NTNU

Norwegian University of
Science and Technology

Deep Reservoir Computing Using Cellular Automata

Andreas Molund

Master of Science in Computer Science
Submission date: June 2017
Supervisor: Gunnar Tufte, IDI
Co-supervisor: Stefano Nichele, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

Recurrent Neural Networks (RNNs) is a prominent concept within artificial intelligence. RNNs are inspired by Biological Neural Networks (BNNs) and provide an intuitive representation of how BNNs work. Derived from the more generic Artificial Neural Networks, the recurrent ones are meant to be used for temporal tasks such as speech recognition because they are capable of memorizing historic input. However, RNNs are very time consuming to train as a result of their inherent nature. Recent inventions such as Echo State Networks and Liquid State Machines have been proposed as RNN alternatives, under the name of Reservoir Computing (RC). RC systems are far more easy to train.

In this thesis, a Cellular Automata (CA) based Reservoir Computing (ReCA) system is implemented. Methods to map both binary and non-binary input data onto automata are employed, in addition to a recurrent architecture to handle sequential input. Furthermore, several ReCA systems are orchestrated in layers (deepReCA), where the input to layer l is the output of layer $l-1$. The DeepReCA is benchmarked with the long short-term memory tasks 5- and 20-bit, and the Japanese vowels time series classification dataset. Results of benchmarks are compared to state-of-the-art results.

Subsequent layers were found to improve upon previous layers, though the improvement was observed to reach an asymptote. CA rules did have different effect on reservoir dynamics, some proved to be better in layer 1, and some proved to be better in subsequent layers. Some results came close to state-of-the-art performance, which makes the proposed system a viable option if less memory at the cost of accuracy is desired. Further tuning of system parameters as well as designing a more advanced input encoding stage is suggested as future work.

Sammendrag

Rekurrente nevrale nettverk (RNN-er) er et prominent konsept innenfor kunstig intelligens. RNN-er er inspirert av biologiske nevrale nettverk (BNN-er) og gir en intuitiv representasjon av hvordan BNN-er fungerer. RNN-er er en kategori av kunstige nevrale nettverk og ment for temporære oppgaver, som for eksempel talegjenkjenning, fordi de er i stand til å huske tidligere input. Slike nettverk er imidlertid tidkrevende å trene på grunn av deres egenart. Oppfinnelser som Echo State-nettverk (ESN-er) og Liquid State-maskiner (LSM-er) er nylig foreslått som alternativer til RNN-er, under navnet Reservoir Computing (RC).

I denne masteroppgaven er et RC-system basert på cellulære tilstandsmaskiner (CA), kalt ReCA, implementert. Metoder for å oversette både binær og ikke-binær input data til tilstandsmaskinene er presentert og anvendt, i tillegg til en rekurrent arkitektur for å håndtere sekvensiell input. Videre er flere ReCA-systemer stablet lagvis til en hierarkisk arkitektur (deepReCA), hvor input til lag l er output fra lag $l-1$. DeepReCA-en er testet på lang-kort-tidshukommelsesoppgavene 5- og 20-bit, og tidsklassifiseringsdatasettet Japanske vokaler. Resultatene av testingen er sammenlignet med resultater til andre ledende systemer.

Påfølgende lag i arkitekturen ble funnet å forbedre resultatene til tidligere lag, dog var forbedringen observert til å nå en asymptote. Forskjellige implementasjoner av CA-ene hadde forskjellig effekt på reservoar-dynamikken; noen viste seg å være bedre i lag 1, og andre viste seg å være bedre i påfølgende lag. Noen resultater kom nærme resultatene til andre ledende systemer, noe som gjør det foreslåtte systemet til et mulig foretrukket system hvis mindre minnebruk fremfor presisjon er ønsket. Som fremtidig arbeid er det foreslått videre testing av systemparametre samt utvikle et mer avansert input-oversettingssteg.

Preface

This master's thesis is written to conclude the Master of Science in Computer Science at The Department of Computer Science at Norwegian University of Science and Technology.

I would like to express gratitude to Prof. Gunnar Tufte for being my supervisor. I would also like to thank my co-supervisor, Prof. Stefano Nichele, who has contributed with invaluable insights and assistance. Gratitude is also given to fellow student Magnus S. Gundersen for useful detailed conversations about literature. Assistance from PhD candidate Johannes Høydahl Jensen is appreciated as well. Lastly, I would like to thank my family and friends for great support throughout my education.

Trondheim, June 2017
Andreas Molund

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Temporal tasks, which humans experience daily, are a great source of inspiration for research within the field of biologically-inspired artificial intelligence. Humans continuously perceive the real-world environment through e.g. visual and audible events, and consequently act based upon these, while taking past temporal context into account. Imagine a ball traveling with great velocity towards your head. If you are not able to know where the ball was a fraction of a second ago, you can neither reach an understanding of in what direction the ball is traveling nor at what speed. Whether it be human or artificial, systems capable of solving temporal tasks must be able to memorize historical information. A Recurrent Neural Network (RNN) is an example of an artificial system, and have been studied for many years. However, training RNNs are usually compute intensive. An alternative is to treat these networks as static reservoirs of neurons, with rich dynamics that allows only the need to train a readout layer [1, 2]. The rich dynamics are to provide the necessary projection of the input features onto a discriminative, and high dimensional space, so it can be linearly separable at readout.

In general, any substrate equipped with these properties can be used as reservoir. The original reservoirs, i.e. Echo State Networks (ESNs), used traditional non-linear units to realize the dynamical system [1]. On the contrary, examples of unconventional material include using waves produced on the surface of water to solve a speech recognition task [3], and extracting information from the primary visual cortex of an anesthesized cat in [4]. The use of Cellular Automata (CA) as the computing substrate was introduced by Yilmaz [5], and several other researchers have proposed related architectures in its aftermath. CA at a microscopic scale are seemingly simple systems that exhibit simple physics, but at a macroscopic scale can reveal complex behavior which might render desired properties. The desired properties are those that makes CA able to support transmission, storage, and modification of information [6], all of which are necessities for computation.

In Deep Learning, increasing the number of layers in an NN can increase its capacity of representation and abstraction, and hence increase performance [7]. Some researchers have adopted this idea and employed it in RC by substituting layers of neurons with reservoirs of dynamical systems [8]. Reservoirs stacked in such a hierarchical manner are able to operate at different time-scales, and thus have diverse temporal representation of the input [9].

In this work, the goal is to unite the effectiveness of cellular automata based reservoir computing and the paradigm of deep learning. This is a continuation of the semester specialization project which showed promising preliminary results [10], and is currently in press in the journal of Complex Systems [11]. Herein, it is proposed a system with deeper architecture than the preliminary work, where the input to layer l is the output of layer $l-1$, and to some extent uses different rules. It is thereafter tested on state-of-the-art problems in the categories of synthetic- and real-world dataset. The work also presents and introduces novel mapping methods concerning mapping external input to CA configuration in addition to inter-layer communication, some of which are inspired from other work. Ultimately, the goal is to attempt to answer the following research questions:

Q1 **How much can several hierarchical layers improve upon a single layer?**

Q2 **How able is CA as a RC substrate to represent and discriminate different representations of external input?**

Q3 **To what degree is a CA based deep RC system a viable option?**

The rest of the thesis is organized as follows. Chapter 2 covers the background material by introducing the concepts of RC and CA. It furthermore presents relevant CA based RC architectures by going through the different components of such a system. Chapter 3 announces the chosen benchmarks and presents in detail the architecture opted for. The chapter's composition lets the reader revisit the components introduced in Chapter 2 in the same order. Chapter 4 explains the benchmarks in detail, presents the results, and discusses the results individually. Chapter 5 analyzes the results and observations in a more generic fashion, and gives some general advice and recommendations. Finally, Chapter 6 concludes the thesis and includes suggestions for future work.

Chapter 2

Background

2.1 Complex Systems

In the real world, some systems contain elements that interact with each other in a variety of ways. An example on one extreme, is two connected cogwheels in which the elements interact with each other by being strongly coupled. A mechanical force in one rotation on one wheel will transfer to equal amount of force into the other. On the other extreme, random coin tosses is an example in which the elements have no interaction at all. These two are examples of simple systems, where the elements therein interact in a simple manner, if at all.

Many systems in the real world do not exhibit such simplicity. These lie a place in between the systems with strongly coupled, and the systems with decoupled elements. They are called complex systems [12, 13], and their elements interact in a nonlinear manner. Examples include the ocean, the brain, an artificial neural network, a riot, an ant colony, a termite ”cathedral”, or the weather. Within complex systems science, which is a field of research that goes across many disciplines, there are two main concepts that concern the majority of systems: Emergence and self-organization. Emergence is present when it is not possible to describe the property of the system as a whole by solely looking at the physics of the individual elements. It is about scale and the non-trivial relationship between the microscopic and macroscopic properties of a system. The other concept, self-organization, concerns properties or behavior that arises or changes in a system as a whole over time. While the concept of emergence emphasizes on the micro-macro relationship, self-organization is rather about the nonlinear macroscopic behavior of the system over time. A self-organizing system may spontaneously change behavior or organize itself into a different structure. There exist systems that show emergence but not self-organization, show self-organization but not emergence, and show both [14].

An illustration of emergence and self-organization coexisting in a system, a flock of boids, is depicted in Figure 2.1. A boid is a bird-oid, or a bird like, object [15]. Here, the physics for each individual are governed by three rules. First, a boid will steer away from other boids to avoid crashing (separation), depicted in Figure 2.1a. Second, a boid will steer towards the average heading of other boids in the proximity (alignment). Third,

a boid will steer towards the average position of other boids in the proximity (cohesion). The summation of all these three forces determines the boids direction and velocity. Even though it is three simple rules, the collective behavior of the whole lets them organize themselves into coherent structures, as seen in Figure 2.1b. In Figure 2.1c, it is added a predator-like object that has the same behavior as all other boids, however, all other boids are programmed with a ”flee” rule that has precedence. After the predator has left the scene, the boids rearrange themselves back into the structures seen in Figure 2.1b.

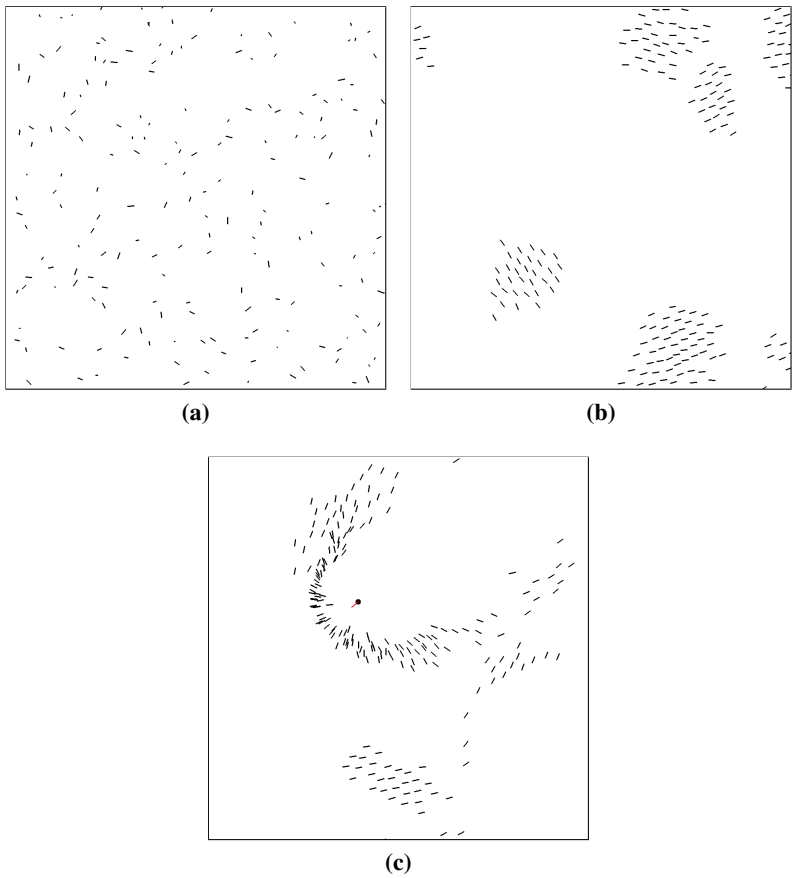

Figure 2.1: Self-organizing bird-like objects. In (a), the boids only follow the separation rule. The boids show self-organizing behavior in (b), when all three rules influence every boid. In (c), a predator enters the scene, and causes chaos, but when it has left, the boids rearrange themselves into similar structures as in (b)

Models of systems in the real world, like the boids are for flocks of birds or schools of fish, are useful analysis tools for scientists. One of the models often used, among others in this thesis, is Cellular Automata (CA). A single cell in CA is simple in its form, yet, through local cell interaction and large number of cells, it can give rise to emergent

behavior. The next section introduces another model, specifically the concept of Reservoir Computing (RC), a computational tool utilizing a ”reservoir” of dynamical interconnected units.

2.2 Reservoir Computing

2.2.1 Fundamentals

Information in feedforward neural networks, are sent one way through layers of neurons; from an input, through a hidden, to an output layer. Neurons in each layer are connected to neurons in the subsequent layer (except for the last one) with weighted edges, and each neuron propagates signals according to its activation function. A Recurrent Neural Network (RNN) contains the same basic elements. However, it has recurrent connections that feed portions of the information back to the internal neurons in the network, making the RNN capable of memorization [16], hence RNNs are a subject for sequential tasks such as speech recognition. RNNs can be trained by different variants of backpropagation [17, 16], all with different computational complexity and time consumption.

One subsequent discovery based upon the fundamentals of RNNs is Echo State Networks (ESNs) by Jaeger [1]. An ESN is a randomly generated RNN, in which the network does not exhibit any layer structure and its internal connection strengths remain fixed, and can be treated as a reservoir of units. The ”echo” is the activation state of the whole network being a function of previous activation states. Training of such a network involves adapting only the weights of a set of output connections.

Another similar discovery is Liquid State Machines (LSMs) by Maas et al. [2]. It is similar to ESN in terms of topology, with an internal randomly generated neural network and problem-specific trained output weights. What differs ESNs and LSMs is more their history. The original LSMs was inspired by biological scenarios, dealt with noise or perturbations, and emphasized on spiking networks, whereas the ESN had a more mathematical approach and performed best without noise. Both inventions were developed simultaneously but independently.

The basic idea of readout nodes that have weighted edges connected to an arbitrary number of neurons inside a reservoir, and only adapting these weights for training, has been named Reservoir Computing (RC). Figure 2.2 depicts this general idea, which is to let the dynamics of the reservoir project the input onto an expressive and discriminative space, so that it becomes linearly separable during readout.

2.2.2 Deep Reservoirs

Deep learning [7, 18, 19] is a technique that makes NNs capable of learning more advanced functions than what conventional machine learning can. In conventional machine learning, careful feature extraction on raw statistical data is needed, and will often require expert knowledge on the data’s domain. Machine learning systems trained on these features can generally predict well on unseen data, but will encounter difficulties when the data’s semantics are represented in several levels, or in higher dimensions. Here is when

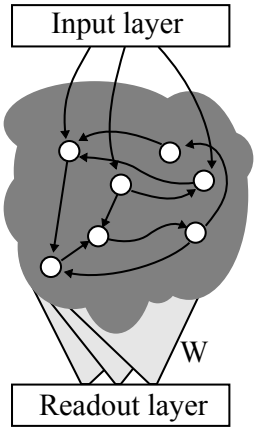

Figure 2.2: A generic reservoir. There is only need to adapt weights W towards a certain target.

deep learning is a more viable approach, because it can automatically learn representations, and in several levels: The model is composed of non-linear layers of modules or processing units, each of which transforms one representation to another more abstract representation.

An example on which deep learning outperforms conventional machine learning is image classification. The initial representation of an image is the very pixels (no features are manually extracted). With deep learning, the first layer typically extracts edges of particular objects in the scene. The second layer can operate on these edges and their arrangements and find motifs. The third layer subsequently can use these motifs further, and so on. Krizhevsky et al. [20] achieved record breaking results on the image classification contest ImageNet LSVRC-2010 with use of a deep convolutional neural network.

Inspired by deep learning and deep neural networks, it has within the reservoir computing research field sprung out ideas on how to improve reservoir computing networks' performance by stacking multiple reservoirs [8, 21, 22]. One reservoir together with its associated readout layer acts then as one layer. Since the hidden units in a reservoir are not trainable, the reservoir's readout layer is trained, and which readout values are sent as input to a subsequent layer.

There are at least two different ways to enter inputs into such systems. One way is to have all subsequent layers operate on only the output of each of their respective previous layer. Another way is to supply the initial input to all layers in addition to outputs of previous layers, which is called Input to All. A third possible way is to supply only the initial input to each layer, which is not really a hierarchical architecture because the "layers" cannot influence each other. The most popular training targets for every layer are the final desired output, i.e. the labels corresponding to the initial input.

One argument for stacking multiple reservoir systems is that the errors of one reservoir system can be corrected by the following one. The following one may learn the semantics of the pattern that it gets as input. However, in order for a subsequent layer to refine a decision, it must take past temporal context into account; a single static decision cannot be refined because it has no temporal semantics.

In [9], it is given an empirical analysis with more emphasis on time-scale and a general view of the potentiality of deep ESN reservoir computing. The authors of it thus has a different, but necessary to science, approach to analyzing deep ESNs than the first proposals of hierarchical reservoirs. In particular, all neurons in layer l are fully connected to all neurons in layer $l+1$, which means no specific trainable readout layer between the ESNs. The reason for that is to further investigate the hierarchical dynamics of RC separately from the learning aspect. On a minor note, although the focus of the paper was critical analysis, the deepESN system was also tested on a short-term memory task. Training then involved concatenating the states of all reservoirs, and adapting a readout model with all these states. It is furthermore presented theoretical ground work in [23]. Overall, it seems that deep reservoir computing is by no means an outdated research area.

2.2.3 Physical Reservoir Implementations

Since the inventions of LSM and ESN, there has been experimented with different physical substances, also called substrates, that replace the RNN as the reservoir. Potentially, any high dimensional dynamic medium or system that has the right dynamic properties can be used. For example, in [4], a linear classifier was used to extract information from the primary visual cortex of an anesthesized cat. In [3], waves produced on the surface of water were used as an LSM to solve a speech recognition task. The genetic regulatory network of the Escherichia Coli bacterium (E. coli) was used as an ESN in [24] and as an LSM in [25]. In [26, 27, 28], unconventional carbon-nanotube materials have been configured as reservoirs through artificial evolution. An optoelectronic reservoir implementation is presented in [29, 30].

2.3 Cellular Automata

Cellular Automata[1] (CA) were inspired by and designed from the study of self-reproducing cells, by von Neumann in the 1940s [31]. They are one of many examples of emergence in complex systems, i.e., the macroscopic properties are hard to explain from solely looking at the microscopic properties. Within a cellular automaton, simple cells communicate locally over discrete time. Locally means that a cell only interacts with its immediate neighbors, thus it has no global insight. The cells are discrete and placed on a regular grid of arbitrary dimension, though the most common ones are 1D and 2D. At each time step, all cells on the grid are updated synchronously based on their physics, where the physics is a transition to a new state based on the previous state of itself and its neighbors. Transition functions are also known as rules.

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

[1] Singular: automaton

Elementary CA (ECA) are one of the simplest kinds of CA. It comprises cells laid out in one dimension, in which $K = 2$ and $N = 3$. The rule space can be enumerated in a base-2 system; each of the $2^8 = 256$ transition functions can be represented by a base-2 number of length 8, as for example rule 110 in Figure 2.4 that is represented as $(01101110)_2$.

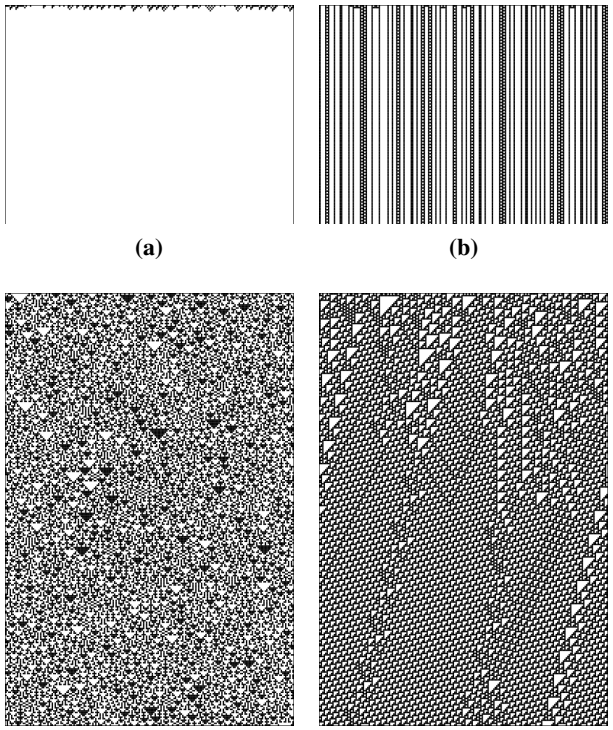

Figure 2.3: Elementary cellular automata iterating downwards. (a) and (b) are cut short. A black cell represents 1. These four are examples of each of Wolfram's classes: (a) is Class I with rule 40, (b) is Class II with rule 108, (c) is Class III with rule 150, and (d) is Class IV with rule 110.

Evolving 1D CA can easily be visualized by plotting the whole automaton, iteration by iteration, downwards, see Figure 2.3 for illustrations. The initial state of an automaton is referred to as a configuration. Wolfram [32, 33] has categorized all one-dimensional CA into four qualitative classes based on what the evolution looks like. CA in class I will always evolve to homogeneous cell states, independent of the initial states, Figure 2.3a. Class II leads to periodic patterns or single everlasting structures, either of which outcome is dependent on initial local regions of cell states, Figure 2.3b. Class III leads to a chaotic and seemingly random pattern, Figure 2.3c. Finally, class IV leads to complex localized structures which are difficult to predict, and may only be found by simulation, see Figure 2.3d.

Langton [6] introduced a scheme for parameterizing rule spaces, called λ. Briefly explained, within a transition function, the value of λ signifies the fraction of transitions

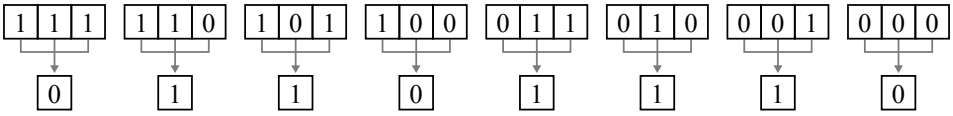

Figure 2.4: ECA rule 110. $(01101110)_2 = (110)_{10}$

that leads to a quiescent state. As an example, rule 110 in Figure 2.4 has $\lambda = 0.625$. If $\lambda = 0.0$, then everything will transition to 0, and the automaton will clearly lead to a homogeneous state. λ is especially useful for large rule spaces where it is hard to exhaustively enumerate all, because it can be used to generate rules with desired behavior. Langton [6] did a qualitative survey on throughout the rule space on 1D CA with $K = 4$ and $N = 5$; rules were generated from different values of λ, from which CA were evolved and analyzed. As the parameter increased from 0.0, the observed behavior underwent various phases, all the way from activity quickly dying out to fully chaotic. In the vicinity of phase transition between ordered and chaotic, a subset of all CA rules, there was observed complex behavior that composed long structures and large correlation lengths. Langton suggested that in this "edge of chaos" is where computation is located.

2.4 Cellular Automata in Reservoir Computing

2.4.1 Overview

The first found published paper of CA based reservoir computing is by Yilmaz [5]. Recently, several other researchers have adopted this approach [34, 35, 10, 36, 37, 38] named ReCA, some of which have had success. ReCA is a novel approach of computing, where CA simply replace ESN as the reservoir substrate, and thereafter is used as an ordinary RC system. There are, however, some clear differences in how CA as a substrate is treated and read compared to ESN. In a classical ESN, input excites neurons, and the state vector is the activation values of all neurons. This state vector is subsequently used in conventional machine learning techniques. In ReCA, the input is translated to a CA configuration, which is then evolved for a certain number of iterations, and then this whole evolution is said to be the state vector. This is to exploit the possible properties that CA can offer.

An example simulation of a ReCA system is presented in Figure 2.5 (rule 90 is used). The initial configuration of the automaton is at the top of the figure, and CA iterations (time) is downwards. Each tick on the vertical axis is external input entering the system by being added to the last automaton iteration before the tick. The whole figure looks like an ordinary CA evolution as in Figure 2.3c, which it is until the second vertical tick. As external input is added with, and not overwrites, CA iterations, there should be indirect traces of the initial input in the very last iterations at the bottom of the figure.

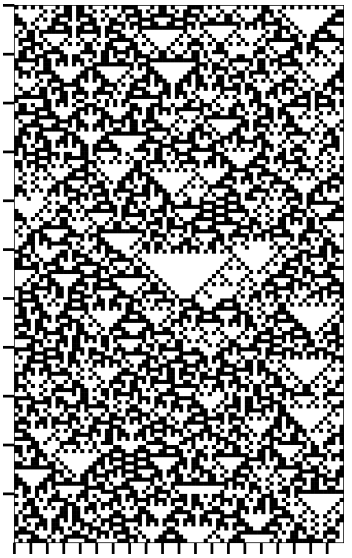

Figure 2.5: An example ReCA simulation. The tics on the vertical axis are where external input is injected (perturbing), and the tics on the horizontal axis are simply multiple versions of the input.

2.4.2 Relevant Work

Encoding

Since the automaton cells take on values from a discrete and finite set, there may be need for mapping schemes to translate input onto them. For tasks of binary nature such as N-bit memory tasks [39] or temporal bit parity and density [40], this is relatively straightforward.

For input with real values, there are several approaches in the literature. First, there is proposed one by Yilmaz [5] in which each cell receives a weighted sum of input features. The weighted sum is then binarized prior to evolution. With this method, one cell can be a receiver of a subset of feature dimensions, and the input can be processed by specific cells.

To other possible binarization schemes for real-valued input are inspired by Binarized Neural Networks (BNNs) [41]. The first one is deterministic binarization:

$$x^b = \begin{cases} 1 & \text{if } x \geq 0.5, \\ 0 & \text{otherwise}, \end{cases} \tag{2.1}$$

where x^b is the binarized variable of real-valued x. The second one is stochastic:

$$x^b = \begin{cases} 1 & \text{with probability } p = \sigma(x), \\ 0 & \text{with probability } 1 - p, \end{cases} \tag{2.2}$$

where σ is the hard sigmoid function:

$$\sigma(x) = \text{clip}(\frac{x+1}{2}, 0, 1) = \max(0, \min(1, \frac{x+1}{2})) \tag{2.3}$$

Both functions are here changed from producing $\{-1, +1\}$ to fit the two states of elementary CA. Indeed, these two schemes have coarse granularity, which can limit what they can express, however, both have proved to be successful with BNNs [41].

Kleyko et al. [37] proposed to quantize real-valued input. Quantization maps a continuous set of input values to a discrete smaller set of elements. This is a rather attractive method because one can adjust the granularity by choosing more of fewer thresholds. In addition, the encoding used to represent each element or level can be composed to whatever works best in the system at hand. Kleyko et al. chose to quantize the input features in their dataset into 4 levels, and each level was encoded to a three-bit representation: `111`, `011`, `001`, and `000`.

McDonald [38] used unary encoding, where one element in the encoded string represents one possible input. It was argued that it can be inefficient for large input values, but used there for simplicity. McDonald also suggested to use gray coding, which is similar to standard binary encoding, but in which only one element changes from one level to a higher level. For example, change in binary representation from integer 3 to 4 are `011` to `100` (three bits change), and in gray coding it renders `011` to `010` (one bit change). This asymmetric change in binary encoding can cause minor variations in input data to have a drastic effect in the response of the ReCA [38]. This is further augmented by Margem et al. [36] stating that when they translated input that originally had unary encoding (Random Permutation and 20-bit memory task [42]) into binary, it lead the task to become harder.

Thus far, the input has been translated into a representation that is feasible to imprint onto elementary cellular automaton cells, i.e. the initial vector, whether it be originally binary input or non-binary as in image pixel intensities or real-valued coefficients. Each feature of the representation can thereafter be mapped randomly onto an automaton, see Figure 2.6a, and evolved through a certain number of iterations I. Increasing the number of random mappings can increase performance [5, 34, 35, 10, 37]. The number of random mappings R can be treated in at least to ways, both of which described by Yilmaz in [34]: Let X be the input vector, X^{P_1} be a single random permutation of it, and function Z be the application of the CA rule. The first way is to treat each X^{P_R} as separate automata, and let them evolve separately:

$$A_0^{P_1} = X^{P_1} \tag{2.4}$$

$$A_1^{P_1} = Z(A_0^{P_1}) \tag{2.5}$$

$$A_2^{P_1} = Z(A_1^{P_1}) \tag{2.6}$$

$$\vdots$$

$$A_I^{P_1} = Z(A_{I-1}^{P_1}) \tag{2.7}$$

$A_1^{P_1}$ through $A_I^{P_1}$ constitutes the evolution of the automaton, and are concatenated to form A^{P_1}:

$$A^{P_1} = [A_1^{P_1}; A_2^{P_1}; \ldots A_I^{P_1}] \tag{2.8}$$

Every configuration of random permutation is evolved this way, and is concatenated to form the state vector:

$$A = [A^{P_1}; A^{P_2}; \ldots A^{P_R}] \tag{2.9}$$

The second way is to concatenate each random permutation into one vector prior to evolution, and not after:

$$X = [X^{P_1}; X^{P_2}; \ldots X^{P_R}] \tag{2.10}$$

$$A_0 = X \tag{2.11}$$

$$A_1 = Z(A_0) \tag{2.12}$$

$$\vdots$$

X is then evolved from A_1 to A_I according to Z, and further used for estimation or classification. It is possible to include the permuted version of the input, i.e. A_0, which for example is the case for the feedforward architecture in [34].

It is also possible to diffuse or use padding in addition to random mappings. Padding is the method of adding elements of no information, in this case zeros, at one or both ends of the mapped vector. These buffers are meant to hold some activity outside of the area where the input is perturbing. Bye [43] invented the idea of diffusing (denoted *input area* therein); it is a sort of padding by inserting zeros at random positions instead of at the end, see Figure 2.6b. It disperses the input to a larger area. ReCA systems that employed diffusing were further investigated in [35, 10].

Regarding the edges of an automaton, they can be considered in one of two cases. If they are ”cyclic”, then the edges are neighbors of each other. In other words, the automaton is orchestrated in a circle, so that the rightmost cell has the leftmost cell as its right neighbor, and vice versa. In the other case, if the edge cells are ”fixed”, then they are not subject for rule application, and they remain in a fixed state, typically 0 [38].

Recurrent Architectures

There is a variety of methods on how recurrency in the CA as a reservoir computing medium can be ensured. Yilmaz [5, 34] sat the baseline for these types of systems, and the more recent architectures in the literature seem to follow these guidelines. As mentioned, after translation, a rule is applied to the automaton for some iterations. Each iteration is recorded so the non-linear evolution becomes a projection of the input onto a state space. For sequential tasks, the system needs to be able to handle several successive inputs, and contain recurrent connectivity to memorize the historical input. ESNs have natively internal recurrent connections and are fed input one step at a time [42]. However, CA do not natively have the same support. One possibility to remedy this, is to flatten the whole input sequence and input it all at once [5], thus it becomes a feedforward architecture. Due

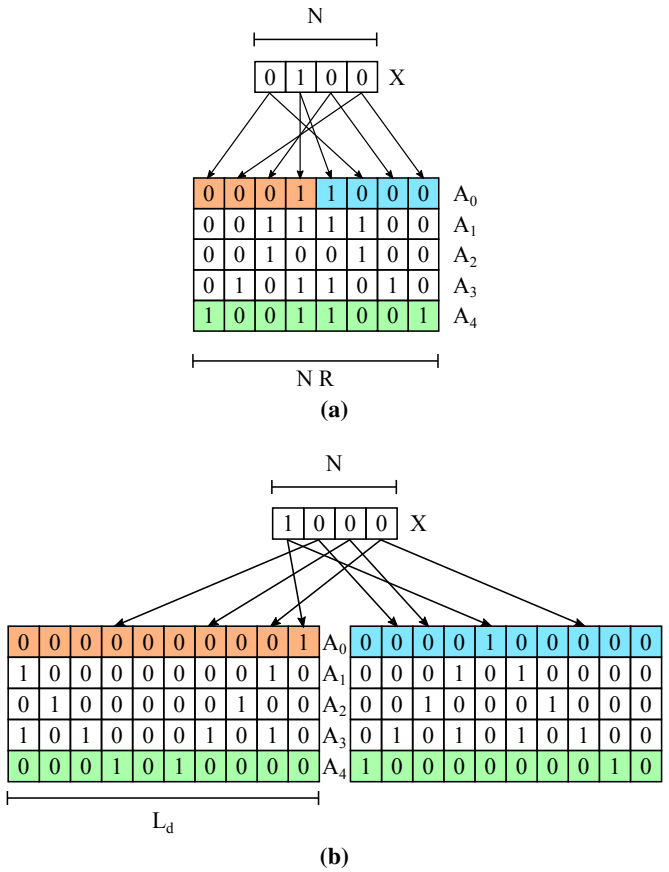

Figure 2.6: Encoding input onto automata by (a) Yilmaz' method [34], and (b) Bye's method [43].

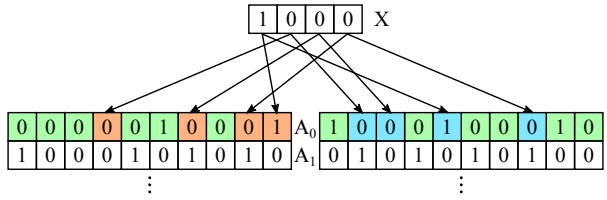

Figure 2.7: Bye's recurrent architecture [43]. The green squares are traces from A_4 in Figure 2.6b

to flattening, it can be memory intensive. This was the first proposed ReCA architecture, and proved to be capable of long short-term memory [5, 34]. Another possibility, is to combine the last iteration of the state vector from one time step with the input at the next, which uses a more intuitive representation of sequential tasks:

$$A_0^t = [A_I^{t-1} + X^{t,P}] \tag{2.13}$$

where $X^{t,P}$ is the permuted version of the input vector at time step t, and the square brackets represent the addition. Several addition methods exist. Since the vectors are binary, one can use any bitwise operation such as XOR [38] or OR. Yilmaz [34] opted for normalized addition, which is a third option, though still bitwise. In normalized addition, if the sum of the two bits is 1, i.e. either $0+1$ or $1+0$, then the output bit is decided randomly $\{0,1\}$, otherwise by OR.

Bye [43] proposed another rather unconventional recurrent architecture, which was later adopted by [35, 10]. For the next time step, the last iteration of the previous state vector is duplicated, after which the next input is permuted and written onto. In other words, instead of mapping the input onto a zero vector as in Figure 2.6b, it is done onto a vector that already contains information as in Figure 2.7:

$$A_0^t = F(A_I^{t-1}, X^t) \tag{2.14}$$

where F is the function that overwrites A_I^{t-1} with the permuted elements of X^t.

Readout

The reservoir readout stage can be any conventional machine learning techniques, such as linear regression or Support Vector Machine (SVM). Linear regression was used in Jaeger's [42] ESN to solve long short-term memory tasks, which implementation was further used without much modification by Yilmaz [5] except for with CA instead of ESN. Inspired by Yilmaz, other ReCA related work that used linear regression include [34, 43, 10]. The SVM approach was employed by Nichele and Gundersen [35], though with a linear kernel.

Kleyko et al. [37] chose to use another classifier for their ReCA system, namely a hyperdimensional classifier. That Cellular Automata/Hyperdimensional Computing based classifier (CAHC) is based on class prototypes and similarities (distance) between them and the computed CA state vectors. The class prototypes are vectors that are formed from each of their respective classes in the transformed training dataset. It is a simple procedure, and does not require any optimization routine [37].

2.4.3 Benefits of Cellular Automata Based Reservoir Computing

Sipper [44] states that the efficiency of cellular computing[2] comes from its simplicity, vast parallelism, and locality. These three principles separate it from the conventional von Neumann architecture, and hence provide new means of computation.

CA as reservoirs provides several benefits over ESNs. One is that the selection of reservoir dynamics or CA is trivial; choosing a rule, or more generic, choosing λ (the

[2] CA are a specific case of cellular computing.

number of quiescent states in the rule) and generate rules accordingly. Even in elementary CA, one of the simplest forms of CA, there exists rules that are Turing complete, i.e. capable of universal computation [45, 33]. Another improvement is the aspect of computational complexity: According to Yilmaz [5], the speedups and energy savings for the 5- and 20-bit memory task are almost two orders of magnitude because of the type (bitwise) and number of operations. Moreover, in the case of the use of additive rules, the implementation can be with bitwise logic, which is suitable for Field Programmable Gate Arrays (FPGAs). Furthermore, the use of linear regression or Support Vector Machine (SVM) for adapting the readout layer can be substituted with summation [5, 38], completely freeing the system from multiplication and division. Such an energy efficient artificial intelligent system may have many domains of application.

Using binary states has also proven to be viable in deep neural networks. In [41, 46], it was proposed Binarized Neural Networks (BNNs) which drastically reduced memory requirements and which implementation ran faster due to binary storage and bitwise operations. BNNs are also feasible to implement on FPGAs [47].

A vast sea of possibilities exists in how to set up ReCA systems. For example, a recent paper by Margem and Yilmaz [36] explores memory enhancements of the CA by adopting more advanced methods prior to evolution. McDonald [38] presented an architecture that combined RC with Extreme Learning Machines (ELMs), in which one CA reservoir state consisted of rules heterogeneous in time; hyperdimensional projection and short-term memory. The preliminary work of this thesis [10] presenting a two-layered hierarchical ReCA for the first time, both with equal parameters, showed promising results. It indicated that two hierarchical reservoirs, where the second operated on the binary outputs of the first, could improve on the performance of a single one. Further research with these possibilities and within this field can provide new understanding and insight in the field of Cellular Automata based Reservoir Computing.

Chapter 3

Methodology

3.1 Benchmarks

To provide means of comparability towards state-of-the-art, it is necessary with benchmarks. Therefore, three tasks within two categories are selected. Not only will the result of benchmarking the system be compared towards earlier work, but also towards variants of the very system presented throughout this chapter, meaning e.g. CA rule or the number of layers.

Pathological synthetic tasks are the first category, specifically some of those proposed by Hochreiter and Schmidhuber [39]. These tasks consist of sequences of inputs, and were designed to test how capable Long Short-Term Memory (LTSM) networks were to learn long term dependencies. Jaeger used these benchmarks to test LSTM performace on ESNs [42], and other researchers have as well (with a narrower selection of tasks) with ReCA in [5, 34, 36, 35, 10]. 5- and 20-bit memory tasks are well known benchmarks within the RC community, and are selected as the benchmarks for the work herein. The resulting performance and results are compared towards Yilmaz' [34] recurrent one-layered architecture of ReCA, and furthermore analyzed with respect to the multi-layered ReCA (deepReCA) proposed in this thesis.

Time series classification is the second category. This is chosen to test the range of ReCA applicability by introducing a real-world, i.e. non-synthetic, scenario to the system. It is an attempt to contribute to others whom also have applied ReCA on non-synthetic tasks, such as Kleyko et al. [37] which succeeded in classifying modalities of medical images, and McDonald [38] which demonstrated excellent results on among others iris classification [48]. Japanese vowels dataset first presented by Kudo et al. [49] is selected for this work. The performance will be compared towards the earlier ReCA system of Bye [43], and other state-of-the-art systems.

3.2 Architecture

3.2.1 Overview

The chosen architecture is presented in detail in this section, which will contain references to Chapter 2. Some parts are inspired from literature, and some is designed during the work. The architecture bifurcates slightly and uses different methods when handling different tasks. Note that the selected synthetic tasks have binary input and fixed sequence lengths, and the selected time series one has real-valued input and variable sequence lengths. The architecture described applies on any task, unless stated otherwise.

Figure 3.1 depicts the main components of a layer in the system, and can be seen as one layer in Figure 3.2. Encoders receive either external input or input from a preceding layer, and translate it into a binary representation. The reservoirs are simply evolving cellular automata along with the recurrent architecture. Readout layers are linear models, intrinsically matrices, that are either adapted during training, or used in matrix multiplication during testing.

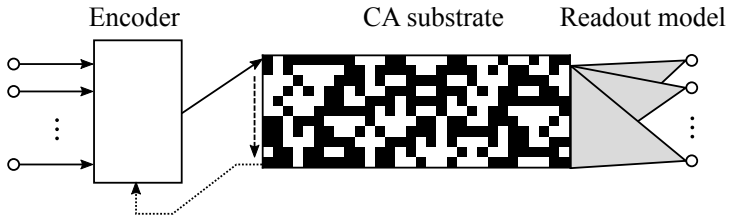

Figure 3.1: One ReCA layer. Dashed arrow implies CA iterations (time). Dotted arrow is information re-entering the system (recurrent).

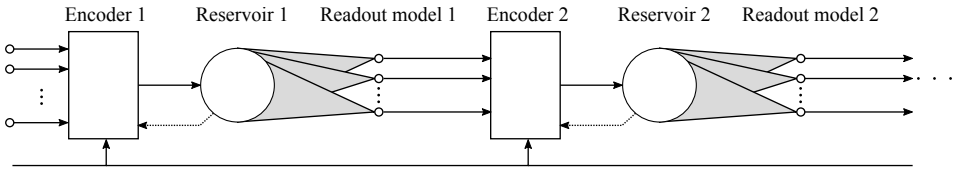

Figure 3.2: A holistic view over the system architecture. Dotted arrow is information re-entering the system (recurrent). Input to All (IA) is also seen.

3.2.2 Encoding

Input is first altered in the encoding stage of ReCA. As mentioned in Chapter 2, there exists many viable methods to opt for in the literature. It really depends on whether the input features already are binary, or need translation. Most benchmarks with data extracted from the real world will have some sort of coefficients, including the Japanese vowels dataset used here. These coefficients are the digitalized versions of the analogue data in the real world. Nevertheless, the simplest encoding scheme used in this work takes

binary input (from the synthetic tasks) and maps it onto CA. For the real-world task, input is first binarized, and thereafter used as binary input.

The base scheme takes binary input, and is relatively simple. If X is the binary input vector, then it is randomly mapped onto a CA configuration as depicted in Figure 2.6a, and with cyclic automaton edges. Thus, X is neither padded nor diffused in any way, and the raw input features are preserved. One reason to choose this method is to allow comparison with literature [34]. This is a different approach than the "diffuse" (see Figure 2.6b) used with modification in the preliminary work of this thesis [10]. One reason for not choosing the diffuse method is because it was experimentally observed to cause too much instability in the system and to yield much variance in the individual runs. Large variance may be due to the large area to which the input is mapped: separate elements of mapped input can both be very far apart and very close. In addition, through visualizing the activity in all but the first reservoir, they had little cell activity (fewer cells were activated), thus diffusing may not be necessary here.

Real-valued input encoding is slightly more advanced, and can be seen as a preprocessing step to the base scheme. Quantization is chosen to be the main method for real-valued encoding scheme, inspired by Kleyko et al. [37]. The thresholds are generated using quantize levels from all values in the whole training data (using test data here would render the experiment invalid). A question to bring up is whether it is needed different thresholds for different independent input features, e.g. four thresholds for each of the 12 coefficients in Japanese vowels. If the input is compound and represents two or more completely separate concepts, or which feature values come in completely different ranges, then it could be necessary. However, if related to image features, i.e. pixel and color intensities, it would be cumbersome to keep track of every threshold. For example, a single CIFAR-10 image has 32^2 pixels, each with red, green, and blue channel values [50]. In this work, the Japanese vowels dataset is the only dataset that needs to be binarized. Each feature x is quantized into five levels using four thresholds or percentiles, where each level is represented with 3 bits in gray code according to Equation 3.1. These entities of 3 bits are subsequently randomly mapped onto an automaton configuration, see Figure 3.3. Quantization is in itself lossy compression, and the main idea is to preserve input as features at the expense of losing details.

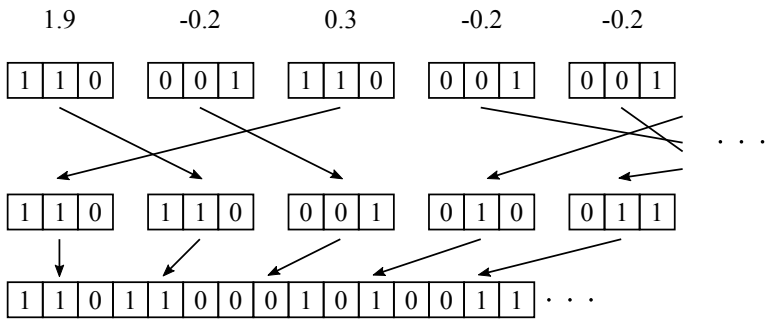

Figure 3.3: An example on how quantizing and random mapping is done. The representations are treated in blocks of three.

$$x^b = \begin{cases} \texttt{000} & \text{if } x < -0.2984444, \\ \texttt{001} & \text{if } -0.2984444 \leq x < -0.1379516, \\ \texttt{011} & \text{if } -0.1379516 \leq x < 0.004259, \\ \texttt{010} & \text{if } 0.004259 \leq x < 0.2079332, \\ \texttt{110} & \text{otherwise.} \end{cases} \tag{3.1}$$

In [43], Bye investigated a different ReCA encoding scheme for the Japanese vowels. As every coefficient in this dataset is a real-value, it was therein proposed to use their direct floating point encoding, meaning using the binary exponent and mantissa field, with minor adjustments. With a double-precision floating-point number, it meant almost 64 bits of information. In addition to the large representation which will require close to 64 cells, due to the nature of how these representations are constructed, little variation in input can mean that a vast portion of the bits are flipped, and hence cause drastic change in CA projection. Analogies can be drawn to binary representation explained in Section 2.4.2. A compression method like quantization can suffice.

So far, the encoding of input has been described. However, because there are multiple layers, there is also needed some translation of the output of one layer that are to enter the next reservoir. Two different methods are used in this work, depending on the category of the task at hand, even though the original output values of any readout layer are floating points. For the two tasks of binary nature, every predicted value is binarized according to Equation 2.1, and further used as this representation.

The output for Japanese vowels is quantized and encoded according to Equation 3.2:

$$o^b = \begin{cases} \texttt{000} & \text{if } o < q_1^l, \\ \texttt{001} & \text{if } q_1^l \leq o < q_2^l, \\ \texttt{011} & \text{if } q_2^l \leq o < q_3^l, \\ \texttt{010} & \text{otherwise,} \end{cases} \tag{3.2}$$

where the thresholds (q_1^l, q_2^l, q_3^l) are quantize levels $(25, 50, 75)$ of all output of the training phase at layer l, and o and o^b are respectively the activation value of one output node and its binarized version. The thresholds are observed to change from one reservoir with one specific dynamics, i.e. rule and (I,R), to another. Thus, they are generated after training (based on the all readout activation values), and stored until testing phase. Online generation is necessary because it is infeasible to manually inspect and set the levels during run-time. The regression model can produce several competing outputs, even though any label in any task in this thesis have only one non-zero element per time step. Quantizing the inter-layer values is an attempt to preserve the information in competing outputs in a way so that the succeeding layer can receive more information per prediction, as opposed to deterministic binarization.

Another detail regarding the Japanese vowels architecture, is that external input is also supplied to every succeeding layer by being concatenated to the output of layer $l-1$. This can be referred to as deepReCA Input to All when using the terms from [9] but replacing ESN with CA. The external input always follows the scheme in Equation 3.1, and the output from layer $l-1$ as aforementioned.

3.2.3 Recurrent Architecture

Elementary cellular automaton replaces the ESN as the reservoir substrate, i.e. cells have three neighbors (including itself), each of which can be in one of two states. This space means that there are 256 rules that can be applied, not all of which are used in this work. After the input has been encoded and prepared for CA evolution, it is randomly mapped R times onto CA states, concatenated to a single configuration (Equation 2.10), and evolved for I iterations according to a specific rule. Normalized addition is used as addition of previous time step and input vector. Essentially, permutation, concatenation, and addition are adopted from the recurrent architecture of Yilmaz in [34], and is the main algorithm used.

In the case of the speech task, only some state vectors are selected to be the ultimate output of the first reservoir. Alternative 3 in [51] with a modification is employed at the first layer, and Alternative 4 is at the final layer:

3. Choose a small integer D, partition each state vector sequence of length l_i into D parts, and select the last state vector A_{n_j} ($n_j = j \times l_i/D, j = 1, ..., D$) in each partition. If D is no divisor of l_i, then select the closest vector (Jaeger et al. [51] chose to interpolate between states here), see an example in Figure 3.4. This gives D vectors per sequence, which are meant to reflect a few equally spaced snapshots of the evolved sequence.

4. Concatenate all state vectors A_{n_j} into a single joined vector.

When Alternative 3 is applied on the state vectors at the first layer, the set of samples (also referred to as time steps) for the rest of the layers will all have sequence lengths $l_i = D$. $D = 4$ in these experiments. Alternative 3 is also applied on external input in the case of Input to All. Alternative 4 in [51] was intended as a method for a one-layered ESN architecture, meaning it would generate one state vector (from D snapshots). However, it was implemented here as a method on the final layer, which already takes in D sized sequences, as a way to produce one ultimate hypothesis/prediction per sequence (this is classification after all). The idea is to let j "section experts" generate hypotheses, and to let subsequent layers refine and aggregate them. An advantage is the shorter sequence length, hence less computation overall for subsequent layers. A possible disadvantage is to lose temporal context, as D is recommended to be a small integer ($D \leq l_{\text{min}}$).

3.2.4 Readout Layer

As one can infer from what described earlier, the number of readout values from the reservoir depends on the input length, and the number of random mappings and iterations. The readout values from one time step, i.e. state vector, is sent into a linear regression model together with its corresponding label. Specifically the `linear_model.LinearRegression` class from scikit-learn which is implemented with plain Ordinary Least Squares [52]. For the ease of training, the model is fitted (offline) all at once with every state vector in every sequence, together with their labels. Every model if fitted towards the same target labels. Even though the elements are from different time steps from different locations in the training set, they are weighted and treated equally because they each retain (to a

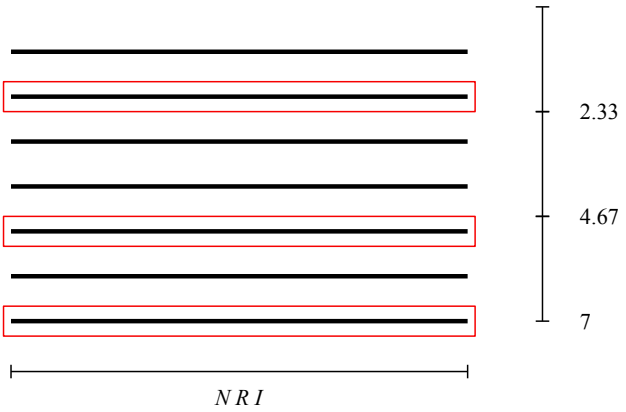

Figure 3.4: An example of how Alternative 3 is applied on a sequence where D is no divisor of l_i. Each bold horizontal line represents a state vector. A bold line surrounded by a red rectangle is the selected vector. Here, $l_i = 7$, $D = 3$

greater or lesser degree) history from their respective ”time lines”. Each corresponding label represent semantics from which the model is to interpret the readout values.

After the model is fit, it can be used to predict. Because of regression, the output values from the predictions are floating points, and are further processed according to one of the two encoding schemes described earlier.

3.3 Training Algorithm

Training the system as a whole involves the procedure that follows. Inputs is binarized, encoded, and mapped onto automata from which the first reservoir computes state vectors. The first regression model is fitted with these vectors towards their corresponding labels. Normally, new state vectors are computed for prediction afterwards, i.e. prediction on training set still because the next reservoir should train on the output of the preceding. This diverges when it comes to the 5-bit task, where the training set equals the testing set (only 32 unique sequences available), so fitting and prediction are performed on the same vectors (as were done in [42]). These predictions or hypotheses are then binarized, encoded, and mapped onto automata that are to be used by the second reservoir – here with the same labels. Using the same training targets for every layer seems to be the trend in multi-layered reservoir computing systems. Every following layer is trained using this procedure.

Testing the trained system involves merely for every layer to consume the output of the previous (external input in the case of the first layer), compute state vectors, and predict.

3.4 Platform

The software itself [53] is implemented in Python [54] programming language version 2.7. Vector operations are provided by The NumPy Array [55], and graphical illustrations are provided by Matplotlib [56]. Python is chosen because of the support and the available packages within its machine learning community. All experiments were run on a computer[1] limited to 4 cores 2.20 GHz compute power. RAM resources varied according to the task and parameters. "Tot. fit time" mentioned in Chapter 4, refers to the time spent on linear regression only (e.g. no CA evolution), and is the sum of time used for regression in each layer.

[1] Supercomputer. 27 nodes of type Dell PE630 with 2 x E5-2630 v4 10 cores 2.20 GHz, 128GB RAM compute power.

Chapter 4

Experiments

4.1 Preliminary Work

Prior to this thesis, there was carried out some preliminary work and preparation [10]. The work is currently in press in the journal of Complex Systems [11]. That was the project that began to investigate the possibilities of deep reservoir computing systems based on cellular automata. It presented some experimental results which showed potential and which encouraged the work of this thesis.

A parameterizable reservoir computing system with cellular automata as reservoir substrate was implemented[1]. It was then tested with several different parameters on the 5-bit memory task. The system was thereafter expanded with another layer, where the output of the first reservoir inputted to the second one. Output of the first reservoir was used to compare the result towards state-of-the-art work of that time, as well as towards the layered version. The motivation to opt for the two-layered system was that the second reservoir would act as a system that corrects some of the mispredictions of the first one.

The results for the layered system showed noticeable improvements when compared to the single-layered. A selection of the findings is presented in Table 4.1. Note that the architecture used differs from the method presented in this thesis; diffusing (Figure 2.6b) was used, and overwriting (Figure 2.7) instead of normalized addition. The greatest improvement (53.4 %) was achieved by rule 165 at $(I,R) = (4,4)$. Again, R is the number of random permutations of the input, and I is the number of CA iterations. Rule 165 proved to be promising in general with its highest average improvement.

[1]The whole code repository can be found on GitHub at `https://github.com/andreasmolund/reca`.

Table 4.1: Improvement of adding a second reservoir (%). Note that $L_d = 40$

Rule	$(I,R) = (2,4)$	(2,8)	(4,4)	(4,8)	(8,8)
90	-10.3	7.6	30.1	11.8	0.0
150	0.0	0.0	0.0	17.8	1.1
60	0.0	32.2	19.5	4.5	0.0
102	0.0	32.9	25.3	3.2	0.0
105	0.0	0.0	0.0	17.4	1.1
153	0.0	21.8	22.5	4.7	0.0
165	0.0	19.9	53.4	13.5	1.1
195	0.0	27.6	26.8	6.7	1.0

4.2 5-bit Memory Task

4.2.1 Synopsis

The 5-bit memory task is a pathological synthetic task designed by Hochreiter and Schmidhuber [39]. It is a memory task to test for long-short-term-memory in systems. Sequences of binary vectors are presented to the system, and the pattern of bits in the first part of the sequences are to be memorized by the system and replicated after a distractor period.

4.2.2 Task Details

A sequence of binary vectors of size four are presented to the system, where each vector represents one time step. The four elements therein are said to be channels and act as signals, thus, only one of them can be activated at a time. This constraint also applies on the output which also is a binary vector, but rather with three elements. In [39], the problem was formulated with four output bits, but the fourth is "unused" (always 0), hence it is omitted in this implementation. In the input, the two first channels carry the memory pattern, the third is the waiting-for-cue or distractor signal, and the fourth is the cue.

For the first 5 time steps in one run, the first two channels in the vector is toggled between on and off. If one of them is on, the other one is off, and vice versa, hence, there are a total of 32 possible combinations for the 5-bit task. From time step 6 throughout the rest of the sequences, the third channel is on, except at time $T_d + 5$ where the fourth channel is on. As an example, a distractor period of $T_d = 200$, means a total sequence length of $T = T_d + 2 \times 5 = 210$. As for the output, for all time steps until $T_d + 5$ inclusive, the third channel is on. Thereafter, the first and second channel are to replicate the 5 first input signals. See Figure 4.1 for an example.

Jaeger [42] defined a successful run as one where at all time steps and in all four (three) output channels, the absolute difference between the network output and its target is less than 0.5. In this thesis, the success criteria are almost the equivalent: Each output vector from a layer is binarized deterministically according to Equation 2.1 and thus have to equal the target output. That means a total of $3 \times T$ bits correctly predicted. In particular, a single value > 1.5 would be binarized to 1, whereas Jaeger's system would render that a failure. Furthermore, the number of successful runs in each trial is recorded, which

serves as the performance measurement. For the sake of clarity, a *run* is one sequence sent though the system, and a *trial* here is 32 runs. In theory, this differs from how e.g. Jaeger [42] and Yilmaz [34] measured performance, because they counted the number of successful trials, one of which is successful if less than 1 percent of the runs fail. In practice, 32 successful runs can be said to be equivalent to 1 successful trial. With respect to analyzing the improvement over several layers, recording only the number of successful trials would yield too weak of a precision. The work that presents the most precision for this task, found in literature, gives results down to the number of false bits [36].

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

Figure 4.1: An example of the 5-bit memory task with a distractor period $T_d = 3$. The cue signal occurs at timestep 8, after which the first and second bit of the output are replicating the equivalent bits in the input (marked in gray).

4.2.3 Results

100 trials were run per combination of rule and reservoir size, and averaged to yield one result figure. Throughout the thesis, the combination of number of CA iterations and random mappings (i.e. random permutations of the input on the initial automaton configuration) are denoted (I, R). All result figures are measured in the average number of successful runs, meaning a max of 32. Precision is given down to two decimal points because of 100 trials. Since the performance from trial to trial did not fluctuate too much, the standard deviations is presented in the appendix instead of here.

Only a sub group of all ECA rules were used, and the selection was based on what has been proven to work in other ReCA systems in literature. Rules from different classes has been tested, e.g. 110 and 54 from Class IV, 90 and 182 from Class III. Class I rules have not been investigated since they cancel out any initial activity. Rule 2, 16, and 36 are Class II rules, which are not necessarily attractive use, but have been tested here for the sake of experimentation, partially inspired by McDonald's ELM architecture [38]. Some

equivalent rules are also tested, as will be discussed later.

To compare and verify this architecture towards [34], it is tested with similar rules and (I,R), see Table 4.2. Table 4.3 presents first findings from the initial deepReCA runs. The whole group of rules that are subject for testing has been tested on the same reservoir parameters to find individual good performance rules. This is a broad testing to eliminate bad rules. Furthermore, the rules that shows most total improvement have been selected for further testing with quite smaller subsequent reservoirs, which results are presented in Figure 4.2. Finally, Figure 4.3 and 4.4 shows results from the run with deepest architecture also using these rules, as an attempt to achieve full score 5-bit task with deeper layers.

Table 4.2: Average number of successful runs for the 5-bit task. One layer. $(I,R) = (32,40)$. Sorted by rule. Standard deviations are located in Table A.1.

Rule	Tot. fit time	Layer 1
54	44.3	31.76
62	44.3	31.78
90	52.8	31.82
102	37.4	31.98
110	37.2	32.00
146	49.8	32.00
150	53.3	32.00
165	54.1	32.00
195	51.7	32.00

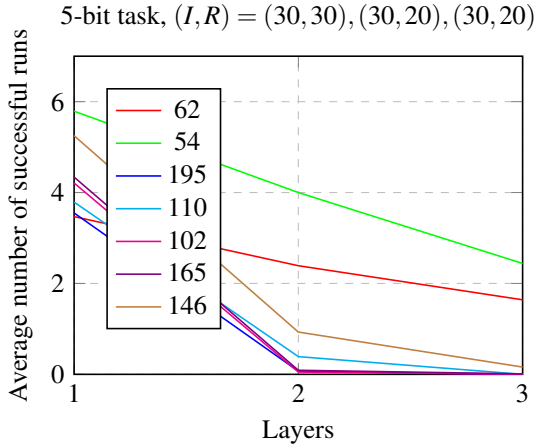

Figure 4.2: Average number of successful runs for the 5-bit task. $(I,R) = (30,30),(30,20),(30,20)$. Standard deviations and accurate numbers are located in Table A.3.

Table 4.3: Average number of successful runs for the 5-bit task. $(I,R) = (30,30),(30,35),(30,35),(30,35)$. Sorted by total improvement, i.e. the difference between Layer 4 and Layer 1, descending. Standard deviations are located in Table A.2.

Rule	Tot. fit time	Layer 1	Layer 2	Layer 3	Layer 4
36	236.5	0.05	6.77	28.27	30.91
62	46.0	3.58	17.71	25.76	29.50
54	46.8	5.50	19.32	26.69	29.64
102	41.9	3.70	16.24	22.89	26.72
146	46.3	5.18	17.59	24.70	28.20
195	45.3	3.86	16.27	22.76	26.65
110	46.6	3.70	15.48	22.18	26.30
165	44.7	3.53	15.57	21.87	26.08
60	47.5	4.36	16.33	22.89	26.84
153	44.5	4.28	15.66	22.15	26.36
90	47.6	4.05	15.93	22.35	25.81
45	46.8	4.08	15.02	21.66	25.76
182	47.5	4.51	13.02	19.14	22.22
106	40.6	4.88	12.11	17.66	22.18
41	40.6	3.38	10.34	16.16	20.59
30	44.3	3.83	8.84	12.03	14.51
22	41.0	4.02	6.64	8.18	8.87
126	45.7	4.07	6.01	7.16	7.32
2	248.2	0.00	0.00	0.08	0.16
16	244.8	0.00	0.00	0.03	0.13
150	42.5	4.33	1.31	0.87	0.59

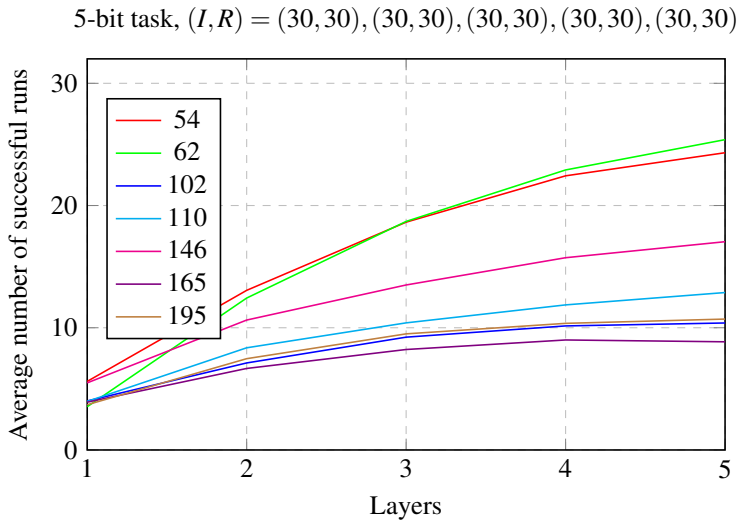

Figure 4.3: Average number of successful runs for the 5-bit task. $(I,R) = (30,30),(30,30),(30,30),(30,30),(30,30)$. Standard deviations and accurate numbers are located in Table A.4.

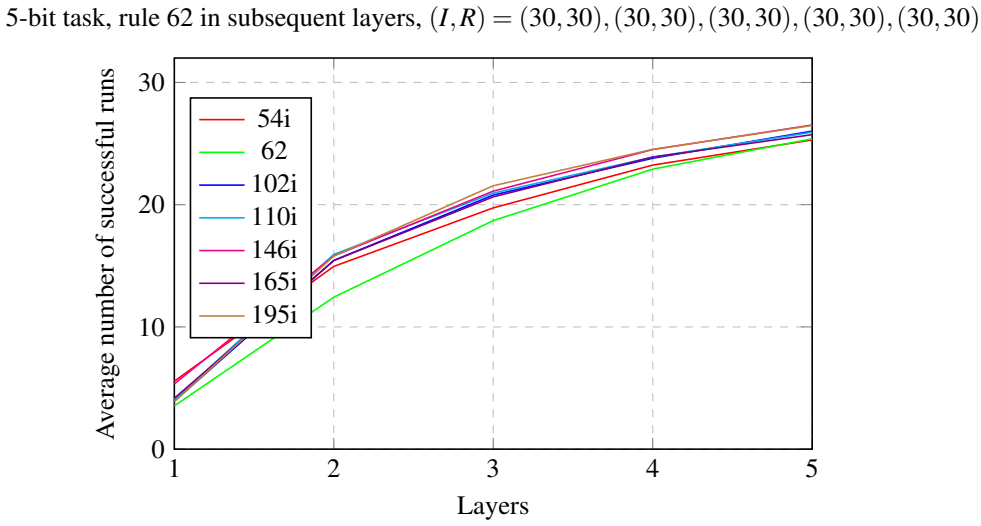

Figure 4.4: Average number of successful runs for the 5-bit task. Every subsequent layer uses rule 62. $(I,R) = (30,30),(30,30),(30,30),(30,30),(30,30)$. Standard deviations and accurate numbers are located in Table A.4.

4.2.4 Discussion

Opting for a quite similar recurrent architecture as Yilmaz [34] is to have some point of reference. First, the results from Table 4.2 can be used for comparison and verification towards Yilmaz' work in terms of similar performance. Second, the same results can serve as a reference point to compare against the performance of the deeper architecture. Findings from [34] are reported in Figure 4.5 for the ease of discussion. Figure 4.5 presents the minimum number R that the different rules need in order to get zero error on the 5-bit task with varying distractor period, with I is fixed to 32. Although Yilmaz found that different rules needed different R, all rules were set to the same R in this thesis. Comparing the results, it is seen that rule 150 gives equal performance in both systems. Rule 90 in Table 4.2 did not manage to get full score, however, it had 4 fewer random mappings to make use of. On the contrary, rule 110 in Table 4.2 did manage to get full score, even though R was set to a smaller value. This might be caused from fine detail differences in the success criteria. Yilmaz' architecture uses the same criteria as Jaeger [42], which means that individual readput values < -0.5 and > 1.5 is said to be failure, whereas it would not be in this thesis. In other words, the success criteria in this thesis are more relaxed.

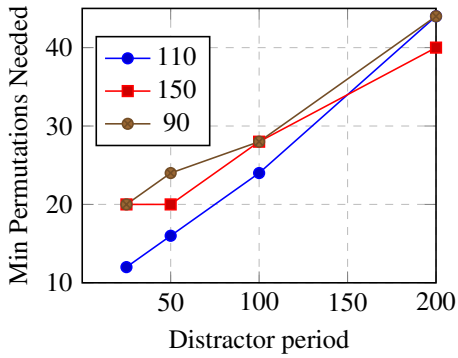

Figure 4.5: Performance of Yilmaz' ReCA [34] on the 5-bit memory task. The points indicate the minimum number of random mappings needed in order to obtain zero error. $I = 32$.

Some rules presented in Table 4.3 are essentially equivalent. Rule 102 is black-white equivalent with 153, i.e. they interchange the black and white cells, and left-right equivalent with rule 60, i.e. they interchange left and right cells. Rule 102 is furthermore both black-white and left-right equivalent with rule 195. With these rules being some form of equivalent, it is not easy to spot major similarities of performance for them in Layer 1 in Table 4.3 (taking other rules also into account). However, noticing the improvement over more layers, the theoretical similar performance become more noticeable, being only maximum 0.48 points away from each other at Layer 4. Not only does this point out that they can perform equal in one setting, but it also indicates that they have similar performance with respect to different representations. Emphasized again, the first layer holds representation of external input, while the second (and successive) holds representations of the first layer's hypotheses.

Some of the best rules in [34], does not necessarily perform as well in a layered setup. An example of that is rule 150, which was presented to be the best in [34] (Figure 4.5) for the same amount of distractor period, and also reproduced in a one-layered setup in Table 4.2. Rule 150 applied in deepReCA does not differ a lot in performance in the first layer compared to other rules (Table 4.3). Its drastic performance decrease occurs at subsequent layers. Additional checks were performed to verify if the divergence was genuine. Rule 90 and 110 did not diverge as noticeable from the main findings, but did not either become the best.

Table 4.3 is sorted by total improvement, i.e. the difference between Layer 4 and Layer 1, descending. The rules that have most improvement are selected for further experimentation, omitting equivalent rules and rule 36 because of its non-dynamical behavior. The next experiment illustrates what happens when the reservoir is too small, or more specifically, decreasing R from 35 to 20 in all subsequent layers. Figure 4.2 presents the findings, and it becomes immediately evident that the decreased reservoir size (computational complexity) affects performance. Even though the first layer's size has been proven to produce sufficient information for subsequent reservoirs to improve upon (Table 4.3), these smaller layers cannot capture the dynamic information.

Finally, Figure 4.3 presents the result of trials with the deepest ReCA, in which every layer has $(30, 30)$. All rules have lost a certain degree of capabilities, some more and some less, which is to be expected due to lower R. Rules that have lost minor capabilities include 54, 62, and 146, whereas rules that have lost major capabilities include 102 and 165. In the case of rule 165, layered architecture benefits the final performance only marginally. Inspecting Figure 4.3, the result at the final layer varies more than the equivalent in Table 4.3. Maybe what is found is a bifurcation point between where rules are capable of capturing the necessary information, and where they are not. Another explanation may be that, as long as the individual layers in deepReCA have sufficient computational capacity to capture the necessary information, almost any rule will perform good, and almost any rule will reach this asymptotic performance as seen in Table 4.3, hence approximately equal results in the final layer.

It was observed that rule 62 had not the best single-layer performance, but the best improvement overall. Therefore, an idea was to use the best single-layer performance rule at Layer 1, and use rule 62 in every subsequent layer. In Figure 4.4, rules have the letter i as postfix to indicate results of this experiment. Here it becomes apparent that every rule now is improved, up until, and even surpassing, the performance of rule 62. Inspecting the average number of correct runs does not give as clear results as opposed to inspecting average number of mispredicted time steps. A time step is said to be mispredicted if any of its binarized values does not equal the target output. Rule 62 yielded a mean of 7.6 mispredicted time steps in layer 5. Changing the rule in the first layer to 146, gave a mean of 6.2. These observations suggest that different rules have different ability to represent a certain concept. In other words, rule 62 was better at representing the hypotheses from a previous layer, whereas rule 146 was better at representing the external input.

4.3 20-bit Memory Task

4.3.1 Synopsis

This task is another task designed by Hochreiter and Schmidhuber [39]. It is of similar structure and type as the 5-bit memory task, although more difficult since it has a more complex pattern to memorize. The memory pattern spans over more time steps and more dimensions. Again, sequences of binary vectors are presented to the system, and the pattern of bits in the first part of the sequences are to be memorized by the system and replicated after a distractor period.

4.3.2 Task Details

The type of vectors that are presented to the system are same as in the 5-bit version. In the input vectors, the last and second to last channels are still the waiting-for-cue and cue, respectively. However, the number of memory channels is increased from 2 to 5, and the duration of the pattern is increased from 5 to 10, see Figure 4.6 for an example. Again, input and output vector channels act as signals, thus only one channel can be active at a certain time step. Possible pattern combinations are therefore in the number of 5^{10}, which is slightly more than 20-bits of information, hence the task's name. Due to the vast quantity of samples, it is infeasible to use all of them for training and testing. Usually, samples for benchmarking is obtained by generating a chosen number of unique sequences, and thereafter selecting one portion for training and the rest for testing. Outputs are in the same format as for the 5-bit task, meaning that for all time steps until $T_d + 10$ inclusive, channel 6 is activated. After $T_d + 10$, the first 5 channels are replicating the memory pattern.

Success criteria here are the same as for the 5-bit task. Each regression output is binarized deterministically (Equation 2.1) and must then equal the target output.

4.3.3 Results

100 trials were run per combination of rule and reservoir size, and averaged to yield one result figure. 500 sequences or runs were used for training, and 100 for testing, all with distractor period of $T_d = 50$. All result figures were originally measured in the average number of successful runs, meaning a max of 100. However, because no combination of rule and (I,R) was able to produce a single correct run, the number of mispredicted time steps is given instead. Recall that the target output for a time step is 7 bits, and if any bit in the binarized version of the readout values for a time step does not equal the 7 target bits, then it is said to be mispredicted.

All rules that were used in the main experiment of the 5-bit task, were tested here as well, though only for a certain (I,R), see Table 4.4. A narrower selection of rules has also been tested with the lowest possible (I,R), Table 4.5. Table 4.6 presents the maximum computational complexity attempted. Above this "maximum", linear regression becomes very time-consuming, towards impractical to run on a desktop computer.

Timestep	Input							Output					
1	0	0	1	0	0	0	0	0	0	0	0	0	1
2	0	1	0	0	0	0	0	0	0	0	0	0	1
3	1	0	0	0	0	0	0	0	0	0	0	0	1
4	0	0	0	0	1	0	0	0	0	0	0	0	1
5	0	0	0	1	0	0	0	0	0	0	0	0	1
6	0	1	0	0	0	0	0	0	0	0	0	0	1
7	0	0	0	0	1	0	0	0	0	0	0	0	1
8	0	0	0	0	1	0	0	0	0	0	0	0	1
9	0	0	1	0	0	0	0	0	0	0	0	0	1
10	1	0	0	0	0	0	0	0	0	0	0	0	1
11	0	0	0	0	0	1	0	0	0	0	0	0	1
12	0	0	0	0	0	1	0	0	0	0	0	0	1
13	0	0	0	0	0	0	1	0	0	0	0	0	1
14	0	0	0	0	0	1	0	0	0	1	0	0	0
15	0	0	0	0	0	1	0	0	1	0	0	0	0
16	0	0	0	0	0	1	0	1	0	0	0	0	0
17	0	0	0	0	0	1	0	0	0	0	0	1	0
18	0	0	0	0	0	1	0	0	0	0	1	0	0
19	0	0	0	0	0	1	0	0	1	0	0	0	0
20	0	0	0	0	0	1	0	0	0	0	0	1	0
21	0	0	0	0	0	1	0	0	0	0	0	1	0
22	0	0	0	0	0	1	0	0	0	1	0	0	0
23	0	0	0	0	0	1	0	1	0	0	0	0	0

Figure 4.6: An example of the 20-bit memory task with a distractor period $T_d = 3$. The cue signal occurs at timestep 13, after which the first 5 bits of the output are replicating the equivalent bits in the input (marked in gray).

Table 4.4: Misclassified time steps for the 20-bit memory task. Here, $(I,R) = (16,20),(16,20),(16,20)$. Standard deviations are located in Table A.5.

Rule	Tot. fit time	Layer 1	Layer 2	Layer 3
2	41.4	1000.6	2058.7	1018.2
16	41.9	1000.6	2049.7	1016.4
22	45.8	1000.6	1001.3	1001.1
30	38.8	1000.8	1000.6	1000.6
36	47.6	1037.2	2090.0	1160.0
41	49.3	1001.3	1002.4	1002.2
45	50.2	1000.7	1000.6	1000.6
54	46.0	1005.9	1063.0	1033.3
60	39.8	1080.5	2538.2	1252.4
62	38.3	1001.1	1002.7	1002.8
90	39.1	1036.6	1821.6	1317.8
102	38.6	1086.9	2572.2	1283.8
106	44.0	1033.0	1187.6	1137.3
110	39.9	1000.6	1000.9	1000.9
126	38.7	1000.6	1000.7	1000.9
146	45.3	1002.4	1169.3	1116.0
150	41.8	1026.5	1242.8	1158.5
182	47.5	1000.7	1000.8	1000.7

Table 4.5: Misclassified time steps for the 20-bit memory task where $(I,R) = (1,0),(1,0),(1,0)$. Standard deviations are located in Table A.6.

Rule	Tot. fit time	Layer 1	Layer 2	Layer 3
2	0.2	1000	1000	1000
16	0.2	1000	1000	1000
22	0.2	1000	1000	1000
54	0.2	1000	1000	1000
90	0.2	1000	1000	1000
110	0.2	1000	1000	1000

Table 4.6: Misclassified time steps for the 20-bit memory task where $(I,R) = (16,100),(16,100),(16,100)$. Standard deviations are located in Table A.7.

Rule	Tot. fit time	Layer 1	Layer 2	Layer 3
54	1318.8	1430.3	1810.9	1497.8
62	1351.6	1422.2	1482.2	1468.5

4.3.4 Discussion

It is not found any use of this recurrent ReCA architecture on the 20-bit memory task in literature. In [34], Yilmaz proposed a feedforward 2D ReCA architecture on which both 5- and 20-bit task were applied. Only the 5-bit version was furthermore tested with elementary CA rules, and only the 5-bit version was tested with recurrent architecture (as mentioned in Section 4.2). It was observed no major increase in (I,R) demand when going from 2D to elementary, but some increase when going from feedforward to recurrent (although decreased memory usage because of the non-flattening). It is reasonable to believe this observation would also relate to the 20-bit memory task. Table 4.7 presents the estimated reservoir size demands for the 20-bit task based on the findings for the 5-bit. These estimates are by no means said to be true, but they give an idea of how much the reservoir size has to be increased in order to produce something useful.

Table 4.7: Minimum (I,R) needed in order for different CA type and architecture to get zero error, found by Yilmaz [34]. $T_d = 200$ for feedforward, and $T_d = 50$ for recurrent. Italic numbers are estimates.

Task	Game of life, feedforward	ECA, feedforward	ECA, recurrent
5-bit	(16,32)	(16,32)	(32,16)
20-bit	(16,384)	*(16,384)*	*(32,192)*

Margem and Yilmaz [36] used a very different architecture which proved well on the 20-bit task. They opted for a more advanced non-linear preprocessing stage prior to reservoir computing stage, and using linear rules in the reservoir for feature expansion (multilayer CA architecture). However, that is a vastly different architecture than used in this thesis, and can be subject for investigation in future work. In addition, their design was meant for only pathological synthetic tasks, whereas the aim for this thesis is among others to apply ReCA in real-world tasks.

1000 mispredicted time steps is the spot at or above which all result figures lie, according to Table 4.4, 4.5, and 4.6. 1000 is the average number of mispredicted time steps per trial. In the sequences presented to the system, the target output of the last 10 time steps differs from the other time steps, and because there are 100 runs per trial, this is what causes $10 \times 100 = 1000$ to be the line. In fact, if the system is parameterized with the minimum of $(1,0)$ as in Table 4.5, it becomes clear that the regression model has not learned anything, and only produces the waiting-for-cue signal. $R = 0$ means that the elements in an input vector are used in the exact order that they appear, in practice, equivalent performance as $R = 1$.

4.4 Japanese Vowels

4.4.1 Synopsis

The Japanese Vowels dataset[2] was originally collected to test a classifier for multidimensional curves (using passing-through regions), first appearing in [49], and has been used as a benchmark since. It contains time series of different length comprised of multiple feature dimensions, each of which is uttered by one of nine males. The utterance is the pronunciation of two Japanese vowels /ae/ successively.

4.4.2 Task Details

The dataset consists of 640 time series. Each of these time series are recordings of one of nine male speakers uttering the Japanese vowels /ae/ successively. One utterance by a speaker forms one time series whose length varies from 7 to 29 time steps. 12-degree linear prediction analysis has already been applied on them to obtain discrete-time series with 12 LPC cepstrum coefficients, and hence 12 feature dimensions. 270 out of the 640 are used for training, within which each male or class has 30 samples. The rest, i.e. 370 time series, are used for testing, and contains 24 to 88 samples for the same speakers. Variable sequence length is due to the goal of the creators. The goal was to collect a dataset closer to the natural expression of a sample and devise a classification method for the case. One sample may be measured over a short period, another over a longer. A sample is presented in Figure 4.7

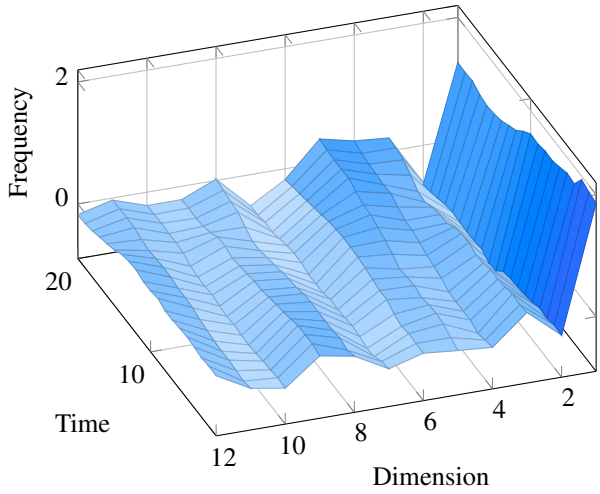

Figure 4.7: One sample, and its frequencies of 12 LPC coefficients through time. This is in fact the first sample in the training set.

[2]Owners and donors: Mineichi Kudo, Jun Toyama, and Masaru Shimbo. Available at `https://kdd.ics.uci.edu/databases/JapaneseVowels/JapaneseVowels.html`

4.4.3 Results

100 trials were run per combination of rule and reservoir size, and averaged to yield one result figure. All result figures are measured in testing recognition rate. In all but the last layer, the rate is the percentage correct classifications out of $D \times 370$ ($D = 4$), where it is out of only 370 in the last.

Table 4.8 presents first findings, from the initial runs. As in the 5-bit experiment, the whole group of rules that are subject for testing has been tested on the same reservoir parameters to find individual good performance rules. Furthermore, the top four rules have been selected for further testing with different reservoir parameters in Figure 4.8. Finally, Figure 4.9 is a rather special experiment, because, while Table 4.8 and Figure 4.8 presents recognition rates for immediate successive layers, Figure 4.9 presents the recognition rate at the final layer when the deepReCA has a specific number of layers.

Table 4.8: Recognition rate (%) for a three-layered architecture with $(I,R) = (20,20),(20,20),(20,20)$. Sorted by rule. Standard deviations are located in Table A.8.

Rule	Tot. fit time	Layer 1	Layer 2	Layer 3
2	15.82	31.08	19.70	72.92
16	15.81	32.17	18.60	69.35
22	13.70	25.49	30.15	58.47
30	13.35	27.77	33.18	65.96
36	15.37	26.30	35.25	65.92
41	13.63	31.86	37.27	67.74
45	15.56	31.70	38.26	68.76
54	13.64	32.52	38.55	64.03
60	13.62	46.98	51.78	72.91
62	13.50	31.62	34.89	61.23
90	13.76	46.29	49.88	72.72
102	13.47	46.40	51.04	73.42
106	13.93	43.52	49.44	74.19
110	13.66	31.92	37.49	68.21
126	13.77	27.86	33.98	64.31
146	13.77	35.17	39.38	64.85
150	13.80	19.33	27.63	56.98
153	17.07	46.77	51.65	72.97
165	16.15	46.35	50.15	72.45
182	13.88	27.27	32.31	59.39
195	16.72	46.46	50.79	72.87

4.4.4 Discussion

Consider first Table 4.8 which provides recognition rates for a greater selection of rules. It becomes apparent that layer 2 improves upon the output of layer 1 for almost every rule, though of minor scale. The exceptions are for class II rules 2 and 16, being left- and right

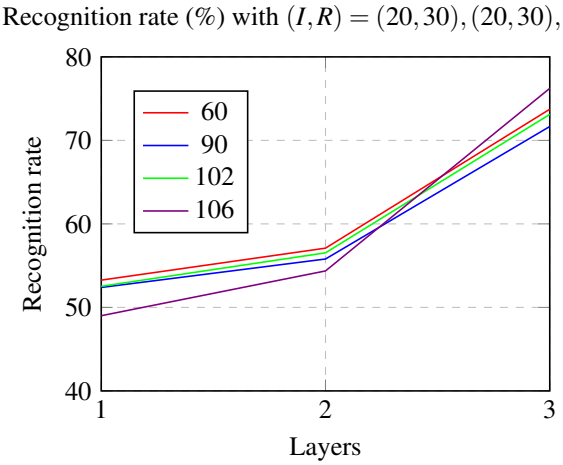

Figure 4.8: Recognition rate (%) for a three layered architecture with $(I,R) = (20,30),(20,30),(20,30)$. Standard deviations and accurate numbers are located in Table A.9.

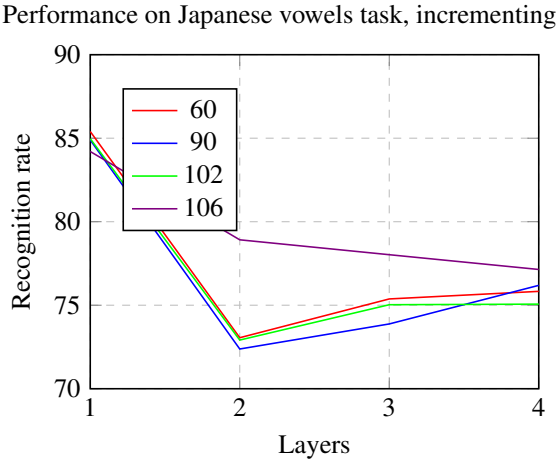

Figure 4.9: Recognition rate (%) at the final layer when incrementing the number of layers. Each layer have $(I,R) = (20,50)$. Standard deviations and accurate numbers are located in Table A.10.

shift rules, respectively. What is slightly off that trend is that even though rule 36 also a class II rule, it is not one of these exceptions. A possible explanation is, while 2 and 16 as transition functions completely discard all neighborhood configurations except for when the left/right neighbor is 1 ($\lambda = \frac{1}{8}$), 36 is slightly more dynamic ($\lambda = \frac{2}{8}$).

Why the overall percentage increase from layer 1 to layer 2 is small may be due to the low temporal context provided by layer 1. D is recommended to be a small integer [51], $D = 4$ in this work, so there is not much past context to consider when subsequent layers generate hypotheses. Let $h^l_{m,j}$ be the hypotheses that layer l produces for a sample, $m = 1,\ldots,9$, $j = 1,\ldots,D$. In an extreme example scenario, layer 1 produces hypotheses $h^1_{m,1} = (0, 0.9, 0, 0, 0, 0, 0, 0, 0)$ where the desired output is $d^1_{m,1} = (1, 0, 0, 0, 0, 0, 0, 0, 0)$. In deepReCA, $h^1_{m,1}$ is encoded and projected to the state space in layer 2. When layer 2 is to interpret this state vector, which is the first state vector in the sequence, it is no past information indicating that the meaning of $h^1_{m,1}$ is really $d^1_{m,1}$. This is partially adjusted for by employing Input-to-All. On the other hand, maybe significant increase from layer to layer (except for the final layer) is not needed in order for the final layer to predict well. If layer 1 produces $h^1_{2,1} = 1$, $h^1_{1,2} = 1$, $h^1_{1,3} = 1$, and 0 for all other $h^1_{m,j}$, yielding one vote for male 1 followed by two votes for male 2, then it is most probable that male 2 is the correct person.

Another observation from Table 4.8 is that the recognition rate increases drastically in the last layer, which in general is the trend regardless of the number of layers. The reason is because the last layer applies Alternative 4 (Section 3.2.3) to yield one prediction. Alternative 4 concatenates all state vectors into one large that thus constitutes the representations of all D hypotheses.

Increasing the number of random mappings should in theory improve performance because it increases the computational complexity. For the 5-bit memory task, this was true. Based on figures in Figure 4.8, it can be inferred to be true here as well, although it is only observed to be true up to a certain degree. When $R > 80$ and $I \approx 20$, the performance starts to stagnate and extra random mappings become redundant.

Equivalent rules do exhibit quite as equal performance, as in the 5-bit task. Inspecting rules 60, 102, 153, and 195 in Table 4.8, it is seen that they are 0.9 percentage points away from each other. Although this can vary between trials, it is surprisingly close. This strengthens the theory that equivalent rules do not have any important dissimilar characteristics when evolved, hence can yield similar performance.

During experimentation with different (I,R) and different number of layers, the addition of more layers did not consistently improve the result at the last layer. This is depicted in Figure 4.9, in which each N layer-column presents the recognition rate at the final layer in an N-layered architecture. The individual rates are thus independent. Notice that when using only 1 layer, the performance is significantly higher, roughly around 10 percentage points. In a 1-layered architecture, the last layer also becomes the first layer, hence it applies Alternative 4 on the computed reservoir state vectors that holds representation of the external input, and subsequently use the new state vectors for regression. Thus, it appears that a readout model that is fitted with vectors that represents the external input (though projected onto CA space), is better than a model that is fitted with vectors that represents hypotheses.

State-of-the-art performance on this task is hard to compete with. The donors of the

dataset presented the first results in [49] where they reached a recognition rate of 94.1 % on the test data. Their system was based on rule-based classification using passing-through regions, exploiting that values in discrete time series curves were entering and exiting thresholds and windows at different rate. For comparison, the dataset was also tested there with a 5-state continuous Hidden Markov Model, reaching a higher success of 96.2 %. Strickert [57] used a 1000 neuron self-organizing neural gas and achieved an error rate of 1.6 %, meaning a total of 6 misclassifications. Jaeger et al. [51] surpassed these recognition rates using ESNs with leaky-integrator neurons, successfully recognizing all utterances in the test data. Zero error was reached therein with 500 combined 4-neuron ESNs (with ”section experts”), where the final hypothesis/prediction is the mean of the individual ESN votes. Bye [43] was the first to apply the Japanese vowels dataset in ReCA. The success rate there was in the range around 30 %, which is not an exceptional performance.

For this deep architecture, the observed maximum recognition rate is 77.1 %, which is better than Bye's proposed ReCA architecture [43]. For the single layer, the maximum is approximately 85 %. One possible reason for not achieving a higher recognition rate is that the quantization method is too coarse, meaning too few levels to properly discriminate each time series. Another related possibility is that the quantize thresholds are feature independent, treating every one of the 12 feature dimensions equally. This could incur a problem considering what was found by Strickert [57]: Higher dimensions ($\rightarrow 12$) is more important or relevant than lower dimensions, and that the articulation endings are more discriminative than their beginnings. Appendix B presents four percentiles for each of the 12 dimensions. A third possibility that can hamper recognition ability, is the ”snapshot” design Alternative 3 (Section 3.2.3). Jaeger's equivalent alternative [51], interpolates between state vectors when the sequence length is not divisible on D, whereas deepReCA selects the closest vector. Interpolation can give more precision because the snapshots becomes more evenly spaced apart. Selecting the closest vector, i.e. Alternative 3 in Section 3.2.3, which in general terms means to round up or down, can be interpreted as a more unstable method. Non-interpolation was chosen herein because of uncertainty of whether interpolation would destroy valuable CA information.

The deepReCA system can come close to state-of-the-art performance for the Japanese vowels task. This shows that a deepReCA system can be applicable on real world tasks, hence a proof of concept. Outperforming state-of-the-art systems is outside the scope of work for this thesis. The use of CA as a reservoir substrate means speedups and lower cost of computation in general. Moreover, the use of hierarchical reservoirs as in deepReCA requires less memory because it is feasible to train and test one smaller layer at a time. Thus, deepReCA systems may be suitable if greater efficiency at the cost of lesser accuracy is desired.

Chapter 5

Analysis

5.1 General

While the previous chapter focused the discussion and observations on specific experiments, this chapter will try to present some general observations and analyze them in a generic fashion. It also serves as advice and recommendation for other researchers who are interested in implementing their own deepReCA system and enter further experimentation and analysis.

5.2 Training Algorithm

The current training algorithm concerns not the linear regression itself, but training the deepReCA as a whole. Section 3.3 presented the chosen algorithm which was designed based upon what is found in literature concerning deep RC with ESNs in [8, 58, 22, 21]. The trend seems to be to use the training target labels as the target for every readout model. Alternatively, in continuous speech recognition, train layer 1 to find short acoustic units with high frequency, and subsequent layers to find longer acoustic units with lower frequency [58].

Nevertheless, there is a problem that can arise when using the whole training dataset for offline (all-at-once) training. When layer 1 has been fitted to a training dataset, it must then predict in order for layer 2 to be trained. A requirement of this prediction is to use no test data because it would render the experiment invalid. If the system is deterministic, e.g. by using XOR as addition method between time steps, this prediction is based upon already seen information, meaning a very low train error. When these seemingly very good predictions are interpreted by layer 2, they cannot be refined much as they already are close to the target. Now, when layer 1 sees the test dataset, and subsequently predicts yielding a higher test error, layer 2 knows neither how to interpret nor refine this higher error. This problem does not arise with use of the very architecture presented in Section 3.3 since it uses normalized addition, a stochastic addition method.

5.3 Reservoir Dynamics

It is by no surprise that different rules provide different dynamics. Each rule has its own ability to express input of which the readout model can linearly separate, though equivalent rules are found to yield similar results. This was observed and understood already in the first use of CA as reservoir substrate [5], and also reproduced in the preliminary work of this thesis [10]. Because only a subset of all 256 rules were used in this thesis, a thorough and complete analysis on what λ provides useful behavior is difficult to elaborate. However, for the rules used, it is observed that rules with $\lambda \geq 0.5$ give good performance, with one exception for rule 146 where $\lambda = 0.375$.

A careful consideration to make is the selection of rule and the addition method as a combination. If the combination of the two is wrong, the ReCA is prone to enter an attractor. An attractor is one or several successive states that have been repeated. In general, a system that has reached an attractor cannot exit it unless it is perturbed [59]. In ReCA, the perturbation comes in the form of external input as in Equation 2.13, however, if the external input remains static, e.g. the waiting-for-cue signal in the memory tasks, the attractor can be reached again. For the synthetic memory tasks, the danger about occurring attractors is that it would render the length of the distractor period meaningless where it should matter. After all, they are long-short term memory tasks. Examples of different scenarios are provided in Figure 5.1, where different bitwise addition methods (addition of input and CA state vector) act in combination with two different rules. Figure 5.1 is a visualization of actual CA states as a ReCA system is solving the 5-bit memory task ($T_d = 20$, sample number 3 of 32). All 30 time steps are shown. Normalized addition is a stochastic addition method, whereas bitwise OR and XOR are deterministic. Attractors are seen in Figure 5.1f, and tend to appear after around 2 distractor periods. A few rules were tried, and it appears to be only the additive (also called linear rules by Yilmaz [34]) rules that can enter attractors. For clarity, the additive ECA rules are 0, 60, 90, 102, 150, 170, 204, and 240 [33]. This is by no means a proof that additive rules under XOR always enters attractors in ReCA, but rather an interesting observation that can be further investigated in future work.

5.4 Temporal Context

Triefenbach et al. [8] proposed a phoneme recognition system using hierarchical ESNs with leaky integrator neurons. The argument for stacking reservoirs was that additional reservoirs could correct errors based on the output of the first reservoir, while implicitly considering the past phonetic context. For the first reservoir, past context was the past external input, and for all subsequent reservoirs, past context was past decisions. The hypothesis turned out to be a valid one: A second reservoir was observed to produce clearer difference between competing outputs than the first reservoir. Past phonetic context becomes past temporal context when speaking more generic, which can further relate to any system consisting of multiple hierarchical reservoirs, including deepReCA. For tasks with several time steps of output, past temporal context can become quite meaningful. The longer the sequence, the more meaningful.

In a simple and extreme case, consider a classification task with sequences of length

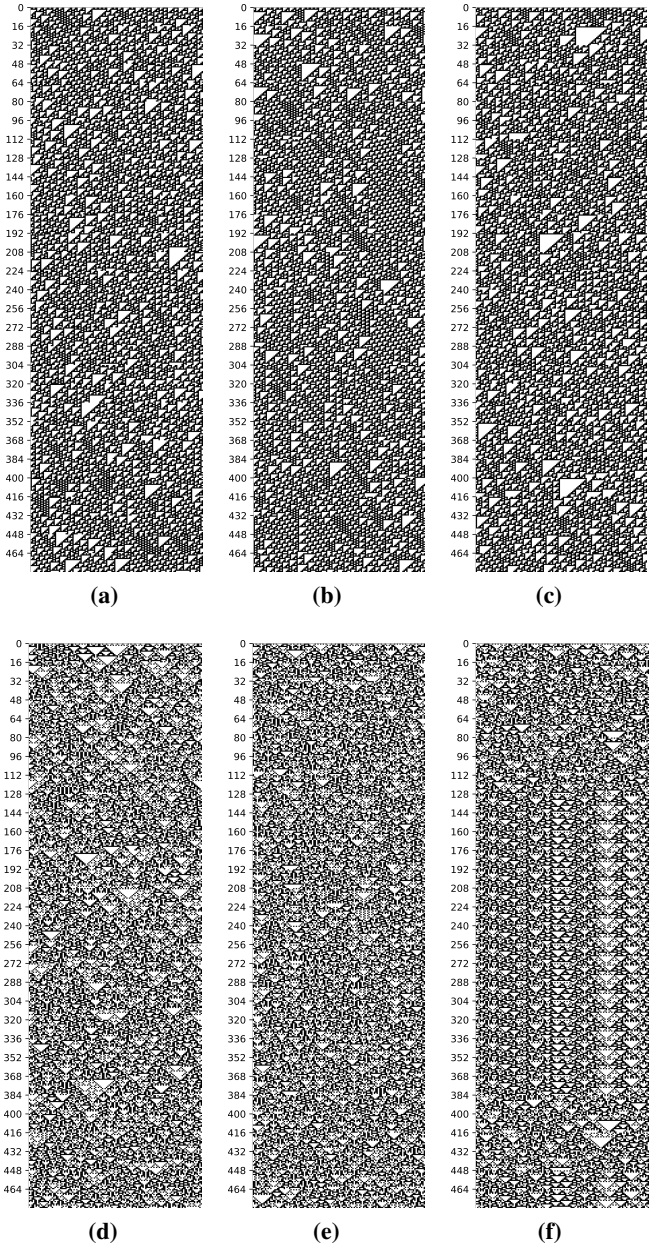

Figure 5.1: Runs of the 5-bit memory task with distractor period 20. The distractor bit starts at tick 80, and the cue bit is activated at tick 384. 32 random mappings and 16 iterations per time step. Rule 110 (top), and rule 90 (bottom). Normalized addition (left), bitwise OR (middle), and bitwise XOR (right).

1, and the desired output to be $\{a, b\}$. This means that output channel 1 is allocated to a, and channel 2 to b. In a layered reservoir computing system, if layer 1 outputs b where the desired output is a, then layer 2 learns the mapping $b \rightarrow a$. When layer 1 thereafter outputs b when that is the factual desired output, then layer 2 will reach an impasse.

In the proposed deepReCA, any layers' output for a time step is not always mutually exclusive, even though the desired output is. For the synthetic tasks, as mentioned in Chapter 3, the raw output values are binarized deterministically by Equation 2.1. In particular, if two or more output values are above the threshold of 0.5, they will each become 1, or, if no value is above the threshold, the output will consist of only zeros. In either case, it will mean a mispredicted time step. For the real-world task, output is binarized by quantization. Ideally, values should be sent to the subsequent reservoir substrate unaltered, which is the case in hierarchical ESNs, but due to the current reservoir substrate, values need to be translated first.

The hypothesis is that non-mutually exclusive bits together with past temporal context will aid a subsequent layer to make a decision. Temporal context may not only provide past inputs as a utility, but also implicitly serve as a guidance for at which point in time the current input is. In other words, how far the current input is from the initial input. Table 5.1 presents selectively chosen sample outputs (binarized) from the 5-bit task, run 26 of 32. The 5-bit is chosen because of its simplicity. Keep in mind that at time step t, a layer has not yet seen the input of time step $t+1$. The first two bits in are the memory pattern, and the third is "waiting for recall cue" signal. Inspecting the table, even though layer 1 outputs `000` both at time 207 and 208, the time at which they occur implies different meaning. Indeed, layer 2 is able to interpret and correct these two mispredictions. Layer 3 is thereafter able to correct all remaining bits.

Table 5.1: Sample binarized output (predictions) for the 5-bit memory task. Sequence 26 of 32. Underlined bits are what layer l (L l) was able to correct. Rule 62, normalized addition, $(I,R) = (30,30),(30,35),(30,35)$

Time step	Desired	L 1	L 2	L 3
206	100	001	001	<u>100</u>
207	010	000	<u>010</u>	010
208	100	000	<u>100</u>	100
209	010	001	001	<u>010</u>
210	010	<u>010</u>	010	010

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, an elementary cellular automata based deep reservoir computing architecture (deepReCA) has been proposed for the first time. The goal was to unite the effectiveness of cellular automata based reservoir computing and the paradigm of deep learning. The main motivation is to let layer l refine and correct mispredictions from layer $l-1$. Furthermore, a system with the proposed architecture has been implemented and benchmarked with pathological synthetic tasks and a real-world task. Specifically, the 5- and 20-bit memory task, and the Japanese vowels time series classification dataset. State-of-the-art work has been used for comparison for each task, all of which are already RC systems.

How much can several hierarchical layers improve upon a single layer (Research question Q1)? It is shown that successive ReCA layers can improve upon the output of previous layers. The greatest improvements occur closer to the first layer, where there is more mispredictions to correct. By measuring the success rate from layer to layer, it becomes clear that the system as a whole reaches an asymptotic performance, analogous to ESN based deep RC systems. If the goal of a task is to achieve 100 % recognition rate or 0 mispredictions, then the deeper (> 2) layers will have difficulties in accomplishing that, due to the aforementioned asymptotic performance.

How able is CA as a RC substrate to represent and discriminate different representations of external input (Research question Q2)? Different reservoir parameters, i.e. rule, the number of iterations I, and the number of random mappings R, are observed to affect the reservoir's ability to discriminate and represent input. In general, greater I and R increased computational capacity. One exception was for the 20-bit memory task, where increasing (I,R) was seen to decrease performance. However, the task was estimated to demand much more (I,R) in order to be solved, rendering the system impractical to train on a desktop computer. Rules were observed to have different impact on the reservoir dynamics. In particular, in the 5-bit task, rule 62 proved better as an "error correcting" rule whereas rule 146 proved better as an "initial hypothesis" rule. Both rules combined surpassed one-ruled deepReCA.

To what degree is a CA based deep RC system a viable option (Research question Q3)? On the 5-bit memory task, the implemented deepReCA achieved results similar to reproduced state-of-the-art results, and thus becomes a theoretical viable option with respect to memory demand: DeepReCA makes it possible to train and test one smaller layer at a time. On the Japanese vowels dataset, deepReCA reached a recognition rate of 77.1 %. Successfully classifying close to 80 %, i.e. the majority of the time series, means that deepReCA is applicable on this real-world task. This thesis has thus given a proof of concept, which opens up for further work based on the findings herein.

6.2 Future Work

This thesis touched upon some of the ideas mentioned in the preliminary work [10]. Yet, there is a vast spectrum of options and methods to choose from. These include among others mapping schemes for non-binary input, or more general, the preprocessing stage before exciting the medium within the reservoirs. There may exist subtler and more advanced schemes in literature which are better at preserving input features. CA as a model for computation is also worth investigating. DeepReCA in this thesis employed only elementary CA (ECA), one of the simplest kinds of CA. Totalistic CA is another type which Wolfram [32] used when discussing the different CA classes. Continuous CA [33] is a third possibility, where the cells take on vales not from a discrete set of states but rather a continuous set, hence the name. Combining Continuous CA with real-valued external input could be interesting, and may relate to Neural GPUs [60]. However, using one of these two alternatives may lead to lack of the effectiveness that ECA offers.

The space of (I,R) parameters have not been examined to full extent. A suggestion here is to find optimal reservoir parameters by using a simple search algorithm, e.g. hill climbing, simulated annealing, or even a simple evolutionary algorithm. Here, the search objective would be to find small enough reservoirs to minimize training time and -memory, yet large enough for them to produce meaningful output for their immediate successor to interpret.

CA completely avoids floating point multiplications as opposed to ESNs. To fully exploit the power of deepReCA, it can be desirable to implement it in FPGAs. Smaller reservoirs mean less training time and -memory for linear regression, and implementation on logic devices offers even more speedups. Taking the implementation another step further would be to replace the linear regression with summation [5]. Combining summation with using additive elementary CA rules, e.g. rule 90, one completely avoids multiplication in the whole system.

Bibliography

[1] Herbert Jaeger. The "echo state" approach to analysing and training recurrent neural networks-with an erratum note. 2001.

[2] Wolfgang Maass, Thomas Natschläger, and Henry Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531–2560, 2002.

[3] Chrisantha Fernando and Sampsa Sojakka. Pattern recognition in a bucket. In *European Conference on Artificial Life (ECAL)*, pages 588–597. Springer, 2003.

[4] Danko Nikolić, Stefan Haeusler, Wolf Singer, and Wolfgang Maass. Temporal dynamics of information content carried by neurons in the primary visual cortex. In *Advances in neural information processing systems*, pages 1041–1048, 2006.

[5] Ozgur Yilmaz. Reservoir computing using cellular automata. *arXiv preprint arXiv:1410.0162*, 2014.

[6] Chris G Langton. Computation at the edge of chaos: phase transitions and emergent computation. *Physica D: Nonlinear Phenomena*, 42(1):12–37, 1990.

[7] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[8] Fabian Triefenbach, Azarakhsh Jalalvand, Benjamin Schrauwen, and Jean-Pierre Martens. Phoneme recognition with large hierarchical reservoirs, 2010.

[9] Claudio Gallicchio, Alessio Micheli, and Luca Pedrelli. Deep reservoir computing: a critical experimental analysis. *Neurocomputing*, 2017.

[10] Stefano Nichele and Andreas Molund. Deep reservoir computing using cellular automata. *arXiv preprint arXiv:1703.02806*, 2017.

[11] Stefano Nichele and Andreas Molund. Deep learning with cellular automata based reservoir computing. In press, 2017.

[12] Yaneer Bar-Yam. *Dynamics of complex systems*, volume 213, chapter 0. Addison-Wesley Reading, MA, 1997.

[13] Hiroki Sayama. *Introduction to the modeling and analysis of complex systems*, chapter 1. Open SUNY Textbooks, 2015.

[14] Tom De Wolf and Tom Holvoet. Emergence versus self-organisation: Different concepts but promising when combined. In *International Workshop on Engineering Self-Organising Applications*, pages 1–15. Springer, 2004.

[15] Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. *ACM SIGGRAPH Computer Graphics*, 21(4):25–34, 1987.

[16] Herbert Jaeger. *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the" echo state network" approach*. GMD-Forschungszentrum Informationstechnik, 2002.

[17] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[18] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[21] Azarakhsh Jalalvand, Glenn Van Wallendael, and Rik Van de Walle. Real-time reservoir computing network-based systems for detection tasks on visual contents. In *Computational Intelligence, Communication Systems and Networks (CICSyN), 2015 7th International Conference on*, pages 146–151. IEEE, 2015.

[22] Azarakhsh Jalalvand, Fabian Triefenbach, Kris Demuynck, and Jean-Pierre Martens. Robust continuous digit recognition using reservoir computing. *Computer Speech & Language*, 30(1):135–158, 2015.

[23] Claudio Gallicchio and Alessio Micheli. Echo state property of deep reservoir computing networks. *Cognitive Computation*, pages 1–14, 2017.

[24] Xianhua Dai. Genetic regulatory systems modeled by recurrent neural network. In *International Symposium on Neural Networks*, pages 519–524. Springer, 2004.

[25] Ben Jones, Dov Stekel, Jon Rowe, and Chrisantha Fernando. Is there a liquid state machine in the bacterium escherichia coli? In *2007 IEEE Symposium on Artificial Life*, pages 187–191. Ieee, 2007.

[26] Matthew Dale, Susan Stepney, Julian F Miller, and Martin Trefzer. Reservoir computing in materio: An evaluation of configuration through evolution. In *IEEE International Conference on Evolvable Systems (ICES)*, 2016.

[27] Matthew Dale, Julian F Miller, Susan Stepney, and Martin A Trefzer. Evolving carbon nanotube reservoir computers. In *International Conference on Unconventional Computation and Natural Computation (UCNC)*, pages 49–61. Springer, 2016.

[28] Matthew Dale, Julian F Miller, and Susan Stepney. Reservoir computing as a model for in-materio computing. In *Advances in Unconventional Computing*, pages 533–571. Springer, 2017.

[29] Y Paquot, F Duport, A Smerieri, J Dambre, B Schrauwen, M Haelterman, and S Massar. Optoelectronic reservoir computing. *arXiv preprint arXiv:1111.7219*, 2011.

[30] Laurent Larger, Miguel C Soriano, Daniel Brunner, Lennert Appeltant, Jose M Gutiérrez, Luis Pesquera, Claudio R Mirasso, and Ingo Fischer. Photonic information processing beyond turing: an optoelectronic implementation of reservoir computing. *Optics Express*, 20(3):3241–3249, 2012.

[31] John Von Neumann, Arthur W Burks, et al. Theory of self-reproducing automata. *IEEE Transactions on Neural Networks*, 5(1):3–14, 1966.

[32] Stephen Wolfram. Universality and complexity in cellular automata. *Physica D: Nonlinear Phenomena*, 10(1):1–35, 1984.

[33] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 2002.

[34] Ozgur Yilmaz. Connectionist-symbolic machine intelligence using cellular automata based reservoir-hyperdimensional computing. *arXiv preprint arXiv:1503.00851*, 2015.

[35] Stefano Nichele and Magnus S Gundersen. Reservoir computing using non-uniform binary cellular automata. *arXiv preprint arXiv:1702.03812*, 2017.

[36] Mrwan Margem and Ozgür Yilmaz. An experimental study on cellular automata reservoir in pathological sequence learning tasks. 2016.

[37] Denis Kleyko, Sumeer Khan, Evgeny Osipov, and Suet-Peng Yong. Modality classification of medical images with distributed representations based on cellular automata reservoir computing. In *2017 IEEE International Symposium on Biomedical Imaging*, 2017.

[38] Nathan McDonald. Reservoir computing and extreme learning machines using pairs of cellular automata rules. *arXiv preprint arXiv:1703.05807*, 2017.

[39] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[40] David Snyder, Alireza Goudarzi, and Christof Teuscher. Computational capabilities of random automata networks for reservoir computing. *Physical Review E*, 87:042808, Apr 2013.

[41] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in Neural Information Processing Systems*, pages 4107–4115, 2016.

[42] Herbert Jaeger. Long short-term memory in echo state networks: Details of a simulation study. Technical report, Jacobs University Bremen, 2012.

[43] Emil Taylor Bye. Investigation of elementary cellular automata for reservoir computing. Master's thesis, Norwegian University of Science and Technology, June 2016.

[44] Moshe Sipper. The emergence of cellular computing, 1999.

[45] Matthew Cook. Universality in elementary cellular automata. *Complex Systems*, 15(1):1–40, 2004.

[46] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A framework for fast, scalable binarized neural network inference. *arXiv preprint arXiv:1612.07119*, 2016.

[47] Nicholas J Fraser, Yaman Umuroglu, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Scaling binarized neural networks on reconfigurable logic. In *To appear in the PARMA-DITAM workshop at HiPEAC*, volume 2017, 2017.

[48] R. A. Fisher. Iris data set. `https://archive.ics.uci.edu/ml/datasets/Iris`, 1936. Accessed: 2017-06-11.

[49] Mineichi Kudo, Jun Toyama, and Masaru Shimbo. Multidimensional curve classification using passing-through regions. *Pattern Recognition Letters*, 20(11):1103–1111, 1999.

[50] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[51] Herbert Jaeger, Mantas Lukoševičius, Dan Popovici, and Udo Siewert. Optimization and applications of echo state networks with leaky-integrator neurons. *Neural Networks*, 20(3):335–352, 2007.

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

[53] Andreas Molund. DeepReCA source code. `https://github.com/andreasmolund/reca`, 2017.

[54] The python language reference. version 2.7, `http://www.python.org`.

[55] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[56] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[57] Marc Strickert. *Self-organizing neural networks for sequence processing*. PhD thesis, University of Bielefeld, Germany, 2004.

[58] Fabian Triefenbach, Azarakhsh Jalalvand, Kris Demuynck, and Jean-Pierre Martens. Acoustic modeling with hierarchical reservoirs. *IEEE Transactions on Audio, Speech, and Language Processing*, 21(11):2439–2450, 2013.

[59] Carlos Gershenson. Introduction to random boolean networks. *arXiv preprint nlin/0408006*, 2004.

[60] Łukasz Kaiser and Ilya Sutskever. Neural GPUs learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

Appendix A

Tables With Standard Deviation

A.1 5-bit

Table A.1: Average number of successful runs for the 5-bit task. One layer. $(I,R) = (32,40)$

Rule	Tot. fit time	Layer 1
54	44.3	31.76 ± 0.51
62	44.3	31.78 ± 0.44
90	52.8	31.82 ± 0.61
102	37.4	31.98 ± 0.14
110	37.2	32.00 ± 0.00
146	49.8	32.00 ± 0.00
150	53.3	32.00 ± 0.00
165	54.1	32.00 ± 0.00
195	51.7	32.00 ± 0.00

Table A.2: Average number of successful runs for the 5-bit task. $(I,R) = (30,30),(30,35),(30,35),(30,35)$.

Rule	Tot. fit time	Layer 1	Layer 2	Layer 3	Layer 4
2	248.2	0.00 ± 0.00	0.00 ± 0.00	0.08 ± 0.31	0.16 ± 0.50
16	244.8	0.00 ± 0.00	0.00 ± 0.00	0.03 ± 0.17	0.13 ± 0.36
22	41.0	4.02 ± 1.92	6.64 ± 2.33	8.18 ± 2.48	8.87 ± 3.08
30	44.3	3.83 ± 2.07	8.84 ± 3.06	12.03 ± 3.03	14.51 ± 2.70
36	236.5	0.05 ± 0.22	6.77 ± 2.53	28.27 ± 2.86	30.91 ± 0.97
41	40.6	3.38 ± 1.79	10.34 ± 2.77	16.16 ± 2.78	20.59 ± 2.50
45	46.8	4.08 ± 2.06	15.02 ± 3.72	21.66 ± 3.01	25.76 ± 2.63
54	46.8	5.50 ± 2.17	19.32 ± 3.21	26.69 ± 2.20	29.64 ± 1.62
60	47.5	4.36 ± 2.15	16.33 ± 3.07	22.89 ± 2.58	26.84 ± 2.24
62	46.0	3.58 ± 1.86	17.71 ± 2.51	25.76 ± 2.28	29.50 ± 1.49
90	47.6	4.05 ± 2.03	15.93 ± 3.47	22.35 ± 3.01	25.81 ± 2.65
102	41.9	3.70 ± 1.96	16.24 ± 2.81	22.89 ± 2.60	26.72 ± 2.04
106	40.6	4.88 ± 2.18	12.11 ± 2.86	17.66 ± 3.11	22.18 ± 2.74
110	46.6	3.70 ± 1.95	15.48 ± 2.92	22.18 ± 2.55	26.30 ± 2.13
126	45.7	4.07 ± 2.10	6.01 ± 2.74	7.16 ± 2.95	7.32 ± 2.67
146	46.3	5.18 ± 2.37	17.59 ± 3.53	24.70 ± 2.40	28.20 ± 1.88
150	42.5	4.33 ± 2.28	1.31 ± 1.10	0.87 ± 0.97	0.59 ± 0.83
182	47.5	4.51 ± 2.14	13.02 ± 3.57	19.14 ± 3.52	22.22 ± 3.64
153	44.5	4.28 ± 2.08	15.66 ± 3.34	22.15 ± 3.10	26.36 ± 2.45
165	44.7	3.53 ± 1.81	15.57 ± 2.80	21.87 ± 2.71	26.08 ± 2.45
195	45.3	3.86 ± 1.86	16.27 ± 3.00	22.76 ± 2.73	26.65 ± 2.07

Table A.3: Average number of successful runs for the 5-bit task. $(I,R) = (30,30),(30,20),(30,20)$.

Rule	Tot. fit time	Layer 1	Layer 2	Layer 3
62	21.3	3.47 ± 2.03	2.39 ± 1.95	1.64 ± 1.57
54	22.2	5.79 ± 2.49	4.00 ± 3.28	2.44 ± 2.75
195	25.1	3.55 ± 1.94	0.07 ± 0.29	0.00 ± 0.00
110	19.2	3.79 ± 2.13	0.39 ± 0.66	0.00 ± 0.00
102	21.7	4.21 ± 1.94	0.05 ± 0.26	0.00 ± 0.00
165	23.5	4.34 ± 2.20	0.09 ± 0.29	0.01 ± 0.10
146	22.0	5.25 ± 2.13	0.93 ± 1.61	0.16 ± 0.52

Table A.4: Average number of successful runs for the 5-bit task. The letter *i* means that rule 62 is used in all but Layer 1. $(I,R) = (30,30),(30,30),(30,30),(30,30),(30,30)$.

Rule	Tot. fit time	Layer 1	Layer 2	Layer 3	Layer 4	Layer 5
54	40.8	5.61 ± 2.55	13.06 ± 4.00	18.64 ± 3.75	22.43 ± 3.42	24.32 ± 2.68
62	39.2	3.54 ± 1.66	12.43 ± 3.10	18.70 ± 3.23	22.91 ± 3.03	25.39 ± 2.68
102	47.2	4.01 ± 1.96	7.12 ± 2.90	9.23 ± 3.26	10.15 ± 3.34	10.39 ± 3.06
110	46.1	3.97 ± 2.18	8.36 ± 2.87	10.40 ± 3.31	11.87 ± 3.55	12.88 ± 3.59
146	50.9	5.49 ± 2.08	10.63 ± 3.49	13.50 ± 3.85	15.73 ± 3.81	17.04 ± 4.13
165	50.5	3.91 ± 2.03	6.67 ± 2.71	8.22 ± 2.81	9.00 ± 2.95	8.85 ± 3.01
195	53.9	3.70 ± 1.73	7.47 ± 2.27	9.50 ± 2.66	10.36 ± 3.06	10.71 ± 3.22
54i	49.2	5.56 ± 2.41	14.95 ± 3.26	19.74 ± 3.58	23.24 ± 3.17	25.30 ± 2.44
102i	49.5	3.93 ± 2.10	15.44 ± 3.26	20.79 ± 3.19	23.80 ± 3.03	26.02 ± 2.53
110i	49.2	4.05 ± 1.79	15.93 ± 3.00	20.95 ± 3.26	23.89 ± 2.86	25.93 ± 2.81
146i	46.9	5.34 ± 2.00	15.81 ± 3.13	21.11 ± 3.03	24.51 ± 2.63	26.52 ± 2.51
165i	49.0	4.17 ± 2.00	15.43 ± 3.38	20.64 ± 2.94	23.90 ± 2.83	25.73 ± 2.49
195i	47.2	3.88 ± 2.26	15.80 ± 3.44	21.55 ± 3.23	24.53 ± 2.95	26.48 ± 2.44

A.2 20-bit

Table A.5: Misclassified time steps for the 20-bit memory task. Here, $(I,R) = (16,20),(16,20),(16,20)$

Rule	Layer 1	Layer 2	Layer 3
2	1000.63 ± 1.02	2058.74 ± 162.90	1018.17 ± 35.32
16	1000.56 ± 0.88	2049.70 ± 185.06	1016.42 ± 21.98
22	1000.55 ± 0.74	1001.30 ± 2.51	1001.10 ± 1.01
30	1000.77 ± 0.98	1000.57 ± 0.75	1000.57 ± 0.79
36	1037.20 ± 11.81	2090.04 ± 199.22	1159.97 ± 99.65
41	1001.28 ± 1.25	1002.40 ± 1.71	1002.21 ± 1.52
45	1000.72 ± 0.88	1000.58 ± 0.83	1000.58 ± 0.83
54	1005.85 ± 9.71	1063.02 ± 94.23	1033.32 ± 47.30
60	1080.48 ± 21.70	2538.23 ± 267.36	1252.39 ± 165.95
62	1001.10 ± 1.15	1002.74 ± 3.14	1002.78 ± 3.93
90	1036.63 ± 22.21	1821.63 ± 501.25	1317.83 ± 508.83
102	1086.89 ± 21.47	2572.22 ± 270.32	1283.78 ± 192.78
106	1033.04 ± 9.25	1187.58 ± 87.14	1137.31 ± 87.48
110	1000.64 ± 0.71	1000.87 ± 0.90	1000.87 ± 0.88
126	1000.56 ± 0.77	1000.65 ± 0.82	1000.85 ± 1.00
146	1002.42 ± 2.83	1169.30 ± 324.35	1115.97 ± 237.19
150	1026.47 ± 29.78	1242.75 ± 514.14	1158.52 ± 329.06
182	1000.70 ± 0.89	1000.83 ± 1.06	1000.71 ± 0.85

Table A.6: Misclassified time steps for the 20-bit memory task where $(I,R) = (1,0),(1,0),(1,0)$

Rule	Layer 1	Layer 2	Layer 3
2	1000.00 ± 0.00	1000.00 ± 0.00	1000.00 ± 0.00
16	1000.00 ± 0.00	1000.00 ± 0.00	1000.00 ± 0.00
22	1000.00 ± 0.00	1000.00 ± 0.00	1000.00 ± 0.00
54	1000.00 ± 0.00	1000.00 ± 0.00	1000.00 ± 0.00
90	1000.00 ± 0.00	1000.00 ± 0.00	1000.00 ± 0.00
110	1000.00 ± 0.00	1000.00 ± 0.00	1000.00 ± 0.00

Table A.7: Misclassified time steps for the 20-bit memory task where $(I,R) = (16,100),(16,100),(16,100)$.

Rule	Layer 1	Layer 2	Layer 3
54	1430.33 ± 27.92	1810.92 ± 95.31	1497.77 ± 46.39
62	1422.20 ± 22.08	1482.15 ± 28.13	1468.48 ± 30.65

A.3 Japanese Vowels

Table A.8: Recognition rate (%) for a three-layered architecture with $(I,R) = (20,20),(20,20),(20,20)$.

Rule	Tot. fit time	Layer 1	Layer 2	Layer 3
2	15.82	31.08 ± 1.91	19.70 ± 1.10	72.92 ± 2.61
16	15.81	32.17 ± 2.27	18.60 ± 1.29	69.35 ± 8.18
22	13.70	25.49 ± 1.24	30.15 ± 1.12	58.47 ± 2.53
30	13.35	27.77 ± 1.06	33.18 ± 1.34	65.96 ± 2.49
36	15.37	26.30 ± 1.21	35.25 ± 1.29	65.92 ± 2.22
41	13.63	31.86 ± 1.19	37.27 ± 1.27	67.74 ± 2.22
45	15.56	31.70 ± 1.27	38.26 ± 1.19	68.76 ± 2.06
54	13.64	32.52 ± 1.15	38.55 ± 1.35	64.03 ± 2.74
60	13.62	46.98 ± 1.05	51.78 ± 1.14	72.91 ± 1.87
62	13.50	31.62 ± 1.07	34.89 ± 1.19	61.23 ± 3.53
90	13.76	46.29 ± 1.11	49.88 ± 1.07	72.72 ± 1.89
102	13.47	46.40 ± 1.24	51.04 ± 1.08	73.42 ± 1.83
106	13.93	43.52 ± 1.32	49.44 ± 1.16	74.19 ± 2.07
110	13.66	31.92 ± 1.32	37.49 ± 1.17	68.21 ± 1.93
126	13.77	27.86 ± 1.15	33.98 ± 1.00	64.31 ± 3.02
146	13.77	35.17 ± 1.18	39.38 ± 1.44	64.85 ± 2.58
150	13.80	19.33 ± 1.07	27.63 ± 0.85	56.98 ± 3.52
153	17.07	46.77 ± 1.44	51.65 ± 1.31	72.97 ± 2.01
165	16.15	46.35 ± 1.20	50.15 ± 1.03	72.45 ± 2.15
182	13.88	27.27 ± 1.31	32.31 ± 1.24	59.39 ± 2.67
195	16.72	46.46 ± 1.27	50.79 ± 1.20	72.87 ± 1.82

Table A.9: Recognition rate (%) for a three layered architecture with $(I,R) = (20,30),(20,30),(20,30)$.

Rule	Tot. fit time	Layer 1	Layer 2	Layer 3
60	25.4	53.27 ± 0.98	57.09 ± 1.06	73.73 ± 2.03
90	24.2	52.39 ± 1.20	55.79 ± 1.12	71.66 ± 2.24
102	25.2	52.52 ± 1.17	56.54 ± 1.12	73.11 ± 2.13
106	25.4	49.00 ± 1.31	54.37 ± 0.94	76.23 ± 2.11

Table A.10: Recognition rate (%) at the final layer when incrementing the number of layers. Each layer have $(I,R) = (20,50)$.

Rule	**Layer 1**	**Layer 2**	**Layer 3**	**Layer 4**
60	85.40 ± 1.36	73.05 ± 1.51	75.38 ± 1.57	75.83 ± 1.89
90	84.89 ± 1.26	72.38 ± 1.72	73.88 ± 1.66	76.19 ± 1.81
102	84.97 ± 1.12	72.91 ± 1.75	75.03 ± 1.55	75.06 ± 2.47
106	84.20 ± 1.26	78.92 ± 1.28	78.02 ± 1.97	77.14 ± 2.41

Appendix B

Percentiles for Japanese Vowels Dataset

Table B.1: More percentiles for the Japanese vowels dataset. Percentiles for the training portion.

Dimension	20-percentile	40-percentile	60-percentile	80-percentile
1	0.3992096	0.7515878	1.0710998	1.3112078
2	-0.901036	-0.6735048	-0.4684772	-0.2016738
3	-0.0067518	0.1855464	0.341525	0.510599
4	-0.5584518	-0.3535176	-0.1743104	0.012458
5	-0.0043106	0.167279	0.3031056	0.460379
6	-0.3585752	-0.2462606	-0.156919	-0.0607078
7	-0.3159534	-0.2169332	-0.1375486	-0.0406686
8	-0.2207288	-0.113377	-0.0134242	0.1177824
9	-0.378214	-0.2623512	-0.1649274	-0.0364958
10	-0.2839054	-0.212775	-0.1528258	-0.0836962
11	-0.103221	-0.0386028	0.0084956	0.0608074
12	-0.0154532	0.065216	0.1268862	0.1960958