An experimental study on cellular automata reservoir in pathological sequence learning tasks

Mrwan MARGEM$^1$ and Özgür YILMAZ$^2$

$^1$ Turgut Ozal University, Department of Electrical and Electronic Engineering, Ankara Turkey mmargem2013@stu.turgutozal.edu.tr
$^2$ Turgut Ozal University, Department of Computer Engineering, Ankara Turkey ozyilmaz@turgutozal.edu.tr; ozguryilmazresearch.org

Abstract. In this study, we are providing a recurrent architecture based on Cellular Automata state evolution. The states of the cells are used as the reservoir of activities as in Echo State Networks. The projection of the input onto this reservoir medium provides a systematic way of remembering previous inputs and combining the memory with a continuous stream of inputs. This is an essential capability for a wide collection of intelligence tasks such as language, continuous vision (i.e. video), symbolic manipulation in a knowledge base etc. We devise intuitive methods for enlarging the memory capacity of the cellular automata state space and reducing the interference between memory traces. Cellular Automata reservoir constructs a novel bridge between computational theory of automata and neural architectures.

The proposed framework is tested on classical synthetic pathological tasks that are widely used in evaluating recurrent algorithms. We show that the proposed algorithm achieves zero error in all tasks, giving a similar performance with Echo State Networks, but even better in many different aspects.

Two other methods are also introduced to training recurrent neural networks; “Covariance representation” that has second order attribute statistics and “Stack representation” that has a local representation, and compare them with Cellular Automata based Reservoir Computing framework that has higher attribute statistics and distributed representation. This raises the question of whether real valued neuron units are essential for solving complex problems that are distributed over time. Our results suggest that, even very sparsely connected binary units with simple computational rules can provide the required computation for intelligent behavior.

Keywords: Recurrent Neural Networks · Reservoir Computing · Cellular Automata · long term dependencies · Distributedness · Computational Complexity

1 Introduction

In this study, we are analyzing how much computation and distributedness of representation is needed to solve sequence-learning tasks which are essential for
many artificial intelligence applications. We define a recurrent formulation and apply that for three different representations: Stack, Covariance and Cellular Automata. In Stack representation, there is no computation involved, the input sequence steps are reserved one after another in raw format. Covariance representation computes the pairwise covariance of the input attributes as in tensor products [1] and memorizes those for each sequence input. Being very similar to Covariance representation in terms of complexity of operations. Cellular Automata holds a distributed representation of high order attribute statistics. Stack representation provides pure memorization, whereas Covariance representation computes useful second order statistics. Cellular Automata representation enables both computation of high order statistics and distributedness. We contrast these three approaches in sequence learning tasks and show that Cellular Automata approach gives superior performance than the other two and equivalent performance with Echo State Networks with significantly less computational demands.

Therefore, one of the main contributions of the paper is a novel framework of cellular automata based reservoir computing in a recurrent setting (called ReCA), that is capable of short-term memory. Cellular automaton is used as the reservoir of dynamical systems. Automaton cells are initialized with the input at each time step, which will be evolved for a period of time using an elementary cellular automaton rule. The evolution of the automaton creates a rich and discriminative space-time volume that can be used as the feature vector. Usage of cellular automaton instead of real valued neurons in reservoir computing framework greatly simplifies the architecture, and makes the computation more transparent for analysis.

Overall our experiments suggest that calculation of high order statistics and representing them in a distributed manner is essential, yet even a very sparse network with identical binary units such as cellular automaton might provide enough computation for sequence learning.

Next, we review reservoir computing and cellular automata, then provide methods and results in the following sections.

1.1 Reservoir Computing

Many real life problems in artificial intelligence (AI) require the system to remember previous inputs. Recurrent Neural Networks (RNN)s are powerful tools of machine learning with memory. In RNNs, the input presented to the network is stored in a distributed fashion after it is transformed into hidden layer activities through a highly nonlinear function. Therefore, they employ very powerful hierarchical computation as well as distributed representation which make them excellent tools for sequence learning tasks. Unfortunately RNNs are difficult to train properly due to the inherent difficulty of learning optimal representations tailored for long-term dependencies [6] [7] and convergence issues [8]. There is some recent approaches to avoid these issues, such as emulating the recurrent computation in feedforward neural network framework [9] or utilizing a reservoir of coupled oscillators [10]and [11].
In 2001 a fundamentally different approach to RNNs design and training was proposed independently by Wolfgang Maass under the name of Liquid State Machines (LSM) [11] and by Herbert Jaeger under the name of Echo State Networks (ESN) [10]. The ESNs and LSMs became lately known as Reservoir Computing (RC) approaches [12].

RC model avoids the shortcomings of conventional training methods in RNNs, by setting up RNNs in the following way [13] and [14]:

1. A recurrent neural network is randomly created and remains unchanged during training. This RNN is called the reservoir. It is excited by the input signal and maintains in its state a nonlinear transformation of the input history.
2. The desired output signal is generated as a linear combination of the neuron’s signals from the input-excited reservoir. This linear combination is learned by linear regression, using the teacher signal as a target.

In RC approach, due to rich dynamics already provided by the reservoir, there is no need to train many recurrent layers and learning takes place only at the output (read-out) stage. This simplification enables usage of recurrent neural networks in complicated tasks that require memory for long-range statistical relationships (spatially and temporally). However one question still remains: how much computation is needed in the reservoir? In Echo State Networks, real valued neurons with nonlinear activation functions are utilized and this still corresponds to a fairly complex neural model. Can we simplify the recurrent architecture further by using identical binary units, and the maximum possible amount of connection sparsity? This corresponds to a elementary cellular automaton array.

1.2 Cellular Automata

Cellular Automata (CA) are discrete dynamical systems with sparse connections. A cellular automaton is an array of cells evolving synchronously according to an identical interaction rule. The evolution of a cell is dependent on the previous states of a surrounding neighborhood of cells as shown in Fig. 1(b) and illustrated in [15] and [16]. The dimension of cellular automata can have an integer value thus it is a grid in general, but one or two dimensions are utilized in most of the studies. CA has either null or periodic boundary. In null boundary configuration the boundary cells are assumed to have a logic 0, but in a periodic boundary the grid is folded. Therefore in a single dimension CA, the right most cell becomes the neighbor of the left most one. The rules applied to each cell can be either identical or different, these two different configurations are termed as uniform and hybrid CA respectively [19] but if the rules change in time (in certain iterations), the configuration is termed as multilayer CA.

Elementary Cellular Automaton (ECA) is a one dimensional CA with binary cells (1 or 0), evolving according to a uniform and non-changing rule. It is the

\[3\] In hybrid, the rules change in space not in time.
Fig. 1. General framework for Cellular Automata based Reservoir Computing with vector lengths for each stage: a. Encoding Stage that consists of: CA: Input expansion using CA, Ri: Representing each input bit by $R_i$ bits and make rotation every time step and Zero Array $R$: Adding zero array with dimension of $R$ for both sides of the input. Note: Not all three parts should be used in all tasks. b. Cellular Automata Reservoir Stage; The output of Encoding Stage is projected onto cellular automaton instead of a neural network in Echo State Networks. (The used CA rule is rule 90 with periodic boundary). c. Read-out Stage; The feature vector with dimension of $L_{CA}$ is trained by Linear regression.

The simplest cellular automaton configuration and in our work we will be using ECA exclusively.

The rules of ECA are classified according to their behavior (Wolfram classes) [16]. Starting from random initial cell values, CA state evolution will show a certain behavior: Class I (Uniform) CA states evolve to a homogeneous behavior, Class II (Periodic) CA states evolve periodically, Class III (Chaotic) CA states evolve chaotically (without any defined pattern) and Class IV (Complex or edge of chaos) can show all these evolution patterns in an unpredictable manner. Despite its simplicity, the fourth class is capable of universal computation i.e. it can simulate a Turing machine [17] and [18].

2 Cellular Automata in Reservoir Computing: ReCA

The introduction of Cellular Automata into reservoir computing framework was proposed in [2] for a feedforward architecture and recurrent formulation was given in [3], and some applications discussed in [4] and [5]. The main idea can be summarized as follows: cellular automata provide powerful enough computation and rich enough representation to be used instead of real valued recurrent neural networks. Cellular automaton is a very sparsely connected network with identical and binary units, thus it gives a lower bound on the amount of model complexity for solving hard problems in AI.
In this paper we are using the recurrent formulation cellular automata reservoir to handle a sequence of inputs. The algorithmic flow of CA based RC (ReCA) is shown in Fig. 1. The encoding stage translates the input into the initial states of CA. In cellular automata reservoir stage, the ECA rules are applied for a fixed period of iterations (I), to evolve the CA initial states. The CA states in the reservoir are concatenated to produce a feature vector that will be used in read-out stage (Linear Regression). The details of the method is given below step by step.

2.1 Encoding Stage

The encoding stage can be divided into three subroutines as shown in Fig.1(a): CA, Ri and Zero Array R.

Utilizing Buffers (Zeros Array R): In encoding stage, the input is translated into the initial states of cellular automaton. For handling a sequence of inputs, each having a length $L_{in}$, the size of the input to reservoir need to be adjusted. An array of zeros with length of $R$ are added to both sides of original input, these buffers will hold the activity of the reservoir corresponding to previous time steps. Then, the expanded input to cellular automata reservoir is of length $L = L_{in} + 2R$, as shown in Fig. 1 and 2.

In most of the experiments, $R$ equals to $I \times T$ ($I$ is the number of CA iterations and $T$ is the sequence length of the input), to guarantee that CA states due to all time sequences have been conserved. But in some experiments to reduce time and space complexity an expansion ratio $f \in [0, 1]$ has been introduced as follows $R = f \times (I \times T)$, thus the size of the reservoir (complexity) decreases with the value of $f$.

Reducing Interference: $R_i$ To improve the accuracy in tasks (mainly memorization), the interference between non zero elements in the reservoir should be reduced. The interference is due to the cancellation of cell activity when two non zero cells collide during additive cellular automaton rules such as rule 90. In order to avoid it, each bit of the input can be represented by $R_i$ bits. After each time step, the location of the non zeros will rotate right one bit, as explained schematically in Fig. 3 and 4. So, the input size is artificially expanded to increase the feature space size and reduce the interference between non zero bits.

4 We should note that: the utilizing buffers is only have to be used in our algorithm, but the other two subroutines are applied selectively according to the task.

5 the value of $R_i$ should be chosen carefully to reduce the interference between non zeros that have different locations in consecutive time steps for rule 90.

6 the rotation right is to reduce the interference between non zeros that have the same location in consecutive time steps.
Fig. 2. a. original input sequence before encoding stage. b. Encoding stage and CA Reservoir stage: Adding zero array with length R to obtain the reservoir input sequence from $X_1$ to $X_T$. Then cellular automaton is initialized with the first time step input of the sequence, so $A_0^{(1)} = X_1$ (with size of L). The CA evolution states $A_i^{(n)}$ (i changes from 1 to I) will be used as a feature space to estimate the output at time step n ($y(n)$) using linear regression in read-out stage.

Fig. 3. Reducing the interference between bits in rule 90 by adding $R_i$ with rotation, when there are non zeros that have the same location in consecutive time steps. (a) The original 3 bit input for 2 time steps, (b) $R_i = 4$ bits to represent each input bit, with two iterations of CA evolution. The new input at $n = 2$ is represented using $R_i$, it might interfere with the previous input, (c) Due to the rotation there is no interference between the “1” from 1st time step n = 1 and the bold “1” from the new time step $n = 2$. 
Fig. 4. Reducing the interference between bits in rule 90 by adding $R_i = 4$ with even value and rotation, when there are non zeros that have a different locations in consecutive time steps. (a) The original 3 bit input for 2 time steps, (b) $R_i = 4$ bits (even number) to represent each input bit, with two iterations of CA evolution. The new input at $n = 2$ is represented using $R_i$, it might interfere with the previous input, (c) Due to the rotation and even value of $R_i$, there is no interference between the “1” from first time step $n = 1$ and the bold “1” from second time step $n = 2$.

Multilayer Cellular Automata Expansion: CA Before zero padding with right-left buffers ($R_i$) and expanding each binary input with $R_i$, the original binary input can be transformed into another binary vector using nonlinear CA rules to increase the nonlinearity of the model. Suppose that we have a binary vector of size $L_{in}$. We apply a rule (eg. 110) onto this vector for a certain amount of iterations, then use the last state of the evolution as our new input. This stage enables a Multilayer CA architecture, in which the first layer projects the input into a nonlinear space, and the next layer evolves it further with linear rules in time to expand the feature space. Linearity in the second layer is essential for lossless injection of the input at each time step.⁷

2.2 Cellular Automata Reservoir Stage

After the data is encoded as the initial states of a cellular automaton, it is passed on a CA reservoir (instead of an ESN as in [10]) for computation. The dynamics of CA provide the necessary projection of the input data onto an expressive and discriminative space, as shown in Fig. 1(b). It was previously shown that the cellular automata reservoir holds a distributed representation of high order attribute statistics [3]. Thus, sequence of inputs at each time step is processed to extract the input statistics and these are represented in a distributed manner as

³⁷ Our method should not be confused with conventional multilayer CA, they are similar only at the first time step. But they both use two different CA rules in different times (iterations). The multilayer CA that have been used in our work is structurally and algorithmically very different from Layered CA which is used to predict nerve axonal extension process [20] and in Cryptographic [21].
Fig. 5. The complete used method for 20 Bit task, by adding the zero array $R$ and each bit of input is represented by $R_i$ bits.
in recurrent neural networks. However, different from recurrent neural networks, for each input we can allocate a distinct memory slot in the reservoir space and avoid interference between different time steps altogether (at the cost of increased feature space). This is due to the fact that linear cellular automata rules\(^8\) propagate the cell activities of the previous time steps in a predictable manner, creating "empty spots" for injection of current input in the sequence.

The cellular automaton is initialized with the first time step input of the sequence \(X_1\) that has been obtained from encoding stage, \(A^{(1)}_0 = X_1\) (with size of \(L\)), where the underscript 0 denotes to initial state and increases to \(I\) (No of CA iterations) and the superscript (1) denotes to the number of time steps (from 1 to \(T\)). Then the cellular automata evolution of \(A^{(1)}_0\) is computed using a pre-specified ECA rule (Z) up to \(I\) iterations as shown in Fig. 1(b) and the following equations:

\[
A^{(1)}_1 = Z(A^{(1)}_0),
\]

\[
A^{(1)}_2 = Z(A^{(1)}_1),
\]

\[
\vdots
\]

\[
A^{(1)}_I = Z(A^{(1)}_{I-1}).
\]

The evolution of CA states are concatenated to obtain a single state vector \(A^{(1)}\) with length of \(L_{CA} = LI\), to be used for estimation at time step \(n = 1\):

\[
A^{(1)} = [A^{(1)}_1; A^{(1)}_2; \ldots; A^{(1)}_I].
\]

At the second time step \(n = 2\), the input \(X_2\) will be inserted into the reservoir state vector. Using XOR operation\(^9\) to the last state vector \(A^{(1)}_I\) with input vector at second time step \(X_2\). Then the new initial state vector of the cellular automaton at time step \(n = 2\) will be as:

\[
A^{(2)}_0 = A^{(1)}_I \oplus X_2.
\]

Where, \(\oplus\) is bitwise XOR.

The cellular automaton is evolved for \(I\) steps to attain \(A^{(2)}\) with the same manner that is mentioned above for \(A^{(1)}\).

\[
A^{(2)} = [A^{(2)}_1; A^{(2)}_2; \ldots; A^{(2)}_I].
\]

\(A^{(2)}\) is used for estimation at second time step \(n = 2\).

This procedure is continued until the end of the sequence at \(n = T\), when \(X_T\) is inserted into the reservoir, and obtaining \(A^{(T)}\) that will be used for estimation at last time step \(n = T\), the details are illustrated in Figs. 2 and 5.

\(^8\) especially for Rule 90 and memory tasks.

\(^9\) any other type of binary addition can be used. XOR was used due to the uniqueness of its output.
2.3 Read-out stage

In this stage the cellular automaton state vectors (feature vector with dimension of $L_{CA}$) are used in linear regression to estimate the output. There are two cases for the output:

1. There is only one output $y$ at last time step $n = T$, therefore the output is a vector with dimension of $L_{out}$:

$$y(T) = W_{out} \ast A^{(T)}$$  \hspace{1cm} (7)

Where, $A^{(T)}$ is the feature vector for last time step, and the dimension of regression parameters matrix $W_{out}$ is $L_{out} \times IL$.

2. There is an output for each time step, therefore the output is a matrix $Y$ with size of $L_{out} \times T$. So the output $y$ for each time step $n > 0$ is:

$$y(n) = W_{out} \ast A^{(n)}$$  \hspace{1cm} (8)

After combining all time steps together; the feature space $A^{(all)}$ becomes as in (9):

$$A^{(all)} = \begin{bmatrix} A^{(1)} & A^{(2)} & \ldots & A^{(T)} \end{bmatrix}$$  \hspace{1cm} (9)

Therefore the output will be $Y = W_{out} \ast A^{(all)}$. Where the dimension of $A^{(all)}$ is $IL \times T$ and the dimension of $W_{out}$ is $L_{out} \times IL$.

3 Covariance Representation

A covariance representation will be introduced to compare it with a distributed representation in $CA$ for recurrent architecture.

The operator $C_k$ will be used for a covariance representation which was introduced\footnote{The operator $C_k$ was applied in \cite{3} on feedforward architecture for 5 Bit task, but in this paper is applied on recurrent architecture for the other pathological tasks.} by O. Yilmaz in \cite{3}.

$$C_k = \Pi_k A_0 \oplus \Pi_{-k} A_0 ,$$  \hspace{1cm} (10)

where $\Pi_1$ and $\Pi_{-1}$ are matrices $+1$ and $-1$ bit rotate operations and $\oplus$ is bitwise XOR.

In $C_k$ the covariance of the attributes that are $2k$ apart from each other, which is hold at the center bit of $k^{th}$ time step evolution as shown in Fig. 6.

In covariance representation there is no relation between time steps evolution with each other, there is only a relation with the initial state. But for $CA$ representation, there is a relation between time steps evolution with each other and of course with the initial state \cite{3}.

The used algorithm is same with $CA$ algorithm that has been explained in section 2 but instead of using $CA$ rules in the reservoir the covariance operator $C_k$ will be used to produce the covariance evolution states that will be used as a feature space to estimate the output.
4 Stack Representation

In distributed representation of CA, there is a relation between time steps evolution with each other and with the initial state, but in covariance representation there is no relation between time steps evolution with each other, there is only a relation with the initial state. To complete the story, Stack Representation is proposed, in this representation, the input sequence is memorized consecutively as shown in Fig. 7. Therefore there is no any iterations in Stack representation as in CA and Covariance representations.

![Fig. 6](image)

The input sequence, there is a relation between time steps evolution with each other and with the initial state, but in covariance representation there is no relation between time steps evolution with each other, there is only a relation with the initial state.

![Fig. 7](image)

The T evolution states shown in Fig. 7 will be used as feature space.
5 Experiments

In the experiments we trained RNNs, using ReCA, Covariance and Stack, on various pathological synthetic tasks that will be listed in next Sect. 5.1. A set of $N_{\text{train}}$ (No of training examples) and $N_{\text{test}}$ (No of test examples) input time series and its associated outputs (Target) is synthesized for each task using the code provided from [22].

The experiments target is to achieve a zero test error (there are no False Bits in a predicted output), i.e. the maximal absolute error over all relevant times and output bits is smaller than 0.5, because we deal with a binary numbers.

The parameters $I$ (No of CA evolution iterations), $R_i$ (No of bits that represent each input bit ) and $N_{\text{train}}$ (No of training set examples) should be tuned to their minimum values using grid search to achieve a zero test error.

Normal equation based linear regression implemented by pseudo-inverse is used in read-out stage, and for the sequence length $T$ is constant in both training and test sets.

The Linear (additive) ECA rules 90 and 150 have been used in the reservoir, while the non linear ECA rules 110, 122 and 40 have been used in multilayer expansion. Rule 90 outperforms rule 150 in memory tasks, due to its features of reducing the interference between nonzero elements. Some examples of the evolution of these rules are illustrated in Fig. 8.

---

5.1 Pathological Synthetic Tasks

In order to test for long-short-term-memory capability of the CA based RC algorithm, we have used the pathological synthetic problems (tasks) proposed by Hochreiter and Schmidhuber in [7] that all exhibit pathological long term dependencies known to be effectively impossible for gradient descent (e.g., Hochreiter et al. [24] in 2001) [23]. These tasks have also been used in many RNN studies, Martens and Sutskever [22] in 2011, Jaeger [22] in 2012 and Pascanu et al. [25] in 2013.

This tasks can be divided into 3 categories: 1. Memory Task (5 bit, 20 bit and

---

\[ \text{Fig. 8. One bit ECA evolution for rules a. 90, b. 150 and c. 110.} \]

---

\[ ^{11} \text{In all experiments } N_{\text{test}} = 100. \]
Random permutation), 2. Temporal order task (2 and 3 symbols) and 3. Arithmetic and logic operation Tasks (XOR, Addition and Multiplication). In this tasks the input data consists of two parts: the Information and the Distractor period. The Distractor period is selected randomly, and does not contain any information, to give extra time steps and make the task harder with longer range dependencies. The output is either a particular function of the inputs information (XOR, Addition, Multiplication and Temporal order tasks) or repeating the same information (from previous time steps) at final time steps (Memory and Random permutation tasks). The difficulty of these tasks increases with the value of time steps $T$, because of longer $T$ exhibits longer range dependencies. The following sections lists these rules in detail.

**Random Permutation Task.** The input in this task is a binary vector of length 100. At each time step, one of the 100 bits gets a “1” value; i.e. we have a 100-symbol place coding, so $L_{in} = 100$.

The task has length $T$. At first time step $n = 1$, one of the first two input bits is randomly assigned to be “1”. For the remaining time steps $n = 2, \ldots, T$, at each time step, one of the 98 remaining inputs is set to be “1” in a random fashion.

The relevant output is at the last time step $n = T$: the target output is the 100-dimensional input vector at time step $n = 1$, thus $L_{out} = 100$.

**Temporal Order Task.** Temporal order task consists of two parts; according to the number of ordered events two or three.

2 Symbols Task. The input vector in this task is six bit length ($L_{in} = 6$), the first two inputs are for a critical event ($A = \text{“01”}$) or ($B = \text{“10”}$) and the last four inputs are distractor. All runs have the same duration $T$. There are two critical event times $T_1$ and $T_2$ associated with critical events A and B. At all time steps except $T_1$ and $T_2$, exactly one of the four distractor inputs is set to “1”. At $n = T_1$ the first or second input is randomly set to “1”; the same is executed for time step $n = T_2$.

The output is 4 bits $L_{out} = 4$. Only at the last time step $n = T$ the output is evaluated. The target output at that time is one of the four possible indicator outputs $(1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0)$ and $(1, 0, 0, 0)$ in agreement with the four possibilities of what the input at times $T_1$ and $T_2$ was. Figure 9 illustrates this task in detail.

3 Symbols Task. The 3 Symbol temporal order task is completely analogous to the 2 symbols temporal order task, except that in this task there are three critical times $T_1, T_2$ and $T_3$, when one of the first two inputs gets a randomly assigned “1”. This makes for 8 possible events, which have to be classified by the 8 bits output of the trained system at the last time step $n = T$, thus the input dimension still six $L_{in} = 6$, but the output will change to $L_{out} = 8$. 
Fig. 9. Temporal Order 2 Symbols Task.

Memory Task.

5 Bit Task. In this task there are four binary input bits and four binary output bits, thus \( L_{in} = L_{out} = 4 \). The first two bits of each of the inputs or outputs carry the memory pattern which is a 2-dimensional vector with 5-time steps, so \( D = 2 \) and \( M = 5 \) as shown in figure [10]. The total length of this task is \( T = T_d + 2M = T_d + 10 \), where \( T_d \) is the length of distractor time steps. The third bit in the inputs is “1” from \( n = 6 \) to \( n = T \) except at \( n = T_d + 5 \). The fourth input bit carries the cue, so it is “1” only at \( n = T_d + 5 \).

For the output, the third output bit should signal the waiting for recall cue condition, so it is “1” from \( n = 1 \) to \( n = T_d + 5 \) and the fourth output bit is unused and should always be zero.

The input sequence is constructed as follows: for the first 5 time steps, one of the first two input bits is randomly set to “1”. Note that there are altogether \( 2^5 = 32 \) possible input patterns, hence this task is named the 5-bit memory task. After completing the third and forth input bits as mentioned above, we find that at every time step exactly one of the four inputs is “1” as shown in figure [10].

The target output is always “0” on all bits except for time steps from \( n = 1 \) to \( n = T_d + 5 \), where it is “1” on the third bit. For time steps from \( n = T_d + 6 \) to the end i.e. \( n = T \) where the input memory pattern is repeated in the first two output bits as shown in figure [10].

Notice that there are only 32 different sequences possible for a given \( T_d \), thus in this task, we have only \( N_{train} = 32 \).

20 Bit Task 20-Bit memory task is structurally identical to the 5-bit memory task as illustrated in previous subsection. But now, the memory pattern is more
complex. Instead of 2-dimensional with 5-time steps in 5-bit task, the memory pattern is 5-dimensional with 10-time steps, so $D = 5$ and $M = 10$ as shown in figure 11. Thus in this task $L_{in} = L_{out} = 7$ and the total length is $T = T_d + 20$, where $T_d$ is the length of distractor time steps. Notice that there are a set of $5^{10}$ different possible patterns, which is a bit more than 20 bit information per input sequence and has given this task its name.

**XOR, Addition and Multiplication Tasks.** These three tasks are same; the difference is only for the used operation XORing, addition or Multiplication. As an example the XOR task is explained in the next section.

**XOR Task** In this task there are two inputs $L_{in} = 2$; The first input $u_1(n)$ is a stream of zeros or ones distributed randomly. The second stream is zeros $u_2(n) = 0$ at all time steps except at two time steps ($n = T_1$ and $n = T_2$) and $u_2(T_1) = u_2(T_2) = 1^{12}$ The objective is that at the end of run (much later than $T_1$ or $T_2$), the output is $u_1(T_1) \oplus u_1(T_2)$, thus the output length $L_{out} = 1$ as shown in figure 12. The task has a sequence length $T$ (No of all time steps).

### 5.2 Binary Encoded Tasks

The inputs of all tasks in [22] have only one nonzero for each time step. The binary encoding is applied to the classical inputs; the location of nonzero element

---

12 The second input bit is a cue signal for localization of $u_1(n)$.
Fig. 11. 20-Bit Task.

Fig. 12. XOR Task.
will be represented by a binary number as shown in Fig. 13, to make the tasks input closer to real data, but this leads the tasks to become harder. These new hard tasks have been used to test our RNN algorithm.

The binary encoding will only be used for Random Permutation and 20 Bit tasks, because the input of the other tasks is very small, so we do not need this encoding.

\[
\begin{array}{c|c|c}
\text{Original Input 5 Bits} & \text{Encoding} & \text{3 Bits} \\
0 0 0 0 1 & \rightarrow & 0 0 1 \\
0 0 0 1 0 & \rightarrow & 0 1 0 \\
0 0 1 0 0 & \rightarrow & 0 1 1 \\
0 1 0 0 0 & \rightarrow & 1 0 0 \\
1 0 0 0 0 & \rightarrow & 1 0 1 \\
\end{array}
\]

\textbf{Fig. 13.} Binary encoding for payload inputs of 20 bit task, the length is reduced from 5 to 3; thus the total input and output length is reduced from 7 to 5.

5.3 Results and Discussion

**General.** Table 1 shows that: the ReCA framework has solved all pathological tasks \textbf{a. directly}, as in random permutation, 5 Bit and temporal order tasks, or \textbf{b.} by expanding the input either using \( R_i \) to reduce the interference between input bits in 20 bit task or using multilayer CA expansion as in addition, multiplication and XOR tasks. Covariance and Stack representations solve only memory and random permutation tasks, but Stack fails in binary encoded 20bit task, although it solves addition, multiplication after using multilayer CA expansion, also the Stack achieves zero test error up to 1000 time steps with \( N_{\text{train}} = 5 \) for 20 bit task which is a very hard task for training in last studies [22][23][25], so the simplicity of the model doesn’t mean uselessness, it is dependent on the application. Therefore, we can conclude that the ReCA has the highest computational power. This superiority of ReCA due to its distributed representation and higher attribute statistics.

ESN could not solve the binary encoded 20 Bit task, even when we have tried to improve the accuracy by increasing the reservoir size \( K \) and changing the weights of input matrix, as shown in Table 2.

**Effect of \( N_{\text{train}} \).** Table 3 shows the effect of increasing \( N_{\text{train}} \) for random permutation and 20 Bit tasks using rule 90, where the increasing of \( N_{\text{train}} \) decreases the number of false predicted bits, i.e. improves the accuracy.
Table 1. Results for Pathological tasks using all proposed training methods for $N_{test} = 100$, the last column (red and bold) is the results, the others are the parameters. Where, $T$ is the sequence length of task input, $I$ is the iterations of CA evolution, $N_{train}$ is the number of training example, %$N_{train}$ is the ratio between training examples and all input possibilities, $f$ is the expansion ratio, $R_t$ is the number of bits to represent each bit of task input, No of False Bits is the number of false bits in predicted output. 

* The number between brackets is the number of iterations are used in multilayer CA. 

** The feature vector for regression consists of last CA evolution state from all time steps.

<table>
<thead>
<tr>
<th>Pathological Task</th>
<th>Method</th>
<th>Multilayer ECA rule Expansion</th>
<th>$T$</th>
<th>$I$</th>
<th>$N_{train}$</th>
<th>%$N_{train}$</th>
<th>$f$</th>
<th>$R_t$</th>
<th>No of False Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 Bit</td>
<td>ECA Rule 90</td>
<td>- 1000</td>
<td>4</td>
<td>32</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>20 Bit Normal</td>
<td>ECA Rule 90</td>
<td>- 500</td>
<td>20</td>
<td>120</td>
<td>1.23E-05</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Covariance</td>
<td>- 500</td>
<td>19</td>
<td>120</td>
<td>1.23E-05</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stack</td>
<td>- 1000</td>
<td>5</td>
<td>532</td>
<td>5.12E-07</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>20 Bit Binary</td>
<td>ECA Rule 90</td>
<td>- 200</td>
<td>24</td>
<td>250</td>
<td>2.56E-05</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Encoded</td>
<td>Covariance</td>
<td>- 100</td>
<td>19</td>
<td>250</td>
<td>2.56E-05</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stack</td>
<td>- 100</td>
<td>500</td>
<td>512</td>
<td>5.12E-05</td>
<td>-</td>
<td>-</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>Random Permutation</td>
<td>ECA Rule 90</td>
<td>- 1000</td>
<td>2</td>
<td>200</td>
<td>200</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Covariance</td>
<td>- 1000</td>
<td>2</td>
<td>200</td>
<td>200</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stack</td>
<td>- 1000</td>
<td>1200</td>
<td>1200</td>
<td>1200</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Temporal Order</td>
<td>ECA Rule 150</td>
<td>- 500</td>
<td>8</td>
<td>6000</td>
<td>1.20E-02</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2 Symbols</td>
<td>Covariance</td>
<td>- 200</td>
<td>8</td>
<td>6000</td>
<td>7.54E-02</td>
<td>1</td>
<td>1</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stack</td>
<td>- 200</td>
<td>6000</td>
<td>6000</td>
<td>7.54E-02</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ECA Rule 150</td>
<td>122 (1)</td>
<td>1000</td>
<td>8</td>
<td>8000</td>
<td>4.00E-03</td>
<td>0.5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Covariance</td>
<td>122 (1)</td>
<td>50</td>
<td>8</td>
<td>1500</td>
<td>3.06E-01</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stack</td>
<td>122 (1)</td>
<td>500</td>
<td>500</td>
<td>8000</td>
<td>2.81E-02</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ECA rule 90**</td>
<td>- 50</td>
<td>24</td>
<td>7000</td>
<td>4.46E-02</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3 Symbols</td>
<td>ECA rule 90**</td>
<td>- 200</td>
<td>16</td>
<td>2000</td>
<td>1.90E-04</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Addition 

<table>
<thead>
<tr>
<th>Method</th>
<th>Multilayer ECA rule Expansion</th>
<th>$T$</th>
<th>$I$</th>
<th>$N_{train}$</th>
<th>%$N_{train}$</th>
<th>$f$</th>
<th>$R_t$</th>
<th>No of False Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECA rule 150</td>
<td>- 40 (1)</td>
<td>500</td>
<td>20</td>
<td>2500</td>
<td>3.93E-15</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Covariance</td>
<td>110 (1)</td>
<td>50</td>
<td>20</td>
<td>3300</td>
<td>2.39E-15</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td>110 (2)</td>
<td>1000</td>
<td>3.30</td>
<td>3500</td>
<td>2.39E-15</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Multiplication 

<table>
<thead>
<tr>
<th>Method</th>
<th>Multilayer ECA rule Expansion</th>
<th>$T$</th>
<th>$I$</th>
<th>$N_{train}$</th>
<th>%$N_{train}$</th>
<th>$f$</th>
<th>$R_t$</th>
<th>No of False Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECA rule 150</td>
<td>- 110 (2)</td>
<td>500</td>
<td>12</td>
<td>2000</td>
<td>1.45E-15</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Covariance</td>
<td>110 (2)</td>
<td>50</td>
<td>12</td>
<td>2000</td>
<td>1.45E-15</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td>110 (2)</td>
<td>1000</td>
<td>14000</td>
<td>1.45E-15</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

XOR 

<table>
<thead>
<tr>
<th>Method</th>
<th>Multilayer ECA rule Expansion</th>
<th>$T$</th>
<th>$I$</th>
<th>$N_{train}$</th>
<th>%$N_{train}$</th>
<th>$f$</th>
<th>$R_t$</th>
<th>No of False Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECA rule 150</td>
<td>- 40 (1)</td>
<td>1000</td>
<td>4</td>
<td>500</td>
<td>3.23E-15</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Covariance</td>
<td>110 (2)</td>
<td>50</td>
<td>4</td>
<td>3200</td>
<td>2.32E-15</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td>110 (2)</td>
<td>50</td>
<td>3200</td>
<td>2.32E-15</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1- For 20 Bit task $D$ is the length of memory pattern and $M$ is the number of memory pattern time steps. 2- In addition and multiplication tasks the input is binary number, in future work the real numbers will be used after binarizing them, so the number of input bits should be extended.
Table 2. 20 Bit Binary Encoded Input using ESN: No of false bits for different values of reservoir size $K$ at $T_d = 200$, $N_{\text{train}} = 500$ and $N_{\text{test}} = 100$. **Left:** Blind and **Right:** Basic, where the input matrix weights $\sigma_{1,2,3} = 2.5 \times 10^{-6}$, $\sigma_4 = 1 \times 10^{-6}$ and $\sigma_5 = 1$; for more details about blind, basic conditions, reservoir size ($K$) and input matrix weights ($\sigma$) in ESN see [22].

<table>
<thead>
<tr>
<th>$K$</th>
<th>No of False Bits</th>
<th>$K$</th>
<th>No of False Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>50</td>
<td>2000</td>
<td>27</td>
</tr>
<tr>
<td>4000</td>
<td>46</td>
<td>4000</td>
<td>15</td>
</tr>
<tr>
<td>6000</td>
<td>27</td>
<td>6000</td>
<td>24</td>
</tr>
<tr>
<td>8000</td>
<td>25</td>
<td>8000</td>
<td>66</td>
</tr>
<tr>
<td>10000</td>
<td>25</td>
<td>10000</td>
<td>59</td>
</tr>
<tr>
<td>15000</td>
<td>81</td>
<td>15000</td>
<td>55</td>
</tr>
</tbody>
</table>

Table 3. The effect of increasing $N_{\text{train}}$ on the number of False Bits using rule 90 in ReCA, where $N_{\text{test}} = 100$ and $f = 1$ for **Left.** Random permutation task: where, $T = 1000$ and $I = 2$, **Right.** 20 Bit task: where, $T_D = 300$, $I = 20$ and $R_i = 2$.

<table>
<thead>
<tr>
<th>$N_{\text{train}}$</th>
<th>False Bits</th>
<th>$N_{\text{train}}$</th>
<th>False Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>22</td>
<td>50</td>
<td>4</td>
</tr>
<tr>
<td>150</td>
<td>2</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>200</td>
<td>0</td>
<td>120</td>
<td>0</td>
</tr>
</tbody>
</table>

**Effect of Expansion Ratio $f$.** The increasing of expansion ratio $f$ reducing the number of false bits as shown in Fig. 14. The zero error is obtained at $f = 0.4$ in 2 Symbol task, that means; we can reduce the feature space (i.e. the space and time complexity will also be reduced) to 0.4 of its maximum value without losing the model accuracy.

**Multilayer CA expansion.** All non linear ECA rules in class IV that have a complex behavior give zero error as shown in Table 4 thus they can be used for input multilayer expansion with some of rules that in other classes. But, on the contrary, all linear (additive) ECA rules can not be used in multilayer expansion as listed in Table 4.

**Comparison with previous approaches.** ReCA outperforms: 1. Martens and Sutskever (2011) [23] and Pascanu et al. (2013) [25] in sequence length $T$, in their studies the zero test error has been obtained for $T$ ranging from 50 to 200, but in our experiments $T$ ranging from 200 to 1000. 2. Jaeger (2012) [22], where the zero test error could not be achieved in 20 Bit binary encoded task using ESNs. ReCA outperforms ESN in computational complexity for most of the tasks as listed in Table 5. There is more than $100 \times$ speedup/energy savings for memory tasks and $8 \times$ for temporal order tasks, but ESN is $2 \times$ faster for XOR task. Comparing the representations of ReCA and ESN, the computation performed in the ReCA is much more transparent for analysis and improvement.
Fig. 14. 2 Symbols Task: The number of False Bits w.r.t. changing of expansion ratio \( f \), where \( T = 50, I = 16, N_{\text{train}} = 900 \) and \( N_{\text{test}} = 100 \) using rule 150 in ReCA.

<table>
<thead>
<tr>
<th>CA Class</th>
<th>CA Rule</th>
<th>No of False Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>250</td>
<td>73</td>
</tr>
<tr>
<td>II</td>
<td>170</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>204</td>
<td>77</td>
</tr>
<tr>
<td>III</td>
<td>60</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>150</td>
<td>80</td>
</tr>
<tr>
<td>IV</td>
<td>41, 54, 106, 110</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4. ECA rules are used in Multilayer expansion for XOR task; all Linear (additive) rules fail to obtain zero error, but all rules in Class IV (Complex behavior) obtain zero error, also some of non linear rules from other classes.
compared to ESNs, in which the state evolution is untraceable due to random and irregular distributivity.

<table>
<thead>
<tr>
<th>Task</th>
<th>ESN (Floating Point ⇒ Bit)</th>
<th>CA (Bit)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 Bit, $T_d = 200$</td>
<td>105.6 M ⇒ 3380 M Bit</td>
<td>24.8 M Bit</td>
<td>136X</td>
</tr>
<tr>
<td>3 Symbols, $T = 200$</td>
<td>5 M ⇒ 160 M Bit</td>
<td>20.5 M Bit</td>
<td>7.8X</td>
</tr>
<tr>
<td>XOR, $T = 1000$</td>
<td>0.2 M ⇒ 6.4 M Bit</td>
<td>16 M Bit</td>
<td>0.4X</td>
</tr>
</tbody>
</table>

Table 5. The comparison of the number of operations and correspondence required bits between ESN and ReCA framework for some pathological tasks.

6 Conclusion

We show that the proposed algorithm ReCA achieves zero error in all pathological synthetic tasks that have long term dependencies.

Cellular Automata Reservoir framework (ReCA) constructs a novel bridge between computational theory of automata and recurrent neural architectures. We show that the ReCA achieves zero error in all pathological synthetic tasks of sequence learning. Sequence learning is an essential capability for a wide collection of intelligence tasks such as language, continuous vision (i.e. video), symbolic manipulation in a knowledge base etc. ReCA outperforms the Covariance and Stack representations because it enables both computation of high order statistics and distributedness, where Stack representation provides pure memorization, and Covariance representation computes only second order statistics.

Usage of cellular automaton instead of real valued neurons in reservoir computing framework greatly simplifies the architecture, and makes the computation more transparent for analysis, and provide enough computation for sequence learning, even though it is a very sparse network with identical binary units. Increasing cellular automata iterations $I$, and the size of each input entry $R_i$, reduces interference in the reservoir in a predictable manner, making the computation more similar to a feedforward architecture (i.e. Stack representation).

After these success results, we think that RNNs which are very powerful, but very hard to train, may now we start to exploit their high potential and see more applications to solve the challenging problems of sequence modeling.

References


