Norwegian University of
Science and Technology

Investigation of Elementary Cellular Automata for Reservoir Computing

Emil Taylor Bye

Master of Science in Computer Science
Submission date: June 2016
Supervisor: Stefano Nichele, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Summary

Reservoir computing is an approach to machine learning. Typical reservoir computing approaches use large, untrained artificial neural networks to transform an input signal. To produce the desired output, a readout layer is trained using linear regression on the neural network.

Recently, several attempts have been made using other kinds of dynamic systems instead of artificial neural networks. Cellular automata are an example of a dynamic system that has been proposed as a replacement.

This thesis attempts to discover whether cellular automata are a viable candidate for use in reservoir computing. Four different tasks solved by other reservoir computing systems are attempted with elementary cellular automata, a limited subset of all possible cellular automata. The effect of changing different properties of the cellular automata are investigated, and the results are compared with the results when performing the same experiments with typical reservoir computing systems.

Reservoir computing seems like a potentially very interesting utilization of cellular automata. However, it is evident that more research into this field is necessary to reach performance comparable to existing reservoir computing systems.

Acknowledgements

I would like to express my very great appreciation to my supervisor, Dr. Stefano Nichele. His insights, guidance and encouragement has proven invaluable and vital to the completion of this thesis. I would also like to offer my special thanks to Solveig Isabel Taylor, who apart from being an excellent proofreader, also did a marvelous job at helping me search for literature. Tore Bye and Marte Taylor Bye have also provided me with much needed feedback after proofreading drafts of my thesis. Finally, a big thanks to Hanna Holm who has kept my morale up during the last few weeks of work to get this thesis completed.

Table of Contents

Summary i

Acknowledgements iii

Table of Contents vi

1 Introduction 1

1.1 Assignment text . . . 1

1.2 Structure of this thesis . . . 2

2 Background 3

2.1 Reservoir computing . . . 3

2.1.1 The development of reservoir computing . . . 3

2.1.2 Physical reservoirs . . . 5

2.1.3 The readout layer . . . 6

2.1.4 Challenges in reservoir computing . . . 6

2.2 Cellular automata . . . 7

2.2.1 Elementary Cellular automata . . . 7

2.2.2 Cellular automaton behavior . . . 8

2.2.3 The edge of chaos . . . 9

2.2.4 Uses of cellular automata . . . 9

2.3 Cellular automata in reservoir computing . . . 10

2.3.1 Why use cellular automata with reservoir computing . . . 10

3 Methodology 11

3.1 The experimental framework . . . 12

3.1.1 The reservoir . . . 13

3.1.2 The readout layer . . . 13

4 Experiments 15
4.1 The 5-bit memory task . . . 15
4.1.1 The problem . . . 15
4.1.2 Solving the problem . . . 16
4.1.3 Results . . . 16
4.1.4 Discussion . . . 16
4.2 30th order NARMA . . . 19
4.2.1 The problem . . . 19
4.2.2 Solving the problem . . . 19
4.2.3 Results . . . 20
4.2.4 Discussion . . . 20
4.3 Japanese vowels dataset . . . 25
4.3.1 The problem . . . 25
4.3.2 Solving the problem . . . 25
4.3.3 Results . . . 26
4.3.4 Discussion . . . 26
4.4 Temporal bit parity and density . . . 29
4.4.1 The problem . . . 29
4.4.2 Solving the problem . . . 30
4.4.3 Results . . . 34
4.4.4 Discussion . . . 34

5 Analysis 37
5.1 Performance of individual cellular automaton rules . . . 37
5.2 Viability . . . 39
5.2.1 Resource constraints . . . 39
5.2.2 Input/output restrictions . . . 39

6 Conclusion 41
6.1 Computational capabilities . . . 41
6.2 Performance . . . 41
6.3 Summary . . . 42
6.4 Further work . . . 42

Bibliography 43

Chapter 1

Introduction

Reservoir computing represents a trend in machine learning where input is transformed by processes in dynamical systems called reservoirs. The learning happens by estimating weights for a linear combination of the perturbed input signal in the reservoir. Current research have mostly been done on untrained artificial neural networks as reservoirs. Recently, however, more work has been done on exploring alternative reservoirs.

Cellular automata are discrete models studied in various fields of research. Complex patterns emerge from seemingly simple automata.

This thesis will look at existing research on using cellular automata in a reservoir computing framework and expand on possibilities for further research on the subject.

1.1 Assignment text

> Investigation of cellular automata for Reservoir Computing. The investigation focus on 1 dimensional elementary (Boolean) cellular automata. Reservoir Computing projects an input onto a nonlinear dynamical system (reservoir with an expressive and discriminating space) and trains a single readout layer to read the output. In this project, state of the art Reservoir Computing will be studied using elementary cellular automata.

Based on the assignment text, a hypothesis and two research questions have been made.

Hypothesis: Cellular automata are able to act as performant reservoirs for tasks traditionally being solved by a reservoir computing approach.

Research question: Are cellular automata capable of acting as reservoirs for tasks that typically are solved by conventional reservoir computing approaches?

Research question: How does changing properties of a cellular automata reservoir affect its performance?

In order to investigate this, a small subset of all cellular automata, elementary cellular automata, are used. The results do therefore not necessarily represent the best possible

results with cellular automata, but should give a clue as to the capabilities of cellular automata.

1.2 Structure of this thesis

This thesis is divided into six chapters.

1. Introduction: Introduces the research questions and motivation for writing this thesis.

2. Background: Overview of current research in fields relevant for this thesis.

3. Experimental methodology: A brief overview of the experiments that will be run and a detailed explanation of the framework that will be used to run the different experiments.

4. Experiments: The actual experiments and specific methodology for the different experiments. Every experiment includes results and analysis specific to each experiment.

5. Analysis: Comparison and analysis of the results from the different experiments.

6. Conclusion: A summary of the results and how they relate to the research questions, including a list of further work.

Chapter 2

Background

2.1 Reservoir computing

Reservoir computing is a framework for machine learning originally inspired by artificial neural networks.

Figure 2.1 illustrates the main components of a reservoir computing system. An input signal is projected to a dynamical system, which then transforms the input according to the dynamics of the system. The final output is then generated, usually as a linear combination of the states of the dynamical system. This means that in many cases, training a reservoir computing system can be performed relatively easy using some form of linear regression.

2.1.1 The development of reservoir computing

Reservoir computing has developed as a combination of several approaches to machine learning that were discovered independently. The first approaches were Echo State Networks by Jaeger (2001), Liquid State Machines by Maass et al. (2002). Table 2.1 displays the number of publications on reservoir computing since 2002, showing how interest in the field has developed the last 15 years.

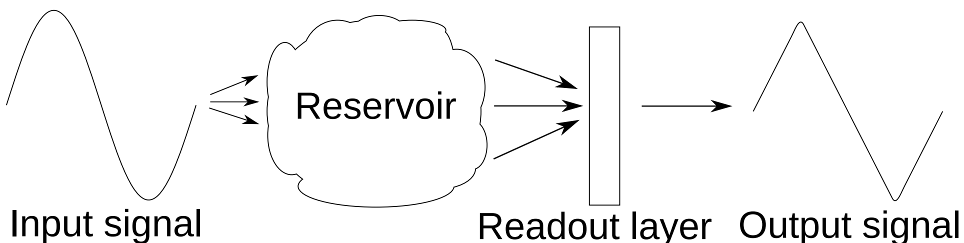

Figure 2.1: Illustration of the main components of a reservoir computing system.[1]

[1] All figures in this chapter were made by Emil Taylor Bye to illustrate the background material.

Table 2.1: Number of publications found in the Scopus database when searching titles, abstracts and keywords for "reservoir computing", "echo state network" and "liquid state machine". The search was performed 14, 2016. Note that the number of publications for echo state network and liquid state machine do not include results that show up when searching for reservoir computing.

	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016
Reservoir computing	0	0	0	0	0	10	22	31	36	33	56	67	71	60	24
Echo state network	0	3	8	17	15	30	36	40	58	47	81	71	83	98	32
Liquid state machine	1	3	5	7	5	16	6	7	8	14	9	8	7	6	2
Total	1	6	13	24	20	56	64	78	102	94	146	146	161	164	58

Echo state networks

Echo state networks are recurrent neural networks with randomly generated input and recurrent connections first introduced by Jaeger (2001). In echo state networks, only the output weights are trained, while the input and recurrent connections in the network remain fixed. Most of the publications in the field of reservoir computing has been about echo state networks (see Table 2.1).

Echo state networks are named after the echo state property, which is deemed necessary for echo state networks to work. The echo state property states that a network should gradually lose information from earlier input. Specifically, if two networks with the same internal dynamics, but in different states are repeatedly given the same input, they should converge to the same state.

In order for an echo state network to have the echo state property, the recurrent weights have to be scaled after being randomly generated. Yildiz et al. (2012) proposes the following recipe in order to ensure the echo state property in echo state networks:

1. Initialize the recurrent weights W with non-negative entries, that is $w_{ij} \geq 0$.
2. Scale W so that $\rho(W) < 1$, where $\rho(W)$ is the spectral radius of W.
3. Change the signs of some of the entries in W to get negative connection weights as well. The number of negative weights may impact performance, but the optimal number of negative weights varies from problem to problem.

Liquid state machines

Liquid state machines are, like echo state networks, artificial neural networks with randomly initialized weights. The difference lies in that liquid state machines are based on spiking neural networks. Instead of transmitting information between neurons continuously, every cell has its own threshold, and it is first when a cell's threshold has been reached, that it will "fire" and propagate information to the nodes it is connected to. Maass et al. (2002) introduced this concept inspired by biological neural networks.

In order for liquid state machines to be computationally useful, they have to have the point-wise separation property. For liquid state machines, this is ensured by using spiking neurons with dynamic synapses.

In order to get output from a liquid state machine, a linear combination of the neuron states are used. This output layer is typically trained using some form of linear regression..

Subsequent work

Steil (2004) expanded on the training rule for recurrent neural networks presented by Atiya and Parlos (2000), creating a new training rule called the backpropagation-decorrelation rule. It was observed that most of the changes ended up happening in the weights to the output nodes, so training of the internal weights was removed for a massive performance gain, down to $O(N)$ from $O(N^2)$. The result ended up being very similar to echo state networks, with the main difference being that the output is being fed back into the network in the case of the backpropagation-decorrelation rule.

The different approaches originate from different fields of research, however they all shared a similar aim. It was observed that for many tasks, updating the internal dynamics of a randomly initialized recurrent neural network is not necessary to achieve good performance. Instead, the desired output could be achieved by training a simple, memoryless output layer.

Since it was first discovered, reservoir computing has been able to achieve better performance than recurrent neural networks on certain benchmarks. Jaeger et al. (2007) used an echo state network for analyzing sound clips of nine Japanese men uttering a single vowel multiple times, correctly predicting who uttered the vowel for every sound clip. Previously, an error rate of 1.8% was the best achieved on that data set. Ilies et al. (2007) used an echo state network to win the 2007 NN3 Artificial Neural Network and Computational Intelligence Forecasting Competition.

ORGANIC (Self-Organized Recurrent Neural Learning for Language Processing) was a research project for speech and handwriting recognition using reservoir computing funded by the European Union. It lasted from 2009 to 2012 resulting in 38 publications.

2.1.2 Physical reservoirs

Most of the research done on reservoir computing has been with reservoirs implemented on conventional computers. However, the research has also created a foundation for exploring ways to solve problems by exploiting the physical properties of different substances.

Fernando and Sojakka (2003) managed to perform speech recognition using waves in a bucket of water, distinguishing between recordings of the word “zero” and “one” with an error rate of 1.5%. The experiment was motivated by liquid state machines, using the water as an artificial neural network, and a linear readout layer to produce the desired output from the waves in the water.

Inspired by liquid state machines, several attempts have also been made at implementing XOR gates using biological systems along with a linear readout layer. Jones et al. (2007) tried to implement a liquid state machine using Escherichia Coli bacteria. The input was given by exposing the bacteria to high and low temperatures, and acidic or basic chemicals. Nikolic et al. (2007) used readouts from the visual cortices of cats, where the input was given as visual stimuli to the cats.

Recently, more research has been done on exploiting various physical phenomena for use in reservoir computing. Appeltant et al. (2011) performed computations using simple electrical circuits as a reservoir. Specifically, a Mackey-Glass oscillator coupled with delayed feedback managed better performance than a liquid state machine with 1200 at correctly classifying recordings of pronunciations of single digits.

Instead of electricity, Vandoorne et al. (2014) used electromagnetic radiation as input to a reservoir consisting of exclusively passive components, namely waveguides, splitters and combiners. The reservoir itself did not consume any energy, and they reported that bandwidths above 100 GBit/second should be very feasible, especially for signal that already are optical.

2.1.3 The readout layer

In order to get results from a reservoir, the readout layer usually combines the states of the reservoir nodes with different weights. These weights are usually attained through some form of linear regression. Ideally this is done by solving the equation $\mathbf{Xw} = \mathbf{y}$. Typically the equation is not solvable, however, so the goal is to find the $\mathbf{w}$ that minimizes the error: $||\mathbf{Xw} - \mathbf{y}||^{\mathbf{2}}$.

Real data usually contains some noise, which may propagate to the reservoir states. When training a reservoir computing system, it is vital to avoid relying on specific noise patterns in the training data. One way to avoid this is to introduce additional noise to the data before performing the regression. This minimizes the chance that the regression finds random patterns of noise that it relies on. Instead of introducing additional noise to the input data, a viable alternative is to penalize large coefficients. In that case, the expression to minimize becomes $||\mathbf{Xw} - \mathbf{y}||^{\mathbf{2}} + \alpha||w||^2$, where $\alpha \geq 0$ is how much large weights should be penalized.

Wyffels et al. (2008) compared dynamic noise injection to ridge regression for use with reservoir computing. The conclusion was that ridge regression was superior in performance, and that optimizing it was easier than generating optimal noise for a given training set.

Triefenbach and Martens (2011) tested alternatives to linear regression for the readout layer. One of the alternatives was logistic regression, a form of regression related to linear regression. With logistic regression, the output is not a number on a continuous spectrum, but selected from a set of possible outputs. This means that logistic regression may be especially attractive for classification tasks.

2.1.4 Challenges in reservoir computing

Although reservoir computing systems have been showed viable for certain tasks, there is still much ongoing research to make reservoir computing a more viable alternative compared to other forms of machine learning.

Lukoševičius et al. (2012) summarized reservoir computing trends. Much of the research within the field of reservoir computing is being done on echo state networks (see Table 2.1). Intuitively, there should be a better way than randomly generating echo state networks. Additionally, there are still a lot of variables that have to be manually tuned to perform well at different tasks. Some kind of self-adapting reservoir could be possible, potentially increasing reservoir performance. The backpropagation-decorrelation already does this to some extent by feeding the output back into the reservoir, but it may be possible for a reservoir to perform unsupervised optimizations based on the input, but not the expected output.

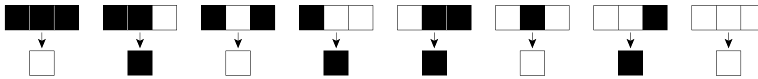

Figure 2.2: Rule table for elementary cellular automaton rule 90

The full extent of the capabilities of different reservoirs has not been explored yet. The "no free lunch"-theorem for supervised machine learning (Wolpert, 2002), when applied to reservoir computing, implies that no single type of reservoir is capable of solving all possible problems. However, where exactly does the limit go, for instance when trying to compute different outputs on the same input stream with two different readout layers?

Section 2.1.2 explored some of the work implementing reservoirs in physical systems. Various models are being explored for potential as reservoirs as well. Snyder et al. (2013) explored random Boolean networks, and Yilmaz (2014) used cellular automata.

2.2 Cellular automata

A cellular automaton is a discrete dynamical of *cells* placed on a regular grid. The dynamics of a cellular automaton is given by $\langle S, n, f \rangle$. S is the set of states a cell a be in, n is the size of a cell's neighborhood, and $f : S^{n+1} \rightarrow S$ is the local rule that governs state transitions in the automaton. Figure 2.2 displays an example of a rule for a cellular automaton with two states where a cell's neighborhood is the cell to the left and to the right of it.

Cellular automaton grids can have arbitrarily many dimensions, but most cellular automata use one, two or three dimensions. The neighborhood of a cell, that is the other cells that determine the next state of a cell, are typically the cells located adjacent to a cell in the grid, although this is not a requirement. However, the neighborhood should have the same structure for every cell.

In this thesis, all the cellular automata are discrete, both in that the cell's state is a discrete value, and that the cells are placed on a discrete grid. Additionally, the automata are synchronous, that is, the state of all the cells are updated at the same time. There have been work on both continuous and asynchronous cellular automata, however, neither is used in this thesis.

Cellular automata are usually uniform, that is, the same rule applies to every single cell. Cattaneo et al. (2009) experimented with non-uniform automata, however. A non-uniform automata is given by $\langle S, \{h_i, n_i\}_{i \in \mathbb{Z}} \rangle$ where $h_i : S^{n+1} \rightarrow S$. Every rule, h_i, control the state transitions for a fixed subset of all the cells in the automaton.

Cellular automata have been studied since the middle of the 20th century. Notable early work include Burks and Von Neumann (1966), where a 29-state cellular automata is shown to be able to support universal computation.

2.2.1 Elementary Cellular automata

Elementary cellular automata are a subset of all cellular automata, consisting of a one-dimensional grid with cells that can be in two states,usually referred to as 0 and 1. The

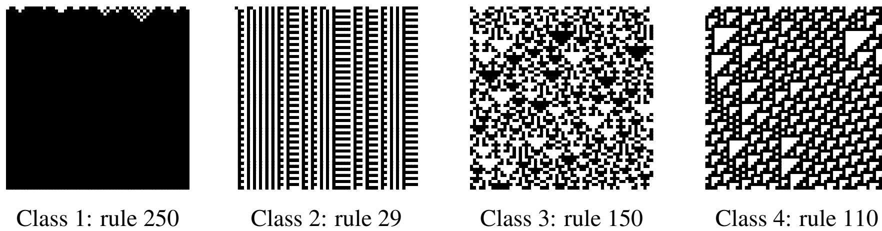

Figure 2.3: Wolfram's four different cellular automaton classes illustrated by four elementary cellular automata with random initial configurations. The top row contains the initial configuration of cells, and the following rows represent one time step each.

neighborhood of a elementary cellular automaton cell is its adjacent cells.

An elementary cellular automata rule depend on three cells, which can be in one of two states. For each of the $2^3 = 8$ combinations of cell states, there are two states a cell can end up in, which means that there are $2^8 = 256$ unique elementary cellular automata rules. According to Wolfram (2002, p. 57), of the 256 elementary cellular automata rules, only 88 are fundamentally inequivalent.

The 256 elementary cellular automata rules are usually numbered according to a specific scheme. Figure 2.2 displays the rule table for rule 90, where black cells are in state 1 and white cells in state 0. The rule number is then made by interpreting the result states as a 8-bit number, where the most significant bit is the next state of a cell if all the cells in its neighborhood is 1. Similarly, the least significant bit is the result state if all the cells in its neighborhood is 0.

Stephen Wolfram used elementary cellular automata in his book, A New Kind of Science (2002). In the book, elementary cellular automata are used to study how very simple systems can support advanced computations.

2.2.2 Cellular automaton behavior

Wolfram (2002, chap. 6) classifies cellular automaton behavior into four different classes in increasing order of complexity. Examples of the four classes can be seen in Figure 2.3.

1. Cellular automata that end up with all cells in the same state for most initial configurations.
2. Cellular automata that stabilizes to fixed or simple periodic structures.
3. Cellular automata that exhibit chaotic and aperiodic behavior.
4. Cellular automata with complex localized structures.

These classes describe different kinds of behavior in cellular automata, but they are qualitative in nature and a cellular automaton can not be easily classified for instance just by looking at its rule. Different configurations of cell states may also alter which class of behavior is seen in a cellular automaton. For instance, rule 110, which is said to be a class 4 automaton will behave like a class 1 automaton if all the cells are 0, as all the cells

will simply stay in the same state indefinitely unless cells are put into other states by an external signal.

A cellular automaton must be in class 3 or 4 to let cells that are far away from each other communicate.

In addition to the four classes, certain cellular automaton rules are *additive*. Additive cellular automata are rules that can be written as an addition of the states in a cell's neighborhood modulo the number of possible states. Elementary cellular automaton rule 90 is an example of such an additive rule, as it can be written as $(c_{-1} + c_1) \bmod 2$, where c_{-1} and c_1 are the cells before and after the current cell.

2.2.3 The edge of chaos

Wolfram's four classes of cellular automaton behavior is a qualitative description of cellular automaton behavior. Langton (1990) attempted to measure complexity of cellular automata by measuring how *chaotic* they were. The theory was that interesting computations happened in automata that were on the "edge of chaos".

A metric, λ, was devised to measure how chaotic different rules behaved. Lambda is measured as the percentage of possible inputs that lead to a state other than the *quiescent* state. The quiescent state may be chosen as any single valid state for a cellular automaton. $\lambda = 0$ means that an automaton stays in the quiescent no matter the initial condition, while $\lambda = 1$ means that an automaton is completely chaotic. The conclusion was that cellular automaton rules around $\lambda = 0.5$, literally half-way between solid state and chaos, exhibited the most complex behavior, and were most likely to belong in Class 4.

Calculating the λ of elementary cellular automata is trivial, and can be done by counting how many input combinations results in a "1", and divide this by 8. However, in this case $\lambda = 1$ is not chaotic, but rather stuck in the non-quiescent state.

2.2.4 Uses of cellular automata

Cellular automata have been widely studied, both specifically how to solve certain problems, but also to study how complex computations can emerge from simple systems.

Neary and Woods (2006) proved that there exists at least one elementary cellular automaton which can act as a Turing machine. They did that by proving that rule 110 could simulate a deterministic Turing machine in polynomial time, showing that it is P-complete. Another cellular automaton that has been shown to emulate a Turing machine is Conway's Game of Life, a two-dimensional cellular automaton. The proof of this was published by Rendell (2002).

Cellular automata have also been used as models to solve a variety of different problems, three different publications are included below to give some context to the extent of the usage of cellular automata. de Carvalho and de Oliveira (2015) used 1-dimensional cellular automata to implement a massively parallel sort. Xiao et al. (2011) explored cellular automata for use in bioinformatics, as a tool to both visualize and explore the significance of various patterns in proteins. Santé et al. (2010) reviewed different models that used cellular automata for simulations of urban growth.

2.3 Cellular automata in reservoir computing

The use of cellular automata in reservoir computing was first researched by Yilmaz (2014). A group of independent cellular automata are used as a reservoir. For each automaton there is a random mapping, mapping the input to cell states. The cellular automata then evolves for a number of time steps after receiving the input. The output layer is then trained with the states of all the cellular automata after every evolution.

This is a new approach both to cellular automata, and to reservoir computing. There is no previous published work where a linear combination of cellular automaton states has been used for computation. For reservoir computing, this represents a new kind of reservoir. Other than the preliminary experiments by Yilmaz (2014, 2015), there has not been published any articles on cellular automata in combination with reservoir computing.

2.3.1 Why use cellular automata with reservoir computing

Investigating the use of cellular automata in combination with reservoir computing can be interesting from several points of view.

It can help with further understanding and adapting reservoirs, providing a different point of view than the traditional neural networks. At the same time, it is another way to utilize cellular automata to perform advanced computations, possibly giving insight in how computations can happen in cellular automaton-like structures.

Yilmaz (2014) reported that cellular automata only used between $\frac{2}{3}$ and $\frac{1}{3}$ as many operations as an echo state network. The energy usage would also improve by two orders of magnitude, due to the fact that echo state network perform mainly floating point operations while cellular automata can be implemented very efficiently on FPGAs. On a custom implementation, linear regression may also be implemented more efficiently, as the matrix multiplication can be implemented using addition instead of multiplication.

There also exist possible applications in unconventional computing. Farstad (2015) managed to get a single-walled carbon nanotube and polymer composite random mesh to behave like various cellular automata rules. In combination with reservoir computing, it is possible that new types of computing devices can be manufactured.

Chapter 3

Methodology

In order to test the performance of cellular automata as reservoirs for reservoir computing, an attempt was made to solve four different problems. Parameters of the reservoirs are varied in each experiment to see how the different parameters affect the performance of the reservoirs.

The first experiment is to try to solve the 5-bit memory task. This is an attempt to recreate the results found by Yilmaz (2014). Afterwards, two common benchmarks for reservoir computing systems are performed, and the results compared with traditional reservoir computing approaches. At last, an attempt is made to solve to simple tasks simultaneously, using both traditional elementary cellular automata and non-uniform cellular automata. An in-depth explanation of the experiments is included in every experiment's section.

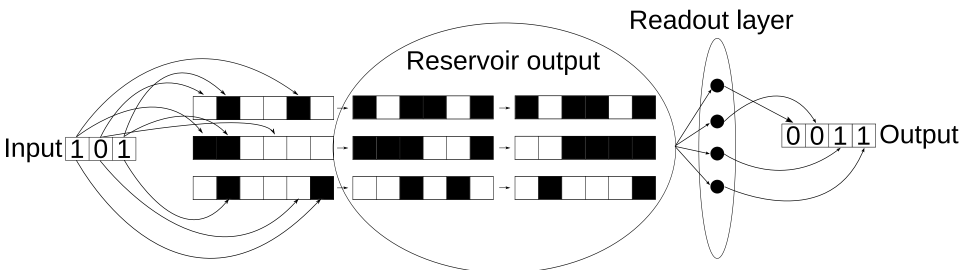

Figure 3.1: Illustration of a cellular automaton reservoir computing system. The input consists of 3 bits, which are mapped to R=3 automata. The automata are transformed I=2 times, and the cell states after each transformation used to produce the output of 4 bits. The transformation rule is rule 90.

3.1 The experimental framework

A framework was written in Python to conduct the experiments. The framework was implemented using the CPython 2.7.9 implementation of the Python programming language. It was modelled after the framework described by Yilmaz (2014). Figure 3.1 illustrates the layout of the framework.

The reservoirs used in this thesis consist of R elementary cellular automata of the same size and with the same rule. For each cellular automaton there is a random mapping, mapping each bit in the input to a single cell. All cells that do not have any input mapped to them are initialized to zero.

Elementary cellular automata rules are only defined for three cells. Instead of setting some boundary condition, cells were arranged in a circle so that every cell had two neighbors. This was done by letting the “first” cell have the “last” cell as it’s left neighbor, and the last cell had the first cell as it’s right neighbor.

After the input has been imprinted on the cellular automata, every automaton is transformed according to its rule I times. The reservoir output is the state of every cell in the R automata after all of the I transformations. This means that for every input, the reservoir output consists of $R \cdot I \cdot$ automaton size values, each either 0 or 1.

In most of the experiments in this thesis, the input is given as a time series. Elementary cellular automaton cells have no notion of memory other than their current state, so overwriting a cell state will potentially erase some of the automaton’s memory. However, if the cellular automaton is large enough, enough memory should hopefully be retained when overwriting cell states.

If the input is given as a time series, the process shown in Figure 3.1 is repeated for every time step. However, for any but the first time step, the input is not imprinted on cellular automata with all states set to 0, but on the automata as they were after the last transformation the previous time step.

The framework is modelled after that described by Yilmaz (2014), but there are a few key differences. For the experiments where the output is a time series, input is given one time step at a time, and the output is predicted for every time step. An alternative is to use sliding windows, were a number of time steps are given as input simultaneously to a new reservoir.

For the 5-bit memory task, Yilmaz (2014) used the entire input from all the time steps as input to the reservoir, and then tried to predict the entire output sequence at once. In this thesis, input is given one time step at a time, predicting the output at every time step. This is more in line with traditional reservoir computing approaches, and utilizes the dynamics of cellular automata to provide a form of fading memory instead of explicitly defining how many time steps input should be remembered for.

Another difference is the size of the automata. In this thesis, the automata used have more cells than the input requires. This is probably required in order to provide the automata with some memory which is not overwritten at every time step. Input size is varied for the experiments, to see how more cells affect the performance.

In this thesis all input is converted to bit strings. Yilmaz (2014) proposes an input scheme for non-binary input, however due to difficulties with implementing it, no attempt was made on using it in this thesis. The effect of this scheme is unknown, but it is certainly an interesting idea that should be tested.

3.1.1 The reservoir

The cellular automata were first implemented in Python. Rule numbers were converted to lookup tables by reversing the process described in Section 2.2.1. The lookup tables for all the 256 rules were then manually inspected to see that they had been converted correctly. Finally, a few rules were tested additionally by randomly initializing automata and manually verifying that cells were transformed according to the given rule.

Later, the Python implementation was replaced with an implementation written in C in order to decrease the run time. Functionality in the C implementation was verified by comparing randomly initialized automata before and after transformation for all 256 rules, with the Python implementation.

3.1.2 The readout layer

For the readout layer, the Python library scikit-learn 0.17 was used. More details on the library can be found in Pedregosa et al. (2011). Scikit-learn was chosen for ease of use, as it contains well documented code for various types of both offline and online learning.

Chapter 4

Experiments

4.1 The 5-bit memory task

4.1.1 The problem

The 5-bit memory problem is a problem first described by Hochreiter and Schmidhuber (1997) as a problem that is hard to solve for recurrent neural networks. A system is given four inputs a_1,a_2,a_3 and a_4 which can be either 0 or 1 at every time step. The first five time steps either a_1 or a_2 is set to 1, and the other to 0. After the first five time steps, the distractor signal a_3 is set to 1 for a distractor period. Finally, the cue signal a_4 is set to 1 and then a_3 is again set to 1 for five time steps.

There are three different outputs, y_1, y_2 and y_3, which also can be either 0 or 1. For all the time steps except the last five, y_1 and y_2 are supposed to be 0 and y_3 1. Then for the last five time steps, y_1 and y_2 are supposed to replicate the signal given on a_1 and a_2 for the first five time steps.

In order to pass the 5-bit memory task, correct output has to be produced for every time step for all 32 combinations of the input signal. The results will be compared with the results found by Yilmaz (2014).

Table 4.1: Parameters for the 5-bit memory task

Cellular automaton rule	All 256 elementary cellular automata
I	2, 4
R	4, 8
Cellular automaton size	40
Distractor period	200

Table 4.2: Successful runs of the 5-bit memory task

Rule	I=2, R=4	I=2, R=8	I=4, R=4	I=4, R=8
60	6%	37.3%	16.4%	46.0%
90	7.7%	17.7%	0.7%	3.2%
102	6.1%	30.4%	14.9%	50.2%
105	0%	1.4%	10.7%	41.7%
150	0.1%	1.3%	12.3%	37.1%
153	3.5%	26.7%	15.4%	43.0%
165	3.4%	13.7%	0%	1.7%
180	0.1%	1.8%	0%	0.0%
195	3.1%	23.1%	13.7%	45.3%

4.1.2 Solving the problem

The framework described in Chapter 3 was used with the parameters shown in Table 4.1. The input was easily mapped to cellular automaton state since the four signals were either 0 or 1.

Linear regression was used as the readout layer, which was trained with all 32 input patterns. Output variables were said to have the value of 0 if they were in the range $(-0.5, 0.5)$ and 1 if they were in the range $(0.5, 1.5)$. If an output variable was not in either of those ranges, the test failed.

Every possible combination of parameters listed in Table 4.1 were repeated 1000 times to observe how they performed at solving the 5-bit memory task.

4.1.3 Results

9 elementary cellular automaton rules managed to get at least one successful run, as can be seen in Table 4.2. 64 cellular automaton rules did not manage to produce correct output for any single input pattern.

Unfortunately, larger values of I and R could not be tested, as the computer running the experiments would run out of memory during regression.

4.1.4 Discussion

Yilmaz (2014) found that the rules 22, 30, 54, 60, 62, 90, 110, 126, 150 and 182 were able to produce correct output for all the 32 different input patterns at least once. Results for four of these rules were included in the paper, a copy of those results can be seen in Table 4.3.

There are only three rules, 60, 90 and 150, that were found to be able to produce no error at least once in the original paper and in this thesis. The original paper was not able to get any correct results with R=8, I=8 or lower parameters, while several configurations manages to produce correct output for lower parameters in this thesis.

Unfortunately, a 100% success rate was not reproduced in this thesis as the computer the experiment ran on, ran out of memory. The original paper managed to run with much larger values for I and R. This is probably at least partly because larger cellular automata

Table 4.3: Successful runs for rule 22, 90, 150 and 182 as reported by Yilmaz (2014, section 3,2). The percentages are changed from showing failed runs to showing successful runs to match the numbers in table 4.2

Rule 90	R=8	16	32	64	Rule 150	R=8	16	32	64
I=8	0%	22%	88%	100%	I=8	0%	20%	92%	100%
16	26%	96%	100%	100%	16	16%	94%	100%	100%
32	96%	98%	100%	No data	32	92%	100%	100%	No data
Rule 182	R=8	16	32	64	Rule 22	R=8	16	32	64
I=8	0%	18%	82%	100%	I=8	0%	22%	80%	100%
16	8%	86%	100%	100%	16	14%	84%	100%	100%
32	88%	100%	100%	No data	32	84%	100%	100%	No data

are used, as there are more cells than the input requires. The implementation in this thesis is not optimized for memory either, so running with larger values of R and I could be possible with better implementations.

Increasing R did increase performance for all the rules that managed to produce correct output. Performance varied more with different I's however, with rules 90 and 165 showing decreased performance for larger I's, even though the other rules generally improved performance with larger I.

Table 4.4: Parameters for the NARMA experiment

Cellular automaton rule	All 256 elementary cellular automata
I	4, 8, 16, 32
R	1, 2, 4
Cellular automaton size	200, 400
Number of cells to which input is mapped	64, 100

4.2 30th order NARMA

In this chapter, the output of a 30th order nonlinear autoregressive-moving-average (NARMA) model will be estimated. The model was introduced by Atiya and Parlos (2000), and has been used as a benchmark for reservoir computing systems.

4.2.1 The problem

$$y(k+1) = \alpha y(k) + \beta y(k) \sum_{i=0}^{29} y(k-i) + \gamma u(k-29)u(k) + \delta \tag{4.1}$$

This problem is to estimate $y(k)$ as defined in equation 4.1 given an input vector **u**. In this experiment, **u** is sampled uniformly from the interval $[0, 0.5]$, and the parameters are defined as $\alpha = 0.2$, $\beta = 0.04$, $\gamma = 1.5$ and $\delta = 0.001$.

Estimating the output is a complex task as the equation describes a highly non-linear system where it is important to remember previous states for a certain time, given by the order of the system. Linear regression by itself is not able to solve this problem, so in this experiment, the computational capability of the reservoir is very important for a good solution.

For every run of this experiment, two data sets with 1000 samples and the corresponding output values are created, one for training and one for testing. The performance of the different reservoirs is scored using normalized root-square-mean error, where a score of 0 means that the reservoir estimates $y(k)$ perfectly. Every combination of the parameters in table 4.6 were tested 10 times.

As a test and comparison of performance, 1000 echo state networks were also created and trained and tested in the same manner as the cellular automaton reservoirs.

4.2.2 Solving the problem

In order to be able to use the framework described in Chapter 3, the input has to be converted to bit strings. The input for this experiment is given as IEEE 754 double-precision floating-point numbers, and the binary representation of those numbers are used as an input in this experiment. Figure 4.1 displays the underlying bit representation of the number 0.5. 64 bits are needed to represent the full range, however since the input numbers are in the range $[0, 0.5]$, the sign bit and the most significant exponent bit can be removed since they will always be 0 for this experiment.

As an attempt to destroy as little of the automaton's memory as possible when writing new input values, they were divided up into two contiguous parts. The first part are the

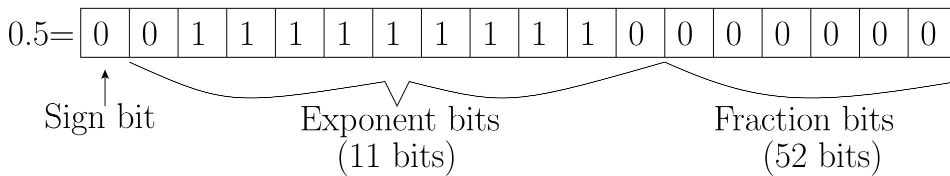

Figure 4.1: The underlying bit representation of 0.5. Note that there are 46 fraction bits that are not displayed in the figure.

cells that the input can be mapped to. The rest of the cells were guaranteed not to be overwritten by the external signal directly.

In order to generate the data sets and echo state networks the Python library Oger toolbox 1.1.3 was used. The echo state networks contained 100 nodes with an input scaling of 0.05. This particular echo state network is quite small compared to typical echo state networks. The specific number of nodes was selected because it was the configuration used in Oger toolbox's tutorial[1] on how to estimate NARMA30 using echo state networks with Oger toolbox. For more details on Oger toolbox, see Verstraeten et al. (2012).

4.2.3 Results

Plots of all the results can be found in Figure 4.2 to 4.5. Additionally, the actual numbers for best, worst and average performance for the different parameters can be found in Table 4.5.

8 elementary cellular automaton rules were able to predict the output with a NRMSE of less than 0.9. They were 85, 15, 240, 170, 119, 48, 34, and 138 in decreasing order of runs with a NRMSE less than 0.9.

Two rules performed exceptionally bad, namely 255 and 0. They were unable to give any meaningful results, as they return just 0 or 1 regardless of input. Therefore, the results of the runs with those two rules were excluded from the results reported in this section.

4.2.4 Discussion

It is very clear that elementary cellular automata perform worse than a simple echo state network. There seem to be some potential, however. The best results with cellular automata were comparable with the worst results with an echo state network. Despite very simple rules and every cell only being able to represent either 0 or 1, it is somewhat possible to predict the output, although with quite a large margin of error.

This problem could possibly benefit from a better input mapping than just using the binary representation of the input. Adding additional states or dimensions could also possibly improve performance by making input of new time steps less destructive to the implicit memory of the automaton.

In this experiment, increasing I leads to poorer performance. A possible explanation is that increasing the number of iterations also erases the memory more quickly as information may be discarded in the transformations. Increasing R seems to generally improve

[1] `http://reservoir-computing.org/node/268` Checked June 17, 2016.

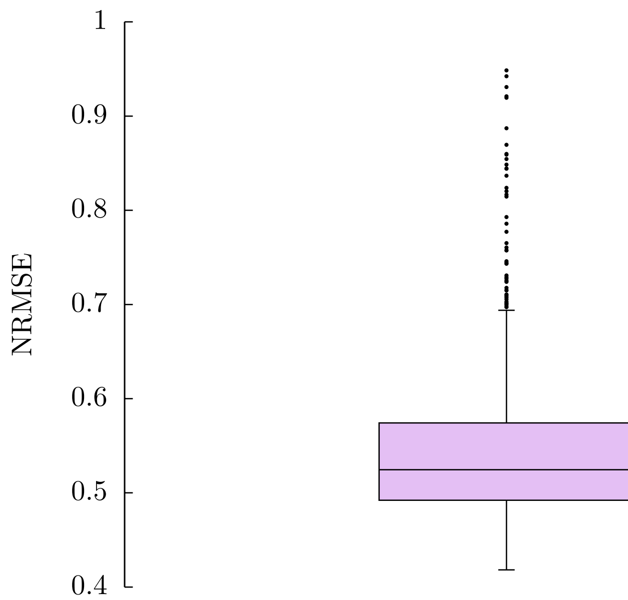

Figure 4.2: Performance for an echo state network with 100 nodes. Note that a lower score is better, as a NRMSE of 0 indicated that the output is predicted completely correctly.

Table 4.5: Best, worst and average values for different parameters

	Best NRMSE	Worst NRMSE	Average NRMSE
Echo state network 100 nodes	0.418	0.949	0.544
I = 4	0.746	9.545	1.672
I = 8	0.764	8.477	1.711
I = 16	0.895	9.055	1.802
I = 32	0.974	11.274	1.920
R = 1	0.766	11.274	1.991
R = 2	0.764	6.414	1.703
R = 4	0.746	7.047	1.634
Automaton size = 200 input cells = 64	0.746	11.274	1.705
Automaton size = 200 input cells = 100	0.937	9.545	1.827
Automaton size = 400 input cells = 64	0.764	7.046	1.726
Automaton size = 400 input cells = 100	0.766	8.764	1.847

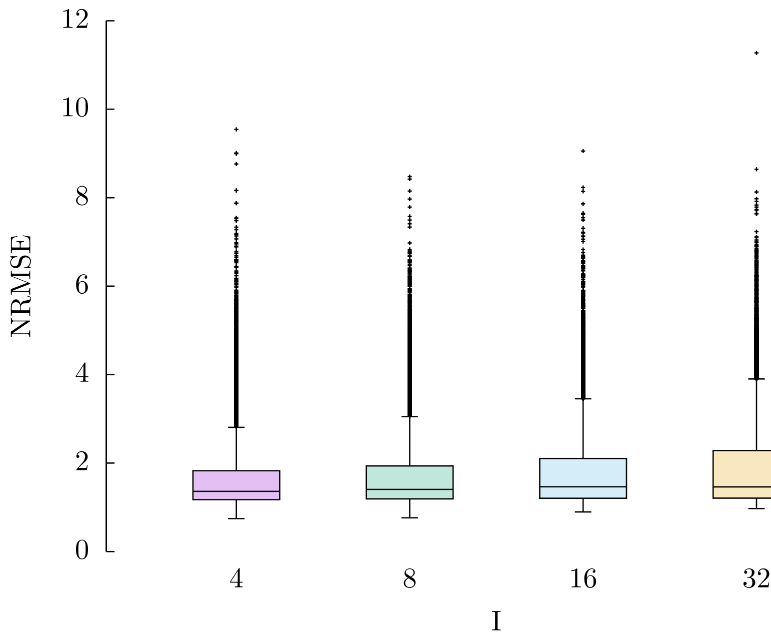

Figure 4.3: Performance for the cellular automata reservoirs for varying I.

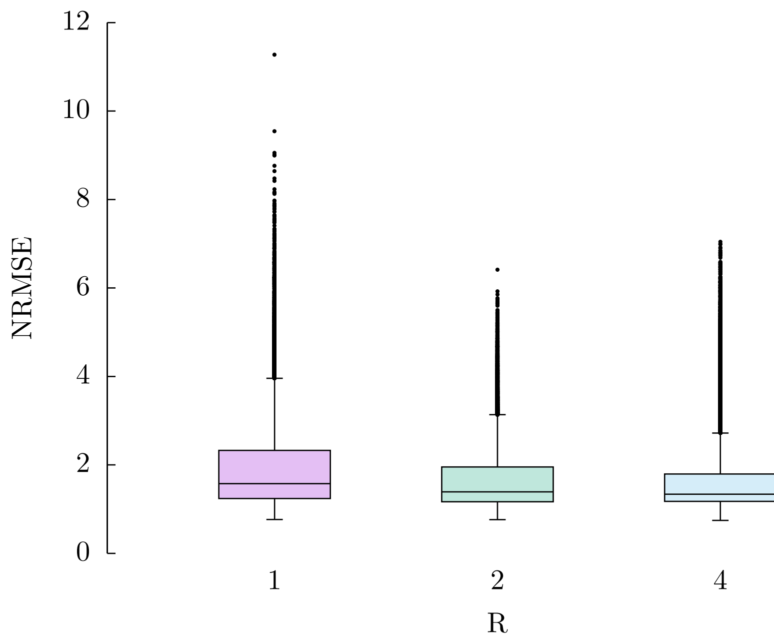

Figure 4.4: Performance for the cellular automata reservoirs for varying R.

performance. Higher R means that the input is projected in more ways, increasing the chance of a projection that is useful for computation.

Finally, it seems that increasing the amount of cells the input can be mapped to lead to poorer performance. A larger area means that there is more space around every input cell on average, which may lead to less interesting interactions happening between input. The total automaton size did not seem to have a dramatic impact on the performance, doubling the size did not produce any drastic changes to the scores.

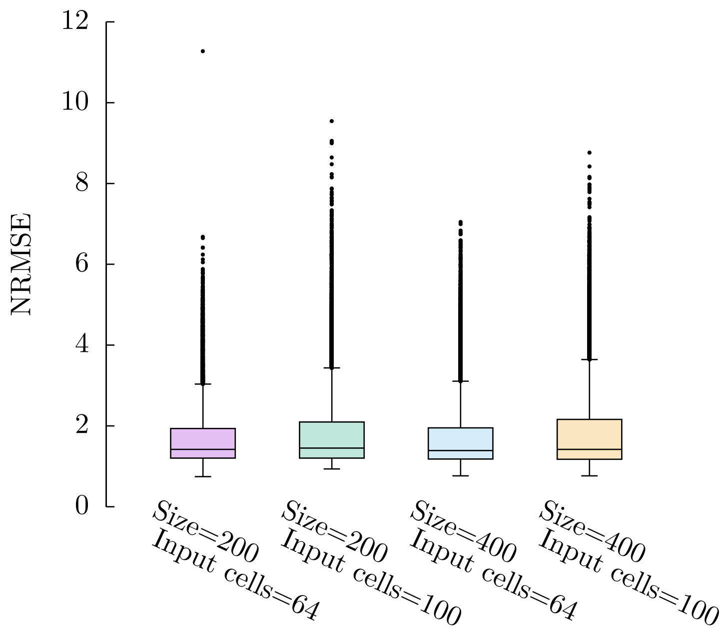

Figure 4.5: Performance for the cellular automata reservoirs for varying automaton sizes and input cells.

Table 4.6: Parameters for the Japanese vowel experiment

Cellular automaton rule	All 256 elementary cellular automata
I	8, 16
R	8, 16
Cellular automaton size	21663, 30000

4.3 Japanese vowels dataset

The Japanese vowels dataset is a dataset presented by Kudo et al. (1999). It was later donated to University of California, Irvine[2]. The dataset consists of sound clips of nine Japanese men uttering the two vowels /ae/ successively.

In this section, an attempt will be made to correctly distinguish the sound clips based on who uttered them. The purpose is to see how well the cellular automata are able to extract relevant information from actual sound clips. As mentioned earlier, Jaeger et al. (2007) managed to correctly classify every single sound clip in the dataset using a reservoir computing approach.

4.3.1 The problem

The task is to correctly identify which of the nine men uttered every sound clip in the data set. The data set consists of 640 sound clips, where 270 are used for training and 370 are used for testing.

Every sound clip has already been analyzed, and a discrete time series with 12 LPC cepstrum coefficients extracted, which is the data being used in this experiment. The time series were all between 7 and 29 steps long.

The results will be compared with the performance of logistic regression on the raw data to see what the effect transforming the dataset with cellular automata has on how well the sound clips are classified.

4.3.2 Solving the problem

In order to decrease the number of bits required to represent the input, the smallest value was found for each of the 12 coefficients, and every coefficient had this value subtracted. This ensured that every coefficient was at or above zero, requiring one less bit for storage.

Additionally, all the time series were padded so that they were 29 time steps long. The padding time steps just consisted of the number 0.0, 12 times.

Unlike the previous sections, input is not given one time step at a time. Instead, a concatenation of all 29 time steps is given to the reservoir at once. The reason for this is that all output prior to the last time step would probably be discarded as the learning algorithms used in this thesis require a fixed input size. So, in order to avoid the first time steps being "forgotten", all time steps are given as input at once.

[2]It is available at `http://kdd.ics.uci.edu/databases/JapaneseVowels/JapaneseVowels.html`, checked June 17, 2016.

Another difference from the earlier sections is that the training data is too large for it all to fit in memory at once, so a online training algorithm had to be used. The readout layer was trained with logistic regression, using stochastic gradient descent in order to train it in an online. This meant that the data sets had to be shuffled before training in order to avoid introducing unwanted biases in the readout layer.

4.3.3 Results

While writing this thesis, it was discovered that an error had been made in the programming for this problem. Only half of the input was actually imprinted on the automata, and all results presented in this section are from runs that had this error. There was not enough time to repeat the entire experiment with the fixed code, but a 10 percentage point increase from the best score by the erroneous code is the best result observed from running the fixed version.

When just using logistic regression on the raw datasets, 98,108108%, or 363 of the 370 sound clips, were correctly paired with the person who uttered it. Performing logistic regression, but with the datasets converted into the bit strings that were used as input to the cellular automata produced 72.162162%, or 267 of 370, correct pairings.

Performance for the cellular automata reservoirs can be seen in Table 4.7 and Figure 4.6 to 4.8.

No rules showed exceptional performance, 104 different rules managed to correctly classify 23.78% of the sound clips.

4.3.4 Discussion

In this experiment it seemed like the cellular automata destroyed the information that was useful for grouping the sound clips. Even with the bug in the code fixed, it only seemed to be able to correctly classify about half the sound clips that were correctly classified without other transformation than converting the numbers to bit strings.

Varying R and the size of the automata does not seem to have much effect, but increasing I seems to destroy valuable information slightly more, leading to fewer correctly classified sound clips.

Apart from the transformation, there are a few other factors that might have contributed to the low rate of correctly classified sound clips. Scikit-learn's documentation states that

Table 4.7: Best, worst and average values for different parameters

	Highest success rate	Lowest success rate	Average success rate
I = 8	23.78%	4.32%	11.54%
I = 16	23.78%	4.32%	11.03%
R = 8	23.78%	4.32%	11.25%
R = 16	23.78%	4.32%	11.33%
Automaton size = 21663	23.78%	4.32%	11.27%
Automaton size = 30000	23.78%	4.32%	11.31%

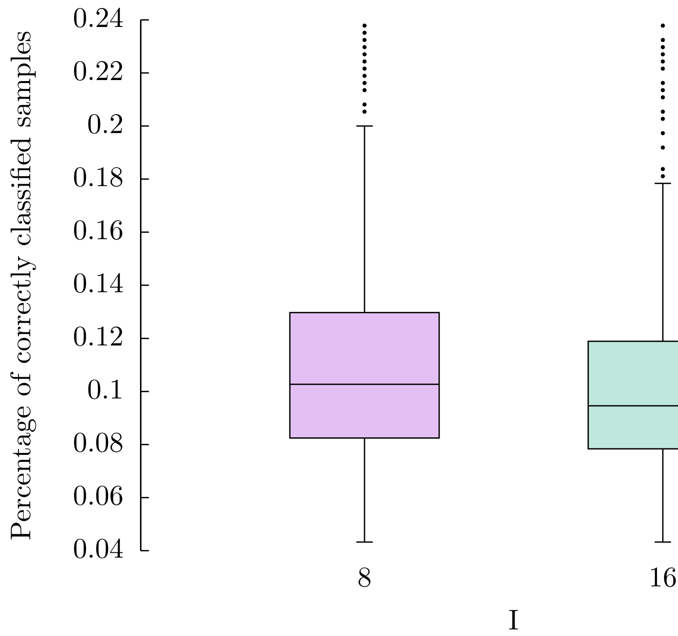

Figure 4.6: Performance for the cellular automata reservoirs for varying I.

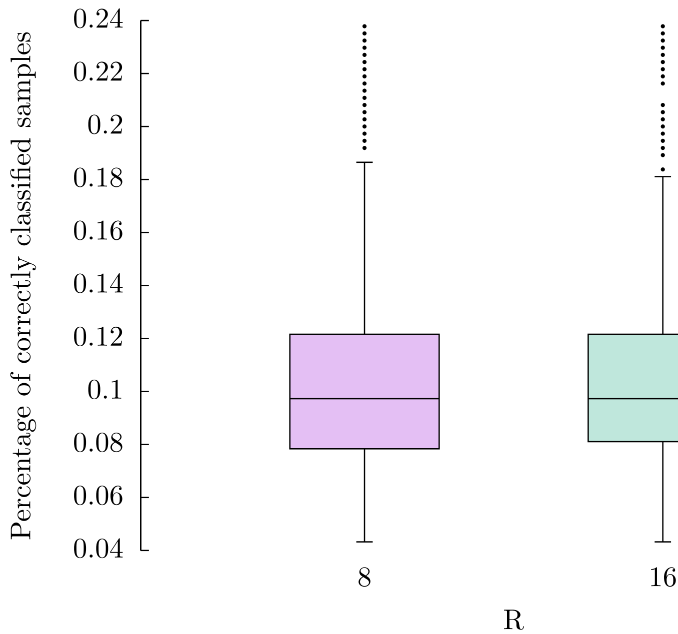

Figure 4.7: Performance for the cellular automata reservoirs for varying R.

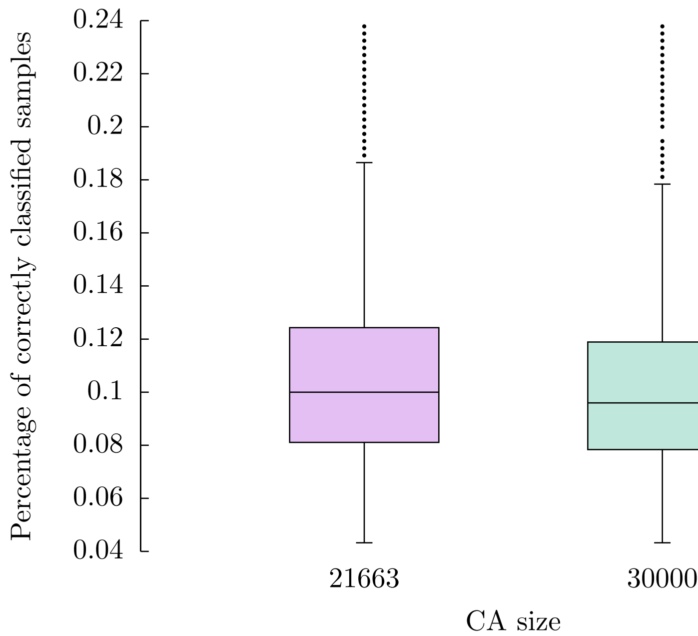

Figure 4.8: Performance for the cellular automata reservoirs for varying automaton sizes.

stochastic descent gradient typically require around 10^6 samples before it converges, and in this experiment only 270 different inputs were available for testing. Modifying the numbers to save space might also have had a negative impact on performance, as the dataset converted to a form that could be used as input to cellular automata had poorer performance than the original dataset.

Lastly, it is possible that greater performance could be achieved by inputting one time step at a time instead of concatenating them and inputting them at once. While information would probably be lost in the process, this more closely emulates the time aspect of the sound clips.

Table 4.8: Parameters for the temporal parity task

Cellular automaton rule	All 256 elementary cellular automata
I	2, 8, 16
R	2, 8, 16
Number of cells the input bit should be mapped to	1, 10
Input area (multiples of input cells)	1.0, 2.0, 4.0
CA size (multiples of input area)	1.0, 2.0, 4.0

Table 4.9: Parameters for the temporal parity and density task

First CA rule	27, 38, 39, 48, 53, 83, 105, 138, 142, 150, 154, 166, 174, 180, 208, 210
Second CA rule	The same rules as the first
I	8, 16
R	8, 16
Number of cells the input bit should be mapped to	10
Input area (multiples of input cells)	2.0, 4.0
CA size (multiples of input area)	2.0, 4.0
Delay for the density output	0, 1, 2, 3, 4, 5, 6

4.4 Temporal bit parity and density

In this section the framework described in Chapter 3 will be expanded by adding a second elementary cellular automaton rule in order to create a non-uniform cellular automaton. The purpose of this is to see whether non-uniform cellular automata can possibly act as better reservoirs than uniform automata. The reservoirs will be tested by having them compute two different functions on the same input stream.

4.4.1 The problem

The two tasks for this experiment are to compute temporal bit parity and temporal bit density for an input stream of bits. Those two function were used by Snyder et al. (2013) to investigate the performance of Random Boolean Networks for reservoir computing.

The input to both tasks is a vector of 0's and 1's. Temporal bit parity should return if there has been an odd number of 1's in the last N time steps. Temporal bit density should return if there has been more 1's than 0's in the last N time steps. Additionally, there can be a delay of τ time steps between an input is given and the output for that time step is expected.

In order to shrink the search space for non-uniform cellular automata, uniform cellular automata are first allowed to solve the temporal bit parity problem. Then the best performing rules are used to look at the performance for non-uniform cellular automata. This is done in order to be able to perform an exhaustive search of the search space.

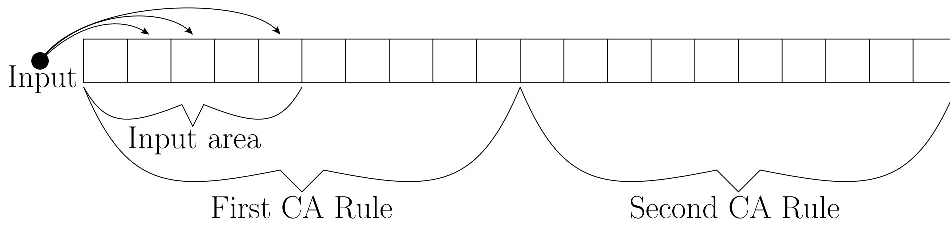

Figure 4.9: Illustration of a non-uniform CA used for temporal parity and density.

4.4.2 Solving the problem

Unlike the other experiments, this attempts to use a non-uniform CA with two different elementary CA rules. Figure 4.9 shows the structure of the non-uniform automata used for this experiment. Another modification that was made from the setup used previously is that instead of mapping the input bit to just one cell, it is mapped to several cells.

Table 4.8 displays the configurations used to solve the temporal bit parity task, and Table 4.9 displays the configurations tested for solving both temporal bit parity and density at the same time. Note that when calculating both functions at the same time, a delay is introduced to the density function. All the different combinations of parameters were tested 10 times.

Training happened using 300 time steps, with a randomly generated bit. Testing was done using 150 time steps.

The readout layer was trained using ridge regression. Similar to earlier experiments, output in the range $(-0.5, 0.5)$ were said to be 0, and output in the range $(0.5, 1.5)$ were said to be 1. For this task, the value of 0 means false while 1 means true:

The result is measured as the percentage of time steps that had a correct output. When calculating two functions, both outputs had to be correct.

Table 4.10: Runs with 100% correct output for the parity task

I	R	Number of runs
2	2	0
2	8	4
2	16	2
8	2	0
8	8	14
8	16	32
16	2	16
16	8	22
16	16	32

Table 4.11: Best, worst and average values for different parameters for the parity task

	Best result	Worst result	Average result
1 input cell	73.33%	0%	49.55%
10 input cells	100%	0%	52.04%
Input area = 1.0	84%	0%	49.29%
Input area = 2.0	100%	0%	52.11%
Input area = 4.0	100%	0%	50.98%
Automaton area = 1.0	100%	0%	51.95%
Automaton area = 2.0	100%	0%	50.54%
Automaton area = 4.0	85.33%	0%	49.88%

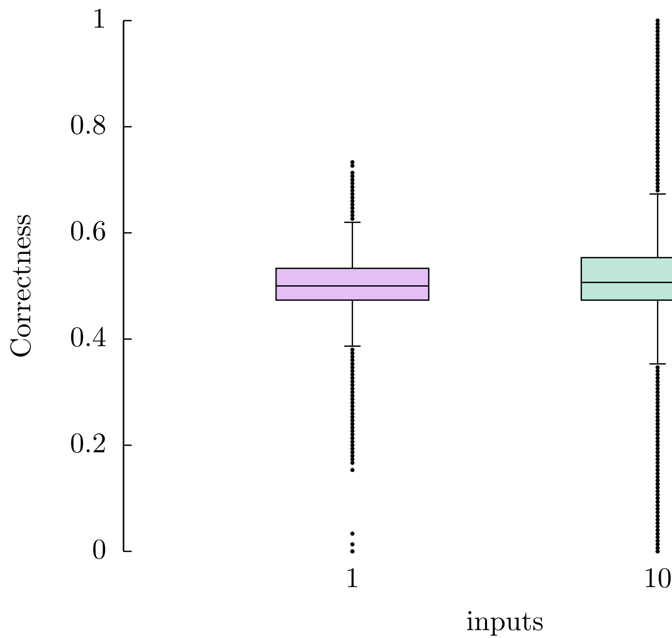

Figure 4.10: Performance for automata grouped by number of inputs for the parity task.

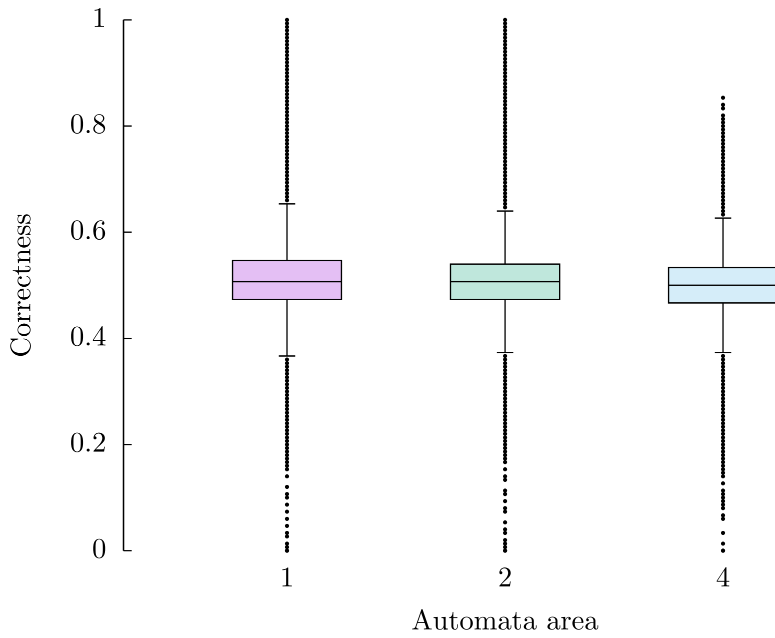

Figure 4.11: Performance for automata grouped by automata area for the parity task.

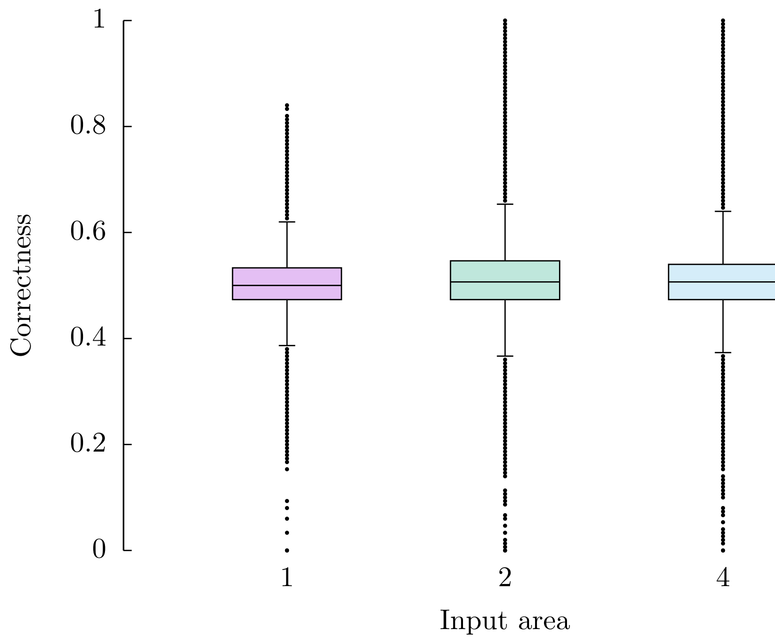

Figure 4.12: Performance for automata grouped by input area for the parity task.

Table 4.12: Best, worst and average values for uniform and non-uniform automata with varying delays.

	Best result	Worst result	Average result
Uniform automata			
Delay = 0	1.000000	0.280000	0.687089
Delay = 1	0.980000	0.300000	0.671021
Delay = 2	0.970000	0.270000	0.655383
Delay = 3	0.950000	0.206667	0.645520
Delay = 4	0.940000	0.313333	0.640092
Delay = 5	0.943333	0.230000	0.621846
Delay = 6	0.893333	0.173333	0.607762
Non-uniform automata			
Delay = 0	1.000000	0.153333	0.701950
Delay = 1	1.000000	0.186667	0.679857
Delay = 2	1.000000	0.210000	0.657042
Delay = 3	1.000000	0.116667	0.635057
Delay = 4	1.000000	0.070000	0.617650
Delay = 5	1.000000	0.146667	0.602109
Delay = 6	1.000000	0.056667	0.589444

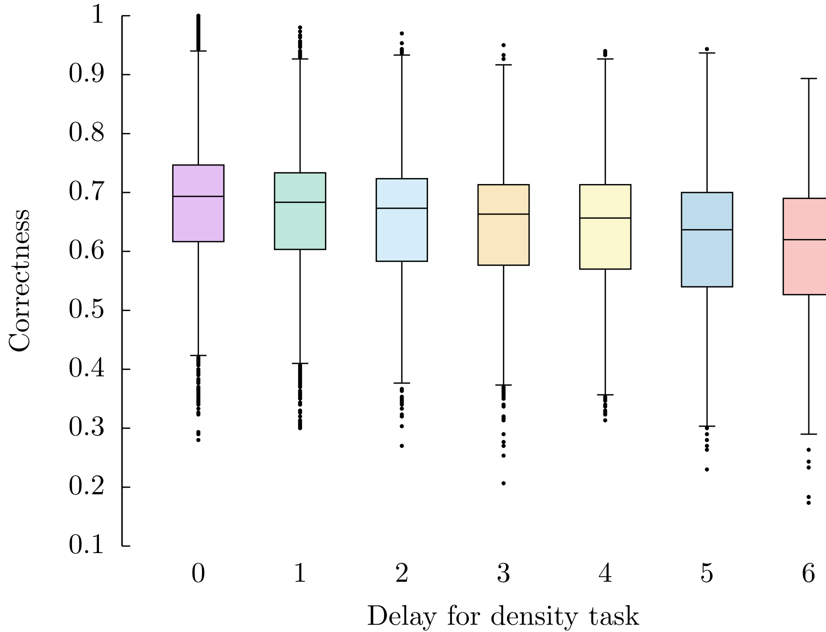

Figure 4.13: Performance for uniform automata for varying delays.

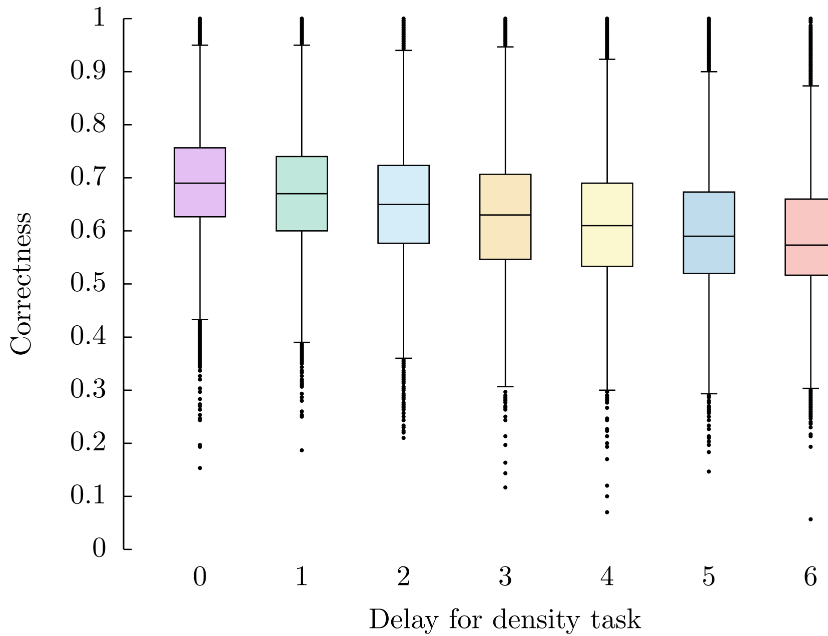

Figure 4.14: Performance for non-uniform automata for varying delays.

4.4.3 Results

For the parity tasks, increasing both I and R had positive effect on the amount of correct classifications. Table 4.10 displays the amount of runs with 100% correct output for different values of I and R. Figure 4.10 to 4.12 display performance for different amount of input cells, and input and automaton areas. Information about the performance of those parameters can also be found in Table 4.11.

The most successful rules for the parity tasks were 154, 210, 39, 27, 83, 180, 166, 53, 105, 138, 142, 174, 48, 150, 208, and 38 in decreasing order of most 100% successful runs. These were the rules that were used in the non-uniform automata.

How the automata performed on both temporal bit parity and bit density at the same time can be seen in Figure 4.13 and 4.14, and in Table 4.12.

4.4.4 Discussion

For the parity task, using just one cell for input did not manage to produce 100% correct results. Optimal performance was achieved when there were twice as many cells as input cells, and leaving a part of the cellular automaton that did not have any input mapped to it did not seem to increase the performance.

When trying to solve two tasks at once, only the non-uniform automata were able to get 100% correct output when there was any delay to the density function. However, with the non-uniform automata, variance was also higher, with the extremes being further from

the mean. The average performance also seemed to suffer a bit in the non-uniform case. Part of the difference may come from a weakness in the methodology, however, as there were 15 times more runs with non-uniform automata than uniform automata.

The non-uniform automata could also possibly be refined in order to increase their performance. An exhaustive search of all the elementary cellular automata rules may find better solutions than with just searching through the rules that performed well at the parity task alone. Making the automata more non-uniform with different structures and sizes for the different rule sections may also enhance the reservoir performance in certain cases.

Chapter 5

Analysis

5.1 Performance of individual cellular automaton rules

Table 5.1 shows lambda values for the elementary cellular automata rules with best performance in the experiments conducted in this thesis. 75% of the rules had a λ of 0.5, even though only 27.3% of the elementary cellular automata rules have a λ of 0.5. A λ value of 0.5 is associated with chaotic rules where information is transmitted rapidly throughout the automaton. Good rules were found outside of $\lambda = 0.5$, but it seems like limiting a search for rules around $\lambda = 0.5$ could limit the search space while keeping many of the best rules. These findings match the theory propsed by Langton (1990), who suggests that cellular automaton rules around $\lambda = 0.5$ are in a critical state between two phases, and that these are the automaton rules with most interesting behavior, capable of supporting the most complex types of computation.

Every additive rule except rule 0, which discards the entire input, and 204, which simply remains static regardless of input, are also among the best performing rules in this thesis. Rule 60, 90, 102 and 150 were all among the rules that managed the 5-bit task. Those four rules are all equivalent with an XOR-operation between two cells in the neighborhood, or all three cells in the case of rule 150. They seem able to retain information about the input pattern even after prolonged period of having a noise signal added.

The two remaining additive rules, 170 and 240, were among the best scoring rules for the 30th order NARMA experiment. They do not perform any calculation, instead, the transformation function copies the state of the cell to the right or the left respectively. Simply estimating NARMA through regression gives very high errors, so something else has to perform some sort of transformation. The only thing that can add any more information is the random mapping to the different automata. In this specific case, different random mappings let some bits live longer than others before they are overwritten by new input.

Table 5.1: The best performing rules and their λ-values

Rule	λ	Which benchmark did the rule perform well in
15	0.5	30th order NARMA
27	0.5	Temporal parity
34	0.25	30th order NARMA
38	0.375	Temporal parity
39	0.5	Temporal parity
48	0.25	30th order NARMA, temporal parity
53	0.5	Temporal parity
60	0.5	5-bit memory task
83	0.5	Temporal parity
85	0.5	30th order NARMA
90	0.5	5-bit memory task
102	0.5	5-bit memory task
105	0.5	5-bit memory task, temporal parity
119	0.75	30th order NARMA
138	0.375	30th order NARMA, temporal parity
142	0.5	Temporal parity
150	0.5	5-bit memory task, temporal parity
153	0.5	5-bit memory task
154	0.5	Temporal parity
165	0.5	5-bit memory task
166	0.5	Temporal parity
170	0.5	30th order NARMA
174	0.625	Temporal parity
180	0.5	5-bit memory task, temporal parity
195	0.5	5-bit memory task
208	0.375	Temporal parity
210	0.5	Temporal parity
240	0.5	30th order NARMA

5.2 Viability

Cellular automata for reservoir computing seem like a promising field of research. The very simplest cellular automata, tested in this thesis, show interesting behavior and are able to achieve surprising performance given the relative simplicity of the dynamics compared to echo state networks.

5.2.1 Resource constraints

Individual computation in the cellular automata are very inexpensive, speed was typically not an issue. Memory usage, however, was mainly the limiting factor of the experiment parameters. This was especially true for all the experiments using offline training methods, as the entire training set with the entire reservoir states for all input, had to be kept in memory at the same time. Online training could potentially prove very useful, as the entire training set is not required to be in memory at the same time. The one attempt at online training in this thesis gave poor results, but that could very possibly be attributed to a bad choice of dataset for use with cellular automata.

5.2.2 Input/output restrictions

With elementary cellular automata the input has to be given as a string of bits. Everything on a conventional computer has an underlying bit representation, but it seems like this may not always provide meaningful solutions, as seen in the Japanese vowels classification. Despite that, it seemed like elementary cellular automata were still able to perform some meaningful transformation on the bits of the real-valued numbers given in 30th order NARMA prediction.

Memory usage was acceptable for the experiments were run, although much memory was wasted, as a single byte was used for every cell state. In order to support larger reservoirs, eight cells could be packed into one byte, although custom code would probably have to be made for performing regression and estimating on the packed cells. Larger reservoirs seem to perform better than smaller ones in many cases. However, performance seems to increase very slowly compared to the reservoir size, so increasing the complexity of the reservoirs may prove more useful as a means of generating better reservoirs.

Chapter 6

Conclusion

6.1 Computational capabilities

Are cellular automata capable of acting as reservoirs for tasks that typically are solved by conventional reservoir computing approaches?

Elementary cellular automata are capable of increasing the dimensionality of input signals, making it possible to estimate non-linear systems with a linear combination of cell states. The cellular automata seem capable of remembering input for a large number of time steps even with when receiving a noise signal. Even though elementary cellular automaton cells are only able to represent a bit each, they are able to perform meaningful operations when the input is given as a floating point number.

When analyzing voice recordings, the automata seemed to destroy information. However, it is quite possible that different encoding schemes and other cellular automaton will be able to improve on the results found in this thesis.

6.2 Performance

How does changing properties of a cellular automaton reservoir affect its performance?

Increasing the size of the reservoirs, either by adding more automata, or transforming them for more iterations, seem to increase the capabilities of cellular automata as reservoirs. Tuning seems to be required for every individual problem however, and increasing the amount of iterations might not lead to increased performance in specific cases.

Larger automata generally imply more memory in the reservoir, but too large automata typically decrease their performance. More memory means potentially more noise, and in many cases it is desirable that old input gradually "fade away" from the reservoir.

Encoding input signals for input to the automata seems to play an important part in the process. Simply mapping input bits to different cells on different automata seems to affect how well a reservoir performs its task.

More complex dynamics, in the form of non-uniform cellular automata also seems

like a viable method to increase what problems cellular automata are capable of solving. A set of different rules operating on the same cell grid may solve more complex problems without a significant difference in the time it takes to calculate the transformations.

6.3 Summary

In conclusion: are cellular automata are able to act as performant reservoirs for tasks traditionally being solved by a reservoir computing approach?

Elementary cellular automata are able to perform some of the same tasks as other reservoir computing systems. Very large reservoirs may be needed to get performance comparable to conventional approaches. However, cellular automata may be implemented very efficiently on certain hardware platforms, making very large reservoirs viable.

6.4 Further work

The use of cellular automata with reservoir computing shows promising results, but much work remains to be done in order to make it a viable alternative to conventional approaches to reservoir computing.

The cellular automaton rule space remains mostly unexplored. Increasing the amount of states could lead to more advanced computations without affecting the amount of instructions it takes to calculate the transformation too much. Additional states could also bring more options when it comes to input. For instance, it is possible to imagine a scheme where new input would not blindly overwrite the underlying state, but interact with it somehow when new input was imprinted on the automaton cells. More states could also be used to bring a more explicit notion of fading memory, where cell state could decay over time.

Increasing the neighborhood size would also let cells interact more easily, possibly requiring less iterations of transformation to perform calculations. Additionally, new dimensions could be added, increasing possibilities for cell states to propagate and interact. More dimensions could also help prevent the loss of data due to overwriting the state of input cells, as there are more cells to store the information that are not overwritten.

Simply adding more cells and dimensions blindly will not necessarily lead to better performance in all cases, though. Fading memory is often a desirable trait in reservoirs, especially for time series estimation, as it usually gets increasingly likely that old information will not be necessary the older it gets. And while noise from old data might not hurt performance directly, too much of it will slow down both the reservoir and the readout layer. It may be interesting to see how much data we really need from the reservoirs, whether some of it can be discarded. By being able to discard some data, it may be possible to support even larger reservoirs to increase performance.

This thesis briefly experimented with non-uniform automata. Instead of having multiple reservoirs for performing different calculations on the same input, is there a "perfect" reservoir consisting of several rules, each exchanging some information with each other, capable of computing everything at once?

Bibliography

All URLs were checked on June 17, 2016.

Appeltant, L., Soriano, M. C., Van der Sande, G., Danckaert, J., Massar, S., Dambre, J., Schrauwen, B., Mirasso, C. R., and Fischer, I. (2011). Information processing using a single dynamical node as complex system. *Nature communications*, 2:468.

Atiya, a. F. and Parlos, a. G. (2000). New results on recurrent network training: unifying the algorithmsand accelerating convergence. *Neural Networks, IEEE Transactions on*, 11(3):697–709.

Burks, A. W. and Von Neumann, J. (1966). *Theory of self-reproducing automata*. University of Illinois Press.

Cattaneo, G., Dennunzio, A., Formenti, E., and Provillard, J. (2009). Non-uniform cellular automata. In *Language and Automata Theory and Applications*, pages 302–313. Springer.

de Carvalho, C. E. and de Oliveira, P. P. (2015). Sorting with one-dimensional cellular automata using odd-even transposition. In *New Contributions in Information Systems and Technologies*, pages 523–532. Springer.

Farstad, S. S. (2015). Evolving Cellular Automata in-Materio. Master's thesis, Norwegian University of Science and Technology.

Fernando, C. and Sojakka, S. (2003). Pattern Recognition in a Bucket. *Advances in Artificial Life*, 2801(12):588–597.

Hettich, S. and Bay, S. (1999). The uci kdd archive. [`http://kdd.ics.uci.edu/`] Irvine, CA: University of California, Department of Information and Computer Science.

Hochreiter, S. and Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780.

Ilies, I., Jaeger, H., and Kosuchinas, O. (2007). Stepping forward through echoes of the past: forecasting with echo state networks. `http://citeseerx.ist.psu.edu/`

viewdoc/download?doi=10.1.1.295.4576{&}rep=rep1{&}type=pdf.

Jaeger, H. (2001). The echo state approach to analysing and training recurrent neural networks. *GMD Report*, (148):1–47.

Jaeger, H., Lukoševičius, M., Popovici, D., and Siewert, U. (2007). Optimization and applications of echo state networks with leaky- integrator neurons. *Neural Networks*, 20(3):335–352.

Jones, B., Stekel, D., Rowe, J., and Fernando, C. (2007). Is there a Liquid State Machine in the Bacterium Escherichia Coli? *2007 IEEE Symposium on Artificial Life*, pages 187–191.

Kudo, M., Toyama, J., and Shimbo, M. (1999). Multidimensional curve classification using passing-through regions. *Pattern Recognition Letters*, 20(11):1103–1111.

Langton, C. G. (1990). Computation at the edge of chaos: Phase transitions and emergent computation. *Physica D: Nonlinear Phenomena*, 42(1-3):12–37.

Lukoševičius, M., Jaeger, H., and Schrauwen, B. (2012). Reservoir Computing Trends. *KI - Künstliche Intelligenz*, 26(4):365–371.

Maass, W., Natschlager, T., and Markram, H. (2002). Real-time computing without stable states: a new framework for neural computation based on perturbations. *Neural Comput*, 14(11):2531–2560.

Neary, T. and Woods, D. (2006). P-completeness of cellular automaton Rule 110. In *Automata, Languages and Programming*, pages 132–143. Springer.

Nikolic, D., Singer, W., Haeusler, S., and Maass, W. (2007). Temporal dynamics of information content carried by neurons in the primary visual cortex. *Advances in Neural Information Processing Systems 19*, pages 1041—-1048.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Rendell, P. (2002). Turing universality of the game of life. In *Collision-based computing*, pages 513–539. Springer London.

Santé, I., García, A. M., Miranda, D., and Crecente, R. (2010). Cellular automata models for the simulation of real-world urban processes: A review and analysis. *Landscape and Urban Planning*, 96(2):108–122.

Snyder, D., Goudarzi, A., and Teuscher, C. (2013). Computational capabilities of random automata networks for reservoir computing. *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, 87(4):1–9.

Steil, J. J. (2004). Backpropagation-Decorrelation: Online recurrent learning with O(N) complexity. *IEEE International Conference on Neural Networks - Conference Proceedings*, 2:843–848.

Triefenbach, F. and Martens, J. P. (2011). Can non-linear readout nodes enhance the performance of reservoir-based speech recognizers? *Proceedings - 1st International Conference on Informatics and Computational Intelligence, ICI 2011*, (1):262–267.

Vandoorne, K., Mechet, P., Van Vaerenbergh, T., Fiers, M., Morthier, G., Verstraeten, D., Schrauwen, B., Dambre, J., and Bienstman, P. (2014). Experimental demonstration of reservoir computing on a silicon photonics chip. *Nature communications*, 5.

Verstraeten, D., Schrauwen, B., Dieleman, S., Brakel, P., Buteneers, P., and Pecevski, D. (2012). Oger: modular learning architectures for large-scale sequential processing. *The Journal of Machine Learning Research*, 13(1):2995–2998.

Wolfram, S. (2002). *A new kind of science. Champaign, IL: Wolfram Media*.

Wolpert, D. H. (2002). The supervised learning no-free-lunch theorems. In *Soft Computing and Industry*, pages 25–42. Springer.

Wyffels, F., Schrauwen, B., and Stroobandt, D. (2008). Stable output feedback in reservoir computing using ridge regression. In *International Conference on Artificial Neural Networks*.

Xiao, X., Wang, P., and Chou, K.-C. (2011). Cellular automata and its applications in protein bioinformatics. *Current Protein and Peptide Science*, 12(6):508–519.

Yildiz, I. B., Jaeger, H., and Kiebel, S. J. (2012). Re-visiting the echo state property. *Neural networks*, 35:1–9.

Yilmaz, O. (2014). Reservoir Computing using Cellular Automata. `http://arxiv.org/abs/1410.0162`.

Yilmaz, O. (2015). Connectionist-Symbolic Machine Intelligence using Cellular Automata based Reservoir-Hyperdimensional Computing. `http://arxiv.org/abs/1503.00851`.