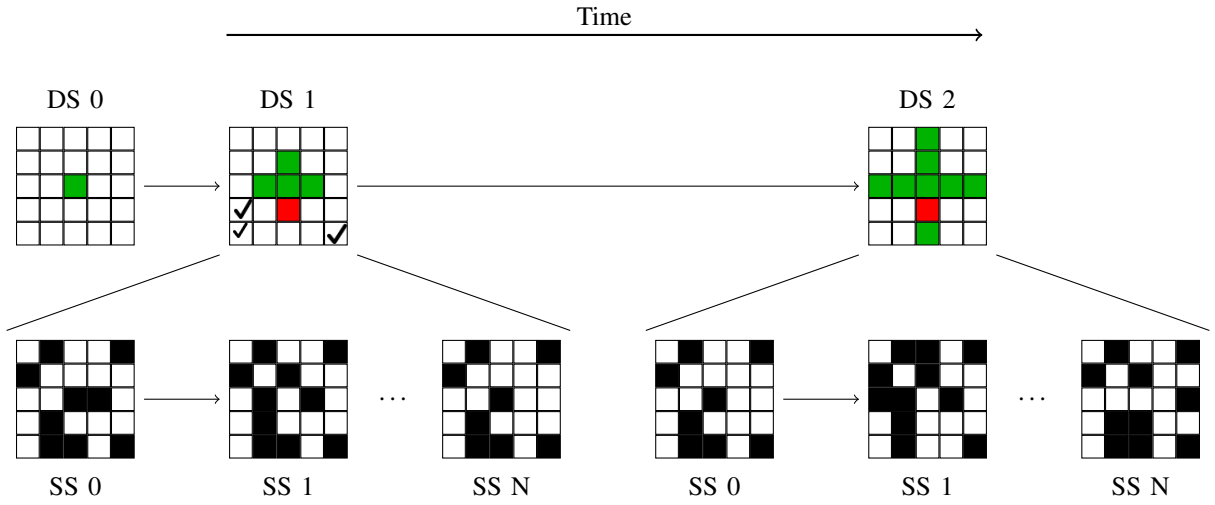

Fig. 5. Development starting from a single green cell using the growth rule in Figure 6.

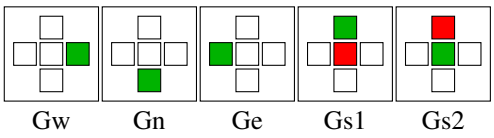

Fig. 6. Growth rules for a cellular developmental system where cells are either empty or of the green type.

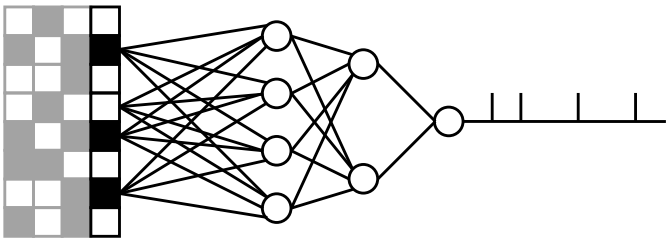

Fig. 8. At each timestep, the state of the CA is fed to the input-layer of the SSNN, which propagates updates through the network layers in a pipelined fashion.

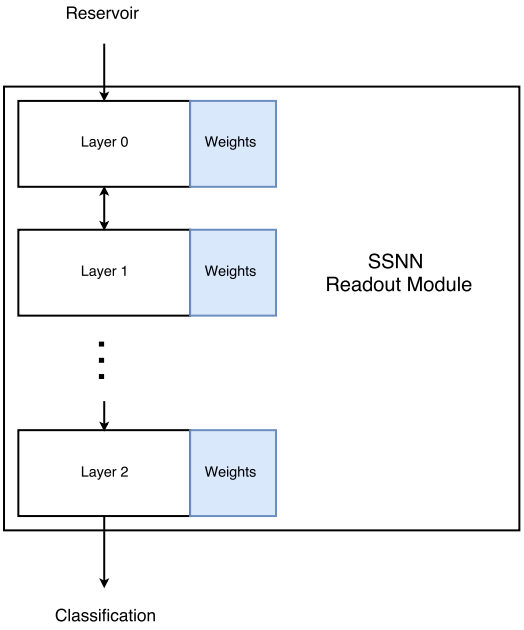

Fig. 7. HW architecture for the SSNN readout module.

and $\sim 24\%$ of available BRAMs are utilized. It should, in other words, be possible to scale the developmental system up to approximately 36000 cells on this hardware. In a full RC-system, some resources will also need to be used by the SSNN, limiting the maximum size of the reservoir further.

V. Methodology

To demonstrate the viability of a SSNN as a readout-layer in a RC-system, a simple software simulation was implemented in the Python programming language. The implementation consists of three modules; a one-dimensional, homogenous, binary CA, an SSNN implementation and a genetic algorithm. Figure 8 shows how the CA and the SSNN is connected.

A. Cellular Automata Module

The CA consists of an array of 0s and 1s, and a step function that, given a CA-array, performs a single transition step on it. As the CA is uniform, the same development rule is applied to each cell every development step. Since the state of the CA is stored in such a simple format, perturbing the system with input is as simple as replacing the array as a whole or modifying parts of it betweeen timesteps.

For the work presented in this paper, CAs 32 cells wide was used, with rule 90 (see Table II) as the transition function. To

TABLE I
$(DS)^2$ RC-MACHINE FGPA RESOURCE USAGE.

Cell Count	LUTs Total	LUTS %	Registers Total	Registers %	BRAMs total	BRAMs %
8×8	194624	15.93	208690	8.54	227.5	18.91
64×64	224442	18.37	221061	9.05	280.0	23.28
72×72	230094	18.84	223355	9.14	286.5	23.82
96×96	251219	20.56	231616	9.48	308.5	25.64
$4 \times 4 \times 4$	195360	15.99	209171	8.56	232.5	19.33
$16 \times 16 \times 16$	249320	20.41	218109	8.93	261	21.70

TABLE II
RULE 90

Neighborhood	111	110	101	100	011	010	001	000
New state	0	1	1	0	1	1	1	0

TABLE III
GA HYPERPARAMETERS

Population size	50
Crossover rate	0.5
Mutation rate	0.4
Crossover function	Braid
Mutation function	Per Genome

accurately model the FPGA implementation, only a subset of the cells are used as input to the readout layer.

B. SSNN Readout Module

The model described in III is implemented as an array of neurons for each layer in the desired network topology. Each neuron is represented as an array of counters and an array of weights, one weight and one counter for each neuron in the previous layer. As in the hardware implementation, networks are fully connected by default, allowing for connections to be pruned by setting a weight value of zero. The simulation also imitates the pipelined dataflow used in the FPGA implementation.

C. Genetic Algorithm

A genetic algorithm was used to search the space of possible weight configurations of the SSNN. The hyperparameters used are shown in Table III.

Each individual in the population represents one possible weight configuration for the topology being optimized. The genotype of each individual is simply a matrix of numbers, each one representing one weight. Crossover of two parent individuals is done by braiding their layers, so that the child has layer one from parent one, layer two from parent two, layer three from parent one, et cetera. Individuals are subject to random mutations performed per genome with some probability p. An individual chosen to be mutated, will have a randomly chosen weight changed, with equal probability of incrementing or decrementing the weight.

Fitness for an individual is calculated by comparing spiketrains output by the network when fed with test data, with the expected spiketrains for that data.

VI. EXPERIMENTS AND RESULTS

Using the software simulation setup described in Section V, we wish to show that the SSNN model is viable to use as a readout layer in an RC system. For this to be the case, it needs to be possible to train the network to produce distinct spiketrains given different sequences of readouts from the reservoir. In other words, given desired output spiketrains for different input perturbations to the reservoir, it needs to be possible to train the SSNN to correctly classify the temporal dynamics arising from the perturbations and to produce the corresponding output spiketrain.

To verify that this is possible, we encode the four different input cases of a two-bit XOR-operation, $00, 01, 10, 11$ as four distinct initial states for the CA and let the presence or absence of a spike from the SSNN output node at given timesteps after some number of initial number of timesteps indicate result of the XOR operation. Specifically, for each of the initial states, the CA/SSNN was allowed to run for 200 timesteps before the output from the SSNN was sampled ten times at 25 timestep intervals. In other words, an optimal weight configuration for the SSNN is one whose output samples are ten zeroes, ten ones, ten ones and ten zeroes for the four initial states respectively. No specific mapping scheme was used to map from XOR-inputs to their respective initial states. Rather at the start of each experiment, four random 32-wide bitstrings are chosen to represent the inputs.

For this experiment, the SSNN was configured with two layers, a two-neuron input-layer fully connected to 8 of the 32 cells in the CA, and an output layer with a single neuron. As described in Section V-C, a genetic algorithm (GA) is used to search through the space of possible weight configurations. Being an optimization algorithm, the GA does not guarantee that an optimal solution will be found. For several of the configurations run during this experiment, optimal solutions were however found, showing that SSNNs are viable as readout layers in RC-systems.

Depending on how large a subsection of the CA's cells' states are fed as input to the SSNN, the convergence rate for the search varies. By using fewer cells, the size of the search space is reduced, along with the amount of dynamic behavior captured, making it harder/potentially impossible to distinguish the dynamics emerging from one initial state from the others. Use too many, and the search space size increases, making it more likely that the search gets stuck in a local maximum. Optimal solutions were found most frequently when 8 of the total 32 cells in the CA were used as output.

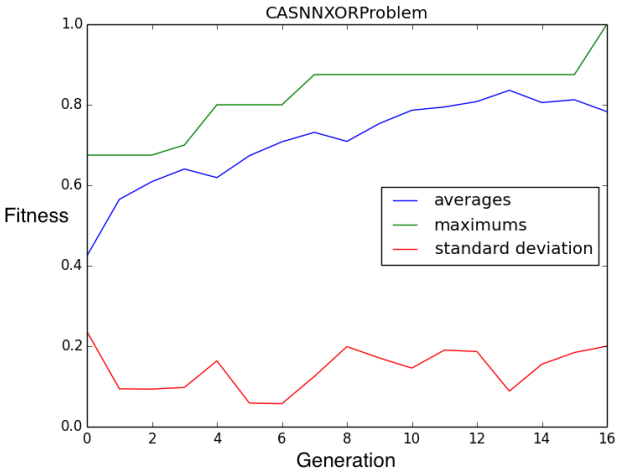

Fig. 9. Average, maximum and standard deviation of fitnesses for one run of the XOR experiment.

A plot of average and maximum fitnesses as well as fitness standard deviation for one run where an optimal solution was found is shown in Figure 9.

VII. Conclusion

In this paper, we present a Simplified Spiking Neural Network model, with the goal of using it as a readout layer in RC systems where the reservoir output is spiking in nature. Through experiments simulated in software, we show that it is possible to train a network based on this model to correctly classify temporal dynamics emerging in a cellular automaton. We also review work done towards an FPGA-implementation of a reservoir computing system consisting of a $(DS)^2$ cellular reservoir and a spiking neural network as a readout layer.

VIII. Future Work

While this paper shows that the SSNN is suitable for use in an RC system, using a genetic algorithm for training is less than desireable, and alternative training schemes should be investigated. Two possible alternatives are SpikeProp [30], a version of the backpropagation training algorithm adapted for spiking neurons, and spike-timing-dependent plasticity [31], an on-line training scheme where synapses over which spikes arrive right before the neuron fires are strengthened, while those over which spikes arrive right after thee neuron has fired are weakened.

The FPGA-based implementation of the SSNN is very simple so far, and needs more work to be practical in use. As mentioned in Section IV-C, weights are set statically at synthesis-time. To allow for easy reconfigurability, they should instead be stored in BRAM. As it is implemented now, the output from the SSNN is not stored anywhere, nor fed back into the reservoir. To facilitate analysis of the results, the output should be buffered for some time, allowing them to be transferred to the host program. It should also be possible to mark specific cells as input cells, into which the output from the SSNN is fed continuously. Since the reservoir is a $(DS)^2$ system, this will allow it to adapt according to the performance of its own dynamics.

Scalability is another issue with the current SSNN implementation. For a configuration with 4000 output cells in the reservoir and 64 neurons in the first layer of the SSNN, $4000 \times 64 = 256000$ registers will be inferred by the current implementation. To allow for flexible experimentation with reservoir sizes, number of output cells and SSNN topologies, the implementation should be changed so that it no longer generates one circuit per neuron per layer, but instead generates a limited number of neuron-circuits per layer. This way, computation in each layer happens over several clock cycles, but more FPGA-resource can be used by the reservoir.

Regarding the RC-machine described in Section IV, while work remains in order to make both the reservoir and readout components complete, much of the foundation is now in place. An interesting use case of the system is to use it to explore the space of growth rules for $(DS)^2$ systems. Rules that allow systems with interesting properties, such as the ability to self-reproduce or robustness, are of particular interest. These rules can potentially be used to guide the growth and development of reservoirs based on biological neurons, such as those used in the NTNU Cyborg project (see https://www.ntnu.edu/cyborg).

References

[1] M. Sipper, "Emergence of cellular computing," *Computer*, vol. 32, no. 7, pp. 18–26, 1999.

[2] J. F. Miller, S. L. Harding, and G. Tufte, "Evolution-in-materio: evolving computation in materials," *Evolutionary Intelligence*, vol. 7, no. 1, pp. 49–67, 2014.

[3] B. Schrauwen, D. Verstraeten, and J. Van Campenhout, "An overview of reservoir computing: theory, applications and implementations," *Proceedings of the 15th European Symposium on Artificial Neural Networks*, no. April, pp. 471–82, 2007.

[4] S. Kumar and P. Bentley, eds., *On Growth, Form and Computers*. Elsevier, 2003.

[5] G. Tufte and P. C. Haddow, "Towards development on a silicon-based cellular computing machine," *Natural Computing*, vol. 4, no. 4, pp. 387–416, 2005.

[6] S. Lawrence, C. L. Giles, S. Fong, S. Lawrence, and A. C. A. Tsoi, "Noisy Time Series Prediction using Recurrent Neural Networks and Grammatical Inference," *Machine Learning*, vol. 44, no. 1, pp. 161–183, 2001.

[7] F. A. Gers and J. Schmidhuber, "LSTM recurrent networks learn simple context-free and context-sensitive languages," *IEEE Transactions on Neural Networks*, vol. 12, no. 6, pp. 1333–1340, 2001.

[8] Z. Wu and S. King, "Investigating gated recurrent networks for speech synthesis," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, vol. 2016-May, pp. 5140–5144, 2016.

[9] B. Hammer and J. J. Steil, "Tutorial: Perspectives on learning with rnns," *Proc. ESANN*, no. April, pp. 357–368, 2002.

[10] H. Jaeger, "The "echo state" approach to analysing and training recurrent neural networks," *GMD Report*, vol. 148, pp. 1–47, 2001.

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

[12] J. J. Steil, "Backpropagation-Decorrelation: Online recurrent learning with O(N) complexity," in *IEEE International Conference on Neural Networks - Conference Proceedings*, vol. 2, pp. 843–848, 2004.

[13] D. Verstraeten, B. Schrauwen, M. D'Haene, and D. Stroobandt, "An experimental unification of reservoir computing methods," *Neural Networks*, vol. 20, pp. 391–403, apr 2007.

[14] D. Snyder, A. Goudarzi, and C. Teuscher, "Computational capabilities of random automata networks for reservoir computing," *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, vol. 87, no. 4, 2013.
[15] O. Yilmaz, "Reservoir Computing using Cellular Automata," *arXiv preprint*, pp. 1–9, 2014.
[16] C. Fernando and S. Sojakka, "Pattern Recognition in a Bucket," *Advances in Artificial Life*, pp. 588–597, 2003.
[17] J. V. Neumann, *Theory of Self-Reproducing Automata*. Champaign, IL, USA: University of Illinois Press, 1966.
[18] C. G. Langton, "Computation at the edge of chaos: Phase transitions and emergent computation," *Physica D: Nonlinear Phenomena*, vol. 42, no. 1-3, pp. 12–37, 1990.
[19] S. Wolfram, *Cellular automata and complexity: Collected papers*, vol. 45. 2003.
[20] J. H. Holland, *Adaptation in Natural and Artificial Systems*, vol. Ann Arbor. 1975.
[21] M. Mitchell, J. P. Crutchfield, and P. T. Hraber, "Evolving cellular automata to perform computations: mechanisms and impediments," *Physica D: Nonlinear Phenomena*, vol. 75, no. 1-3, pp. 361–391, 1994.
[22] G. Tufte, "The discrete dynamics of developmental systems," in *2009 IEEE Congress on Evolutionary Computation, CEC 2009*, pp. 2209–2216, 2009.
[23] G. Tufte and O. R. Lykkebø, "Evolution-in-Materio of a dynamical system with dynamical structures," *Proceedings of the Artificial Life Conference 2016*, 2016.
[24] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, 1943.
[25] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Networks*, vol. 10, no. 9, pp. 1659–1671, 1997.
[26] G. Pipa and R. Cao, "Extended Liquid Computing in Networks of Spiking Neurons Supervisor Spiking Neurons," 2010.
[27] G. Tufte, "From Evo to EvoDevo: Mapping and Adaptation in Artificial Development," in *Evolutionary Computation*, ch. 12, 2009.
[28] P. T. Lundal, "The Cellular Automata Research Platform: Revised, Rebuilt and Enhanced," *Master Thesis*, no. June, 2015.
[29] P. Haddow and G. Tufte, "An evolvable hardware FPGA for adaptive hardware," in *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No.00TH8512)*, vol. 1, pp. 553–560, 2000.
[30] S. M. Bohte, J. N. Kok, and H. La Poutré, "Error-backpropagation in temporally encoded networks of spiking neurons," 2002.
[31] H. Markram, W. Gerstner, and P. J. Sjöström, "Spike-timing-dependent plasticity: A comprehensive overview," *Frontiers in Synaptic Neuroscience*, vol. 4, no. JULY, pp. 2010–2012, 2012.