Exploring elementary cellular automata rules as a reservoir for solving reinforcement learning tasks

Aksel Susegg

Thesis submitted for the degree of
Master in Applied Computer and Information Technology (ACIT)
30 credits

Department of Computer Science
Faculty of Technology, Art and Design

OSLO METROPOLITAN UNIVERSITY

Spring 2023

Exploring elementary cellular automata rules as a reservoir for solving reinforcement learning tasks

Aksel Susegg

© 2023 Aksel Susegg

Exploring elementary cellular automata rules as a reservoir for solving reinforcement learning tasks

http://www.oslomet.no/

Printed: Oslo Metropolitan University

Abstract

This thesis presents a novel approach for solving reinforcement learning tasks using elementary cellular automata (ECA) based reservoir computing (RC). Combining theories from these three fields to create a baseline for further investigation. The main objective is to investigate the unique ECA rules and their properties, and test them in different reinforcement learning (RL) environments. ECA based RC have previously been tested on the x-bit memory benchmark, where it has shown capabilities of long short-term memory. For many RL tasks, memory has been shown to be a vital part for good performance. This, together with the reservoirs computationally efficient methods of learning, makes this a quick method for training high performing models. In the experiments performed, it is shown that variations in the reservoir size, number of iterations, number of cells updated and how they are updated can have a huge impact on performance when certain rules are used.

Acknowledgments

I would like to thank my supervisor Tom Glover and co-supervisor Stefano Nichele for an exiting project. It has been inspiring to work with you two.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

Over the last couple of years, a large variety of huge and powerful AI models have been published. Many of these models shows a capability to solve complex tasks like we have not seen before. Midjourney, DALL-E 2 and Stable Diffusion took the public by storm when their image generation quality became that of a professional artist, sparking many discussions surrounding the use of these models. The latest model to gain a huge interest within the public domain, is ChatGPT. It is based on GPT-3 which is a language model that have been trained on a large percentage of text on the internet. It was then tuned towards a chat bot by training on conversations. After its launch, it quickly gathered a huge user base. In fact, some say the high interest made it the fastest growing consumer application ever. The popularity of these models only serves as a testament to the reality of AI being integrated more and more into our daily lives, and we are likely to see ever more advanced AI's in the future.

There is however a cost when it comes to these large models. Firstly, is the training process. The cost can reach many millions of dollars in electricity alone. When factoring in the cost of servers for training and storage, GPU's for training, gathering of training data and so on, the cost can reach tens of millions. Secondly is the cost of using and maintaining these models. These numbers can reach as high as hundreds of thousands of dollars every day. Lastly, there is a discussion to be made of the environmental impact such large models have when so much electricity is used for training and maintain them. These problems might not be as profound at this very moment and can be seen as manageable, but will drastically be more profound in the future as models grow larger and more advanced. The number of trainable neurons is expected to exponentially rise, and the training time and cost alongside it.

To overcome these problems, there is a huge incentive to find new computationally efficient methods of training these large models. Promising research in new methods of

performing calculations, mainly matrix multiplication, seems right around the corner. Examples of these are optical computers which uses photons instead of electrons, and variable resistor cells that can store decimal numbers instead of binary numbers. The advantage of the latter is that the number of memory accesses per matrix multiplication is greatly reduced, since the voltage running through sequentially placed resistor cells can be added together. If any of these methods worked, we would be looking at computation potentially being orders of magnitude faster and efficient than current computers. Unfortunately, none of them do yet, and is dependent on breakthroughs in their respective domains to become viable options. Both of these methods are hardware-based changes, but this thesis will cover an AI model where we are working with structural changes instead.

Researchers at OsloMet Living Technology Lab proposes that the use of Reservoir Computing (RC) can be such a structure that can greatly improve the training efficiency of AI models. With RC, all weights between neurons remain fixed, with only a single trainable output layer, which has the potential to greatly reduce the amount of training. Research surrounding the use of RC, shows promising results, especially considering that the field is relatively new. When RC first started to emerge, it acted as an answer to the challenging and computationally expensive training of recurrent neural networks (RNN). The substrate of these reservoirs would be an RNN with randomly connected nodes, but with fixed weights and a single trainable output layer. Whats more, is that the substrate of a reservoir is not bound to the digital world, but can be expanded to physical systems [17], like a bucket of water where the ripples on the surface can act as the reservoirs substrate.

In this thesis, we will explore one such substrate, but of a different kind. For a system to be able to function as a reservoir, it most contain certain properties. Firstly, the inputs to the system must be able to propagate throughout the reservoir. secondly, as information propagates, its values must not amplifies. Cellular automate (CA) is one such system with seemingly simple rules but with complex behaviour. The CA system we will be testing as a reservoir substrate, is the simple one-dimensional elementary cellular automata (ECA). Over the last couple of years, this ECA based RC (ReCA) have been tested against a benchmark known as the x-bit memory benchmark, where the goal is to reproduce a sequence of x-bits after a certain period have past. The objective of this thesis is to expand on this research, and test and benchmark the capabilities of ReCA in a reinforcement learning setting.

1.2 Problem Statement

Using elementary cellular automata as a reservoir substrate is a novel approach for solving reinforcement learning tasks. The problem statements are therefor exploratory in nature, and is defined as the following:

1. By using as little aid as possible, is the model capable of solving RL environments?
2. How does changes in the reservoirs parameters affect the performance?
3. What are favourable and detrimental properties of ECA rules with different reservoir setups?

Chapter 2

Theory

In this chapter, we will take a closer look into the background theory surrounding this thesis. It is divided into the three main sections covering the general topics of cellular automata, reinforcement learning and reservoir computing.

2.1 Cellular Automata

Cellular automata (CA) is a type of mathematical model that is used to simulate complex systems. First theorised in the 1940s by mathematician John von Neumann, it is composed of a grid of cells, usually of two dimensions but not limited to. Each cell can be in one of a finite number of states. These states are often referred to as on or off, one or zero, alive or dead, and so on. This makes up a two-state system, but cellular automata in general can have any number of states. The state of each cell is determined by the states of its neighbouring cells, according to a set of rules. An example of this is Conway's Game of Life [5]. This is the most popular two-dimensional two-state system where a cell is described as alive or dead. A cell is affected by its surrounding eight cells and the ruleset describing the system is as follows: 1) Any live cell with two or three live neighbours survives, 2) Any dead cell with three live neighbours becomes a live cell, 3) All other live cells die in the next generation. Similarly, all other dead cells stay dead. It is also important to note that all the states of a generation are updated simultaneously. With this simple ruleset, a variety of complex behaviour can emerge.

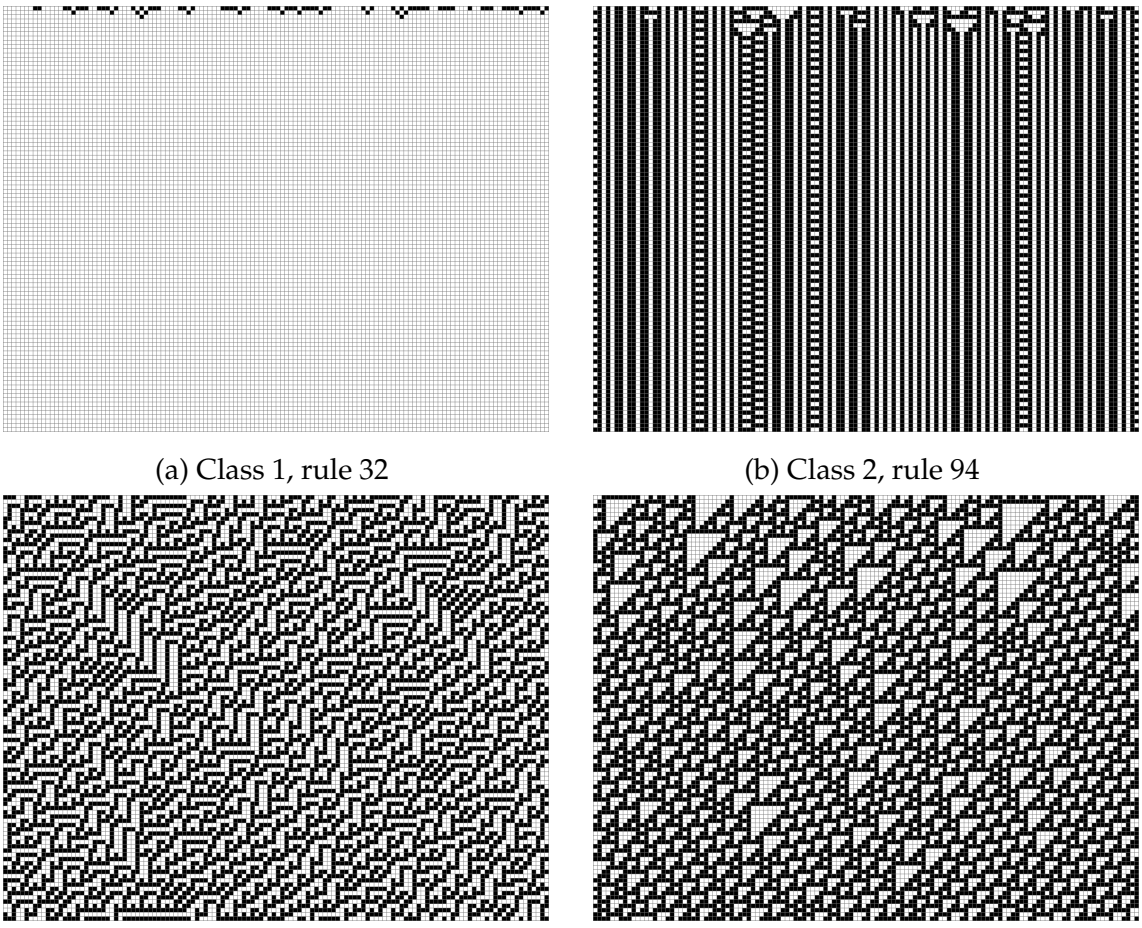
2.1.1 Elementary Cellular Automata

Table 2.1: Input and output of rule 82

Input	111	110	101	100	011	010	001	000
Output	0	1	0	1	0	0	1	0

Elementary cellular automata (ECA) is a specific type of CA. It is defined as a one-dimensional two-state system where the rules determine the state of a cell based on the state of the neighbouring left and right cell. It is regarded as the simplest form of CA, with a ruleset consisting of 256 total rules, with names ranging from 0 to 255. When updating the state of a cell, three input cells are used to determine the next state. Three inputs of binary numbers have eight total permutations where the name of an ECA rule corresponds to the output in binary. For example, rule 82 in binary is: 01010010, describing the output of the 8 different 3-bit inputs, which can be seen in Table 2.1. ECAs are often used to demonstrate the complex behaviours that can arise from simple rules.

2.1.2 Classification



(a) Class 1, rule 32

(b) Class 2, rule 94

(c) Class 3, rule 45

(d) Class 4, rule 110

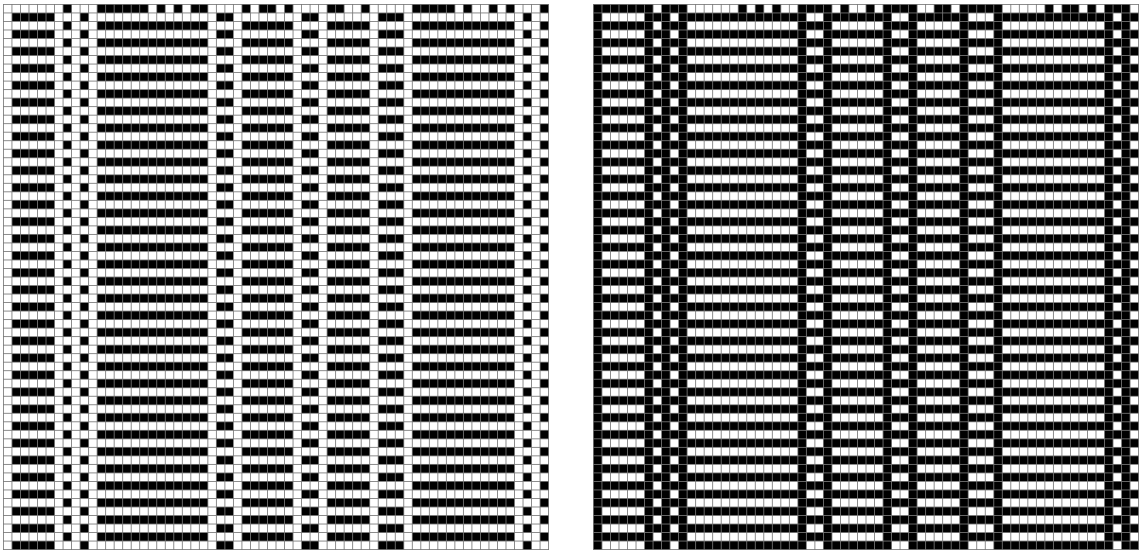
Figure 2.1: Examples of elementary cellular automata classes randomly initialised.

It has been generally accepted that CAs can be classified into 4 categories. They were first proposed by Stephen Wolfram in 1984 and were for the most part only covering

ECA rules [18]. Since then, adjustments to these classifications have been modified to fit CAs in more general terms. The classifications main objective is to tell how a system will evolve over time given a random initial state. Class 1 is defined as any rule in which the system evolves into a uniform state. A clear example of this, would be rule 0. Any output from this rule is 0, and any subsequent generations will therefore only exist of 0s. Another example of this, is rule 32 as seen in Figure 2.1a. Class 2 contain rules where the pattern tends to be still, or contains stable oscillations. These rules have a periodic behaviour. Class 3 is defined as the class where the rules does not settle into a particular pattern and is often described as chaotic. The last class, class 4, contains rules that exhibit both order and randomness and refer to rules with complex behaviour. It can be hard to distinguish between class 3 and 4, since many class 3 rules can have regions of patterns that seem to be stable. Examples of class 2, 3 and 4 can be seen in Figure 2.1b, c and d respectively.

Unique rules

All 256 rules does not have properties that uniquely separates them from each other. Many rules are either mirrored, complement or mirrored complement of another one. Table 2.2 shows which rules are unique with their corresponding equivalent rules. Out of the 256 rules, only 88 of them display unique properties. Figure 2.2 illustrates how rule 1 and 127 are complement to one another. Given an opposite starting configuration, rule 127 will evolve in a pattern that is exactly opposite to that of rule 1.



(a) Rule 1 (b) Rule 127

Figure 2.2: Examples of complementary rules.

Table 2.2: Unique rules and their equivalent counter rules.

Rule	Equivalent rules	Rule	Equivalent rules	Rule	Equivalent rules
0	255	35	49, 59, 115	108	201
1	127	36	219	110	124, 137, 193
2	16, 191, 247	37	91	122	161
3	17, 63, 119	38	52, 155, 211	126	129
4	223	40	96, 235, 249	128	254
5	95	41	97, 107, 121	130	144, 190, 246
6	20, 159, 215	42	112, 171, 241	132	222
7	21, 31, 87	43	113	134	148, 158, 214
8	64, 239, 253	44	100, 203, 217	136	192, 238, 252
9	65, 111, 125	45	75, 89, 101	138	174, 208, 244
10	80, 175, 245	46	116, 139, 209	140	196, 206, 220
11	47, 81, 117	50	179	142	212
12	68, 207, 221	51		146	182
13	69, 79, 93	54	147	150	
14	84, 143, 213	56	98, 185, 227	152	188, 194, 230
15	85	57	99	154	166, 180, 210
18	183	58	114, 163, 177	156	198
19	55	60	102, 153, 195	160	250
22	151	62	118, 131, 145	162	176, 186, 242
23		72	237	164	218
24	66, 189, 231	73	109	168	224, 234, 248
25	61, 67, 103	74	88, 173, 229	170	240
26	82, 167, 181	76	205	172	202, 216, 228
27	39, 53, 83	77		178	
28	70, 157, 199	78	92, 141, 197	184	226
29	71	90	165	200	236
30	86, 135, 149	94	133	204	
32	251	104	233	232	
33	123	105			
34	48, 187, 243	106	120, 169, 225		

Classifying the unique rules with respect to the classes presented by Wolfram, we can observe how they are distributed in Table 4.2. Class 4 with complex behaviour contains the least amount with only four rules, class 1 and 3 contains 8 and 11 rules respectively, but the majority of the rules falls under class 2 with periodic behaviour, containing 65 of them. Other classification methods used for the ECA rules, are e.g. by symmetric behaviours, categorised as symmetric, semi-asymmetric, and full-asymmetric. Or using ECA with memory (ECAM) to determine a rules capability of transforming from one class to another depending on a memory function, where they are classified into strong, moderate and weak. An in-depth study and comparison of different ECA classifications can be found in "A Note on Elementary Cellular Automata Classification" [11].

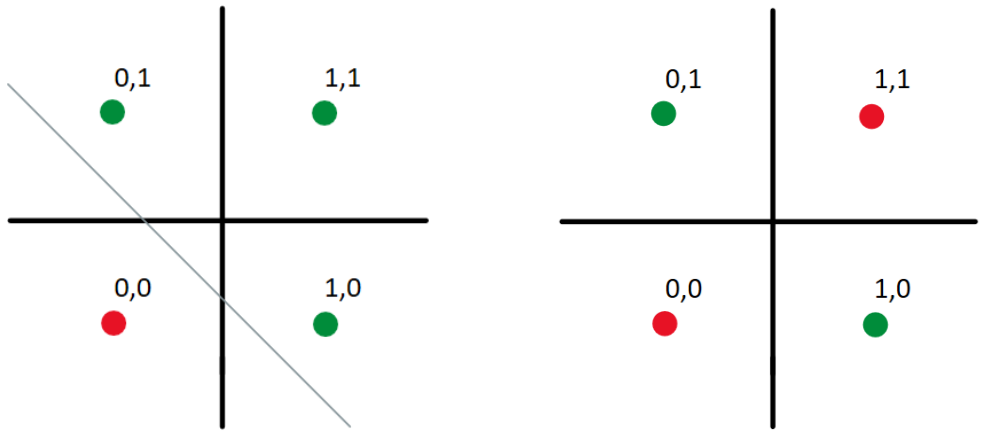
Table 2.3: Unique rules grouped by Wolfram's classification

Class	Rules
1	0, 8, 32, 40, 128, 136, 160, 168
2	1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 19, 23, 24, 25, 26, 27, 28, 29, 33, 34, 35, 36, 37, 38, 42, 43, 44, 46, 50, 51, 56, 57, 58, 62, 72, 73, 74, 76, 77, 78, 94, 104, 108, 130, 132, 134, 138, 140, 142, 152, 154, 156, 162, 164, 170, 172, 178, 184, 200, 204, 232
3	18, 22, 30, 45, 60, 90, 105, 122, 126, 146, 150
4	41, 54, 106, 110

2.1.3 Edge of chaos

At the border between orderly and chaotic, exists a transition space called the edge of chaos. This phrase is used to describe the line where a system crosses from being somewhat orderly to completely random. In CA, it was found that many systems that exhibited life like patters, or self-organisation, often exists close to that line. It has been observed that in many systems, for example nature, physics or economics, when modelled also operates in this region. There is a common consent that CA systems with good computational properties will probably operate within the edge of chaos [10].

2.2 Single and Multi layer perceptron



(a) OR (b) XOR

Figure 2.3: Illustration of the XOR problem.

The simplest form of artificial neural networks (ANN), is the single layer perceptron (SLP). It consists of a single layer of inputs and outputs with no hidden layers. The SLP is commonly known as a linear classifier due to the fact that it cannot solve nonlinear problems. To illustrate this, binary operations can be used. The OR operation, which will output 1 as long as one of the inputs is 1, can be linearly separated. This operation can therefor be modelled by an SLP. The XOR operation on the other hand, which will only output 1 if only one of the inputs are 1, cannot be linearly separated. The training will never reach a point where all the cases are separated. This is illustrated in Figure 2.5 and is known as the XOR-problem.

To solve the XOR-problem, we need a model capable of nonlinear classifications. The multi layer perceptron (MLP) expands on the SLP by introducing layers between the input and output layer. These hidden layers makes the MLP capable of solving nonlinear problems, but only if using nonlinear activation functions. If a MLP uses a linear activation function for all its neurons, the model can be reduced to a linear input-output model. A common nonlinear activation function is the Rectified linear unit (ReLU), where if given an input x will output x if x is greater than 0, else output 0, expressed as: $(x) = max(0, x)$.

2.3 Reinforcement learning

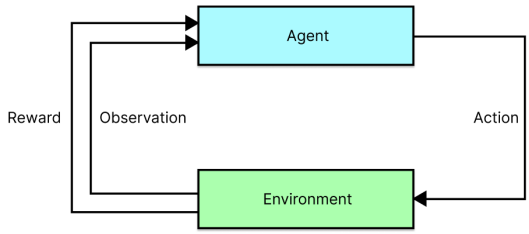


Figure 2.4: Illustration of the relation between agent and environment.

Reinforcement learning is one of the main subfileds of machine learning, the others being supervised and unsupervised learning. It differs from theses by not having any data, labeled or unlabelled to train on, but rather through interactions with an environment. This makes reinforcement learning a suitable candidate for solving complex decision-making tasks without having any previous knowledge of the environment. Reinforcement learning and deep reinforcement learning have been shown to excel in many domains, such as board games like Chess and Go, or playing video games [16]. To do this, an agent interacts with an environment by taking an action. The action can for example be to move in a certain direction, which will cause the environment to change. From the environment, the agent receives a state and a

reward. The state describes what the environment looks like, and is typically a RGB image or numerical representations of things in the environment or the agent itself. The reward is a numerical value which indicates if the action taken was good or bad given the previous state. Then, the agent performs a new action, and the cycle continues. Figure 2.4 shows a illustration of this cycle. Over time, the agent learns to perform actions, given a state, which maximises the cumulative future reward. This is called a policy, and is whats being updated when an agent is trained. In section 4.1 a more in depth explanation can be read about the action, reward and state representations of the different environments used in this thesis.

2.3.1 Exploration and Exploitation

In reinforcement learning, the principals of exploration and exploitation is important to understand. By exploration, we mean the process of exploring the environment to find which polices are good and which are bad. Exploitation is thereafter the process of exploiting the good polices. Balancing the two is the tricky part, and if not done correctly can lead to agents performing suboptimal actions. This is usually the result of doing to little exploration and moving over to exploitation to quickly. On the other hand, if exploration if perform for to long, a lot of time and computation might be wasted.

$$\epsilon = \epsilon * decay\_rate \tag{2.1}$$

A simple technique for managing the balance between exploration and exploitation is to use epsilon decay as seen in Function 2.1. Here, ϵ start with the value 1, and the decay_rate is close to 1 but smaller, e.g. 0,999. The value of ϵ can be updated every step, episode or epoch. During training, the agent will perform a random action with a probability equal to ϵ, and follow its optimal policy with a probability of $(1 - \epsilon)$. At the start of the training, when ϵ is close to 1, the agent will perform mostly random actions, exploring the environment. But as the training continues, the value of ϵ is slowly decreased, resulting in the agent following its optimal policy more closely. When training is done, ϵ is 0 and the agent will only perform actions it regards as best. This is a relatively naive solution to the exploration and exploitation problem, as the agent is still likely to become stuck within a local optimal. Other, more advanced functions might include more parameters to update the epsilon value, e.g. number of steps, reward and state. Other methods for encouraging exploration can be to give the agent an intrinsic bonus reward for finding states it has not seen before, or swapping out the epsilon equation completely with a separate network that is trained to decide if the agent should focus on exploration or exploitation [1].

2.3.2 Recurrent Neural Network

A big challenge in RL is for the agent to know which actions lead to rewards, both positive and negative. In many environments, especially in games, the agent might not be rewarded until hundreds of steps after a particular action have been taken. The opposite might also be the case. What set of actions led to a negative reward which comes much later. To combat this problem, recurrent neural networks (RNN) was introduced to give the agent some sort of memory. Unlike a feed-forward neural network, the RNNs have at least one connection which loops back to an earlier node in the network. One such method, which saw huge progress, was with the inclusion of Long short-term memory (LSTM), first introduced by [9].

Instead of stacking multiple observations to imitate history as inputs, the LSTM unit(s) could sit at the end of the network, combining previously extracted features with the current one. This made it possible for the agent to backtrack and train on history dependant values. Models like the Deep Recurrent Q-Network (DRQN) [8], and the actor-critic (A3C, LSTM) model introduced in [12], both utilised this technique to improve on the state-of-the-art average score in the Atari 57 game suite at their time of publishment.

2.4 Reservoir Computing

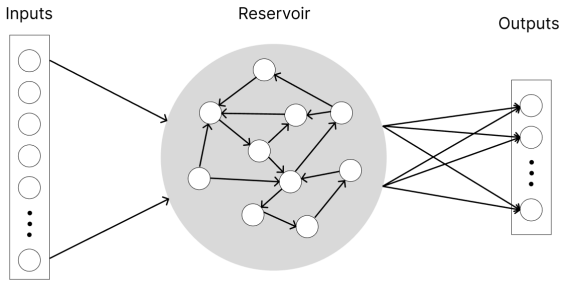


Figure 2.5: Illustration of the dynamics of a reservoir.

Reservoir computing further extends upon the concept of RNN, but unlike RNN, have randomly connected nodes. Inputs to the system is mapped to a higher dimension through the dynamics of the reservoir. The reservoir computer therefore consists of three parts; an input which maps an input to random or fixed locations within the reservoir, the reservoir itself which have randomly connected nodes, and the output which interpreters the state of the reservoir to make predictions. This makes reservoir computers an efficient network to train and has been shown to be more robust to noisy and missing data than other types of RNNs. Because of its recurrent nature, reservoir computers is a useful technique that can be applied to make sequential predictions.

2.4.1 Liquid state machines

One type of reservoir computer, is the liquid state machine (LSM). It has, for the most part, the same structure as the ordinary reservoir computer, except that is uses spiking neural networks (SNN). The reservoir will receive signals which varies in time, and the nodes themselves, which are randomly connected, will also receive time dependant signals from each other. This structure makes the LSM emit patterns which are highly dependant on space and time, much like ripples from a stone dropped in water, which is where the liquid part of the name comes from. Due to the recurrent nature of the LSM, it has been shown to process temporal data well, being able to extract features over time. This makes it good at tasks such as speech recognition and computer vision.

The LSM have also been used to model biological systems. One such system, is the dynamics of neurons in the brain. The brain is, of course, such a complex system that it can't be modelled with randomly connected neurons. But, where we find similarities is in the ability to perform multiple calculations, on various time scales, all within the same network. This makes LSMs a close resemblance to how the brain perform operations. But at the same time, makes them also difficult to study. Even with a relatively small reservoir of neurons, it can be hard to figure out where and what processes are being performed.

2.4.2 Echo state networks

The term reservoir computing is a generalisation used to encapsulate complex systems that utilises recursion. This includes network architectures such as RNN, LSM and echo state networks (ESN). ESN works much in the same way as RNN as updates happen in desecrate time steps, but instead of having trainable connections, all connections between nodes in the reservoir remain fixed. The only layer which is trained, is the output layer, also referred to as the readout layer. This is usually a linear dense layer which is trained to interpret the high dimensional state of the reservoir. With only one layer to train, ESNs are usually faster and not so computationally heavy to train as a RNN, where backpropagation is performed throughout the whole network. One property of the ESN, is that inputs to the network is imagined to being echoed throughout its nodes, with the inputs strength slowly fading away as new inputs are introduced, hence the name "echo".

2.4.3 Cellular automata as a reservoir computer

The first instance where cellular automata was used as a reservoir, can be found in the paper titled "Reservoir Computing using Cellular Automata" [19]. Here, the author discusses and demonstrate the use of such a system, highlighting the low

computational complexity compared to ESN using neurons, stating that it requires order of magnitude less computation. The paper illustrates the long short-term memory capabilities of this system by demonstrating it on the 5 and 20 bit memory task, using both Game of Life and elementary cellular automata as the reservoirs substrate.

Chapter 3

Related Work

In this section, we will take a closer look at previous works that this thesis builds upon. As of the release of this thesis, there has been little to none published articles using an ECA based RC model for solving RL problems. Therefor, the focus will be on research with similar structures or papers outlining the core principals in which this thesis is built upon.

3.1 The Dynamical Landscape of Reservoir Computing with Elementary Cellular Automata

The first paper we will look at, has not yet been published. It is called: "Investigating rules and parameters of Reservoir Computing with Elementary Cellular Automata, with a criticism of rule 90 and the 5-bit Memory Benchmark" [6] and works as a predecessor to this thesis. It is currently under review and is a continuation of the work presented in [7]. The paper's focus is on investigating the capabilities of ECA based RC (ReCA), and the properties of individual rules. Also, an extensive search of many parameters regarding the ReCA was tested to find how the properties of a rule would change. For the testing, the 5-bit memory benchmark was utilised, more commonly described as the x-bit memory benchmark. A benchmark in which a model is asked to remember a sequence of bits, and after a certain period, is given a cue to reproduce the same sequence. Thus, testing if a model is capable of memorising. Doing this in the context of ReCA, the input bits are placed into the CA grid using the XOR operation with the current state. The CA will then iterate over the grid, applying the rule to the state several steps before the next input is given. To produce an output, the system uses a classification model that can read the current state, optionally together with previous steps. This classification model is a linear model, meaning that there are no hidden layers, ensuring that the RC is doing the separation and not the neural network.

As mentioned, a variety of parameters regarding the benchmark and the CA was also tested. These include the number of regions the CA was divided into denoted as R, the width of R (Ld), CA iterations between inputs (I), number of inputs between the input and output sequence (distraction period Dp), and the number of bits to memorise (Nb). As mentioned in the paper, there exists many more parameters to explore, such as encoding and decoding strategies, but these were not investigated in this paper. In total, 3 experiments were done. These include, changing I, different length of Dp and seeing how small changes in Ld and Dp affected the performance. Changes in I showed that some rules benefited from I being an even number, like rule 3, 60 and 90, while others benefited from an odd number, like rule 15 and 56. Changes in Dp showed that many rules struggled with high values of Dp, especially rules that exhibit chaotic patterns. Here, the problem is compared to the fading memory problem. This experiment also showed that some rules do not have this problem, in fact, some showed better performance with greater Dp value. A subsequent experiment with a much higher Dp value of 4000 showed that rules like 162, 10, 105, 150 and 170 can settle into a periodic pattern such that changing Dp does not affect their performance. In the last experiment, small changes in Dp and Ld were tested. It showed that in some cases, like the first experiment, some rules drastically change performance with small changes to the parameters. Most notably scored rule 90 and 150 overall good with a Ld of 40, but 0 with Ld of 39 or 41. On the other hand, rule 54 generally scored higher with a Ld of 40 and above.

As this paper shows, the properties of the ECA rules can drastically change with small changes in the reservoir setup only. With a certain configuration, one rule can perform well and another poor. But with small changes, this can be flipped around. The researchers also criticises the x-bit memory benchmark and theorise that it might not be the best method for exploring the properties of the ECA rules used in an RC setting.

3.2 DQN

For this thesis, we will also need a way to train the model using RL. Due to the focus being on the ECA based RC, a simple method should suffice. Therefore, we will draw inspiration from the DQN techniques presented in the "Playing Atari with Deep Reinforcement Learning" [13] paper. This paper introduced a new way of training RL agents that could work in a majority of environments. Instead of using a state-action Q-table, which was a common practice up to this point, they trained a Deep Neural Network to approximate the Q-value given a state-action pair. This made it possible to train agents in environments with a huge variety in both states and actions, compared

to Q-tables which grew to an unreasonable size if the state or action space was too large. To demonstrate this, they trained and tested agents in different Atari games, and achieved scores comparable to humans in some of them.

They also showed how several techniques for more efficient and more stable training could be used in combination with deep reinforcement learning. This included having an experience replay memory, a separate target network and how to update the Q-value network based on the current state, action, reward, and the next state. The experience replay memory works by appending the current state, action, reward and next state to a long list. Every time the network is trained, a batch can be randomly sampled from this list and be used to train the network. This has two primary objectives. The first one is to have a consistently large batch of data to sample from, and the second one is to keep the agent from forgetting policies it has already learned. Without the experience replay memory, an agent can quickly get stuck in a suboptimal policy due to there being no variation in the training data. Imagine an environment with 2 actions, left and right. If the agent decides to mostly take left actions, and training is only done on the latest samples, the agent will eventually only step left as this is what it is being conditioned to do.

The experience replay memory becomes even more powerful when the agent explores the environment more. This is because the models will have a more diverse dataset to sample from. To achieve this, they used an epsilon greedy exploration policy. At every step throughout the training, there is a probability equal to this epsilon value to perform a random action. This epsilon value is usually high in the beginning and is slowly reduced over the course of the training. This makes the agent explore more in the beginning and exploit more towards the end. To achieve a more stable training, the researchers used two networks. We can call one the policy network, and the other for the target network. The policy network is the main network that is train and will do the decision making. But when training the policy network, the Q-values from the target network are used instead. The target network's parameter is also periodically synchronised with the policy network.

Over the years, many more complicated and sophisticated DQN based techniques have shown how effective this method can be. Many of the techniques that were included in this paper are still present in many of the modern RL models. In some models, they have been modified to be even better. Like in [15] where the sampling of the experience replay was modified to pick samples that had the potential to be more useful for training more often. Or in [1], where the researchers introduced what they called a meta-controller to help decide when the agent should prioritise exploration versus exploitation. But many of the principals remain the same.

3.3 Reinforcement Learning with Convolutional Reservoir Computing

This paper [3] presents a novel approach for performing RL tasks using convolutional neural networks (CNN) and reservoir computing (RC) named the RCRC model. This model is tested in two environments; *CarRacing-v0* and *DoomTakeCover-v0*, showing results that matched some of the best results at the time of publishing. In these environments, training is done directly on the RGB image frames. The model consists of three parts, a fixed randomly initialised CNN, a echo state network (ESN) with fixed random weights and connections, and a single trainable weight matrix. The CNN extracts and compresses features from the images which are passed to the ESN. The state of the ESN is then used together with the weight matrix to produce an action.

The researchers underline a couple of key characteristics of the RCRC model: 1) Because the weights of the CNN and ESN are random and remain fixed, the computational cost of the model is very low. 2) Only a single weight matrix needs to be trained.

Furthermore, the researchers highlights the ease of use of the model, comparing it to models which requires pre-training or computationally heavy training functions. Additionally remarking the models low training time and the wide variety of applications it can handle.

Chapter 4

Methodology

This chapter is divided into three main sections. The first one will describe the environments in which the model was tested. The second sections will go over how the model was implemented, firstly a description of the individual parts, and then how the whole training loop will work. This section will also contain descriptions of different design choices that were considered and why the ones used were chosen. In the last section, the different experiments are explained. This includes how they were conducted and what training parameters were used.

4.1 Environment

For the training environments used, two of varying difficulty were chosen. They are both from the openai Gym library [2], developed for testing RL related models. All Gym environments are built with the same structure, with two main functions, reset and step. The reset function, which as the name implies, will reset the environment to its starting position. The step function takes an action as an argument, applies it and returns the new state of the environment as an observation alongside a reward and if the environment was terminated or not. Every environment also has an action space and an observation shape. The action space describes what actions can be taken in the environment, while the observation shape describes how a single observation is represented. In addition to the observation space, observational low and high values are usually also defined. Together, these describe the range of values each observation is bound by.

4.1.1 Cart Pole

The objective of the Cart Pole environment is to balance a pole on top of a cart for as long as possible or up to a set number of steps. At every time step, the environment

requires an action. The action must be one of two discrete values, 0 or 1. If 0 is chosen, a force is applied to the cart in the left direction, and if 1 is chosen, the force will be towards the right. The observation contains 4 values, these being the carts position and velocity, and the poles angle and angle velocity. The angle of the pole is given in radians, and when the environment is reset, all values will be set to a randomly uniform distributed value between -0,05 and 0,05. The environment will terminate if one of two conditions are met, these being if the carts position is outside the range of -2,4 and 2,4 or if the pole angle is outside the range of -0,2095 and 0,2095 ($\pm 12^{\circ}$). An additional constraint can be set to terminate if the number of steps reaches beyond a certain number, with a default value of 500. When a step is taken, the reward will always return 1, even if the environment terminates. This means that for an agent to achieve a high total reward, it must be able to keep the cart close to the centre of the screen while having the pole in an upright position. Given that the environments starting state is somewhat randomised, a good performing agent should also be capable of quickly straightening up the pole.

4.1.2 Bipedal Walker

In the Bipedal Walker environment, you are tasked with controlling four joints of a robot. The four joints are controlled by applying a torque to a motor located in the hip and knee of two legs. These 4 joints takes a continuous value between -1 and 1 as inputs. The objective is for the robot to walk forward without falling over. Any forward movement is rewarded. If it falls over, and the head hits the ground, the agent is punished by a reward of -100. Applying torque does also result in a negative reward, albeit a small amount. A good performing agent in this environment, is one that is able to efficiently move forward. The observations returned from the environment consists of 24 values; hull angle speed, angular velocity, horizontal and vertical speed, position of joints, joints angular speed and if the legs are in contact with ground. 10 additional lidar measurements measuring the range to the ground at different angels are also included. The environment will terminate if one of three conditions are met: 1) a total reward of 300 is achieved, 2) after 1600 steps are taken, 3) the hull hits the ground. When the environment is reset, the walker will always start in the same position, with its two legs on the ground and the hull in a horizontal position. Some variation in terrain height will occur from episode to episode.

4.2 Setup

The complete model is separated into 3 main parts; the ECA, reservoir and agent. This section is dedicated to describing the setup of each of these individual parts. The last

section will cover the training loop and how these parts are combined into a complete model.

4.2.1 ECA

Out of the three parts, the ECA is the simplest with only 1 method. When initialised, it takes a valid ECA rule, either as an integer or a binary list, and maps the 8 possible 3-bit inputs to the rules defined output. The ECA-class will store both the rule in binary-list form, and the complete rule mappings. The method, which is called *iterate*, takes a binary list and a wrapping method as inputs. There are three valid wrapping methods implemented which are commonly used in ECA. These describe how the ECA will handle the edges of the CA. All three methods adds one additional value to the left and right-most position of the input list. This must be done to preserve the original total size of the ECA. If no padding is applied, and the first generation has a size of 100, the second generation will only have a size 98 and the next one 96 and so on. Valid paddings are *0*, *1* or *wrap*. *0* will add a 0 to the edges, *1* will add a 1 to the edges, and *wrap* will copy the values from the left and right-most position and add them to the opposite side, making the ECA appear to be continuous. After validating that both the input list is a valid binary list, and that a valid padding method is chosen, the rule is applied to the input list. Finally, the method returns the output list containing the new generation.

4.2.2 Reservoir

The reservoir part contain the main contribution towards working with ReCA based RL. There are multiple steps we need to implement for it to work as intended. The reservoir itself will only need to store 1's and 0's for each cell in a list referred to as cells. A list of the history of cell states for previous generations will also be needed. In general, we will need three methods, one for updating the current state of the reservoir, one for reading the state of the reservoir and one for resetting it. Lastly, we should also be able to save and load the reservoirs configurations.

Update

When updating the reservoir, we are only interested in changing the state of cells of the last generation. This function will receive the observations which are returned from the environment. Since the state of a single cell can only be 1 or 0, and the observations are generally floating point numbers, the first problem we face is; how can we map the floating point numbers to a binary format? When doing this, we must also keep in mind the trade-off between size, accuracy and interoperability. The observations

consists of 32 bits, but directly converting them to binary means we will have 64 binary number per observation. For a simple environment, like cart pole with only 4 observations, this might be doable, but for a more advanced one with many more observations, the reservoirs size can quickly grow out of proportions. Also, having that high of a precision for all the observations, will in most cases be way more than needed.

Casting is the next logical approach to take. First, the observations would be normalised, then scaled to a preferable precision, and lastly, only the integer part of the float would be cast to a binary number. To illustrate this using the cart pole environment and only one observation, the process would look like the following using the carts x-position = 1 and an accuracy of 8 bits: Normalise by dividing with the observations max value $obs\_norm = x/obs\_max(x) := 1/2.4 = 0.41\overline{6}$. Then scale by the $accuracy^2$ if unsigned, and $(accuracy - 1)^2$ if signed: $scaled = obs\_norm * (accuracy - 1)^2 := 0.41\overline{6} * 128 = 53.\overline{3}$. Lastly, cast 53 to an 8-bit signed binary number, which is equal to 00110101. If the observation instead was -1, the result would be 11001011. Using this method ensures that we utilise as much information as possible. It is good when it comes to using a small number of bits per observation while maintaining a high accuracy. But a potential problem with this method lays with its interoperability. When counting in binary, bits are constantly changing values, especially bits to the far right. Continuing using the carts position as an example, when the cart moves from the scaled value of 63 to 64, the bit representations will drastically change from 00111111 to 01000000. This can make it difficult for the agent to perceive the state of the reservoir since small changes in the environment can have a huge effect. As to why this is considered a suboptimal strategy, a more in-depth explanation can be found in Section 4.2.3.

Linear translation was used instead. This method aims to keep the bits as consistent and interpretable as possible, but in doing so, also sacrifices some precision. The method divides the observation range from low to high into equally spaced chunks. The observation is placed in one of these chunks according to its value. The mapped observation will then consist of 1's for all the chunks up to that point, and 0's for all after. Given a range from -1 to 1, and an accuracy of 8, the dividing values would be the following: [-1, -0.714, -0.429, -0.143, 0.143, 0.429, 0.714, 1]. If the observation is -0.9, the bit representation would be 10000000, and 0 would be 11110000 and so on. If the observation is outside the lower or upper bounds, the converted observation will only consist of 0's or 1's respectively. The method is called linear translation due to the separation between 1 and 0 is linearly transferable from the observation. As mentioned, this method will require a larger size to achieve the same accuracy, but the

added benefit of having the mappings more consistent with small changes hopefully outweighs the negatives. This method was chosen to be used in all the following experiments.

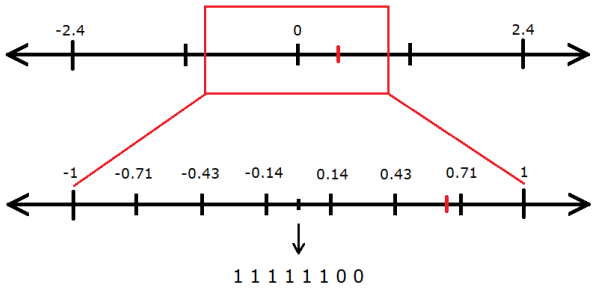
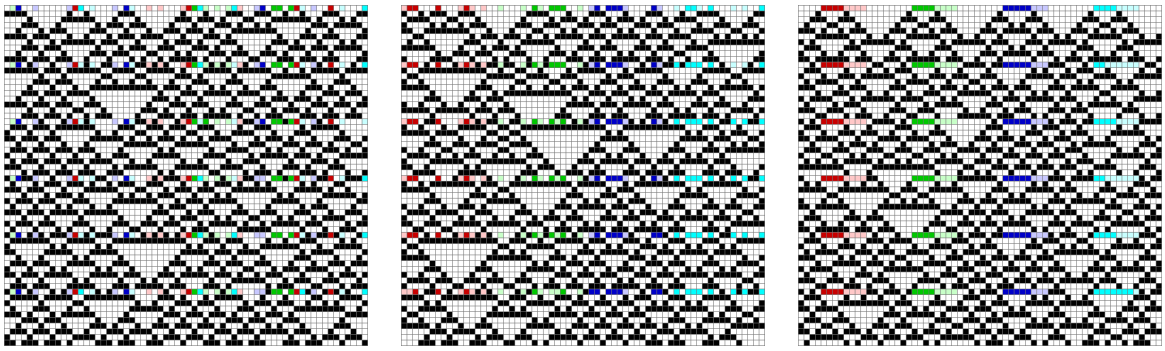


Figure 4.1: Illustration of the linear translation method.

Max observation values are the values defined in the environment that the observations cannot go beyond. For each observation, a minimum and maximum value is defined. Since the accuracy of the bits are drastically reduced when using the *linear translation* method, defining a custom range of possible values becomes important. For the carts position, a range of -2.4 to 2.4 is defined by the environment. If we instead define a range of -0.5 to 0.5 we also increase the accuracy for observations in that range. Figure 4.1 illustrates how this would work. In some cases, such as the cart and pole velocity, the range of possible values goes from negative infinity to positive infinity. Here, a custom range of values must be defined. Table 4.1 shows the values used for the cart pole environment in all of the experiments.

Table 4.1: Custom minimum and maximum vales for observations in the cart pole environment.

Observation	Minimum	Maximum
Cart position	-0.5	0.5
Cart velocity	-2	2
Pole angle	-0.1	0.1
Pole angle velocity	-0.5	0.5



(a) Global_random (b) Local_random (c) Fully_local

Figure 4.2: Illustration of the three different observation mapping methods.

Placing the converted observations into the reservoir is the next problem to tackle. This can have a major impact on the results, depending on how it is done. When working with conventional reservoirs, this is typically done in a random fashion. In our implementation, two additional methods will be tested. The first one, referred to as *global_random*, is the traditional method where all values are mapped to random locations in the last generation, as shown in Figure 4.2a. *Local_random* aims to restrict the placement of values based on the observation. This works by dividing the reservoir into sections and randomly map each observation to their distinct section. An example of this can be seen in Figure 4.2b. The sections will always be divided into equally large portions with no overlap. This also means that the total number of cells in a generation must be greater than the $number\_of\_obseravtions * accuracy\_per\_observation$. The last method, which is called *fully_local*, contains no randomness and will directly place the observation at the centre of each section, as shown in Figure 4.2c. Once a mapping is created, it will not change for that reservoir, meaning that as the reservoir is being updated, every value from the observation will update the value at the same location every time. It is also important to note that the *fully_local* method exists in the subset of *local_random* which again exists in the subset of *global_random*, and no two observations map to the same location.

With the *global_random* method, one additional parameter called *partitions* can be specified. The number of partitions specifies how many times each encoded bit is placed in the reservoir. If this number is 2, the reservoir will be divided into two partitions, and every encoded bit will be mapped to each of them.

There is also worth mentioning how the values in the reservoir is updated. This can be done in a plethora of ways, especially if neighbouring values are taken into consideration. In this thesis, there was unfortunately no time to do an extensive search of these possibilities, and keeping in line with maintaining simplicity, only two options were considered: Performing an XOR operation between the observation and

the value of the cell, and directly placing the encoded observation value in the cell, overwriting the current value. Utilising the XOR operation could help the reservoir with persevering its history due to the current value being taken into consideration.

Read

Reading the state of the reservoir means to return the current state of the reservoir. It considers only one variable, and that is how many rows the agent takes as input, or rather, how many of the last generations it should return. When this variable is 1, the method returns only the last generation. Considering the following scenario where the reservoir is on generation 20 and the agent takes 5 rows as input, the *read* method will return a 1D array consisting of the following generations: 16, 17, 18, 19 and 20. One thing that also needs to be considered, is what to do when the number of generations returned is less than the number of iterations performed per update. This will cause the read method to not have access to generations less than 1. This edge-case is handled by returning only 0's for all generations before the earliest generation. Given the case of reading 5 rows on generation 3, the method will return the following: all 0's, all 0's, generation 1, 2 and 3. Before the rows are returned, a reshape is performed to flatten the data such that all the rows will be consecutively placed in a 1D list.

Reset

The reset function is called whenever the environment is reset, or rather, at the start of every episode. The reset method for the reservoir will not touch reservoir specific variables, such as the update method or the observations mappings. Only the generation counter, the history, and lastly the cells of the first generation will all be set to 0. This is to always have a consistent starting state of the reservoir, where previous runs does not affect the current one. One interesting approach for future work could be to randomise the starting state of the first generation.

Configurations

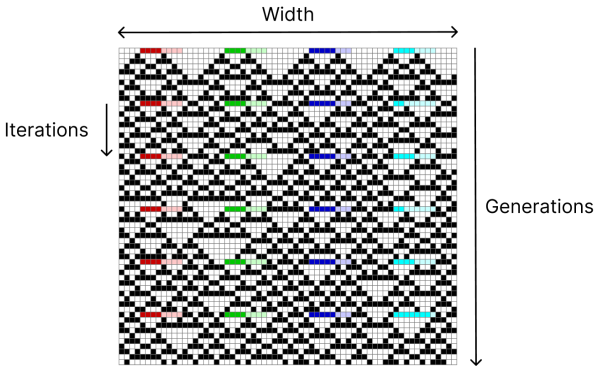


Figure 4.3: Example of reservoir with rule 54, 64 width, 10 iterations and 60 generations. The observation mappings method used is *fully_local*.

Figure 4.3 displays a reservoir where the relations between width, iterations and generations are shown. Note that all reservoirs in this thesis is shown with generations moving downwards. Some of the cells in the reservoir is coloured. These indicates cells that have been updated by encoded observations. The colours represents the different observations from the cart pole environment. Red is for the carts position, green is the carts velocity, blue is the pole angle and cyan is the poles angular velocity. Every colour has two distinct shades. Dark represents a 1 and light represents a 0.

Throughout this section of the reservoir, there have been many reservoir related mentions of variables and methods. For a short and concise overview, a list of these have been compiled below containing name, explanation and potential values.

- **Rule** is an ECA rule dictating which rule is applied, with a value in the range of [0, 255].
- **Cell** holds a single reservoir state value.
- **Width** is the number of cells in a generation. Can be any number above 0.
- **Cells** holds the current state of the reservoir, has a length of **width**.
- **Generation** is the count of how many times the ECA rule have been applied to the **cells**, mapping a observation to the reservoir does not increase the count.
- **Rows** contain a history of all the states, whenever the rule is applied to the **cells**, the new state is appended to this list.
- **Rows_input** is the number of how many of the last rows should be returned to the agent.

- **Accuracy_per_observation** specifies how many bits one observation is converted to.
- **Observation_mappings** specifies where the converted bits are placed in the reservoir. It is a list of unique indexes between [0, **width**], created once per reservoir either by the *global_random*, *local_random* or *fully_local* method.
- **Partitions** is the number of partitions the reservoir is divided into. Every encoded bit is mapped to each partition. Defaults to 1 unless specified.
- **Mapping_update_method** specifies how the converted bits are placed in the reservoir. Either by XOR or overwriting.
- **Iterations** is how many times the rule is applied to the cells after the observations have been placed in the reservoir.

Saving and loading the reservoir can also be done. This will save or load the following reservoir specific parameters: rule, width, rows_input, accuracy_per_observation, observation_mappings and iterations.

4.2.3 Agent

The agent part of the model is the only part where any training will occur. This will mainly consist of the trainable network that will get inputs from the reservoir, and subsequently pick an action to perform. The agent model will mainly have three methods. One for creating the network structure, one for training the network and lastly a method for performing a predict. Additional methods for saving and loading will also be implemented. These will save or load only the network weights and biases. In conjunction with the network, the agent will also store the state, action, reward, new state, and if the environment was terminated for every step taken in the environment in whats called a replay memory buffer. When the agents network is trained, a random batch of samples from this replay buffer will be used. This comes from the theories introduce in [13], which was presented in Section 3.2.

Network

The network is the main part of the agent. For the training of this network, a second one will also be used, a prediction and a target network. Both of these will be copies of each other, but during training, the target network will be used for predicting on future states while the prediction network will predict on current states. When fitting the network to its new outputs, only the prediction network will be trained, and after a certain number of episodes, usually 10, the prediction network weights are copied

over to the target network. This is done to reduce major fluctuations in the results due to the future predictions being done on a consistent network. This is also from the theories introduced in [13], which was presented in Section 3.2.

Linear topology is the structure of the networks. This refers to the number of layers in the network and not the activation functions used. This is also known as a single layer perceptron (SLP). By using a linear topology, the network will only have input and output nodes with no hidden layers. The number of inputs are dependant on two variables. First the **width** of the reservoir, and second the **row_input** which were described in the previous section. The inputs are one dimensional, so the number of inputs will therefore be the product of these two. The number of outputs is the number of actions that can be taken in the environment. The reason as to why a linear topology was chosen has to do with the investigatory nature of the ECA rules. If hidden layers were used, so-called multi layer perceptron (MLP), the network alone could be capable of approximating a function for separating the data, and take credit away from any eventual computational properties of the rule. By not having any hidden layer, the task of separating the data therefor lies with the rule of the reservoir and not with the neural network. The new task of the network becomes therefor to read the state of the reservoir and produce an action and nothing else.

Fitting the network

The training method consists of three main steps. In the first one, a minibatch is sampled from the replay memory buffer. The main and target network then make predictions on the current and next state respectively. The predicted values will resemble what the network thinks the future reward will be for each action. The next step is to calculate what the new output should be when taking the action chosen and the reward into consideration. If the action did not result in a terminated state, the new output for that action is calculated by summing the current predicted value from the target network by the reward received, the result of which is then multiplied by a discount factor. However, if the action did end in a terminated state, the new output is set equal to the reward. The last step is to update the network based on the state and the new output for that particular action.

4.2.4 Training loop

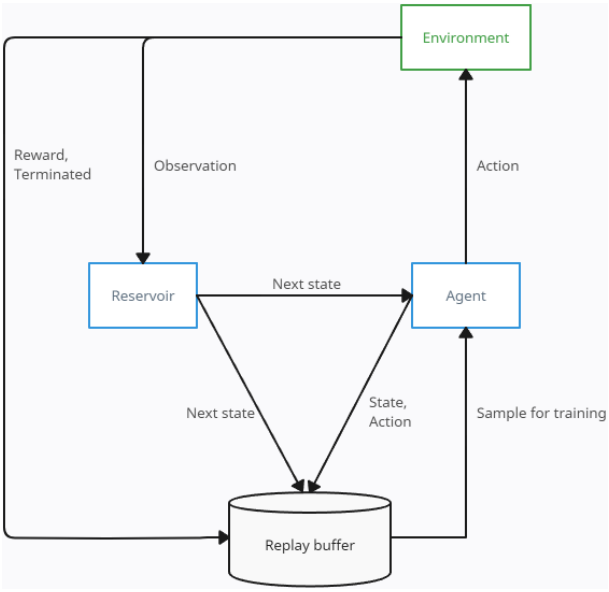


Figure 4.4: Illustration of one step in the environment.

With all the individual parts of the model explained, it is time to put together the whole training loop. Before any training can take place, one must first pick the rule to test. Once that is done, both the reservoir and the agent can be created. At the start of every episode, both the reservoir and the environment must be reset. The environment will return the initial state which is used to update the reservoir. The reservoirs state can now be read. Next, a action must be sampled. This is either randomly selected or chosen by the agents network depending on the epsilon value. If the agent is to chose the action, it will do so based on the state read from the reservoir. The action is then performed in the environment which returns an observation, reward and if it was terminated. The reservoir is again updated and the new state is read. The agents replay memory is then updated with the state, action, reward, new state and terminated. A single fitting operation is performed before the loop continues. This is done until the environment terminates. Lastly, the epsilon value is decreased, completing one episode. This loop will continue to run until a set number of episodes have been reached. A schematic of on step can be seen on Figure 4.4.

4.3 Experiments

In this section, we will go over the different experiments planned to perform. The nature of these revolve around testing rules in different environments, finding good parameters regarding both the reservoir and the agent. Before starting, there are a few rules out of the 256 that can be excluded from any testing. These rules are either

mirrored, complement or mirrored complement to others. The number of rules with unique properties is 88, which will help reduce total training time since performing 256 tests per parameter change is no longer needed. These unique rules can be seen in Figure 2.2 back in Section 2.1.2.

4.3.1 Parameters

Before starting the training, there are a few parameters that needs to be specified. Mainly the number of episode and the epsilon decay rate. The values of these can be seen in Table 4.2.

For updating the agents network, a learning rate of 0.001 and a batch size of 64 were used. The max length of the replay buffer was set at 50,000, and no training would occur before it reached a length of 1,000. A discount factor of 0.95 was used when calculating the Q-values of the next state used in agents train function. The target network would be updated every 10 episodes for training sessions less than 5,000 episodes, and every 100 episodes when greater. These parameters stayed the same throughout all of the experiments.

Table 4.2: Epsilon decay rate used at different number of episodes

Number of episodes	Epsilon decay rate
1,000	0.9975
5,000	0.9990
10,000	0.9995
20,000	0.9997

4.3.2 Generations

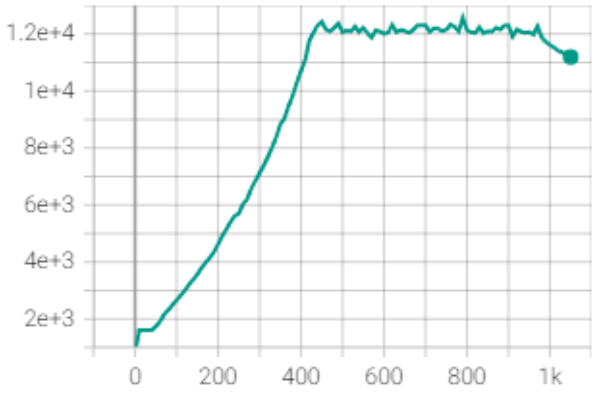


Figure 4.5: Plot of memory usage during training. x-axis show memory usage in MB, y-axis shows episodes.

Chapter 5 is divided into two main sections, named 1. and 2. generation. The reason for this is that half way through the experiments phase, a major change to the code had to be made. In the 1. generation implementation, the python library Keras [4] was used for all the operations surrounding the agent networks creation, update and prediction. All training with this implementation was done using a *NVIDIA GeForce RTX 3060 Ti* graphics card for GPU accelerated training. Unfortunately, a memory leakage was discovered in the latest version, resulting in all of the systems memory being used and eventually crashing as seen in 4.5. A workaround was implemented to clear the systems memory if its usage reached 95%. This resulted in training being significantly slower.

In the 2. generation, all training was done using the PyTorch [14] library instead. Additionally, the overhead cost of transferring the training data to and from the GPU, made the difference between GPU and CPU training negligible and the CPU version of PyTorch was used. All experiments conducted in this implementation was done on a *Intel(R) Core(TM) i5-12400F* CPU. The 1. generation was capable of performing 4-5 steps per second, whereas the 2. generation now was capable of performing approximately 50 steps per second on average. A 10 times increase in performance.

4.3.3 General ECA rule testing

The goal of these tests is to get a general understanding of how the different rules, in combination with different reservoir setups impact the performance, and if they work in a particular setting or not. In these experiments, all 88 unique rules are trained using the same setup, with the exception of the observation mappings, which is randomly picked when a new reservoir is created.

4.3.4 Scoring

In section 5.2, the term score is used to rank rules in the different experiments. To calculate the score, the finished trained models are loaded and given 100 attempts at solving the environment. During their attempts, no random actions are taken, the model will always pick which action to take. For the cart pole environment, if the model reaches 500 steps, the attempt is terminated. Managing to consistently achieve 500 steps is considered solved. The score is therefore the average reward over the 100 attempts.

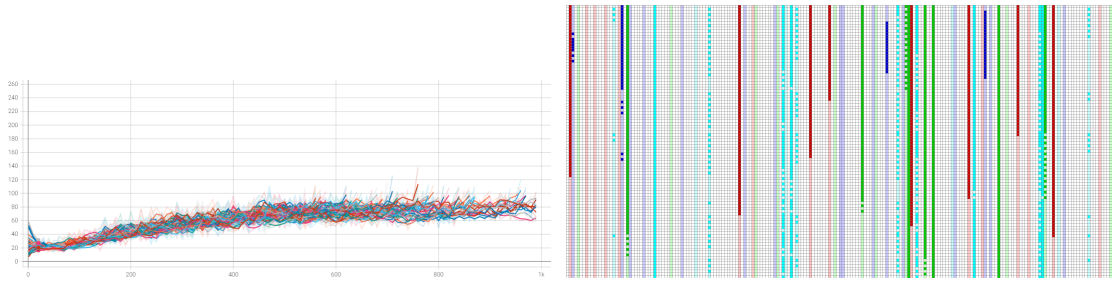
Chapter 5

Results

This chapter contain the results from the experiments performed. The chapter is divided into two main sections, one for each generation of implementation.

5.1 1. Generation

5.1.1 Experiment 0, 0 iterations



(a) 64 runs, x-axis is episodes and y-axis is average reward over 10 episodes

(b) Example reservoir from one run

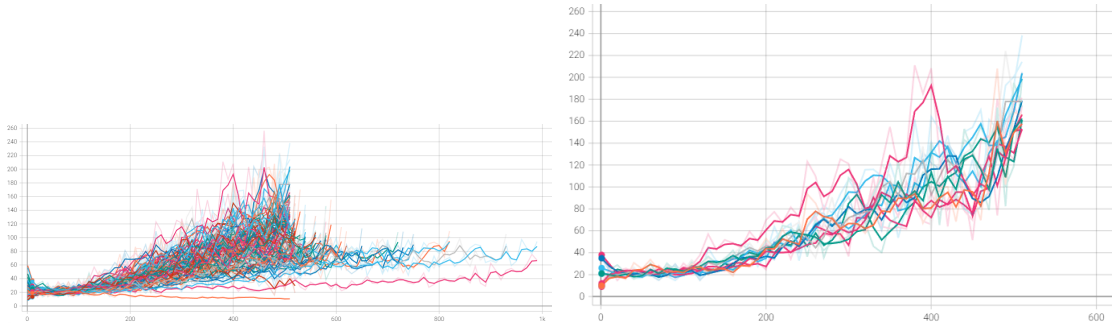
Figure 5.1: Runs using 0 iterations, 200 width, 16 accuracy per observation and 1 row as inputs to the left. On the right is a reservoir showing generations from top to bottom

To get a baseline of the performance using the implemented methods, a total of 64 runs was conducted where the rule was applied zero times between updates. This will effectively not allow the data to permute through the reservoir. The networks training data will therefor only consist of the converted bits from the environment. A plot of the training result and an example of one of the reservoirs can be seen in Figure 5.1. In these initial tests, a couple of extra constraints were put in to speed up the training process. One of these were to stop the training if the model archived an average reward

above 100 after 500 episodes. This will in most cases mean that the agent is capable of some sort of learning. Observing the plot reveals that all instances performed roughly the same with only slight variations. About 80% of the models managed to get an average score above 100 between 500 and 1000 episodes, 20% did not and no model managed to get an average score above 100 before the 500 episode mark. This shows some learning capabilities, although slow. Training a couple more models without stopping, showed that this model tend to converge between 80 and 120. Observing some of this models in the environment, showed that a common behaviour for most of these models were to overwhelmingly move in one direction, causing the pole to be balanced as the cart moved off the screen if the spawning angle was right. If the spawning angle was opposite, the model would quickly fail. This resulted in scores around 100-150 if good starting conditions, and 20-50 if bad.

In Figure 5.1b the reservoir shows one of these "good" runs where the cart moves to the left, keeping the pole in a negative angle. The carts position is coloured in red, dark if 1 and light if 0. As the cart travels to the left, notice how one column after another changes to a light colour. Observing the green colour, describing the carts velocity, we can also deduct that the cart is accelerating towards the left. The angle is described by the blue colour, and the angular velocity is the cyan colour. Notice also how the angular velocity is close to 0 by the number of 1's and 0's being close to eight at any row. Since the *global_random* observation mappings was used, it might be a bit hard to tell, but in Section 5.2, a similar experiment was performed where a much clearer figure can be seen.

5.1.2 Experiment 1, general performance



(a) All 88 unique rules

(b) Top 10 rules in order: 74, 35, 184, 58, 168, 172, 57, 152, 138, 232

Figure 5.2: Two plots showing a training run using 5 iterations, 200 width, 16 accuracy per observation and 5 row as inputs. x-axis is episodes and y-axis is average reward over 10 episodes

In the second training run, all 88 unique rules were tested. This time, the rule was applied 5 times between updates, the reservoir had a width of 200, with 16 bit accuracy per observation and 5 rows as inputs using the *global_random* observation mappings. These parameters were chosen as a good starting point due to their middle ground of not having a small or large size, number of iterations and including plenty of generations as training data. The ratio between number of cells overwritten and the width is approximately 1/3, given by: $update\_ratio = num\_observations * accuracy\_per\_observation/width$. This gives cells from previous generations the opportunity to impact the current ones. Note that if this ratio is 1, the entire reservoir would be overwritten when updated.

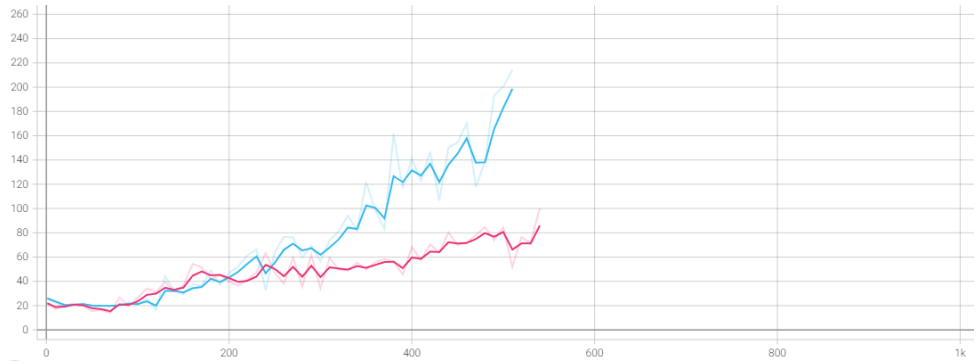


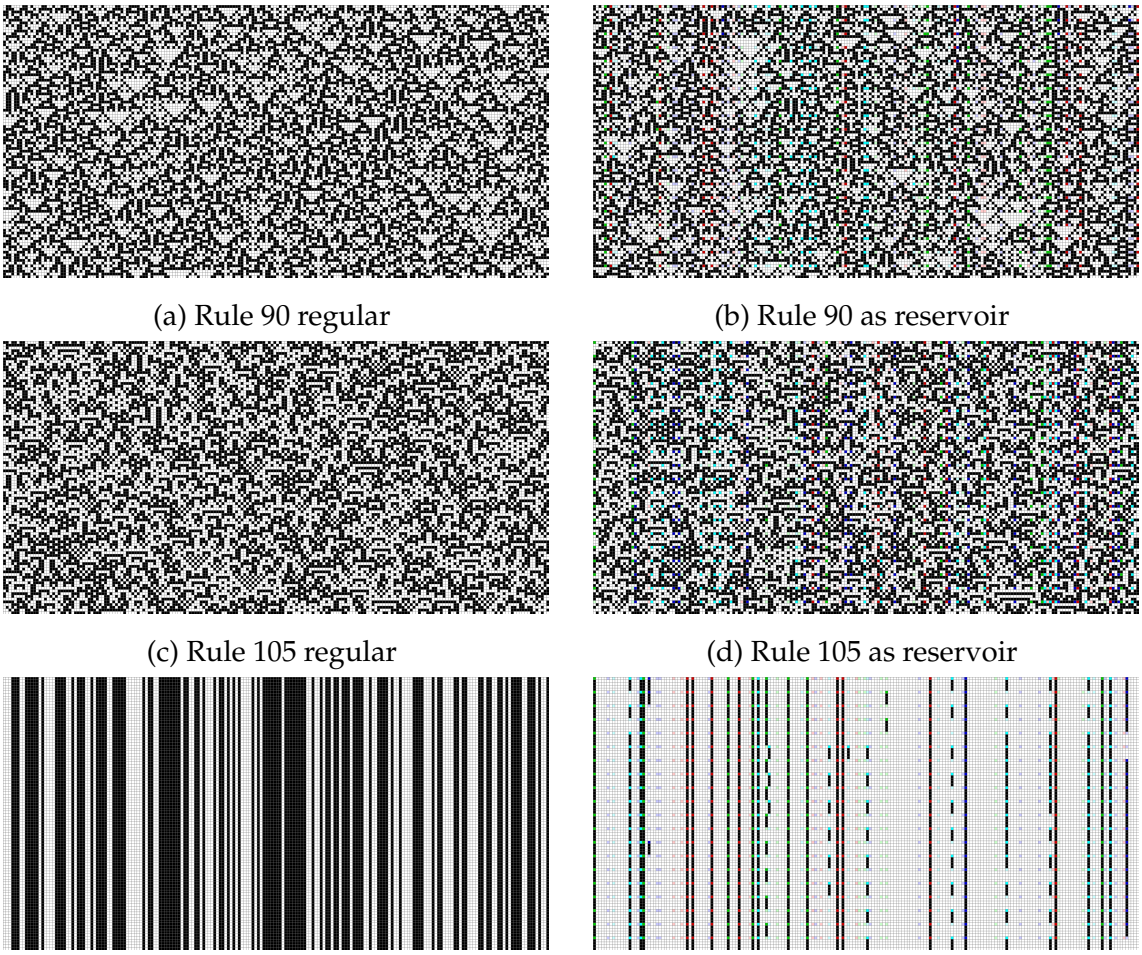
Figure 5.3: Comparison of the best performing run with 0 and 5 iterations. The x-axis is episodes and the y-axis shows average reward over 10 episodes. The rule applied is 74, shown in blue and the pink has no rule applied.

Comparing the best results from these two runs, shown in Figure 5.3, indicates that applying the rule gives the model a large advantage. From the previous section, we know that the model where 0 iterations where done between updates are about to converge. While the model where rule 74 was applied seems to still be on a steep learning curve. Of course, one could argue that the run where 5 iterations was applied had 5 times more trainable neurons, and had therefor an advantage. Increasing the number of trainable neurons can be done in two ways. Either by increasing the width of the reservoir or increasing the number of rows passed to the network. Since no rule is applied, the model would be unable to take advantage of the extra width, which leaves only the second option. Increasing the number of rows can be done in two ways. The first way is to copy the cell states downward x number of times and give them to the network. This, in a sense, will make the network have multiple neurons connected to the same cell state. The problem with this, is that the cells will all be either 1 or 0, and since the readout layer is linear, the output of these neurons will be summed and act as a single value regardless. The second way is by directly including multiple updated rows. This shows a slight increase in performance, but not to a point

where it can be deemed significant.

In Figure 5.2, the results from all 88 unique rules can be observed. The same constraints are still applied where one rule will stop training if it reaches an average above 100 after 500 episodes. Even with this, the full run took around 150 hours to complete. Many of the models are yet to converge, but training all of them the full 1000 episodes would mean triple or quadruple the training time as the performance would further increase. From Figure 5.2a, we can observe that these parameters results in a large spread of performances. Some rules are barley increasing in performance beyond performing random actions, while others reaches close to 200 after 500 episodes. As expected, rule 0 performs the worst with rule 128, 150 and 105 not being able to score an average score above 100 after the full 1000 episodes. Other weak performing rules include 136, 146, 90, 204, 1 and 126.

In Figure 5.2b, the top ten rules are singled out. By observing the classifications of the best and worst rules, a pattern starts to emerge. Out of the worst performing rules, three of them belongs to the class 1 ECA classification, two to class 2 and five to class 3. The two rules belonging to class 2, 1 and 204, can at a first glance look to behave dissimilar, but the two share the property of only moving cell states downward with no diagonal movement. The difference is that rule 1 will do it in an alternate fashion, while rule 204 will directly copy the states. These properties are similar to the case described above where multiple rows should be included in the training to have more neurons per state. As shown, these properties perform among the weakest, not beating any of the experiments with 0 iteration. The other weak performing rules all have uniform or chaotic behaviour. This usually results in models where the reservoirs state disappear after a couple of iterations with minimal cell interactions, or reservoirs where the state seem random meaning small changes in the observation results in vastly different states. Beneath are examples of reservoir with weak and moderate results shown in Figure 5.4 and 5.5.



(a) Rule 90 regular

(b) Rule 90 as reservoir

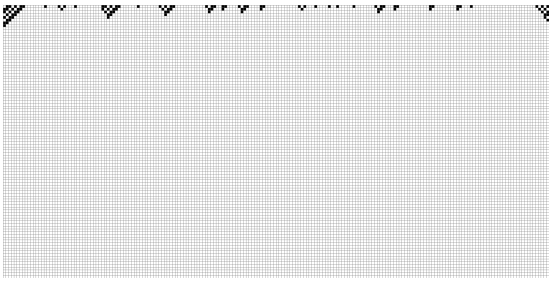
(c) Rule 105 regular

(d) Rule 105 as reservoir

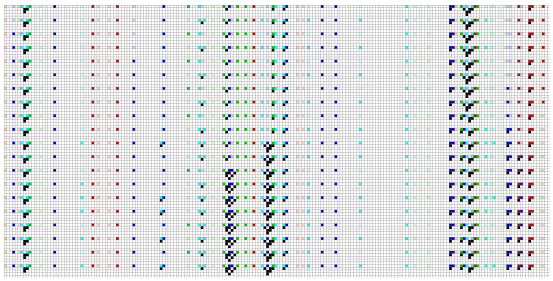
(e) Rule 204 regular

(f) Rule 204 as reservoir

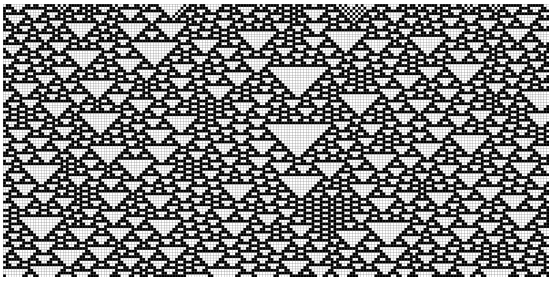
Figure 5.4: Examples of weak rules from random initialisation on the left and when used as reservoir on the right.



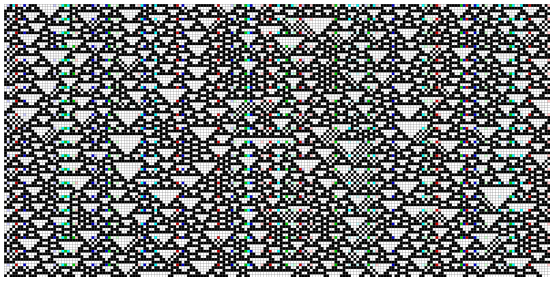
(a) Rule 40 regular



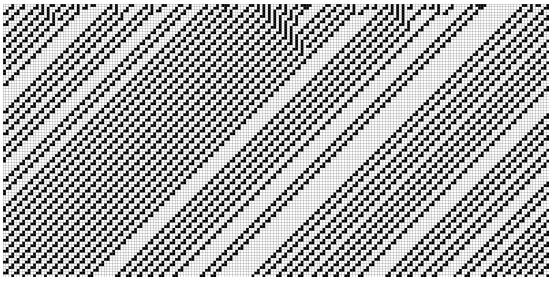
(b) Rule 40 as reservoir



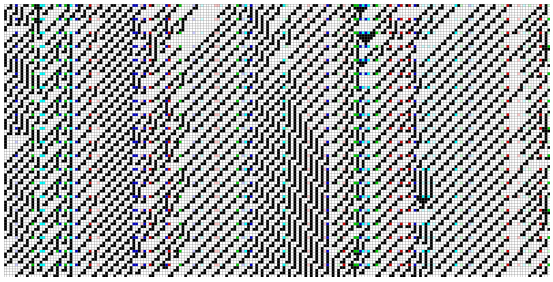
(c) Rule 122 regular



(d) Rule 122 as reservoir

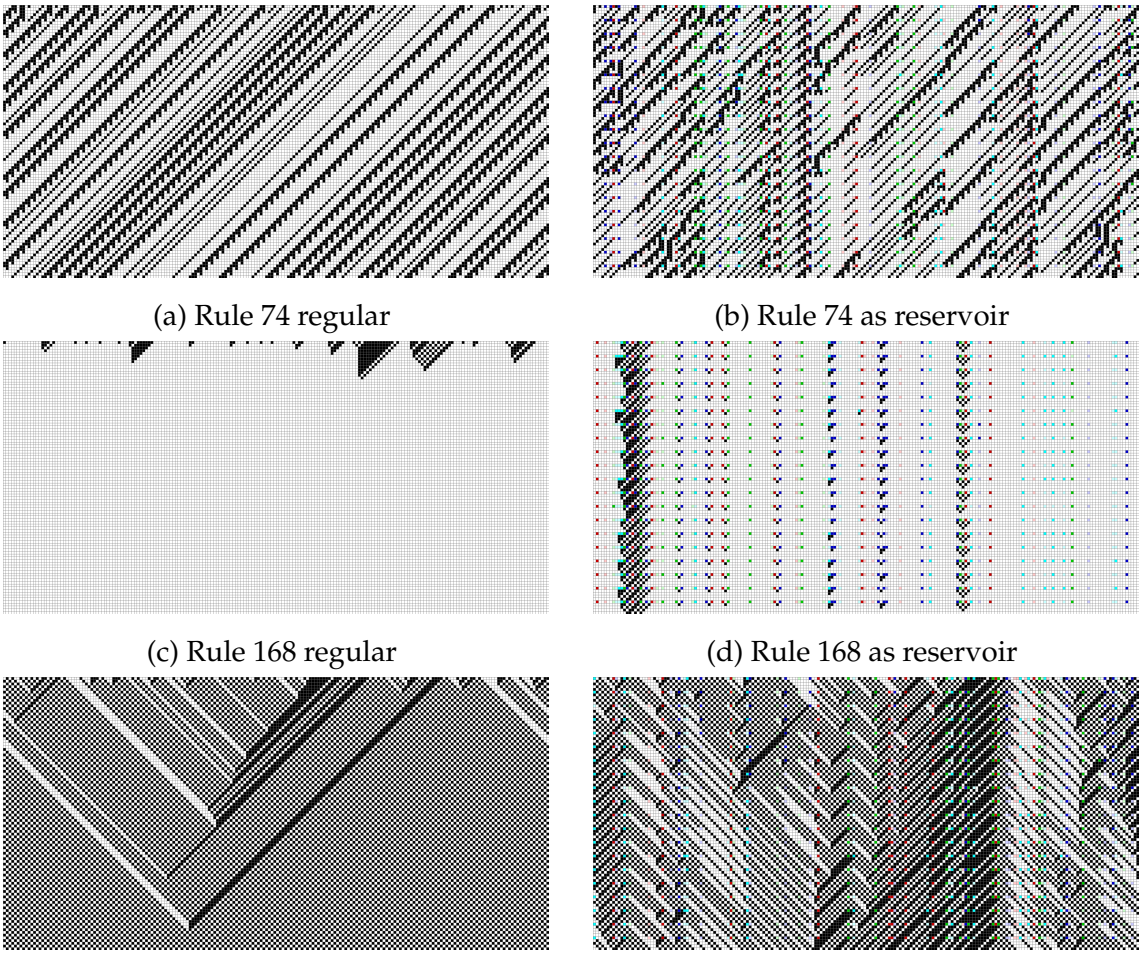


(e) Rule 134 regular



(f) Rule 134 as reservoir

Figure 5.5: Examples of moderate rules from random initialisation on the left and when used as reservoir on the right.



(a) Rule 74 regular

(b) Rule 74 as reservoir

(c) Rule 168 regular

(d) Rule 168 as reservoir

(e) Rule 184 regular

(f) Rule 184 as reservoir

Figure 5.6: Examples of good rules from random initialisation on the left and when used as reservoir on the right.

Out of the top 10 best performing models, all of them belongs to the ECA class 2 with the exception of rule 168 which is in class 1. Some reservoirs of the models that performed among the best is shown in Figure 5.6. These rules can at a first glance appear chaotic, especially when observing the reservoirs in Figure 5.6b and f. Observing how the ECA behaves when starting from a random initialisation, on the other hand, shows that the rules quickly settles into periodic patterns, often in a diagonal motion. This can be seen in Figure 5.6a and e. This allows information from one observation to travel and interact with other observations. One interesting behaviour of rule 184, is that these "streams of information" seems to cancel each other out. These two properties seem to highly resemble the two properties necessary for a successfully reservoir; 1) inputs to the system must be able to propagate throughout the reservoir. 2) as information propagates, it must not amplifies.

Rule 168 appears to work quite differently than the two other rules with only some diagonal movement if multiple 1's are close by. As a class 1 rule, it quickly settles

into uniformly 0's after few iterations. As a reservoir this can be favourable since it allows some cell interaction while limiting the amount of noise. In Figure 5.6d, it is still noticeable which regions have high or low activity. This rule might benefit from using a more carefully crafted observation mappings, or a smaller reservoir in general.

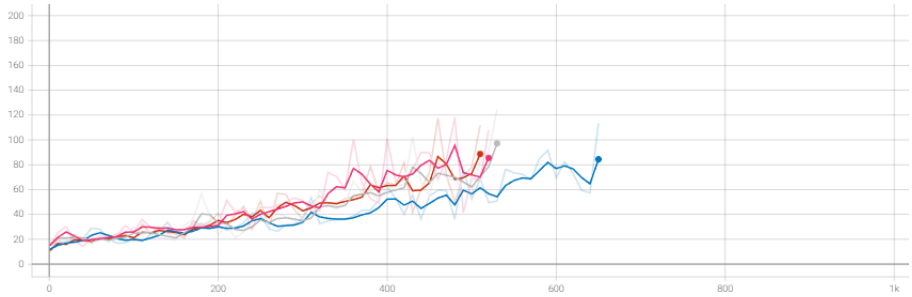


Figure 5.7: Results of the four complex rules 41 (blue), 54 (red), 106 (pink) and 110 (grey). x-axis (or X-axis) shows episodes and y-axis shows average reward over 10 episodes

Surprisingly, no class 4 rules of the complex type were among the best or the worst. Looking at the plot shown in Figure 5.7, they can be observed achieving similar results which are barely any better than the worst. However, they seem to be improving at a steady pace. Due to their complexity, and therefor the many possible states the reservoir can be in, they are likely to need much more training to achieve similar results as the top performing rules.

5.1.3 Experiment 2, Longer training

In the second round of training, only a handful of rules were picked out to be trained in a model for a longer period. The same parameters were used as in the previous round, with the removal of the restriction that stopped training if the model achieved an average score above a certain number. In addition, the two other observation mappings method *local_random* and *fully_local*, were also tested.

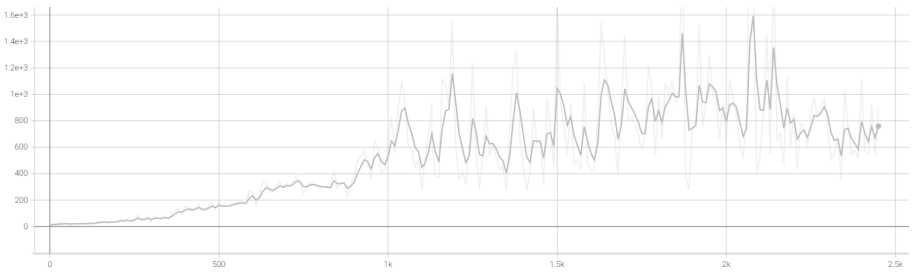


Figure 5.8: Rule 74 training results. x-axis shows episodes and y-axis shows average reward over 10 episodes

In the first training session, the best performing setup from the previous round was trained for just under 2,500 episodes. The rule used was 74 and the results can be seen in Figure 5.8. The training is steadily improving for the first 1000 episodes. Then the results oscillates regularly between 400 and 1400, before settling at around 700-800. It took approximately 72 hours to complete.

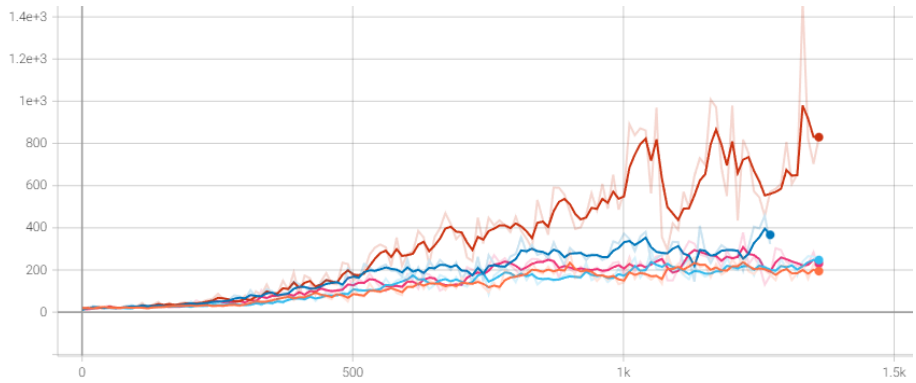
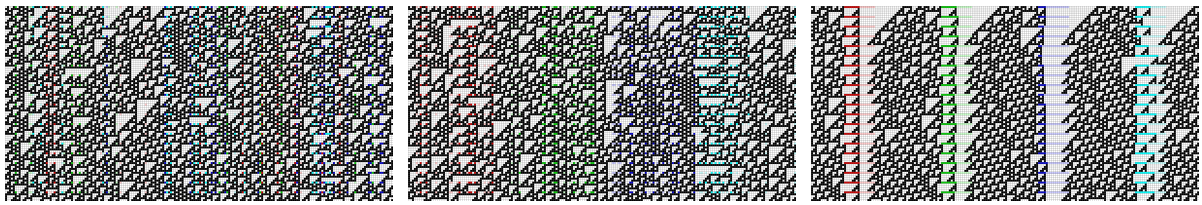


Figure 5.9: Results of rule 184 (red), 168 (dark blue), 41 (light blue), 90 (pink) and 110 (orange). x-axis shows episodes and y-axis shows average reward over 10 episodes

Next, one rule from each ECA class were trained for 1,400 episodes. One additional rule from class 4 were added as the results from the first one did not perform well. The results are shown in Figure 5.9. The rule used from class 2 was 184 and shows a learning curve similar to that of 74s from the previous graph. Compared to the other four plots, it performs significantly better. Rule 41, 90 and 110 all ends at around 200 showing no further improvements over the last 800 episodes. Rule 168, from class 1, ends at a slightly higher 240, but is also struggling to further improve.

Observation mappings

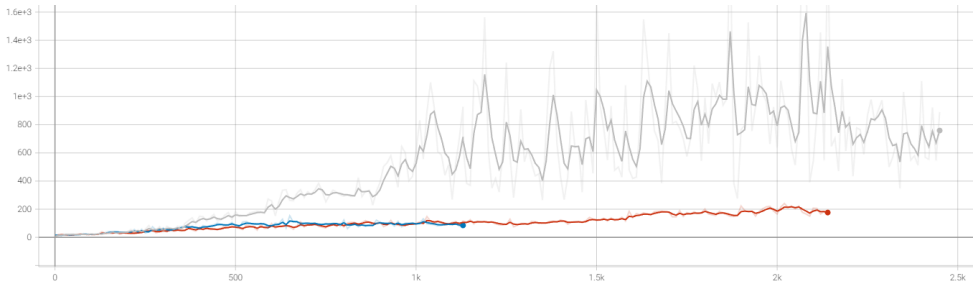


(a) Global random (b) Local Random (c) Fully local

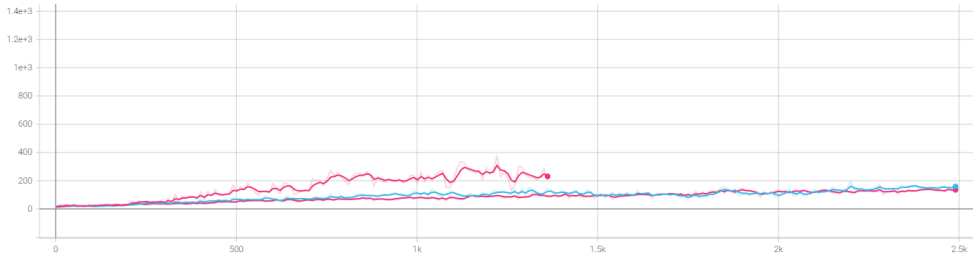
Figure 5.10: Models where the *global_random* (GR), *local_random* (LR) and *fully_local* (FL) methods were used. The rule applied is 110.

The next test is focused on how different observation mappings impact performance. Figure 5.10 shows the three different mappings tested where the coloured cells

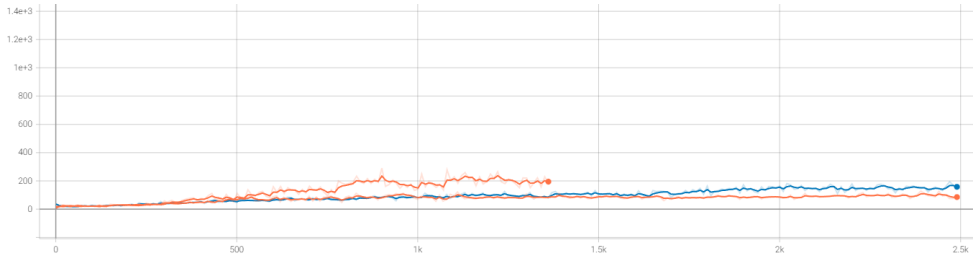
indicates where the reservoir is updated, and the colour corresponds to different observations.



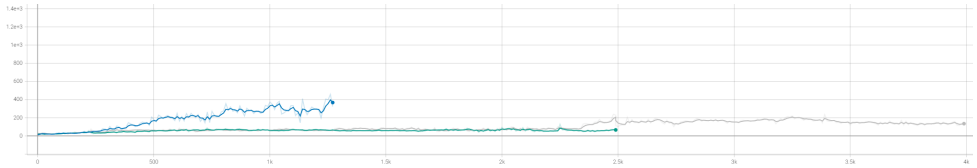
(a) Rule 74, GR is grey, LR is blue and FL is orange



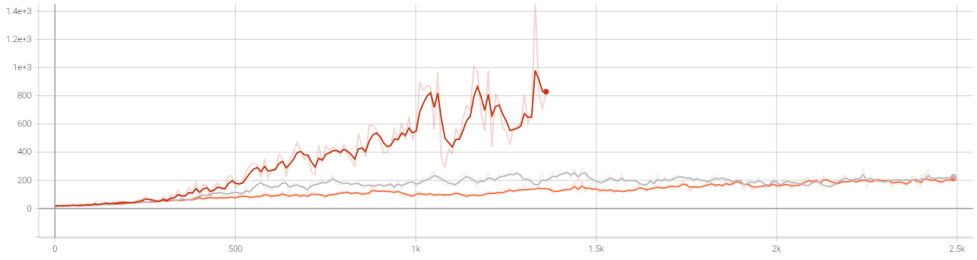
(b) Rule 90, GR is upper pink, LR is blue and FL is lower pink



(c) Rule 110, GR is upper orange, LR is lower orange and FL is blue



(d) Rule 168, GR is blue, LR is green and FL is grey



(e) Rule 184, GR is red, LR is grey and FL is orange

Figure 5.11: Five models trained with three different observation mappings. x-axis shows episodes and y-axis shows average reward over 10 episodes

The results from testing different observation mappings can be seen in Figure 5.11. It shows that the *global_random* method achieves higher rewards than the other two

methods in all the cases tested. Notice in Figure 5.10c, where the reservoir of rule 110 is displayed, how the history is allowed to be passed downwards between the updated columns, but with no interactions between the observations. As soon as one observation reaches one of the others, it is overwritten before any interactions can happen. This is also prevalent in 5.10b, where minimal interactions between the carts position and the pole angle, or the carts velocity and the poles angular velocity is possible. This shows a clear example of how a bad observation mapping that do not allow for many interactions between observations to happen, causes the model to struggle. This is best shown in Figure 5.11a and c where the *global_random* mapping performed much better, but is also visible in 5.11b, c, and d even when the *global_random* mapping struggles in the first place.

The *fully_local* method used in 5.11c showed a sudden spike in performance right before the training ended at 2,500 episodes, and was therefore allowed to train for 4,000 episodes. This was to make sure that the model had properly converged before stopping the training. It did not further improve and seem only to have hit a "streak of luck" around that point.

5.1.4 Experiment 3, Bipedal Walker

Moving to a more advanced environment meant that a couple of changes had to be made. Firstly the environment takes four continues values as input, which was discretized to 11 actions per input. This also meant that the loss calculation was done on four batches of actions four separate times. The reason for having 11 possible actions was so that the model could pick actions in increments of 0.2 from -1 to 1, including 0.

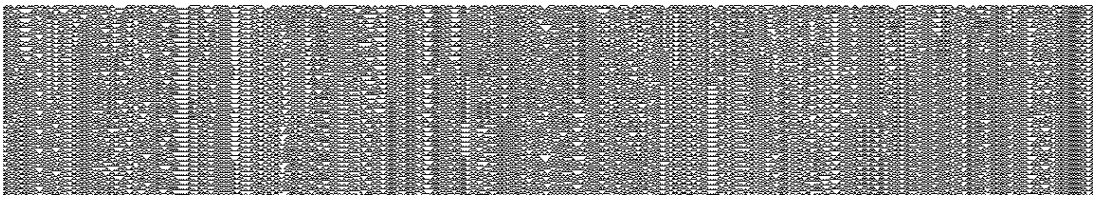


Figure 5.12: Reservoir with width 1,150 and rule 54 applied.

The reservoir parameters used were a width of 1,150, 5 iterations between updates and an accuracy per observation of 16 bits. Since the environment has 24 observations, the update ratio is kept at three. Two additional constraints were also set in place for each episode: 1) the episode would terminate if the number of steps reached 1,000, 2) the environment would terminate if cumulative reward reached 300.

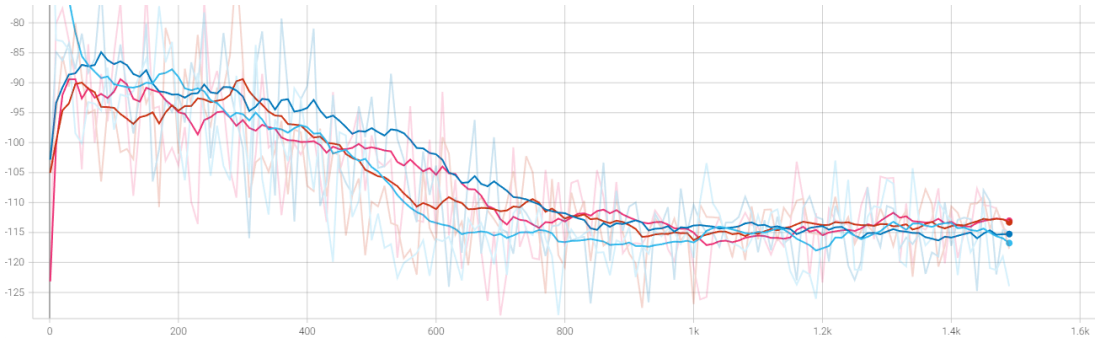


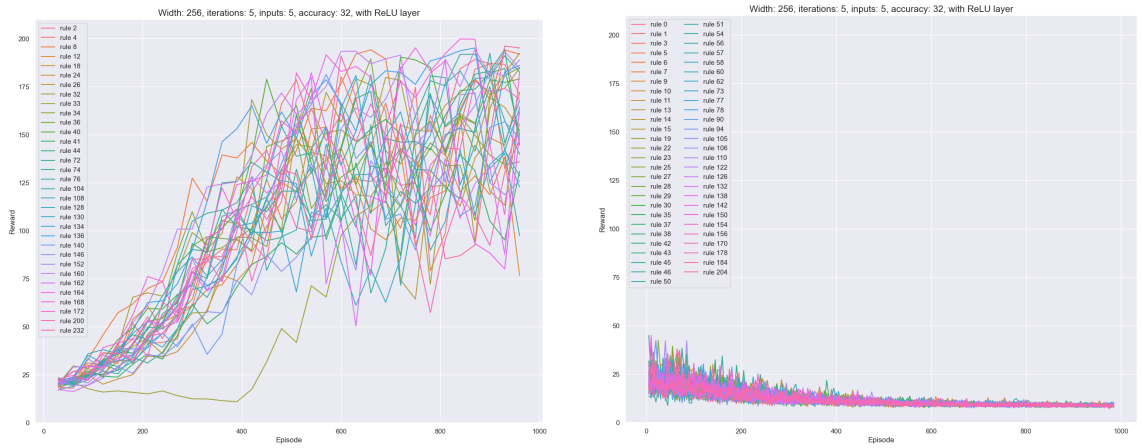
Figure 5.13: Results from four training's in the bipedal walker environment. Displaying results from rule 90 (pink), 35 (orange), 184 (dark blue) and 168 (light blue). x-axis is episodes and y-axis is average reward over 10 episodes. Note that the x-axis ranges from -125 to -80.

By observing 5.13, it becomes clear that the models did not perform well in this environment. All rules tested follows the same pattern, scoring around -90 at the start and ending at around -115. Rule 54 and 74 were also tested, but were preemptively stopped due to performing identical to the four others. Common strategies that the models adapted throughout the training included; kicking up one of the legs behind them to fall forward, or quickly spreading the legs such that the head would not touch the ground and staying in this position for the remainder of the episode. These strategies are not particularly good, but considering that applying motor torque gives a small negative reward, it makes sense as to why the models defaults to moving as little as possible, or at least makes sure to fall forward. There are much that can be done to improve the performance, but due to the time limit quickly approaching, it was decided not to pursue the subject further and instead move back to the cart-pole environment to further continue testing there.

5.2 2. Generation

This section covers the results from the optimised training using the PyTorch library. See Section 4.3.2 for details. In short, training could be done 10 times as fast. Before any of the following tests were conducted, a small sample of similar tests as in Section 5.1.1 and 5.1.2 were conducted, achieving close to identical results. *All episodes are terminated if 200 steps is reached

5.2.1 Experiment 4, with ReLU



(a) 33 rules with improvements

(b) 55 rules with no improvements

Figure 5.14: Training plot of models with ReLU layer

The focus for this experiment was to see whether the models would achieved better results by increasing the accuracy per observation to 32. Unfortunately, as the next test involving comparisons to regular neural networks were being prepared at the same time, a ReLU layer after the outputs was included in the agents network. This was not discovered until after all models had finished training. It was considered not to include the results from this run, but an interesting artefact was found when plotting the rewards from the training. As seen in Figure 5.14, 55 out of the 88 unique rules did not manage to improve with the additional ReLU layer. Also, the the rules that showed improvements seemed to perform better overall, but this can be due to the extra accuracy or higher update ratio.

Table 5.1: Average score over 100 trials of the top 10 rules

Rule	Class	Score
72	2	476.57
168	1	338.70
140	2	326.81
12	2	279.86
76	2	238.00
8	1	231.89
160	1	221.80
36	2	200.80
24	2	198.82
4	1	198.16

In this run, the width was 256 using 5 iterations, 5 rows as input and a accuracy of 32 per observation meaning the update ratio is 1/2. From the results shown in Table 5.1, we can observe that the rules belonging to class 1 and 2 are still on the top, with no rule from class 3 or 4 present. The best performing rule from class 3 and 4 was 18 and 41 with scores of 98.39 and 17.21 respectively.

5.2.2 Neural network comparison

(a) 1,000 episode training of four NN topologies.

(b) 10,000 episode training comparison of a NN and rule 33 and 34.

Figure 5.15: Training plot of NNs with different topologies on the left, and comparison of a NN and two reservoirs with rule 33 and 34 on the right.

To be able to compare the model with a more traditional approach, a couple of regular DQN models was trained using different network typologies. For this to be possible, two changes had to be made. First, the observation would not be passed through to the reservoir but directly given to the agent. Second, the agents network topology was altered. All other aspects remained exactly the same. The name of the models that can be seen in 5.15a and b, refers to the number of nodes in each layer. Note that all models have 4 inputs and 2 outputs with ReLU layers between all of them. Observing Figure 5.15a, it becomes clear that the NNs struggle to improve over the course of such a short training session. In Figure 5.15b, where the number of episodes have been increased to 10,000, the two implementations seems to even out over time as training progresses.

There are two additional things to point out here: 1) for the first 3,000 episodes the models using the reservoir excels way in front of the NN, before being cough up to. 2) although being cough up to, the NNs performance fluctuate much more throughout the rest of the training.

5.2.3 Experiment 5, large width

This run and the next one, focuses on how the models performance differs with changes to the reservoir. In this first case, a wider reservoir with more space and fewer iterations is tested. The parameters used were a width of 384 with 3 iterations and rows for the network inputs. The accuracy per observation was set to 16 meaning that the update ratio is 1/6. With such a small update ratio, much more history is able to pass through generations without being overwritten.

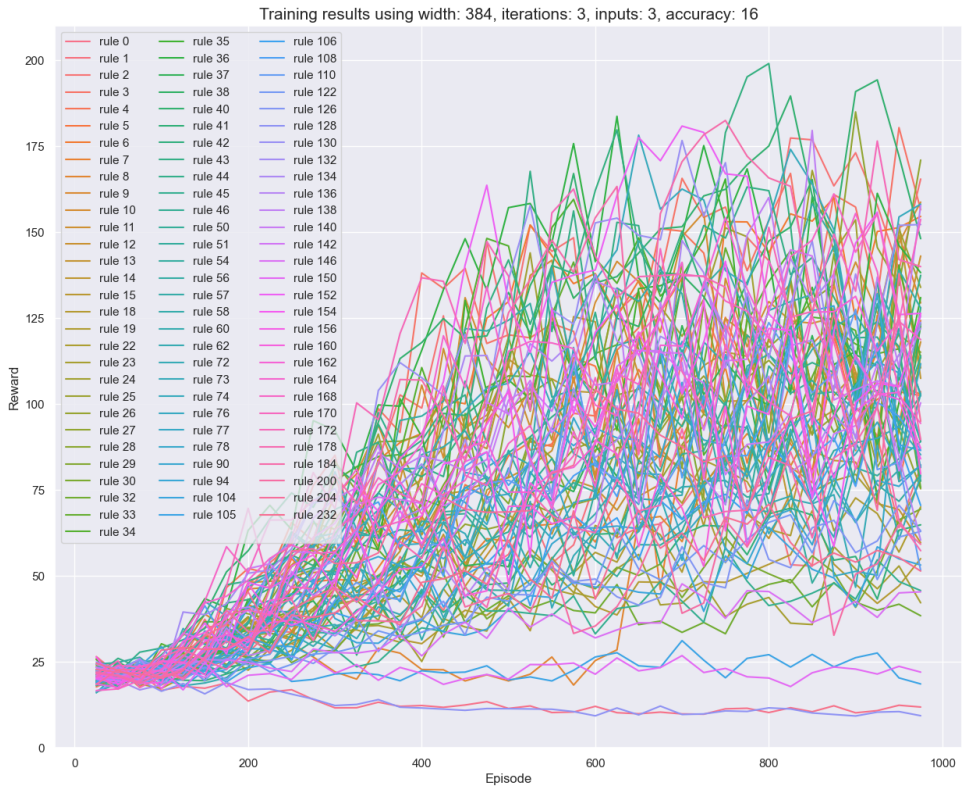


Figure 5.16

Table 5.2: Average score over 100 trials, including top score from class 3 and 4.

Rank	Rule	Class	Score
1	8	1	463.15
2	130	2	299.39
3	10	2	205.71
4	15	2	200.64
5	36	2	188.90
6	57	2	182.40
7	26	2	175.91
8	28	2	173.28
9	38	2	172.94
10	4	2	172.82
⋮	⋮	⋮	⋮
18	60	3	152.05
⋮	⋮	⋮	⋮
36	54	4	99.96

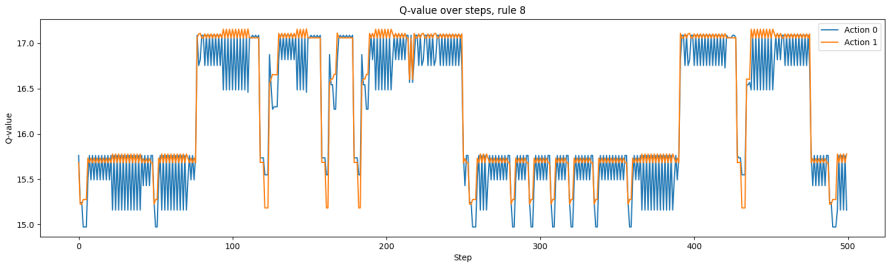
Scoring this run, shown in Table 5.2, it becomes clear that rules from class 1 and 2 are still on top, with rule 8 scoring significantly higher than any other. It is interesting to see that a class 1 rule outperform the others with this type of setup. Rules from class 3 and 4 generally did not improve much over the course of this run, with the exception of rule 60 which almost managed to reach top 10.

The score of rule 8 was so high, that it warranted some extra investigation. Rule 8 is only able to create an activated cell with 011 as inputs, meaning that the reservoir will only consist of 0's after two generations. For the reservoir to not be perceived as linear, some adjacent cells must be updated. As the update ratio is lowered, the likelihood of this decreases. The reservoir of rule 8 in this case has five pairs of cells that is updated. This has therefore the potential to activate five additional neurons. Even with this small amount of trainable neurons, the model is still able to perform better than any other. To see if this was a one of a kind type of situation, five additional models with rule 8 was trained using the same setup but with new random observation mappings. The score of these new models were; 155.80, 87.09, 72.19, 45.39 and 9.38, confirming that the original model was "lucky".

Plotting the Q-values over one episode gives some additional information about the behaviour of rule 8. The plot can be seen in Figure 5.17a showing the model oscillating between two major states. Observing the environment shows that the model slowly moves back and forth between two points on the screen. The plot is stopped after 500

steps, but letting the model continue, it has been observed reaching 10,000 steps. For a simple environment, simple setups such as this one can be quite effective. But for a more advanced environment, where the importance of history becomes more apparent, this behaviour is likely to not translate well. The Q-values appears to be fairly static, only moving between a couple of values each.

In Figure 5.17b on the other hand, we can observe the Q-values predicted when rule 60 interacts with the environment and they display a much more dynamic range of values. In this episode, the model reached 390 steps, but even when the performance is worse and less consistent then rule 8, observing it play displays model which is more believable in its capabilities. It is not moving as much back and forth, and is constantly correcting itself when the pole starts to tilt. At the end of the episode, the pole almost tips over twice, but is able to save itself, although not for long. With longer training, the potential for improvement is more clear, and it would probably translate better to a more complex environment.

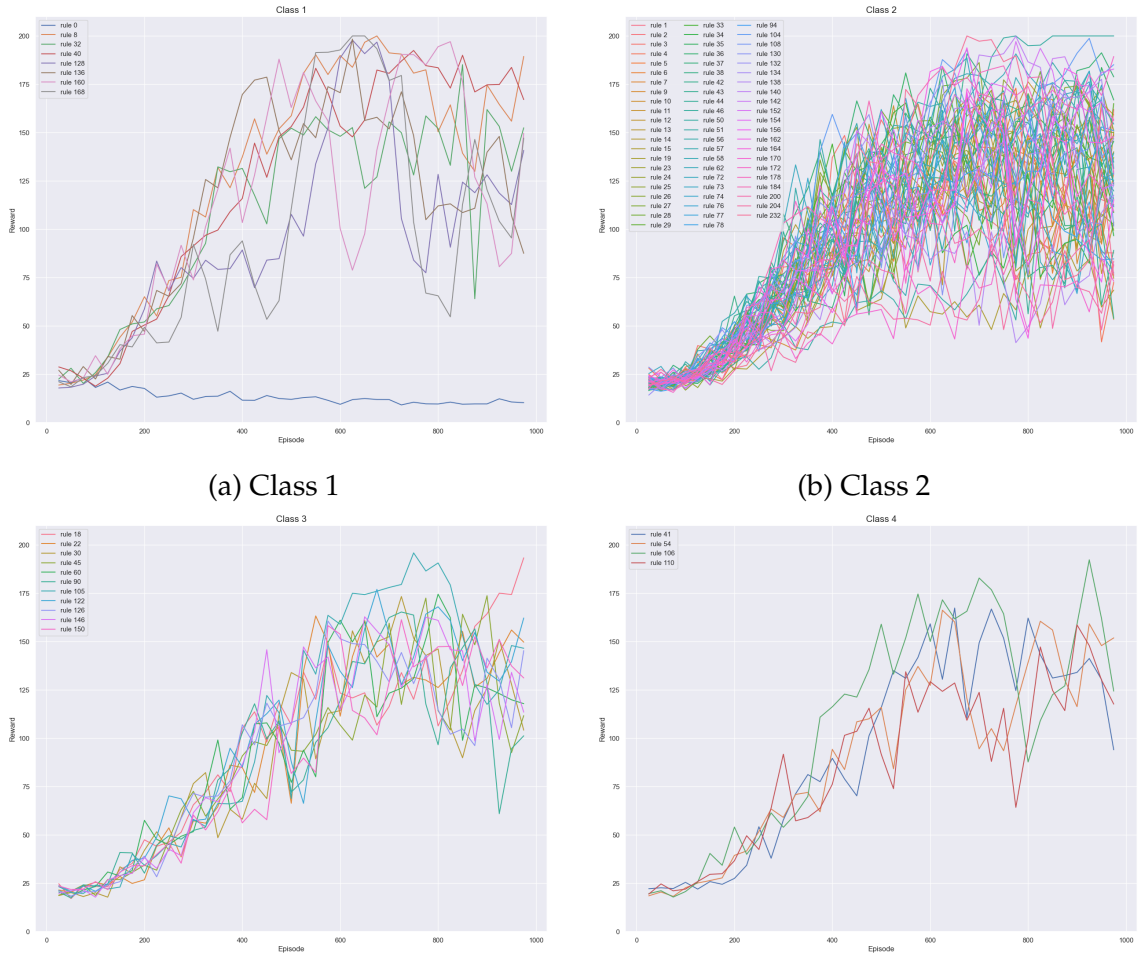


(a) Model with rule 8

(b) Model with rule 60

Figure 5.17: Plot of predicted Q-values for each action over one episode.

5.2.4 Experiment 6, small width



(a) Class 1

(b) Class 2

(c) Class 3

(d) Class 4

Figure 5.18: Training plot of models with narrow reservoir split into respective classes.

Table 5.3: Average score over 100 trials, including top score from all classes.

Rank	Rule	Class	Score
1	50	2	492.15
2	7	2	490.02
3	140	2	479.63
4	104	2	476.08
5	56	2	455.58
6	60	3	435.85
7	62	2	423.00
8	36	2	372.68
9	106	4	337.28
10	178	2	316.13
11	40	1	315.13

This run tests the opposite of the last experiment, smaller reservoir width with many iterations between updates. The parameters used was a width of 64, 15 iterations and rows as input and an accuracy of 16. This means that the update ratio would be 1, meaning that the entire reservoir would be overwritten when updated. The results from this experiment, displayed in Table 5.3, shows a overall good performance. In the top 11, a diverse group of rules is present, with representations from each class. A reason as to why so many rules show good performance with this setup is most likely from the update ratio being 1. Since the whole reservoir is overwritten, and thus no previous states can affect the current one, two similar observations will create the exact same state. This will result in more consistent states which can be easier for the agent to interpret. In a simple environment, such as this one, this can be quite favourable, especially for short training sessions. But in a more complex one, where memory is more important, this will probably not work as well.

5.2.5 Experiment 7, width, accuracy and partition relation

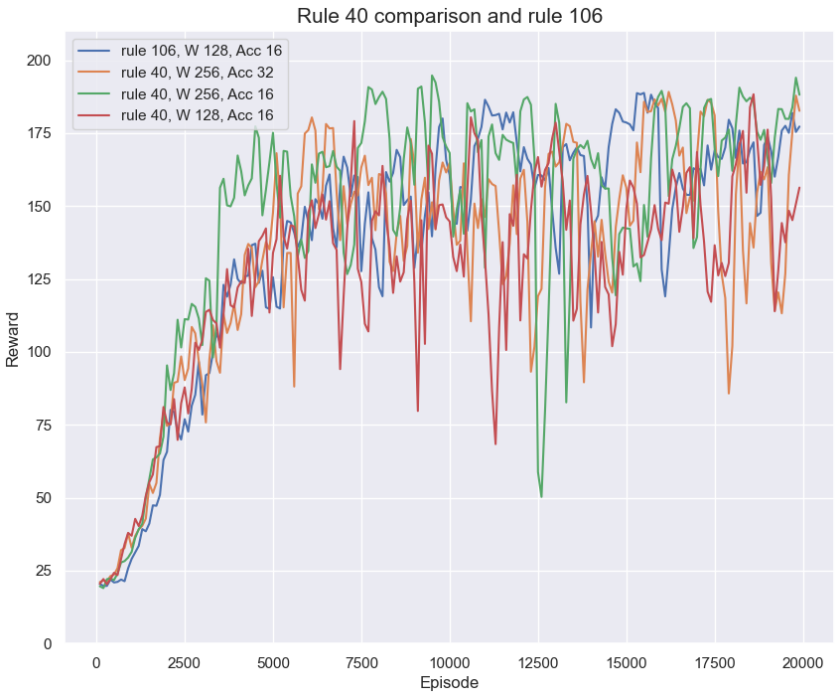


Figure 5.19: Comparison of rule 40 using different reservoir setups and rule 106. The plot of rule 40 with width 256 and accuracy of 16, shown in green, have 2 partitions.

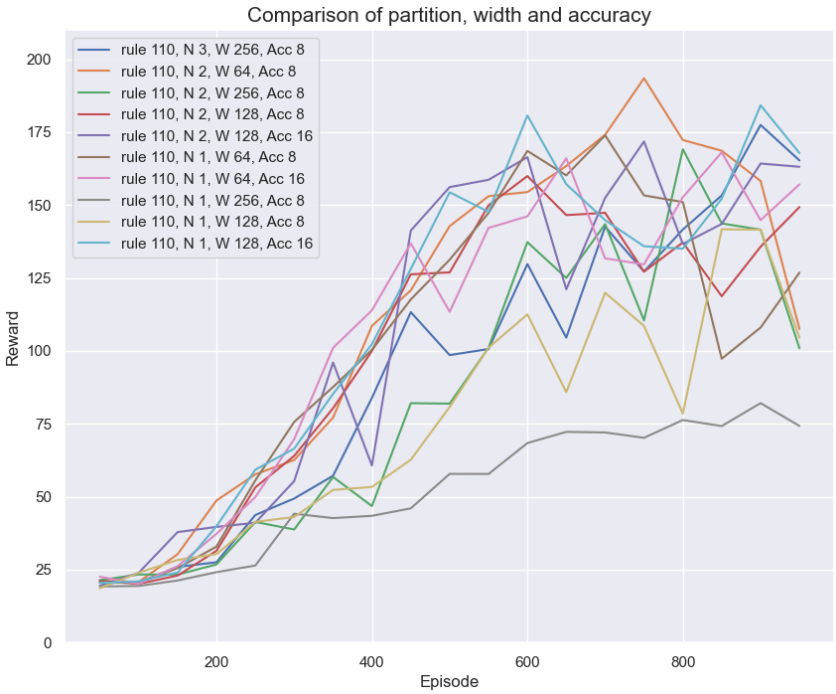


Figure 5.20: Comparison of rule 110 using different reservoir setups. The N in the plot name represents the number of partitions

In experiment 7, the focus was on the trade off between reservoir width, accuracy and partitions. Figure 5.19 shows four plots, three are of rule 40 using three different setups and one is of 106. All models used 4 iterations and inputs, and had the same update ratio of 1/2. They were train for 20,000 episodes, and show results that is not very distinguishable. For the first 5,000 episodes, the model using 2 partitions seems to be improving at a faster rate, but is caught up to shortly thereafter. After that, all of the models fluctuates significantly in performance. When the training session is this long, the difference between the setups seems to be insignificant. Even the difference between a class 1 and class 4 rule.

In Figure 5.20, we can observe how 10 reservoir using rule 110 responds to different setups. All the models in this plot used 10 iterations and 5 inputs. The model using 1 partition, 256 width and 8 accuracy shows significantly worse performance than any other. The update ratio of this one is 1/8. As experiment 5 showed, an update ratio this low generally performs poorly. Other poorly performing models from this experiment, usually had 8 accuracy and 1 partition.

Chapter 6

Summary & Conclusions

This thesis have explored the capabilities of using ECA as a RC to perform reinforcement learning tasks. To achieve this, an encoding strategy named linear translation was used to turn the observations returned from the environment into bits, focusing on interoperability rather than accuracy. The encoded bits was then placed in the reservoir following an observation mapping strategy, and iterated upon following a ECA rule. An agent could then read the state of the reservoir to determine what action to take.

Different observation mapping strategies in combination with reservoir parameters such as width, iterations, inputs and accuracy per observation could then be tested alongside a rule to find good and poor setups. This was mainly done using the cart pole environment.

In experiment 0, where no rule was applied to the reservoir, it was established that the model was incapable of balancing the pole consistently beyond 100 steps. All models with this setup showed behaviour of slowly falling to one side, and quickly falling to the other, depending on the starting position. Comparing this with the results from experiment 1, we find a clear improvement in performance when ECA rules are being applied in the reservoir.

From experiment 2, it is quite apparent that how the observations are mapped into the reservoir impact the performance significantly. Forcing all the converted bits from one observation to be directly adjacent to each other, by using the *fully_local* mapping method, the results show much poorer performance than if they were mapped randomly. A reason as to why this is, is because this method traps information in the centre cells, not allowing it to efficiently spread out to the rest of the reservoir. This is also the case with the *local_random* method, showing similar results as the *fully_local* method, but more importantly that the interactions between different observations makes a huge difference, and have a much higher likelihood of succeeding.

The *global_random* showed to produce the best results when compared to the other

two, but is not without its flaws. Since both *local_random* and *fully_local* exists as a subset of *global_random*, the *global_random* has a chance to create bad performing mappings. With larger reservoirs, the likelihood of creating mappings with these exact configurations becomes low, but other bad performing mappings still exists. Rule 8's performance in experiment 5 is a clear example of this, where the same setup was used to train multiple models using rule 8 but with different *global_random* mappings. This resulted in the models score ranging from being among the worst and the best.

Some testing was done in the more complex environment Bipedal Walker. The results from this was not satisfactory. A multitude of bad factors played into this. First, the model structure in general was suboptimal. Using discrete actions meant a much larger state-action space than necessary. Compressing the output to the defined continuous range of -1 to 1, e.g. by using a *tanh* layer would probably have been a better option. Second, the short training session of 1,500 meant that the models did not have much time to explore the environment, and since moving resulted in negative reward, all models developed a strategy along the lines of spreading the legs to not fall over and not move for the rest of the episode. Even with the small number of episodes, training still took a long time due to the inefficient implementation. Because of this, it was decided to not pursue this environment any further, instead moving back and continue testing in the cart pole environment. Third, the rules that were tested was picked due to their good performance in experiment 2. As . Lastly, as later experiments with the cart pole showed, higher update ratio improved overall performance, it is therefor likely that a reservoir with a smaller width could have further improved the results.

Experiment 5 and 6 tested the performance when the width was large and small, or rather when the update ratio was large and small. We observed a significant drop off in performance when the update ratio reaches 1/6. At this point, a good observation mapping is rarely created with the *global_random* mapping method. The updated cells become so sparse that rules with local interactions are so far apart that these rarely happen, and the performance resemble that of a linear network. Rules that are not dependant on local interactions, mostly class 3 and 4 rules, did not perform in this setting either. At least not in such a short training session. When increasing the update ratio to 1, models became significantly better at performing consistently. Most likely due to the state of the reservoir being consistent with the observation.

In the last experiment, the trade off between width, accuracy and partitions was tested. The result of which showed that in longer training session, the difference was not much. And in shorter ones, as the width gets larger, both increments in accuracy or partitions improved performance in a equal manner. As it seems, the most important element is to keep the update ratio in between 1/3 and 1.

When comparing the results with a neural network implementation using the exact

same setup, we observe that the neural networks struggles to reach same level of performance when there is a low numbers of episodes. For training sessions with higher number of episodes, the reservoir model displays a much faster learning curve in the beginning, but they finish roughly at equal performance.

The work presented in this theses shows potential, but there is still much more investigation need.

There is uch work can be done to improve upon the model presented in this theses. The bla bla.

observation mappings that are less susceptible to unlucky randomness. A more sophisticated way of placing the bits in the reservoir.

using multiple partitions are in many cases better than increasing the accuracy.

saw sudden drop or heavy fluctuation in performance

- Fast learning
- time in 2. generation
-
- CHECKLIST:
- numbers formatted correctly
- Figures where relevant, and not shit
- All links and ref are correct
-
-

6.1 Future work

- Testing different update techniques, clusters of bits, bits separated by n spaces, at different iterations, multiple reservoirs.
-
-

Bibliography

[1] Adrià Puigdomènech Badia et al. *Agent57: Outperforming the Atari Human Benchmark*. 2020. arXiv: 2003.13350 [cs.LG].

[2] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.

[3] Hanten Chang and Katsuya Futagami. *Reinforcement Learning with Convolutional Reservoir Computing*. 2019. arXiv: 1912.04161 [cs.NE].

[4] Francois Chollet et al. *Keras*. 2015. URL: https://github.com/fchollet/keras.

[5] Martin Gardner. 'Mathematical Games'. In: *Scientific American* 223.4 (1970), pp. 120–123. DOI: 10.1038/scientificamerican1070-120.

[6] Tom Eivind Glove et al. 'Investigating rules and parameters ofReservoir Computing with ElementaryCellular Automata, with a criticism of rule90 and the 5-bit Memory Benchmark.' In: *Is under peer review* ().

[7] *The Dynamical Landscape of Reservoir Computing with Elementary Cellular Automata*. Vol. ALIFE 2021: The 2021 Conference on Artificial Life. ALIFE 2022: The 2022 Conference on Artificial Life. 102. July 2021. DOI: 10.1162/isal_a_00440. eprint: https://direct.mit.edu/isal/proceedings-pdf/isal/33/102/1930018/isal_a_00440.pdf. URL: https://doi.org/10.1162/isal%5C_a%5C_00440.

[8] Matthew Hausknecht and Peter Stone. *Deep Recurrent Q-Learning for Partially Observable MDPs*. 2017. arXiv: 1507.06527 [cs.LG].

[9] Sepp Hochreiter and Jürgen Schmidhuber. 'Long Short-term Memory'. In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[10] Chris G. Langton. 'Computation at the edge of chaos: Phase transitions and emergent computation'. In: *Physica D: Nonlinear Phenomena* 42.1 (1990), pp. 12–37. ISSN: 0167-2789. DOI: https://doi.org/10.1016/0167-2789(90)90064-V. URL: https://www.sciencedirect.com/science/article/pii/016727899090064V.

[11] Genaro J. Martinez. *A Note on Elementary Cellular Automata Classification*. 2013. arXiv: 1306.5577 [nlin.CG].

[12] Volodymyr Mnih et al. *Asynchronous Methods for Deep Reinforcement Learning*. 2016. arXiv: 1602.01783 [cs.LG].

[13] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[14] Adam Paszke et al. 'PyTorch: An Imperative Style, High-Performance Deep Learning Library'. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[15] Tom Schaul et al. *Prioritized Experience Replay*. 2016. arXiv: 1511.05952 [cs.LG].

[16] Julian Schrittwieser et al. 'Mastering Atari, Go, chess and shogi by planning with a learned model'. In: *Nature* 588.7839 (Dec. 2020), pp. 604–609. DOI: 10.1038/s41586-020-03051-4. URL: https://doi.org/10.1038%5C%2Fs41586-020-03051-4.

[17] Gouhei Tanaka et al. 'Recent advances in physical reservoir computing: A review'. In: *Neural Networks* 115 (2019), pp. 100–123. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2019.03.005. URL: https://www.sciencedirect.com/science/article/pii/S0893608019300784.

[18] Stephen Wolfram. 'Universality and complexity in cellular automata'. In: *Physica D: Nonlinear Phenomena* 10.1-2 (1984), pp. 1–35.

[19] Ozgur Yilmaz. *Reservoir Computing using Cellular Automata*. 2014. arXiv: 1410.0162 [cs.NE].