NTNU
Norwegian University of
Science and Technology

Exploring a Developmental Reservoir Computing System Using Self-Modifying Recurrent Cartesian Genetic Programming

Anders Lima

Master of Science in Computer Science
Submission date: June 2017
Supervisor: Gunnar Tufte, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

Inspired by biology, numerous new computational models have been proposed as alternatives to cope with the ever-growing complexity of the traditional von Neumann architecture. Vastly parallel systems comprising simple units that only interact locally, form the basis of many of those new systems.

In this thesis, we combine ideas proposed in the field of bio-inspired unconventional architectures. Specifically, we explore the possibility of evolving a set of rules for a developmental reservoir. The reservoir is the heart of a computational model, coined reservoir computing. A reservoir computing system works by perturbing the reservoir with a stream of data. The reservoir will extract high-dimensional features of the data stream, which is classified in a readout layer by a linear classifier. A static network with a random recurrent topology is often used as a reservoir. However, we propose a self-modifying reservoir that is able to develop and adapt to the perturbations, by changing its size or structure. This allows the reservoir to self-organise in a way that enables it to transform the input into a high-dimensional feature set. Additionally, development will enable a large reservoir to be grown from a relatively small genotype.

The system implemented is an extension of a recurrent Cartesian genetic programming reservoir computing system presented in the specialisation project by the author. The extension support the self-modifying operations required in a developmental system.

Fitness functions based on separability and development is used in the endeavour of finding a self-organising computationally capable reservoir. We will explore how the developmental properties, genotype size, and the environment affect the reservoir. Additionally, the temporal parity problem is solved to demonstrate the system's performance.

The results show that finding a genotype that develops into a reservoir with the aforementioned features is rather difficult. Nevertheless, examples of working genotypes are found, serving as a proof of concept.

Sammendrag

Mange nye beregningsmodeller, inspirert av biologi, har blitt foreslått som alternativer for å takle den stadig voksende kompleksiteten i den tradisjonelle von Neumann-arkitekturen. Superparallelle systemer som består av enkle enheter som bare samhandler lokalt, danner grunnlaget for mange av disse nye systemene.

I denne oppgaven kombinerer vi ideer som foreslås innen bio-inspirerte ukonvensjonelle arkitekturer. Nærmere bestemt, undersøker vi muligheten for å utvikle et sett med regler for et utviklingsreservoar. Reservoaret er hjertet av en beregningsmodell, som kalles Reservoir Computing (RC). Et RC-system virker ved å forstyrre reservoaret med en datastrøm. Reservoaret vil trekke ut høydimensjonale trekk ved datastrømmen, som klassifiseres i et avlesingslag ved hjelp av en lineær klassifikator. Et statisk nettverk med en tilfeldig rekurrent topologi brukes ofte som reservoar. Imidlertid foreslår vi et selvmodifiserende reservoar som er i stand til å utvikle og tilpasse seg forstyrrelser, ved å endre størrelsen eller strukturen. Dette gjør at reservoaret kan organisere seg selv på en måte som gjør det mulig å omforme datastrømmen til et høydimensjonalt sett av karakterer. I tillegg vil utviklingen gjøre det mulig for et stort reservoar å bli utviklet fra en relativt liten genotype.

Systemet som er implementert er en forlengelse av et rekurrent kartesisk genetisk programmering (RCGP) RC-system presentert i spesialiseringsprosjektet av forfatteren. Utvidelsen støtter de selvmodifiserende operasjonene som kreves i et utviklingssystem.

Treningsfunksjoner basert på separabilitet og utvikling, brukes i forsøket på å finne et selvorganiserende beregnende reservoar. Vi vil undersøke hvordan utviklingsegenskapene, genotypestørrelsen og miljøet påvirker reservoaret. I tillegg løses et paritetsproblem for å demonstrere systemets ytelse.

Resultatene viser at det er ganske vanskelig å finne en genotype som utvikler seg til et reservoar med de nevnte egenskapene. Likevel fant vi eksempler på gode genotyper som fungerer som et bevis på konseptet.

Preface

This thesis is submitted to the Department of Computer Science at the Norwegian University of Science and Technology in Trondheim as partial fulfilment of the requirements for the degree of Master of Science. The work presented is a result of a specialisation project (fall 2016) and the master project (spring 2017) and was completed under supervision by Professor Gunnar Tufte.

I would like to thank Gunnar Tufte for guiding me through the last two semesters, and for providing valuable guidance and feedback.

Anders Lima
Trondheim, June 12, 2017

Table of Contents

List of Tables

List of Figures

Acronyms

AE Artificial Embryogeny. 17

ANN Artificial Neural Network. 1, 22

CA Cellular Automaton. 1

CC Computational Capability. 3, 30, 31, 38, 39

CGP Cartesian Genetic Programming. 2, 5, 7, 9, 18, 26, 27

DNA Deoxyribonucleic Acid. 14

DS Dynamical System. 12

EA Evolutionary Algorithm. 2, 3, 5, 6, 24

EC Evolutionary Computation. 5

EP Evolutionary Programming. 6

ES Evolutionary Strategy. 6, 24

ESN Echo State Network. 13

GA Genetic Algorithm. 6

GP Genetic Programming. 6, 7

LSM Liquid State Machine. 12

RBN Random Boolean Network. 1, 13

RC Reservoir Computing. 1, 2, 4, 5, 10, 12, 13, 19, 21–25, 37, 45, 46

RCGP Recurrent Cartesian Genetic Programming. 2, 9, 19, 21, 25, 45

RCGPANN Recurrent Cartesian Genetic Programming of Artificial Neural Networks. 9

RNN Recurrent Neural Network. 1, 2, 9, 10, 13

SM self-modifying. 2, 4, 18, 19, 21, 24–28, 45, 46

SMCGP Self-Modifying Cartesian Genetic Programming. 2, 5, 18, 21, 24, 27

SMRCGP Self-Modifying Recurrent Cartesian Genetic Programming. ix, 2, 4, 21, 24, 25, 27, 30, 36, 46

Chapter 1

Introduction

Every year computers are getting more and more powerful. However, this comes at the cost of increased complexity. To keep up with the ever-growing demand for computational power, we need to discover new, simpler, and more scalable computational models. By inspiration from nature, we will investigate how a combination of bio-inspired systems can come together to make an unconventional, robust, self-organising computational architecture incomparable to the traditional von Neumann architecture (von Neumann, 1945) prevalent in the last 70 years.

A flock of birds in an example of self-organisation in nature. The birds are able to move gracefully as if they were a single organism, despite the lack of a "master bird" controlling them. This behaviour is achievable when every bird follow simple rules, as keeping a minimum distance to the next bird and flying in the same direction as the birds ahead. Further, if a bird were to die, it would not impact the flock's ability to move in unison, which makes it a robust system. This behaviour is related to vastly parallel computational systems, based on local interactions between simple units. Cellular Automata (CAs) (Neumann and Burks, 1966) and Random Boolean Networks (RBNs) (Kauffman, 1969) are examples of dynamical systems following this paradigm.

Inspired by the neurons in brains, Artificial Neural Networks (ANNs) is an increasingly popular computational model. Feed forward neural networks, like convolutional neural networks, have successfully been applied to image recognition tasks (Le Cun et al., 1989). This is a spatial problem, however temporal problems have been solved by a recurrent variant, Recurrent Neural Networks (RNNs). They are also examples of dynamic systems, but in contrast to CAs and RBNs they use continuous values.

RNN is a popular computational model when solving complex time-series problems that require some degree of memory, such as natural language processing or sun spot prediction. However, training such networks is a hard and time consuming process (Pacanu et al., 2013; Bengio et al., 1994). Reservoir Computing (RC) approach this concern by using a network with a random recurrent topology and functions. This network is perturbed by an input stream, which the network transforms to a higher dimension, where the features can be classified by means of a linear classifier (Schrauwen et al., 2007; Jaeger, 2010;

Snyder et al., 2012). The RC-model excel in real-time computation on temporal data and have been successfully applied in various task, including robot motor control, weather- and financial prediction, noise reduction and speech-, voice- and handwriting recognition (Schrauwen et al., 2007; Lukoševičius et al., 2012).

This thesis tries to evolve a self-organising reservoir possessing certain desirable properties, including the *separation property*. This property, amongst others, have been reported to enable the reservoir in transforming the input data to a higher dimensional space where it is linearly separable, and thus classifiable with the logistic regression method (Natschläger et al., 2005).

Evolution has been used by scientists for many decades to find solutions to engineering and computational problems. Inspired by biological evolution, candidate solutions are mutated and reproduced to explore random directions of the solution space. Dynamical systems, like RNNs, are hard to program explicitly. Therefore, by using artificial evolution, we can evolve a good reservoir, without having to do an exhaustive search through all possible variations.

The dynamic properties required in a reservoir can be achieved in many different ways, two notable examples are a bucket of water (Fernando and Sojakka, 2003) and a cat's brain (Nikolić et al., 2007). In this thesis, however, the reservoir is implemented as a Cartesian Genetic Programming (CGP) (Miller and Thomson, 2000) graph. CGP is a model in the group of Evolutionary Algorithms (EAs), where we can employ artificial evolution to find a suitable execution graphs.

In the specialisation project by the author (Lima, 2016), a reservoir computing system was developed using Recurrent Cartesian Genetic Programming (RCGP). A computationally capable reservoir was successfully evolved and a correlation between the accuracy on solving problems and the separation property was found. Additionally, the RCGP-reservoir demonstrated ability to generalise and several tasks were solved by simply training a new readout layer.

In nature, however, evolution does not alter specific physical or behavioural traits of animals or plants directly, but rather it changes their genome. Further, the genome does not explicitly define the structure of these highly complex individuals. The genome acts as a recipe or blueprint on how to build the organism. The organism then develops by means of self-modifying operations. This creates a distinction between the genotype and the phenotype, which allows small genotypes to develop into large, complex phenotypes.

This thesis introduces development to the model presented in the specialisation project, allowing the reservoir to develop and adapt to perturbations. To evolve such a reservoir, we propose a Self-Modifying Recurrent Cartesian Genetic Programming (SMRCGP)-RC system, which adds recurrent capabilities to the Self-Modifying Cartesian Genetic Programming (SMCGP) model proposed by Harding et al. (2007). We will extend the CGP-Library developed by Turner and Miller (2015) to support self-modifying (SM) operations, and use this to generate the network we use as a reservoir. The library is also extended with various fitness functions, in the endeavour of evolving a computationally capable reservoir. The addition of self-modification, will hopefully allow the reservoir to self-organise and adapt to the perturbations. The readout layer will be implemented in Python using scikit-learn (Pedregosa et al., 2011).

A simplified version of the system built, is depicted in Figure 1.1. Here we can see an

Evolutionary Algorithm (EA) that will generate a number of candidate reservoirs. These reservoirs will be perturbed by an input stream. They might also develop over the course of execution, growing, shrinking, or alter itself in another way. The features of the reservoir are recorded by the EA, and the reservoir is assigned a fitness value. The best reservoirs, based on this value, are mutated to generate new, possibly better reservoirs. When the evolutionary search for a reservoir is complete, we train the weights of the readout layer to predict the correct result.

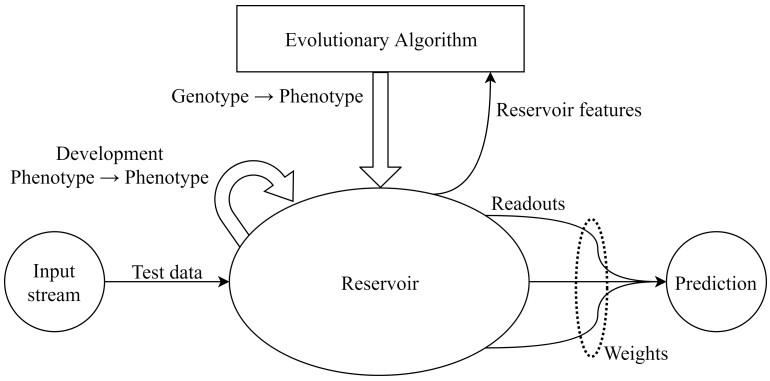


Figure 1.1: Simplified overview of the reservoir computing system

To verify the reservoir computing system's ability to perform useful computations, we will perturb the reservoir with a random bit-stream and train the system to solve the *temporal parity* problem. We will investigate if there is a correlation between Computational Capability and the accuracy of solving the task. The reservoir size required to solve the task will be tested, and finally we will investigate how well a reservoir adapt to new perturbations.

1.1 Terminology

Throughout this thesis we will use terminology borrowed from the realm of biology.

The terms borrowed from biology usually refer to a simplified digital version of the original term. In a population of possible solutions or *individuals*, we may refer to these as *genotypes* or *chromosomes*. These terms might be a bit misleading, as in nature every cell in an organism carries a number of chromosomes (made of DNA). The collection of these chromosomes in a cell are known as the genotype or *genome*. In this thesis, however, all individuals are related only to a single chromosome.

These chromosomes are made up of *genes* ordered in linear succession. A gene may control a feature of the organism or individual, in nature it might be eye colour, but in computers, a floating-point number or a character in a string. Variations of a gene is called *alleles* and the position in the chromosome is called *loci*. (Michalewicz, 1996)

1.2 Research Goals

The goal of this thesis is to implement and explore the possibility of a self-modifying-RC-system. Additionally, we aim to answer the following research questions:

Research Question 1: Is it possible to use a Self-Modifying Recurrent Cartesian Genetic Programming-graph as a reservoir?

Research Question 2: How does the genotype size affect the ability to develop into a functional reservoir?

Research Question 3: Will the self-modifying properties allow the graph to adapt to new environments, i.e. perturbations?

1.3 Thesis structure

The structure of this thesis is as follows: The background theory of the proposed system is explained in Chapter 2. The design and implementation is presented in Chapter 3 together with an explanation of the measures used to evaluate the system. The experiments, the task used in those, and the results will be presented in Chapter 4. Chapter 5 concludes this thesis with a discussion of the implemented system and the results.

Chapter 2

Background

This chapter will introduce the background theory of this thesis. First, we will look at how biological evolution have inspired a myriad of techniques and algorithms in computer science, including Cartesian Genetic Programming (CGP), which form the basis of the system implemented in this thesis.

Then, a brief introduction to complex and dynamical systems is given before the concept of Reservoir Computing (RC), and its relation to neural networks is explained. In the same section, we also look at how these reservoirs can do computations on the edge of chaos, and how they self-organise.

Towards the end of this chapter the concept of development is explained, and some examples of how it has been modelled is given. Combining development with CGP gave rise to Self-Modifying Cartesian Genetic Programming (SMCGP). This is what we will later use to implement the reservoir, and is explained in the second last section, before this chapter concludes with a summary of the preliminary work of this thesis.

2.1 Evolutionary Computation

The theory of evolution was proposed by Charles Darwin in the infamous book titled "On the Origin of Species" from 1895 (Darwin, 1859). Evolution is the change of physical or behavioural traits through successive generations, and suggests that the plethora of species we have today evolved from a common ancestor.

In the 1950s and 1960s engineers and computer scientists looked for inspiration to solve their problems, which some found in evolutionary biology. They followed the ideas of evolutionary biology, and successfully applied them as an optimisation tool on engineering problems (Mitchell, 1998). Many different approaches were developed independently, and algorithms and methods in computer science inspired by natural evolution have later been grouped together in a family named Evolutionary Computation (EC). Evolutionary Algorithms (EAs) is a group in this family, using a population-based, stochastic search method (Bentley and Kumar, 2003). The general idea of a EA is shown in Algorithm 1.

Algorithm 1: Evolutionary algorithm

Set initial generation $n \leftarrow 0$;
$A_0 \leftarrow$ Initial set of random individuals;
repeat
    foreach *individual* $a \in A_n$ **do**
        `evaluate` a;
    end
    `select` a set of parents B_n from the most fit $a \in A_n$;
    `reproduce` new children C_n from B_n;
    $n \leftarrow n + 1$;
    $A_n \leftarrow C_{n-1}$;
until *fitness target met or* $n >$ *maxGen*;

The four main types of algorithms in the Evolutionary Algorithm group include: Genetic Algorithm (GA) (Holland, 1975), Genetic Programming (GP) (Koza, 1992), Evolutionary Strategy (ES) (Rechenberg, 1973) and Evolutionary Programming (EP) (Fogel et al., 1966). We will in the remainder of this section take a closer look at the former three.

2.1.1 Genetic Algorithms

Holland (1975) introduced GAs as an abstraction of biological evolution. He represented the chromosomes as bit-strings, where each gene is represented as "1" or "0". A population is produced, and each individual is assigned a value of fitness. Higher fitness score of an individual allows it to produce more offspring than individuals with low fitness score. The offspring is produced using crossover (which mimics sexual reproduction by recombinating two chromosomes), mutation (random bit-flips) or inversion (rearranging gene order) (Mitchell, 1998). Mutations and inversions will allow the search to explore new random directions in the solution space.

GAs were further developed with Holland's colleagues and students, including Goldberg, who have written one of the most cited books on the topic: (Goldberg, 1989).

2.1.2 Evolutionary Strategies

Evolutionary Strategy (ES) is a population-based multimembered method of selecting a set of individuals to be passed down successive generations of an evolutionary search (Rechenberg, 1973). The strategy will deterministically select a fixed number of individuals from a population who, based on fitness ranking, will serve as parents in the next generation. In contrast to GA, the actual fitness score does not matter, only the ranking achieved relative to the other individuals.

More formally we denote the algorithm as $(\mu^{+}_{,}\lambda)$-ES, where $\mu, \lambda \in \mathbb{N}^{+}$. If using $(\mu + \lambda)$-ES, then μ parents create λ offspring, and based on fitness, λ of the $\mu + \lambda$ least fit individuals are discarded. If using (μ, λ)-ES, all parents are discarded. However, this require that $\lambda > \mu$, for there to be contest in the selection (Beyer and Schwefel, 2002).

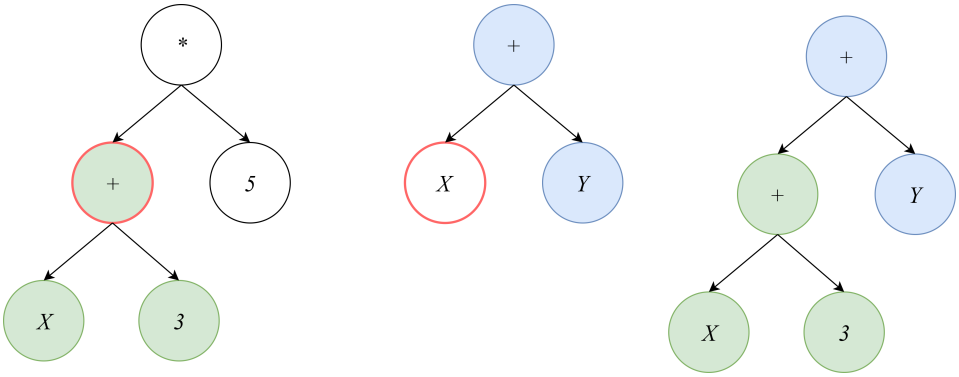


Figure 2.1: Two GP-trees producing a third using genetic crossover.

2.1.3 Genetic Programming

Koza (1992) argues that many machine learning and artificial intelligence problems can be viewed as requiring to discover a computer program that produce some desired output, given a set of inputs. He proposes a model where one does not have to write a such program explicitly, but rather let the system evolve to produce a program that solves the task. By applying the principles of Darwinism and natural selection to a pool of individuals, represented as programs or trees, he shows that it is possible to evolve a program capable of solving complex tasks, specifically the Boolean 11-bit multiplexer[1] (Koza, 1994).

A simple program can be visualised as a tree, Figure 2.1 depicts three different GP-trees, named T1, T2 and T3, from left to right. For instance, the T1 program solves the expression $((X+3)*5)$. Through evolution, new individuals (GP-trees) will be reproduced by means of genetic crossover between individuals in the current pool, called parents. The crossover point in this case is selected to be the nodes highlighted in red. The result of a crossover between the parents T1 and T2, is the child T3. This new program will evaluate the expression $((X + 3) + Y)$, which contain elements of both parents, highlighted in green and blue.

2.2 Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) was developed by Miller and Thomson (2000) to evolve digital circuits. It is a form of genetic programming, where an acyclic graph of compute nodes is placed in a regular grid. This is in contrast to the GP approach where nodes take the form of a tree. The nodes in CGP are indexed by their Cartesian coordinates in the grid, hence the naming, "Cartesian". An example of a CGP graph is shown in Figure 2.2. The upper number in each node denotes the node index[2] and the lower one

[1] The task of the 11-bit Boolean multiplexer is to decode a 3-bit binary address (000, 001, 010, ... , 111) and return the value of the corresponding data register (d0, d1, ... , d7). Thus, the Boolean 11-multiplexer is a function of 11 arguments: three, a0 to a2, determine the address, and eight, d0 to d7, determine the answer.

[2] Strictly speaking, output nodes are not nodes, but rather values pointing to the nodes whose values will serve as outputs, hence they do not have an index.

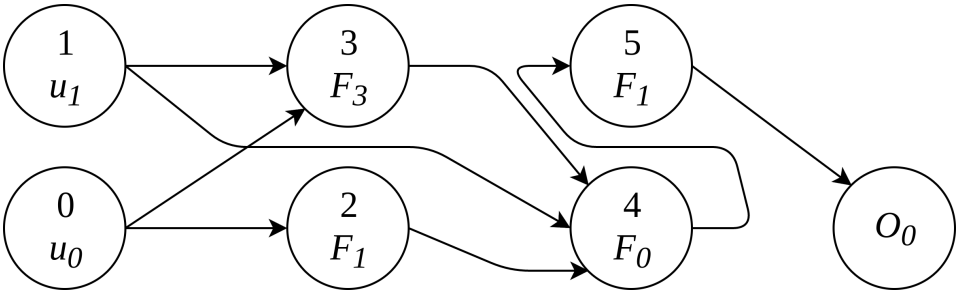
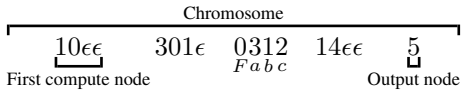


Figure 2.2: A Cartesian Genetic Programming graph comprising two inputs, four compute nodes and one output. The upper number denotes the node indices, and the lower the function or input associated with it.

denote the input/output number or the node function. Here we can see two input nodes, feeding values to three of the four compute nodes, and lastly one output node returning the value of node 5.

The chromosome describing the graph is of fixed length, as the number of nodes is predetermined. However, this does not oblige each node to do something useful or even be connected (directly or indirectly) to an output. Nodes irrelevant to the output (i.e. there are no paths from the node to an output) can simply be removed after the evolutionary search is finished, before executing the phenotype. This form of computing has also been shown to facilitate genetic drift. That is the ability for an inactive part of the graph to evolve without affecting its fitness, and which after several generations may become active. This feature has shown to be helpful to evolutionary process (Miller and Smith, 2006).

Originally the chromosomes were represented as a two-dimensional matrix, however, this is an unnecessary constraint as it is simple to represent the same the information in one dimension. We can see from Figure 2.2 that the nodes are laid out in a two-by-four matrix, but are indexed by scalar values (0 through 5). An example chromosome (that maps to the phenotype in Figure 2.2) is shown below.



We can see that there are four times four genes (or numbers), representing the compute nodes (3-5), followed by a single gene that denotes the node to be read as an output. Specifically, F is the node function (index in function table) and a, b and c are the source of the incoming connections (inputs).[3] ϵ denote 'not connected'.

2.2.1 Recurrent Cartesian Genetic Programming

Recurrent Cartesian Genetic Programming (RCGP) is an extension of CGP, allowing recurrent edges in the network, making the graph cyclic (Turner and Miller, 2014). This

[3]This is slightly simplified as the number of input nodes, total nodes, output nodes, node arity, and function set, is also part of the chromosome, but they are static over the course of evolution, i.e. immune to mutations.

relaxation allows the network to remember previous input and harbour dynamic behaviour.

A new parameter is introduced, in addition to the parameters of CGP, to control the occurrence of recurrent connections. This parameter specifies the probability for a mutated input gene to select a node that is "ahead" of itself, possibly introducing a cycle in the graph. Setting this parameter to 0.0 will effectively render the system indistinguishable from CGP.

The recurrency make the model somewhat comparable to RNNs, main difference being that it is the network topology and functions that are trained, not the weights of the inputs. Nonetheless, it can be used to solve similar problems. For instance, it has shown to perform well on partially observable tasks, including sunspot prediction and the artificial ant problem. Recently, the same authors have used RCGP to evolve artificial Recurrent Neural Networks, by using weighted connections and transfer functions (e.g. sigmoid or logistic), named Recurrent Cartesian Genetic Programming of Artificial Neural Networks (RCGPANN). This gave even better results than RCGP on predicting Mackey-Glass and sunspots (Turner and Miller, 2016).

2.3 Logistic regression

Logistic regression is a method of mapping a set of independent variables to a probability of an observation being member of a certain class. It was first proposed by Cox (1958).

The probability p that a set of features Φ belong to a given class $\mathcal{C}^x$ is given by Equation 2.1

$$p(\mathcal{C}^1|\Phi) = y(\Phi) = \sigma(\mathbf{w}^T\Phi) \tag{2.1}$$

, where $\mathbf{w}$ is the weight vector, and $\sigma(\cdot)$ is the logistic sigmoid function[4]. If we consider the problem of two-class classification ($x \in \{0, 1\}$), the probability of the features belonging to the other class is given in Equation 2.2 (Bishop, 2013).

$$p(\mathcal{C}^2|\Phi) = 1 - p(\mathcal{C}^1|\Phi) \tag{2.2}$$

Training is done by maximising the log likelihood of n class-feature tuples $\{(\mathcal{C}_i^x, \Phi_i)_{0,n-1}\}$, called the training set:

$$\underset{\mathbf{w}}{max} \sum_{i=0}^{n-1} log\ p(\mathcal{C}_i^x|\Phi_i\mathbf{w}) \tag{2.3}$$

2.4 Complex and Dynamical Systems

A *complex system* was informally defined by Simon (1965) as "one made up of a large number of parts that interact in a nonsimple way". Sayama (2013) elaborated on this and presented the following definition:

[4] $\sigma(a) = \frac{1}{1+exp(-a)}$

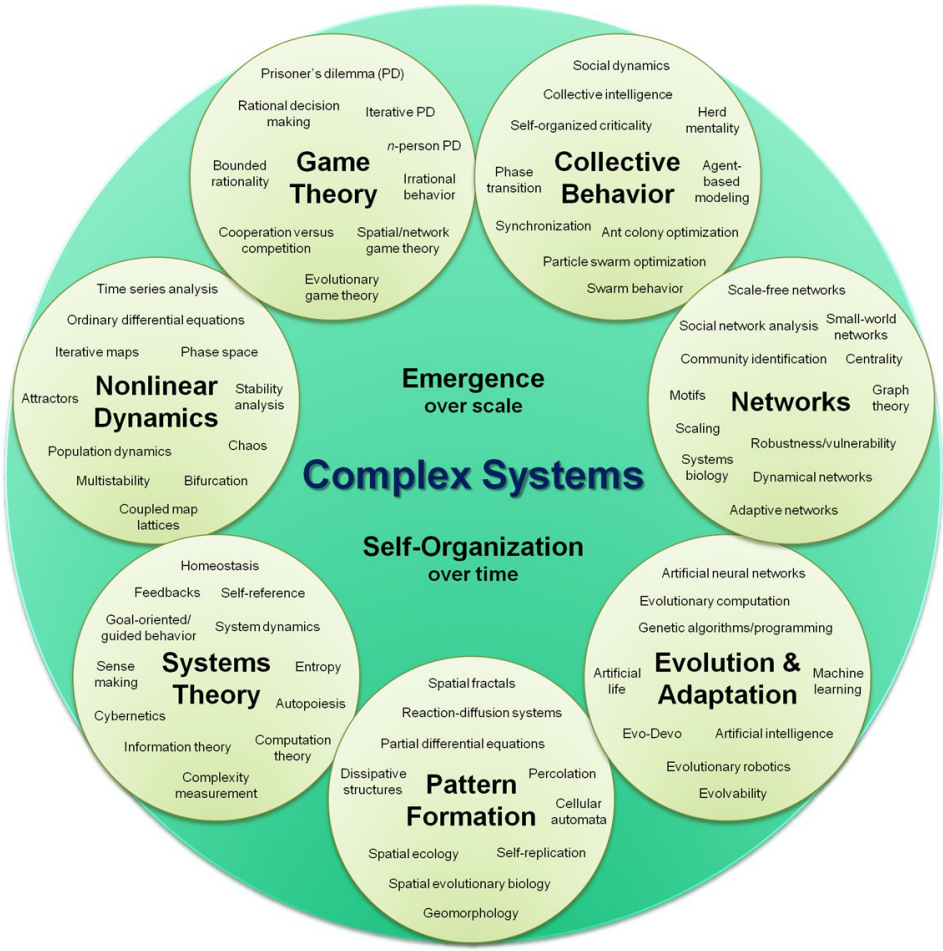


Figure 2.3: Complex systems overview. Figure created by Sayama (2010).

> Complex systems are networks made of a number of components that interact with each other, typically in a nonlinear fashion. These systems may arise and evolve through self-organisation, such that they are neither completely regular nor completely random, permitting development of emergent behaviour at macroscopic scales.

Complex systems a rather broad scientific area, Figure 2.3 gives an overview of the different fields it comprises.

A dynamical system is a system whose state is uniquely specified by a set of variables and whose behaviour is described by predefined rules (Sayama, 2013). It can be modelled in both continuous and discrete time, where the discrete one is known as a recurrence or difference equation, (see Equation 2.4), and in continuous time, its known as a differential equation (see Equation 2.5).

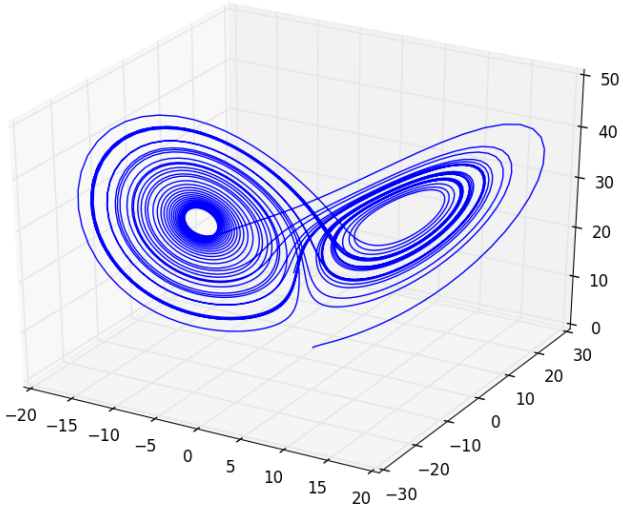


Figure 2.4: A solution of the Lorenz attractor with $\rho = 28$, $\sigma = 10$ and $\beta = \frac{8}{3}$ (made with matplotlib and scipy).

$$x_t = F(x_{t-1}, t) \tag{2.4}$$

$$\frac{dx}{dt} = F(x, t) \tag{2.5}$$

The set of all possible states in which a dynamical system may be, is referred to as the phase space. If the system enters a repeating pattern of states, it is said to have entered an attractor (Milnor, 1985). The union of all states that converge towards an attractor is known as the basin of the given attractor. An attractor can be a single state, known as a point attractor, when the state maps to itself. If a simple cycle of states is repeated, we have limit cycles or periodic orbits. Even more complex attractors, named strange attractors exists (see Figure 2.4), when the system operates in a more chaotic manner (Lorenz, 1963).

The aforementioned definitions and equations does, however, only consider deterministic systems. The design presented in Section 3.1 may include elements of stochasticity, which makes it hard to reason about. The theoretical foundation of stochastic dynamical systems is beyond the scope of this thesis, but interested readers may consult (Bhattacharya and Majumdar, 2004).

2.4.1 Reservoir Computing

Recurrent Neural Network (RNN) is a popular model when solving complex time series problems that require some degree of memory, like natural language processing or sun

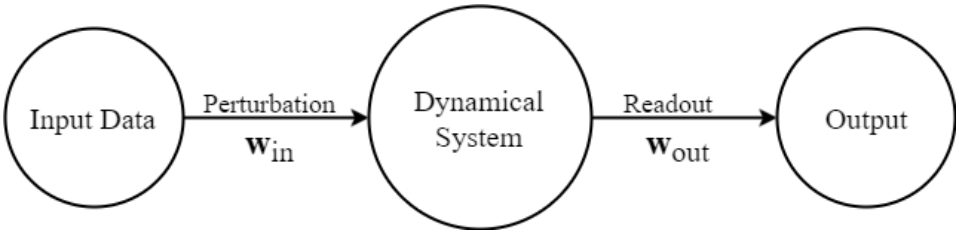


Figure 2.5: The three main parts of a Reservoir Computing System.

spot prediction. However, training such networks is a hard and time consuming process (Pacanu et al., 2013; Bengio et al., 1994). Reservoir Computing (RC) approach this concern by leaving the edges in the network untrained, using machine learning only in a simple readout layer. If the untrained network, or reservoir, is appropriately constructed, i.e. it exhibit certain features, it can be possible to extract useful information using a linear classifier (Schrauwen et al., 2007). The heart of reservoir systems is the complex behaviour expressed in the reservoir. Thus, an RC-system can be referred to as a complex system.

Figure 2.5 show the general idea of an RC-system, where input data perturb a Dynamical System (DS), and a set of readout edges with weights $\mathbf{W}_{out}$ are trained to extract desired features and map it to the output. $\mathbf{W}_{in}$ and the internal weights in the DS will remain fixed throughout training. One can also look at a reservoir as a mapping from a low-dimension input-space to a high-dimensional feature-space. It is this transformation that enables the linear classifier in the readout layer to classify the state of the reservoir.

Echo State Networks and Liquid State Machines

The two original Reservoir Computing (RC)-models were developed independently in the early 2000s. The Liquid State Machine (LSM) was proposed by Maass et al. (2002) as a real-time computational model for time-varying data. The work is biologically inspired and operates in continuous time. Echo State Networks (ESNs) were introduced by (Jaeger and Haas, 2004) as way of better utilising RNNs, without the difficulties of training. He proposes a reservoir that comprise randomly connected analogue neurons, driven by the input in discrete time. He views the reservoir as an "echo" of the previous inputs to the reservoir. The systems have in common that they use high-dimensional dynamical system in combination with a statistical learning method to perform computations.

The RC-model excels in real-time computation on temporal data and has been successfully applied in various task, including robot motor control, weather- and financial prediction, noise reduction and speech-, voice- and handwriting recognition (Schrauwen et al., 2007; Lukoševičius et al., 2012).

Reservoirs has been implemented in various ways, one of the more common methods is using a Random Boolean Network (RBN) (Snyder et al., 2012). There also exists electro-optical implementations (Duport et al., 2016). Even more original implementations have been reported, including using a bucket of water (Fernando and Sojakka, 2003) or cat's brains (Nikolić et al., 2007).

2.4.2 Computational Capability

The evolutionary process of a genetic algorithm needs guidance in the form of a fitness function, quantifying the desirable properties. The use of a genetic algorithm to guide a dynamic system towards a given behaviour was proposed by Packard (1988). This section introduces some of the metrics used in this thesis. The specifics on how these metrics are measured will be presented in Section 3.3.2

Several authors have suggested the separation property as an important emergent macroscopic property of a dynamical system performing computations (Maass et al., 2002). This property quantifies the system's ability to differentiate two distinct input streams, i.e. two different streams should put the system in two unique states. A system lacking separation will diminish or disable the readout function's ability to deduce any information from the system state.

This property has been proposed by others under different names:

- *NM*-Separation was purposed as a predictor for computational power by Natschläger et al. (2005).

- Snyder et al. (2012) split the separation property into *Kernel Quality* and *Generalization Rank* in their pursuit for an optimal Random Boolean Network reservoir.

The second property, named fading memory is presented as the second part of the computational capability metric in (Snyder et al., 2013; Natschläger et al., 2005). It is required for the network to eventually forget previous perturbations, i.e. the input will be "echoing" in the network a finite amount of time, hence the "echo state" naming of Jaeger's ESN model. A reservoir with fading memory will (if it remains unperturbed) eventually enter an attractor or steady state.

2.4.3 Computing at the Edge of Chaos

Computation in a general necessitates three properties; storing information, transmitting information and information processing (Lizier et al., 2014). A reservoir can operate in different dynamical regimes, and the aforementioned features has been shown to emerge when the reservoir resides in the transition between order and disorder, which is frequently referred to as the *edge of chaos*. This is also where Computational Capability seems to emerge (Langton, 1990; Packard, 1988). As a result, we will try to evolve a reservoir, which dynamics reside at the edge of chaos. Figure 2.6 depicts three examples of trajectories through the phase-space of the three dynamical regimes.

Although chaotic behaviour might look like random behaviour, it is deterministic. Chaos occurs when the system never falls into any attractors, and thereby do not show any repeating behaviour. A chaotic system is very sensitive to its starting conditions. A small variation, and the system might develop completely different. This behaviour is also known as the "butterfly effect". The system will, however, if started from the same initial state, repeat the exact same behaviour (Sayama, 2013).

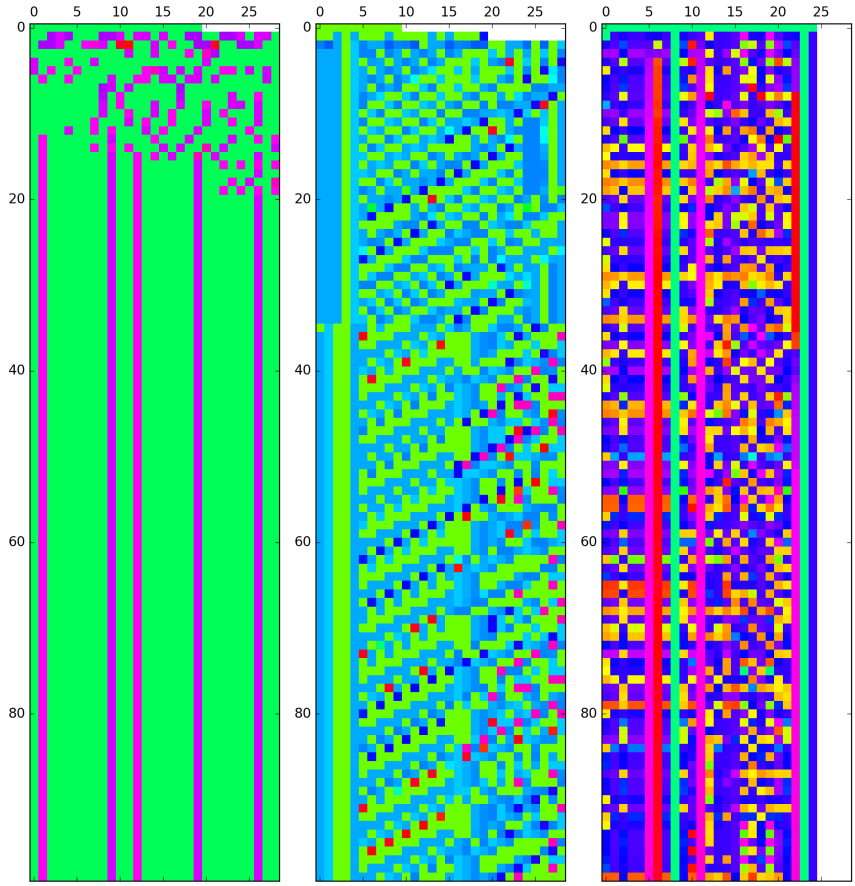


Figure 2.6: Three state-time plots recorded from the developed system when not perturbed. Each plot shows a reservoir with (a) ordered, (b) critical and (c) chaotic dynamics. The node indices are printed on top, while the iteration/time is on the left. The colours represent the relative values output from each node.

2.4.4 Self-organisation

The definition presented in the beginning of Section 2.4 by Sayama, mentions that complex system may arise by means of self-organisation. When a system is organised, seemingly without being put in place by a single entity, but rather emerge spontaneously, it can be said to have self-organised. One example of this is when a liquid freeze. There is no master molecule, directing the others into a crystallised formation.

Another example is magnetisation, where each piece of a magnetic material has its own spin, or "magnetic direction". The material is disordered, and since every part is configured in a random direction, the magnetic fields cancel each other out. If we heat the material, allowing for increased movement of each part, and then decrease the temperature again, the individual parts will, through local interactions, start to align themselves. As equal sides (two magnetic North poles) repel each other, the system as a whole will eventually have the same magnetic spin, or direction (Heylighen, 2001).

2.5 Development

Evolution has brought about a plethora of various plants and animals. In order to construct all these complex organisms, evolution created the process of development. Development is the self-organised process of creating a structure based on genetic information, the environment and its own properties. In biology, the genetic information is encoded in the DNA molecules that form chromosomes, which makes up the genome. This information does not describe how the organism should look or behave, but rather it describes the process of how to grow, and develop the organism. This section can be succinctly summarised by quoting Bentley and Kumar (2003):

> "Evolution designs life, but development builds it."

2.5.1 Morphogenesis

The development of structural features and patterns is known as *morphogenesis* (Doursat et al., 2013). Turing's pioneering work (Turing, 1952) on morphogenesis explains the chemical basis of how patterns in nature are developed, e.g. the dots on a leopard or stripes on a zebra(-fish). These patters take form by a reaction-diffusion system, starting from a uniform state, gradually forming patterns (see Figure 2.7 for an example). The patterns emerge as a result of local interactions (e.g. chemical reactions) and spatial spreading, or diffusion. This behaviour can be mathematically expressed as Equation 2.6, where $u = u(x, t)$, D is a matrix of diffusion coefficients and f is a function from $\mathbb{R}^n \to \mathbb{R}^n$, that is the local interactions in the system.

$$\frac{\partial u}{\partial t} = D\nabla^2 u + f(u) \tag{2.6}$$

From a computer scientist's view, Lindenmayer-systems or L-systems, might be a more interesting developmental model (Lindenmayer, 1968). It was originally introduced as a model for the growth of algae and fungi. The system model development as a rewriting

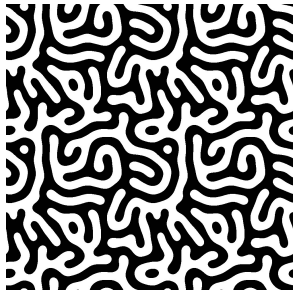


Figure 2.7: Turing pattern. Image courtesy of Jonathan McCabe (`https://flic.kr/p/a2Uh9q`).

system, where a string is rewritten by applying a set of rules. Formally, an L-system can be expressed by a context-free grammar, G, defined as a triplet:

$$G = (\Sigma, \omega, P) \tag{2.7}$$

, where Σ is the alphabet (a set of symbols), ω is the start string or axiom and P is the set of production rules. As an example, we can model a fractal tree with the following grammar:

$$\begin{array}{ll} \Sigma: & \{0, 1\} \\ \omega: & 0 \\ P: & \{1 \rightarrow 11, 0 \rightarrow 1[0]0\} \end{array}$$

, where '[' denote a 45-degree left angle and ']' a right angle. If we apply the production rules repeatedly on the string, starting with the axiom, it will expand, following Table 2.1. We can also show this graphically, if we represent 0 as a branch with a leaf, and 1 as a branch (or stem), we get a growing tree, as depicted in Figure 2.8. We could continue this growth indefinitely to get a tree as large as we want.

axiom	0
1st iteration	1[0]0
2nd iteration	11[1[0]0]1[0]0
3rd iteration	1111[11[1[0]0]1[0]0]11[1[0]0]1[0]0

Table 2.1: Development of a fractal tree

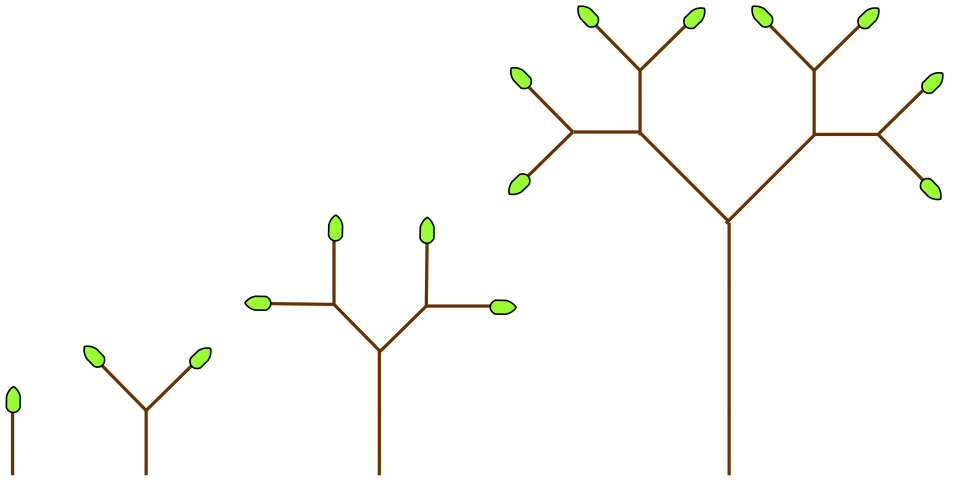


Figure 2.8: Growth of fractal tree, from the axiom as a sprout to the left to a large tree to the right.

2.5.2 Artificial Embryogeny

The developmental phase of artificial evolution is named artificial embryogenesis (AE) or Computational Development. In nature, the process where an organism goes from being a single cell, the *zygote*, to becoming a multicellular embryo, through dividing, growing and differentiating its cells, is named embryogenesis.

Miller (2004) presented a developmental system where an organism was able to autonomously grow a French flag. The multicellular organism he evolved could replicate and differentiate the cells through local interactions. Furthermore, the evolved organism also showed a remarkable ability to recover from damage.

Most genetic algorithms do not distinct between genotype and phenotype, or use a simple direct mapping between them. When development is introduced, the mapping from genotype to phenotype becomes indirect (Stanley, 2002). The phenotype may develop over time, so that there is possibly an infinite number of unique phenotypes, developed from a single genotype. Ideally one would want the phenotype to quickly develop certain structure and grow to a certain size.

One of the reasons why one would want an indirect encoding, is that direct encodings does not scale well. Large complex systems, e.g. neural networks, where each weight is represented as a floating-point number in the genotype, will get very large and hard to evolve (Harding and Miller, 2006). The size to which the phenotype grows should be independent of the size of the genotype. If the system hold this property, it is said to be *scale free*.

It is the property of self-organisation that allows a system to develop autonomously, however, it is the evolution that allows for adaption to an environment (Heylighen, 2001).

Environment

The environment in which the phenotype is developed might have great influence on the development itself. The magnet in the example in Section 2.4.4, for instance, is likely to align with the outer magnetic field. Kowaliw et al. (2007) developed truss based structures with spatial constraints and gravity, as environmental influence.

A system that is able to self-organise with regards to the environment is said to be adaptive. Tufte (2008) was able to evolve adaptable organisms by including environmental information into the gene regulatory mechanism of the organism.

2.6 Self-Modifying Cartesian Genetic Programming

Combining the ideas of development with CGP, we get a new system introduced by Harding et al. (2007), named Self-Modifying Cartesian Genetic Programming (SMCGP). This system extend the CGP model presented in Section 2.2 by allowing the graph to alter itself. It is implemented by adding a new type of nodes: self-modifying (SM)-nodes. These nodes have the ability to change the graph during execution, so that it develops over time. Incorporating the SM-nodes in the compute graph, allows the system to perform computations from the start, and concurrent with development.

As a result of the development, there are no longer a direct mapping between the genotype and phenotype. A single genotype may develop into an infinite number of unique phenotypes, depending on the environment (input data) in which it develops, and the features of the graph itself.

The genotype is slightly modified to accommodate some extra information required by the new SM nodes. The modification includes the addition of three extra genes per node, called *parameters*. Depending on the node in which they reside, they might specify the source or destination of duplication or removal of nodes in the graph. A detailed description of the nodes is presented in Section 3.2.3.

Figure 2.9 show the development of an SMCGP graph, starting with a phenotype that is a direct copy of the genotype, and then subsequent iterations where the phenotype develops.

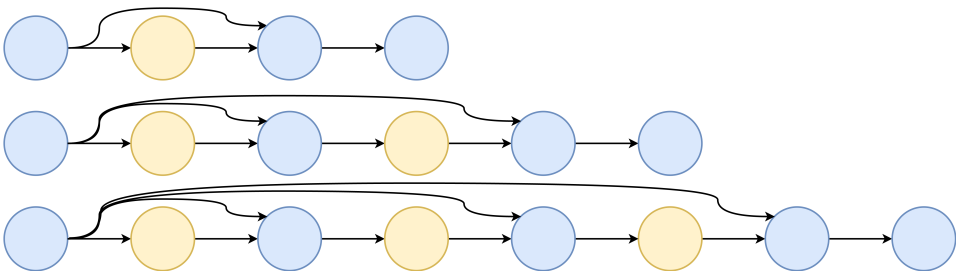


Figure 2.9: Simplified example of a growing SMCGP graph over four iterations. Starting with 4 nodes, including one SM-node (in yellow). One SM-node (the first) is activated at each iteration. This node duplicates itself and the next node, and inserts the copy at the its current position.

For recent developments of SMCGP, one can consult (Harding et al., 2010, 2011).

2.7 Reservoir Computing using RCGP

As much of the work presented in this thesis is based on a specialisation project[5], this section will summarise the work and results from said project[6].

The paper explores the possibility of evolving a reservoir using Recurrent Cartesian Genetic Programming. The system developed comprise the three major components of an RC system (Figure 2.5): an input layer, providing data to the reservoir, the reservoir itself, represented as an RCGP graph, and a readout layer. The readout layer employs logistic regression to classify the state of the reservoir.

The RCGP implementation is adapted from the CGP-Library (Turner and Miller, 2015), and the readout layer utilises scikit-learn (Pedregosa et al., 2011), a python library for machine learning, to implement the logistic regression.

Fitness functions based on separability and fading memory, the same as presented in Section 2.4.2, are tested in the endeavour of finding a computationally capable reservoir. The evolved reservoir is tasked with solving the temporal parity and temporal density problems, as a measure of its performance. The hyperparameters used for evolving most of the reservoirs is given in Table B.1.

Several experiments are conducted. First, to measure the correlation between the computational capability metric developed, and the ability to solve the temporal parity and density problem. Then the impact of the reservoir size on the ability to evolve solutions is tested. Finally, the generality of developed reservoirs is tested by evolving a reservoir to solve one task, and then use the same reservoir to solve other tasks, by only training a new readout layer, i.e. the weighting of the readout nodes.

It was shown a correlation between accuracy and the computational capability metric. However, it did not correlate with the fading memory portion of the metric, only the separation property. The results are available in Figures B.1 and B.2.

It was also shown that the number of nodes in the reservoir had a high impact on the system's ability to evolve a reservoir with high fitness. Figure B.3 show that reservoirs with more nodes faster evolve solutions with higher accuracy. Finally, the system was able to demonstrate a high level of generality. A system evolved to solve the temporal density problem with window size 5, was able to solve the other problems with an average accuracy over 90% (see Figure B.4), by only training a new readout layer.

[5] Course information is available at:
`http://www.ntnu.edu/studies/courses/TDT4501`

[6] The full paper is available at:
`https://folk.ntnu.no/anderlim/SpecializationProject_AndersLima.pdf`

Chapter 3

Methodology

To obtain a self-organising, dynamic structure, we propose Self-Modifying Recurrent Cartesian Genetic Programming (SMRCGP), as the reservoir in our system. SMRCGP combines the Recurrent Cartesian Genetic Programming model with the self-modifying properties of the Self-Modifying Cartesian Genetic Programming model. This chapter starts with providing and abstracted overview of the SMRCGP-RC-system design. Following that is an overview of how the various parts of the system is implemented. The last section will explain the measures used when evaluating the system.

3.1 Design

The design origins from the work presented in the specialisation project (Lima, 2016), which follows the Reservoir Computing idea described in Section 2.4.1. A summary of the project is available in Section 2.7. The changes made are chiefly concerned with adding support for self-modification and development. The base design of the system still follows the reservoir computing paradigm, and can thus be divided into the following three major components:

- Input layer to read and provide the reservoir with input data.
- Reservoir in which the input data flow to unveil hidden features of the input, i.e. is transformed to a higher dimension.
- Readout layer where the reservoir state is mapped to the desired output.

We will explain the workings of these components by use of an example. The three listed components are depicted in Figure 3.1, and is an example of how the working reservoir could be connected. In this example, we have one node in the input layer, which means there will only be one new data sample available to the reservoir at every time step. This value is fed into three nodes in the reservoir. These nodes can be referred to as the *input set* of the reservoir. One can say that the input data *perturb* the reservoir.

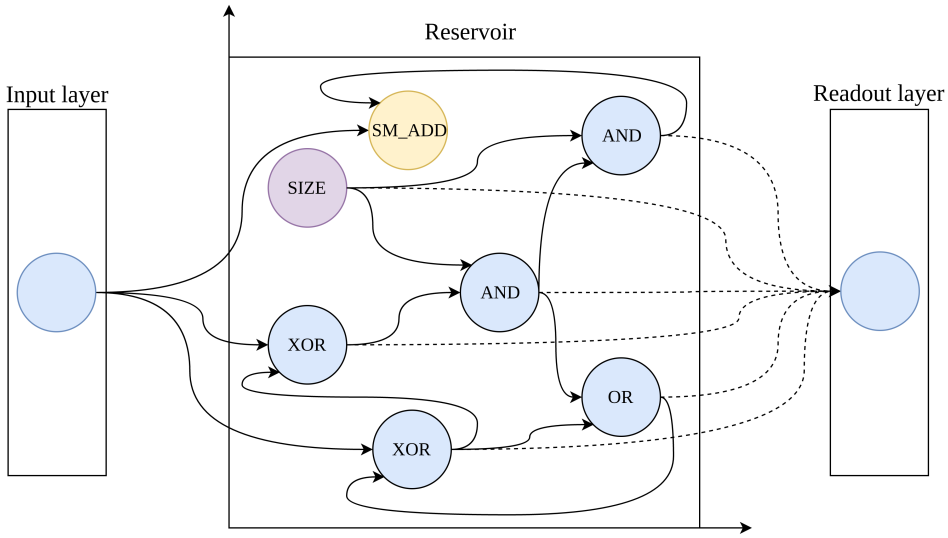


Figure 3.1: An instantiation of a simple RC-system with one input node, four computing nodes and two self-modifying nodes.

The reservoir itself comprise seven nodes, where five of them are marked blue, one purple and one yellow. This indicate the node category, which will be explained in detail later in Section 3.2.3. In Artificial Neural Network (ANN) terminology these nodes represent the hidden layer of the network. The purpose of these nodes is thus to transform the input stream to a high-dimensional feature space.

The readout layer, like the input layer, only contain one node. Therefore, the system will only save a single value at every time step. However, this node is connected to all the nodes who produce an output value (SM-nodes do not output any value), thus the result is a weighted value of these connected nodes. The set of these six nodes can be referred to as the *output set*. The lines are dashed, to signalise that they are subject to scaling by the logistic regression method.

Keep in mind that the above only stands an example, and is not likely to be a useful instantiation of the RC-system. In order for a useful reservoir to take form, we need to include some additional components to our system. Some of which are listed below:

- Machine learning module for configuring the readout layer
- Evolutionary algorithm to evolve the genome/staring graph
- Fitness function to evaluate a genotype
- Target filter to produce solution data for training.

An overview of all the components and their relations is available in Figure 3.2

To summarise the model, we can say it is a system which comprise the following: a number of input nodes I, connected to some nodes in the reservoir $\mathcal{R}$, a genetic algorithm

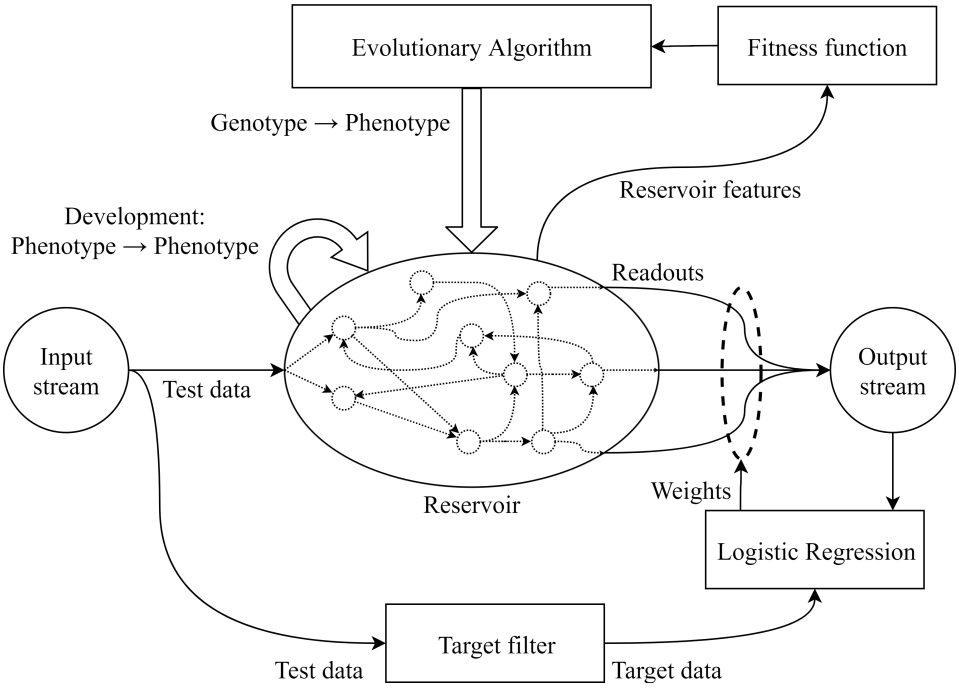


Figure 3.2: Overview of the reservoir computing system

$\mathcal{G}$ to evolve $\mathcal{R}$, which develops over time, and a readout layer $\mathcal{L}$ to classify the reservoir state. There is also a target filter $\mathcal{E}$ to produce the correct output for the readout layer's training and accuracy calculation. At last we have the Fitness function $\mathcal{F}$, used by $\mathcal{G}$ to quantify the genotype potential.

3.1.1 The Parts of the System

In this section, we will take a closer look at each of the components in the RC-system, shown in Figure 3.2.

Input

The size of the input set, I in the system is defined in the parameters and does not change during the course of execution. The input nodes are always at the start of the graph, and cannot be moved or deleted. The output values of these nodes is defined as a stream u, where $u_i \in \mathbb{R}^{|I|}$. If we use Boolean functions, the stream should be binary, and is defined as $u_i \in \{0,1\}^{|I|}$. i is the iteration number, i.e. the values are updated every iteration. We are chiefly concerned with single stream data, in this thesis, so $|I| = 1$, which implies that $u_i \in \mathbb{R}$.

Reservoir

We have relaxed the specification of the original SMCGP model to allow recurrent edges in the graph. This is done to allow the reservoir to have memory and harbour dynamic behaviour. As a result, we get the proposed SMRCGP model.

Thus, the reservoir, $\mathcal{R}$, is a SMRCGP-graph, which behaves as follows. Before the first iteration, the graph is a direct copy of the chromosome, and the values are initialised to 0.0. Then, at each iteration, the output of all nodes is updated according to the node function (see, Section 3.2.3). To make the order at which the nodes are updated inconsequential, each node store the input values, before the first node updates its output.

Additionally, a SM-node might be activated, such that the graph is updated. An activation may be self-induced (a result of the values in the reservoir), or as a result of perturbations by u. The node values are copied over to the new phenotype, so that the possible dynamics in the reservoir is not flushed whenever the graph is updated.

Evolutionary Algorithm

The Evolutionary Algorithm used by in the RC-system is the ES explained in Section 2.1.2. To evolve a computationally capable reservoir, we need to explore some of the possible reservoirs in the solution space. The task of this module is to produce a population of λ new $\mathcal{R}$ by mutating the currently μ best $\mathcal{R}$. By discarding the worst performing $\mathcal{R}$, and mutating the λ best, one will hopefully find a satisfactory individual, $\mathcal{R}$, after a number generations.

Fitness Function

In order for the Evolutionary Algorithm (EA) to rank the best individuals, we use a fitness function to evaluate the genotypes. This is done by executing the reservoir a fixed number of times (determined by `numDev`), while recording the state of each node. The measures used in the fitness functions are explained further in Section 3.3.

Target Filter

The target filter's task is to generate the solution data required by the logistic regression in the readout layer.

Readout Layer

The last part of the RC-system is the readout layer, which will map the state of the reservoir to a single class, $\mathcal{C}$, or value u_{out}. For the readout layer $\mathcal{L}$ to work, it will need to be trained, i.e. the weighting w of each node in the output set must be set such that it minimises the error when predicting the class $\mathcal{C}^x$. We use logistic regression (see Section 2.3) in $\mathcal{L}$ to find the weight vector $\mathbf{w}$ of the readout nodes. The state of $\mathcal{R}$ at time step t, S_t, is used as the features Φ_i in the logistic regression algorithm.

If the Evolutionary Algorithm have found a computationally capable $\mathcal{R}$, logistic regression should be able to quickly converge to a $\mathbf{w}$-vector, so that u_{out} matches u_{target}.

3.1.2 System Execution

This subsection will elaborate on how a computationally capable reservoir is produced. Algorithm 2 show how the reservoir $\mathcal{R}$ is evolved. After this algorithm is finished $\mathcal{L}$ can be trained to solve a particular problem, by supplying a training dataset. When $\mathcal{L}$ is sufficiently trained, the system can classify an arbitrary stream of data.

Algorithm 2: SMRCGP-RC-system execution

Input: u, maxGen
Output: Computationally capable $\mathcal{R}$

$u_{target} \leftarrow \mathcal{E}(u)$;
$gen \leftarrow 0$;
$\mathcal{G}$ generate λ random genotypes $\mathcal{R}$;
repeat
  foreach $\mathcal{R}$ **do**
    foreach *input* $x \in u$ **do**
      foreach *Node* $N_y \in \mathcal{R}$ **do**
        $N_y^{inputs} \leftarrow N_{inputs}^{output}$,
      end
      foreach *Node* $N_y \in \mathcal{R}$ **do**
        $N_y^{output} \leftarrow F_N({N_y}^{inputs})$;
      end
      Record $S_x \leftarrow$ as the state of $\mathcal{R}$;
    end
    Compute fitness of $\mathcal{R}$ as $\mathcal{F}(S)$;
  end
  Select the μ most fit $\mathcal{R}$;
  Generate a new set of λ $\mathcal{R}$ using $\mathcal{G}$;
  $gen \leftarrow gen + 1$;
until *gen* = *maxGen or* fitness goal reached;

3.2 Implementation

The specialisation project by the author (Lima, 2016), extensively relied on Andrew Turner's CGP-Library v.2.4[1] (Turner and Miller, 2015) to evolve RCGPs. In this thesis, we are concerned with evolving self-modifying RCGPs, which at the time is not supported by the CGP-Library. Therefore, we will need to extend the capabilities of the library to support self-modifying graphs. This section elaborates on the specifics of the implementation. gives an overview of the changes necessary, and how it is implemented.

[1] `http://cgplibrary.co.uk/`

3.2.1 Hyperparameters

Some aspects of the reservoir and how it is evolved are defined by a set of parameters. We call these parameters hyperparameters, so they are not confused with the parameters used in the execution (i.e. node or function parameters). They are usually the first thing that is specified in the code. Table 3.1 is an example of the parameters typical parameters. Most of the parameters are self-explanatory, however, we will give a brief explanation of each.

Inputs are the number of nodes in the input layer, i.e. the number of simultaneous input streams the reservoir accepts.

Nodes are the number of nodes in the reservoir (hidden nodes).

Outputs are the number of readouts from the graph. This value can be ignored since the readout layer will read all the nodes in the reservoir anyways.

Arity is the default number of inputs a node have. In Section 3.2.3 we will see that some nodes have a fixed number of inputs, and is thus not affected by this value.

Mutation Type specifies one of several ways to mutate the chromosome. The two most common are `probabilistic` which mutates every gene with equal probability, and `point`, which mutates a fixed number of genes equal to the total number of genes multiplied with the `mutation_rate`.

Mutation Rate specifies either the probability of a gene mutating or the percentage of genes to mutate, depending on the `mutation_type`.

Recurrent Connection Probability determines the chance that a mutated input gene points to a node "ahead" of itself, and possibly creating a recurrent connection.

Shortcut Connections determines if an output can be directly connected to an input.

Fitness Function sets the fitness function to evaluate the chromosomes.

Selection Scheme sets the function that determines which chromosomes that are selected as parents for the next generation. We use the default one, which always selects the most fit individuals. An alternative is tournament selection.

Reproduction Scheme sets the function that determines how new individuals are generated. We use the default one which selects one random parents which mutated to produce the children. An alternative might be crossover.

Threads sets the number of CPU-threads to use.

Number of Generations is the maximum number of generations the evolutionary algorithm will search.

DevRuns is the number of development steps taken into account when we evaluate the size of the graph.

Function Set lists the functions the nodes may have, followed by the size of the set in parenthesis.

3.2.2 Chromosomes

Both genotypes and phenotypes are referred to as chromosomes in the implementation. The information regarding a chromosome is contained within the `chromosome`-struct (see Listing A.3). The chromosome can be initialised by reading a file (`.chr`), from the parameters or by copying an existing chromosome in memory. If its initialised from parameters, only the size, inputs, outputs, arity and function set is predetermined. The node functions, inputs and arguments are initialised to random values.

Table 3.1: Typical SMRCGP hyperparameters

Evolutionary Strategy	(1+4)-ES
Inputs	1
Nodes	50
Outputs	1
Node Arity	2
Mutation Type	point
Mutation Rate	0.050000
Recurrent Connection Probability	0.400000
Shortcut Connections	0
Fitness Function	temporalParityAccuracy
Target Fitness	0.000000
Selection Scheme	selectFittest
Reproduction Scheme	mutateRandomParent
Threads	4
Number of Generations	1000
DevRuns	50
Function Set	add sub sin sm-dup sm-del index (6)

3.2.3 Nodes

Arguably the most important part of the system, are the nodes that make up the reservoir. Originally, in CGP, there were three kinds of nodes, namely **Input**, **Output** and **Compute**. Extending to the self-modifying version, we add the **Self-Modifying** nodes. In addition, we introduce a new kind if node named **Info** nodes. The compute, self-modifying and info nodes are similar in that they all are defined by the `node`-struct (see Listing A.1). This struct store all the information of a node, including what function the node should perform, the inputs it takes, the resulting output, and the arguments. However, not all three node types use all the information, but generalising these nodes to use the same structure simplifies the implementation.

It is worth mentioning that in contrast to the original SMCGP definition by Harding et al. (2007), the node indices are absolute, as in CGP.

Input Nodes

We already explained the behaviour of the input nodes in Section 3.1.1, but an elaboration on the implementation follows anyhow. At the initialisation of the system a data set is loaded into memory from a file. This is a comma separated file (`.csv`), that contain both the input stream, u and the correct output u_{target}. The data can then be retrieved by calling `getDataSetSampleInputs()` and providing the current iteration and index of the input node $\in I$.

Name	Operation
Add (ADD)	Add A_0 nodes after $(i + A_1)$
Delete (DEL)	Delete the nodes between $(i + A_0)$ and $(i + A_0 + A_1)$
Move (MOV)	Move the nodes between $(i + A_0)$ and $(i + A_0 + A_1)$ and insert after $(i + A_0 + A_2)$.
Duplicate (DUP)	Copy the nodes between $(i + A_0)$ and $(i + A_0 + A_1)$ and insert after $(i + A_0 + A_2)$.
Move, relative (MOVR)	Move the nodes between $(i + A_0)$ and $(i + A_0 + A_1)$ and insert after $(i + A_0 + A_2)$. Update the inputs to keep the relative distance.
Duplicate, relative (DUPR)	Copy the nodes between $(i + A_0)$ and $(i + A_0 + A_1)$ and insert after $(i + A_0 + A_2)$. Update the inputs to keep the relative distance.

Table 3.2: Self-modifying node definitions. The index of the SM-node is denoted by i, and the argument list as A

Self-Modifying Nodes

The developmental part of the system is implemented through a set of special nodes, named SM-nodes. An SM-node is activated when its first input is greater than its second. When these nodes are activated they are added to a ToDo-list with its position, function and arguments (see Listing A.4. After all the nodes have been executed, the ToDo-list is checked, and a predefined number of self-modifying operations are performed.

Examples of the SM-node supported are, SM_ADD, SM_DEL, the complete set of SM nodes are listed in Table 3.2. The ability to reuse parts of the graph is reported as an important capability of developing systems (Stanley, 2002). This ability is introduced through the SM_DUP and SM_DUPR nodes.

A large portion of the CGP-Library is changed to support self-modification during execution. As an example, the `executeChromosome()` function has been altered to return a new chromosome. This way, the chromosome is able to evolve over consecutive calls to `executeChromosome()`.

Info nodes

To give the graph some idea of its environment and status, we introduce a subset of nodes named Info nodes. These nodes aim to provide helpful information, aiding the graph's ability to adapt and be more robust. The different Info Nodes implemented is listed in Table 3.3.

The information nodes behave much like input nodes, in the way that is they do not take any inputs and can be considered primary nodes. Although the nodes in the implementation contain an argument list, it does not make use of it.

The `Index` or `Size` node for instance is very useful in limiting the growth of the graph. As input to an SM-node (directly or indirectly) the graph can grow until a certain

Name	Operation
Index (INDEX)	Outputs the index of the current node
Size (SIZE)	Outputs the number of nodes in the chromosome
Inputs (INPUTS)	Outputs the number of input nodes in the chromosome

Table 3.3: Information node definitions.

Name	Max Inputs	Operation
Real values		
add	n	$I_0 + I_1 + ... + I_{n-1}$
sub	n	$I_0 - I_1 - ... - I_{n-1}$
mul	n	$I_0 * I_1 * ... * I_{n-1}$
div	n	$I_0/I_1/.../I_{n-1}$
abs	1	$\|I_0\|$
sqrt	1	$\sqrt{I_0}$
sq	1	${I_0}^2$
cube	1	${I_0}^3$
pow	2	${I_0}^{I_1}$
exp	1	e^{I_0}
sine	1	$sin(I_0)$
cos	1	$cos(I_0)$
tan	1	$tan(I_0)$
Boolean values		
and	n	$I_0 \wedge I_1 \wedge ... \wedge I_{n-1}$
or	n	$I_0 \vee I_1 \vee ... \vee I_{n-1}$
xor	n	$I_0 \oplus I_1 \oplus ... \oplus I_{n-1}$
not	1	$\neg I_0$

Table 3.4: Compute node definitions. I_n is input number n

size before slowing down. Without these nodes, it might be hard to evolve a genome that is self-regulating.

Compute Nodes

The nodes in the graph responsible for producing output values are named compute nodes. They simply take a number of input values, and outputs a single value, according to their function definition. Table 3.4 list the functions supported by the system.

3.2.4 Fitness Functions

An evolving chromosome requires the fitness function to evaluate the chromosome in a new way. Since the SM_ADD node introduce new nodes with random properties, we have

a stochastic design. Therefore, we cannot evaluate a genotype by inspecting a single phenotype, as every phenotype could possibly develop differently each time. To cope with the stochasticity, we evaluate the genotype as an average over a given number of phenotypic developments.[2] The number of samples are defined in the parameters as `num_avg`.

Several different fitness functions have been developed, to evaluate various aspects of the system.

Developmental Fitness

It is crucial that the graph is able to grow to a size that can provide the complexity demanded by a given problem. A developmental fitness function was implemented to evaluate the initial development of a phenotype. The chromosome under evaluation is set to run for as many iterations as the `DevRuns` parameter specifies. Then the distance to a target size is calculated. Since the graph might grow indefinitely, and at a rate where the size after `DevRuns` is incidentally at the target size, we reset the graph, and run chromosome again for $2 \times$ `DevRuns`. A graph continuously growing will then overshoot the target size, the maximum distance to the target of both runs are used as a measure of developmental fitness.

Computational Fitness

How capable the reservoir is in performing computations need to be evaluated. This is done by the `computationalFitness` function. It is based on a Computational Capability (CC) metric, which will be explained in-depth in Section 3.3.2.

Accuracy Fitness

The `accuracyFitness` will evaluate the phenotype's ability to solve the actual problem in question. See Section 3.3.3 for how it is measured.

3.2.5 Libraries and Tools

This section present three libraries and tools used. The first two is used to implement the system. The third one is used to produce the figures presented in Chapter 4.

The CGP-Library

The SMRCGP implementation employed in this thesis is written as an extension to Andrew Turner's CGP-Library v.2.4[3] (Turner and Miller, 2015). The library is written in the C programming language, and can run on Linux, Windows and Mac OS. It is well documented and open sourced under the GNU Lesser General Public Licence[4].

[2] Later it was decided to not include the SM_ADD node to the function sets, as it gives inconsitent result that are hard to evaluate. Further it multiplies the evaluation time with `num_avg`. Changing the function to add a node with the NOP function might be a better implementation. This require a SM_CHF (change function) to later change the added node's function.

[3] `http://cgplibrary.co.uk/`

[4] `https://www.gnu.org/licenses/lgpl.html`

scikit-learn

scikit-learn was originally developed by David Cournapeau and is a free, open-source machine-learning library for python (Pedregosa et al., 2011). It supports a multitude of supervised and unsupervised learning algorithms, including, but not limited to, general linear models, support vector machines, naive Bayes, clustering and co-variance estimation.

Matplotlib

Matplotlib was initially developed by John Hunter, and is a 2D graphing and plotting tool for python (Droettboom et al., 2017). All graphs and plots in this thesis are produced using matplotlib.

3.3 Measures

Various measures are used by the fitness functions to evaluate different aspects of the system.

3.3.1 Self-Regulation

The first property we wanted to evaluate was the graph's capability to self-regulate. To test this, we tried to evolve a graph that was able to grow to a certain size, and then uphold the given size, even when perturbed. We used the developmental fitness function explained in Section 3.2.4 to evaluate this property.

$$\mathcal{Z}(\mathcal{R}, n_{target}, i) = \texttt{max}(\texttt{abs}(n_{target} - |N_a|),\ \texttt{abs}(n_{target} - |N_b|)) \tag{3.1}$$

, where n_{target} is the target size of the phenotype/graph, i is the parameter `devRuns` and N_a us the set of nodes in $\mathcal{R}$ after i iterations, and N_b is after $2 \times i$ iterations.

3.3.2 Computational Capability

The Computational Capability (CC) was introduced in Section 2.4.2 as a metric used by the evolutionary algorithm to find the most fit individuals. In the specialisation project, we followed the definition from Snyder et al. (2013) and defined the CC, or Δ of a reservoir $\mathcal{R}$, with input stream of length $\mathcal{T}$ to be:

$$\Delta(\mathcal{R}, \mathcal{T}, \tau) = \mathcal{S}_\tau(\mathcal{R}, \mathcal{T}) - \mathcal{M}(\mathcal{R}, \mathcal{T}) \tag{3.2}$$

, where $\mathcal{S}_\tau$ is a measure of the separation property of two streams where the τ first input values differ and $\mathcal{M}$ is a measure of the fading memory.

Measuring Separation

To measure how well the reservoir separates two different input streams, we need first to quantify the difference of two streams. Let u_a be an input stream and u_b be a variation of

u_a, where $|u_a| = |u_b| = \mathcal{T}$. The distance between two different streams in a reservoir, with N nodes can then be defined as

$$\mathcal{D}(\mathcal{R}, u_a, u_b) = \sum_{i=0}^{\mathcal{T}} \sum_{j=0}^{N} \frac{|A_{ij} - B_{ij}|}{\mathcal{T}N} \tag{3.3}$$

, where A and B are the states of the reservoirs perturbed with input u_a and u_b, respectively.

Following closely the function definitions presented in Snyder et al. (2013), we define the reservoirs ability to separate input streams of length $\mathcal{T}$, with the first τ values differ to be

$$\mathcal{S}_\tau(\mathcal{R}, \mathcal{T}) = \mathcal{D}(\mathcal{R}, u_a, u_b) \tag{3.4}$$

, where

$$u_b = \begin{cases} 1 - u_a, & \text{if } i < \mathcal{T} - \tau \\ u_a, & \text{otherwise} \end{cases}$$

Measuring Fading Memory

Again, we follow closely the definition from Snyder et al. (2013), and define the ability to forget past perturbations as

$$\mathcal{M}(\mathcal{R}, \mathcal{T}) = \mathcal{D}(\mathcal{R}, u_a, u_c) \tag{3.5}$$

, where

$$u_c = \begin{cases} 1 - u_a, & \text{if } i = 0 \\ u_a, & \text{otherwise} \end{cases}$$

However, it was shown in (Lima, 2016) that the fading memory metric had little or no correlation with the accuracy of solving the given task. The results are available in Figures B.1(c) and B.2(c). We will therefore spare the processor from computing this value, and instead use the following definition of Δ:

$$\Delta(\mathcal{R}, \mathcal{T}, \tau) = \mathcal{S}_\tau(\mathcal{R}, \mathcal{T}) \tag{3.6}$$

3.3.3 Task Accuracy

To determine how well the system compute a task, we will use a measure of accuracy. The accuracy is simply the fraction of correct outputs over the total number of outputs:

$$accuracy = 1 - \frac{sum(prediction \neq target\_data)}{len(prediction)} \tag{3.7}$$

Chapter 4

Experiments and Results

Several experiments were conducted to evaluate different aspects of the system. This chapter is structured such that each experiment is contained within a section. Each experiment will be given a short introduction, followed by the experimental setup. The results are then presented, before each section concludes with a short discussion.

The first experiment conducted evaluates the impact of the info nodes on the reservoirs ability to self-organise. Then we will test if the computational capability metric is correlated with the reservoir's accuracy on solving real problems. We will then test how small the genotype can be, and still be able to develop into a functional reservoir. At last we investigate the impact of the environment (input stream) on the reservoir structure.

Before we present the experiments, we will explain the problems the reservoirs are tasked with solving, and why they are used. This chapter conclude with a look at some interesting genotypes found while testing the system.

4.1 Tasks

To evaluate the system's ability to perform useful tasks we feed the reservoir with a stream of random values $v \in \{0, 1\}$. The system is then tasked with computing the *temporal parity*, $\mathcal{A}_n$ or *temporal density*, $\mathcal{B}_n$ of n bits, known as the *window size*.

The temporal parity task is a simple and good task for testing the system, as it necessitates memory in the reservoir. In simple words, it should classify bit i in a stream of $\mathcal{T}$ bits as "1" if the $n + \tau$ to τ most recent bits consist of an even number of "1" values, and "0" otherwise. τ is a delay introduced to allow the stream to propagate in the reservoir before it is classified.

Temporal density is concerned with finding the bit majority of the $n + \tau$ to τ most recent bits in the input stream.

P	Q	$P \oplus Q$
0	0	0
0	1	1
1	0	1
1	1	0

Table 4.1: XOR Truth table

4.1.1 Temporal Parity

This parity check is essentially the same as applying the XOR ($\oplus$) operation to the values in the window. The truth table for the operation in shown in Table 4.1. The target output stream generated by $\mathcal{E}$ can thus be defined as:

$$u_i^{target} = u_{i-\tau} \oplus u_{i-\tau-1} \oplus ... \oplus u_{i-\tau-(n-1)} \tag{4.1}$$

Another reason for using this parity or XOR task is that it is not linearly separable, in contrast to AND or OR. Logistic regression, described in Section 2.3, is a linear classifier, and consequently, will not be able to classify XOR. This is illustrated in Figure 4.1, where we can see that the outputs from AND and OR can be separated with one line, but XOR cannot. Each axis represents an input value (P or Q) and the colour of the dot show the result (blue is "0"/false and red is "1"/true). The issue of not being able to separate the result of XOR by a line is known as the XOR-problem.

In Figure 4.2 we have transformed the problem to a higher dimension, $\{0,1\}^2 \rightarrow \{0,1\}^3$, where we can see that it is separable by a plane.

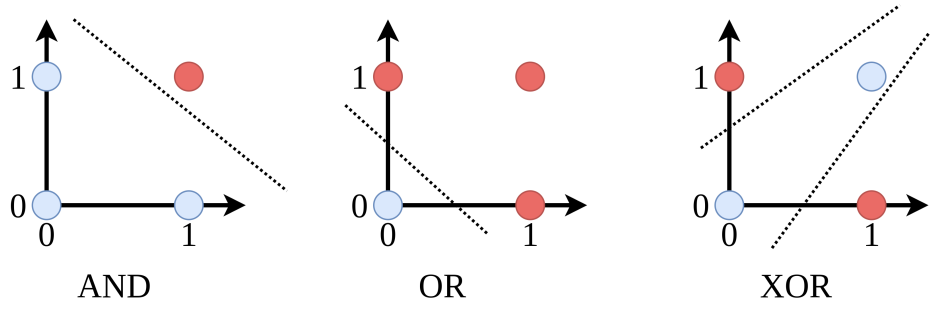


Figure 4.1: Separating values in AND, OR and XOR. Blue denotes 0 and red 1.

4.1.2 Temporal Density

Temporal density is concerned with finding the bit majority among the n most recent bits in the input stream. In other words, if there are more "1"s than "0"s in the last n bits, the correct output is "1", if not "0". This task is a bit simpler to solve, as it is linearly

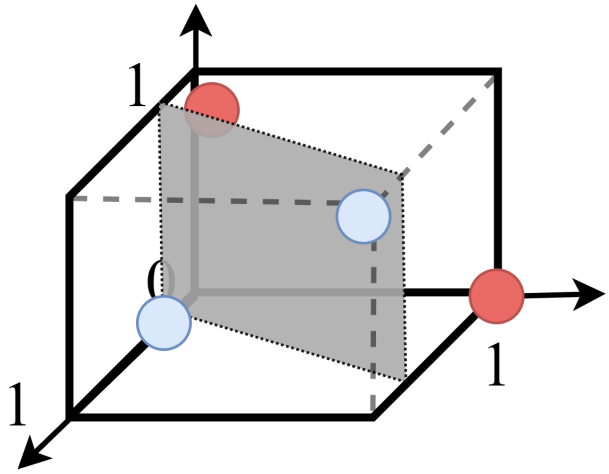


Figure 4.2: Separating XOR with a plane (in grey, bounded by dashed lines) through three dimensions. Blue denotes 0 and red 1.

separable. It does however require the reservoir to have a memory equal or greater than the window size.

The target stream when solving this task is defined as:

$$u_i^{target} = \begin{cases} 1, \text{ if } (u_{i-\tau} + u_{i-\tau-1} + ... + u_{i-\tau-(n-1)})/n > 0.5 \\ 0, \text{ otherwise} \end{cases} \tag{4.2}$$

4.2 Experiments

This section presents four of the experiments conducted to evaluate the system. All experiments were conducted on an workstation with a 8-core Intel® Core™ i7-4770 CPU @ 3.40GHz, and 16Gb of system memory, running Ubuntu 16.04 LTS.

4.2.1 Self-Regulation

The first property we want to evaluate is the graph's capability to self-regulate. To test this, we try to evolve a graph that is able to grow to a certain size, and then uphold the given size, even when perturbed. We use the developmental fitness function explained in Section 3.2.4 to evaluate this property.

One hypothesis we want to test is that the INDEX node will have a large impact on the reservoir's ability to self-regulate its size. Without this node, it the graph have no explicit information about its own size and would have to infer that information from attractor length or similar.

Table 4.2: Typical SMRCGP hyperparameters

Evolutionary Strategy	(2+8)-ES
Inputs	1
Nodes	40
Outputs	1
Node Arity	2
Mutation Type	point
Mutation Rate	0.050000
Recurrent Connection Probability	0.400000
Shortcut Connections	0
Fitness Function	ComputationalFitness
Target Fitness	0.000000
Selection Scheme	selectFittest
Reproduction Scheme	mutateRandomParent
Number of Generations	1000
DevRuns	50
Function Set	add sub mul sin cos sm-dup sm-mov sm-dupr sm-movr sm-del index (11)

Experimental setup

We evolve 50 reservoirs, twice, using the `developmentalFitness` function, setting the target graph size to 200 nodes. The first 50 runs lack the INDEX node in the function set, the 50 last do not. Table 4.2 show the hyperparameters used in the evolution, of course adding the INDEX node in the last 50 runs. The system will stop adding nodes if the graph exceeds 400 nodes to prevent the system from growing indefinitely and slowing down.

The `developmentalFitness` function will penalise phenotypes that after `devRuns` iterations is above $2 \times$ targetSize, below 10 nodes or still at the starting point (40 nodes).

Results

Figure 4.3 shows how 50 different chromosomes without the INDEX node grow over the course of 300 iterations. Adding the INDEX node to the function set, we get the results shown in Figure 4.4.

Discussion

In both cases, it is easy to observe that the majority of phenotypes will grow indefinitely (if it were not for the 400-node cap). With the INDEX node, we see from Figure 4.4 that only three of the genotypes were able to grow and sustain their size within the 150-350 node range. Without the INDEX node, only two genotypes managed to do the same. This is however a far too small data set to draw any conclusions, except that it is hard to evolve this behaviour in only 1000 generations. Of the 100 chromosomes, there were only a single that was able to regulate its size to around the target of 200 nodes (purple line in Figure 4.3).

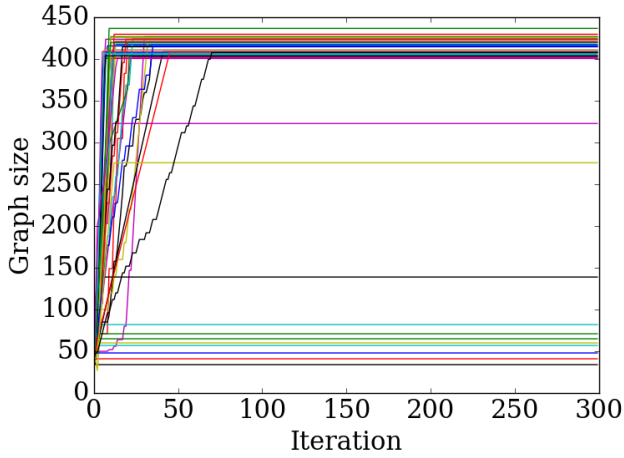


Figure 4.3: Graph size of 50 different chromosomes without any INDEX node, run for 300 iterations. There is a hard limit at 400 nodes, at which the graph is prevented from further growth

In both cases, we can observe that roughly 10% of the chromosomes are stabilising in the range of +/- 40 nodes from the starting point at 40 nodes.

4.2.2 Computational Capabilities

In the end, there is no use for this Reservoir Computing-system if it is not able to do any computations. We will in this experiment test the accuracy of the reservoir on a real problem, namely the temporal parity problem, introduced in Section 4.1.1.

Experimental setup

The test data is a list 1000 input-output pairs (u and u_{target}). The input stream is randomly generated, and the correct outputs are calculated by the target function for the temporal parity task, $\mathcal{E}_{\mathcal{A}_3}$. The evolution is set to run for a maximum of 1000 generations, but will terminate the search if an individual report 100% accuracy. First the fitness function is set to Computational Capability; the experiment is then repeated with accuracy as the fitness measure.

Results

Figures 4.5 and 4.6 show the Accuracy when solving $\mathcal{A}_3$ plotted against the Computational Capability-fitness. The former is evolved using CC as fitness measure, whereas the latter is evolved with the accuracy as fitness measure. Both figures contain 9000 points, each representing a single individual. It looks like there are fewer points in Figure 4.5, but is the result of many overlapping points.

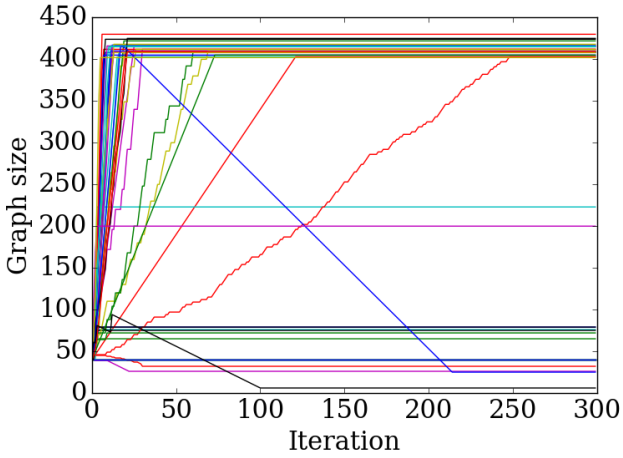


Figure 4.4: Graph size of 50 different chromosomes with the INDEX node, run for 300 iterations. There is a hard limit at 400 nodes, at which the graph is prevented from further growth

Discussion

In contrast to the equivalent experiment performed in (Lima, 2016) (results are available in Figures B.1 and B.2, but note that the CC values are not fitness values, but the measure itself), it is very hard to observe any correlation between Computational Capability and Accuracy. In Figure 4.5 there seems to be equal amount of 100% accurate solutions with bad CC as there are with good. However, when evolved with Accuracy as fitness measure the majority of solutions found seem to have a higher CC.

4.2.3 Genotype Size

One of the benefits of using a developmental model is that one can encode a complex and large organism, using a relatively small genotype. In this experiment, we will investigate the impact of the genotype size on the ability to self-regulate and become computationally capable.

Experimental setup

We evolve 50 genotypes, the first 25 with a genotype size of 20, and then 25 with genotype size of 40. The fitness function used is a combination of CC and development.

Results

Figure 4.7 show the how well the reservoirs are able to separate the input stream, when starting from 20 or 40 nodes in the genotype. 25 runs for each case is plotted, where the shaded areas are bounded by the 25th and 75th percentiles, while the line between

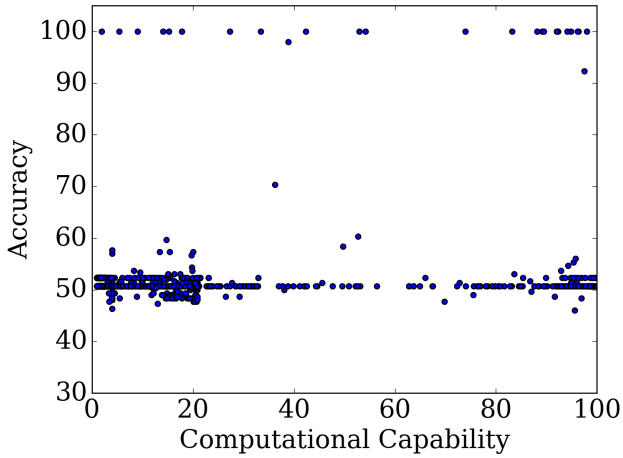


Figure 4.5: Accuracy plotted against the Computational Capability fitness of 9000 individuals (taken from all stages of evolution), when evolved with CC as fitness measure.

them is the median. The y-axis is the fitness on separation property (1 is optimal). Figures 4.9 and 4.10 show how 25 genotypes of size 40 and 20, respectively, grows over time. Figure 4.8 shows the accuracy obtained on $\mathcal{A}_3$ and $\mathcal{A}_5$ when the genotype size N $\in \{20, 40\}$.

Discussion

From Figure 4.7 we can see that it is easier to achieve better fitness score on separation when we have fewer nodes. This is probably a result of the fact that the fitness score is based on the percentage of the nodes that have changed its output due to different input data. When we have a larger number of nodes it is less likely that an equally large portion of the nodes are affected by the input. Figures 4.9 and 4.10 show that a smaller genotype are likely to have a slower growth rate.

4.2.4 Environment

The environment, which in this system is the input stream, should have an impact on the development of the phenotype. We will in this experiment test how a change in the environment affects the size of the phenotype.

Experimental setup

The experiment from Section 4.2.1 is repeated, only with a different input stream: $u_i = 0.0$ for $0 \leq i < 100,$ and $\in \{0.0, 1.0\}$ for $100 \leq i$.

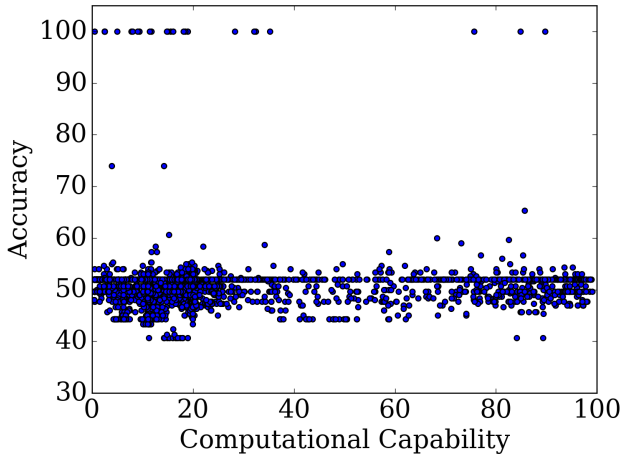


Figure 4.6: Accuracy plotted against the Computational Capability fitness of 9000 individuals (taken from all stages of evolution), when evolved with Accuracy as fitness measure.

Results

Figures 4.11 and 4.12 show how a change in the environment affects the reservoir size.

Discussion

The figures show that only a few genotypes respond to the change in perturbations. The INDEX node does not seem to have any effect on the ability to adapt to the environment. However, no part of the fitness functions evaluated the ability to adapt to changing perturbations, making this behaviour probable.

4.3 Successful Individuals

This section does not display a single experiment, but shows some interesting results that were found while testing the system.

4.3.1 Examples

One of the first genotypes that was able to achieve 100% accuracy on $\mathcal{A}_3$ is shown in Figure 4.13 in time-state diagram.

4.3.2 Discussion

The reservoir in Figure 4.13 is interesting in several ways. The individual is both able to self-regulate and adapt properly to perturbations. We can see that without perturbations it will after approximately 30 iterations settle into a stable state. This is after a

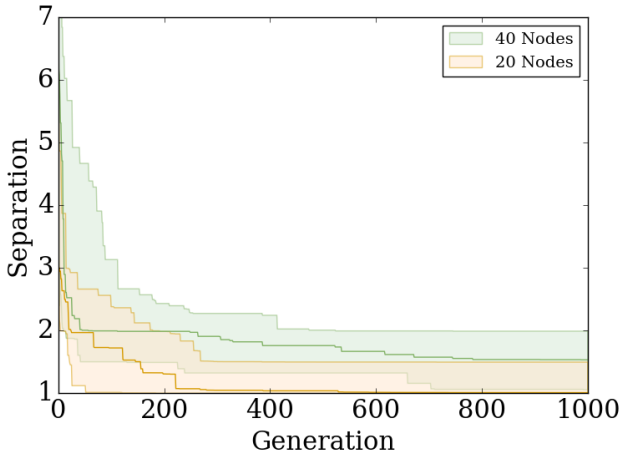


Figure 4.7: Separation fitness over 1000 generations, with a genotype size of 20 and 40. Evolution is run 25 times on each size, and the median fitness value is plotted as the line, surrounded with a shaded area representing the 1st and 3rd quartile

brief period of self-modifications. When the perturbations start after 100 iterations, the reservoir undergoes a $\tilde{1}75$ iterations long transient period of self-modifications. After the self-modifications have ceased, the reservoir exhibit a dynamic behaviour, that enables the readout layer to classify the perturbations with 100% certainty.

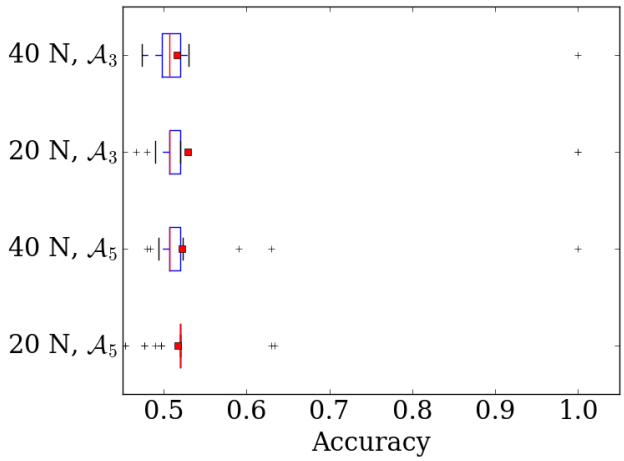


Figure 4.8: Accuracy of 25 runs when $N \in \{20, 40\}$ and Task is $\in \{\mathcal{A}_3, \mathcal{A}_5\}$

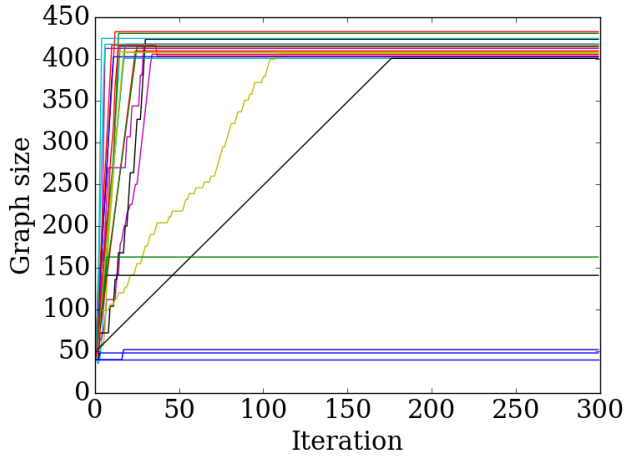


Figure 4.9: Graph development with genotype size 40.

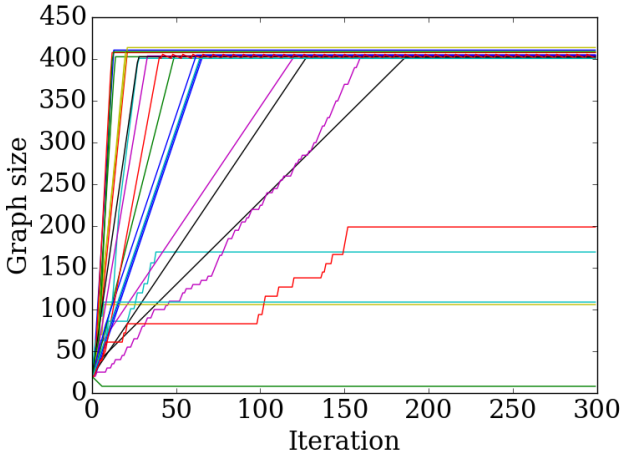


Figure 4.10: Graph development with genotype size 20.

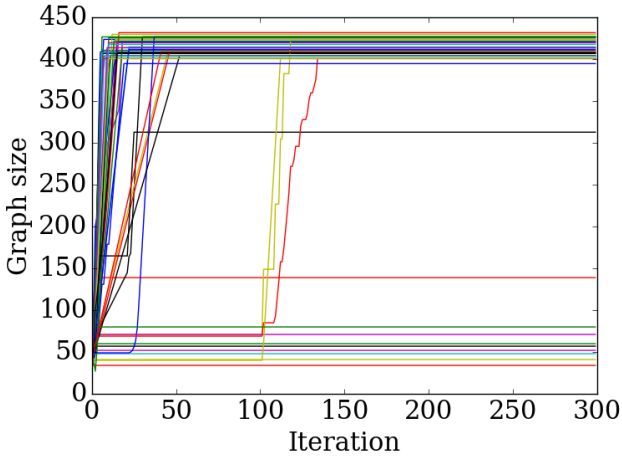


Figure 4.11: Graph size of 50 different chromosomes with the INDEX node, run for 300 iterations. There is a hard limit at 400 nodes, at which the graph is prevented from further growth. Input is 0.0 for the 100 first iterations, before the stream becomes random values $\in \{0.0, 1.0\}$.

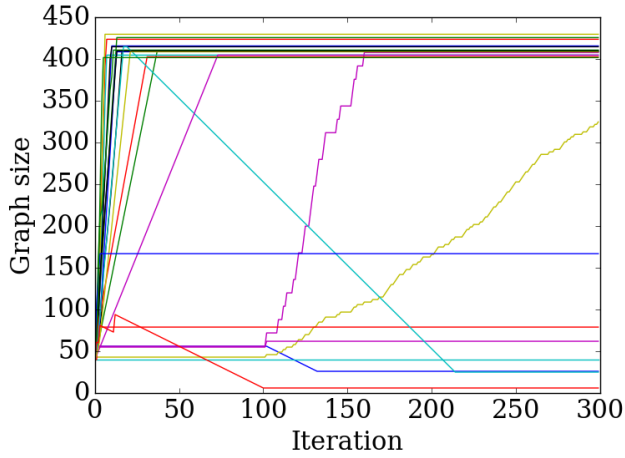


Figure 4.12: Graph size of 50 different chromosomes without any INDEX node, run for 300 iterations. There is a hard limit at 400 nodes, at which the graph is prevented from further growth. Input is 0.0 for the 100 first iterations, before the stream becomes random values $\in \{0.0, 1.0\}$.

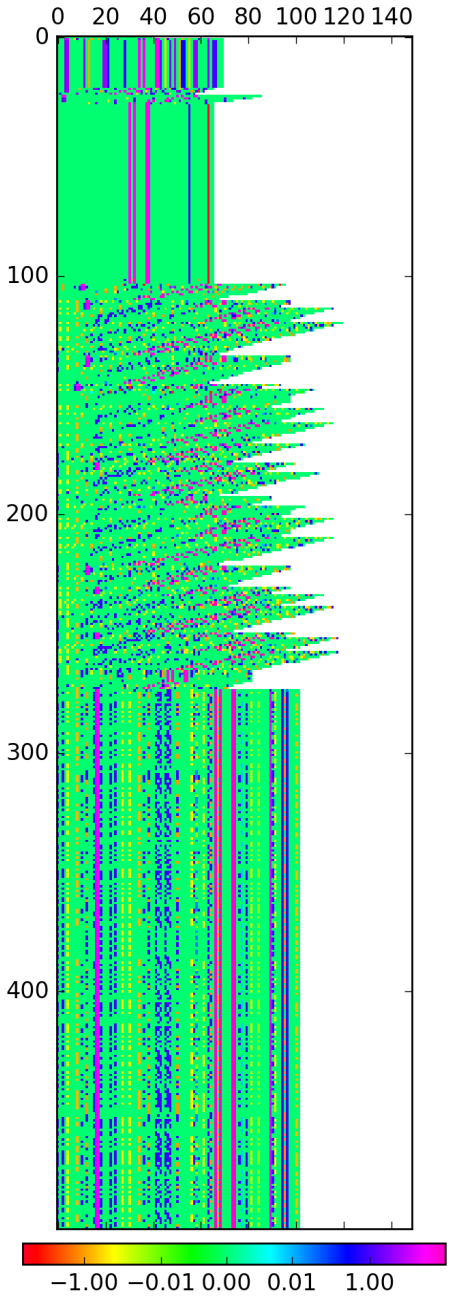


Figure 4.13: One of the first reservoirs to report 100% accuracy on $\mathcal{A}_3$. The reservoir is not perturbed before iteration 100. The x-axis indicates the node index, while time flows downwards. The node values are mapped to colours on a logarithmic scale.

Chapter 5

Conclusion

The proposed system was able to find some successful individuals, which stands as a proof of concept. These individuals started from a small genotype, e.g. 20 or 40 nodes, and could develop into larger graphs exhibiting the dynamical properties that is required to function as a reservoir. The genotype responsible for the dynamics shown in Figure 4.13 is a prime example of what we were trying to achieve in this thesis.

However, the introduction of development through self-modification to the RCGP-RC-system did not result in increased performance in finding suitable reservoirs. Evolving a genotype that develops a computationally capable reservoir is, not surprisingly, much harder than to evolve a computationally capable reservoir directly. Difficulties of evolving developmental encodings for complex tasks has been reported by Harding and Miller (2006). Further, Siddiqi and Lucas (1998) reports that developmental encodings do not perform better than direct ones.

The majority of the solutions failed to self-organise in a way that limited the size of the reservoir. The inability to do so renders the reservoir useless as the computer will eventually run out of memory or computational power required to evaluate the graph. A hard limit on the graph size was implemented to prevent this, but it breaks the self-organisational property of the system.

A more precise developmental fitness function might have aided the evolutionary search in finding reservoirs with better ability to self-organise and adapt. Unfortunately, the majority of the work went into developing a functional extension of the CGP-Library, therefore little time was left to explore developmental fitness functions.

The extension of the CGP-Library with self-modifying-properties included a significant performance penalty. The exact reason was not found, but a probable cause is that during graph development, the data structures for chromosomes and nodes are frequently initialised and freed. Additionally, the code for training the readout layer was not parallelised, which made an evolutionary search where the fitness function included the accuracy measure unbearably slow.

Nevertheless, the inclusion of self-modifications in a reservoir to make it adaptable is an interesting idea, however further investigation requires a better implementation of

a self-modifying structure, with the ability to perform computations. Additionally, more refined fitness functions which faster converge on solutions, should be explored. With the current implementation, the RCGP system presented in the specialisation project is a faster way of acquiring computationally capable reservoirs.

As a bouns, Appendix C contains a couple of phenotypes with interesting visual developments.

5.1 Answers to Research Questions

This section answers the research questions asked in Section 1.2.

Research Question 1: Is it possible to use a Self-Modifying Recurrent Cartesian Genetic Programming-graph as a reservoir?

Several chromosomes were successful in solving the temporal density problem. This shows that it is possible to use a SMRCGP-graph as a reservoir, however, suitable reservoirs are hard to find.

Research Question 2: How does the genotype size affect the ability to develop into a functional reservoir?

A reduced genotype size resulted is a slower growth rate, however, the effect on the ability to solve the temporal parity problem was insignificant.

Research Question 3: Will the self-modifying properties allow the graph to adapt to new environments, i.e. perturbations?

Again, we found examples that confirm the possibility, yet no effective way of evolving such behaviour was found.

The goal of this thesis was to explore the possibility of evolving a self-modifying reservoir for use in an RC-system. The results show that it is possible, although the reservoir is much harder to evolve, compared to the non-developmental variant. The implemented system works as a proof-of-concept, but needs further research before the model can adapted for broader use.

5.2 Future Work

The extension of the CGP-Library contains bugs and performance issues. These bugs should be ruled out or maybe an alternative execution model could be tested. A suggestion is separating the self-modification from the execution graph.

Only small set of possible SM-nodes were implemented. Other SM-nodes might have an interesting impact on the SMRCGP graph development and adaptability. Harding et al. (2007) suggest a larger set of SM-nodes. The addition of nodes similar to CHF (change function) and COPY-STOP (copy all nodes until the stop node) should be investigated. The SM_ADD node could be change to add a node of a certain function, to eliminate the stochasticity. This may also require a node to change the arguments of nodes.

Fitness functions to better evaluate the ability to develop and adapt could be explored. Implementing incremental fitness functions might also help the evolutionary search, as suggested by Nichele and Tufte (2013). Additionally, Harding et al. (2007) reported a performance increase with an incremental fitness function.

Bibliography

Bengio, Y., Simard, P., Frasconi, P., 1994. Learning Long-Term Dependencies with Gradient Descent is Difficult. IEEE Transactions on Neural Networks 5 (2), 157–166.

Bentley, P., Kumar, S., 2003. On Growth, Form and Computers. Academic Press, Amsterdam.
URL http://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=195521&site=eds-live

Beyer, H.-G., Schwefel, H.-P., 2002. Evolution strategies – A comprehensive introduction. Natural Computing 1 (1), 3 – 52.

Bhattacharya, R., Majumdar, M., 2004. Random dynamical systems: A review. Economic Theory 23 (1), 13–38.

Bishop, C. M., 2013. Pattern Recognition and Machine Learning. Vol. 53.

Cox, D. R., 1958. The Regression Analysis of Binary Sequences. Journal of the Royal Statistical Society 20 (2), 215–242.

Darwin, C., 1859. On the Origin of Species. Vol. 146.

Doursat, R., Sayama, H., Michel, O., 2013. A review of morphogenetic engineering. Natural Computing 12 (4), 517–535.

Droettboom, M., Caswell, T. A., Hunter, J., Firing, E., Nielsen, J. H., Varoquaux, N., Root, B., Elson, P., Dale, D., Lee, J.-J., Andrade, E. S. D., Seppänen, J. K., McDougall, D., May, R., Lee, A., Straw, A., Stansby, D., Hobson, P., Yu, T. S., Ma, E., Gohlke, C., Silvester, S., Moad, C., Schulz, J., Vincent, A. F., Würtz, P., Ariza, F., Cimarron, Hisch, T., Kniazev, N., 1 2017. Matplotlib/Matplotlib V2.0.1.

Duport, F., Smerieri, A., Akrout, A., Haelterman, M., Massar, S., 2016. Fully analogue photonic reservoir computer. Scientific Reports 6 (October 2015), 22381.

Fernando, C., Sojakka, S., 2003. Pattern Recognition in a Bucket. Advances in Artificial Life, 588–597.

Fogel, L. J., Owens, A. J., Walsh, M. J., 1966. Artificial Intelligence Through Simulated Evolution. John Wiley \& Sons.
URL https://books.google.no/books?id=QMLaAAAAMAAJ

Goldberg, D. E., 1989. Genetic Algorithms in Search, Optimization, and Machine Learning. Vol. Addison-We.

Harding, S., Miller, J. F., 2006. A comparison between developmental and direct encodings. Researchgate.Net.
URL http://www.researchgate.net/publication/238079542_An_update_of_the_GECCO_2006_Paper_The_Dead_State/file/9c960528b5ab35ba8a.pdf

Harding, S., Miller, J. F., Banzhaf, W., 2010. Developments in Cartesian Genetic Programming: Self-modifying CGP. Genetic Programming and Evolvable Machines 11 (3-4), 397–439.

Harding, S., Miller, J. F., Banzhaf, W., 2011. SMCGP2: Self Modifying Cartesian Genetic Programming in Two Dimensions. Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, 1491–1498.
URL http://doi.acm.org/10.1145/2001576.2001777

Harding, S. L., Miller, J. F., Banzhaf, W., 2007. Self-modifying cartesian genetic programming. In: GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation. Vol. 1. pp. 1021–1028.

Heylighen, F., 2001. The science of self-organization and adaptativity. The Encyclopedia of Life Support Systems, 1–26.

Holland, J. H., 1975. Adaptation in Natural and Artificial Systems. Vol. Ann Arbor.
URL http://www.citeulike.org/group/664/article/400721

Jaeger, H., 2010. The " echo state " approach to analysing and training recurrent neural networks – with an Erratum note 1. GMD Report (148), 1–47.

Jaeger, H., Haas, H., 2004. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. science 304 (5667), 78–80.

Kauffman, S., 1969. Metabolic stability and epigenesis in randomly constructed genetic nets. Journal of Theoretical Biology 22 (3), 437–467.
URL http://linkinghub.elsevier.com/retrieve/pii/0022519369900150

Kowaliw, T., Grogono, P., Kharma, N., 2007. Environment as a spatial constraint on the growth of structural form. Proceedings of the 9th annual conference on Genetic and evolutionary computation GECCO 07 2, 1037.
URL http://portal.acm.org/citation.cfm?doid=1276958.1277163

Koza, J. R., 1992. Genetic Programming: On the Programming of Computers by Means of Natural Selection. Vol. 1. MIT press.

Koza, J. R., 1994. Genetic programming as a means for programming computers by natural selection. Statistics and Computing 4 (2), 87–112.
URL http://dx.doi.org/10.1007/BF00175355

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

Le Cun, Y., Jackel, L., Boser, B., Denker, J., Graf, H., Guyon, I., Henderson, D., Howard, R., Hubbard, W., 1989. Handwritten digit recognition: applications of neural network chips and automatic learning. IEEE Communications Magazine 27 (11), 41–46.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=41400

Lima, A., 2016. Reservoir Computing using Recurrent Cartesian Genetic Programming. Tech. rep., Institute of Computer Science and Informatics, Norwegian University of Science and Technology, Trondheim.
URL https://folk.ntnu.no/anderlim/SpecializationProject_AndersLima.pdf

Lindenmayer, A., 1968. Mathematical models for cellular interaction in development, Parts I and II. Journal of Theoretical Biology 18, 280–315.

Lizier, J. T., Prokopenko, M., Zomaya, A. Y., 2014. A Framework for the Local Information Dynamics of Distributed Computation in Complex Systems. In: Prokopenko, M. (Ed.), Guided Self-Organization: Inception. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 115–158.

Lorenz, E. N., 1963. Deterministic Nonperiodic Flow.
URL http://journals.ametsoc.org/doi/abs/10.1175/1520-0469%281963%29020%3C0130%3ADNF%3E2.0.CO%3B2

Lukoševičius, M., Jaeger, H., Schrauwen, B., 2012. Reservoir Computing Trends. KI - Künstliche Intelligenz, 365–371.

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

Michalewicz, Z., 1996. Genetic Algorithms + Data Structures = Evolution Programs.
URL http://link.springer.com/10.1007/978-3-662-03315-9

Miller, J. F., 2004. Evolving a Self-Repairing, Self-Regulating, French Flag Organism. Genetic and Evolutionary Computation – GECCO 2004, 129 – 139.
URL http://www.springerlink.com/content/u7ex2reycalmju46

Miller, J. F., Smith, S. L., 2006. Redundancy and computational efficiency in cartesian genetic programming. IEEE Transactions on Evolutionary Computation 10 (2), 167–174.

Miller, J. F., Thomson, P., 2000. Cartesian Genetic Programming. In: Poli, R., Banzhaf, W., Langdon, W. B., Miller, J., Nordin, P., Fogarty, T. C. (Eds.), Genetic Programming: European Conference, EuroGP 2000, Edinburgh, Scotland, UK, April 15-16, 2000. Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 121–132.
URL http://dx.doi.org/10.1007/978-3-540-46239-2_9

Milnor, J., 1985. On the concept of attractor. Communications in Mathematical Physics 99 (2), 177–195.

Mitchell, M., 1998. An Introduction to Genetic Algorithms. MIT Press, Cambridge, MA, USA.

Natschläger, T., Bertschinger, N., Legenstein, R., 2005. At the edge of chaos: Real-time computations and self-organized criticality in recurrent neural networks. In: Advances in Neural Information Processing Systems. Vol. 17. pp. 145–152.
URL http://books.nips.cc/papers/files/nips17/NIPS2004_0380.pdf`

Neumann, J. v., Burks, A. W., 1966. Theory of Self-Reproducing Automata.

Nichele, S., Tufte, G., 2013. Evolution of Incremental Complex Behavior on Cellular Machines. Ecal 2013, 63–70.

Nikolić, D., Häusler, S., Singer, W., Maass, W., Nikoli, D., Haeusler, S., 2007. Temporal dynamics of information content carried by neurons in the primary visual cortex. Theoretical Computer Science 20, 1041–1048.

Pacanu, R., Mikolov, T., Bengio, Y., 2013. On the Difficulties of Training Recurrent Neural Networks. Icml (2).

Packard, N., 1988. Adaption towards the edge of chaos. Dynamic patterns in complex systems (Singapore:World Scientific), 293–301.

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

Rechenberg, I., 1973. Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Problemata, 15. Frommann-Holzboog.
URL https://books.google.no/books?id=-WAQAQAAMAAJ

Sayama, 2013. Modeling Complex systems. Vol. 53.

Sayama, H., 2010. Complex systems organizational map.
URL https://commons.wikimedia.org/wiki/File:Complex_systems_organizational_map.jpg

Schrauwen, B., Verstraeten, D., Van Campenhout, J., 2007. An overview of reservoir computing: theory, applications and implementations. Proceedings of the 15th European Symposium on Artificial Neural Networks (April), 471–82.

Siddiqi, A., Lucas, S., 1998. A comparison of matrix rewriting versus direct encoding for evolving neural networks. 1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360), 392–397.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=699787

Simon, H. A., 1965. The architecture of complexity. General systems 10 (1965), 63–76.

Snyder, D., Goudarzi, A., Teuscher, C., 2012. Finding Optimal Random Boolean Networks for Reservoir Computing. Artificial Life, 259–266.

Snyder, D., Goudarzi, A., Teuscher, C., 2013. Computational capabilities of random automata networks for reservoir computing. Physical Review E - Statistical, Nonlinear, and Soft Matter Physics 87 (4), 1–8.

Stanley, K. O., 2002. A Taxonomy for Artificial Embryogeny. Evolutionary computation 10 (2), 99–127.
URL http://www.ncbi.nlm.nih.gov/pubmed/12180173

Tufte, G., 2008. Evolution, Development and Environment Toward Adaptation through Phenotypic Plasticity and Exploitation of External Information. Artificial life XI The Eleventh International Conference on the Simulation and Synthesis of Living Systems 11, 624–631.
URL http://mitpress.mit.edu/books/chapters/0262287196chap81.pdf

Turing, A. M., 1952. The Chemical Basis of Morphogenenis. Sciences 237 (641), 37–72.
URL http://www.cecm.usp.br/~cewinter/aulas/artigos/2009/Turing_1952.pdf

Turner, A. J., Miller, J. F., 2014. Recurrent Cartesian Genetic Programming. Parallel Problem Solving from Nature – PPSN XIII 8672, 476–486.

Turner, A. J., Miller, J. F., 2015. Introducing a cross platform open source Cartesian Genetic Programming library. Genetic Programming and Evolvable Machines 16 (1), 83–91.

Turner, A. J., Miller, J. F., 2016. Recurrent Cartesian Genetic Programming of Artificial Neural Networks. Genetic Programming and Evolvable Machines 18 (2), 1–28.

von Neumann, J., 1945. First Draft of a Report on the EDAVAC. American Mathematical Society 15 (1), 1–10.

Appendix A

Data Structures

Listing A.1: Node Struct

```
struct node {
    int function;
    int *inputs;
    double *weights;
    int active;
    double output;
    int maxArity;
    int actArity;
    int arguments[NUM_ARGUMENTS];
};
```

Listing A.2: FunctionSet Struct

```
struct functionSet {
    int numFunctions;
    char functionNames[FUNCTIONSETSIZE][FUNCTIONNAMELENGTH];
    int maxNumInputs[FUNCTIONSETSIZE];
    node_type type[FUNCTIONSETSIZE];
    double (*infoFunctions[FUNCTIONSETSIZE])(struct chromosome* chromo,
        int index, double info_weight);
    struct chromosome* (*smFunctions[FUNCTIONSETSIZE])(struct chromosome *
        chromo_in, int index, int sm_args[NUM_ARGUMENTS]);
    double (*functions[FUNCTIONSETSIZE])(const int numInputs, const double
         *inputs, const double *connectionWeights);
};
```

Listing A.3: Chromosome Struct

```
struct chromosome {
    int numInputs;
    int numOutputs;
    int numNodes;
    int numActiveNodes;
    int arity;
    struct node **nodes;
    int *outputNodes;
    int *activeNodes;
    int *smNodes;
    double fitness;
    double *outputValues;
    struct functionSet *funcSet;
    double *nodeInputsHold;
    int generation;
    int numToDos;
    struct ToDo todoList[TODO_LENGTH];
    double recurrentConnectionProbability;
};
```

Listing A.4: Todo Struct

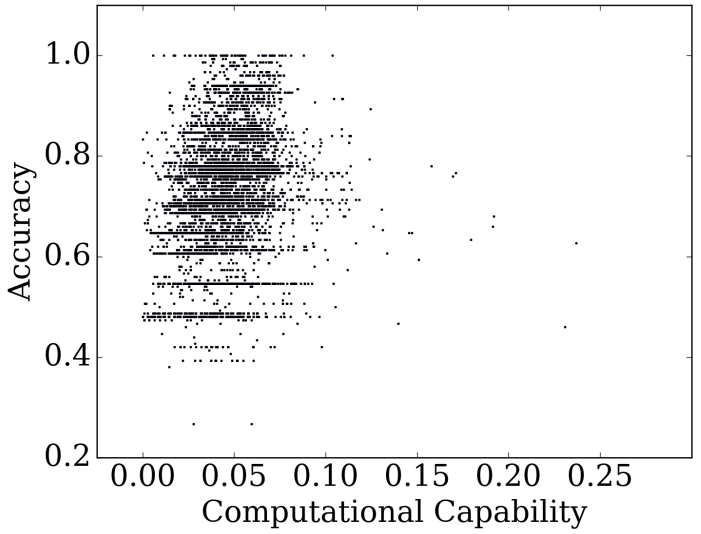
```
struct ToDo {
    int position;
    int arguments[NUM_ARGUMENTS];
    int function;
};
```

Appendix B

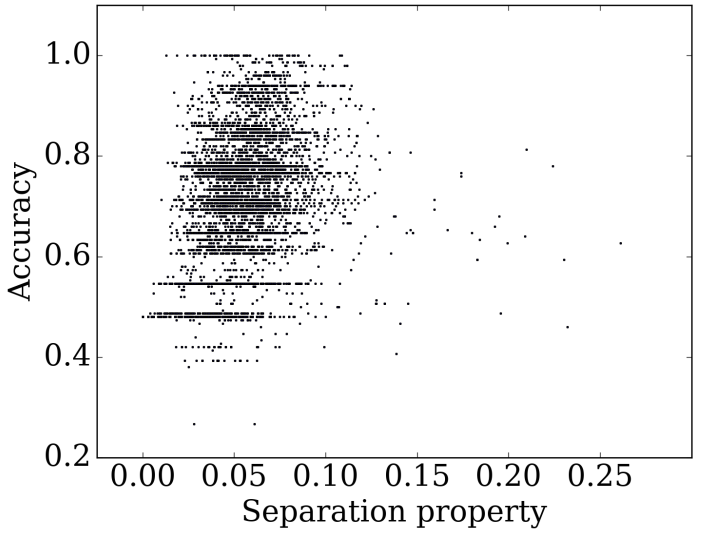
Specialisation Project

Table B.1: RCGP Hyperparameters used in (Lima, 2016)

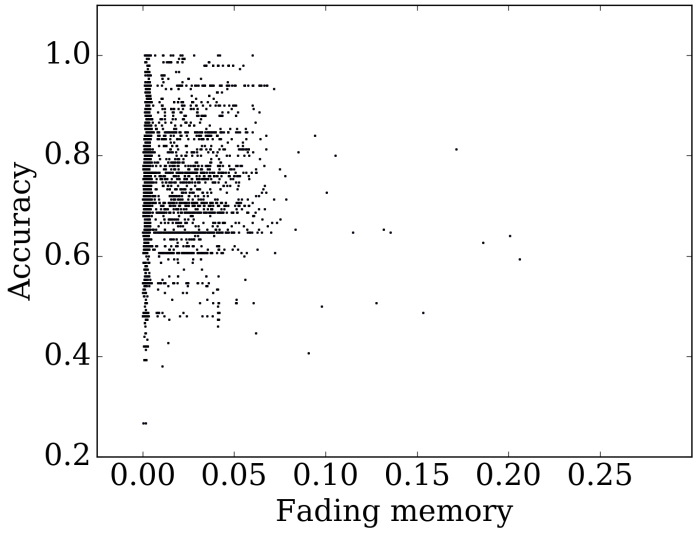
Evolutionary Strategy	(1+4)-ES
Inputs	1
Nodes	50
Outputs	1
Node Arity	2
Mutation Type	probabilistic
Mutation Rate	0.050000
Recurrent Connection Probability	0.050000
Shortcut Connections	1
Fitness Function	temporalParityAccuracy
Target Fitness	0.000000
Selection Scheme	selectFittest
Reproduction Scheme	mutateRandomParent
Number of Generations	1000
Function Set	and or xor (3)


Accuracy
1.0
0.8
0.6
0.4
0.2
0.00
0.05
0.10
0.15
0.20
0.25
Computational Capability

(a)

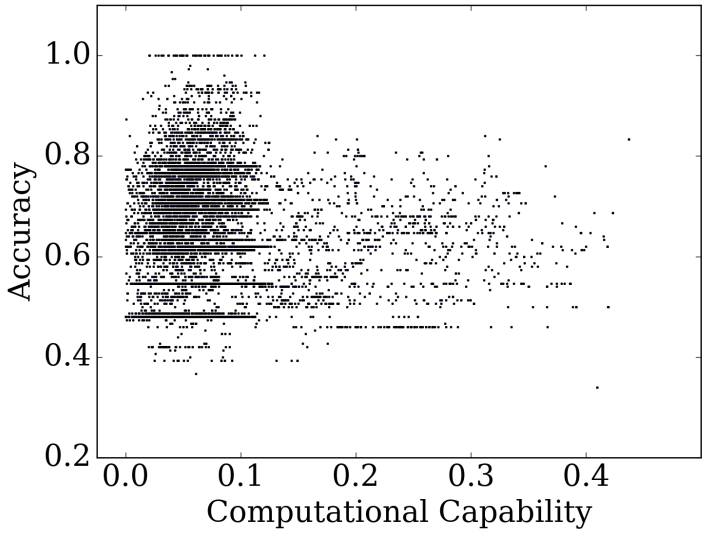

Accuracy
1.0
0.8
0.6
0.4
0.2
0.00
0.05
0.10
0.15
0.20
0.25
Separation property

(b)

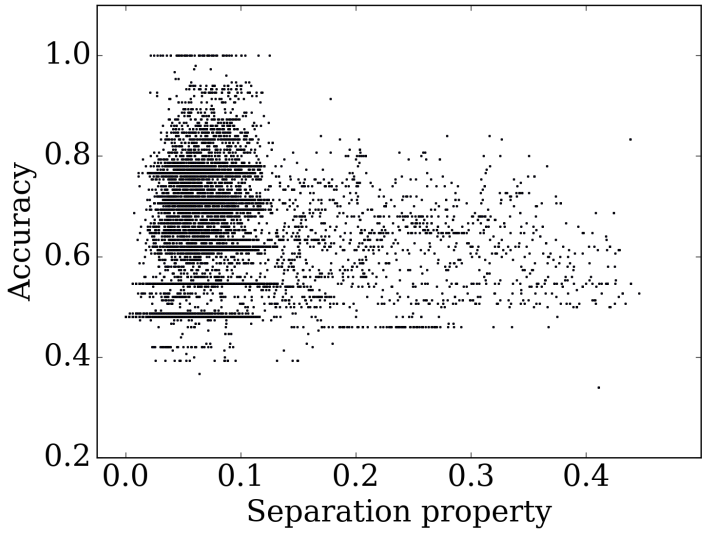


(c)

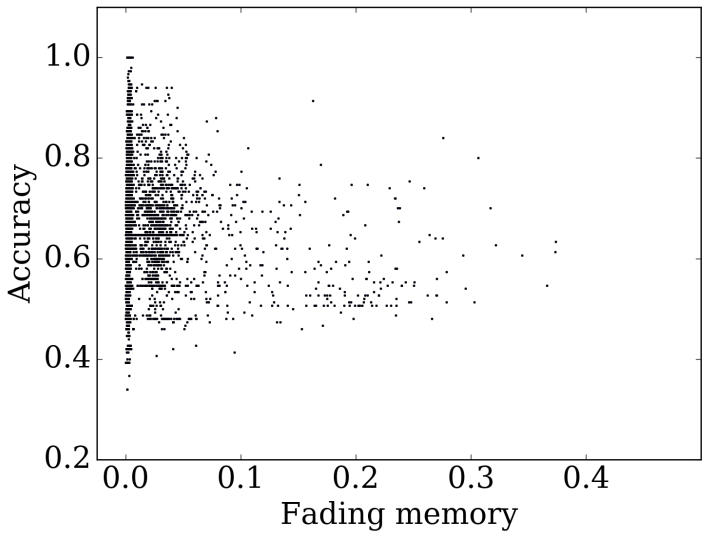
Figure B.1: The Accuracy plotted against Computational Capability (a), Separation property (b) and Fading memory (c), for each individual in every generation for 50 runs. The fitness was based only on accuracy on the temporal density problem.


Accuracy
1.0
0.8
0.6
0.4
0.2
0.0
0.1
0.2
0.3
0.4
Computational Capability

(a)


Accuracy
1.0
0.8
0.6
0.4
0.2
0.0
0.1
0.2
0.3
0.4
Separation property

(b)



(c)

Figure B.2: The Accuracy plotted against Computational Capability (a), Separation property (b) and Fading memory (c), for each individual in every generation for 50 runs. The fitness was based only on Computational Capability, but 100% accuracy on the temporal density problem would terminate the search.

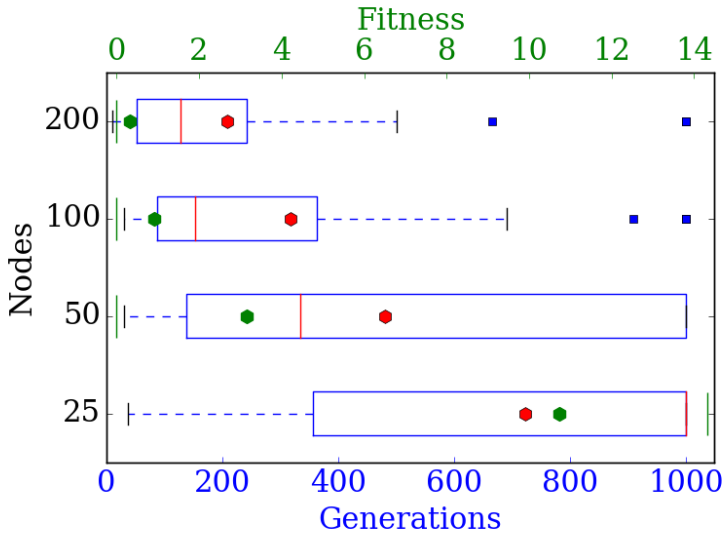


Figure B.3: Number of generations and fitness achieved over 50 runs, with $N \in [25, 50, 100, 200]$. Evolution is capped at 1000 generations. The fitness is plotted as $100 - accuracy \times 100$, and green vertical line is median, and hexagon is mean fitness.

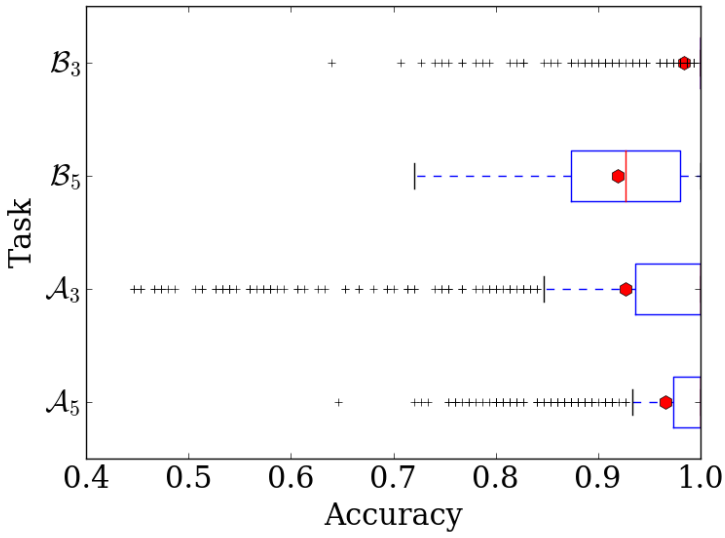


Figure B.4: Accuracy when training the readout layer on a given task $\in \{\mathcal{A}_5, \mathcal{A}_3, \mathcal{B}_5, \mathcal{B}_3\}$, when the reservoir is evolved to solve $\mathcal{A}_5$. 100 randomly generated streams of length 50 are fed to the system and the resulting $accuracy$ is plotted.

Appendix C

Bonus Phenotypes

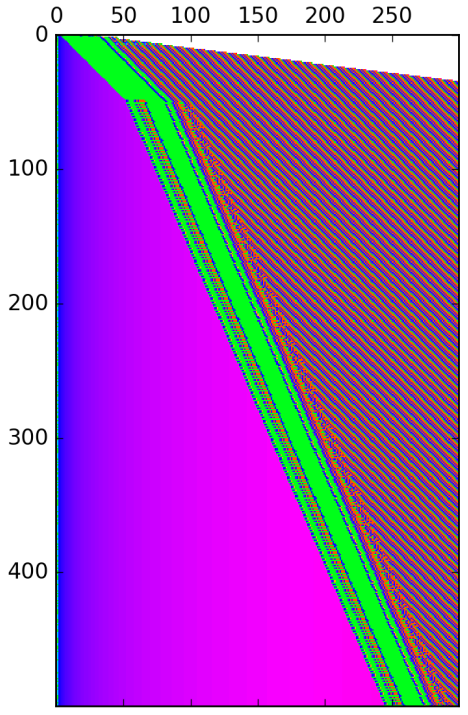


Figure C.1: Bonus phenotype 1.

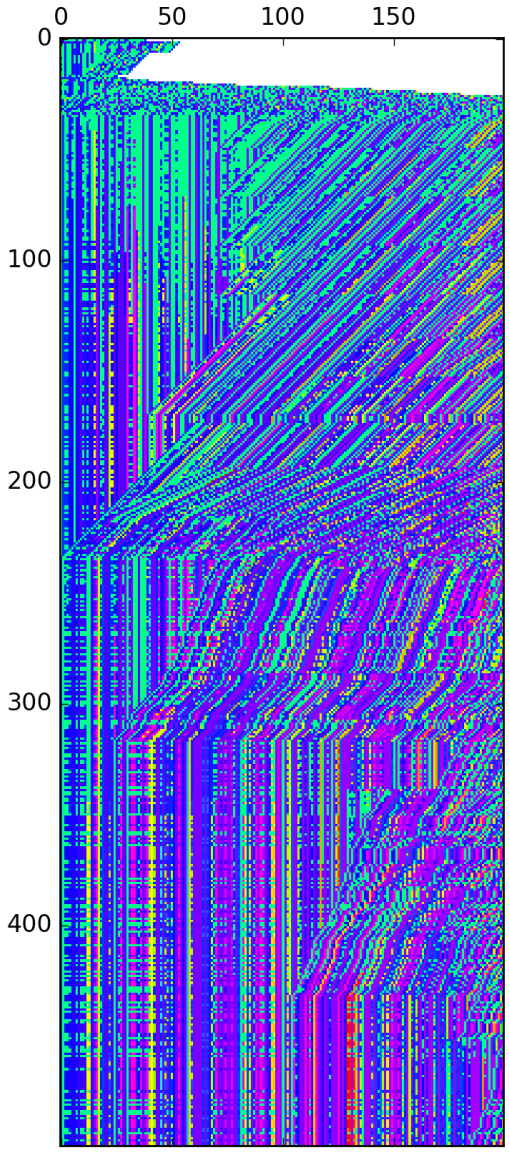


Figure C.2: Bonus phenotype 2.