The Pennsylvania State University
The Graduate School

NEURAL NETWORKS AND SELF-ORGANIZING CELLULAR AUTOMATA

A Thesis in
Computer Science
by
Matthew A. Robinson

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2021

The thesis of Matthew A. Robinson was reviewed and approved by the following:

Thang N. Bui
Associate Director, School of Science, Engineering, and Technology
Thesis Advisor

Susan Lemieux Eskin
Assistant Teaching Professor of Physics

Linda Null
Associate Professor of Computer Science

Sukmoon Chang
Associate Professor of Computer Science

Jeremy Blum
Associate Professor of Computer Science

Hyuntae Na
Assistant Professor of Computer Science

Hien Nguyen
Assistant Professor of Computer Science

Md Faisal Kabir
Assistant Professor Of Computer Science

Abstract

This thesis presents a framework that uses neural networks to evolve local behaviors within cellular automata to form complex collective behavior. Our framework shows how starting with an initial state and a target state for a cellular automaton, we can evolve a neural network that computes translationally invariant local interactions to form that target state. We demonstrate this framework on various test images as target states to understand its behavior and to develop optimizations. Through this, we present several novel extensions and improvements to this framework including a gradual and companion learning approach akin to transfer learning. Furthermore, we test this framework on a related problem from microbiology: predicting protein interactions from global behavior, and propose several other problems to investigate.

Table of Contents

List of Figures

List of Tables

List of Symbols

A_t	State of a cellular automaton at time t, p. 4
$A_t^{i,j}$	Neighborhood around cell i, j of state A_t, p. 4
Δ	Transition rule for a cellular automaton, p. 4
Δ^k	Transition rule applied k times, p. 4
T	Target state for the rule discovery problem, p. 7
M_i	ith kernel for the Moore filter, p. 10
Δ_F	Convolution function, p. 10
M_5	Kernel for the Identity filter, p. 10
S_x	Sobel kernel for the x axis, p. 10
S_y	Sobel kernel for the y axis, p. 10
K_i	ith kernel for a filter, p. 10
W	Neural network as a function, p. 12
Adam	Adam optimizer [1], p. 12
RMSE	Root mean squared error, p. 9
$\hat{\epsilon}$	Target error, p. 12
L	Laplacian filter, p. 23

Acknowledgments

I would like to thank Dr. Thang Bui for his endless guidance and wisdom. A number of major optimizations and ideas explored in this thesis are a result of his questioning and curiosity. I would also like to thank the thesis reader Dr. Susan Eskin and the committee members for reviewing this thesis, and Dr. Donald Knuth for inventing TEXwhich was used to typeset this thesis. Furthermore, I would like to thank my friends, my employer, my family, and my inspirational partner for all their support.

Chapter 1

Introduction

The fundamental units of study in this thesis are *self-organizing systems*. Self-organizing systems are famously difficult-to-understand structures that are ubiquitous in our world. Briefly, self-organizing systems have global behaviors that are emergent from interactions purely at a much lower level. The individuals that make up the system (often called *agents*) are usually simple and identical, yet yield complex global behavior when they are brought together as a system.

Examples of self-organizing systems include snowflakes, ant colonies [2], embryos, traffic, and more. Consider the ant colony. The agents of the system are the ants. They are nearly identical, relatively simple units that interact only with their immediate surroundings, and are not governed by any centralized authority. Despite this, they can construct very complex colony structures. Similarly, we see this with snowflake formation: complex fractal patterns delicately arranged emerging purely from interactions between identical and simple water molecules.

We focus on one self-organizing construction called a cellular automaton. A *cellular automaton* (CA) is a grid of cells with local transition rules between nearby cells. Once an initial state for a CA is picked, it transitions autonomously based on those transition rules. Many cellular automata, such as John Conway's *Game of Life* [3], exhibit remarkably complex behaviors from simple transition rules. This makes CAs a brilliant example of self-organizing systems. It is generally quite difficult to establish connections between the local rules and the global behavior of CAs. It is especially difficult to construct local rules that produce desired global behaviors. This is the problem we use neural networks to investigate: given an initial state and a target state for a cellular automaton, find the transition rule that will cause the CA to form that target state from the initial state. By encoding the transition rules as a neural network, we can evolve the network to converge on a transition rule that generates the target state. Each cellular automaton, paired with its evolved transition rule, is a self-organizing machine that, purely from local interactions, can perform a kind of 'collective cognition', just like the self-organizing systems mentioned earlier.

Understanding this automatic 'local rule discovery' framework has mathematical and philosophical

implications in other fields. Usually, we try to understand dynamical systems by identifying the rules at the lowest level *from first principles* and deduction; when we simulate the system with these rules we compare how close it gets to some observations then make necessary adjustments. In this thesis we do this in reverse: we have some global behavior we want to understand, and we have a computer discovery the local rules almost entirely on its own. It is astounding that even quite small neural networks are capable of such a difficult and seemingly intractable task. Self-organizing systems are ubiquitous and have seen applications in industrial ecology and supply chain management [4], biology and developmental evolution [5], machine learning [6], and more. Automatic rule development schemes like these have the potential to help us understand these systems from a completely different angle and even overturn the traditionally slow process of deriving local interaction rules by hand.

In this thesis, we extend the recent work of Mordvintsev et al. [7] presented in the open *Distill* journal, an interactive online journal supported by Google and OpenAI to spread scholarly machine learning research to a wider audience. Mordvintsev et al. use the same kind of model (cellular automata backed by neural networks) to replicate small images from Google's emoji set. In this thesis one can find a detailed explanation of these models and their applications, a replication and verification of results from the *Distill* paper, a newly implemented model that simplifies and extends the original work, and a variety of experiments that yield significant performance benefits over the original work.

In Chapter 2 we introduce the background information needed to understand the formal definition of the rule discovery problem for cellular automata. In Chapter 3 we present the technique we use to construct the neural network and evolve it to form a set of well-known test images. In Chapter 4 we present the results from using this technique as well as several optimizations and extensions. We finish with a summary of the results and ongoing research in Chapter 5.

Chapter 2

Background & Formalisms

In this chapter, we introduce background materials including cellular automata, the rule discovery problem, and neural networks. Cellular automata provide the backbone we need to formally study self-organizing behavior through the rule discovery problem, and neural networks provide the machine learning power needed to investigate the problem.

2.1 Cellular Automata

A well-known model for self-organizing behavior is the *cellular automaton*. A cellular automaton is a grid of cells governed by a local transition rule that applies uniformly to every cell in the grid. Cellular automata (CAs) have been a constant source of creative and scientific potential since their conception in the minds of some of the greatest researchers of the time. They were popularized by Conway's Game of Life [3], a particular type of cellular automata with simple rules but exceptionally complex global behavior. An example configuration of the Game of Life can be seen in Figure 2.1

The *transition rule* takes each cell from one state to the next based on each cell's immediate neighbors. Each cell stores a number that encodes its current state (in the Game of Life, these are 1 for 'alive' and 0 for 'dead'.) Because the cells are discrete squares, we say that CAs are *spatially discrete*. Additionally, because each cell is only affected by its state and the states of its immediate neighbors, we say that CAs are *local*. Furthermore, the rules that the cells obey are the same across the grid. The only thing that makes one cell act differently from another is either its state at that moment in time or the states of its neighbors. This makes CAs (or more specifically, their transition rules), *translationally invariant*. This is a **very** restrictive feature that ensures we have entirely local rules as described in the introduction, and it dramatically improves our hope of modeling real-world self-organizing systems. This makes sense because almost all credible hypotheses and theories we make about the world are invariant in this way: while the world may look different in distant places, we can at least assume the laws of physics

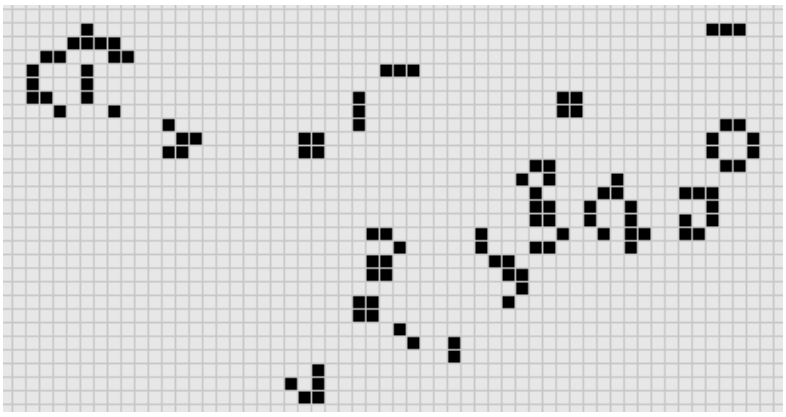


Figure 2.1: *A snapshot of one instance of Conway's Game of Life. Cells colored* ***black*** *are considered 'alive' and cells colored* ***white*** *are considered 'dead'. Each cell will transition to the next snapshot in time by the set of transition rules that make* Conway's Game of Life. *Image credit: Chris Lipa, Cornell University*

are the same everywhere.

Cellular automata have numerous advantages for simulating all kinds of mathematical machinery: (a) they are often easier to comprehend and store than smooth, continuous grids and functions, (b) there are fewer problems with mathematical singularities and other pathologies on discrete grids as opposed to continuous spaces, and (c) they fit within finite spans of memory without forcing additional compromises over any other data storage scheme.

Cellular automata do come with drawbacks, including stability and numerical issues [8], but this will not be a problem for any of the research goals examined in this thesis. It should not be too much of a surprise that CAs are being investigated for foundational physics and other natural science applications since they conveniently encapsulate systems of discrete autonomous agents and there is strong multi-disciplinary evidence that our world can be approximated well by discrete agents.

2.2 Formal Definition

Formally, a *cellular automaton* is an $m \times n$ array A of real numbers whose content changes over time according to a transition rule Δ. We denote by A_t the state of A at time t, i.e., the contents of A at time t. Also, denote by $A_t^{i,j}$ the 3×3 neighborhood of A_t around cell $A[i, j]$.

The state of A is updated by applying the transition rule to 3×3 neighborhoods of A as follows.

$$A_{t+1}[i, j] \leftarrow A_t[i, j] + \Delta(A_t^{i,j}),$$

where $1 \leq i \leq m$ and $1 \leq j \leq n$. The case when the transition rule is applied to cells on the boundary is discussed in the next section. Note that the transition rule does not explicitly depend on t. For convenience, we also write $\Delta(A_t)$ to denote A_{t+1}.

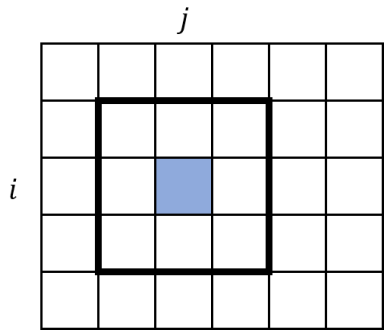


Figure 2.2: *An illustration of the neighborhood $A_t^{i,j}$ around cell $A[i,j]$.*

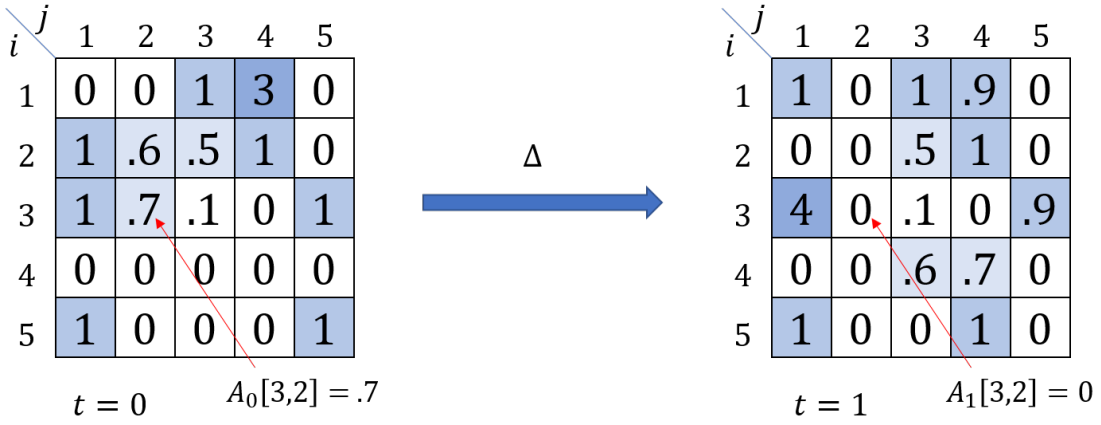


Figure 2.3: *An illustration of a real-number-valued cellular automaton on a 5×5 grid at two adjacent moments in time.*

Additionally, we denote by Δ^k the application of Δ k times, and Δ^k can be defined as follows.

$$\Delta^1(A_t) = \Delta(A_t),$$
$$\Delta^{k+1}(A_t) = \Delta(\Delta^k(A_t)) \quad \text{for } k \geq 1.$$

We focus on a specific type of transition rule called an *NN-transition rule*, which is a transition rule encoded by a neural network. More detail on neural networks is described in Section 2.4.

Note that Δ acts on neighborhoods that are of fixed size, namely 3×3, so this model has *locality*. Additionally, since Δ acts on neighborhoods of values and not on coordinates explicitly, this model is *translationally invariant*, i.e., the rule is applied uniformly to all of the cells in the grid. Each value at $t+1$ is computed solely using the values at time t. This feature makes our CAs *synchronous*; this feature can be thought of as having the cells transitioning 'all at once'. There are drawbacks to this approach, including making it exceptionally more difficult to identify or stabilize important properties of the CA [9], but it comes with the advantage that we do not have to worry about the order in which we choose to compute the new values, making it easier

to implement synchronous CAs on parallel hardware like GPUs.

Since A is an array of real numbers, we call these *continuous* cellular automata to distinguish them from cellular automata over a discrete set of values. These continuous cellular automata can be easily extended to store multiple values per cell by storing vectors of real numbers instead of real numbers. We use this extension in Chapter 3 to create cellular automata that self-organize into target images.

2.3 Boundary Strategies

In this section, we discuss different strategies for applying the transition rule to cells that are on the boundary. In these cases, the 3×3 neighborhood does not lie entirely within the grid. To deal with this problem, we pad the grid with a *virtual boundary* one cell thick. The strategy will dictate how the cells in the virtual boundary are initialized at the beginning of each iteration. The four strategies we use are the following.

- **Torus**: We imagine the grid has the form of a torus. That is, cells on the left and right edges are considered adjacent, and cells on the top and bottom are considered adjacent. We fill the cells in the virtual boundary by copying the cell value opposite to them on the grid.
- **Mirror**: Fill each cell of the virtual boundary with the value of its closest neighbor in the grid.
- **α-Fill**: Fill each cell of the virtual boundary with constant α.
- **Random**: Fill each cell of the virtual boundary with a number selected at random from the interval $[0, 1]$.

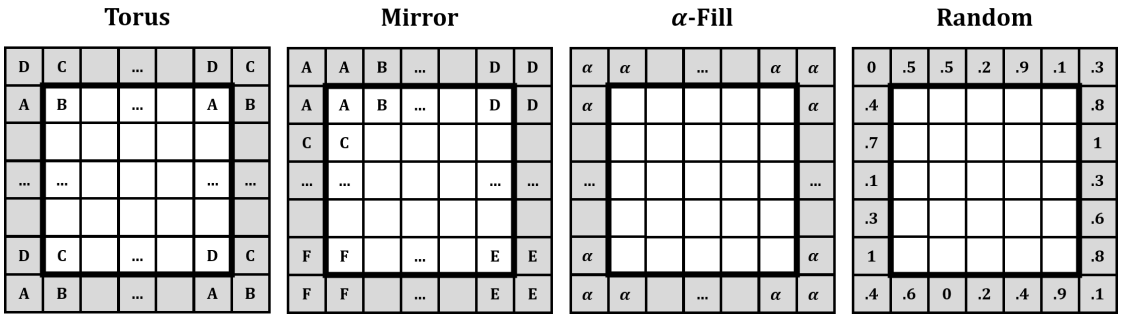


Figure 2.4: *An illustration of the four different boundary strategies applied to a 5×5 CA.*

2.4 Neural Networks

We use neural networks to encode transition functions that can be automatically evolved by an algorithm. An *artificial neural network* (or just *neural network* in this thesis) is a computational

model inspired by biological neural networks. Neural networks are composed of computational units called *neurons* connected by wires. A neural network is a black box that takes in a set of numbers and does some processing to it to produce some other numerical output. The neurons are arranged in layers according to the order that this processing is computed. The *input layer* receives the set of input data and passes it to any number of *hidden layers* which do most of the heavy processing. The final layer, called the *output layer* contains the processed data for any given input [10].

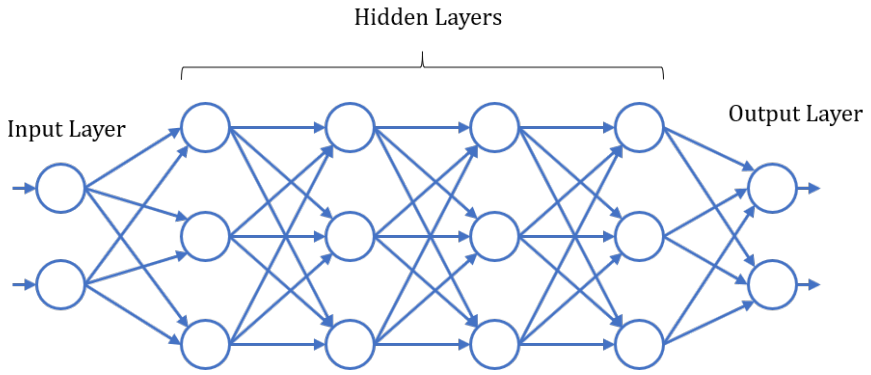


Figure 2.5: *An example of a small neural network with six layers (four hidden) and sixteen neurons. The edges between neurons feed activation values through the network from input to output. The so-called 'hidden layers' are referred to that way because they act largely as a black box and we only interact with the impact they have on the output layer.*

The most important aspect of a neural network is its capacity to automatically learn to process data in the desired way. We do this by fixing a *loss function*, a function that measures the difference between the outputs from the neural network and the outputs we want instead. By representing the strength of each wire by a weight, we can use an optimizer to change the weights to minimize this loss function. We use an optimizer called *Adam* [1]. Adam takes as input a neural network, a loss function, a target prediction, and an actual prediction from the neural network and returns a new neural network with adjusted parameters to minimize the loss between the target and actual predictions. We can use neural networks to encode the transition rules for cellular automata, and in turn, use this optimization technique to evolve complex self-organizing behaviors. A transition rule that uses a neural network to perform its computation is called an *NN-transition rule*.

2.5 The Rule Discovery Problem

We are now ready to talk about the main focus problem of this thesis called the *rule discovery problem*.

Rule Discovery
Input: A cellular automaton A, an initial state A_0, an integer k called the *lifetime*, and a target state T.
Output: An NN-transition rule Δ such that $\Delta^k(A_0) = T$.

This problem is usually difficult to answer analytically even for the simplest of systems; furthermore, there are closely related problems that have been proven to be *unsolvable* in general. For example, since Conway's Game of Life can simulate universal Turing Machines [11], finding whether an arbitrary initial state will eventually reach a dormant or repeating state is equivalent to the Halting Problem, which is known to be undecidable. However, by using neural networks to encode Δ, we will show for some systems it is possible to evolve the transition rule that forms the target state T to 1% error and less. We expect these results to transfer to many other systems.

Translational invariance is **vital** in the context of the rule discovery problem. Suppose we allow the transition rule to depend explicitly on the coordinates of the cells and call this non-translationally-invariant rule Δ'. For clarity, suppose Δ' only receives coordinates and the current cell value. This transition rule acts on A as follows.

$$A_{t+1}[i,j] \leftarrow A_t[i,j] + \Delta'\left(A_t[i,j], i, j\right).$$

To solve the problem with this type of transition function, all we have to do is ensure that every cell immediately transitions to the target value for that location.

$$\Delta'(A_t[i,j], i, j) = T[i,j] - A_t[i,j],$$

where T is the target state. With coordinate information in Δ', this problem is no longer about rule discovery and has a trivial solution. Thus, keeping the transition rule translationally invariant is key to ensuring our investigation studies non-trivial self-organizing behavior.

Chapter 3

Rule Discovery for Evolving Images

In this chapter, we present an image-based version of the rule discovery problem introduced in Chapter 2 and the algorithm we use to investigate the problem. We let the cells of our cellular automata take on vectors of real numbers and map these to pixels of an image. We use a variety of well-known test images as target states and we use neural networks to automatically generate compatible transition functions to form those target states. This gives us a variety of the rule discovery problem in the context of images. In this chapter, we use the term "state" and "image" interchangeably since we map images to states of a cellular automaton. Denote by $\mathrm{RMSE}(A, B)$ the *Root Mean Square Error* between states A and B of size $m \times n$; this error is computed as follows.

$$\mathrm{RMSE}(A, B) = \sqrt{\frac{1}{mn} \sum_{i,j} \|A[i, j] - B[i, j]\|^2}.$$

The rule discovery problem for images is the following.

Rule Discovery (for images)
Input: A vector-valued cellular automaton A, an initial image A_0, an integer k called the *lifetime*, a target image T, and a target error $\hat{\epsilon}$.
Output: An NN-transition rule Δ such that $\mathrm{RMSE}(\Delta^k(A_0), T) \leq \hat{\epsilon}$.

We call our algorithm to solve this problem the **RuleEvolver**. Our algorithm takes an initial image, a target image, a lifetime, and a target error, and will produce an NN-transition rule that contains the necessary machinery to take the initial image to the target image after k steps in a cellular automaton within the given target error. See Figure 3.1 for a summary of the RuleEvolver algorithm.

3.1 Channels

To map images to cellular automata and back, we store vectors of real numbers in every cell instead of real numbers. The first three components of these vectors store the red/green/blue

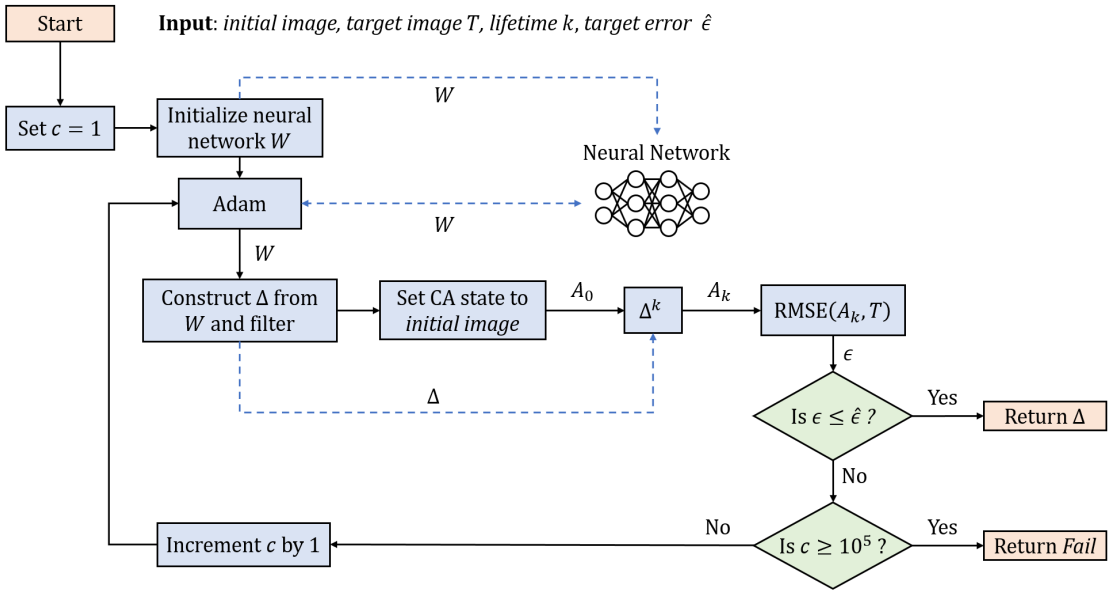


Figure 3.1: *Abstract representation of the rule evolution algorithm. Given the problem inputs we run our RuleEvolver to obtain an NN-transition rule that takes the initial image as a cellular automaton to the target image.*

color data for the images and the rest of the vector is used for additional state information. We take on a standard color space of $[0, 1]$ where 0 means no color and 1 means maximum color on that component. In image processing, the components are often referred to as *channels*. We use this terminology to refer to a specific component of the vectors, e.g., the red channel, or the green channel. We use the term *auxiliary channels*, described in Section 4.3, to refer to channels that are used for additional state information. Formally, we say A is an $m \times n$ array of *vectors*. Our transition rule still functions in the same way, now taking in neighborhoods of vectors instead of real numbers, producing vector-valued changes in cell values.

3.2 Convolutional Filters

Recall that the transition rule operates on 3×3 neighborhoods of the grid. Because we use neural networks to encode the transition rule, these neighborhoods are the input to a neural network. However, many neural networks perform significantly better on data that has been pre-processed to eliminate redundant or unimportant data. To allow this kind of pre-processing, we apply convolution to the input. *Convolution*, denoted by Δ_F, is an operation that is applied on an array in a manner similar to a transition rule. Convolution takes as input an array of data and a set of 3×3 matrices called *kernels*. Let A_t by the state of an $m \times n$ cellular automaton A at time t. The content of cell $A_t[i, j]$ after applying convolution Δ_F with kernels $K_1, \ldots, K_l$ is a

vector of length l defined as follows.

$$\Delta_F(A_t^{i,j}, K_1, K_2, \ldots, K_l) = \begin{bmatrix} A_t^{i,j} \cdot K_1 \\ A_t^{i,j} \cdot K_2 \\ \vdots \\ A_t^{i,j} \cdot K_l \end{bmatrix}, \tag{3.1}$$

where $A_t^{i,j}$ denotes a 3×3 neighborhood around cell $A[i,j]$ and each K_i is a 3×3 kernel. The dot product between a 3×3 neighborhood N and a kernel K is computed as follows.

$$N \cdot K = \sum_{1 \le i,j \le 3} N[i,j]K[i,j].$$

Just like a transition rule, we denote by $\Delta_F(A, K_1, \ldots, K_l)$ the result of applying convolution to every neighborhood in A. In other words, if A is an $m \times n$ array, then $\Delta_F(A, K_1, \ldots, K_l)$ is an $m \times n$ array where each of the cells is an l-vector defined by Equation 3.1 above. We use four different sets of kernels in this thesis. We refer to convolution with a specific set of kernels as a *filter*.

1. **Identity Filter:** In the identity filter, a cell $A[i,j]$ is unchanged in the filtered result. Its kernel denoted by M_5 and is defined as follows.

$$M_5 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

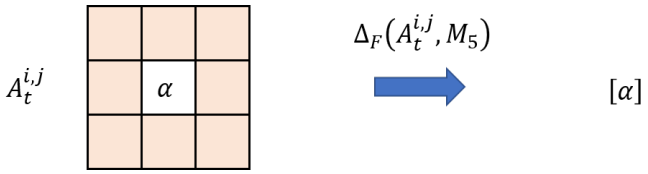


Figure 3.2: *Action of the Identity filter on a neighborhood* $A_t^{i,j}$.

2. **Moore Filter:** The Moore filter converts each cell into a vector storing its 3×3 neighborhood. It uses the following set of nine kernels.

$$M_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad M_2 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \ldots \quad M_9 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

3. **Sobel Filter:** The *Sobel filter* uses two kernels, S_x and S_y, to compute the overall change

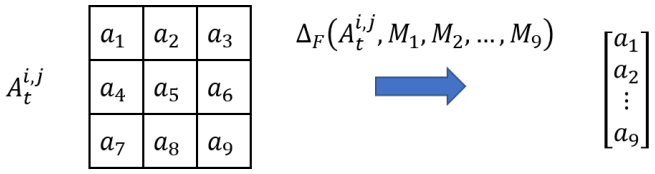


Figure 3.3: *Action of the Moore filter on a neighborhood $A_t^{i,j}$.*

in the cell values from left to right or top to bottom respectively. A cell $A[i, j]$ is converted into a 2-vector describing this change in each axis, estimating the overall change in values from left-to-right or top-to-bottom respectively.

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

Note $S_y = S_x^T$, where S_x^T denotes the matrix transpose of S_x.

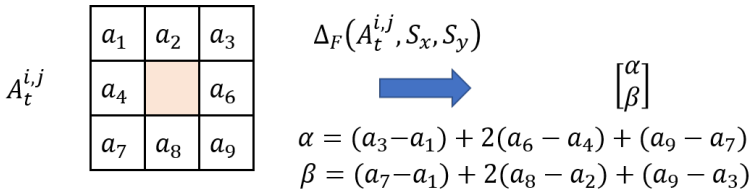


Figure 3.4: *Action of the Sobel filter on a neighborhood $A_t^{i,j}$.*

4. **Sobel-state Filter:** The *Sobel-state filter* uses the Identity filter and the Sobel filter together. Its set of kernels is $\{M_5, S_x, S_y\}$.

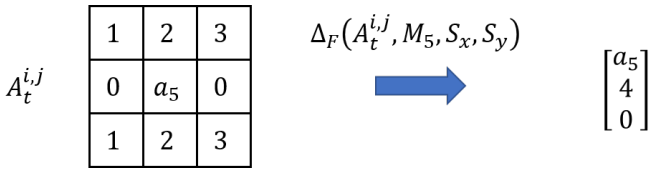


Figure 3.5: *Example action of the Sobel-state filter on a neighborhood $A_t^{i,j}$. The center cell is mapped to the first component of the output vector, and the change of the cell values along the x and y axes are mapped to the second and third component respectively, as defined in Figure 3.4.*

3.3 Evolving NN-Transition Rules

To evolve NN-transition rules we need to determine the structure of the neural network that encodes Δ. This process is simple: we need one input neuron for every component of the pre-processed neighborhoods, i.e., a neuron for each component that comes out of the filter, and an

output neuron for each channel of the cellular automaton. For a neural network W, denote by $W(v)$ the computation of the neural network on an input vector v. We use the Adam optimizer [1] described in Section 2.4 to adjust the parameters of W to fit a target prediction. Adam takes as input a neural network, a loss function, a prediction, and a target prediction. It returns a new neural network with adjusted parameters. To ensure that the algorithm always terminates, we keep track of the number of iterations of the optimization process with c and stop the algorithm if the number of iterations exceeds 10^5. For each iteration, we run the optimizer, construct a transition rule, and run the CA before computing the new loss. The basic RuleEvolver algorithm follows.

Algorithm: **RuleEvolver** $(A_0, T, k, \hat{\epsilon})$

Input: A_0 and T are images, k is an integer, $\hat{\epsilon} \geq 0$

Output: An NN-transition rule Δ where $\text{RMSE}(\Delta^k(A_0), T) \leq \hat{\epsilon}$ or *Fail* message

```
Initialize a 3-channel CA with boundary strategy 0-Fill
Initialize a neural network W
ϵ ⟵ ∞
A ⟵ A_0
c ⟵ 1
while ϵ > ε̂ do
    if c > 10^5 then
        return Fail
    W ⟵ Adam(W, RMSE, A, T)
    Construct new NN-transition rule Δ such that Δ(A^{i,j}) = W(Δ_F(A^{i,j}, M_5, S_x, S_y))
    A ⟵ A_0
    for t = 1 to k do
        A ⟵ Δ(A)
    ϵ ⟵ RMSE(A, T)
    c ⟵ c + 1
return Δ
```

Algorithm 1: Finding an NN-transition rule Δ for an initial image A_0, a target image T, a lifetime k, and a target error $\hat{\epsilon}$.

There are some free parameters of this construction such as the number of hidden neurons in the neural network and the learning rate for the neural network optimizer. In the next chapter, we will present experiments on these free parameters and also explore modifications such as additional channels in the CA, different filters, and other boundary strategies.

Chapter 4

Results and Improvements

This chapter is a comprehensive report of the experiments on the RuleEvolver described in Chapter 3. We describe some important emergent behaviors of the model and use them to make optimizations. We show how the problem difficulty scales with image size and complexity. We end with some novel extensions to the baseline rule evolution strategy and an application to biology.

We use *Tensorflow 2.0* for constructing and training the neural networks used throughout this thesis. These experiments were run on a machine with a Intel i7-4700K quad-core CPU (turbo enabled) and dual-channel DDR3 RAM at 1600 MT/s. Tensorflow was not compiled to use AVX or SIMD for these experiments. We use 26 test images each around 100×100 pixels with varying types of image content, including color/black-and-white data, people, places, and things. To keep running time low enough, we downsample these images to smaller sizes such as 32×32 pixels for use as target images. See Section 4.9 to see how the size of the target image affects the running time of the RuleEvolver. These images are provided by the University of Southern California's Signal and Image Processing Institute (URL: `http://sipi.usc.edu/database/`) and by Fabien Petitcolas for research use. Full credit and authorship information is reproduced in Section A.2, in the Appendix.

4.1 Baseline Results

In this section, we present our baseline results using the RuleEvolver described in Chapter 3. Figure 4.1 demonstrates the time it takes for the RuleEvolver to converge on an NN-transition rule for each 32×32 target image. Other initial states are tested and the algorithm converges similarly for those. More details about initial states can be found in Section 4.5. In the rest of this chapter, we present some emergent behaviors of the model and justify the use of the free parameters given in these samples. Furthermore, we present optimizations that dramatically reduce the running time across all of the test images.

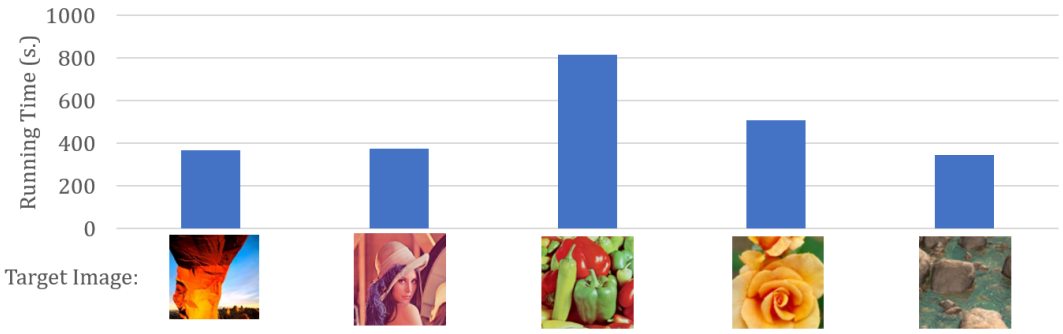


Figure 4.1: *Baseline results before any optimizations for the RuleEvolver. Along the horizontal axis, one can see the target states tested. These data are averages over 40 runs for each target image. The initial state is a blank white image. The parameters used in this experiment are: target error - 1%, image size - 32×32 pixels, learning rate - 3.5×10^{-3}, network size - 1 hidden layer with 256 neurons.*

4.2 Nucleation Sites

A core emergent feature of the rule discovery problem is the concept of nucleation sites. Suppose we have a state A with constant values. We call this a *uniform state*. Suppose we have a transition rule Δ and the Torus boundary strategy. Because Δ is translationally invariant, when it is applied to a uniform state the result must also be a uniform state. Suppose we change a single cell of A such that A is no longer a uniform state. Now any neighborhood that contains the changed cell can proceed differently and the next state can also be non-uniform. This single cell acts as an example of what we call a *nucleation site* because it tends to catalyze the formation of non-trivial structures within the cellular automaton.

Notice that for some boundary strategies, the entire virtual boundary can act as a nucleation site. Suppose we have a uniform state filled with constant C but we use α-Fill for $\alpha \neq C$. Then the virtual boundary acts like the changed cell in the previous example. A boundary strategy that can be a nucleation site is called a *nucleating boundary*, and a boundary strategy that cannot is called a *non-nucleating boundary strategy*. In Table 4.1, one can see whether a

Table 4.1: *Table of boundary strategies and whether they are nucleating or not under the filters*

	Identity filter	Sobel filter	Other filters
Torus	No[a]	No	Never[c]
Mirror	No	No	Sometimes[d]
α-Fill	No	Yes[b]	Sometimes
Random	No	Yes	Sometimes

[a] Torus with the Identity filter is non-nucleating,
[b] α-Fill is nucleating with the Sobel filter,
[c] Torus is non-nucleating for all possible filters,
[d] Mirror is nucleating for some filters.

particular boundary strategy acts as a nucleation site under each filter. Note that if the target state is non-uniform, then either the initial state must be non-uniform or the boundary strategy

must be nucleating. This follows trivially from the fact that a uniform state stays uniform unless we have a nucleating boundary. This justifies our use of the 0-Fill boundary strategy in the basic RuleEvolver. With it, we can be sure that there is always some nucleation site with the Sobel-state filter, so even uniform initial states will work with it. We can see what nucleation sites look like in Figure 4.2. Here both the initial state and the boundary act as nucleation sites. Compare this to Figure 4.3; here, with a non-nucleating boundary strategy, the only place where non-uniform data forms is from the nucleation site in the center of the initial state.

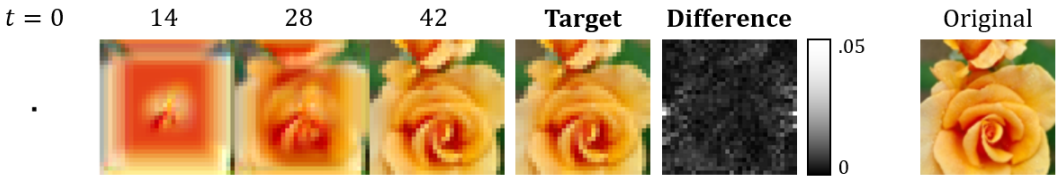


Figure 4.2: *Sample of an evolved 'Rose' CA. The initial state, a centered black dot, is seen on the left. The grid size is* 32×32 *and the lifetime is 42 which is 10 more than the grid length, ensuring the growing data has just enough time to reach the entire grid. The target image is downsampled from the original image seen on the right. The difference image shows the errors between the target and the state at* $t = 42$*. The error is calculated by the difference in the colors at each cell from the interval* $[0, 1]$*. The errors are normalized to the range given to the right of the difference image to exaggerate the errors so they are easier to see. Here we use a Sobel-state filter with the 0-Fill boundary strategy.*

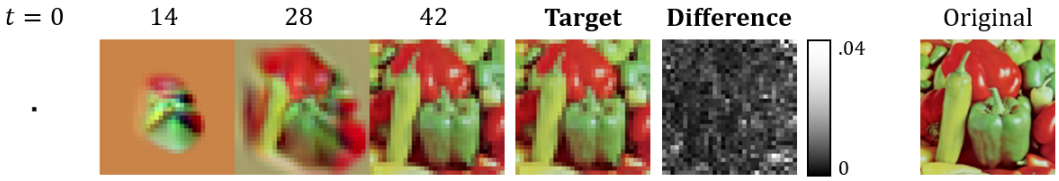


Figure 4.3: *Sample of an evolved 'Peppers' CA. This is with the Mirror boundary strategy which is non-nucleating with respect to the Sobel-state filter.*

4.3 Auxiliary Channels

In Section 4.2, we explained that nucleation sites are a key emergent feature relevant to this problem. We are confident that nucleation sites should help the network converge on NN-transition rules. However, many target images contain regions of data that are antagonistic to this goal. Take, for example, an image with a large region of solid color. This will prevent the formation of nucleation sites in that region and the neural network will have to be smart enough to form nucleation sites in that region only while the cellular automaton is far from its lifetime. To deal with this problem, we add additional components to the vectors of the grid and let the network use those additional components for nucleation sites. We call these *auxiliary channels*.

The baseline version of the image problem contains only three vector components for red, green, and blue color data. With auxiliary channels, all we have to do is extend these vectors

while keeping the loss function the same. That is, we still only measure the loss against the target using the first three components of each vector. Because we put no restrictions on the way the network uses the auxiliary channels, they are a black box to us. As seen in Figure 4.4, adding

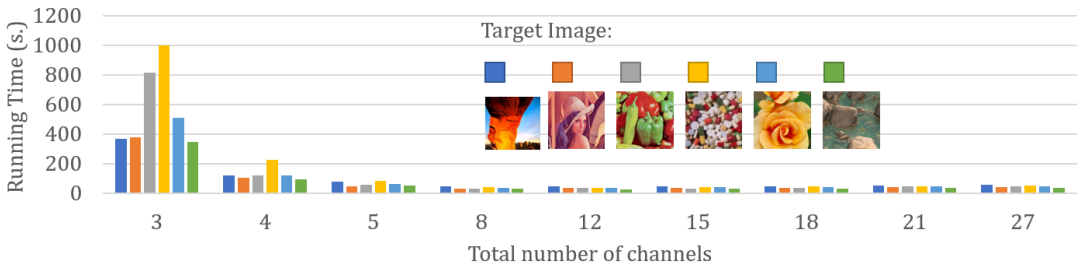


Figure 4.4: *Comparison for average running time using different numbers of auxiliary channels. The parameters used in this experiment are: target error - 1%, image size - 20×20 pixels, learning rate - 3.5×10^{-3}, network size - 1 hidden layer with 256 neurons. Colors are used to index the images.*

these additional channels **drastically** reduces the time it takes for the network to converge. This is strong support for the idea that the auxiliary channels help the system self-organize since this effect is so strong even though this is much more data for the neural network to process. Furthermore, we do not even have to demand that the network does anything in particular with these channels. From Figure 4.5, we can see that the auxilliary channels tend to build regions of color mapping to different parts of the grid. We speculate this is way that the auxilliary channels are used to help the RuleEvolver form complex images.

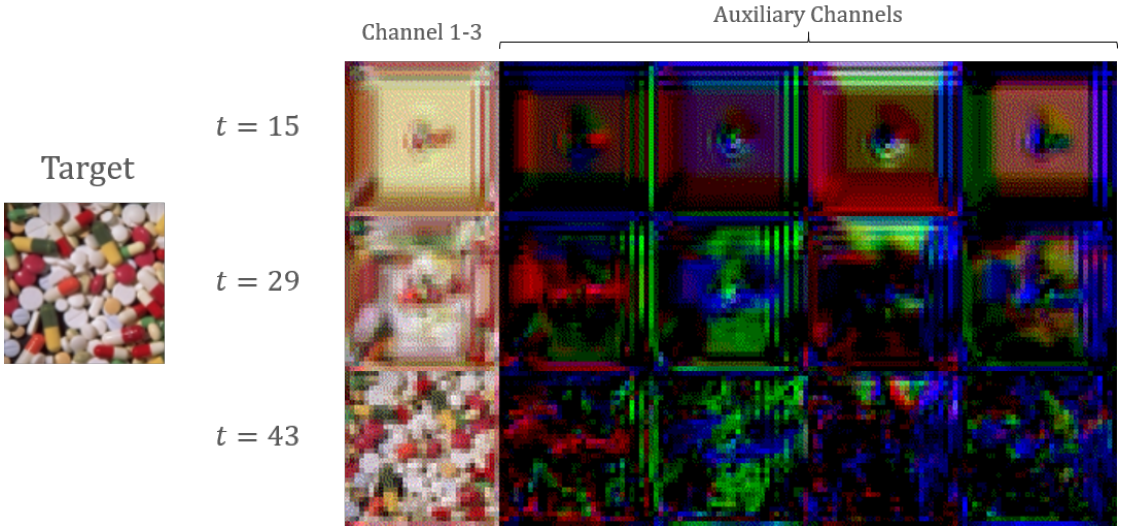


Figure 4.5: *Evolved 'Pills' with a color representation of its 12 auxiliary channels. Notice the dramatic and sharp regions of color the network forms in the auxiliary channels. Note also how the channels are used to store non-uniformity for different regions of the image.*

4.4 Boundary Strategies

Since the virtual boundary can be a nucleation site, we expect a significant reduction in running time using nucleating boundary strategies. This is clear by Figure 4.6 which shows that the nucleating boundary strategies 0-Fill and 1-Fill dominate. These results are fairly consistent even with very different target images. It is not surprising that Random is the most difficult for the network since it is forced to deal with random boundary values it has never seen before on every iteration of the cellular automaton. By Figure 4.7, we see that the network evolves a rule that copes with the random noise generated from the random virtual boundary. This is a small demonstration of the generalized learning capabilities of this technique, as well as its flexibility to handle different boundary conditions.

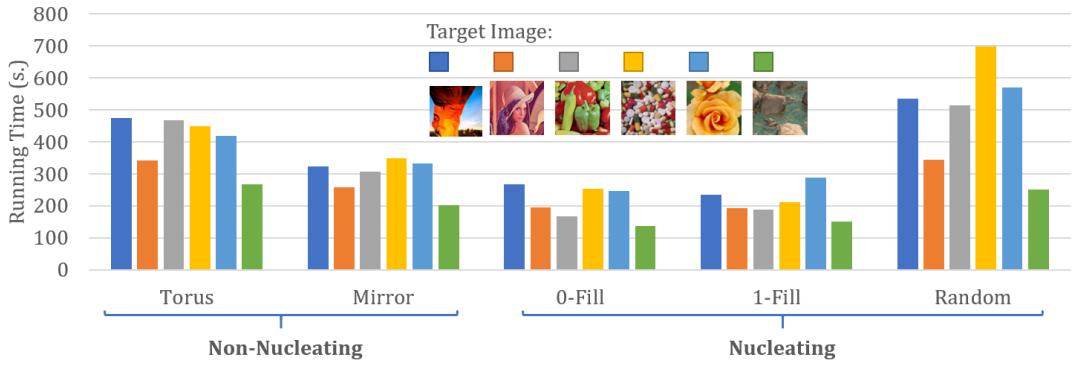


Figure 4.6: *Comparison for average running time using different boundary strategies. The parameters used in this experiment are: target error - 1%, image size - 32 × 32 pixels, learning rate - 3.5×10^{-3}, network size - 1 hidden layer with 256 neurons, auxilliary channels - 12, filter - Sobel-state filter. Colors are used to index the images.*

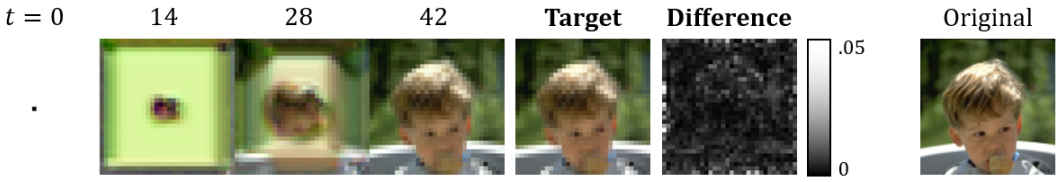


Figure 4.7: *A sample of an evolved 'Kid' CA with the Random boundary strategy. Note the visual artifacts on the left side generated by the interaction of the Sobel filter on the edge with the random values outside the grid.*

We use 0-Fill in our RuleEvolver for these reasons. However, this does not mean the other strategies are not useful. First, we have allowed the boundary strategy to be independent of the rule discovery problem, but we can impose the virtual boundary as a condition on the grid. Depending on the system we are ultimately modeling, this is sometimes necessary. For example, in Section 4.13 we show how our work on rule discovery for images can be extended to work with protein-protein interactions, but for those interactions to make any physical sense, we are

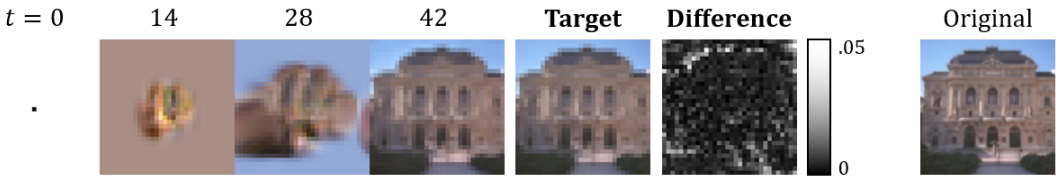


Figure 4.8: *Sample of an evolved 'Opera' CA. This time the Torus boundary strategy is used. Because this is a non-nucleating boundary strategy, the only nucleation site that forms is from the black centered dot from the initial state. Notice how on the last few frames, the quickly expanding wavefront forming on the left side of the image wraps around to the other side of the grid to meet up with the rightward expanding wave.*

required to impose the Torus boundary strategy on the system. Furthermore, the non-nucleating boundaries like Mirror or Torus give us better insight into the self-organizing capabilities of our RuleEvolver since they inject less of their own biases into the evolution of the transition rule. Consider Figure 4.8; here, we can see how the network exploits the toroidal shape of the grid and uses it to form the image from left-to-right. This unusual and clever behavior is hidden from us when using a virtual boundary that stops the growing patterns.

4.5 Initial States

To form any non-trivial target state, either the initial state must contain at least one nucleation site or the boundary strategy must be nucleating. In this section, we present some preliminary results for a few different initial states. To be sure that the boundary strategy does not interfere with the results, we restrict these experiments to the non-nucleating Mirror strategy.

Figure 4.9: *Some of the initial states tested on the entire test image set. The model successfully evolves a transition rule that takes a CA from each of these initial states into each of the target images in the test image set.*

With a uniform initial state, as expected, the algorithm fails to reach the target loss within the required number of iterations and makes almost no progress at all. However, the model converges successfully for every pair of the initial states shown in Figure 4.9 with every target image from the test image set. Initial state B is a particularly interesting example because it is non-uniform but what we might see as homogenous, yet the method finds no problem with them. Of course, the initial state has a strong impact on the running time. We suspect that initial states which have a lot of nucleation sites, but not too many, perform the best. This is supported by the results in Figure 4.10, as initial states C and D do very well across the entire

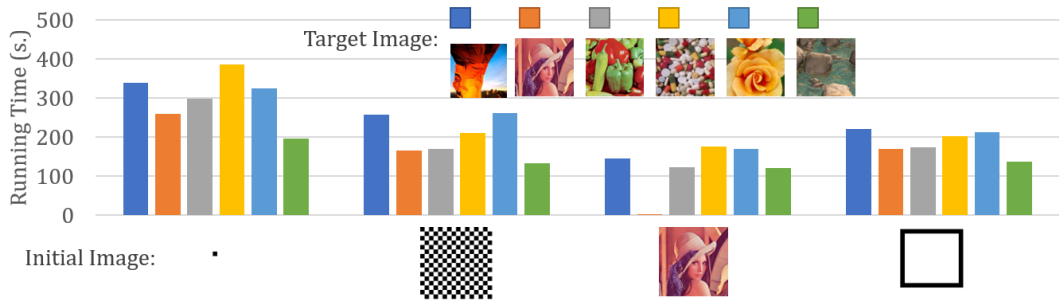


Figure 4.10: *Comparison of running time for pairs of initial states and target states. The parameters used in this experiment are: target error - 1%, image size - 32×32 pixels, learning rate - 3.5×10^{-3}, network size - 1 hidden layer with 256 neurons, auxilliary channels - 12, filter - Sobel-state filter, boundary strategy - Mirror. Colors are used to index the target images.*

image set. After this section, assume that if no initial state is given we are using the centered black dot state. We do this to ensure the effect of the initial state is minimized when measuring the effect of other parameters.

The model even converges with random initial states. We create a random initial state by uniformly selecting the values from $[0, 1]$. However, it makes a significant difference whether we choose the random initial state once, or whether we create a new random initial state for each iteration of the RuleEvolver algorithm. When the initial state is changed for each iteration of the algorithm, we call it *dynamically random*. When using a dynamically random initial state, the network has to perform generalized learning to handle initial states it has never seen before for every training step. This yields a transition rule that can form the target image from many different initial states. However, this makes the running time very long, as seen in Figure 4.11, and the model is highly unpredictable unless a nucleating boundary strategy is used. This

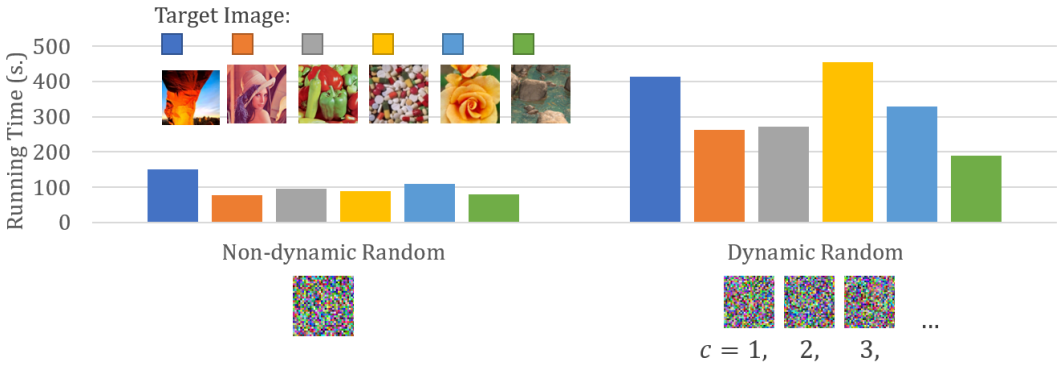


Figure 4.11: *Comparison of running time for using a non-dynamic random initial state vs. dynamic random initial states. For dynamic random states, each iteration c of the RuleEvolver, a new random state is used. The parameters used in this experiment are: target error - 1%, image size - 32×32 pixels, learning rate - 3.5×10^{-3}, network size - 1 hidden layer with 256 neurons, auxilliary channels - 12, filter - Sobel-state filter, boundary strategy - 0-Fill. Colors are used to index the target images.*

makes sense because there is very little the model can evolve about the transition rule without a nucleating boundary because the nucleations that do form from the random initial states are unpredictable. With a nucleating boundary, there is a chance for the model to evole a rule that copes with the random noise and uses the boundary to nucleate.

4.6 Hyperparameters

The free parameters of a machine learning construct are often called *hyperparameters* in machine learning research. We test the size of the neural network in Figure 4.12, showing that one hidden layer outperforms two layers for layers with a critical number of neurons. The optimal number of neurons does not depend strongly on the target image, as seen in Figure 4.13. The *learning rate*,

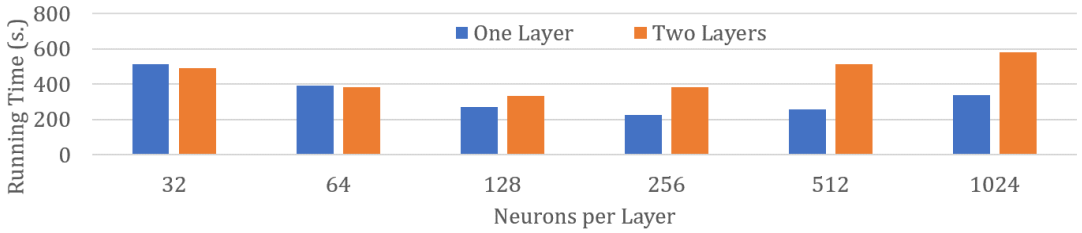


Figure 4.12: *Comparison of running time for various network topologies. The layers and number of neurons refer to the hidden layers of the network. When there are two hidden layers, then both layers have the same size. target error - 1%, image size - 32×32 pixels, learning rate - 3.5×10^{-3}, auxilliary channels - 12, filter - Sobel-state filter, boundary strategy - 0-Fill.*

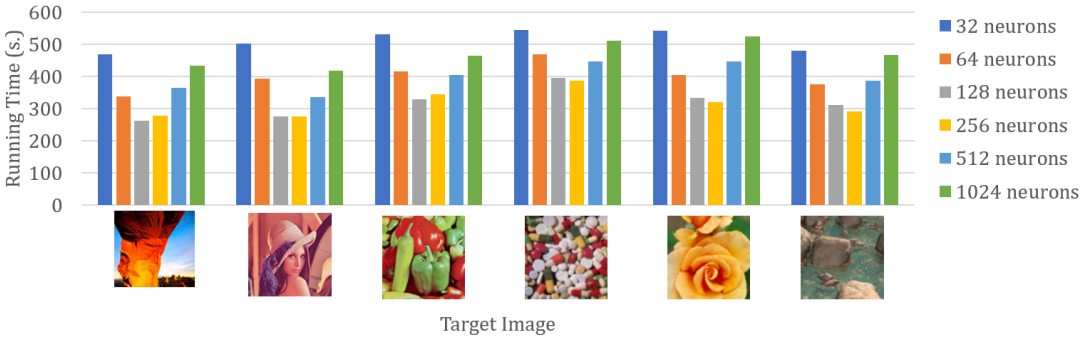


Figure 4.13: *Comparison of running time for networks of various widths across some of the more difficult samples of the image set. Notice that the performance for each network is consistent across the test images here. This consistency extends to the rest of the test image set. The parameters used in this experiment are: target error - 1%, image size - 32×32 pixels, learning rate - 3.5×10^{-3}, network size - 1 hidden layer, auxilliary channels - 12, filter - Sobel-state filter, boundary strategy - 0-Fill.*

a number that describes how strongly the optimizer reacts to the loss function when adjusting the neural network, does not depend strongly on the target image either, as seen in Figure 4.14. There is no one ideal set of hyperparameters since the best choice depends on numerous factors

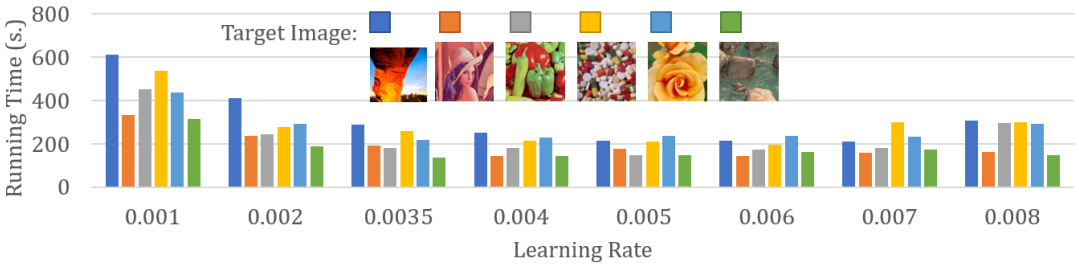


Figure 4.14: *Comparison of running time for different learning rates. Notice that the learning rate is not strongly dependent on the target image. The parameters used in this experiment are: target error - 1%, image size - 32×32 pixels, network size - 1 hidden layer with 256 neurons, auxilliary channels - 12, filter - Sobel-state filter, boundary strategy - 0-Fill. Colors are used to index the target images.*

including the size of the target image, the structure of the image, and the tradeoff we want for accuracy and running time. Despite this, the parameters are fairly consistent across both the image set and the range of CA grids we tested.

4.7 Filters

In Section 3.2 we introduced filters and claimed that this filtering step can improve running time. Here we show how each of these filters affects the time it takes to evolve the NN-transition rules. From Figure 4.15, it is clear that the Sobel-state filter outperforms the Moore filter on every

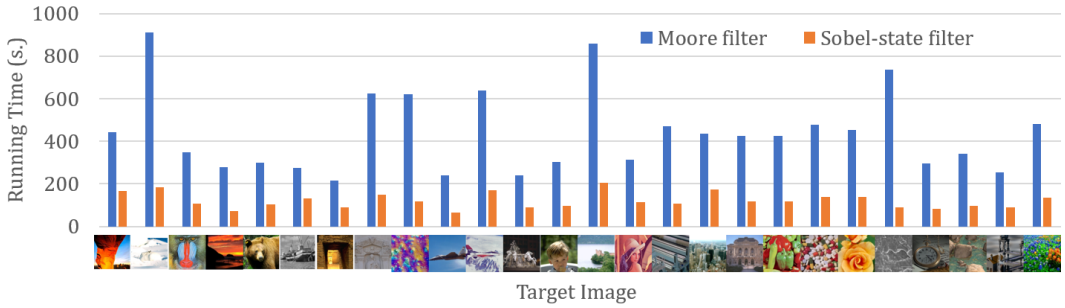


Figure 4.15: *Comparison of the running time using either the Moore filter or the Sobel-state filter across the test image set. The parameters used in this experiment are: target error - 1%, image size - 32×32 pixels, learning rate - 3.5×10^{-3}, network size - 1 hidden layer with 256 neurons, auxilliary channels - 12, boundary strategy - 0-Fill.*

test image. We speculate the reason for the stark improvement using the Sobel-state filter is because the Sobel filter encourages the network to prioritize edge formation which in turn causes nucleation sites to emerge. This is because the Sobel filter is commonly used for edge detection in image processing.

4.8 Extensions to the Loss Function

An advantage of our RuleEvolver is its flexibility with respect to the loss. If we want the network to prioritize something in its evolution of a transition rule, we can adjust the loss. We already know that prioritizing edge formation can reduce running time, so here we modify the loss function with an additional edge formation heuristic. We do this by computing a convolutional filter on the target image and the final state of the cellular automaton and computing the root mean squared error on those filtered states. We use a filter called the *Laplacian filter*. This filter is common in image processing where it is used for edge detection. It has the following kernel.

$$L = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}.$$

For a final state A_k and a target state T, the Laplacian loss LRMSE is computed as follows.

$$\text{LRMSE}(A_k, T) = \text{RMSE}(\Delta_F(A_k, L), \Delta_F(T, L)).$$

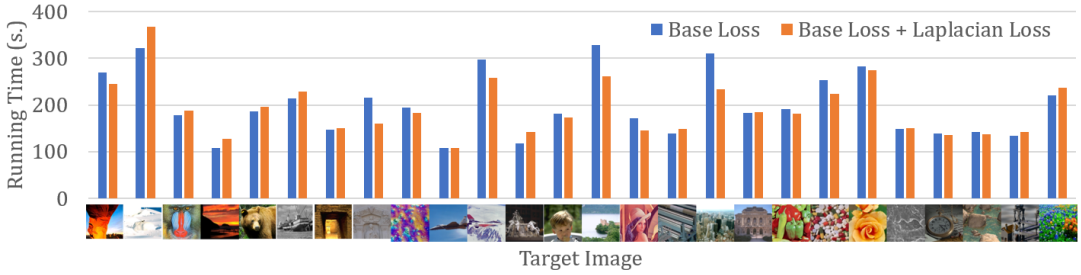


Figure 4.16: *Comparison of the running time using the base (root mean squared error) loss function and using the base loss with the Laplacian loss function. The parameters used in this experiment are: target error - 1%, image size - 32×32 pixels, learning rate - 3.5×10^{-3}, network size - 1 hidden layer with 256 neurons, auxilliary channels - 12, boundary strategy - 0-Fill.*

As seen from Figure 4.16, the results from adding the Laplacian loss are not strong. To be as conservative as possible, the target error is kept the same between both samples. Targets where the Laplacian loss seem to make the biggest difference are ones with sharp edges, but there are other targets with sharp edges that do not seem to benefit. Further research is required to see if the target error can be adjusted when adding Laplacian loss while maintaining the same degree of image fidelity. From our samples, it seems there is no perceptible change in image quality between the two loss functions at 1% target error. If the Laplacian loss results in images that look better at a less restrictive loss, this would dramatically reduce the actual running time to reach a target level of image quality. We found no significant changes to the way the networks evolve and the way the cellular automata with those transition functions self-organize using the base loss and the Laplacian loss together.

4.9 Scalability

Translational invariance means the rule discovery problem quickly explodes in difficulty as the problem size gets larger. As more content is added to the grid to replicate, the model not only has to generate that additional content but also has to do so in a way that does not conflict with the existing content in other locations in the grid. As seen in Figure 4.17, the relationship between

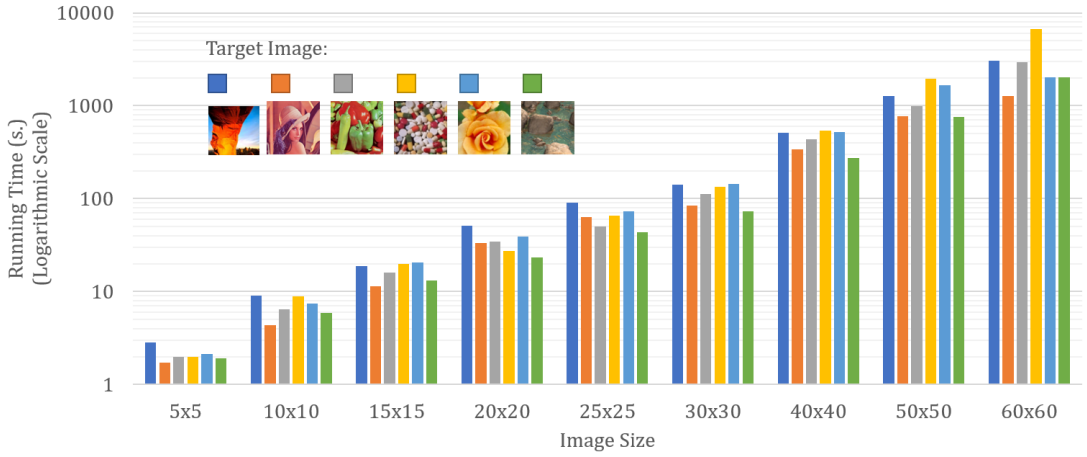


Figure 4.17: *Comparison of the running time for problems of increasing size. Note the logarithmic scale. The images correspond to the bars in each cluster read from left to right. The parameters used in this experiment are: target error - 1%, network size - 1 hidden layer with 256 neurons, auxilliary channels - 12, boundary strategy - 0-Fill. Colors are used to index the target images.*

image size and running time is exponential in our experiments. As the size of the problem grows, it makes sense that other parameters of the configuration, such as the size of the neural network, should also grow. Future research is needed to determine how to select these parameters based on the size of the problem.

4.10 Gradual Evolution

We can exploit both nucleation sites and the model's capacity for limited generalized learning with respect to regeneration to improve the RuleEvolver. In the baseline technique, we measure the loss of the final state against the target image and feed that loss into the neural network for learning. However, if we can instead encourage the network to gradually evolve sections of the target state, we can exploit the fact that these partially formed states act as nucleation sites for the rest of the image. The idea is simple: we endow the target state with a gradually increasing window that is used for the loss measurement, as seen in Figure 4.18. We use the target error to decide when to increase the size of the window, i.e., the next stage of gradual evolution only starts when the measured loss within that window is no more than the target error. Basic experiments on gradual evolution result in promising improvements in performance as seen in Table 4.2. These benefits are not just localized for a few of the images. Across our set of 26 test

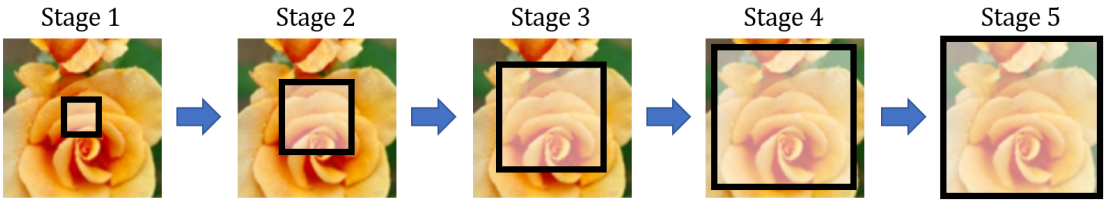


Figure 4.18: *An example of how gradual evolution is performed. The expanding window is the area of the image that is used as the target for the loss function. By the end of Stage 5, an evolved NN-transition rule is produced for the entire target image.*

images, gradual evolution produces a significant improvement in almost all of them, as seen in Figure 4.19. However, the gradual approach is very sensitive to the problem configuration. For one, the ideal jump size changes in response to many factors, some of which are hard to predict.

Table 4.2: *Reduction in running time averaged across the 26 test images using the gradual approach. The parameters used in this experiment are: target error - 1%, image size - 32×32 pixels, learning rate - 3.5×10^{-3}, network size - 1 hidden layer with 256 neurons, auxilliary channels - 12, boundary strategy - 0-Fill.*

Non-gradual (sec.)	Gradual (sec.)	Percent improvement	p-value
133	96.7	37.5%	$< 10^{-28}$

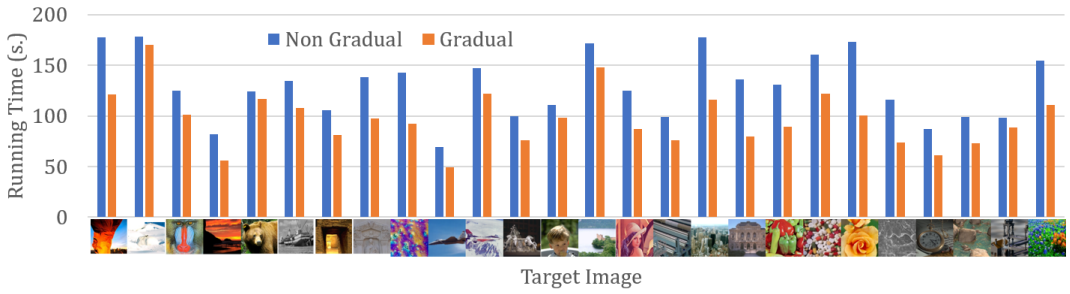


Figure 4.19: *Running time for non-gradual and gradual evolution across 26 test images. The parameters used in this experiment are: target error - 1%, image size - 32×32 pixels, learning rate - 3.5×10^{-3}, network size - 1 hidden layer with 256 neurons, auxilliary channels - 12, boundary strategy - 0-Fill.*

The gradual evolution strategy also provides us with some insight into the difficulty in reproducing different regions of the target image. Consider the time it takes to reach each stage of the gradual evolution. As seen in Figure 4.20, there is a spike in loss every time the gradual window is increased, and the lengths of the segments on the loss graph can tell us how difficult that segment is. Further research is needed to find ways of exploiting this data to improve the gradual approach. One could tune the jump sizes to try to achieve more consistent performance for each segment, or the jump sizes could adjust for the increased area of the outer segments. Furthermore, some pre-processing of the image could yield useful data that can be used to maximize the amount of non-uniformity that is generated as the gradual approach finishes

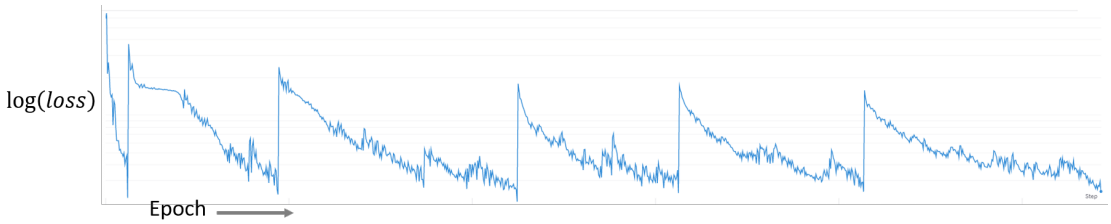


Figure 4.20: *Logarithm of the loss during gradual evolution. Notice the spike up in loss every time the target window is increased. This occurs when a fixed target error is reached. This sample has six gradual evolution segments.*

each segment. See Figure 4.21 for a look into the behavior of the model under gradual evolution.

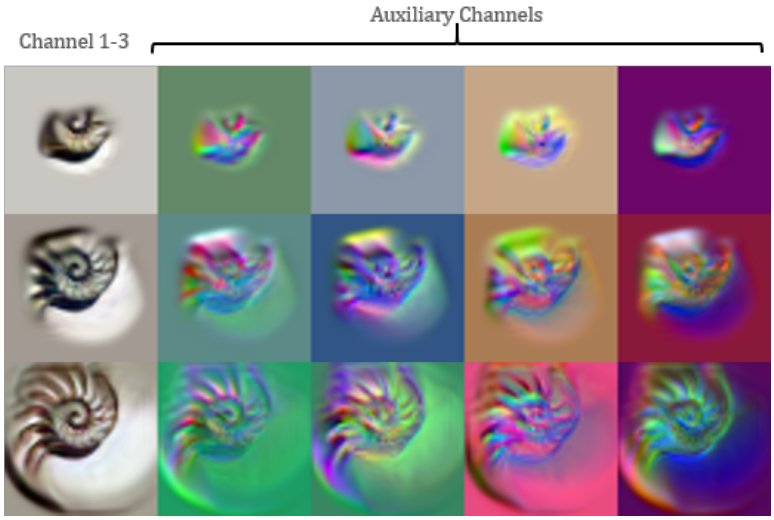


Figure 4.21: *A sample of evolving 'Nautilus' using gradual evolution. We are displaying the twelve auxiliary channels as color images on the right. Reading down the graph one can see the* ***final*** *state of the CA at each stage of gradual learning. Notice how well the expanding segment of the nautilus is evolved before the next stage.*

4.11 Companion Evolution

We can exploit another aspect of nucleation sites and the emergent behaviors of the rule discovery system to yield even more improvements. Recall from Section 4.3 that simply adding auxiliary channels that do not contribute to the loss dramatically improves the learning time. It stands to reason that one channel in the cellular automaton can provide nucleation sites for other channels. However, with auxiliary channels, we are eventually just wasting that data since nothing that is of any use to us is produced in them. If there is a way we can reap the benefits of auxiliary channels while also producing something useful to the rule discovery problem in them, we could

do even better. We do this with another novel extension we call *companion evolution*. Consider the following two experiments for comparison.

Experiment 1 (The Baseline Technique): Suppose one has two different target images A and B to run through our RuleEvolver. Using the methods described previously, one would first run it on image A and yield an NN-transition rule Δ_A. We would repeat this process separately to obtain the transition rule Δ_B for image B. Suppose the running time for the RuleEvolver for each image A and B is r_A and r_B respectively.

Experiment 2 (The Companion Technique): Now suppose instead we create a new configuration, but instead of three target channels for the red-green-blue (RGB) data in the images, we use six: the first three channels correspond to the RGB data for image A, and the next three channels for image B. Our loss is measured the usual way, and our target state is a six-channel array constructed by putting image A in the first three channels and image B in the last three channels. Running our RuleEvolver on this six-channel structure yields a single transition rule Δ that reproduces image A in the first three channels and image B in the next three channels. See Figure 4.22 for a summary of this approach. Suppose the running time to produce the combined Δ is r.

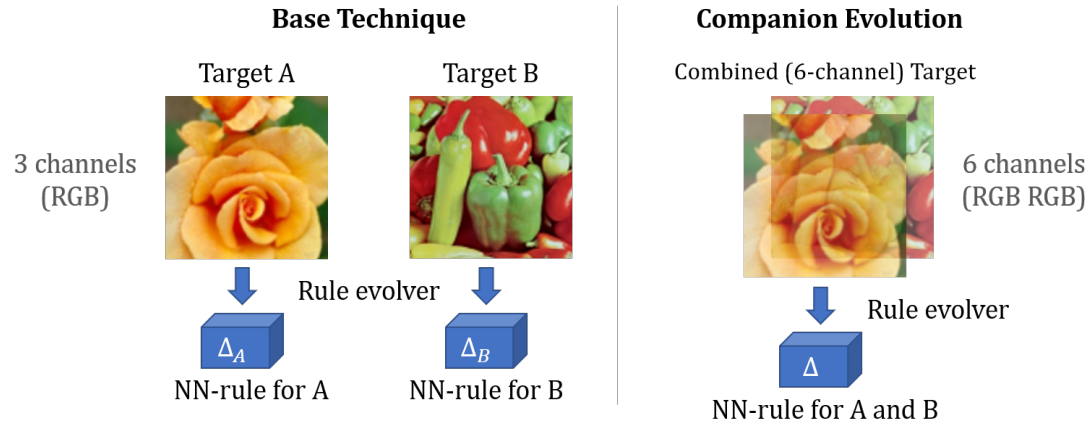


Figure 4.22: *Summary of companion evolution*

The NN-transition rule Δ is a solution to the rule discovery problem for both image A and image B; all we have to do is choose which of the three channels to use to get one image or the other. This means that if $r < r_A + r_B$, the companion evolution technique is better. It turns out, for many image combinations and other a variety of experimental parameters, this is indeed the case. The improvements for each pair of images under companion learning can be seen in Figure 4.23. Each percentage measures the relative difference in r and $r_A + r_B$. The cells on the main diagonal should be interpreted not as an improvement but as a statement about how the model reacts to changes in the channel structure since we do not yield a useful improvement to Δ by replicating the image twice. For many pairs, the time improvement produced by running the model with the companion model is significant and very large, in some cases accomplishing

the combined tasks in less than **half** the time while also producing a simpler transition rule.

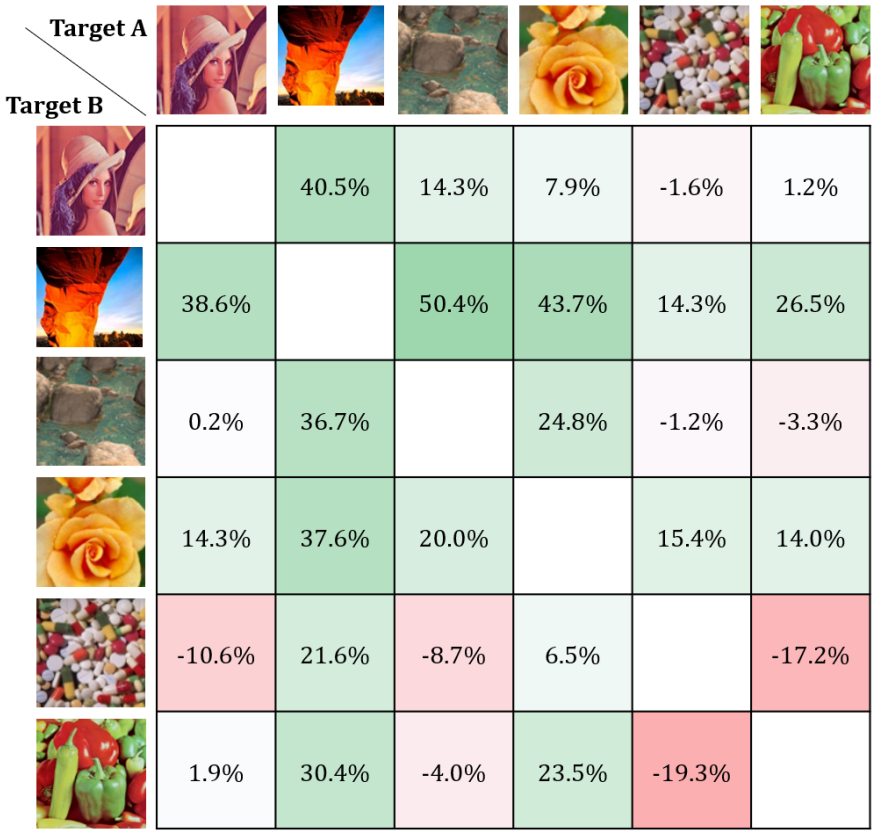


Figure 4.23: *Percent improvement in total running time by companion learning. Each cell represents a companion running pair between the images on the corresponding row and column.*

This technique only works if we know ahead of time that we want to evolve two different samples. What if we evolve a rule on its own with c auxiliary channels and then try to inject this evolved model into a companion model? We cannot keep all c auxiliary channels because adding another channel to a completed model completely disrupts the topology of the neural network. Likewise, if we try to replace one of the auxiliary channels with a companion instead, the neural network has to learn to cope with the loss of an auxiliary channel it was evolved to use before. This is an area of ongoing research that could dramatically overhaul the re-evolution process.

4.12 Interesting Observations

The hallmark of self-organizing systems is the emergence of unexpected global behavior from countless local interactions. Our combination of neural networks with self-organizing systems yields countless examples of such unexpected and sometimes strange behavior. We cannot include all such results, but here we present a few examples of such behavior.

4.12.1 Wave-like Patterns with Sobel filter

When we use the Sobel-state filter, we frequently observe patterns of moving waves and stripes. One particularly salient example of this can be found in Figure 4.24. We speculate this is related to *Turing patterns* that arise in chemical diffusion systems originally described by Alan Turing in his works on morphogenesis [12] due to its qualitative similarities as well as the fact that the Sobel-state filter mimics the computations that those patterns arise in.

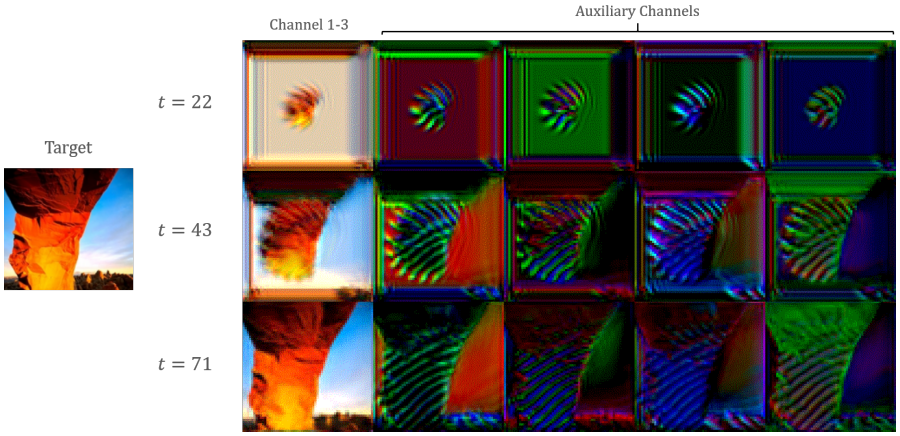


Figure 4.24: *A sample of emergent behavior in an evolved model for 'Arch' when using the Sobel-state filter. Notice the strong stripes forming as the image grows and also that the stripes remain in the auxiliary channels even in the final state. The parameters used in this experiment are: target error - 1%, image size - 60×60 pixels, learning rate - 3.5×10^{-3}, network size - 1 hidden layer with 256 neurons, auxilliary channels - 12, boundary strategy - 0-Fill.*

4.12.2 Checkerboards with Moore filter

While, in our samples, the Sobel-state filter results in the best running times, the Moore filter described in Section 3.2 results in a unique 'checkerboard' emergent behavior as seen in Figure 4.25

This could be another case of generating non-uniformity to help the system form the target image. This is supported by the fact that we do not see this effect when using Sobel-state filters in the pre-processing step; for a checkerboard pattern, the Sobel values vanish because the gradient in either direction for a checkerboard is always zero.

4.13 Self-Organizing Protein Systems

Self-organizing systems research is largely inspired by the real world, especially biology, as understanding how the complexities of living systems self-organize is an astoundingly difficult and important problem. Here we present a modest application of our rule discovery framework to biology. Initial and early results were produced as a final project for *COMP 597: Special*

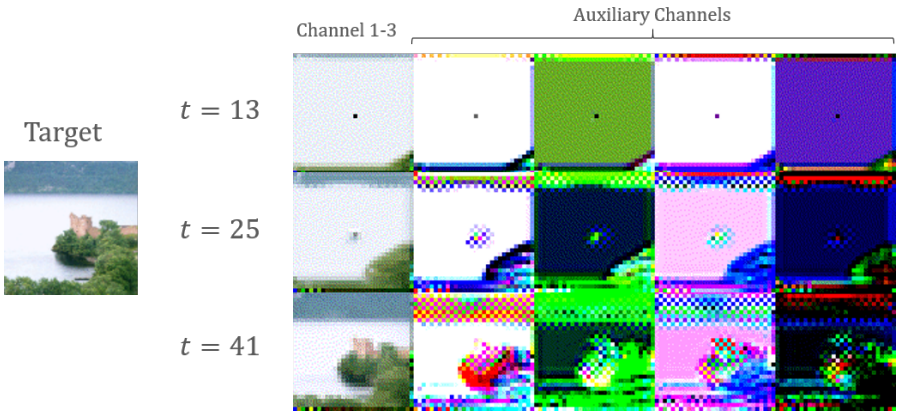


Figure 4.25: *A sample of emergent behavior in an evolved model for 'Lake' when using the Moore filter. Notice how the checkerboard patterns are erased from the channels that are measured against the loss function. Note also that in the middle snapshot that the checkerboard patterns are visible even in the left-most frame. The parameters used in this experiment are: target error - 1%, image size -* 32×32 *pixels, learning rate -* 3.5×10^{-3}*, network size - 1 hidden layer with 256 neurons, auxilliary channels - 12, boundary strategy - 0-Fill.*

Topics taught by Dr. Hyuntae Na at Penn State Harrisburg. We present those results along with further results produced after the end of that class.

Proteins are one of the building blocks shared by all living things. Understanding how proteins interact helps us understand cell life cycles, behaviors, and more. Multi-protein systems are a rich topic in self-organization because we are confident that we understand the basic idea of how these systems come together but we know little about the earliest forms of life. Suppose we have a set of proteins that we expect to play an important role in some biological process. Mapping each of these proteins to a cell in a cellular automaton, we can represent the internal states of the proteins as the vectors stored in the cells, and we can describe the behavior of that system by the behavior of the corresponding CA. Finally, we can describe the interaction of those proteins by an NN-transition rule as we use throughout this thesis. This yields a biological instance of the rule discovery problem: *Given a multi-protein system and known details about how it behaves, find the interaction rules of those proteins.* To demonstrate this framework on a real biological system, we use the model of the cell cycle of yeast. This model consists of a network of the proteins seen in Figure 4.26 connected by edges that describe the known interactions between those proteins [13].

Because this model is known, we obtain a set of behaviors this multi-protein system should have. Then we run our RuleEvolver and compare the evolved NN-transition rule to the known interactions. The model seen in Figure 4.26 produces a sequence of nine important stages of the yeast protein states. We let this be a sequence of target states and our RuleEvolver finds an NN-transition rule that replicates all nine of these stages. A modified version of the gradual evolution technique presented in Section 4.10 dramatically improves the performance of this model **twelve-fold** with extreme statistical significance ($p < 0.0001$.) This modified gradual

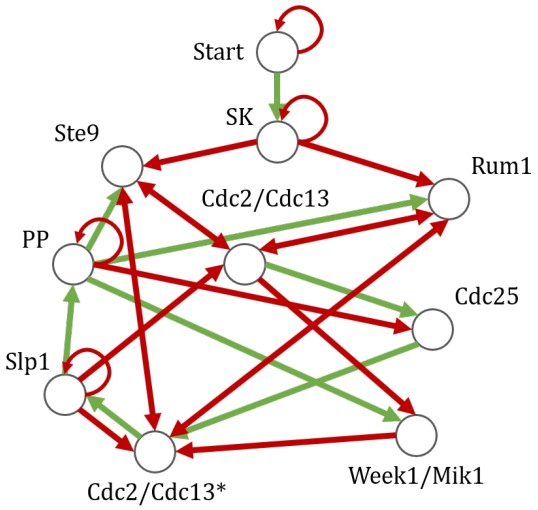


Figure 4.26: *Boolean Network of a fission Yeast cell cycle [13]. Each node is a protein that takes part in the cell division process and the links determine the activation/inhibition interaction between the endpoints.*

approach is performed by gradually including more of the target stages into the loss until all nine of the stages are included.

The rule discovery model has an important advantage over the baseline protein network model from Figure 4.26. The baseline model can only capture a sequence of states for the proteins, not the time it takes for each of those stages to be reached. However, with our RuleEvolver, not only can we find a transition rule that generates all nine of the stages, but it also allows us to describe how long those stages should last. This is vital for modeling purposes especially since the example given here simulates a cell replication process. However, we do not yet know how to extract the interaction rules encoded by the NN-transition rule since it is a black box. This is an area of ongoing research into this problem to make our RuleEvolver more useful in biological modeling.

Chapter 5

Conclusions

Cellular automata are quite general models as computational devices as demonstrated by their ability to model all kinds of dynamical systems. Using these models with neural networks we learn that nucleation sites are key in translationally invariant models. However, too many nucleation sites can also complicate the problem as demonstrated by some of the more complex test images tested with the gradual approach and the companion learning approach from Chapter 3. This conclusion agrees with a long-standing belief among cellular automata researchers that complexity arrives on the "edge of chaos", a term coined by Norman Packard and repeated throughout complex systems research [14] due to the discovery that many interesting emergent behaviors in CAs occur in the transition from totally chaotic systems (random noise) and ordered, predictable systems.

Our success in replicating the behaviors of protein interaction models in Section 4.13 is a glimpse into the power of this framework. We put even more restrictions on our networks than we really need to model the yeast cell cycle, yet our relatively tiny neural networks quickly find a way to develop the collective behavior required to evolve that behavior. While we should not expect this framework to generate physically plausible interaction rules for atomic systems any time soon, the explosive growth in machine learning right now means we should take seriously the possibility of using a rule discovery framework to aid both in developing realistic models for existing systems and in evolving entirely novel behaviors for future bio-engineering efforts.

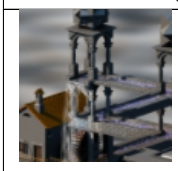
Appendix

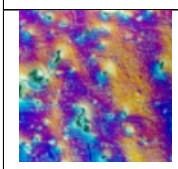
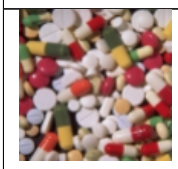
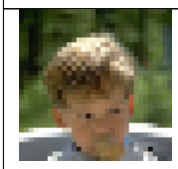
A.1 Code and Experimental Results

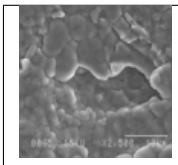
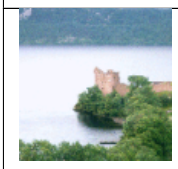
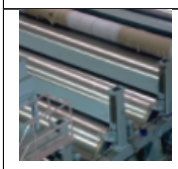
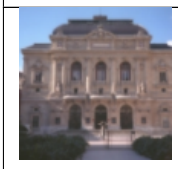
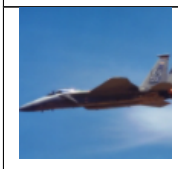
All code can be found at `https://github.com/omegachysis/thesis` and experimental results hosted for a limited time at `https://wandb.ai/omegachysis/neural-cellular-automata`. Please contact the maintainer of the GitHub repository for any questions.

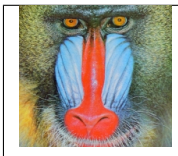
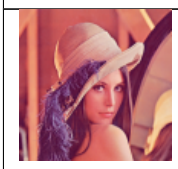
A.2 Test Images

Our test image suite is, in part, provided by the University of Southern California's Signal and Image Processing Institute (`http://sipi.usc.edu/database/`) The image collection is provided by Fabien Petitcolas (`https://www.petitcolas.net/watermarking/image_database/`) for research use. The following authorship information provided by Fabien Petitcolas is reproduced here:

Modified Image	Name	Author
	Waterfall	Sascha Ledinsky
	Always running, never the same...	Vives Piqueres
	Pocket Watch on a Gold Chain	Kevin Odhner

	Wildflowers	Robert E. Barber
	Localised corrosion on an electropolished Al-Zn-Mg-Cu alloy	Gérald Deshais
	Pills	Karel de Gendre
	Bandon beach	Robert E. Barber
	Fontaine des Terreaux	Éric Labouré
	Brandy rose	Toni Lankerd
	Fourvière Cathedral, north wall	F. A. P. Petitcolas
	Black Bear	Robert E. Barber
	Kid	Karel de Gendre

	Intergranular Stress Corrosion Cracking of an Al-Zn-Mg-Cu alloy	Gérald Deshais
	Loch Ness	Patrick Loo
	Paper machine	Karel de Gendre
	Skyline Arch	Robert E. Barber
	Pueblo Bonito	Robert E. Barber
	Opera House of Lyon	None listed
	F15	Toni Lankerd
	New-York	Patrick Loo
	Arctic Hare	Robert E. Barber

	Baboon	Signal and Image Processing Institute, USC
	F16	Signal and Image Processing Institute, USC
	Fishing boat	Signal and Image Processing Institute, USC
	Lena	Signal and Image Processing Institute, USC
	Peppers	Signal and Image Processing Institute, USC

Bibliography

[1] Kingma, D. P. and J. Ba (2017), "Adam: A Method for Stochastic Optimization," 1412.6980.

[2] Garnier, S. (2011) *From Ants to Robots and Back: How Robotics Can Contribute to the Study of Collective Animal Behavior*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 105–120.
URL https://doi.org/10.1007/978-3-642-20760-0_5

[3] Gardner, M. (1970) "MATHEMATICAL GAMES," *Scientific American*, **223**(4), pp. 120–123.
URL http://www.jstor.org/stable/24927642

[4] Chertow, M. and J. Ehrenfeld (2012) "Organizing Self-Organizing Systems," *Journal of industrial ecology*, **16**(1), pp. 13–27.

[5] Newman, S. A., T. Glimm, and R. Bhat (2018) "The vertebrate limb: An evolving complex of self-organizing systems," *Progress in biophysics and molecular biology*, **137**, pp. 12–24.

[6] Ultsch, A. (1993) "Self-Organizing Neural Networks for Visualisation and Classification," in *Information and Classification*, Springer Berlin Heidelberg, pp. 307–313.
URL https://doi.org/10.1007%2F978-3-642-50974-2_31

[7] Mordvintsev, A., E. Randazzo, E. Niklasson, and M. Levin (2020) "Growing Neural Cellular Automata," *Distill*, https://distill.pub/2020/growing-ca.

[8] Toffoli, T. (1984) "Cellular automata as an alternative to (rather than an approximation of) differential equations in modeling physics," *Physica D: Nonlinear Phenomena*, **10**(1), pp. 117 – 127.
URL http://www.sciencedirect.com/science/article/pii/0167278984902549

[9] Schönfisch, B. and A. de Roos (1999) "Synchronous and asynchronous updating in cellular automata," *Biosystems*, **51**(3), pp. 123 – 143.
URL http://www.sciencedirect.com/science/article/pii/S0303264799000258

[10] Schmidhuber, J. (2015) "Deep learning in neural networks: An overview," *Neural Networks*, **61**, pp. 85–117.
URL https://doi.org/10.1016%2Fj.neunet.2014.09.003

[11] Rendell, P. (2016) *Turing Machine in Conway Game of Life*, Springer International Publishing, Cham, pp. 149–154.

[12] Turing, A. M. (1952) "The Chemical Basis of Morphogenesis," *Philosophical transactions of the Royal Society of London. Series B, Biological sciences*, **237**(641), pp. 37–72.

[13] Davidich, M. I. and S. Bornholdt (2008;2007;) "Boolean network model predicts cell cycle sequence of fission yeast," *PloS one*, **3**(2), pp. e1672–e1672.

[14] Sughimura, N., R. Suzuki, and T. Arita (2014) "Non-uniform Cellular Automata based on Open-ended Rule Evolution," *Artificial life and robotics*, **19**(2), pp. 120–126.

Vita

Matthew A. Robinson

Penn State Harrisburg

Graduation: May 2021

M.S. in Computer Science, B.S. in Computer Science, with Honors

Accomplishments

- Member of Capital College Honors Program
- Peer Tutor at the Learning Center
- Directed STEM programs at Middletown Public Library
- Graduate Assistant
- Teaching Assistant for graduate-level classes
- AP Scholar with distinction

Skills and Academic Studies

- Multivariable Calculus and Linear Algebra
- Algorithms and Data Structures
- Computational and Structural Biology
- Artificial Intelligence and Machine Learning
- Statistics and Data Analysis
- Programming Language Design
- Computer Graphics
- C/C++, Python, C#, Julia, Java, LaTeX