Deep Learning with Cellular Automaton-Based Reservoir Computing

Stefano Nichele
Department of Computer Science
Oslo and Akershus University College of Applied Sciences
Oslo, Norway
stefano.nichele@hioa.no

Andreas Molund
Department of Computer and Information Science
Norwegian University of Science and Technology
Trondheim, Norway
andrmolu@stud.ntnu.no

Recurrent neural networks (RNNs) have been a prominent concept within artificial intelligence. They are inspired by biological neural networks (BNNs) and provide an intuitive and abstract representation of how BNNs work. Derived from the more generic artificial neural networks (ANNs), the recurrent ones are meant to be used for temporal tasks, such as speech recognition, because they are capable of memorizing historic input. However, such networks are very time consuming to train as a result of their inherent nature. Recently, echo state networks and liquid state machines have been proposed as possible RNN alternatives, under the name of reservoir computing (RC). Reservoir computers are far easier to train. In this paper, cellular automata (CAs) are used as a reservoir and are tested on the five-bit memory task (a well-known benchmark within the RC community). The work herein provides a method of mapping binary inputs from the task onto the automata and a recurrent architecture for handling the sequential aspects. Furthermore, a layered (deep) reservoir architecture is proposed. Performances are compared to earlier work, in addition to the single-layer version. Results show that the single cellular automaton (CA) reservoir system yields similar results to state-of-the-art work. The system comprised of two layered reservoirs does show a noticeable improvement compared to a single CA reservoir. This work lays the foundation for implementations of deep learning with CA-based reservoir systems.

1. Introduction

Temporal tasks, which we humans experience daily, are a great source of inspiration for research within the fields of complex systems and biologically inspired artificial intelligence. Systems capable of

https://doi.org/10.25088/ComplexSystems.26.4.319

solving temporal tasks must be able to memorize historical data. Recurrent neural networks (RNNs) are an example of a system of that sort and have been studied for many years. However, training RNNs is usually compute intensive. One alternative is to consider recurrent networks as an untrained reservoir of rich dynamics and only train an external feed-forward readout layer. The rich dynamics are to provide the necessary projection of the input features onto a discriminative and high-dimensional space. Basically, any substrate equipped with these properties can be used as a reservoir. This paper investigates the use of cellular automata (CAs) computing substrates, inspired by [1].

Cellular automata at a microscopic scale are seemingly simple systems that exhibit simple physics but at a macroscopic scale can reveal complex behavior that might provide the needed reservoir properties. Specifically, CAs are able to support transmission, storage and modification of information [2], all of which are necessary properties to support computation.

Furthermore, stacking reservoir systems in a multilayered setup to offer additional computational capabilities has been successfully applied in [3], using a traditional RNN as a reservoir.

The focus of the work herein is to explore series of CA reservoirs. As such, a system with a single cellular automaton (CA) reservoir has been implemented first, and a second reservoir has been stacked at the end of the first one, to investigate whether two smaller layered reservoirs can replace a single larger one with regard to computational capacity. The single CA reservoir system is therefore compared to earlier work, as well as to the layered version.

The paper is organized as follows. Section 2 presents background information. Section 3 describes the specific method and system architecture in detail. Section 4 provides the experimental setup, and Section 5 outlines the experimental results. A discussion is given in Section 6. Finally, Section 7 provides ideas for future work, and Section 8 concludes the paper.

2. Background

2.1 Reservoir Computing

2.1.1 Fundamentals

Information in feedforward neural networks (NNs) is sent one way through layers of neurons: from an input layer through one (or more) hidden layers to an output layer. Neurons in each layer are connected to neurons in the subsequent layer (except the last one) with weighted edges, and each neuron propagates signals according to its activation

function. An RNN contains the same basic elements. However, it has recurrent connections that feed portions of the information back to the internal neurons in the network, making the RNN capable of memorization [4], hence RNNs are promising architectures for processing of sequential tasks' data, for example, speech recognition. Ways of training RNNs are different variants of backpropagation [4, 5], all with different computational complexity and time consumption.

One fairly recent discovery based upon the fundamentals of RNNs is echo state networks (ESNs) [6]. An ESN is a randomly generated RNN, in which the network does not exhibit any layer structure, and its internal connection weights remain fixed (untrained) and can be treated as a reservoir of dynamics. The "echo state property" is the activation state of the whole network being a function of previous activation states. Training of such a network involves adapting only the weights of a set of output connections.

Another similar discovery is liquid state machines (LSMs) [7]. Liquid state machines are similar to ESNs in terms of topology, with an internal randomly generated neural network and problem-specific trained output weights.

The basic idea of having readout nodes with trained weights connected to an arbitrary number of neurons inside the untrained reservoir has been named reservoir computing (RC). Figure 1 depicts this general RC idea.

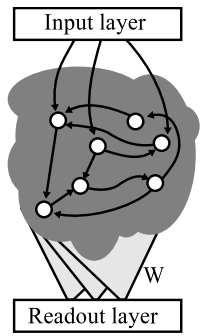


Figure 1. A generic reservoir. It is only necessary to adapt weights W to a certain target.

2.1.2 Physical Reservoir Implementations

Different physical substrates have been shown to possess the necessary rich dynamics to act as a reservoir. Potentially, any high-dimensional dynamic medium or system that has the desired dynamic properties can be used. For example, in [8] a linear classifier was used to extract information from the primary visual cortex of an anes-

thetized cat. In [9], waves produced on the surface of water were used as an LSM to solve a speech recognition task. The genetic regulatory network of the *Escherichia coli* bacterium (*E. coli*) was used as an ESN in [10] and as an LSM in [11]. In [12–14], unconventional carbon nanotube materials were configured as a reservoir through artificial evolution. An optoelectronic reservoir implementation is presented in [15, 16].

2.1.3 Deep Reservoirs

Within the RC research field, it has been suggested that reservoir performances may be improved by stacking multiple reservoirs [3, 17, 18]. A critical analysis of deep reservoir systems is given in [19]. In a deep reservoir system, since the hidden units in the reservoir are not trainable, the reservoir's readout values are sent as input to the next reservoir. Thus, the reservoir and its associated output layer are stacked in a multilayered (possibly deep) architecture. This technique is inspired by deep neural networks, in which adding layers of hidden units increases the ability of representation and abstraction, and thus the performance of the system. One argument for stacking multiple reservoir systems is that the errors of one reservoir may be corrected by the following one, which may learn the semantics of the pattern that it gets as input. As an example, in [3] a deep reservoir architecture based on ESNs proved successful in phoneme recognition.

2.2 Cellular Automata

Cellular automata were inspired by the study of self-reproducing machines by von Neumann in the 1940s [20]. Cellular automata are able to show emergent behavior; that is, the macroscopic properties are hard to explain from solely looking at the microscopic properties. Within a CA, simple cells communicate locally over discrete time. Locally means that a cell only interacts with its immediate neighbors; thus it has no global control. The cells are discrete and placed on a regular grid of arbitrary dimension. The most common ones are one- and two-dimensional. At each time step, all cells on the grid are updated synchronously based on their physics, that is, a transition to a new state based on the previous state of the cell itself and its neighbors. Such transition tables are also referred to as CA rules.

Regarding the rule space, if K is the number of states a cell can be in, and N is the number of neighbors (including itself), then K^N is the total number of possible neighborhood states. Furthermore, each element is transitioning to one of K states; thus, the transition function space is of size K^{K^N}. For example, in a universe where cells have five possible states and three neighbors, there are $5^{5^3} \approx 2.4 \times 10^{87}$ different rules or possible transition functions.

Elementary CAs are the simplest class of one-dimensional CAs. They comprise cells laid out in one dimension, in which $K = 2$ and $N = 3$. The rule space can be enumerated in a base-2 system; each of the $2^8 = 256$ transition functions can be represented by a base-2 number of length 8, as for example rule 110 shown later in Figure 3 that is represented as $(01\,101\,110)_2$.

Going a step in a more general direction, all one-dimensional CAs were categorized by Wolfram [21] into four qualitative classes, based on the resulting evolution, that is, the emergent CA behavior. Evolving one-dimensional CAs can easily be visualized by plotting the whole spacetime diagram, iteration by iteration, downward; see Figure 2 for illustrations. Cellular automata in class I will always evolve to homogeneous cell states, independent of the initial states. Class II leads to periodic patterns or single everlasting structures, either of which outcome is dependent on initial local regions of cell states. Class III leads to a chaotic and seemingly random pattern. Finally, class IV leads to complex localized structures that are difficult to predict; see Figure 2(d).

Langton introduced a scheme for parameterizing rule spaces in [2], namely the λ parameter. Briefly explained, within a transition function, the value of λ represents the fraction of transitions that lead to a quiescent state. As an example, rule 110 in Figure 3 has $\lambda = 0.625$. If $\lambda = 0.0$, then everything will transition to 0, and the automaton will clearly lead to a homogeneous state. λ is especially useful for large rule spaces where it is hard to exhaustively enumerate all rules, because it can be used to generate rules with the desired behavior. Langton [2] did a qualitative survey throughout the rule space on one-dimensional CAs with $K = 4$ and $N = 5$; rules were generated from different values of λ, from which CAs were evolved and analyzed. As the parameter increased from 0.0 to 1.0, the observed behavior underwent various phases, all the way from activity quickly dying out to fully chaotic. In the vicinity of phase transition between ordered and chaotic, a subset of all CA rules was observed to lead to complex behavior that produced long-lasting structures and large correlation lengths. Langton suggested that in this "edge of chaos" region is where computation may spontaneously emerge.

2.3 Cellular Automata in Reservoir Computing

Cellular automata reservoirs were first introduced in [1], and discussed subsequently in [22–24]. In [25] the usage of nonuniform CAs was proposed, and in [26] a CA reservoir system was used for modality classification of medical images. In [27], pairs of CA rules were used for reservoir computing and extreme learning machines.

https://doi.org/10.25088/ComplexSystems.26.4.319

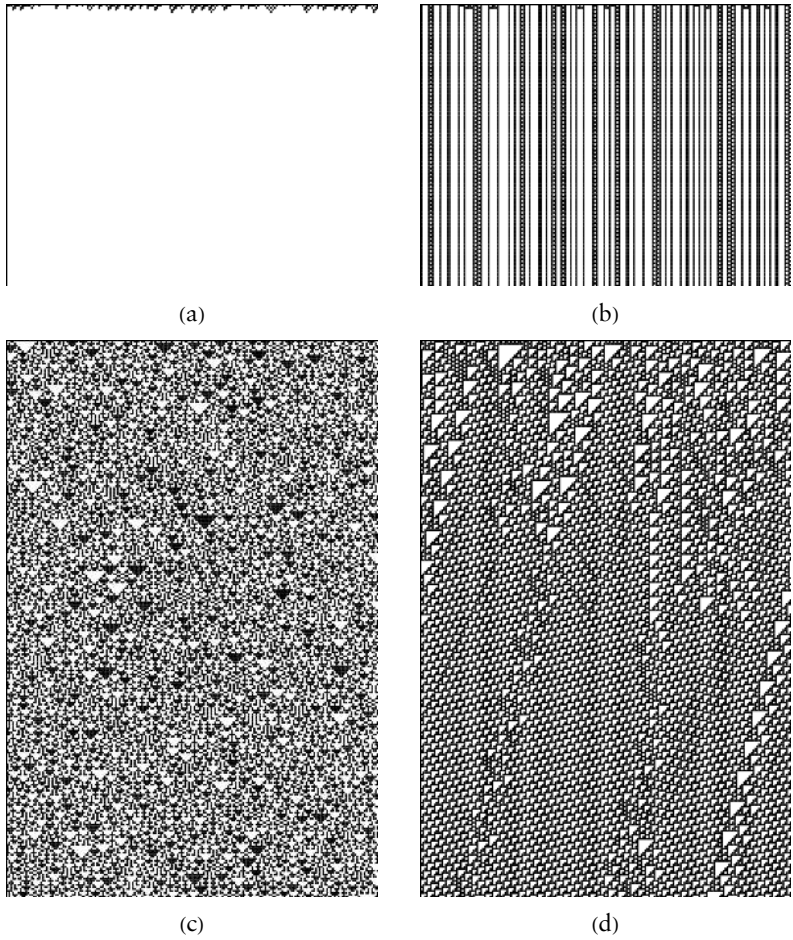


Figure 2. Elementary cellular automata iterating downward. (a) and (b) are cut short. A black cell represents 1. These four are examples of each of Wolfram's classes: (a) is class I with rule 40, (b) is class II with rule 108, (c) is class III with rule 150, and (d) is class IV with rule 110.

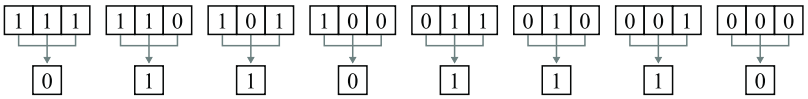


Figure 3. The elementary CA rule 110 $(01\,101\,110)_2 = (110)_{10}$.

Since the automata cells take on values from a discrete and finite set, mapping schemes to translate inputs onto CAs may be needed. For problems and tasks of a binary nature, such as five-bit memory

tasks [28] and temporal bit parity and density [29], this is relatively straightforward. For input with real values, there are different proposed mapping methods [1, 25, 26]. After translation, a rule is then applied to the automaton for some iterations, each of which is recorded, so the nonlinear evolution becomes a projection of the input onto a discriminating state space. This projection is later used in regression and classification for the task at hand.

Cellular automata as reservoirs provide several benefits over ESNs. One is that the selection of reservoir, that is, the CA transition table, is trivial; it is merely a choice of a CA rule with the wanted dynamics. Even in elementary CAs, one of the simplest forms, there exist rules that are Turing complete, that is, capable of universal computation [30, 31]. Another improvement is the aspect of computational complexity. According to [1], the speedups and energy savings for the N-bit task are almost two orders of magnitude because of the numbers and type (bitwise) of operations. Binarized variations of deep neural networks [32, 33] and neural GPUs [34] have been recently suggested, in order to allow easier implementations in hardware than conventional deep neural network architectures. In [35], binarized neural networks are implemented on field-programmable gate arrays (FPGAs). Such binary implementations are well suited for reconfigurable logic devices. One advantage of using binary CAs (locally connected) over deep neural networks (fully connected) is a significantly lower memory cost and computation cost (binary operations implemented with a lookup table or bitwise logic in case of additive rules).

A vast sea of possibilities exists regarding how to set up a CA reservoir system. For example, in a recent paper [24], memory enhancements of the CA are explored by adopting pre-processing methods prior to evolution. Further research with these possibilities can provide new understanding and insight in the field of reservoir computing with cellular automata (ReCA) and CA-based deep learning.

3. Method

In this section, the ReCA system used is described in detail (Figure 4). The first implementation comprises a single reservoir tested (with several parameters) on the five-bit memory task to compare to state-of-the-art results [22, 23]. In the second implementation, an additional reservoir is added, whose input is the output of the first one. This larger system is tested on the same task (five-bit memory) for comparison. The code base that is used in this paper is available for download at https://github.com/andreasmolund/reca.

https://doi.org/10.25088/ComplexSystems.26.4.319

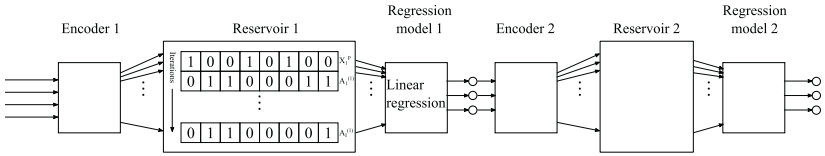


Figure 4. System architecture.

3.1 System Architecture

Elementary cellular automata are used as the medium in the reservoirs; that is, their cells have three neighbors (including themselves), each of which can be in one of two states. This means that there are 256 rules that can be applied, not all of which are used in this paper. A selection of rules is presented in Section 5.

In the encoding stage, input to the system is mapped onto the automata. Since the problem can be represented with binary vectors, the input elements translate directly into cell states. This is one of the two input-to-automaton options proposed in [1], with the other one being for nonbinary input data. In addition to regular translation, the encoder is also responsible for padding and for diffusing the input onto an area that is of greater size than the input, if desired. Padding is the method of adding elements of no information, in this case zeros, at some end of the mapped vector. These buffers are meant to hold some activity outside of the area where the input is perturbing. Thus, diffusing is a sort of padding by inserting zeros at random positions instead of at the end. It disperses the input to a larger area. The length of the area to which the input is diffused is denoted L_d. Currently, out of these two methods of enlarging the memory capacity, only the diffuse parameter L_d is set. Figure 5 illustrates how the system is mapping input to automata.

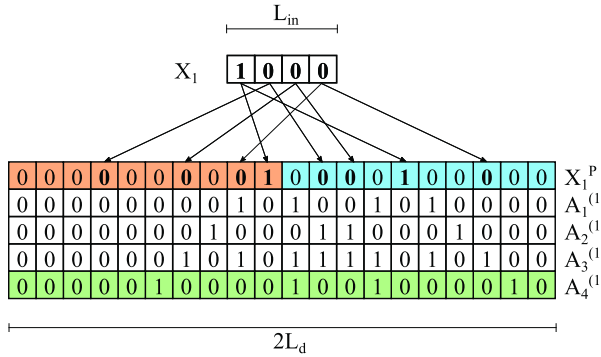


Figure 5. Encoding input onto an automaton. $L_{in} = 4$, $R = 2$, $L_d = 10$, $I = 4$. The two different colors of X_1^P signify the two different random mappings.

A reservoir can consist of R separate CAs, each of which initial configuration is a randomly mapped input. At system initialization, the indexes used for random mapping are generated, which means that the mappings are final and do not change throughout computation. These automata are concatenated at the beginning of evolution to form a large initial configuration of size $R \times L_d$. It is also possible to concatenate them after they are done iterating, but that proved to yield worse results. At the boundaries, the automata are wrapped around; that is, the rightmost cell has the leftmost cell as its right neighbor, and vice versa.

For the five-bit memory task described later in this paper, the system needs to be able to handle sequential inputs. In [22], a recurrent architecture for CAs in reservoir computing is proposed.

The system is initialized with an input X_1 at the first time step. X_1 is permuted; that is, its elements are randomly mapped onto a vector of zeros according to the mapping scheme R times and concatenated to form the initial configuration of the automaton X_1^P:

$$X_1^P = \left[X_1^{P_1}; X_1^{P_2}; \ldots X_1^{P_R}\right].$$

The automaton is now of length $R \times L_d$. Z is said to be the transition function, or rule, and is applied to the automaton for I iterations. This renders an expressive and discriminative spacetime volume of the input:

$$\begin{aligned} A_1^{(1)} &= Z\left(X_1^P\right) \\ A_2^{(1)} &= Z\left(A_1^{(1)}\right) \\ &\vdots \\ A_I^{(1)} &= Z\left(A_{I-1}^{(1)}\right). \end{aligned} \quad s$$

$A_1^{(1)}$ through $A_I^{(1)}$ constitutes the evolution of the automaton and is concatenated to form a state vector used for estimation at the first time step. It is possible to include the permuted version of the input, that is, the state before the first application of the rule, which is the case for the feedforward architecture in [22], for example. However, it is excluded here:

$$A^{(1)} = \left[A_1^{(1)}; A_2^{(1)}; \ldots A_I^{(1)}\right].$$

Because this is a recurrent architecture, a fraction of the state vector is combined with the next input. Several methods exist; XOR, "normalized addition," which is adopted in [22], and a variant of overwriting, which is implemented in [23]. For the subsequent time step, depicted in Figure 6, the last iteration of the previous state

vector is duplicated, after which the next input is permuted and written onto. In other words, instead of mapping the input onto a zero vector, it is done onto a vector that already contains information:

$$X_2^P = Y\left(X_2, A_I^{(1)}\right)$$

where Y is the function that overwrites $A_I^{(1)}$ with the permuted elements of X_2. One implication about this process is that the operation cannot be vectorized to an elementwise operand and hence hamper performance. The positive is that no input information is lost; for example, one of the input bits being zero (a signal is switched to off) will affect the subsequent evolution. In other methods such as the probabilistic normalized addition, we rely on an increasing number of random mappings to increase the probability that input information is preserved. To obtain the next state vector $A^{(2)}$, the transition function is applied on X_2^P for I iterations and concatenated:

$$A^{(2)} = \left[A_1^{(2)}; A_2^{(2)}; \ldots A_I^{(2)}\right].$$

$A^{(2)}$ is consequently used for estimation of time step 2. This process is repeated for every time step of the input sequence.

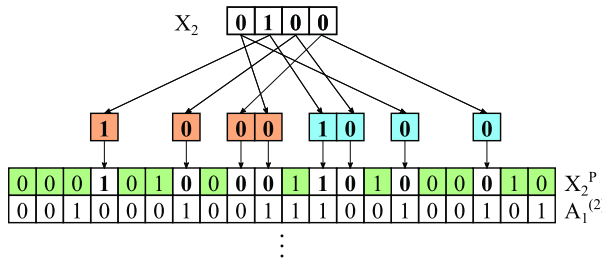


Figure 6. Combining input with portions of the previous state. X_2^P has traces of $A_4^{(1)}$ from Figure 5.

3.2 Readout

As we can infer from what was described earlier, the number of readout values from the reservoir depends on the diffuse length and the number of random mappings and iterations. The readout values from one time step are sent into a linear regression model together with their corresponding labels. Specifically, the linear_model.LinearRegression class from scikit-learn [36] is used. For ease of training, the model is fitted all at once with the output from all time steps for each element in the training set, together with their labels. Even though the elements are from different time steps, from different locations in the training set, they are weighted and treated equally because they each retain (to a greater or lesser degree) history from their

respective "time lines." Each corresponding label represents semantics from which the model is to interpret the readout values.

After the model is fitted, it can be used to predict. Because linear regression is used, the output values from the predictions are floating points. The output value x is binarized according to equation (1)

$$x^b = \begin{cases} 0 & \text{if } x < 0.5, \\ 1 & \text{otherwise.} \end{cases} \tag{1}$$

3.3 Deep Cellular Automaton Reservoir

The RC framework described so far consists of one CA reservoir. This approach is now expanded with new components as depicted in Figure 4, that is, a second encoder, reservoir and regression model. After the values of the first readout stage are classified, it is used as input to the second system. Both regression models are fitted to the same target labels. One motivation for connecting two reservoirs together is that the second can correct some of the wrong predictions of the first one.

Training the system as a whole (and really testing it as well) involves the procedure that follows. Inputs are encoded and mapped onto automata from which the first reservoir computes state vectors. As this input is now transformed already, it is stored to be used for later prediction. The first regression model is fitted with these feature vectors to the corresponding labels and does its prediction immediately after. These predictions are binarized, encoded and mapped onto automata from which the second reservoir computes new state vectors. Consequently, the second regression model follows the same training procedure as the first one, only with these new state vectors. When the training is completed, the second regression model is tested for classification.

4. Experiment

4.1 Five-Bit Memory Task

One sequential benchmark that has been applied to reservoir computing systems is the N-bit memory task [1, 22, 23, 25, 37], which is found to be hard for feedforward architectures [28].

In the five-bit memory task, a sequence of binary vectors of size four is presented to the system, where each vector represents one time step. The four elements therein act as signals; thus, only one of them can be 1 at a time step. This constraint also applies on the output, which is also a binary vector, but rather with three elements. In [28], the problem was formulated with four output bits, but the fourth is "unused," hence it is omitted in the implementation herein.

https://doi.org/10.25088/ComplexSystems.26.4.319

For the first five time steps in one run, the first two bits in the input vector are toggled between 0 and 1. This is the information that the system is to remember. If one of them is 1, the other one is 0, and vice versa; hence, there is a total of 32 possible combinations for the five-bit task. From time step 6 throughout the rest of the sequences, the third bit is set to 1, except at time $T_d + 5$, where the fourth bit is set to 1. The third bit is the distractor signal and indicates that the system is waiting for the cue signal (i.e., the fourth bit). All runs presented in Section 5 are with $T_d = 200$, meaning a total sequence length $T = T_d + 2 \times 5 = 210$.

As for the output, for all time steps until $T_d + 5$ inclusive, the third bit is 1. Thereafter, the first and second bit are to replicate the first five input signals. See Figure 7 for an example.

Timestep	Input				Output		
1	1	0	0	0	0	0	1
2	0	1	0	0	0	0	1
3	0	1	0	0	0	0	1
4	1	0	0	0	0	0	1
5	1	0	0	0	0	0	1
6	0	0	1	0	0	0	1
7	0	0	1	0	0	0	1
8	0	0	0	1	0	0	1
9	0	0	1	0	1	0	0
10	0	0	1	0	0	1	0
11	0	0	1	0	0	1	0
12	0	0	1	0	1	0	0
13	0	0	1	0	1	0	0

Figure 7. An example of the five-bit memory task with a distractor period $T_d = 3$. The cue signal occurs at time step 8, after which the first and second bit of the output are replicating the equivalent bits in the input (marked in gray).

4.2 System Setup

To collect the results, all 32 possible input patterns are used for both training and testing. Because the mappings are final throughout one run and hence the system is deterministic, the regression model is basically fitted to the same output that it is to predict. One run is said to be successful if the system can predict the right output for every time step for all of the 32 possible testing sets. That means a total of $3 \times 210 \times 32$ bits correctly predicted. Either 1000 or 100 runs are performed (and specified in the results figure), as explained in Section 5. The diffuse length is $L_d = 40$.

5. Results

The results of the five-bit memory task are presented in this section and summarized in Tables 1 and 2. Only the most promising rules have been tested with large combinations of I and R. The combination of I and R is denoted (I, R) throughout the rest of the paper.

Rule	(I,R)=(2,4)	(2,8)	(4,4)	(4,8)	(8,8)	(8,16)	(16,8)	(16,16)
90	18.5	45.9	29.2	66.1	100	100	98	100
150	0.2	1.8	6.7	33.7	89	100	100	100
182	0.0	0.0	0.0	0.1	0	100	99	100
22	0.0	0.0	0.0	0.0	0	99	100	100
60	4.5	22.7	28.2	71.2	99	100	100	
102	6.0	24.0	28.1	69.7	97	100		
105	0.3	2.5	7.9	31.7	90			
153	3.1	20.2	28.9	70.6	99			
165	3.4	29.2	14.6	56.1	94			
180	0.0	0.0	0.0	0.0	0			
195	3.4	21.4	26.5	67.2	98			

Table 1. The correctness (%) from the first reservoir computing system. Up until $(I, R) = (4, 8)$ inclusive, 1000 runs were executed, hence the single-decimal precision. With greater I and R, only 100 runs were executed.

Rule	(I,R)=(2,4)	(2,8)	(4,4)	(4,8)	(8,8)	(8,16)	(16,8)	(16,16)
90	16.6	49.4	38.0	73.9	100	100	99	100
150	0.3	3.5	10.4	39.7	90	100	100	100
182	0.0	0.0	0.0	6.0	2	100	100	100
22	0.0	0.0	0.0	0.0	0	100	100	100
60	9.4	30.0	33.7	74.4	99	100	100	
102	9.8	31.9	35.2	71.9	97	100		
105	0.7	3.7	11.5	37.2	91			
153	5.0	24.6	35.4	73.9	99			
165	4.8	35.0	22.4	63.7	95			
180	0.1	0.2	0.1	0.1	0			
195	5.4	27.3	33.6	71.7	99			

Table 2. The correctness (%) from the second reservoir computing system. What was mentioned in the caption of Table 1 still applies here.

From $(8, 8)$ inclusive and up, the results are based on 100 runs. Under that, they are based on 1000 runs; hence they are given with a single-decimal precision.

Only a selection of all 256 possible rules is selected for experimentation. Rules are selected based on available literature for comparison, for example, [1, 23]. In [1], it is found that rules 22, 30, 126,

150, 182, 110, 54, 62, 90 and 60 are able to give 0 error for some combinations of (I, R), where the best-performing rules are 90, 150, 182 and 22, in decreasing order. In [23], results are provided for rules 60, 90, 102, 105, 150, 153, 165, 180 and 195.

6. Discussion

6.1 Comparison with Earlier Work

The individual results in [23] are not quite equal to the equivalent in Table 1. Some values differ noticeably, for example, rules 90 and 165 at $(4, 4)$, which in [23] result in a lower correctness. The differences may be due to the different concatenation of random mappings, that is, prior to CA evolution herein, whereas it is concatenated after in [23]. Furthermore, no rule is able to achieve 100% correctness under $(8, 8)$, which is also the case in [23].

For the five-bit task with a distractor period of 200, the best-performing rule in [22] needs a minimum of $(I, R) = (32, 40)$ to produce 100% correct results. That means a state vector of size (or trainable parameters) $32 \times 40 \times 4 = 5120$. The method proposed herein needs $I \times R \times L_d = 8 \times 8 \times 40 = 2560$, according to Table 1.

Some rules presented in Table 1 are essentially equivalent. Rule 102 is black-white equivalent with 153—that is, they interchange the black and white cells—and left-right equivalent with rule 60—that is, they interchange left and right cells. The rule is furthermore both black-white and left-right equivalent with rule 195. With these four rules being somehow equivalent, the obtained results in Table 1 are also approximately equal.

Padding versus diffusing is furthermore experimented with, although not documented in this paper. In the few executed tests, padding alone is observed to produce more stable results. On the contrary, with only diffusion, the tests seemed to yield better results overall but with higher variance. The reason is most likely due to the larger area to which the input is mapped; individual elements of mapped input can be both very far apart and very close, while they are immediately adjacent when padding.

Another side experiment was executed to compare the fitting time when doubling the reservoir size. The number of random mappings was doubled from four (i.e., number of trainable parameters or readout nodes was $I \times R \times L_d = 4 \times 4 \times 40 = 640$). When doubling R and hence doubling the state vector size, the outcome was an increase in fitting time by a factor of 3.4. A set of 32 was used, each with 211 sequences. It is a rough figure, but it gives an indication of the computational complexity.

6.2 Layered Reservoirs

It is quite intriguing how information represented by three bits from the first reservoir computing system can be corrected by the second one. From a human's perspective, three bits is not much to interpret. Nevertheless, the dynamics of the architecture and expressiveness of CAs prove to be able to improve the result from a single reservoir system.

Table 2 provides results on the correctness of the two-layer reservoir. The results in each cell are directly comparable to the equivalent in Table 1; for example, rule 150 at $(4, 4)$ improved from 6.7% correctness to 10.4%.

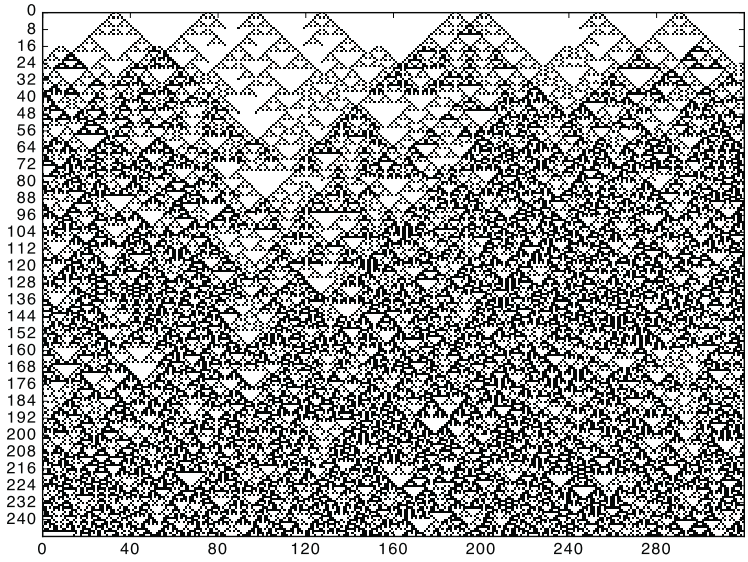
Comparing the two tables, no rule in the second reservoir managed to get 100% before the first reservoir. The rule that first is able to predict 100% correctly is 90 at $(8, 8)$, and in a multilayered setup, the same still applies: rule 90 at $(8, 8)$. Below this (I, R) and where the first reservoir gets $> 10\%$, the best improvement gain is 53.4% for rule 165 at $(4, 4)$. Rule 165 also has the highest average improvement with 21.7%.

Overall performance seems to increase with two reservoirs, the only decrease being rule 90 at the lowest (I, R). One possible explanation for the decrease is that the reservoir has reached an attractor. If it has, it must have occurred within the distractor period where the input signal does not change. The system has been observed to reach an attractor when the addition method in the recurrent architecture is XOR, in which case the system reached an attractor after around two time steps within the distractor period. However, the described implementation in this paper does not use XOR.

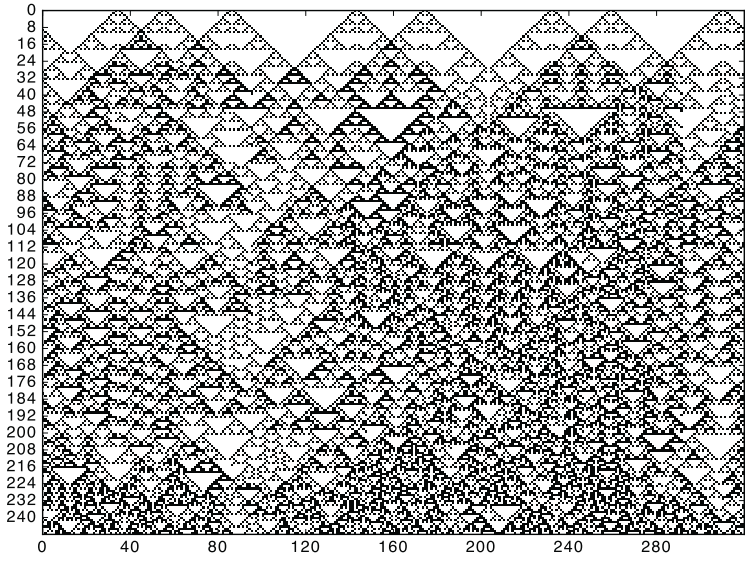
An intuitive comparison would be to compare whether adding a second reservoir of equal capacity can perform better than a single reservoir with twice the random mappings. In that case, a viable configuration of rule and (I, R) does not seem to exist. However, when adding training time in the equation, a tradeoff might be reasonable.

Figure 8 is a visualization of the actual CA states on a successful run on the five-bit task with two reservoirs, although the distractor period is shortened down to 20 time steps. Each tick on the vertical axis signifies the beginning of a new time step, and right before each tick, new input is added onto the CA state. The input itself cannot be spotted, but the effects of it can (to a certain degree). Spotting the input signals is feasible at early time steps, but gets more difficult at later iterations.

https://doi.org/10.25088/ComplexSystems.26.4.319



(a)



(b)

Figure 8. An example run on the five-bit task with two reservoirs. (a) is the first reservoir and (b) is the second. $I = 8$, $R = 8$, $L_d = 40$, and the distractor period is shortened to 20.

7. Future Work

The results presented in this paper show that a system with two-layer reservoirs performs better than a single reservoir. This paper briefly touched upon one part of the vast spectrum of options and methods to opt for a practical implementation of ReCA systems. These options include the mapping method, investigating a larger rule space, the target of training for each regression model, the parameters for each reservoir, the rule of each reservoir and the possibility of using two-dimensional CAs (e.g., Conway's Game of Life). Especially interesting is the mapping method, or more generally, preprocessing before exciting the medium within the reservoir. For example, in [24], buffering and methods for handling subsequent inputs yield promising results.

One of the most interesting avenues for future work is to experiment further with more than two reservoirs, that is, a deep ReCA system.

In addition, because of the nature of CAs, ReCA systems are suitable for implementation in FPGAs. Cellular automata completely avoid floating-point multiplications as opposed to ESNs, and furthermore, the regression stage can be replaced by summation [1].

8. Conclusion

In this paper, a reservoir computing system with cellular automata serving as the reservoir was implemented. Such a system was tested on the five-bit memory task. The system was also expanded to a two-layer reservoir, in which the output of the first reservoir inputs to the second reservoir. Output of the first reservoir was used to compare the result with state-of-the-art work, as well as to the implementation with a layered reservoir. One of the main motivations for opting for a two-layer system is that the second reservoir can correct some of the incorrect predictions of the first one.

The results for the layered system show noticeable improvements when compared to the single reservoir system. The greatest improvement (53.42%) was achieved by rule 165 at $(4, 4)$. Rule 165 proved to be promising in general, with an average improvement of 21.71%.

Overall, the second reservoir does improve the results of the first one to a certain degree. This work lays the foundation for cellular automaton-based deep learning implementations.

https://doi.org/10.25088/ComplexSystems.26.4.319

References

[1] O. Yilmaz, "Reservoir Computing Using Cellular Automata." arxiv.org/abs/1410.0162.

[2] C. G. Langton, "Computation at the Edge of Chaos: Phase Transitions and Emergent Computation," *Physica D: Nonlinear Phenomena*, **42**(1–3), 1990 pp. 12–37. doi:10.1016/0167-2789(90)90064-V.

[3] F. Triefenbach, A. Jalalvand, B. Schrauwen and J.-P. Martens, *Phoneme Recognition with Large Hierarchical Reservoirs*, Ghent University Std., 2010. (Oct 16, 2017) papers.nips.cc/paper/4056-phoneme-recognition-with-large-hierarchical-reservoirs.pdf.

[4] H. Jaeger, *A Tutorial on Training Recurrent Neural Networks, Covering BPPT, RTRL, EKF and the "Echo State Network" Approach*, GMD Report 159, Germany: German National Research Center for Information Technology, 2002. (Oct 16, 2017) minds.jacobs-university.de/sites/default/files/uploads/papers/ESNTutorialRev.pdf.

[5] P. J. Werbos, "Backpropagation through Time: What It Does and How to Do It," *Proceedings of the IEEE*, **78**(10), 1990 pp. 1550–1560. doi:10.1109/5.58337.

[6] H. Jaeger, *The "Echo State" Approach to Analysing and Training Recurrent Neural Networks—with an Erratum Note*, GMD Report 148, Bonn, Germany: German National Research Center for Information Technology, 2001. (Oct 17, 2017) minds.jacobs-university.de/sites/default/files/uploads/papers/EchoStatesTechRep.pdf.

[7] W. Maass, T. Natschläger and H. Markram, "Real-Time Computing without Stable States: A New Framework for Neural Computation Based on Perturbations," *Neural Computation*, **14**(11), 2002 pp. 2531–2560. doi:10.1162/089976602760407955.

[8] D. Nikolić, S. Haeusler, W. Singer and W. Maass, "Temporal Dynamics of Information Content Carried by Neurons in the Primary Visual Cortex," in *Proceedings of the 19th International Conference on Neural Information Processing Systems (NIPS'06)*, Cambridge, MA: MIT Press, 2006 pp. 1041–1048. dl.acm.org/citation.cfm?id=2976456.2976587.

[9] C. Fernando and S. Sojakka, "Pattern Recognition in a Bucket," in *Proceedings of Advances in Artificial Life (ECAL 2003)*, Dortmund, Germany (W. Banzhaf, J. Ziegler, T. Christaller, P. Dittrich and J. T. Kim, eds.), Berlin, Heidelberg: Springer, 2003 pp. 588–597. doi:10.1007/978-3-540-39432-7_63.

[10] X. Dai, "Genetic Regulatory Systems Modeled by Recurrent Neural Network," in *Proceedings, Part II, Advances in Neural Networks: International Symposium on Neural Networks (ISNN 2004)* Dalian, China. Berlin, Heidelberg: Springer, 2004 pp. 519–524. doi:10.1007/978-3-540-28648-6_ 83.

[11] B. Jones, D. Stekel, J. Rowe and C. Fernando, "Is There a Liquid State Machine in the Bacterium *Escherichia coli*?," in *Proceedings of the IEEE Symposium on Artificial Life 2007 (ALIFE'07)*, Honolulu, HI, IEEE, 2007 pp. 187–191. doi:10.1109/ALIFE.2007.367795.

[12] M. Dale, J. F. Miller and S. Stepney, "Reservoir Computing as a Model for *in-Materio* Computing," *Advances in Unconventional Computing: Volume 1: Theory* (A. Adamatzky, ed.), Cham, Switzerland: Springer International Publishing, 2017 pp. 533–571. doi:10.1007/978-3-319-33924-5_22.

[13] M. Dale, J. F. Miller, S. Stepney and M. A. Trefzer, "Evolving Carbon Nanotube Reservoir Computers," in *Proceedings of International Conference on Unconventional Computation and Natural Computation (UCNC 2016)*, Manchester, UK; Cham, Switzerland: Springer International Publishing, 2016 pp. 49–61. doi:10.1007/978-3-319-41312-9_5.

[14] M. Dale, S. Stepney, J. F. Miller and M. Trefzer, "Reservoir Computing *in Materio*: An Evaluation of Configuration through Evolution," in *Proceedings of the 2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, Athens, Greece, IEEE, 2016. doi:10.1109/SSCI.2016.7850170.

[15] Y. Paquot, F. Duport, A. Smerieri, J. Dambre, B. Schrauwen, M. Haelterman and S. Massar, "Optoelectronic Reservoir Computing." arxiv.org/abs/1111.7219.

[16] L. Larger, M. C. Soriano, D. Brunner, L. Appeltant, J. M. Gutiérrez, L. Pesquera, C. R. Mirasso and I. Fischer, "Photonic Information Processing beyond Turing: An Optoelectronic Implementation of Reservoir Computing," *Optics Express*, **20**(3), 2012 pp. 3241–3249. doi:10.1364/OE.20.003241.

[17] A. Jalalvand, G. Van Wallendael, and R. Van De Walle, "Real-Time Reservoir Computing Network-Based Systems for Detection Tasks on Visual Contents," in *2015 7th International Conference on Computational Intelligence, Communication Systems and Networks (CICSyN)*, Riga, Latvia, IEEE, 2015 pp. 146–151. doi:10.1109/CICSyN.2015.35.

[18] A. Jalalvand, F. Triefenbach, K. Demuynck, and J.-P. Martens, "Robust Continuous Digit Recognition Using Reservoir Computing," *Computer Speech & Language*, **30**(1), 2015 pp. 135–158. doi:10.1016/j.csl.2014.09.006.

[19] C. Gallicchio and A. Micheli, "Deep Reservoir Computing: A Critical Analysis," in *Proceedings of European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2016)*, Bruges, Belgium, 2016 pp. 497–502. www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2016-175.pdf.

[20] J. Von Neumann, *Theory of Self-Reproducing Automata* (A. W. Burks, ed.), Urbana, IL: University of Illinois Press, 1966 pp. 3–14.

[21] S. Wolfram, "Universality and Complexity in Cellular Automata," *Physica D: Nonlinear Phenomena*, **10**(1–2), 1984 pp. 1–35. doi:10.1016/0167-2789(84)90245-8.

[22] O. Yilmaz, "Connectionist-Symbolic Machine Intelligence Using Cellular Automata Based Reservoir-Hyperdimensional Computing." arxiv.org/abs/1503.00851.

[23] E. T. Bye, "Investigation of Elementary Cellular Automata for Reservoir Computing," Master's thesis, NTNU, Norway, 2016. brage.bibsys.no/xmlui/handle/11250/2415318.

[24] M. Margem and O. Yilmaz, "How Much Computation and Distributedness Is Needed in Sequence Learning Tasks?," in *Artificial General Intelligence, 9th International Conference, (AGI 2016)*. New York (B. Steunebrink, P. Wang and B. Goertzel, eds.), New York: Springer, 2016 pp. 274–283. doi:10.1007/978-3-319-41649-6_ 28.

[25] S. Nichele and M. S. Gundersen, "Reservoir Computing Using Non-uniform Binary Cellular Automata." arxiv.org/abs/1702.03812.

[26] D. Kleyko, S. Khan, E. Osipov and S.-P. Yong, "Modality Classification of Medical Images with Distributed Representations Based on Cellular Automata Reservoir Computing," in *Proceedings of the 2017 14th International Symposium on Biomedical Imaging (ISBI 2017)*, Melbourne, Australia, 2017, IEEE, 2017. doi:10.1109/ISBI.2017.7950697.

[27] N. McDonald, "Reservoir Computing and Extreme Learning Machines Using Pairs of Cellular Automata Rules." arxiv.org/abs/1703.05807.

[28] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, **9**(8), 1997 pp. 1735–1780. doi:10.1162/neco.1997.9.8.1735.

[29] D. Snyder, A. Goudarzi and C. Teuscher, "Computational Capabilities of Random Automata Networks for Reservoir Computing," *Physical Review E*, **87**(4), 2013 042808. doi:10.1103/PhysRevE.87.042808.

[30] S. Wolfram, *A New Kind of Science*, Champaign, IL: Wolfram Media, Inc., 2002.

[31] M. Cook, "Universality in Elementary Cellular Automata," *Complex Systems*, **15**(1), 2004 pp. 1–40. www.complex-systems.com/pdf/15-1-1.pdf.

[32] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv and Y. Bengio, "Binarized Neural Networks," in *Advances in Neural Information Processing Systems 29 (NIPS 2016)* (D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon and R. Garnett, eds.), Curran Associates, Inc., 2016 pp. 4107–4115.

[33] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre and K. Vissers, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference." arxiv.org/abs/1612.07119.

[34] Ł. Kaiser and I. Sutskever, "Neural GPUs Learn Algorithms." arxiv.org/abs/1511.08228.

[35] N. J. Fraser, Y. Umuroglu, G. Gambardella, M. Blott, P. Leong, M. Jahre and K. Vissers, "Scaling Binarized Neural Networks on Reconfigurable Logic," in *Proceedings of the 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM '17)*, Stockholm, Sweden, New York: ACM, 2017 pp. 25–30. doi:10.1145/3029580.3029586.

[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, et al., "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, **12**, 2011 pp. 2825–2830. www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf.

[37] H. Jaeger, "Echo State Network," *Scholarpedia*, **2**(9), 2007 2330. doi:10.4249/scholarpedia.2330.