EvoDynamic: a framework for the evolution of generally represented dynamical systems and its application to criticality

Sidney Pontes-Filho[1,2][0000−0002−0489−5652], Pedro Lind[1], Anis Yazidi[1], Jianhua Zhang[1], Hugo Hammer[1], Gustavo B. M. Mello[1], Ioanna Sandvig[3], Gunnar Tufte[2], and Stefano Nichele[1,4]

[1] Department of Computer Science, Oslo Metropolitan University, Oslo, Norway
[2] Department of Computer Science, Norwegian University of Science and Technology, Trondheim, Norway
[3] Department of Neuromedicine and Movement Science, Norwegian University of Science and Technology, Trondheim, Norway
[4] Holistic Systems, SimulaMet, Oslo, Norway
Email: sidneyp@oslomet.no

Abstract. Dynamical systems possess a computational capacity that may be exploited in a reservoir computing paradigm. This paper presents a general representation of dynamical systems which is based on matrix multiplication. That is similar to how an artificial neural network (ANN) would be represented in a deep learning library and its computation would be faster because of the optimized matrix operations that such type of libraries have. 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 more general framework for dynamical systems. Our objective is to evolve such systems to optimize their usage in reservoir computing and to model physical computing substrates. Furthermore, we present promising preliminary results toward the evolution of complex behavior and criticality using genetic algorithm in stochastic elementary cellular automata.

Keywords: Cellular automata · Dynamical systems · Implementation · Reservoir computing · Evolution · Criticality

1 Introduction

A cellular automaton (CA) is the simplest computing system where the emergence of complex dynamics from local interactions might take place. It consists

This is a post-peer-review, pre-copyedit version of a conference proceeding published in Applications of Evolutionary Computation 23rd European Conference, EvoApplications 2020, Held as Part of EvoStar 2020, Proceedings, that is part of the Lecture Notes in Computer Science book series (volume 12104). The final authenticated version is available online at: https://doi.org/10.1007/978-3-030-43722-0_9

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 [30], which is unidimensional with three neighbors and eight update cases, and Conway's Game of Life [24], which is two-dimensional with nine neighbors and three update cases.

Table 1 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. Such simplification means that reservoir computing utilizes the non-linear dynamical system to perform a non-linear transformation from non-linear data to higher dimensional linear data. Such linearized data can be applied in linear machine learning methods which are faster for training and computing because has less trainable variables and operations. Hence, reservoir computing is more energy efficient than deep learning methods and it can even yield competitive results, especially for temporal data [25,27]. Basically, 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 [16].

Reservoir computers are most useful when the substrate's dynamics are at the "edge of chaos" [17], meaning a range of dynamical behaviors that is between order and disorder. Cellular automata with such dynamical behavior are capable of being exploited as reservoirs [21,22]. Other systems can also exhibit similar dynamics. The coupled map lattice [15] 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 [10] is a generalization of CA where random connectivity exists. Echo state network [13] is an artificial neural network (ANN) with random topology while liquid state machine [18] 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 states traversed during the computation [19]. Such trajectory may be guided by system parameters [23]. Another characteristic of a dynamical system, which is crucial for computation, is to be in a critical state,

Table 1: 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

as indicated by Langton [17]. If the attractors of the system are in the critical state, this characteristic is called self-organized criticality [7].

Besides, computation in dynamical systems may be carried out in physical substrates [27], such as networks of biological neurons [3] or in nanoscale materials [8]. Finding the correct abstraction for the computation in a dynamical system, e.g. CA, is an open problem [20].

All the systems described in Table 1 are sparsely connected and can be represented by a weighted adjacency matrix, such as a graph. The connectivity from a layer to another in a fully connected feedforward ANN is represented with a weighted adjacency matrix that contains the weights of each connection. Our CA implementation is similar to this, but the connectivity goes from the "layer" of cells to itself.

The goal of representing CA with a weighted 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 conditions during execution; and also allowing the inclusion of other major dynamical systems, independent of the type of the state, time and connectivity. Such initial implementation is the first component of a Python framework under development, based on TensorFlow deep neural network library [4]. Therefore, it benefits from powerful and parallel computing systems with multi-CPU and multi-GPU. One of the framework's goals is to have a balance between performance and generalization of computing dynamical systems, since general methods are slower than specialized ones. Nevertheless, this framework, called EvoDynamic, aims at evolving (i.e., using evolutionary algorithms) 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. Such improvement of reservoirs is applied similarly in [26], where the internal connectivity of a reservoir is trained to increase its performance to several tasks. Moreover, evolution will 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 [3], and the latter possesses computation capability as shown by a square lattice of nanomagnets [14]. Those substrates are the main interest of the SOCRATES project [1] which aims to explore a dynamic, robust and energy efficient hardware for data analysis.

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

This paper is organized as follows. Section 2 describes our method according to which we use weighted adjacency matrix to compute CA. Section 3 presents

the results obtained from the method. Section 4 discusses the initial advances and future plan of EvoDynamic framework and Section 5 concludes this paper.

2 Method

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

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

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 $\mathbf{c}_{t+1}$ is calculated from the result of the activation function f which receives as argument the dot product between the weighted adjacency matrix $\mathbf{A}$ and the current states of the CA $\mathbf{c}_t$. $\mathbf{c}$ is always a column vector of size $len(\mathbf{c}) \times 1$, that does not depend on how many dimensions the CA has, and $\mathbf{A}$ is a matrix of size $len(\mathbf{c}) \times len(\mathbf{c})$. Hence the result of $\mathbf{A} \cdot \mathbf{c}$ is also a column vector of size $len(\mathbf{c}) \times 1$ as $\mathbf{c}$.

The implementation of cellular automata as an artificial neural network requires the procedural generation of the weighted 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 than the dense counterpart. However, it depends on the configuration of the tensors (or, in our case, the adjacency matrices) [28]. 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

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

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 weighted 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 weighted adjacency matrix for one-dimensional CA, such as the elementary CA, where $widthCA$ is the width or number of

Algorithm 1 Generation of weighted adjacency matrix for 1D cellular automaton

```
1: procedure GENERATECA1D
2:     numberOfCells ← widthCA
3:     A ← 0^(numberOfCells×numberOfCells)          ▷ Adjacency matrix initialization
4:     for i ← {0..(numberOfCells − 1)} do
5:         for j ← {−indexNeighborCenter..(len(neighborhood) − indexNeighborCenter − 1)} do
6:             currentNeighbor ← neighborhood_(j+indexNeighborCenter)
7:             if currentNeighbor ≠ 0 ∧ (isWrappedGrid ∨ (¬isWrappedGrid ∧ (0 ≤ (i + j) < widthCA)) then
8:                 A_(i,((i+j) mod widthCA)) ← currentNeighbor
9:     return A
```

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 a 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** 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 weighted 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.

Algorithm 2 Generation of adjacency matrix of 2D cellular automaton

procedure GENERATECA2D
  $numberOfCells \leftarrow widthCA * heightCA$
  $\mathbf{A} \leftarrow \mathbf{0}^{numberOfCells \times numberOfCells}$ ▷ Adjacency matrix initialization
  $widthNB, heightNB \leftarrow shape(\mathbf{Neighborhood})$
  for $i \leftarrow \{0..(numberOfCells - 1)\}$ **do**
    for $j \leftarrow \{-\mathbf{indexNeighborCenter}_0..(widthNB - \mathbf{indexNeighborCenter}_0 - 1)\}$ **do**
      for $k \leftarrow \{-\mathbf{indexNeighborCenter}_1..(heightNB - \mathbf{indexNeighborCenter}_1 - 1)\}$ **do**
        $currentNeighbor \leftarrow Neighborhood_{j+indexNeighborCenter}$
        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**
          $\mathbf{A}_{i,(((i+k) \bmod widthCA)+((\lfloor i/widthCA \rfloor + j) \bmod heightCA)*widthCA)} \leftarrow currentNeighbor$
  return A

3 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 weighted 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 equal 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 (similar to Game of Life but with less number of neighbors) 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 [29] 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 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. That means the neighborhood of a cell considers this cell

as a neighbor too. Therefore, the connections are bidirectional and the 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.

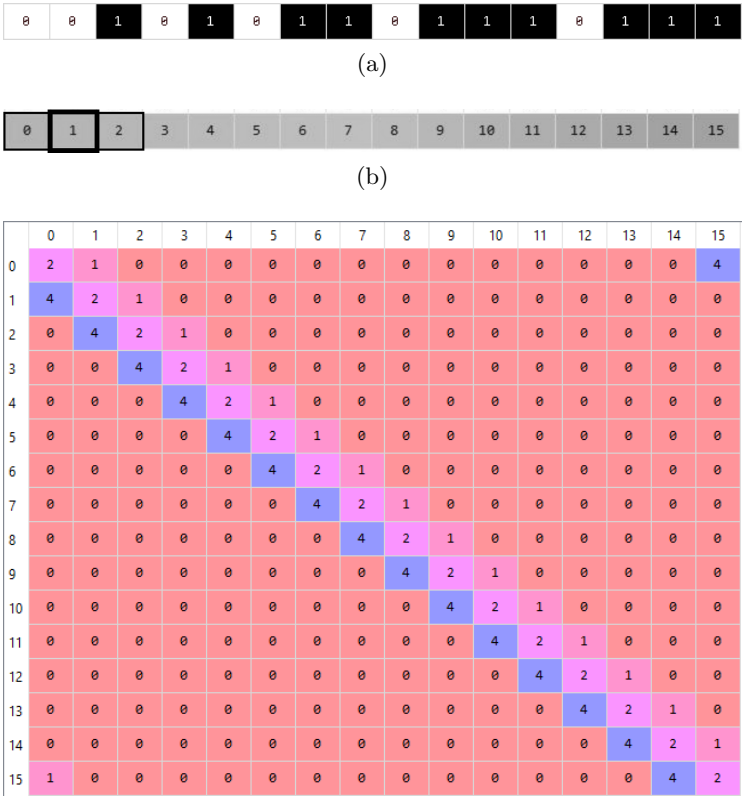
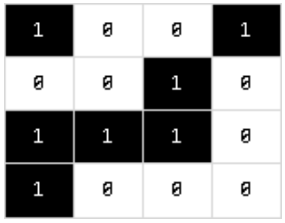


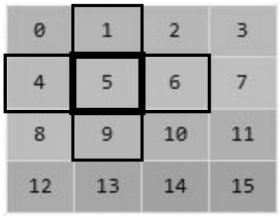
Fig. 1: Elementary cellular automaton with 16 cells and wrapped grid. (a) Example of the grid of cells with states. (b) Indices of the cells and standard pattern neighborhood of elementary CA where thick border means the central cell and thin border means the neighbors. (c) Generated weighted adjacency matrix for the described elementary CA.

4 On-going and future applications with EvoDynamic

The method of implementing a CA as an artificial neural network is beneficial for the further development of EvoDynamic framework. Since the implementation of



(a)



(b)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	0	1	1	0	0	0	0	0	0	0	1	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0
2	0	1	0	1	0	0	1	0	0	0	0	0	0	0	1	0
3	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	1
4	1	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0
5	0	1	0	0	1	0	1	0	0	1	0	0	0	0	0	0
6	0	0	1	0	0	1	0	1	0	0	1	0	0	0	0	0
7	0	0	0	1	1	0	1	0	0	0	0	1	0	0	0	0
8	0	0	0	0	1	0	0	0	0	1	0	1	1	0	0	0
9	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	0
10	0	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0
11	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0	1
12	1	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1
13	0	1	0	0	0	0	0	0	0	1	0	0	1	0	1	0
14	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	1
15	0	0	0	1	0	0	0	0	0	0	0	1	1	0	1	0

(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 the described 2D CA.

all sparsely connected networks in Table 1 is already planned in forthcoming releases of the Python framework, EvoDynamic shall have a general representation to all of them. Therefore, CAs are treated as ANNs and then can be extended to random Boolean network by shuffling the connections, and to the models that are already ANNs, such as echo state networks and liquid state machines. 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. The following subsections explain two on-going applications with CA that use the EvoDynamic framework.

4.1 State trajectory

An example of methods to guide the evolution of dynamical system is the state trajectory. This method can be used to cluster similar states for model abstraction and to measure the quality of the reservoir. Therefore, a graph can be formed and analysis can be made by searching for attractors and cycles. For visualization of the state trajectory, 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. The depicted dynamical system is Conway's Game of Life with 7x7 cells and wrapped boundaries. A glider is its initial state (Fig. 3a) and this system cycles over 28 unique states as illustrated in the state transition diagram of Fig. 3l.

4.2 Towards the evolution for criticality

Evolution of dynamical systems is a feature currently under development of EvoDynamic framework. The first on-going evolution task of our framework is to find systems with criticality [7] using genetic algorithm, in order to allow for better computational capacity [17]. The first dynamical system for this task is a modified version of stochastic elementary cellular automata (SECA) introduced by Baetens et al. [6]. Our stochastic elementary cellular automaton works as a 1D three neighbors elementary CA, but the next state in time $t+1$ of the central cell c_i is defined by a probability p to be 1 and a probability $1-p$ to be 0 for each of the eight different neighborhood patterns this CA has. Formally, probability p is represented by

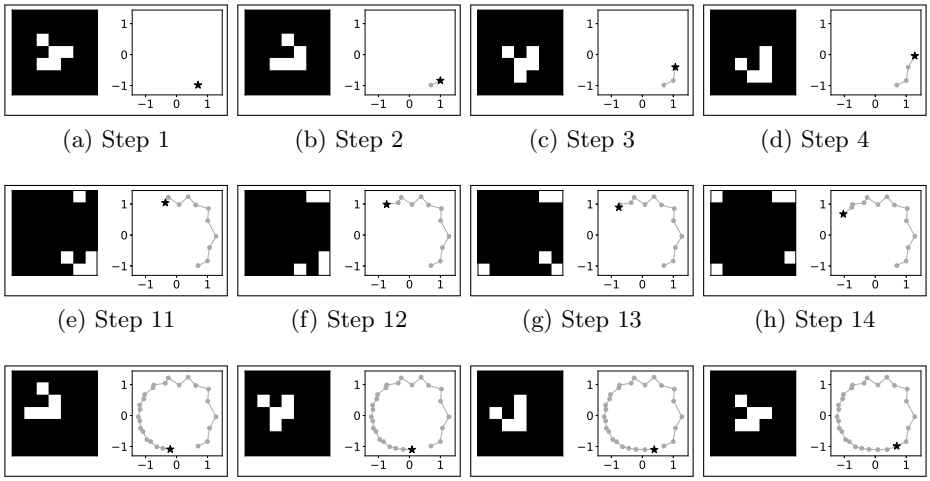
$$p = P(c_{i,t+1} = 1 | N(c_{i,t})) \tag{3}$$

where the neighborhood pattern $N(c_{i,t})$ is denoted as

$$N(c_{i,t}) = (c_{i-1,t}, c_{i,t}, c_{i+1,t}). \tag{4}$$

The genetic algorithm for criticality is guided by a fitness function which mainly verifies if the probability distributions of avalanche size (i.e., cluster size[5]

[5] Cluster size stands for the number of repetitions of a state that happened consecutively without any interruption of another state.



(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4

(e) Step 11 (f) Step 12 (g) Step 13 (h) Step 14

(i) Step 26 (j) Step 27 (k) Step 28 (l) Step 29

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.

in space) and duration (i.e., cluster size in time) follow a power-law distribution. Such verification can be done by checking how linear is the probability distribution in a log-log plot, by performing goodness-of-fit tests based on the Kolmogorov-Smirnov (KS) statistic and by comparing the power-law model with the exponential model using log-likelihood ratio [9]. For our fitness function, we estimate the candidate distributions with the linear fitting of the first 10 points of the log-log plot using least squares regression, which was verified to be not biased and gives a fast and acceptable estimation of the slope of the power-law distribution [12]. After the linear 10-points fitting, the model is tested using KS statistic. One benefit of using such estimation method is that when the model is not a power-law, the KS statistic reports a large error, i.e., an error greater than one. Another objective in the fitness function is the coefficient of determination [31], but for a complete linear fit of the log-log plot. The fitness function also considers the number of unique states of the stochastic elementary CA, the number of bins in the raw histogram and the value of the estimated power-law exponent. All these fitness function objectives are calculated using a randomly initialized CA of 1,000 cells with wrapped boundaries during 1,000 time-steps. The avalanche size and duration are computed for the cell values 0 and 1, thus producing four different distributions (see Fig. 4) for extracting vectors of their

normalized number of histogram bins[6] bin; coefficient of determination R^2 of complete linear fitting; KS statistic D and estimated power-law exponent $\hat{\alpha}$ from the 10-points linear estimation. The fitness score s for each objective is then calculated by the following equations:

$$bin_s = \tanh(5 * (0.9 * \max(bin) + 0.1 * \text{mean}(bin))), \tag{5}$$

$$R_s^2 = \text{mean}(R^2), \tag{6}$$

$$D_s = \exp(-(0.9 * \min(D) + 0.1 * \text{mean}(D))), \tag{7}$$

$$\hat{\alpha}_s = \text{mean}(\hat{\alpha}), \tag{8}$$

$$unique_s = \frac{\#uniqueStates}{\#timesteps}. \tag{9}$$

The (5)-(9) are all objective values for calculating the fitness score s. Those values are real numbers between zero and one, except the score for the estimated power-law exponent $\hat{\alpha}_s$, and they have weights attributed to them regarding their level of importance and for compensating small and large values. The following equation denotes how the fitness score s is calculated:

$$s = 10 * bin_s + 10 * R_s^2 + 10 * D_s + 0.1 * \hat{\alpha}_s + 10 * unique_s. \tag{10}$$

The genetic algorithm has 40 individuals that evolve through 100 generations. The optimization performed by GA is to maximize the fitness score. The genome of the individuals has eight real number genes with a value range between zero and one. Each gene represents the probability of the next state becoming one (i.e., p in (3)) for its respective neighborhood pattern. The selection of two parents is done by deterministic tournament selection [11]. After that, the crossover between the genomes of the parents can happen with probability 0.8, then each gene can be exchanged with probability 0.5. Afterward, a mutation occurs to a gene with probability 0.1. This mutation adds a random value from a normal distribution with mean and standard deviation equals to, respectively, 0 and 0.2. The mating process of the two parents produces an offspring of two new individuals who replace the parents in the next generation. An example of an evolved genome for the best resulting individual is presented in Table 2. The fitness score s and all objective scores with their respective weights for calculating s are in Table 3.

With the genome or probabilities of the eight different neighborhood patterns of the best evolved individual, we can produce the log-log plots of the probability distribution of avalanche size and duration for the states zero and one. Such plots

[6] The actual number of histogram bins is normalized or divided by the possible total number of bins.

Table 2: Best individual

Neighborhood $N(c_{i,t})$	**Probability** p
(0,0,0)	0.103009
(0,0,1)	0.536786
(0,1,0)	0.216794
(0,1,1)	0.393468
(1,0,0)	0.679836
(1,0,1)	0.175458
(1,1,0)	0.724778
(1,1,1)	1.000000

Table 3: Fitness score of the best individual

Objective	**Score**
$10 * bin_s$	9.780749590096136
$10 * R_s^2$	8.832520186440096
$10 * D_s$	9.655719560019996
$0.1 * \hat{\alpha}_s$	0.18022617747972156
$10 * unique_s$	10.0
s	**38.44921551403595**

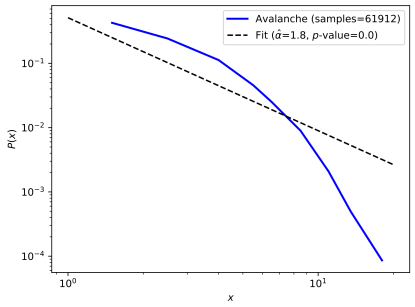
are depicted in Fig. 4. The p-value of goodness-of-fit test is calculated using 1,000 randomly generated data with 10,000 samples applying the power-law exponent $\hat{\alpha}$ estimated by maximum likelihood estimation method with minimum x of the distribution fixed to 1. The Fig. 4a and Fig. 4b show the avalanche size and duration for the state 0 or black. They present distributions that are not a power-law because they do not fit the power-law estimation (the black dashed line). Moreover, the p-value is equal to 0.0 which proves that those two distributions are not a power-law. The Fig. 4c and Fig. 4d present the avalanche size and duration for the state 1 or white. Those distributions follow a power-law because, visually, the estimated power-law distribution fits the empirical probability distribution and, quantitatively, the p-value is equal to 1.0 which means that 100% of the KS statistic of the generated data is greater than the KS statistic of the empirical distribution of avalanche size and duration of state 1. The number of samples in those distributions (62,731 for avalanche size and 52,902 for avalanche duration) confirms that the p-value is trustworthy. Such power-law analysis is performed by utilizing the powerlaw Python library [5]. It is important to warn that high fitness scores do not mean p-values closer to 1.0 and the goodness-of-fit test is not part of the fitness score because it is a slow process.

A sample of the resulting stochastic elementary cellular automaton of the best individual is illustrated in Fig. 5. This CA, as seen, has no static nor periodic states, and no random evolution of its states. Therefore, this dynamical system

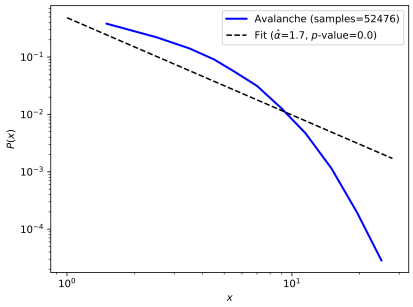
is between a strongly and weakly coupled substrate. Therefore, the CA presents patterns or structures that mean the cells are interdependent in this system.

5 Conclusion

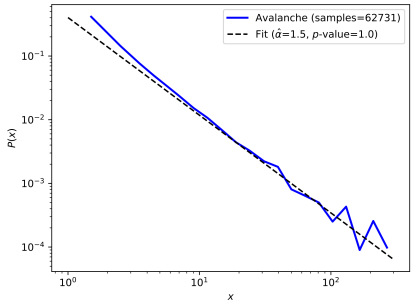
In this paper, we present an alternative method to implement a cellular automaton. This allows any CA to be computed as an artificial neural network. That means, any lookup table can be an activation function, and any neighborhood and dimensionality can be represented as a weight matrix. Therefore, this will help to extend the CA implementation to more complex dynamical systems, such as random Boolean networks, echo state networks and liquid state machines. Furthermore, the EvoDynamic framework is built on a deep learning library, TensorFlow, which permits the acceleration and parallelization of matrix operations when applied on computational platforms with fast CPUs and



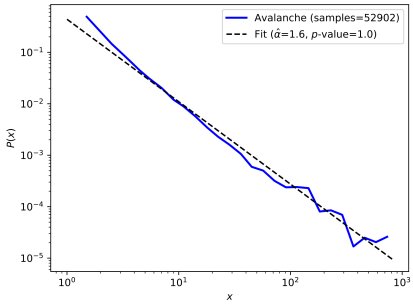
(a) Avalanche size of state 0



(b) Avalanche duration of state 0



(c) Avalanche size of state 1



(d) Avalanche duration of state 1

Fig. 4: Avalanche size and duration of the two states 0 and 1 of the evolved stochastic elementary CA.

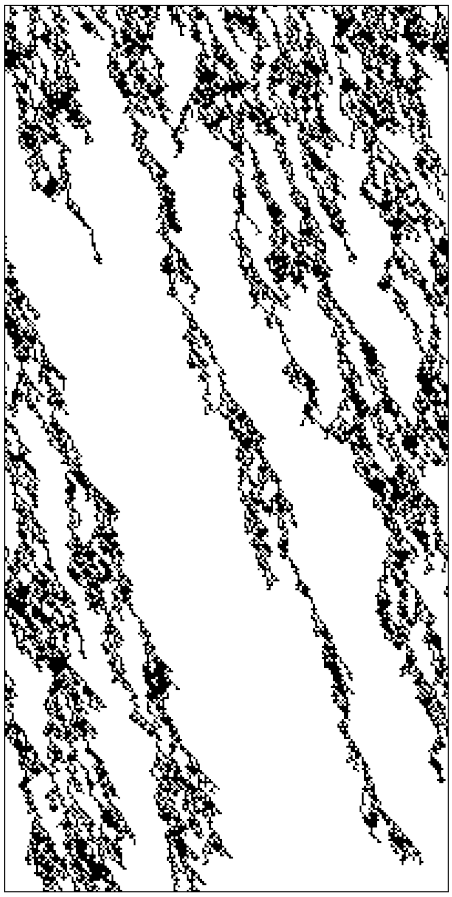


Fig. 5: Sample of the best evolved stochastic elementary CA of 200 cells (horizontal axis) randomly initialized with wrapped boundaries through 400 time-steps (vertical axis).

GPUs. The planned future implementations of EvoDynamic are presented and discussed. The state trajectory is an important feature for the targeted future tasks. The evolution with genetic algorithm towards criticality of stochastic CA is showing promising results and our next goal can be for self-organized criticality. The future work for the CA implementation is to develop algorithms to procedurally generate weighted adjacency matrices for 3D and multidimensional cellular automata with different types of cells, such as the cells with hexagonal shape in a 2D CA.

Acknowledgments

We thank Kristine Heiney for thoughtful discussions about self-organized criticality.

References

1. SOCRATES – Self-Organizing Computational substRATES, https://www.ntnu.edu/socrates
2. Conway's game of life implemented using tensorflow 2d convolution function (2016), https://github.com/conceptacid/conv2d_life
3. Aaser, P., Knudsen, M., Ramstad, O.H., van de Wijdeven, R., Nichele, S., Sandvig, I., Tufte, G., Stefan Bauer, U., Halaas, Ø., Hendseth, S., Sandvig, A., Valderhaug, V.: 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) (29), 430–437 (2017). https://doi.org/10.1162/isal_a_072
4. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X.: Tensorflow: A system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). pp. 265–283. USENIX Association, Savannah, GA (2016)
5. Alstott, J., Bullmore, E., Plenz, D.: powerlaw: A python package for analysis of heavy-tailed distributions. PLOS ONE **9**(1), 1–11 (01 2014). https://doi.org/10.1371/journal.pone.0085777
6. Baetens, J.M., Van der Meeren, W., De Baets, B.: On the dynamics of stochastic elementary cellular automata. Journal of Cellular Automata **12** (2016)
7. Bak, P., Tang, C., Wiesenfeld, K.: Self-organized criticality: An explanation of the 1/f noise. Phys. Rev. Lett. **59**, 381–384 (Jul 1987). https://doi.org/10.1103/PhysRevLett.59.381
8. Broersma, H., Miller, J.F., Nichele, S.: Computational Matter: Evolving Computational Functions in Nanoscale Materials, pp. 397–428. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-33921-4_16
9. Clauset, A., Shalizi, C.R., Newman, M.E.: Power-law distributions in empirical data. SIAM review **51**(4), 661–703 (2009)
10. Gershenson, C.: Introduction to random boolean networks. arXiv preprint nlin/0408006 (2004)
11. Goldberg, D.E., Deb, K.: A comparative analysis of selection schemes used in genetic algorithms. In: Foundations of genetic algorithms, vol. 1, pp. 69–93. Elsevier (1991)
12. Goldstein, M.L., Morris, S.A., Yen, G.G.: Problems with fitting to the power-law distribution. The European Physical Journal B-Condensed Matter and Complex Systems **41**(2), 255–258 (2004)
13. Jaeger, H., Haas, H.: Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. Science **304**(5667), 78–80 (2004). https://doi.org/10.1126/science.1091277

14. Jensen, J.H., Folven, E., Tufte, G.: 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) (30), 15–22 (2018). https://doi.org/10.1162/isal_a_00011
15. Kaneko, K.: Overview of coupled map lattices. Chaos: An Interdisciplinary Journal of Nonlinear Science **2**(3), 279–282 (1992)
16. Konkoli, Z., Nichele, S., Dale, M., Stepney, S.: Reservoir Computing with Computational Matter, pp. 269–293. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-65826-1_14
17. Langton, C.G.: Computation at the edge of chaos: Phase transitions and emergent computation. Physica D: Nonlinear Phenomena **42**(1), 12 – 37 (1990). https://doi.org/https://doi.org/10.1016/0167-2789(90)90064-V
18. Maass, W., Markram, H.: On the computational power of circuits of spiking neurons. Journal of Computer and System Sciences **69**(4), 593 – 616 (2004). https://doi.org/https://doi.org/10.1016/j.jcss.2004.04.001
19. Nichele, S., Tufte, G.: Trajectories and attractors as specification for the evolution of behaviour in cellular automata. In: IEEE Congress on Evolutionary Computation. pp. 1–8 (July 2010). https://doi.org/10.1109/CEC.2010.5586115
20. Nichele, S., Farstad, S.S., Tufte, G.: Universality of evolved cellular automata in-materio. International Journal of Unconventional Computing **13**(1) (2017)
21. Nichele, S., Gundersen, M.S.: Reservoir computing using nonuniform binary cellular automata. Complex Systems **26**(3), 225–245 (Sep 2017). https://doi.org/10.25088/complexsystems.26.3.225
22. Nichele, S., Molund, A.: Deep learning with cellular automaton-based reservoir computing. Complex Systems **26**(4), 319–339 (Dec 2017). https://doi.org/10.25088/complexsystems.26.4.319
23. Nichele, S., Tufte, G.: Genome parameters as information to forecast emergent developmental behaviors. In: Durand-Lose, J., Jonoska, N. (eds.) Unconventional Computation and Natural Computation. pp. 186–197. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
24. Rendell, P.: Turing Universality of the Game of Life, pp. 513–539. Springer London, London (2002). https://doi.org/10.1007/978-1-4471-0129-1_18
25. Schrauwen, B., Verstraeten, D., Van Campenhout, J.: An overview of reservoir computing: theory, applications and implementations. In: Proceedings of the 15th European Symposium on Artificial Neural Networks. p. 471-482 2007. pp. 471–482 (2007)
26. Subramoney, A., Scherr, F., Maass, W.: Reservoirs learn to learn. arXiv preprint arXiv:1909.07486 (2019)
27. Tanaka, G., Yamane, T., Héroux, J.B., Nakane, R., Kanazawa, N., Takeda, S., Numata, H., Nakano, D., Hirose, A.: Recent advances in physical reservoir computing: A review. Neural Networks **115**, 100 – 123 (2019). https://doi.org/https://doi.org/10.1016/j.neunet.2019.03.005
28. TensorFlow: tf.sparse.sparse_dense_matmul — tensorflow core r1.14 — tensorflow, https://www.tensorflow.org/api_docs/python/tf/sparse/sparse_dense_matmul
29. Toffoli, T., Margolus, N.: Cellular automata machines: a new environment for modeling. MIT press (1987)
30. Wolfram, S.: A new kind of science, vol. 5. Wolfram media Champaign, IL (2002)
31. Wright, S.: Correlation and causation. Journal of Agricultural Research **20**, 557–580 (1921)