Artificial Intelligence Review
https://doi.org/10.1007/s10462-020-09815-8

Feed-forward versus recurrent architecture and local versus cellular automata distributed representation in reservoir computing for sequence memory learning

Mrwan Margem[1] · **Osman S. Gedik**[2]

Abstract
Reservoir computing based on cellular automata (ReCA) constructs a novel bridge between automata computational theory and recurrent neural networks. ReCA has been trained to solve 5-bit memory tasks. Several methods are proposed to implement the reservoir where the distributed representation of cellular automata (CA) in recurrent architecture could solve the 5-bit tasks with minimum complexity and minimum number of training examples. CA distributed representation in recurrent architecture outperforms the local representation in recurrent architecture (stack reservoir), then echo state networks and feed-forward architecture using local or distributed representation. Extracted features from the reservoir, using the natural diffusion of CA states in the reservoir offers the state-of-the-art results in terms of feature vector length and the required training examples. Another extension is obtained by combining the reservoir CA states using XOR, Binary or Gray operator to produce a single feature vector to reduce the feature space. This method gives promising results, however using the natural diffusion of CA states still outperform. ReCA can be considered to operate around the lower bound of complexity; due to using the elementary CA in the reservoir.

Keywords ReCA · Reservoir computing · Cellular automata · Recurrent architecture · Feed-forward architecture · Distributed representation · Local representation · 5-Bit memory task

✉ Mrwan Margem
m_mrwan@yahoo.com

Osman S. Gedik
gedik@ybu.edu.tr

[1] Department of Control Engineering, College of Electronic Technology-Tripoli, Al-Jaraba Street, 21821 Tripoli, Libya

[2] Department of Computer Engineering, Ankara Yildirim Beyazit University, 06220 Keçiören, Ankara, Turkey

Published online: 12 February 2020

1 Introduction

Many complex real-life systems require processing time-dependent data; therefore, these systems should remember the previous inputs, i.e., they must have a memory.

The feed-forward Artificial Neural Networks (ANNs) cannot be used to simulate these types of systems; except if the feed-forward connections change to recurrent connections (there are feed-back connections), in this case, the new network is called recurrent neural network (RNN).

RNNs are powerful tools for machine learning (ML) with memory Goodfellow et al. (2016), but it is very difficult to train them using traditional methods as gradient descent Bengio et al. (1994). To simplify the training of RNNs; they will be split into two layers: the *untrained* layer which is randomly and sparsely connected recurrent neurons called (a reservoir) and the *trainable* feed-forward ANNs (a read-out layer). This simplification became known as reservoir computing (RC) (Jaeger 2001; Maass et al. 2002).

The high dimensional projection provided by the RNN reservoir can be substituted by cellular automata (CA) evolution as in Yilmaz (2015b) and then was called ReCA (reservoir computing based on cellular automata) Margem and Yilmaz (2016), which greatly simplifies the architecture complexity and makes the computation faster and more transparent for analysis (Margem and Yilmaz 2016; Margem and Gedik 2019).

In this paper, ReCA will be improved by manipulating the reservoir output using CA distributed representation in recurrent and feed-forward architecture to extract the feature space; this space will be used in the regressor to predict the output of 5 bit memory task (well-known benchmark task in RNNs field). the ordinary 5-bit task is also modified by creating a test set to adapt the machine learning principles for testing the models. In addition, the local representation has been also used in recurrent and feed-forward architecture to solve such tasks, and compare their results with CA distributed representation results. Then, our results have been compared with the state-of-the-art results of these tasks.

This paper is structured as follows: The following Sects. 1.1, 1.2 and 1.3 are quick review of RC, CA and reservoir computing based on cellular automata. Section 2 presents ReCA implementation. The problem formulation is described in Sect. 3. Section 4 is for the experimental setup. The results and discussion are presented in Sect. 5. Finally, the conclusion and future work are in Sect. 6.

1.1 Reservoir computing

RNNs are connectionist computational systems in which the information is processed in the states of their hidden units through the sequence of time. This makes them suitable to solve time-dependent tasks (problems) such as speech recognition, language modeling, financial data analysis, etc. RNNs are universal approximators of dynamical systems Funahashi and Nakamura (1993) and can simulate Turing machines Siegelmann and Sontag (1995), but it is very difficult to train them, due to convergence problems Doya (1992) and the difficulties of finding optimal representations for long-term memory learning (Bengio et al. 1994; Hochreiter and Schmidhuber 1997). Reservoir Computing (RC) is an approach that could avoid these difficulties by splitting the network into two layers: the first one is an *untrained* layer which is randomly and sparsely connected recurrent neurons called (reservoir) whilst the second layer is a *trainable* feed-forward ANNs (read-out layer). RC paradigm was

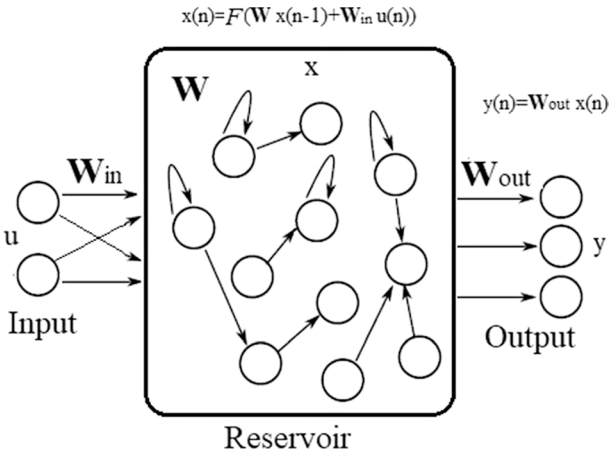


Fig. 1 Reservoir computing paradigm: The input **u** is projected randomly in an RNN (reservoir) and the evolution states of the reservoir **x**(n) are harvested to calculate the output weight matrix $\mathbf{W}_{out}$ which is then used to predict the output **y**

independently introduced to RNNs field by H. Jaeger (Echo State Networks (ESNs)) Jaeger (2001) and by W. Maass [Liquid State Machines (LSMs)] Maass et al. (2002).

Mathematically speaking; in RC model as shown in Fig. 1, given K examples for training. The target is to find M-dimensional output **y**(n) as a function of L-dimensional input **u**(n), where n is the time step number n = 1, 2, ..., T and T is the input sequence length. Using ESNs as an example Lukoševičius and Jaeger (2009); the update equation of the reservoir states $x(n)$ is as follows:

$$\mathbf{x}(n) = F(\mathbf{W}.\mathbf{x}(n-1) + \mathbf{W}_{in}.\mathbf{u}(n)), \tag{1}$$

where $\mathbf{x}(n)$ is the reservoir state vector at time step n, $\mathbf{x}(0) = \mathbf{0}$, F is the reservoir activation function, $\mathbf{W}_{in}$ is an $N \times L$ matrix of random projection of the input to the reservoir, $\mathbf{W}$ is an $N \times N$ matrix which describes the internal connections and weights of the reservoir neurons and N is the reservoir size (the number of reservoir neurons).

An important parameter in the reservoir is the spectral radius $\rho(\mathbf{W})$ of the reservoir matrix (the maximum eigenvalue of $\mathbf{W}$), which should be less than 1 to ensure the *echo state property*[1] in the reservoir to acquire a proper functioning of randomly connected in the reservoir Lukoševičius and Jaeger (2009). Also, it is preferred to operate ESNs at the edge of chaos to obtain a high computational power in the reservoir (Bertschinger and Natschläger 2004; Legenstein and Maass 2007).

Due to the rich dynamics in the reservoir, the output can be simply expressed as the weighted sum of the reservoir states as in Eq. (2):

$$\mathbf{y}(n) = \mathbf{W}_{out}.\mathbf{x}(n), \tag{2}$$

where $\mathbf{W}_{out}$ is the $(M \times N)$-dimensional output weight matrix which is only trained whilst $\mathbf{W}_{in}$ and $\mathbf{W}$ are created randomly and are not changed during training. However, in traditional RNNs, the training methods adapt all the weights ($\mathbf{W}_{in}$, $\mathbf{W}$, and $\mathbf{W}_{out}$).

$\mathbf{W}_{out}$ can be obtained directly from Eq. (3):

[1] The network should gradually lose information that has been received from previous states and inputs.

$$\mathbf{W}_{out} = \mathbf{Y}_{train}.(\mathbf{X}_{train})^{\dagger}, \tag{3}$$

where $(\mathbf{X}_{train})^{\dagger}$ is the pseudo-inverse of $\mathbf{X}_{train}$ which is the state collection matrix from all training examples and $\mathbf{Y}_{train}$ is the collection output matrix from all training examples (the target). Thus, the most expensive calculation in training stage of the RC model is to find the pseudo-inverse of $\mathbf{X}_{train}$.

Finally, to predict new output values $\hat{\mathbf{y}}$, other than the output $\mathbf{y}$ of the K training examples that have been used to find the output weight matrix $\mathbf{W}_{out}$; Eq. (4) is used as follows:

$$\hat{\mathbf{y}}(n) = \mathbf{W}_{out}.\hat{\mathbf{x}}(n), \tag{4}$$

where $\hat{\mathbf{x}}(n)$ is the new reservoir states that are obtained from the new input. For more details, Lukoševičius (2012) presents practical techniques and recommendations for applying ESNs.

Some of high dimensional dynamic physical systems can be considered as a computational model Adamatzky (2001), which led the researchers to utilize these systems as a hardware reservoir; starting by a water waves in a bucket Fernando and Sojakka (2003), the genetic regulatory network of the *Escherichia coli* bacterium (Dai 2004; Jones et al. 2007), optoelectronics (Paquot et al. 2012; Larger et al. 2012; Ortín et al. 2015), random Boolean networks (RBNs) Snyder et al. (2013), and finally using carbon nanotubes as a reservoir Dale et al. (2016, 2017).

1.2 Cellular automata

Cellular automata (CA) were originally studied by J. von Neuman in the 1950s and were introduced as a mathematical model for the self-reproducing phenomena in Von Neumann (1951). CA are composed of simple computational nodes that are combined to produce a network that can be evolved to extremely complicated behavior and then can perform a larger computational task.

Precisely, CA are a discrete (in time and space) parallel computational model composing of a lattice of cells. The cell state evolves in time according to a certain transition function (rule), depending on its current state and the state of neighbors Wolfram (2002). The states of all cells in the array are updated simultaneously. Each update is called an iteration and the total number of iterations is denoted by I. In this paper, elementary cellular automata (ECA) will only be used, which is a 1-dimensional CA, i.e., array of cells, each cell has only two states 1 or 0, thus the number of states $S = 2$. The new cell state, after updating, depends on its previous state and the previous state of the left and right neighbors, hence the number of used cells for any state updating $K = 3$.

In view of, ECA evolution rule space there are $S^{(S^K)} = 2^{(2^3)} = 256$ different rules Li and Packard (1990). But these 256 rules have a lot of equivalences. Consequently, a similar behavior can be classified in a set of rules, thus ECA rule space was reduced to 88 equivalent sets of rules Martnez (2013). Figure 2 shows the eight (2^3) possible combinations and the Boolean expression that represent rule 150 as an example of ECA rules which have got their numbers by converting the binary string of their possible combinations to a decimal number, i.e., $(10010110)_2$ is $(150)_{10}$ as illustrated in Fig. 2. The boundaries of CA grid are either null or periodic; for ECA in the null boundary the two last cells in the grid row are fixed to 0, but for a periodic boundary, the grid is folded, i.e., the left-most cell becomes the neighbor of the right-most one and vice versa.

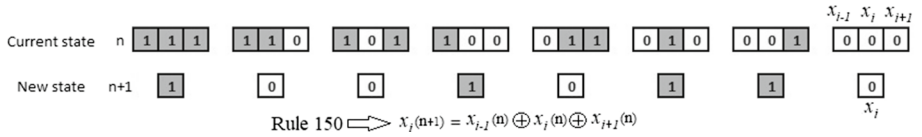


Fig. 2 ECA Rule 150: All possible combination for one iteration of the center cell x_i for rule 150 and its Boolean expression. The binary string of all possible combinations $(10010110)_2$ produces the rule number $(150)_{10}$

Starting by a random initial vector, the evolution of CA states obeys different behaviors which can be categorized into 4 classes (Wolfram classes) Wolfram (2002). Class I (Uniform): the initial states evolve to a stable (constant) state, e.g., rule 235, 255 all the cells become 1's and rule 0, 32 all the cells become 0's; Class II (Periodic): CA states evolve into static or oscillating states, e.g., rule 42, 162; Class III (Chaotic): CA states evolve chaotically, e.g., rule 90, 105, 150 and Class IV (Complex or Edge of Chaos): the initial vector evolves in an unpredictable manner (complex behaviors), e.g., rule 106, 110. The fourth class rules have very high computational representation (universal computation) (Martnez et al. 2013b, c; Cook 2004). The most interesting rules are in the Classes III and IV because of their applications; the chaotic rules are used for random number generators and cryptography, also due to the capacity of class IV rules to contain complex systems; it has been used in the simulation of particles, waves, or gliders (Adamatzky 2001; Wolfram 2002; Langton 1986; Toffoli and Margolus 1987). In this paper, we will show that the periodic rules of class II, despite their simplicity, they can also be used in machine learning, and even they give the best results in generalized 5-bit memory task.

Conventional CA are memory-less, i.e., a new state of a cell depends only on the neighbors at the previous time step. A relatively new class of CA is Elementary Cellular Automata with Memory (ECAM), which is an ECA composed with an added memory function. This new class extended the domain of rules based on the ECA domain. In ECAM the basic ECA rule should be selected and then it is composed with a memory function. Therefore, the memory function will determine if the original ECA rule conserves the same original Wolframs class or not. Following this principle, the ECAM rules can be classified as follows: ***Strong***: The memory functions are unable to change the rule class, ***Moderate***: The memory function can transform the rule to another class and conserve the original class as well, and ***Weak***: The memory function transforms the rule class to another class quickly (Martnez 2013; Martnez et al. 2013a; Alonso-Sanz and Martin 2006).

1.3 Reservoir computing based on cellular automata

The first appearance of reservoir computing based on Boolean networks was in 2013 using random Boolean networks (RBNs) Snyder et al. (2013), in 2014, the RBNs have been replaced by cellular automata CA in Yilmaz (2014), which are a special case of RBNs with a regular structure, then such framework of CA has been used in detail for connectionist machine intelligence Yilmaz (2015b) and symbolic computation Yilmaz (2015a, c). In previous studies Yilmaz (2014, 2015b), the initial (first time step) L_{in}-dimensional input is randomly permuted R times to produce the $(L_{in}.R)$-dimensional vector which is projected into the reservoir as initial state X_0 and evolved up to I iterations using a certain ECA rule. The last CA iteration state at first time step is *added* to the second time step

(*Lin.R*)-dimensional initial state X_1 using *normalized* addition[2] to create the recurrent connection which memorizes the history of the first and second time step inputs, and so on up to the last time step. The obtained state vectors from the CA reservoir at every time step are used to predict the corresponding time step output Yilmaz (2014, 2015b). In Margem and Yilmaz (2016), another type of CA reservoir was designed including the use of zero array buffers instead of the input random projection in Yilmaz (2014, 2015b), this model was called ReCA. Using zero buffers in ReCA enhances the results to solve all pathological sequence tasks. These tasks have been introduced in 1997 by Hochreiter and Schmidhuber to examine Long short-term memory (LSTM) method in Hochreiter and Schmidhuber (1997), then they have been widely used in RNNs literature. In 2017, RC based on CA was used by applying Non-Uniform ECA (using 2 rules horizontally) Nichele and Gundersen (2017) or by repeating the entire model in series (deep reservoir) Nichele and Molund (2017). These two methods were tested using only one type of pathological sequence tasks (5-bit memory task), which is a binary dataset. Also in 2017, the RC based on CA was tested using a non-binary dataset by applying a multilayer CA in the reservoir; by using 2 rules vertically in McDonald (2017), the first rule is for projection and the other is for memorization. ReCA, in software and hardware implementation, was tested using a pattern recognition task of handwritten numbers (MNIST), which is a time-independent task Morn et al. (2018). Finally, in 2019, ReCA in Margem and Yilmaz (2016) was improved by using various methods to extract the features from the CA reservoir; these methods have been tested using all pathological sequence tasks, the results have shown a significant reduction in the model complexity Margem and Gedik (2019). In the last studies for ReCA (Margem and Yilmaz 2016; Margem and Gedik 2019) the additive CA rules 90 and 150 have only been used in the reservoir, but in this paper all ECA rules has been checked to solve the 5-bit memory task training set (32 examples) as in all last literature (Jaeger 2012; Margem and Yilmaz 2016; Nichele and Gundersen 2017; Nichele and Molund 2017; Margem and Gedik 2019). Here, we also generalize our work by dividing the 32 examples of the ordinary 5-bit memory task into two sets (training and testing) to adapt the machine learning concepts where the model is trained by training set and then should be tested by using new examples other than training examples, which are called testing set.

2 ReCA implementation

In ReCA framework of Fig. 3, the cellular automata are used in the reservoir instead of real-valued recurrent neural networks in echo state networks of Fig. 1. Cellular automata provide powerful enough computation and rich enough representation Margem and Yilmaz (2016) with low complexity Margem and Gedik (2019) to be used to solve hard sequence tasks. Figure 3 shows the algorithmic flow of ReCA consisting of three stages: encoding, reservoir, and read-out. The encoding stage expands the input using zero buffers to produce the initial states of CA reservoir. These states are evolved in the reservoir using a certain ECA rule for I iterations to extract the input statistics; which are represented in a distributed manner as in RNNs Yilmaz (2015b); Margem and Yilmaz (2016). The evolved states are used as a feature space for linear regression in the read-out stage to find the output weight matrix $\mathbf{W}_{out}$ which is then used to predict the output $\mathbf{y}$.

[2] In normalized addition; $0 + 0 = 0, 1 + 1 = 1$ and for $1 + 0$ or $0 + 1$ the result is decided randomly 0 or 1.

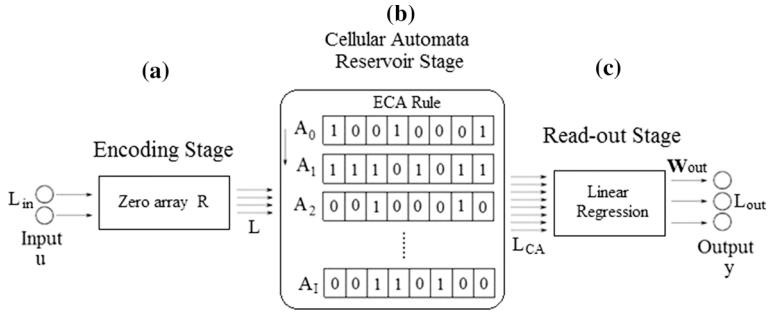


Fig. 3 ReCA framework: **a** Encoding stage: Two *R*-dimensional buffers are added to both sides of the original input **u**, **b** CA reservoir stage: The L-dimensional vector A_0 which is obtained from the encoding stage is used in the reservoir as initial state for ECA rule, then it evolves *I* iterations, and **c** Read-out stage: The evolved states in the reservoir will be used as a feature space for linear regression to find the output weight matrix $\mathbf{W}_{out}$ which is then used to predict the output **y**

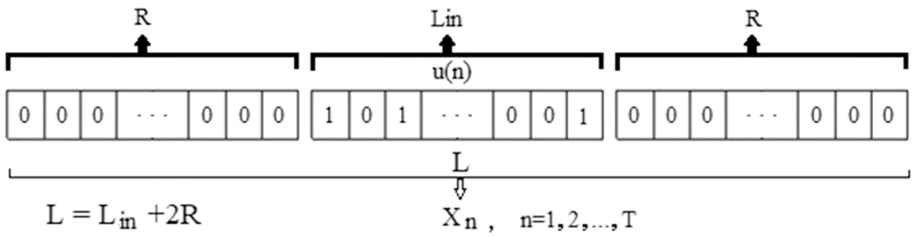


Fig. 4 Encoding stages: Two R-dimensional zero vectors (buffers) are added to both sides of the original input **u**(n), to produce the L-dimensional initial state X_n that will be evolved using ECA rule in the reservoir stage

ReCA has been used in recurrent architecture to deal with sequence problems, but it can also be used in feed-forward architecture to solve sequence tasks Yilmaz (2015b) or time-independent tasks McDonald (2017) and Morn et al. (2018). Also, Sect. 2.4 describes how ReCA can be used in feedforward architecture to solve time-dependent tasks.

2.1 Encoding stage

The original input **u** with dimension of L_{in} is expanded by adding an array of R zeros to both sides of the input **u**, then the expanded input becomes with a length of $L = L_{in} + 2R$ as shown in Figs. 3a and 4; this expansion is to allow the natural diffusion of CA states into the reservoir (please, see Fig. 7). To hold all ECA sequence representations for all time steps in the reservoir; the buffer dimension R should be equal to $I \times T$ where I is the number of iterations and T is the input sequence length. Due to the reach dynamics in the reservoir, the dimension of the zero array R can be reduced to $R = f.(I \times T)$, where f is the expansion ratio; $f \in [0, 1]$ to reduce the model complexity. There are other methods have been used to process the original inputs **u** before the reservoir as expand each input bit to multiple bits to reduce the interference between non-zeros in the reservoir; for details please (see Margem and Yilmaz 2016; Margem and Gedik 2019). Another ECA rule can also be applied to the original input **u** before adding the zero buffers (Multilayer CA expansion) to increase the effectiveness and efficiency of the model Margem and Yilmaz (2016).

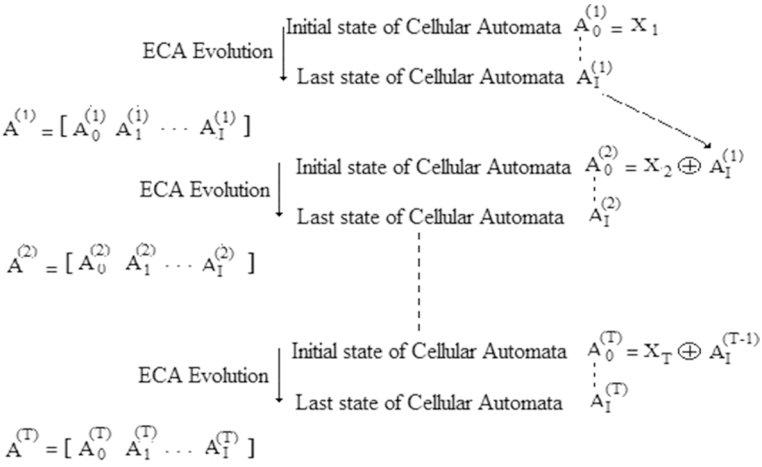


Fig. 5 Reservoir stage: The evolution of CA states $A_i(n)$ (i from 1 to I iterations) to produce the feature vector $A^{(n)}$ (with dimension of $L_{CA} = L.I$) that is used in linear regression to predict the output $\mathbf{y}(n)$. Using bit-wise XOR operation to insert the input X_n (n from 2 to T) into the reservoir to create the initial state according to every time step

In this paper, we confine to add the buffers R to the input, the other methods can be added in the future work.

2.2 Reservoir stage

The output of encoding stage X_n vectors (n changes from 1 to T) are mapped into the CA reservoir instead of an ESNs as in Jaeger (2012); to produce an expressive and discriminative space that will be used to predict the output of sequence tasks. The CA reservoir is initialized by the expanded input at first time step X_1 which is evolved by a certain ECA rule to generate $A_i^{(1)}$ vectors where the subscript i denotes to the iteration number which changes from 1 to I and the superscript (1) denotes to the time step which changes from 1 to T (T is the input sequence length), then they are concatenated to obtain a single L_{CA}-dimensional vector $A^{(1)}$ ($L_{CA} = L.I$) which is used as features of linear regression to predict the output at first time step $\mathbf{y}(1)$ as depicted in Fig. 5. To create the initial state at second time step $A_0^{(2)}$; the modified input at second time step X_2 is combined with the last iteration from the previous time step $A_I^{(1)}$ using an insertion function[3] which is in our case a bit-wise XOR operation, i.e., $A_0^{(2)} = X_2 \oplus A_I^{(1)}$. These steps are repeated until the last time step $n = T$ to obtain the feature vector $A^{(T)}$ which memorizes the history of all input sequences, the reservoir stage details are demonstrated in Fig. 5.

Instead of using XOR operation, there is another choice for the insertion function where the second input with a dimension of L_{in} can be directly inserted into the reservoir by *overwriting* the middle cells in $A_I^{(1)}$ with this input to produce the second initial state $A_0^{(2)}$. The overwriting (deleting) of $A_I^{(1)}$ middle cells does not lose the first time

[3] Insertion function is to insert a new input time step into the reservoir.

step information, because the two evolution sides (the propagation in the buffers) of $A_I^{(1)}$ are still conserving the information of the first time step. Therefore, to ensure that, there is no any loss of information; the number of iterations I should be large enough to propagate the whole input information into the buffers depending on the task nature (the size and information position) and ECA rule. Then, the same steps will be repeated up to the last time step to produce the vector $A^{(T)}$ which also memorizes the history of all input sequences. In results section, this option will be called **Overwrite** and using XOR operation will be called **Normal**.

2.3 Read-out stage

For Each time step n, there is an output $\mathbf{y}(n)$ with dimension of L_{out}. All these outputs can be combined in a single output matrix $\mathbf{Y}$ of size $L_{out} \times T$. As mentioned in Sect. 1.1; the output $\mathbf{Y}$ can be expressed as a linear combination of the reservoir states as follows:

$$\mathbf{Y} = \mathbf{W}_{out}.\mathbf{A} \tag{5}$$

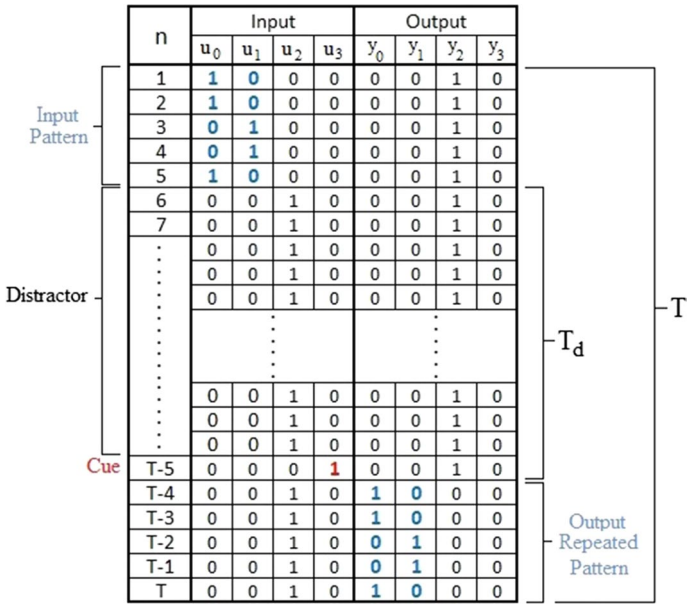
where $\mathbf{W}_{out}$ is of size $L_{out} \times I.L$, and matrix $\mathbf{A}$ is the combination of all evolved states at whole time steps in the reservoir of size $I.L \times T$ as shown in Eq. (6):

$$\mathbf{A} = [A^{(1)}A^{(2)}...A^{(T)}] \tag{6}$$

2.4 Feed-forward architecture

Sequential data can also be handled using feed-forward architectures by using a portion of the input to predict an output as in sliding window methods Dietterich (2002) or using the whole input sequence to predict all the output as in Yilmaz (2015b). In our work the second method will be used via two ways: **a**—All the flattened input sequence is projected to the CA reservoir at once, or **b**—The input of each time step is separately projected to the reservoir then the obtained CA evolution states of all time steps are concatenated together to produce the feature vector that will be used to predict the whole output. In the results section, the first method will be called **All-ff** and the second one called **Each-ff**.

In feed-forward case the buffer dimension R should be equal to I to hold all CA sequence representations instead of $R = I \times T$ for recurrent architecture in Sect. 2.1, thus the feature dimension in feed-forward architecture is smaller than it in recurrent architecture. But, the advantage of recurrent architecture over the feed-forward is the capability to keep a fixed size representation by adjusting the expansion ratio f to a certain value for the feature vector which is similar to the reservoir size N (number of neurons) in echo state networks ESNs. The CA feed-forward architecture is very similar to Extreme Learning Machines ELMs (Huang et al. 2004; Huang et al. 2011), where the random connections in ELM are replaced by the CA reservoir.



n	Input				Output			
	u_0	u_1	u_2	u_3	y_0	y_1	y_2	y_3
1	1	0	0	0	0	0	1	0
2	1	0	0	0	0	0	1	0
3	0	1	0	0	0	0	1	0
4	0	1	0	0	0	0	1	0
5	1	0	0	0	0	0	1	0
6	0	0	1	0	0	0	1	0
7	0	0	1	0	0	0	1	0
⋮	0	0	1	0	0	0	1	0
	0	0	1	0	0	0	1	0
	0	0	1	0	0	0	1	0
	⋮				⋮			
	0	0	1	0	0	0	1	0
	0	0	1	0	0	0	1	0
	0	0	1	0	0	0	1	0
T-5	0	0	0	1	0	0	1	0
T-4	0	0	1	0	1	0	0	0
T-3	0	0	1	0	1	0	0	0
T-2	0	0	1	0	0	1	0	0
T-1	0	0	1	0	0	1	0	0
T	0	0	1	0	1	0	0	0

Fig. 6 An example of 5-bit task: The input and output length $L_{in} = L_{out} = 4$ bits. The first 5-time steps are the 2-dimensional input memory pattern and the last 5-time steps are the main output which is a repeated (memorized) input pattern (Blue numbers), but we have to note that the whole output bits (4 bits) for all time steps (T) should be predicted. The distractor input [0 0 1 0] is at the middle of the task for T_d time steps, and the last time step of it is a cue signal (Red one) as a mark to repeat the input pattern in the output. The total sequence length of the task is $T = T_d + 10$. (Color figure online)

3 Problem formulation

The ReCA framework has been tested for long short-term memory capability using 5-bit memory task, which is a part of several pathological synthetic tasks that has been created in Hochreiter and Schmidhuber (1997) with minor modifications in Martens and Sutskever (2011). These tasks offer long-term dependencies and are effectively impossible to solve using gradient descent Hochreiter et al. (2001).

3.1 Benchmark tasks

The 5-bit memory task is widely used in RNNs literature, (e.g., Martens and Sutskever 2011; Jaeger 2012; Pascanu et al. 2013; Yilmaz 2015b; Margem and Yilmaz 2016; Nichele and Gundersen 2017; Nichele and Molund 2017; and very recently Margem and Gedik 2019). This task is one of the hardest tasks for ESNs in Jaeger (2012), also as reported in Yilmaz (2015b) and Nichele and Molund (2017), it is problematic to solve the 5-bit memory task using feed-forward architectures. But, as it will be shown later in Sect. 5.3, it is possible to solve this task using feed-forward architecture with some limitations using ECA and even without using it. The weakness of using this task is that; due to its small number of examples (only 32 examples), the whole examples are used to train the model, and then the model is tested using the same 32 examples and finding the error, this approach has

been used in Yilmaz (2015b), Margem and Yilmaz (2016), Margem and Gedik (2019), Nichele and Gundersen (2017), Nichele and Molund (2017) and Jaeger (2012), thus the proposed models are not general as it will be explained later in Sect. 3.1.2.

3.1.1 5-Bit memory task

The 5-bit task is a binary dataset in which the input and output length are 4 bits $L_{in} = L_{out} = 4$ bits. The information in this task is a memory pattern which is the sequence of the first two bits for five-time steps where one of these two bits is randomly set to 1; the blue pattern in Fig. 6. Thus, there are $2^5 = 32$ possible patterns, thats why this task is named the 5-bit task. Then, at the sixth time step, the input memory pattern is followed by a distractor [0 0 1 0] for period T_d that expands the task dimension and adds irrelevant data to increase the task difficulty. Finally, the initial memory pattern should be repeated in the output after arriving a *cue* signal in the fourth input bit at the last time step of the distractor. Thus, the total sequence length of this task $T = T_d + 10$. The fourth output bit is always zero, so it should be dropped, but we kept it; because it is included in the original task. The details of 5-bit task are demonstrated in Fig. 6.

3.1.2 Generalized 5-bit memory task

Due to, the weakness of using 5-bit task, which is mentioned in Sect. 3.1, we can say that, the model is only designed for training set and there is no guarantee for the model to solve other examples that have the same behavior of the training examples, i.e., the model is not general which is a disadvantage for any model. Therefore, to generalize the model and respect the machine learning principles for model evaluation, the dataset should be divided into two sets that generally have different examples; training and testing set. The training set is used for training to calculate the model parameters, and then the model is tested using the input of testing examples to predict their output and finding the error between the given output in testing examples and predicted one to evaluate the model. The small dataset is a challenge in machine learning, but nevertheless, we will divide the 32 examples into two sets and even we will search the smallest number of examples that can efficiently train the model to give zero test error. Thus, all the 32 examples of the original 5-bit task are randomly shuffled, then the first N_{train} examples are selected for training and the rest examples $N_{test} = 32 - N_{train}$ are used for model testing.

4 Experiments

The ReCA framework will be tested for long short-term memory capability using 5-bit memory tasks. In training stage, the features will be extracted from the CA evolution states in the reservoir with their associated outputs (Target) that are used for training to compute the linear regression weights (output weight matrix parameters) $\mathbf{W}_{out}$. Therefore, we should initially figure out the features (A_{train}), that will be used for training.

4.1 Feature space

There are multiple options to extract the features from ECA evolution of states called CA_{out} matrix (as shown in Fig. 7) that have been obtained from the reservoir in a certain

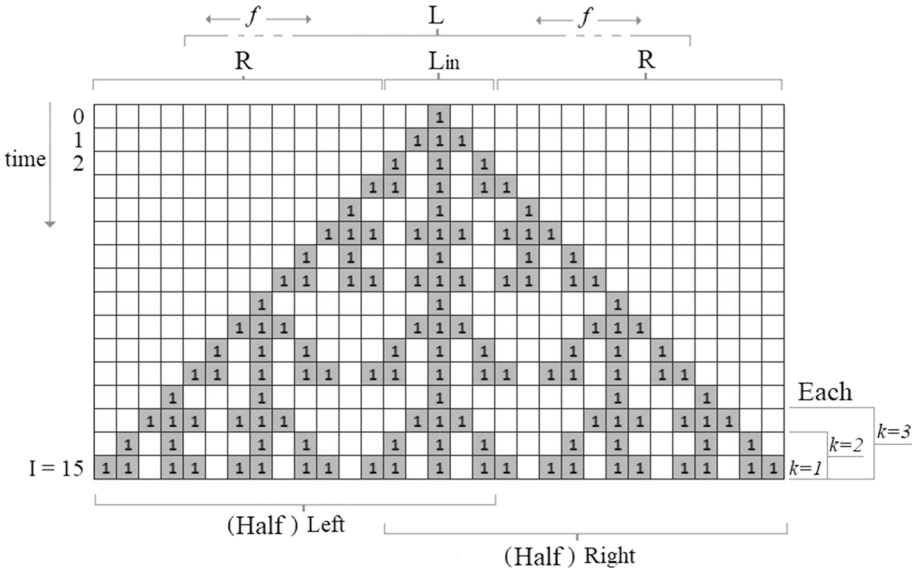


Fig. 7 Feature extraction from the space-time diagram of the CA evolution states in the reservoir at a certain time step n, which is also called $\mathbf{CA}_{out}$ matrix. As an example, ECA rule 150 is used with a single 1 initial state and 15 iterations. Only k iterations can be used as features in **Each** option and/or using only the **Half** (Right or left) side of $\mathbf{CA}_{out}$ matrix as features, and/or using the features after reducing the columns of $\mathbf{CA}_{out}$ matrix by selecting expansion ratio $f < 1$. *Note*: White squares represent the zeros

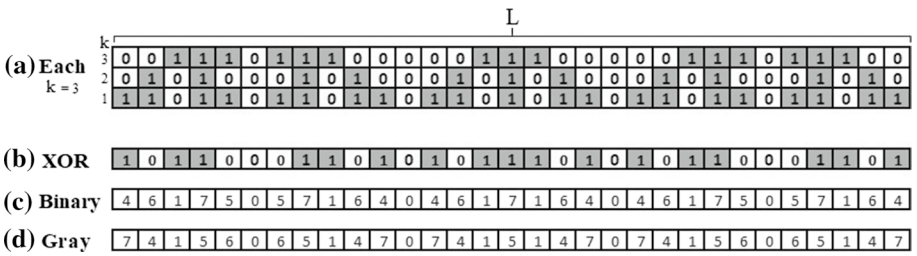


Fig. 8 Types of feature extraction **a** Using 3 iterations out of 15 from $\mathbf{CA}_{out}$ matrix in Fig. 7; i.e., $k = 3$, **b** Using bit-wise XOR operation for all columns of the matrix in part **a** to produce the feature vector, **c** Converting the binary value of each column of the matrix in part **a** to decimal number, and **d** Using the Gray code instead of the binary code in (**c**)

time step n. All evolution states can be directly used as features; hence the dimension of A_{train} will be $I \times L$ after the concatenation of all rows of CA_{out} matrix as demonstrated in Fig. 5, but due to the large space of this option, it will not be used; because in this paper, we are seeking minimum complexity in space and time. Other options will be adopted as follows:

1. **Each**: For each time step n; k states of CA evolution are only selected to be used as a feature space as demonstrated in Fig. 7. Figure 8a depicts an example of **Each** case where only three states $k = 3$ have been used from the CA_{out} matrix of Fig. 7 to produce

a feature matrix with size of $3 \times L$, thus the feature vector A dimension is $3L$. In general, the feature vector dimension $L_{CA} = (L_{in} + 2R).k$.

2. Expansion ratio f: In **Each** option, the feature space reduction was by selecting k rows from CA_{out} matrix that has I rows (iterations). Also, the feature space reduction can be reached by selecting only some columns from CA_{out} matrix. In other words, the feature space reduction, in this case, is done by reducing the two buffers dimension. Hence, the new dimension of the zero buffer will be $R = f.(I \times T)$, where the expansion ratio $f \in [0, 1]$ as shown in Fig. 7.
3. **Half**: The dimension of the feature space can be reduced to the half by using only one side (*right* or *left*) of CA_{out} matrix as described in Fig. 7, i.e., the feature dimension $L_{CA} = (L_{in} + R).k$.
4. **XOR**: The matrix CA_{out} can be reduced to one row; using bit-wise XOR operation for every column of CA_{out} as shown in Fig. 8b where $k = 3$. Thus, the dimension $L_{CA} = (L_{in} + 2R) = L$. This case can be considered as an ECA with memory (ECAM); because there is a memory function (XOR operation) added to ECA, but the difference in our case is that this function is only applied after the last iteration, whilst in ECAM the function is applied in all iterations Martnez et al. (2013a).
5. **Binary**: Every column vector in CA_{out} matrix can be represented by a decimal number according to its binary value with the least significant bit (LSB) at the first row[4]. Figure 8c presents the **Binary** option where $k = 3$ in Fig. 8a, also it can be used for the whole columns of CA_{out} matrix in Fig. 7, the dimension $L_{CA} = L$. Converting the features using binary code has also been used in McDonald (2017) utilizing two ECA rules in the reservoir.
6. **Gray**: The binary code can be replaced by Gray code (Fig. 8d) which is normally used to decrease the noise effect on the binary bits in digital communications and binary counters.

Notes: **1-** Due to the rich dynamics that are provided by the CA reservoir; the first *three* options (**Each**, **Half** and *f*) can be used together with any option of the last three cases (**XOR**, **Binary**, and **Gray**) to get a significant complexity reduction as will be shown later in the results. Also, both insertion functions **Normal** or **Overwrite** can be used with **XOR**, **Binary**, and **Gray** options. For simplicity, **Normal** has only been used in this paper with **XOR**, **Binary** and **Gray** options, the **Overwrite** function will be used in future work. **2-** For **XOR**, **Binary** and **Gray** options, k should be greater than or equal 2 ($k \geq 2$); because if $k = 1$ they will be as individual **Normal** or **Overwrite** methods. The advantage of these three options is the low dimension of feature vector L_{CA} even if k is large still $L_{CA} = L$, i.e., it is independent on the value of k as in **Normal** or **Overwrite** methods where $L_{CA} = L.k$.

4.2 Training and testing stage

After preparing the training features A_{train} as in Sect. 4.1; they will be used to compute the linear regression weights ($\mathbf{W}_{out}$) via pseudo-inverse as illustrated in Eq. (7):

[4] The LSB can also be at the last row.

$$\mathbf{W}_{out} = \mathbf{y}_{train}.(A_{train})^{\dagger} \tag{7}$$

where $\mathbf{y}_{train}$ is the output of training set (target) and $(A_{train})^{\dagger}$ is the pseudo-inverse of A_{train}.

In testing stage, the predicted output from test set $\hat{\mathbf{y}}_{test}$ can be obtained using Eq. (8):

$$\hat{\mathbf{y}}_{test} = \mathbf{W}_{out}.A_{test}. \tag{8}$$

where A_{test} is the harvested features from the N_{test} examples of test set using the various options in Sect. 4.1.

Note: In the ordinary 5-bit task in Sect. 3.1.1, there is no test set. Therefore, Eq. (8) is used to compute the predicted output of all 32 examples of the training set as follows: $\hat{\mathbf{y}}_{train} = \mathbf{W}_{out}.A_{train}$.

4.3 Model evaluation

The dataset of 5-bit task is binary, so that each bit of the predicted output $\hat{y}_i$ should be binarized to $\hat{y}_{(b)i}$ as follows:

$$\hat{y}_{(b)i} = \begin{cases} 0 \;\; if \;\; \hat{y}_i < 0.5 \\ 1 \;\; if \;\; \hat{y}_i \geq 0.5 \end{cases} \tag{9}$$

The binarized output will be used to evaluate the model by computing the errors as follows: $training\ error = |\hat{\mathbf{y}}_{train} - target|$ for ordinary 5-bit task, and $testing\ error = |\hat{\mathbf{y}}_{(b)test} - \mathbf{y}_{test}|$ for generalized 5-bit task. This error represents the number of false bits over all relevant time steps in output bits. The goal of our experiments is to achieve zero error with minimal complexity. Hence, the parameters I (CA evolution iterations), k (the number of selected iterations in **Each** option) and f (the expansion ratio) should be tuned to their minimum values; in order to obtain the shortest length of the feature vector L_{CA}, using the whole equivalent sets of ECA rules that are listed in Martnez (2013) excluding the 8 sets of Class I rules[5]; Hence, from Martnez (2013); the ECA rules that will be used in this study are 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 19, 23, 24, 25, 26, 27, 28, 29, 33, 34, 35, 36, 37, 38, 42, 43, 44, 46, 50, 51, 56, 57, 58, 62, 72, 73, 74, 76, 77, 78, 94, 104, 108, 130, 132, 134, 138, 140, 142, 152, 154, 156, 162, 164, 170, 172, 178, 184, 200, 204 and 232 from class II and 18, 22, 30, 45, 60, 90, 105, 122, 126, 146 and 150 from class III and 41, 54, 106 and 110 from class IV, as well as rule 165 (the conjugate of rule 90) and rule 102 (the reflection of rule 60) Wolfram (1994). Thus, the total number of used rules in our experiments becomes 82 rules rather than the whole 256 ECA rules.

5 Experimental results and discussion

The ReCA framework has been tested using 5-bit task (32 examples for training) as in Jaeger (2012), Yilmaz (2015b), Margem and Yilmaz (2016), Nichele and Gundersen (2017), Nichele and Molund (2017) and Margem and Gedik (2019), and using generalized 5-bit task (32 examples for training and testing) which is created in this paper to follow the

[5] Because, the evolution of class I rules vanishes after the first iteration in 5-bit task, due to the single nonzero in its input at each time step as shown in Fig. 6.

Table 1 ECA rules that achieve zero error in 5-bit task using **Normal** and **Overwrite** methods with parameters (I, k, f) and feature vector dimension L_{CA} where I is the number of all CA iterations in the reservoir, k is the number of selected iterations that will only be used for training in **Each** option, and f is the expansion ratio

Parameters (I, k, f)	Normal		Overwrite	
	Rules	L_{CA} (bits)	Rules	L_{CA} (bits)
(3, 3, 1)	106, *30, 45, 105, 165, 150*, 2, 10, 34, 38, 42, 46, 56, 74, 130, 138, 162, 170, 184.	3792	106, *30, 45, 90, 102, 105, 165, 150* , 2, 10, 11, 15, 34, 38, 42, 43, 46, 56, 74, 130, 138, 154, 162, 170, 184.	3792
(2, 2, 1)	*105, 165,* 15, 34, 38, 42, 162, 170.	1688	*90*, 38, 42, 154, 170.	1688
(2, 1, 1) **Half**	*165*, 15, 34, 42, 162, 170.	424	*90,* 42, 170.	424
(2, 1, 0.5)	15, 34, 42, 162, 170.	424	42, 170.	424
(2, 1, f)	*165*, $f = 0.26$	**224**	*165*, $f = 0.33$	**282**

Underline values for class IV rules, italic values for class III and black for class II. The minimum dimension L_{CA} is in bold numbers. *Note*: For **Half** option, the right side was used in rule 15, any side from the both can be used in rules 90, 165 and the left side was used in the other rules

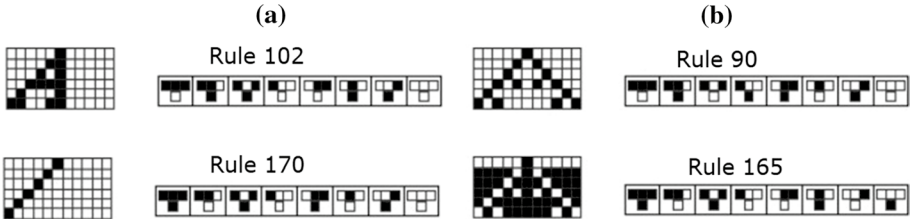


Fig. 9 Ways of information flow in ECA; five iterations space-time diagram of different ECA rules for a single non-zero initial state, with one iteration of a center cell for each rule: **a** One-way, and **b** Two-way. White squares represent the zeros

machine learning concepts for evaluation the models. Finally, our results have been compared with the state-of-the-art results of RC based on CA approaches.

In all experiments; the distractor period $T_d = 200$, hence the total sequence length $T = 210$ time steps. For notation, (I, k, f) indicates the number of total iterations I in the reservoir, k is the number of selected iterations in **Each** option and f is the expansion ratio unless otherwise described. In our work, we are looking for minimum complexity, i.e., minimum value of I, k, and f that will produce a minimum[6] feature vector dimension L_{CA} in bits to achieve zero error. The expression "the model could solve the task" means that the model could predict correctly (zero error) all the four output bits for every time step for the whole 32 examples, i.e., $(4 \times 210 \times 32)$ bits are correctly predicted.

5.1 5-Bit task

All methods with several rules and different Wolfram classes could solve the task and achieve zero error as listed in Table 1 for **Normal** and **Overwrite** methods and Table 2 for **XOR**, **Binary**, and **Gray** options with **Normal** method, i.e., using XOR operator as an insertion function as explained in Sect. 2.2.

Table 1 indicates that; The minimum complexity was attained at (2, 1, 0.26) using rule 165 from class III in **Normal** method where $L_{CA} = 224$ bits, then **Overwrite** method using also rule 165 at (2, 1, 0.33) where $L_{CA} = 282$ bits.

There is an obvious risk of information loss when using **Overwrite** with *two* iterations ($I = 2$) and one-way flow rules[7] (38, 42 and 170) for the input with 4 bits dimension; the last two inputs u_3 and u_4 in Fig. 6 will disappear (be deleted) after using the **Overwrite** insertion function in the reservoir for the next time step. The question is: why did **Overwrite** method provide good results as listed in Table 1? Because, the input memory pattern is in the first two channels u_1 and u_2 thus, they will not disappear (they will be conserved in the buffers R) using *two* iterations and then they can be repeated at the output as demonstrated in Fig. 6. The non-zero in the third input u_3 is not important for the output it is only a distractor, but the cue signal in forth input u_4 is important because after its arrival the output will repeat the input memory pattern, thus the importance here is when does the cue signal arrive?, the cue signal still appears in the CA reservoir state at time-step

[6] Since, L_{in} and T are constants in all experiments, hence the value of L_{CA} depends only on I, k, and f as illustrated in Table 6. Then, the feature vector with dimension L_{CA} will be used in the regressor (read-out stage) to find the pseudo-inverse that implies the most expensive computational part in the model.

[7] One side of propagation as shown in Fig. 9.

Table 2 ECA rules that achieve zero error in 5-bit task using **XOR**, **Binary**, and **Gray** options with parameters (I, k, f) and feature vector dimension L_{CA} where I is the number of all CA iterations in the reservoir, k is the number of selected iterations that will only be used for training in Each option, and f is the expansion ratio

Parameters (I, k, f)	XOR		Binary		Gray	
	Rules	L_{CA} (bits)	Rules	L_{CA} (bits)	Rules	L_{CA} (bits)
(3, 3, 1)	2, 10, 38, 74, 130, 138.	1264	106, *165*, 2, 10, 34, 38, 42, 46, 56, 74, 130, 138, 162, 170, 184.	1264	106, *90, 165*, 2, 10, 34, 38, 42, 46, 56, 74, 130, 138, 162, 170, 184.	1264
(3, 2, 1)	106, 2, 10, 15, 24, 34, 38, 42, 46, 56, 74, 130, 138, 152, 162, 170, 184.	1264	106, *165,* 2, 10, 24, 34, 38, 42, 46, 56, 74, 130, 138, 152, 162, 170, 184.	1264	106, *90, 165,* 2, 10, 15, 24, 34, 38, 42, 46, 56, 74, 130, 138, 152, 162, 170, 184.	1264
(2, 2, 1)	34, 162.	844	*165,* 15, 34, 38, 42, 162, 170.	844	*165,* 34, 38, 42, 162, 170.	844
(2, 2, 1) **Half**, and (2, 2, 0.5)	34, 162.	**424**	*165,*[a] 15, 34, 38, 42, 162, 170.	**424**	34, 38, 42, 162, 170.	**424**

Underline values for class IV rules, italic values for class III, and black for class II. The minimum dimension L_{CA} is in bold numbers. *Note*: For **Half** option, the right side was used in rule 15, any side from the both can be used in rules 165, and the left side was used in the other rules

[a]For rule 165, $f = 0.66$ not 0.5, thus $L_{CA} = 560$ bits

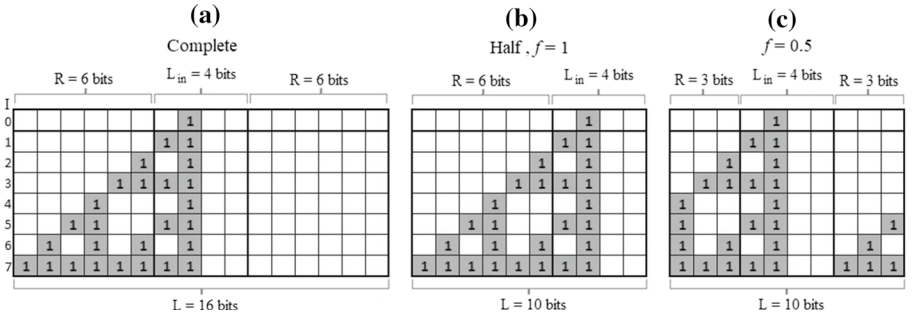


Fig. 10 An example of feature space for a one-way rule: **a** Complete; using the whole space (Both sides and $f = 1$), hence Largest length $L = 16$ bits, **b Half**, $f = 1$; Using only the left side but f should be equal to 1 to conserve the whole information, and **c** $f = 0.5$; Using both sides for $R = 3$ bits instead of 6 bits in (**b**). The first three columns in (**b**) are transferred to last three columns in (**c**) after applying *periodic* boundary, thus (**b**) and (**c**) are identical in the regressor. *Note* White squares represent the zeros

(T-5) (Fig. 6) then it is followed by the input memory pattern in the output. Hence, the **Overwrite** method, with the nature of 5-bit task, could save the necessary task information and provide good results. But, in general, using **Overwrite** method requires the number of iterations $I \geq$ (dimension of L_{in}) / 2 for ECA rules with two-way flow or $I \geq$ (dimension of L_{in}) for ECA rules with one-way flow, Fig. 9 exhibits examples of such rules.

As shown in Table 2 for **XOR**, **Binary** and **Gray**, the minimum complexity was at (2, 2, 0.5) and (2, 2, 1) **Half** where $L_{CA} = 424$ bits; using the rules (15, 34, 38, 42, 162, and 170) from class II and 165 from Class III for only **Half** option.

Note: In the **XOR**, **Binary** and **Gray** options, we did not select $k = 1$; because they will be as **Normal** method when $k = 1$ which was already listed in Table 1.

In the cases where $f = 0.5$ and **Half** with $f = 1$ (not less) are almost the same because we use periodic boundary for *one-way* rules (15, 34, 38, 42, 162 and 170) as represented later in Fig. 11. Hence, the same features are obtained in the 2 cases; because it is just a columns permutation as described in Fig. 10b, c that are obtained from the complete configuration in Fig. 10a, which does not affect the regressor results in the read-out stage. But, the difference is in the rules 165 and 90, where they are *two-way* rules, therefore the obtained information from both sides are different. Moreover, for rule 165 with **Half** option; f can be less than 1 ($f = 0.9$ experimentally in 5-bit task) due to its high distributedness, that has also been proved for rules 90 and 150 in Margem and Yilmaz (2016) and Margem and Gedik (2019).

5.2 Generalized 5-bit task

As mentioned in Sect. 3.1.2, the training and testing sets are selected randomly from the whole 32 examples of 5-bit task. Hence, there are many different samples (examples). Therefore, to get stable estimates of model performance; Monte Carlo cross-validation technique was applied where the algorithm run should be repeated a lot of times[8]Kuhn and Johnson (2013). The run is said to be successful, i.e., zero test error, if the model can find the correct binary value for all the 4 output bits in every time step for whole testing

[8] The experiments have been repeated 100 times, i.e., $N_{trials} = 100$ runs (trials).

Table 3 ECA rules that Succeeded to achieve zero test error using several methods for feature extraction with *two* and *three* examples for training and 100 trials, where I is the number of all CA iterations in the reservoir, k is the number of selected iterations that will only be used for training in **Each** option, and f is the expansion ratio

N_{train}	Method	(I, k, f)	ECA rule	L_{CA} (bits)
2	**Normal**	(4, 1, 0.5)	106, 2, 10, 15, 34, 38, 42, 74, 138, 162, 170, 184.	844
	Overwrite[a]	(3, 1, 0.5)	42, 170.	**634**
	XOR[a]	(3, 2, 0.5)	42, 170.	**634**
	Binary	(4, 2, 0.5)	106, 34, 42, 162, 170.	844
	Gray[b]	(4, 2, 0.5)	34, 162.	844
	Normal	(3, 1, 0.5)	106, 34, 42, 162, 170.	634
3	**Overwrite**	(2, 1, 0.5)	42, 170.	**424**
	XOR, Binary, and Gray	(3, 2, 0.5)	106, 34, 42, 162, 170.	634

Underline values for class IV rules, and black for class II. The minimum dimension L_{CA} is in bold number

[a]Rules 106, 34, and 162 give 1 error for **Overwrite** and 2 errors for **XOR** in 100 trials

[b]Rules 106, 42, and 170 give 1 error for **Gray** in 100 trials

examples, this means $(4 \times 210 \times N_{test})$ bits are correctly predicted for one trial. In order to solve the task, the correct prediction should be for 100 trials.

ReCA could solve the generalized 5-bit task using a very small number of training examples $N_{train} = 2$ or 3 to achieve zero test error for testing examples N_{test} =30 or 29 as listed in Table 3. The minimum complexity was at $(I, k, f) = (2, 1, 0.5)$ using **Overwrite** method with the rules 42 and 170 from class II where $N_{train} = 3$ and $L_{CA} = 424$ bits. But, for $N_{train} = 2$ the task becomes harder. Therefore, the feature vector dimension increases to $L_{CA} = 634$ bits to achieve zero test error at (3, 1, 0.5) using **Overwrite** and **XOR** methods with the rules 42 and 170 from class II as shown in Table 3. Increasing the values of ReCA parameters I, k and/or f makes more rules can achieve zero test error but of course with larger complexity.

5.3 Feed-forward and local representation

In CA feed-forward architecture there are *two* methods to project the sequence input into the reservoir **All-ff** or **Each-ff** as explained in Sect. 2.4. The most of ECA rules except the rules 23, 72, 104, 200 and 232 could solve the 5-bit task using those two methods with a small value of the parameters $(I, k) = (1, 1)$. **All-ff** outperforms in complexity; the dimension of its feature vector is $L_{CA} = 842$ bits while 1260 bits for **Each-ff**. But, for generalized 5-bit task **Each-ff** outperforms, where the model is trained using only 10 examples to achieve zero test error while 11 examples are required for training to obtain zero test error in **All-ff** model.

The *small* number of Iterations I in the previous results (recurrent and feed-forward) and the *shift* property[9] in the most success rules (Fig. 11) lead us to test the local representation

[9] The shift property is especially for a single non-zero initial state as in our case for 5-bit task.

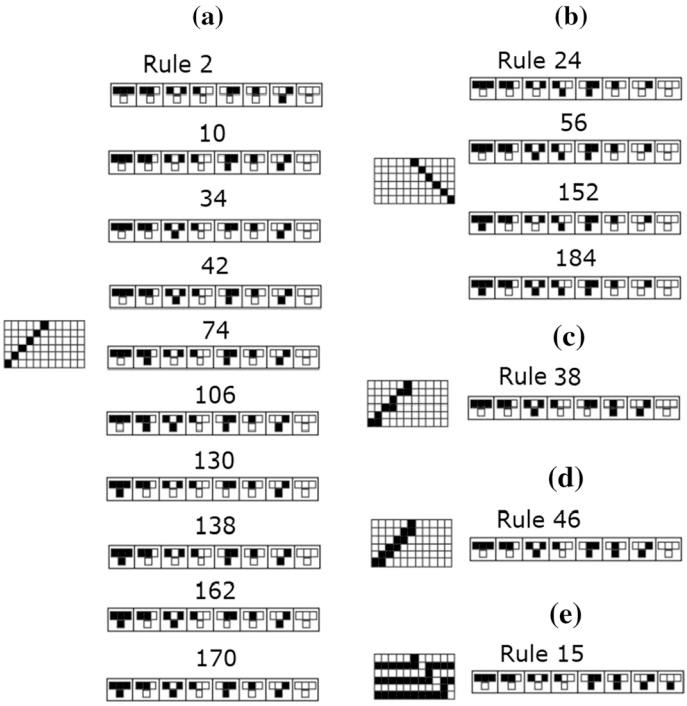


Fig. 11 Shift rules; five iterations space-time diagram of different ECA rules for a single non-zero initial state, with one iteration of a center cell for each rule. **a** left shift, **b** right shift, **c** left shift single and double bits, **d** left shift double bits, and **e** inverted right shift. *Note* The shift is independent on the initial state only for two rules 170 and 15 where their Boolean expressions are $x_i(n+1) = x_{i+1}(n)$ and $x_i(n+1) = \bar{x}_{i-1}(n)$ respectively, where $\bar{x}$ is the inverse of x

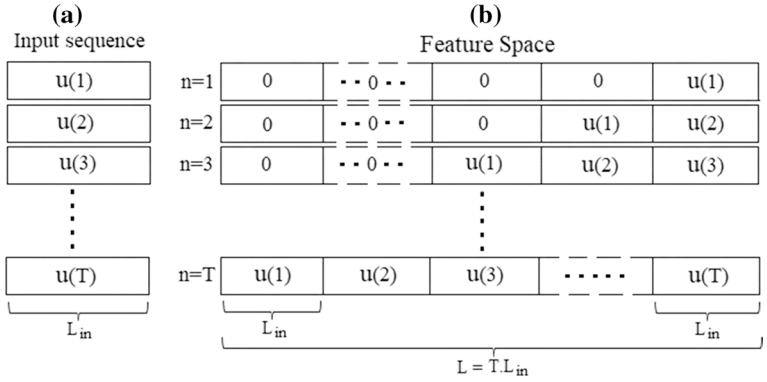


Fig. 12 Stack reservoir (Local recurrent architecture): **a** The input sequence **u**, and **b** The input is memorized consecutively step by step to produce the feature space; the first row at $n = 1$ is used to predict the output at first time step, the second row is used to predict the output at second time step and so on

Table 4 The minimum complexity results for 5-bit task where $T_d = 200$; using several approaches that utilize recurrent architecture of RC based on CA, except stack reservoir which is used local representation instead of CA

Method	Reference	Successful Rules	L_{CA} (bits)
Normal	This paper	165	224
Overwrite	This paper	165	282
Non-Uniform CA	Nichele and Gundersen (2017)	90 and 165	320
ReCA **HALF EACH LAST**	Margem and Gedik (2019)	150	390
XOR	This paper	34, 162.	424
Binary	This paper	15, 34, 38, 42, 162, 170.	424
Gray	This paper	34, 38, 42, 162, 170.	424
ReCA **LAST**	Margem and Yilmaz (2016)[a]	90	688
Stack Reservoir	This paper	×	840
Deep learning CA Single reservoir	Nichele and Molund (2017)	90	2560
Deep learning CA Two reservoirs	Nichele and Molund (2017)	90 and 90	4480
CA Based Feature Expansion and RC	Yilmaz (2015b)	150	4864

The results are listed in ascending order, i.e., the best result is on the top. For more details, please see Table 6 and the appropriate reference in the 2nd column

[a]In this reference, $T_d = 1000$. For that, the experiment has been repeated for $T_d = 200$

in 5-bit tasks. On this point there are two notes should be presented: **1-** It is an encroachment for using the word 'shift' for these rules because they are shift rules only at first time step but after insertion the second time step to the reservoir, there is no guarantee to stay as shift rules, it is dependent on the non-zero positions in the new initial state at second time step and so on for the other time steps. **2-** The local representation is applied only in the reservoir, thus the model still has a computational power from the regressor in the read-out stage.

In local representation, there are also feed-forward and recurrent architectures. In the local feed forward, the input sequence is flattened (vectorization) and is directly used as features for training to predict the whole output, it will be called (without CA) as will be shown in Table 5 for comparison section. But, in local recurrent, the sequence input is memorized consecutively step by step as shown in Fig. 12. Thus, the insertion function is similar to the *stack memory*. The first row of feature space which has only the first time step input is used to predict the output at first time step and so on up to the last time step consisting of all input time steps as demonstrated in Fig. 12. This representation can also be considered as a type of *variable size sliding window* that has been used in Deypir et al. (2012) and Kang and Guo (2009) for frequent itemset mining and speech identification respectively, but the variation is in the amount of given information to each time step not in the windows length. This representation is not a fixed length; hence from this point of view; it is similar to feed-forward, which is problematic for large sequence length T and large input dimension L_{in}. This representation has been used in Margem and Yilmaz (2016) for comparison with ReCA to study the required distributedness and computational to solve sequence tasks and was called stack reservoir.

Surprisingly, the local feed-forward and stack reservoir could solve the 5-bit task with feature vector dimension $L_{CA} = 840$ bits for both; but, this vector is used to predict the whole output for local feed-forward (without CA) while for stack representation it is only

Table 5 Minimum training examples to achieve zero test error in the generalized 5-bit task, where $T_d = 200$, using different methods

Architecture	Method	N_{train}	(I, k, f) or N
Recurrent	**Normal**	**2**	(4, 1, 0.5)
	Overwrite	**2**	(3, 1, 0.5)
	XOR	**2**	(3, 2, 0.5)
	Binary and **Gray**	**2**	(4, 2, 0.5)
	Stack reservoir	3	×
	ESNs (smart)	10	$N = 200$
	ESNs (Basic)	22	$N = 500$
	ESNs (Blind)	28	$N = 2500$
Feed-forward	**All-ff**	11	$(I, k) = (1, 1)$
	Each-ff	10	$(I, k) = (1, 1)$
	Without CA	11	×

used to predict the output in a single time step. For generalized 5-bit, the local feed-forward (without CA) and stack reservoir could also solve this task using 11 training examples for local feed-forward and 3 examples for stack reservoir. Even with these surprising results for feed-forward and local representation methods; ReCA is still the best choice for sequence learning as it will be shown in the discussion section.

5.4 Comparison with other approaches

The model complexity is a useful metric to compare different approaches. Therefore, the feature vector dimension L_{CA} (bits) will be used to compare between different RC based on CA approaches (a snapshot of this approaches is presented in Sect. 1.3) that have been used 5-bit task. Table 4 lists the best results of recurrent architectures from those approaches.

For the generalized 5-bit task, there are no other results except our results in this paper. Therefore, to compare our results with echo state networks, the ESNs experiment has been reproduced as in Jaeger (2012) and applying the generalized 5-bit task to ESNs with three levels of effort/expertise (Blind, Basic, and Smart) Jaeger (2012). Table 5 shows the minimum training examples that are required to obtain zero test error using different methods of recurrent and feed-forward architectures.

The ESNs with the highest level of effort (Smart) Jaeger (2012) could achieve zero test error using 10 training examples with $N = 200$ neurons in the reservoir while for basic and blind the value of training examples increased to 22 and 28 respectively. But, all ReCA methods in this paper could achieve zero test error using only 2 or 3 training examples with a very low effort 3 or 2 iterations, using 634 bits or 424 bits (Table 3) as a feature vector L_{CA} instead of 6400 bits in the best case (smart) of ESNs for 32 bits floating point.

5.5 Discussion

ReCA constructs a novel bridge between computational theory of automata and recurrent neural networks. Despite the simple CA computation, ReCA could solve the 5-bit sequence memory tasks with very low complexity, so it can be considered to operate in the lower bound on complexity.

Different methods are provided to extract the features from CA evolution states in ReCA model. 5-bit and generalized 5-bit tasks have been used in order to compare and evaluate

these methods with each other and with the state-of-the-art of other approaches in RC based on CA domain. The presented results in Tables 1 and 2 reveal that some of ECA rules in classes II, III and IV with all proposed methods could solve the 5-bit task with different levels of complexity.

The minimum complexity was obtained using **Normal** method and rule 165 from class III with feature vector length of 224 bits due to its high distributedness (more computational power), compared with shift rules, which enabled it to collect large information in a small dimension. Increasing the model complexity by increasing I increases the number of rules that could solve the 5-bit task due to the computational power increasing as proved in Yilmaz (2015b).

Based on Table 3, some of the class II rules and rule 106 from class IV with all proposed methods could solve the generalized 5-bit task with different levels of complexity. The minimum complexity was obtained using **Overwrite** and **XOR** methods with rules 42 and 170 from class II with feature vector length of 634 bits using only *two* examples for training, while for *three* training examples the minimum complexity was obtained using **Overwrite** method with rules 42 and 170 from class II with feature vector length of 424 bits. Increasing the training examples provides the model more information which makes the task simpler and then more ECA rules can solve the generalized task with less complexity. Also, increasing the model complexity by increasing I increases the model computational power to use less training examples and more rules can also solve the task with few training examples. Hence, there is a trade-off between complexity and the required training examples, but we should be careful of the *overfitting* in complicated (large complexity) models, then further training examples will be required to generalize the model. That is why the minimum complexity $L_{CA} = 224$ bits has been reached for the 5-bit task where all the 32 examples have been used for training, i.e., the model is not general.

ReCA could achieve zero test error using only *two* training examples which is the *lowest limit number* for training for any model, so we can say that ReCA can be used in the cases of lack of examples, especially the same result was obtained in Margem and Yilmaz (2016) for all the 7 pathological tasks. But still, we must examine ReCA for large space of tasks as it will be done in the future work.

In most literature, the researchers concentrate on classes III and IV of ECA. This is due to the chaotic behavior of class III which is used in cryptography and random number generators, and the complex behaviors of class IV with its computational universality. But, as it has been illustrated in this paper, class II ECA rules have reached the best results in 5-bit memory tasks, so class II rules should be given more prominence in the next studies.

Combining all iterations of CA evolution states using **XOR**, **Binary** or **Gray** provided an efficient feature vector which could solve the both types of 5-bit tasks with larger complexity than **Normal** and **Overwrite** methods. To take advantages of **XOR**, **Binary** or **Gray** options; **Normal** and **Overwrite** methods should have $k \geq 2$ because in this case, the parameter k changes to 1 in **XOR**, **Binary** or **Gray** options, which reduces the dimension of feature vector to be used in the read-out stage. The reservoir size for all methods is equally and depends on I, k and f; the difference is in the feature vector dimension L_{CA}[10], which will be used in the regressor to find the pseudo-inverse that implies the most expensive computational part.

[10] For **Normal** and **Overwrite**, L_{CA} depends on I, k, and f, but it depends only on I and f for **XOR**, **Binary** and **Gray**.

For 5-bit task, the best results have been obtained using rule 165 which outperformed its conjugate rule 90 due to the increasing of non-zero bits (Fig. 9b) that decreasing the mean pairwise correlation of the CA features for different inputs as in Yilmaz (2015b), i.e., improving the computational power Lukoševičius and Jaeger (2009). But, both rules 165 and 90 could not solve the generalized 5-bit task due to their complex behavior (rich dynamics) which is introduced the model to *overfitting*, to overcome this problem; an excess of training examples are required or reducing the model complexity as we have done by using the shift rules 42 and 170 that have less dynamics.

It is noted that only class II ECA rules could solve the generalized 5-bit task, but due to the relative simplicity of their behavior and to avoid the *underfitting*; the model needs more iterations I for a small number of training examples or needs more training examples for a small number of iterations. We should not be confused with rule 106 which could solve the generalized 5-bit task. However, it is from class IV but its behavior is similar to the shift rules from class II for a single non-zero input as illustrated in Fig. 11. Although those simple shift rules obtained the best results for the generalized 5-bit task, we should check ReCA using other types of tasks in future work to find which ECA rules are the best for each task.

The 82 ECA rules from the whole 256 rules have only been used in our experiments, these rules have been selected from the equivalent sets of ECA rules Martnez (2013). It is not enough because, as an example, the rule 90 and its conjugate rule 165 (also rule 60 and its reflection rule 102) are in the same equivalent set in Martnez (2013) but experimentally they gave different results. Therefore, we have to exploit the whole ECA rule space in future work.

The stack reservoir is a recurrent architecture because its states in the reservoir memorize the input information gradually (time-step by time-step). But, on the other hand, the stack reservoir has not a fixed length, so it is similar to feed-forward architecture.

In spite of good results were obtained from the feed-forward models for 5-bit task, there are some points should be discussed here: feed-forward is conceptually not plausible; because the subsequent (new) inputs are used to predict some previous (old) output. So, the provided information is increased for prediction, that is why it gives good results. Also, the 5-bit task has been designed in Hochreiter and Schmidhuber (1997) to examine long short-term memory (LSTM) which is a type of RNNs, thus suitable tasks other than 5-bit task should be used to test the feed-forward model.

Feed-forward architecture needs more training examples to solve the generalized 5-bit task compared with the recurrent architectures as listed in Table 5. Finally, the Stack and feed-forward models are problematic for large input dimension L_{in} and/or large sequence T; because they have not a fixed length, i.e., their feature vector length increases proportionally with L_{in} and T.

The comparison in Table 4 indicates that the best results of all RC based on CA approaches have been achieved using the additive rules (165, 90 and 150) or shift rules (15, 34, 38, 42, 162 and rule 170 which is also additive) that provide lossless injection of input at each time step, i.e., maximizes one-to-one correspondence between input sequence and the reservoir activity due to the sequence. Moreover, the additive rules can be represented as linear functions modulo two. Thus, these rules allow to compute independently the evolution for different initial states, then the results can be combined by simply adding which significantly simplifies the hardware implementation of these rules.

The comparison of best results in Table 4 demonstrate that the feature expansion using zero *buffers* as in our method is better than using random permutation as in the other approaches; this is due to the natural information diffusion into the CA reservoir, thus there is no information loss. But, the random permutation of the inputs need a lot of permutations (large size) to ensure there is no information loss. Also, the permutations increase the random interference between the input cells into the reservoir while the model becomes more robust using the zero buffers; because the evolution states of the input obey a certain rule for a single input (not multiple random permuted inputs as in the other approaches). This interpretation agrees with our results where the **Normal** and **Overwrite** methods outperform the other options (**XOR**, **Binary** and **Gray**) as illustrated in Table 4; because in **Normal** and **Overwrite** methods the feature vector is the pure CA evolution states without any intervention but in **XOR**, **Binary** and **Gray** options, the feature vector is modified by an operator. This interpretation of course is valid for the used 5-bit tasks, therefore these methods should be tested using different types of tasks, then study which method is the best for every task as will be done in future work.

In ReCA, there is no parameter selected randomly as in ESNs or the other approaches of RC based on CA (Yilmaz 2015b; Nichele and Gundersen 2017; Nichele and Molund 2017), thats why we did not repeat our experiments multiple times in 5-bit task as in Jaeger (2012), Yilmaz (2015b), Nichele and Gundersen (2017) and Nichele and Molund (2017), but for the generalized 5-bit task we have repeated the experiments 100 times (100 trials) due to the random selection of training examples from the whole dataset, not from our model.

For the 5-bit task, ReCA has outperformed ESNs as demonstrated in Yilmaz (2015b) and Margem and Yilmaz (2016). For the generalized 5-bit task, ReCA also outperforms ESNs where ReCA could solve the task using only *two* training examples with a very simple model but (smart) ESNs which is the most complex model could solve this task using *ten* training examples while for the lower complexity models (basic) and (blind) ESNs, they require 22 and 28 training examples respectively to solve this task.

The ESN multiplications in the reservoir and read-out stage (the regressor) is replaced with bit-wise logic in the reservoir and can be replaced by summation in read-out stage due to its binary features which speed up the model. ReCA is also suitable for hardware implementation using ordinary logic gates or Field Programmable Gate Arrays (FPGA) due to the bit-wise logic operations (Not, And and Or) in the reservoir instead of the floating point multiplication in ESNs reservoir, resulting in reduced the complexity in space, time, and power consumption.

6 Conclusion and future work

Reservoir computing based on cellular automata ReCA constructs a novel bridge between automata computational theory and recurrent neural architectures. In this paper, ReCA has been implemented to solve memory tasks. Several methods have been proposed to extract the features from the reservoir, either using the cellular automata evolution states directly (the natural diffusion of CA states in the reservoir) or combining all states together in one

vector using XOR, Binary or Gray operation to produce one feature vector. The results were close; the direct method outperforms and offers the state-of-the-art results using rule 165 (additive rule) for 5-bit task, and rules 42, 170 (shift rules) for generalized 5-bit task using only *two* training examples which is the *lowest* number of training examples for any model.

Concerning the model complexity, a sparsely connected network with binary units like elementary cellular automata in ReCA reservoir could solve a sequence memory task that has 210 time steps using only 224 bits as features to achieve zero error in 5-bit task and using 634 bits to solve the generalized 5-bit task using only *two* training examples. Thus, it can be considered that ReCA operates around a lower bound on complexity.

The increasing of training examples to *three* reduces the feature vector dimension to 424 bits which decreases to 224 bits in 5-bit task i.e., 32 training examples.

The distributed representation of CA in the reservoir with a recurrent architecture provides the high computational power to the model to solve the generalized 5-bit task using only *two* training examples while the stack reservoir (local representation with a recurrent architecture) requires *three* training examples with longer feature vector, whereas the feed-forward architecture (local and distributed representation) and echo state networks with the most complicated level (smart) require *ten* training examples.

Usage of cellular automata in the reservoir computing paradigm greatly simplifies the architecture, makes the computation more transparent for analysis, and provide enough computation for 5-bit memory tasks, furthermore the reservoir can be implemented using ordinary logic gates or Field programmable gate arrays FPGAs, resulting in reduced the complexity in space, time and power consumption.

ReCA framework is very novel, so there is a lot of research should be done. The previous sections imply some points to enrich the ReCA research that is listed as follows:

- Use the rest of 265 rules that have not been used in this study.
- Repeating the experiments using **Overwrite** method with the combining state options **XOR**, **Binary**, and **Gray**.
- Expand the work in this paper to solve all the pathological tasks, non-binary tasks, and more realistic tasks.
- Preprocessing the input before the reservoir as in Margem and Yilmaz (2016) where it provided promising results.
- Other training methods can be used in the read-out stage rather than linear regression, e.g., logistic regression, support vector machines SVMs or even feed-forward neural networks.

Appendix 1: Calculation details of the result comparison

See Table 6.

Table 6 The calculation details of minimum complexity results in Table 4 for 5-bit task using several approaches

Method	L_{CA} equation	The values of L_{CA} parameters	L_{CA} (bits)
Normal	$(L_{in} + 2(f.I.T)).k$	$(4 + 2(0.26 \times 2 \times 210)) \times 1$	224
Overwrite	$(L_{in} + 2(f.I.T)).k$	$(4 + 2(0.33 \times 2 \times 210)) \times 1$	282
Non-Uniform CA	$(C.L_{in}).R.I$	$(10 \times 4) \times 4 \times 2$	320
ReCA **HALF EACH LAST**	$(L_{in} + (f.I.T)).k$	$(4 + (0.15 \times 4 \times 210)) \times 3$	390
XOR	$L_{in} + 2(f.I.T)$	$4 + 2(0.5 \times 2 \times 210)$	424
Binary	$L_{in} + 2(f.I.T)$	$4 + 2(0.5 \times 2 \times 210)$	424
Gray	$L_{in} + 2(f.I.T)$	$4 + 2(0.5 \times 2 \times 210)$	424
ReCA **LAST**	$(L_{in} + 2(f.I.T)).I$	$(4 + 2(0.1 \times 4 \times 210)) \times 4$	688
Stack Reservoir	$T.L_{in}$	210×4	840
Deep learning CA Single reservoir	$L_d.R.I$	$40 \times 8 \times 8$	2560
Deep learning CA Two reservoirs	$L_{d1}.R_1.I_1 + L_{d2}.R_2.I_2$	$(40 \times 8 \times 8) + (30 \times 8 \times 8)$	4480
CA Based Feature Expansion and RC	$L_{in}.R.I$	$4 \times 38 \times 32$	4864

Appendix 2: Pseudo-code

Algorithm 2.1 is the Pseudo-code that has been used to create the matrix of training features A_{train} which is used in read-out stage to find $\mathbf{W}_{out}$. The output weight matrix $\mathbf{W}_{out}$ will be used to find the predicted output $\hat{\mathbf{y}}_{train}$ in 5-bit task and $\hat{\mathbf{y}}_{test}$ in generalized 5-bit task as explained in Sect. 4.2.

Algorithm 2.1 5-Bit(Input_Train, RuleNo, I, k, f)

T = The sequence length of Input_Train ▷ Input_Train is the input of training set
L_{in} = The number of bits of Input_Train
$R = f(I \times T)$
Buffer = $zeros(T, R)$
CA_Input = [Buffer Input_Train Buffer] ▷ adding 2 buffers to both sides of the input
$L = L_{in} + 2R$ ▷ The number of columns of CA_Input
for i=1:T **do**
  if $i == 1$ **then**
    InitialState = CA_Input$(i, 1 : L)$
    CA_output$(1 : I, 1 : L, i)$= ECA(InitialState, RuleNo, I) ▷ Evolving the 1^{st} time step of CA_Input using ECA rule to I iterations.
  else
    InitialState = CA_Input$(i, 1 : L) \oplus$ CA_output$(I, 1 : L, i - 1)$ ▷ XOR Insertion function to insert a new input time step into the reservoir.
    CA_output$(1 : I, 1 : L, i)$= ECA(InitialState, RuleNo, I)
  end if
  CA_Train$(1 : (k \times L), i)$=Concatenate (CA_output$(I - k + 1 : I, 1 : L, i)$) ▷ Concatenate CA_output using only last k states for each time step (**EACH** option).
end for

The algorithm 2.1 should be repeated for N_{train} training examples and placed the obtained CA_Train matrices consecutively to produce the matrix $\mathbf{A}_{train}$ with size of $(k \times L, N_{train} \times T)$. Then, we use Eq. (7) to find the output weight matrix $\mathbf{W}_{out}$.

The line 12 in algorithm 2.1 should be replaced by the following line: InitialState = [CA_output$(I, 1 : R, i-1)$ CA_Input$(i, R+1 : R+L_{in})$ CA_output$(I, R+L_{in}+1 : L, i-1)$] for the **Overwrite** insertion function.

The function (Concatenate) in line 15 of the algorithm 2.1 will be replaced by (**XOR**), (**Binary**), or (**Gray**) according to the used option.

To find $\mathbf{A}_{test}$ in generalized 5-bit task, the algorithm 2.1 is used; where the Input_Train set is replaced by Input_Test set. Then, N_{test} examples will be used from the testing set.

References

Adamatzky A (2001) Computing in nonlinear media and automata collectives. CRC Press, Boca Raton

Alonso-Sanz R, Martin M (2006) Elementary cellular automata with elementary memory rules in cells: the case of linear rules. J Cell Autom 1(1):71–87

Bengio Y, Simard P, Frasconi P (1994) Learning long-term dependencies with gradient descent is difficult. IEEE Trans Neural Netw 5(2):157–166

Bertschinger N, Natschläger T (2004) Real-time computation at the edge of chaos in recurrent neural networks. Neural Comput 16(7):1413–1436

Cook M (2004) Universality in elementary cellular automata. Complex Syst 15(1):1–40

Dai X (2004) Genetic regulatory systems modeled by recurrent neural network. In: International symposium on neural networks. Springer, pp 519–524

Dale M, Miller JF, Stepney S, Trefzer MA (2016) Evolving carbon nanotube reservoir computers. In: International conference on unconventional computation and natural computation. Springer, pp 49–61

Dale M, Miller JF, Stepney S (2017) Reservoir computing as a model for in-materio computing. In: Adamatzky A (ed) Advances in unconventional computing. Springer, pp 533–571

Deypir M, Sadreddini MH, Hashemi S (2012) Towards a variable size sliding window model for frequent itemset mining over data streams. Comput Ind Eng 63(1):161–172

Dietterich TG (2002) Machine learning for sequential data: a review. In: Joint IAPR international workshops on statistical techniques in pattern recognition (SPR) and structural and syntactic pattern recognition (SSPR), vol 88. Springer, pp 15–30

Doya K (1992) Bifurcations in the learning of recurrent neural networks. In: IEEE international symposium on circuits and systems, vol 6. IEEE, pp 2777–2780

Fernando C, Sojakka S (2003) Pattern recognition in a bucket. In: European conference on artificial life. Springer, pp 588–597

Funahashi K, Nakamura Y (1993) Approximation of dynamical systems by continuous time recurrent neural networks. Neural Netw 6(6):801–806

Goodfellow I, Bengio Y, Courville A (2016) Deep learning. MIT Press, Cambridge

Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780

Hochreiter S, Bengio Y, Frasconi P, Schmidhuber J (2001) Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In: A field guide to dynamical recurrent neural networks, chap 14. Wiley-IEEE Press, pp 237–244

Huang GB, Zhu QY, Siew CK (2004) Extreme learning machine: a new learning scheme of feedforward neural networks. In: IEEE international joint conference on neural networks, IJCNN. 2004, vol 2, pp 985–990

Huang GB, Wang DH, Lan Y (2011) Extreme learning machines: a survey. Int J Mach Learn Cybern 2(2):107–122

Jaeger H (2001) The echo state approach to analysing and training recurrent neural networks-with an erratum note. Bonn, Germany: German National Research Center for Information Technology GMD Technical Report 148(34):13

Jaeger H (2012) Long short-term memory in echo state networks: details of a simulation study. Technical Report 27, Jacobs University Bremen

Jones B, Stekel D, Rowe J, Fernando C (2007) Is there a liquid state machine in the bacterium *Escherichia coli*? In: IEEE symposium on artificial life. IEEE, pp 187–191

Kang G, Guo S (2009) Variable sliding window DTW speech identification algorithm. In: Ninth international conference on hybrid intelligent systems, vol 1. IEEE, pp 304–307
Kuhn M, Johnson K (2013) Applied predictive modeling, vol 26. Springer, New York
Langton CG (1986) Studying artificial life with cellular automata. Physica D 22(1–3):120–149
Larger L, Soriano MC, Brunner D, Appeltant L, Gutiérrez JM, Pesquera L, Mirasso CR, Fischer I (2012) Photonic information processing beyond turing: an optoelectronic implementation of reservoir computing. Opt Express 20(3):3241–3249
Legenstein R, Maass W (2007) Edge of chaos and prediction of computational performance for neural circuit models. Neural Netw 20(3):323–334
Li W, Packard N (1990) The structure of the elementary cellular automata rule space. Complex Syst 4(3):281–297
Lukoševičius M (2012) A practical guide to applying echo state networks. In: Montavon G, Orr G, Müller KR (eds) Neural networks: tricks of the trade. Springer, pp 659–686
Lukoševičius M, Jaeger H (2009) Reservoir computing approaches to recurrent neural network training. Comput Sci Rev 3(3):127–149
Maass W, Natschläger T, Markram H (2002) Real-time computing without stable states: a new framework for neural computation based on perturbations. Neural Comput 14(11):2531–2560
Margem M, Gedik OS (2019) Reservoir computing based on cellular automata (ReCA) in sequence learning. J Cell Autom 14(1–2):153–170
Margem M, Yilmaz O (2016) How much computation and distributedness is needed in sequence learning tasks? In: International conference on artificial general intelligence, AGI-16. Springer, pp 274–283
Martens J, Sutskever I (2011) Learning recurrent neural networks with hessian-free optimization. In: the 28th International conference on machine learning (ICML-11), pp 1033–1040
Martnez GJ (2013) A note on elementary cellular automata classification. J Cell Autom 8(3–4):233–259
Martnez GJ, Adamatzky A, Alonso-Sanz R (2013a) Designing complex dynamics in cellular automata with memory. Int J Bifurc Chaos 23(10):1330035
Martnez GJ, Seck-Tuoh-Mora JC, Zenil H (2013b) Computation and universality: class iv versus class iii cellular automata. J Cell Autom 7(5–6):393–430
Martnez GJ, Seck-Tuoh-Mora JC, Zenil H (2013c) Wolframs classification and computation in cellular automata classes iii and iv. In: Zenil H (ed) Irreducibility and computational equivalence. Springer, pp 237–259
McDonald N (2017) Reservoir computing & extreme learning machines using pairs of cellular automata rules. In: International joint conference on neural networks (IJCNN), USA, vol 88. pp 2429–2436
Morn A, Frasser CF, Rossell JL (2018) Reservoir computing hardware with cellular automata. arXiv :1806.04932
Nichele S, Gundersen MS (2017) Reservoir computing using non-uniform binary cellular automata. Complex Syst 26(3):225–245
Nichele S, Molund A (2017) Deep learning with cellular automaton-based reservoir computing. Complex Syst 26(4):319–339
Ortín S, Soriano MC, Pesquera L, Brunner D, San-Martín D, Fischer I, Mirasso C, Gutiérrez J (2015) A unified framework for reservoir computing and extreme learning machines based on a single time-delayed neuron. Sci Rep 5:14945
Paquot Y, Duport F, Smerieri A, Dambre J, Schrauwen B, Haelterman M, Massar S (2012) Optoelectronic reservoir computing. Sci Rep 2:287
Pascanu R, Mikolov T, Bengio Y (2013) On the difficulty of training recurrent neural networks. In: the 30th International conference on machine learning, Atlanta, Georgia, USA
Siegelmann HT, Sontag ED (1995) On the computational power of neural nets. J Comput Syst Sci 50(1):132–150
Snyder D, Goudarzi A, Teuscher C (2013) Computational capabilities of random automata networks for reservoir computing. Phys Rev E 87(4):042808
Toffoli T, Margolus N (1987) Cellular automata machines: a new environment for modeling. MIT Press, Cambridge
Von Neumann J (1951) The general and logical theory of automata. Cereb Mech Behav Hixon Symp 1:1–41
Wolfram S (1994) Tables of cellular automaton properties. In: Wolfram S (ed) Cellular automata and complexity: collected papers. Westview Press, pp 513–584
Wolfram S (2002) A new kind of science, vol 5. Wolfram Media, Champaign
Yilmaz O (2014) Reservoir computing using cellular automata. arXiv:1410.0162

Yilmaz O (2015a) Analogy making and logical inference on images using cellular automata based hyperdimensional computing. In: NIPS, workshop on cognitive computation
Yilmaz O (2015b) Machine learning using cellular automata based feature expansion and reservoir computing. J Cell Autom 10(5–6):435–472
Yilmaz O (2015c) Symbolic computation using cellular automata-based hyperdimensional computing. Neural Comput 27(12):2661–2692