A general representation of dynamical systems for reservoir computing

Sidney Pontes-Filho[*,†,§], Anis Yazidi[*], Jianhua Zhang[*], Hugo Hammer[*],
Gustavo B. M. Mello[*], Ioanna Sandvig[‡], Gunnar Tufte[†] and Stefano Nichele[*]
[*]*Department of Computer Science, Oslo Metropolitan University, Oslo, Norway*
[†]*Department of Computer Science, Norwegian University of Science and Technology, Trondheim, Norway*
[‡]*Department of Neuromedicine and Movement Science, Norwegian University of Science and Technology, Trondheim, Norway*
Email: [§]sidneyp@oslomet.no

Abstract—**Dynamical systems are capable of performing computation in a reservoir computing paradigm. This paper presents a general representation of these systems as an artificial neural network (ANN). Initially, we implement the simplest dynamical system, a cellular automaton. The mathematical fundamentals behind an ANN are maintained, but the weights of the connections and the activation function are adjusted to work as an update rule in the context of cellular automata. The advantages of such implementation are its usage on specialized and optimized deep learning libraries, the capabilities to generalize it to other types of networks and the possibility to evolve cellular automata and other dynamical systems in terms of connectivity, update and learning rules. Our implementation of cellular automata constitutes an initial step towards a general framework for dynamical systems. It aims to evolve such systems to optimize their usage in reservoir computing and to model physical computing substrates.**

I. INTRODUCTION

A cellular automaton (CA) is the simplest computing system where the emergence of complex dynamics from local interactions might take place. It consists of a grid of cells with a finite number of states that change according to simple rules depending on the neighborhood and own state in discrete time-steps. Some notable examples are the elementary CA [1], which is unidimensional with three neighbors and eight update rules, and Conway's Game of Life [2], which is two-dimensional with nine neighbors and three update rules.

Table I presents some computing systems that are capable of giving rise to the emergence of complex dynamics. Those systems can be exploited by reservoir computing which is a paradigm that resorts to dynamical systems to simplify complex data. Hence, simpler and faster machine learning methods can be applied with such simplified data. Reservoir computing is more energy efficient than deep learning methods and it can even yield competitive results, especially for temporal data [3]. In short, reservoir computing exploits a dynamical system that possesses the echo state property and fading memory, where the internals of the reservoir are untrained and the only training happens at the linear readout stage [4]. Reservoir computers are most useful when the substrate's dynamics are at the "edge of chaos", meaning a range of dynamical behaviors that is between order and disorder [5]. Cellular automata with such dynamical behavior are capable of being exploited as

TABLE I
EXAMPLES OF DYNAMICAL SYSTEMS.

Dynamical system	State	Time	Connectivity
Cellular automata	Discrete	Discrete	Regular
Coupled map lattice	Continuous	Discrete	Regular
Random Boolean network	Discrete	Discrete	Random
Echo state network	Continuous	Discrete	Random
Liquid state machine	Discrete	Continuous	Random

reservoirs [6], [7]. Other systems can also exhibit the same dynamics. The coupled map lattice [8] is very similar to CA, the only exception is that the coupled map lattice has continuous states which are updated by a recurrence equation involving the neighborhood. Random Boolean network [9] is a generalization of CA where random connectivity exists. Echo state network [10] is an artificial neural network (ANN) with random topology while liquid state machine [11] is similar to echo state network with the difference that it is a spiking neural network that communicates through discrete-events (spikes) over continuous time. One important aspect of the computation performed in a dynamical system is the trajectory of system's states traversed during the computation [12]. Such trajectory may be guided by system parameters [13]. Computation in dynamical systems may be carried out in physical substrates [14], such as networks of biological neurons [15] or in other nanoscale materials [16]. Finding the correct abstraction for the computation in a dynamical system, e.g. CA, is an open problem [17]. All the systems described in Table I are sparsely connected and can be represented by an adjacency matrix, such as a graph. A fully connected feedforward ANN represents its connectivity from a layer to another with an adjacency matrix that contains the weights of each connection. Our CA implementation is similar to this, but the connectivity is from the "layer" of cells to itself.

The goal of representing CA with an adjacency matrix is to implement a framework which facilitates the development of all types of CAs, from unidimensional to multidimensional, with all kinds of lattices and without any boundary checks during execution; and also the inclusion of the major dynamical systems, independent of the type of the state, time and

connectivity. Such initial implementation is the first part of a Python framework under development, based on TensorFlow deep neural network library [18]. Therefore, it benefits from powerful and parallel computing systems with multi-CPU and multi-GPU. This framework, called EvoDynamic[1], aims at evolving the connectivity, update and learning rules of sparsely connected networks to improve their usage for reservoir computing guided by the echo state property, fading memory, state trajectory and other quality measurements, and to model the dynamics and behavior of physical reservoirs, such as *in-vitro* biological neural networks interfaced with microelectrode arrays and nanomagnetic ensembles. Those two substrates have real applicability as reservoirs. For example, the former substrate is applied to control a robot, in fact making it into a cyborg, a closed-loop biological-artificial neuro-system [15], and the latter possesses computation capability as shown by a square lattice of nanomagnets [19]. Those substrates are the main interest of the SOCRATES project [20] which aims to explore a dynamic, robust and energy efficient hardware for data analysis.

There are some implementations of CA similar to the one of EvoDynamic framework. They normally implement Conway's Game of Life by applying 2D convolution with a kernel that is used to count the neighbors, then the resulting matrix consists of the number of neighboring cells and is used to update the CA. One such implementation, also based on TensorFlow, is available open-source in [21].

This paper is organized as follows. Section II describes our method according to which we use adjacency matrix to compute CA. Section III presents the results obtained from the method. Section IV discusses the future plan of EvoDynamic framework and Section V concludes this paper.

II. METHOD

In our proposed method, the equation to calculate the next states of the cells in a cellular automaton is

$$\mathbf{ca}_{t+1} = f(\mathbf{A} \cdot \mathbf{ca}_t). \quad (1)$$

It is similar to the equation of the forward pass of an artificial neural network, but without the bias. The layer is connected to itself, and the activation function f defines the update rules of the CA. The next states of the CA $\mathbf{ca}_{t+1}$ is calculated from the result of the activation function f which receives as argument the dot product between the adjacency matrix $\mathbf{A}$ and the current states of the CA $\mathbf{ca}_t$. $\mathbf{ca}$ is always a column vector of size $len(\mathbf{ca}) \times 1$, that does not depend on how many dimensions the CA has, and $\mathbf{A}$ is a matrix of size $len(\mathbf{ca}) \times len(\mathbf{ca})$. Hence the result of $\mathbf{A} \cdot \mathbf{ca}$ is also a column vector of size $len(\mathbf{ca}) \times 1$ as $\mathbf{ca}$.

The implementation of cellular automata as an artificial neural network requires the procedural generation of the adjacency matrix of the grid. In this way, any lattice type or multidimensional CAs can be implemented using the same approach.

The adjacency matrix of a sparsely connected network contains many zeros because of the small number of connections. Since we implement it on TensorFlow, the data type of the adjacency matrix is preferably a `SparseTensor`. A dot product with this data type can be up to 9 times faster depending on the configuration of the tensors [22]. The update rule of the CA alters the weights of the connections in the adjacency matrix. In a CA whose cells have two states meaning "dead" (zero) or "alive" (one), the weights in the adjacency matrix are one for connection and zero for no connection, such as an ordinary adjacency matrix. Such matrix facilitates the description of the update rule for counting the number of "alive" neighbors because the result of the dot product between the adjacency matrix and the cell state vector is the vector that contains the number of "alive" neighbors for each cell. If the pattern of the neighborhood matters in the update rule, each cell has its neighbors encoded as a n-ary string where n means the number of states that a cell can have. In this case the weights of the connections with the neighbors are n-base identifiers and are calculated by

$$neighbor_i = n^i, \forall i \in \{0..len(\mathbf{neighbors}) - 1\}. \quad (2)$$

Where **neighbors** is a vector of the cell's neighbors. In the adjacency matrix, each neighbor receives a weight according to (2). The result of the dot product with such adjacency matrix is a vector that consists of unique integers per neighborhood pattern. Thus, the activation function is a lookup table from integer (i.e., pattern) to next state.

Algorithm 1 generates the adjacency matrix for one-dimensional CA, such as the elementary CA. Where $widthCA$ is the width or number of cells of a unidimensional CA and **neighborhood** is a vector which describes the region around the center cell. The connection weights depend on the type of update rule as previously explained. For example, in case of an elementary CA **neighborhood** = [4 2 1]. $indexNeighborCenter$ is the index of the center cell in the **neighborhood** whose starting index is zero. $isWrappedGrid$ is a Boolean value that works as a flag for adding wrapped grid or not. A wrapped grid for one-dimensional CA means that the initial and final cells are neighbors. With all these parameters, Algorithm 1 creates an adjacency matrix by looping over the indices of the cells (from zero to $numberOfCells - 1$) with an inner loop for the indices of the neighbors. If the selected $currentNeighbor$ is a non-zero value and its indices do not affect the boundary condition, then the value of $currentNeighbor$ is assigned to the adjacency matrix $\mathbf{A}$ in the indices that correspond to the connection between the current cell in the outer loop and the actual index of $currentNeighbor$. Finally, this procedure returns the adjacency matrix $\mathbf{A}$.

To procedurally generate an adjacency matrix for 2D CA instead of 1D CA, the algorithm needs to have small adjustments. Algorithm 2 shows that for two-dimensional CA, such as Conway's Game of Life. In this case, the height of the CA is an argument passed as $heightCA$. **Neighborhood**

Algorithm 1 Generation of adjacency matrix for 1D cellular automaton

1: **procedure** GENERATECA1D($widthCA$, **neighborhood**, $indexNeighborCenter$, $isWrappedGrid$)
2:     $numberOfCells \leftarrow widthCA$
3:     $\mathbf{A} \leftarrow \mathbf{0}^{numberOfCells \times numberOfCells}$ ▷ Adjacency matrix initialization
4:     **for** $i \leftarrow \{0..numberOfCells - 1\}$ **do**
5:         **for** $j \leftarrow \{-indexNeighborCenter..len(neighborhood) - indexNeighborCenter - 1\}$ **do**
6:             $currentNeighbor \leftarrow neighborhood_{j+indexNeighborCenter}$
7:             **if** $currentNeighbor \neq 0 \land (isWrappedGrid \lor (\neg isWrappedGrid \land (0 \leq (i + j) < widthCA))$ **then**
8:                 $\mathbf{A}_{i,((i+j) \bmod widthCA)} \leftarrow currentNeighbor$
9:     **return A**

is a 2D matrix and **indexNeighborCenter** is a vector of two components meaning the indices of the center of **Neighborhood**. This procedure is similar to the one in Algorithm 1, but it contains one more loop for the additional dimension.

The activation function for CA is different from the ones used for ANN. For CA, it contains the update rules that verify the vector returned by the dot product between the adjacency matrix and the vector of states. Normally, the update rules of the CA are implemented as a lookup table from neighborhood to next state. In our implementation, the lookup table maps the resulting vector of the dot product to the next state of the central cell.

III. RESULTS

This section presents the results of the proposed method and it also stands for the preliminary results of the EvoDynamic framework.

Fig. 1 illustrates a wrapped elementary CA described in the procedure of Algorithm 1 and its generated adjacency matrix. Fig. 1a shows the appearance of the desired elementary CA with 16 cells (i.e., $widthCA = 16$). Fig. 1b describes its pattern 3-neighborhood and the indices of the cells. Fig 1c shows the result of the Algorithm 1 with the neighborhood calculated by (2) for pattern matching in the activation function. In Fig. 1c, we can verify that the left neighbor has weight equals to 4 (or 2^2 for the most significant bit), central cell weight is 2 (or 2^1) and right neighbor weight is 1 (or 2^0 for the least significant bit) as defined by (2). Since the CA is wrapped, we can notice in row index 0 of the adjacency matrix in Fig. 1c that the left neighbor of cell 0 is the cell 15, and in row index 15 that the right neighbor of cell 15 is the cell 0.

Fig. 2 describes a wrapped 2D CA for Algorithm 2 and shows the resulting adjacency matrix. Fig. 2a illustrates the desired two-dimensional CA with 16 cells (i.e., $widthCA = 4$ and $heightCA = 4$). Fig. 2b presents the von Neumann neighborhood [23] which is used for counting the number of "alive" neighbors (the connection weights are only zero and one, and **Neighborhood** argument of Algorithm 2 defines it). It also shows the index distribution of the CA whose order is preserved after flatting it to a column vector. Fig 2c contains the generated adjacency matrix of Algorithm 2 for the described 2D CA. Fig. 2b shows an example of a central
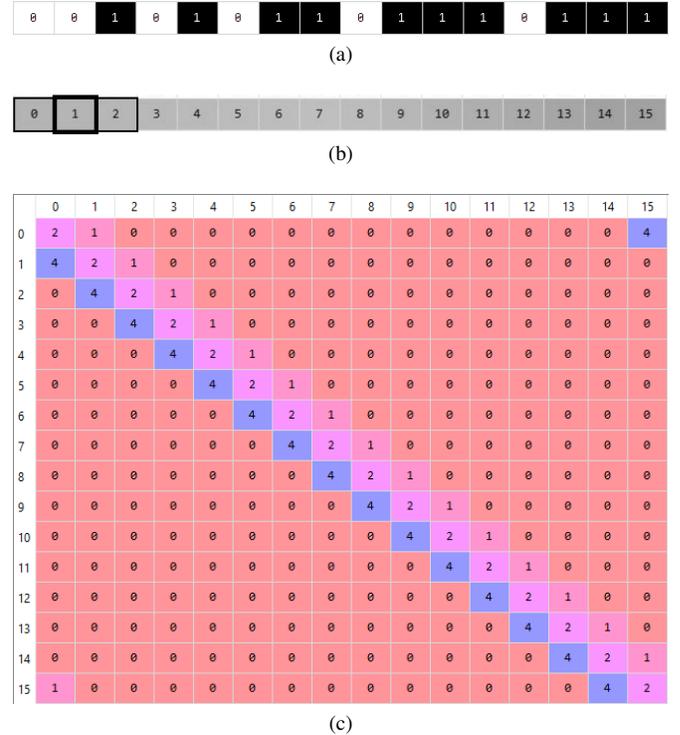
Fig. 1. Elementary cellular automaton with 16 cells and wrapped grid. (a) Example of the grid of cells with states. (b) Indices of the cells and standard pattern neighborhood of elementary CA where thick border means the central cell and thin border means the neighbors. (c) Generated adjacency matrix for this elementary CA.

cell with its neighbors, the index of this central cell is 5 and the row index 5 in the adjacency matrix of Fig. 2c presents the same neighbor indices, i.e., 1, 4, 6 and 9. Since this is a symmetric matrix, the columns have the same connectivity of the rows. Therefore, this adjacency matrix represents an undirected graph. The wrapping effect is also observable. For example, the neighbors of the cell index 0 are 1, 3, 4 and 12. So the neighbors 3 and 12 are the ones that the wrapped grid allowed to exist for cell index 0.

IV. EvoDynamic FUTURE

The method of implementing a CA as an artificial neural network will be beneficial for the future of EvoDynamic framework. Since the implementation of all sparsely connected networks in Table I are already planned in future releases

---

Algorithm 2 Generation of adjacency matrix of 2D cellular automaton

---
1: **procedure** GENERATECA2D($widthCA, heightCA, \textbf{Neighborhood}, \textbf{indexNeighborCenter}, isWrappedGrid$)
2:     $numberOfCells \leftarrow widthCA * heightCA$
3:     $\textbf{A} \leftarrow \textbf{0}^{numberOfCells \times numberOfCells}$ ▷ Adjacency matrix initialization
4:     $widthNB, heightNB \leftarrow shape(\textbf{Neighborhood})$
5:     **for** $i \leftarrow \{0..numberOfCells - 1\}$ **do**
6:         **for** $j \leftarrow \{-\textbf{indexNeighborCenter}_0..widthNB - \textbf{indexNeighborCenter}_0 - 1\}$ **do**
7:             **for** $k \leftarrow \{-\textbf{indexNeighborCenter}_1..heightNB - \textbf{indexNeighborCenter}_1 - 1\}$ **do**
8:                 $currentNeighbor \leftarrow Neighborhood_{j+indexNeighborCenter}$
9:                 **if** $currentNeighbor \neq 0 \wedge (isWrappedGrid \vee (\neg isWrappedGrid \wedge (0 \leq ((i \bmod heightCA) + j) <$
$widthCA) \wedge (0 \leq (\lfloor i/widthCA \rfloor + k) < heightCA))$ **then**
10:                     $\textbf{A}_{i,(((i+k) \bmod widthCA)+((\lfloor i/widthCA \rfloor+j) \bmod heightCA)*widthCA)} \leftarrow currentNeighbor$
11:     **return A**

(a)   (b)

(c)

Fig. 2. 2D cellular automaton with 16 cells (4×4) and wrapped grid. (a) Example of the grid of cells with states. (b) Indices of the cells and von Neumann counting neighborhood of 2D CA where thick border means the current cell and thin border means the neighbors. (c) Generated adjacency matrix for this 2D CA.

of the Python framework, EvoDynamic must have a general representation to all of them. Therefore we are treating CA as an ANN. Moreover, EvoDynamic framework will evolve the connectivity, update and learning rules of the dynamical systems for reservoir computing improvement and physical substrate modeling. This common representation facilitates the evolution of such systems and models which will be guided by several methods that measure the quality of a reservoir or the similarity to a dataset. One example of these methods is the state trajectory. For visualization, we use principal component analysis (PCA) to reduce the dimensionality of the states and present them as a state transition diagram as shown in Fig. 3.

V. CONCLUSION

In this paper, we present an alternative method to implement a cellular automaton. This allows any CA to be computed as an artificial neural network. Therefore, this will help to extend the CA implementation to more complex dynamical systems, such as echo state networks and liquid state machines. Furthermore, the EvoDynamic framework is built on a deep learning library, TensorFlow, which permits the acceleration of the execution when applied on parallel computational platforms with fast CPUs and GPUs. The future work for this CA implementation is to develop algorithms to procedurally generate adjacency matrices for 3D and multidimensional cellular automata with different types of cells, such as the cells with hexagonal shape.

REFERENCES

[1] S. Wolfram, *A new kind of science*. Wolfram media Champaign, IL, 2002, vol. 5.

[2] P. Rendell, *Turing Universality of the Game of Life*. London: Springer London, 2002, pp. 513–539. [Online]. Available: https://doi.org/10.1007/978-1-4471-0129-1_18

[3] B. Schrauwen, D. Verstraeten, and J. Van Campenhout, "An overview of reservoir computing: theory, applications and implementations," in *Proceedings of the 15th European Symposium on Artificial Neural Networks. p. 471-482 2007*, 2007, pp. 471–482.

[4] Z. Konkoli, S. Nichele, M. Dale, and S. Stepney, *Reservoir Computing with Computational Matter*. Cham: Springer International Publishing, 2018, pp. 269–293. [Online]. Available: https://doi.org/10.1007/978-3-319-65826-1_14

[5] C. G. Langton, "Computation at the edge of chaos: Phase transitions and emergent computation," *Physica D: Nonlinear Phenomena*, vol. 42, no. 1, pp. 12 – 37, 1990. [Online]. Available: http://www.sciencedirect.com/science/article/pii/016727899090064V

[6] S. Nichele and M. S. Gundersen, "Reservoir computing using nonuniform binary cellular automata," *Complex Systems*, vol. 26, no. 3, pp. 225–245, Sep. 2017. [Online]. Available: https://doi.org/10.25088/complexsystems.26.3.225

[7] S. Nichele and A. Molund, "Deep learning with cellular automaton-based reservoir computing," *Complex Systems*, vol. 26, no. 4, pp. 319–339, Dec. 2017. [Online]. Available: https://doi.org/10.25088/complexsystems.26.4.319

[8] K. Kaneko, "Overview of coupled map lattices," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 2, no. 3, pp. 279–282, 1992.
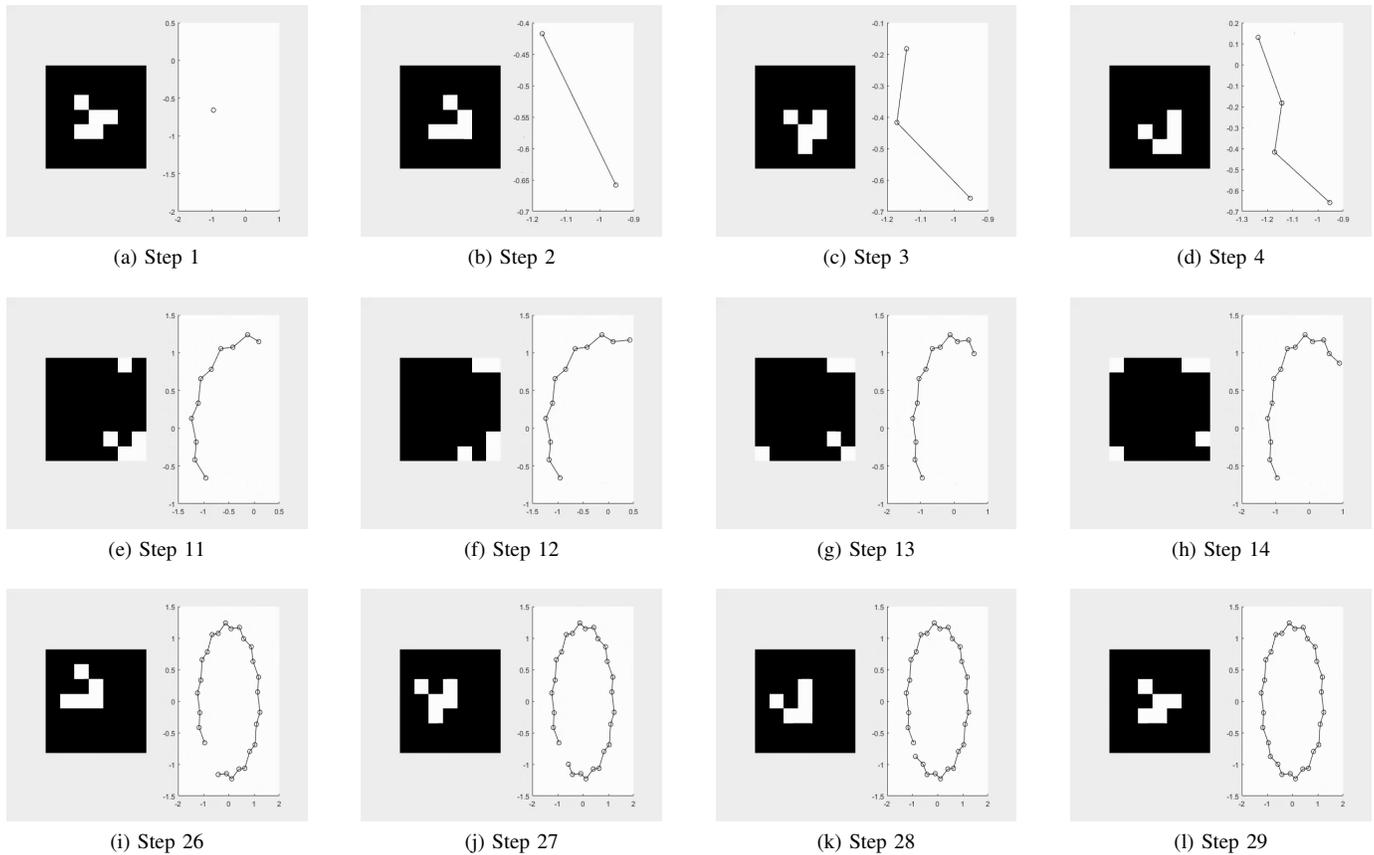
Fig. 3. States of Conway's Game of Life in a 7x7 wrapped lattice alongside their PCA-transformed state transition diagrams of the two first principal components. (a) Initial state is a glider. (a)-(d) Four first steps in this CA. (e)-(h) Four intermediate steps in this CA while reaching the wrapped border. (i)-(l) Four last steps in this CA before repeating the initial state and closing a cycle.

[9] C. Gershenson, "Introduction to random boolean networks," *arXiv preprint nlin/0408006*, 2004.

[10] H. Jaeger and H. Haas, "Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication," *Science*, vol. 304, no. 5667, pp. 78–80, 2004. [Online]. Available: https://science.sciencemag.org/content/304/5667/78

[11] W. Maass and H. Markram, "On the computational power of circuits of spiking neurons," *Journal of Computer and System Sciences*, vol. 69, no. 4, pp. 593 – 616, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022000004000406

[12] S. Nichele and G. Tufte, "Trajectories and attractors as specification for the evolution of behaviour in cellular automata," in *IEEE Congress on Evolutionary Computation*, July 2010, pp. 1–8.

[13] S. Nichele and G. Tufte, "Genome parameters as information to forecast emergent developmental behaviors," in *Unconventional Computation and Natural Computation*, J. Durand-Lose and N. Jonoska, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 186–197.

[14] G. Tanaka, T. Yamane, J. B. Hroux, R. Nakane, N. Kanazawa, S. Takeda, H. Numata, D. Nakano, and A. Hirose, "Recent advances in physical reservoir computing: A review," *Neural Networks*, vol. 115, pp. 100 – 123, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608019300784

[15] P. Aaser, M. Knudsen, O. H. Ramstad, R. van de Wijdeven, S. Nichele, I. Sandvig, G. Tufte, U. Stefan Bauer, . Halaas, S. Hendseth, A. Sandvig, and V. Valderhaug, "Towards making a cyborg: A closed-loop reservoir-neuro system," *The 2018 Conference on Artificial Life: A Hybrid of the European Conference on Artificial Life (ECAL) and the International Conference on the Synthesis and Simulation of Living Systems (ALIFE)*, no. 29, pp. 430–437, 2017. [Online]. Available: https://www.mitpressjournals.org/doi/abs/10.1162/isal_a_072

[16] H. Broersma, J. F. Miller, and S. Nichele, *Computational Matter: Evolving Computational Functions in Nanoscale Materials*. Cham: Springer International Publishing, 2017, pp. 397–428. [Online]. Available: https://doi.org/10.1007/978-3-319-33921-4_16

[17] S. Nichele, S. S. Farstad, and G. Tufte, "Universality of evolved cellular automata in-materio." *International Journal of Unconventional Computing*, vol. 13, no. 1, 2017.

[18] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[19] J. H. Jensen, E. Folven, and G. Tufte, "Computation in artificial spin ice," *The 2018 Conference on Artificial Life: A Hybrid of the European Conference on Artificial Life (ECAL) and the International Conference on the Synthesis and Simulation of Living Systems (ALIFE)*, no. 30, pp. 15–22, 2018. [Online]. Available: https://www.mitpressjournals.org/doi/abs/10.1162/isal_a_00011

[20] SOCRATES Self-Organizing Computational substRATES. [Online]. Available: https://www.ntnu.edu/socrates

[21] "Conway's game of life implemented using tensorflow 2d convolution function," 2016. [Online]. Available: https://github.com/conceptacid/conv2d_life

[22] TensorFlow, "tf.sparse.sparse_dense_matmul — tensorflow core r1.14 — tensorflow." [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/sparse/sparse_dense_matmul

[23] T. Toffoli and N. Margolus, *Cellular automata machines: a new environment for modeling*. MIT press, 1987.