ABSTRACT

Title of Dissertation: RESERVOIR COMPUTING WITH BOOLEAN LOGIC NETWORK CIRCUITS

Heidi Baumgartner Komkov
Doctor of Philosophy, 2021

Dissertation Directed by: Professor Daniel P Lathrop
Department of Physics

To push the frontiers of machine learning, completely new computing architectures must be explored which efficiently use hardware resources. We test an unconventional use of digital logic gate circuits for reservoir computing, a machine learning algorithm that is used for rapid time series processing. In our approach, logic gates are configured into networks that can exhibit complex dynamics. Rather than the gates explicitly computing pre-programmed instructions, they are used collectively as a dynamical system that transforms input data into a higher dimensional representation. We probe the dynamics of such circuits using discrete components on a circuit board as well as an FPGA implementation. We show favorable machine learning performance, including radiofrequency classification accuracy comparable to a state of the art convolutional neural network with a fraction of the trainable parameters. Finally, we discuss the design and fabrication of a reservoir computing ASIC for high-speed time series processing.

Reservoir Computing with Boolean Logic Network Circuits

by

Heidi Baumgartner Komkov

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2021

Dissertation Committee:
Dr. Daniel P. Lathrop, Chair/Advisor
Dr. Timothy Horiuchi
Dr. Thomas Antonsen
Dr. Timothy Koeth
Dr. Brian Beaudoin
Dr. Vedran Lekic

Acknowledgments

I would like to express my deep appreciation to all current and former team members in the Lathrop group: Daniel Lathrop, Alessandro Restelli, Brian Hunt, Liam Pocher, Liam Shaughnessy, Itamar Shani, Anthony Mautino, and John Rzasa.

Thanks as well to Brian Beaudoin, Timothy Koeth, and Irving Haber, from the University of Maryland Electron Ring (UMER), where I spent my first years of graduate school, as well as my fellow UMER graduate students Kiersten Ruisard and Levon Dovlatyan.

Thank you to Advait Madhavan and Jabez McClelland in the Alternative Computing Group at the Nanoscale Device Characterization Division of the Physical Measurement Laboratory at the National Institute of Standards and Technology (NIST) for providing me with no-cost wafer space for the ASIC described in Chapter 6.

I am grateful for the help and advice of the staff at the Institute for Research in Electronics and Applied Physics (IREAP) as well as the UMD NanoCenter.

And most of all, I thank my family for supporting me in more ways than I can count.

This material is based in part upon the work supported by the National Science Foundation (NSF) (Nos. EAR 1417148 and 1909055), as well as the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE 1840340. Our research is partially supported through a DoD contract under the Laboratory of Telecommunication Sciences Partnership with the University of Maryland.

Table of Contents

Chapter 1: Introduction

1.1 Trends in Computing

The development of powerful, energy-efficient, and low-cost computers in the last half century brought about the information age, profoundly changing the world we live in. In 2019, 93% of people on Earth lived within reach of a mobile broadband service, and 53.6% were internet users, limited mainly by the affordability of computing devices in low-income countries [1]. As the affordability gap continues to shrink and a larger fraction of the global population gains access to the internet, demands for connectivity, computing power, and data storage will continue to rise dramatically [2].

Moore's law —- the observation that the number of transistors per unit area doubles roughly every two years [3] —- has long held true, partly because the semiconductor industry has used it as a performance target. However, fundamental physics limitations to the size of transistors are rapidly approaching, and Gordon Moore himself predicts that his eponymous rule of thumb will stop being valid in 2025 [4]. A related principle, Dennard scaling, observed that as transistors shrink, their voltage and current scale linearly, keeping power per unit area roughly constant. Dennard scaling came to an end in the mid-2000s as transistors became

small enough for leakage current to become non-negligible, and voltages became low enough to run into threshold voltage limitations. As a result, processor speeds have not exceeded about 4GHz since the mid-2000s. To compensate for stagnation in clock frequency, chips have become multi-cored; however not all tasks take advantage of being parallelized [5].

In a rapidly growing digital world, new fundamental computing paradigms are needed to meet increasing global demands for connectivity and number-crunching power. There are several general approaches that development can follow [6], from making the current state of the art more efficient, to changing computing paradigms completely.

1.1.1 More Moore

While bottom-up improvements to processors have slowed down, top-down improvements to the computing stack may make computer applications faster in the post-Moore era. Approaches include software performance engineering, hardware streamlining, and new algorithm development. However, as Leiserson et al. argue, these piecemeal improvements are unlikely to create the broad gains in computer performance like bottom-up approaches have accomplished in the last half century [7].

1.1.2 More than Moore

Novel devices, processes, and materials may exceed CMOS performance and unlock new regimes of computation. Emerging devices include tunneling FETs, spintronic devices, graphene and carbon nanotube transistors [8]. Nonvolatile memories showing promise include phase change memory, spin-torque transfer memory, and resistive memory [9]. Photonic and quantum computing also fall under the beyond-CMOS umbrella.

Most beyond-CMOS approaches are still firmly within the realm of experimentation and are far from practical implementation. The beyond CMOS roadmap depends upon the development of new devices and materials, whose wide-scale manufacturability remains unproven.

1.2 The Machine Learning Explosion

Machine learning (ML), a term popularized in a 1959 paper by computing pioneer Arthur Samuel [10], describes algorithms that give computers the ability to learn patterns and rules in data without being explicitly programmed with those rules. Today, it encompasses many sub-fields including computer vision, natural language processing, and deep learning. It is nearly impossible to escape the influence of machine learning in our technological society —- it is everywhere from voice-activated virtual assistants, to ads internet users are served, to increasingly popular self-driving vehicles.

While there are many algorithms under the machine learning umbrella, neural

networks form the basis of deep learning and are arguably the most influential concept. They are a family of algorithms inspired by neurons in the brain. Biological neurons in the brain receive signals from other neurons. When a cumulative signal threshold is surpassed, a spike-like signal is passed to subsequent neurons. The synaptic connections between neurons change as the brain learns, strengthening or weakening based on the relative timing of signals.

1.2.1 Machine Learning Opportunities

While machine learning has been studied since the dawn of computers, only in the last decade has it exploded in popularity due to several interrelated factors. The first is the decrease in cost of processors, memory, and network hardware which has enabled the generation and storage of vast quantities of data. Secondly, algorithmic developments allowed for more efficient training of deep neural networks. Finally, the availability of graphics processing units (GPUs), while originally for media processing and games, has allowed for efficient processing of neural networks as they parallelize matrix multiplication operations.

Deep learning has in many ways enabled computers to surpass human abilities. Today, machines can translate language in real time [11], win against humans at games such as Go [12] and Starcraft [13], and autonomously drive vehicles that could make roads safer [14]. Computers have been outperforming human accuracy at the Imagenet image classification challenge since 2015 [15].

It is easy to take for granted the breathtaking progress that machine learning

has made, but the current pace cannot continue for much longer. Deep learning comes at a tremendous energy cost: between 2012 and 2018 there was a 300,000 times increase in the amount of computing power state-of-the-art networks required [15]. Deep learning is also nearing its computational limits on present-day hardware: Thompson et al. [16] studied the amount of processing incremental improvements to major ML models will require, and the point of diminishing returns is fast approaching.

As the need to improve the efficiency of deep learning has become clear, neural network compression techniques have been developed. For example, quantizing neural networks, representing their internal states with fewer bits to reduce the computational overhead. Also, pruning them, meaning removing insignificant connections to avoid needing to compute them. In fact, entirely binarized neural networks, in which weights are only one or zero, have been developed without much of an accuracy reduction in some applications [17, 18]. As a major energy cost comes from repeated DRAM accesses, re-using terms in memory as much as possible is another strategy [5].

Specialized neural network hardware is also gaining in popularity. This is especially significant as neural network demands move increasingly to edge devices like smartphones, which have stringent power limitations but abundant machine learning needs [19]. Major companies are investing in dedicated hardware to accelerate their machine learning models, including Google which designed Tensor Processing Units (TPUs) in-house to accelerate the multiply-accumulate operations that many of Google's services need [20]. Many neural network accelerator startups are ap-

pearing to address the needs of industry, including Cerebras [1], Mythic[2], Fathom Computing[3], Lightmatter[4], Lightelligence[5], Luminous Computing[6], and LightOn[7], to name a few.

1.2.2 Our Approach

To push the frontiers of machine learning at a time when conventional hardware is nearing its limits, completely new computing architectures must be explored which efficiently use hardware resources. We propose an unconventional use of digital logic gates which holds the promise to more densely encode information than conventional digital computation. In our approach, digital logic gate circuits are configured into large random graphs that can exhibit complex dynamics. Rather than the gates explicitly computing pre-programmed instructions, they are used collectively as a dynamical system that becomed generally synchronized to input data. The system evolves through time, exhibiting behavior that is dependent upon the external inputs. The state of the system is captured and then applied as an input to a simple readout map. Effectively, the network replaces many layers of a deep neural network, performing all of its operations in parallel.

This method is a type of *reservoir computing*, in which our *reservoir* is a logic gate network. It can be used for classification, regression, or time series prediction.

[1] https://cerebras.net/
[2] https://www.mythic-ai.com/
[3] https://www.fathomcomputing.com/
[4] https://lightmatter.co/
[5] https://www.lightelligence.ai/
[6] https://luminous.co/
[7] https://lighton.ai/

In contrast to neural networks with many layers, it uses only a single trained output layer, which allows it to be very fast.

1.2.3 Thesis Overview

In this thesis, Chapter 2 explores machine learning and reservoir computing theory and background literature. Chapter 3 discusses background information about Boolean network theory and circuit implementations. Chapter 4 covers our results from studies on Boolean networks implemented on a field-programmable gate array (FPGA), based on designs published in Komkov et al. [21] and experiments in Shani et al. [22] and Komkov et al. [23]. Chapter 5 studies a network implemented using discrete logic gates on a printed circuit board, which allows the full analog behavior of the network to be probed [24]. Chapter 6 describes the design of a monolithic chip implementing the same network architecture, based on designs in patents [25, 26]. Chapter 7 describes the application of software-based reservoir computing to the prediction of a nonlinear particle accelerator time series from the University of Maryland Electron Ring, which was published in Komkov et al. [27].

Chapter 2: Machine Learning and Reservoir Computing Theory and Literature

2.1 Neural Network Theory

2.1.1 The Artificial Neuron

Neural networks are machine learning architectures based on the artificial neuron. While biological neurons transmit voltage pulses and adjust their connection strengths to other neurons based on relative pulse timing, the artificial neuron is a static concept amenable to computation with combinational logic. A diagram of the artificial neuron is shown in figure 2.1 and it is described by equations 2.1.

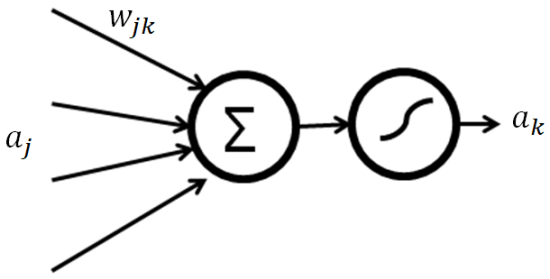


Figure 2.1: The artificial neuron. Inputs a_j are weighted by w_{jk} and summed, then passed through a nonlinear activation function to arrive at output a_k.

$$z_k = \sigma(\sum_{i=1}^{n} a_j w_{jk}), \;\; a_k = \sigma(z_k), \tag{2.1}$$

The inputs, or *input activations* of the artificial neuron, denoted a_j, are multiplied by weights w_{jk}. All the weighted values are summed to arrive at the intermediate value z_k. This value is then passed through a typically nonlinear function σ, called the *activation function*, to compute the neuron's *output activation* a_k. The activation function is commonly a hyperbolic tangent (tanh), but it can also be a step function (a hard threshold), sigmoidal, linear, a computationally friendly piecewise linear approximation called a rectified linear unit (ReLu), or a number of other options. A nonlinear activation function is needed to solve nonlinear problems; otherwise, the function of the artificial neurons simplifies to matrix multiplication.

2.1.2 The Feed-forward Neural Network

Artificial neurons configured into sequential layers, such as in figure 2.2, form a feed-forward neural network (FFNN), a structure sometimes called a multi-layer perceptron (MLP) (though historically, *perceptron* refers to a neuron specifically with a step activation function). The first layer of neurons is called the input layer, the last layer is called the output layer, and any layers in between are called hidden layers. *Deep learning* encompasses architectures based in part on feed-forward neural networks, often with many layers – hence "deep".

In figure 2.2, a feed-forward neural network with one hidden layer is shown. In this diagram, activations have the subscript i in the input layer, j in the hidden layer, and k in the output layer, and weights are denoted w_{ij} and w_{jk} between input and hidden, and hidden and output layers respectively. Equation 2.2 is an output

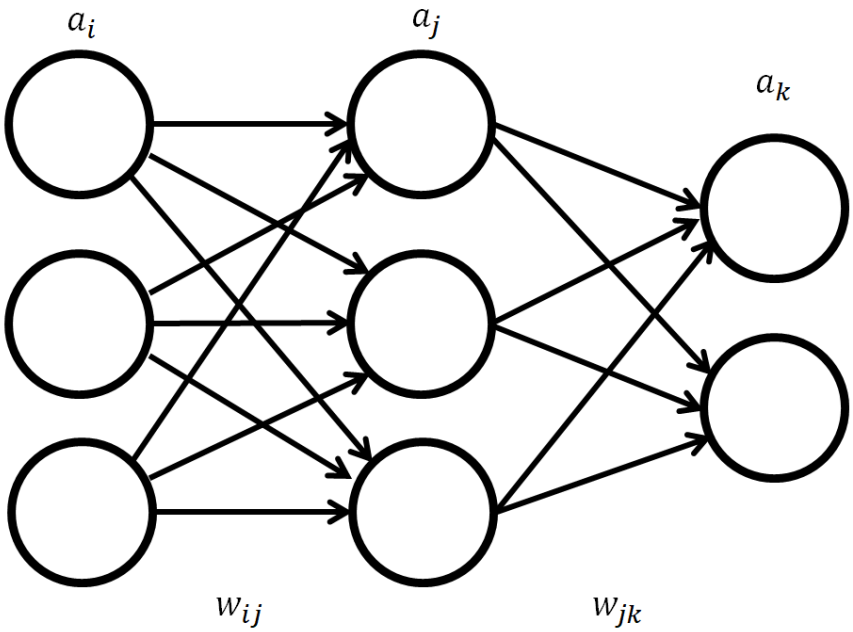


Figure 2.2: A feed-forward neural network with one hidden layer. Neuron activations are denoted a_i, a_j, and a_k for the input, hidden, and output layers, respectively. Weights between them are w_{ij} and w_{jk}. The output dependence upon all other parameters is in 2.2.

activation written explicitly in terms of all of the other parameters.

$$a_k = \sigma_k(\sum(\sigma_j(\sum a_i w_{ij}))w_{jk}) \tag{2.2}$$

2.1.3 Using a Neural Network

Artificial neural networks can be used for classification or regression problems. Inputs to the neural network are numerical features of data. Outputs represent what we desire to learn from the data. A classifier has as many output neurons as there are classes, and the output activations represent the probability of the input data belonging to a particular category. The outputs of a regressor are continuous values.

Forward propagation is the computation of all of the hidden and output neuron activations within the network given some input. For a network with one hidden layer, it would be the evaluation of equation 2.2 for all outputs. *Training* is the process of adjusting the weights within the neural network to arrive at a desired outcome.

2.1.4 Training a Neural Network

Neural networks are trained by example rather than by programming explicit rules. To train the network, first forward propagate training data. Choose an *loss function* (also called *error function* or *cost function*) E, which quantifies how far the output is from a training goal. Then, determine how each of the trainable parameters—the weights in the network—depend upon the loss function by computing the gradient of the loss function with respect to each of the weights. Finally, take a step to decrease the loss function by updating each of the weights in the direction of the negative gradient.

A popular loss function choice for regression tasks is mean squared error (MSE), which is the sum of squared errors:

$$E = \frac{1}{2N} \sum_{k=1}^{N} (t_k - a_k)^2 \tag{2.3}$$

In this equation, a_k are output activations, t_k are the target values, and there are N output neurons. A different loss function often used for classification problems is cross-entropy loss, which can be used when output activations and targets are

between zero and one, and heavily penalizes outliers:

$$E = \frac{1}{N} \sum_{k=1}^{N} a_k log(t_k) \tag{2.4}$$

To compute the gradient of the loss function with respect to the weights, apply the chain rule to expand the partial derivatives. In a FFNN with one hidden layer, the gradient of the output layer can be written:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial a_k} \frac{\partial a_k}{\partial z_k} \frac{\partial z_k}{\partial w_{jk}}, \tag{2.5}$$

and the gradient of the hidden layer weights can be written

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial a_k} \frac{\partial a_k}{\partial z_k} \frac{\partial z_k}{\partial a_j} \frac{\partial a_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}}. \tag{2.6}$$

Each of the intermediate values in these expressions can be computed from equation 2.2 and the definition of the loss function. In some cases the result is very simple, for example in a neural network with linear activations and no hidden layers, using an MSE loss function, the gradient (equation 2.5) reduces to a straightforward expression:

$$\frac{\partial E}{\partial w_{jk}} = (a_k - t_k)\sigma_k'(z_k)a_j = (a_k - t_k)z_k a_j \tag{2.7}$$

2.1.5 Backpropagation and Gradient Descent

In a process called *gradient descent*, weights are updated in the negative direction of the gradient, which is multiplied by a learning rate η which determines the size of the step:

$$\Delta w_{jk} = -\eta \frac{\partial E}{\partial w_{jk}} \tag{2.8}$$

The weight updates are computed layer by layer, starting from the output and moving backwards. This algorithm is called *backpropagation* and was introduced in the 1980s [28].

As the backpropagation routine is computationally intensive, it is advantageous to forward propagate batches of data and perform weight updates based on averaged results, rather than doing so one sample at a time. This is called *batch gradient descent*. On the other hand, batch size is limited by memory, and the whole dataset usually cannot be forward propagated all at once. Using sub-samples of the dataset to update the network is called *stochastic gradient descent* (SGD) as the trajectory through the error landscape is stochastic, depending upon the random subsamples of data in the batches.

To solve a machine learning problem, the choice of algorithm and its related *hyperparameters* require careful consideration. Hyperparameters are variables affecting the a machine learning model's structure, and are usually not learned but rather set by the designer. In a feedforward neural network, they would include the number of layers, the size of the layers, the choice of activation function, choice of

error function, and the choice of optimization algorithm and its configurable parameters such as learning rate.

There are many pitfalls that one can encounter while iteratively optimizing a high dimensional objective function. For example, using too small a learning rate, it's possible to get stuck in a local error minimum or a plateau; too large of a learning rate, and the steps can completely skip over or climb away from the global minimum. Modern optimizers use adaptive learning rates for this reason. In fact, pure stochastic gradient descent as described in this section is rarely used in practice in such a simple form, but it is the basis for the more sophisticated optimization techniques commonly found in machine learning software packages today, such as Adam [29].

Another issue that deep neural networks encounter is the *vanishing gradient problem*, where the gradient becomes smaller and smaller as it passes backwards through the layers, and weights at the front of the network are practically no longer updated. A related issue is the *exploding gradient problem*, when the gradients are too large and result in an untrainable network. These are significant issues in neural networks with many hidden layers, and also in recurrent neural networks.

2.1.6 Recurrent Neural Networks

Recurrent neural networks (RNNs) are used for processing temporal or sequential information, where information in each time step has dependencies on activity of the past. There are many recurrent neural network architectures, all having in

common neurons, or groups of neurons, with feedback connections, as contrasted to the strictly feed forward designs discussed in the last sections. Recurrence gives the neural network memory, as information can circulate in a loop and persist.

The predominant approach to training recurrent neural networks is back propagation through time (BPTT), in which the state of the system at every timestep is unfurled into a feed-forward configuration as shown in figure 2.3. Then, conventional gradient descent and backpropagation is used to train the weights. BPTT is computationally intensive and notoriously suffers from exploding and vanishing gradients that make training unstable.

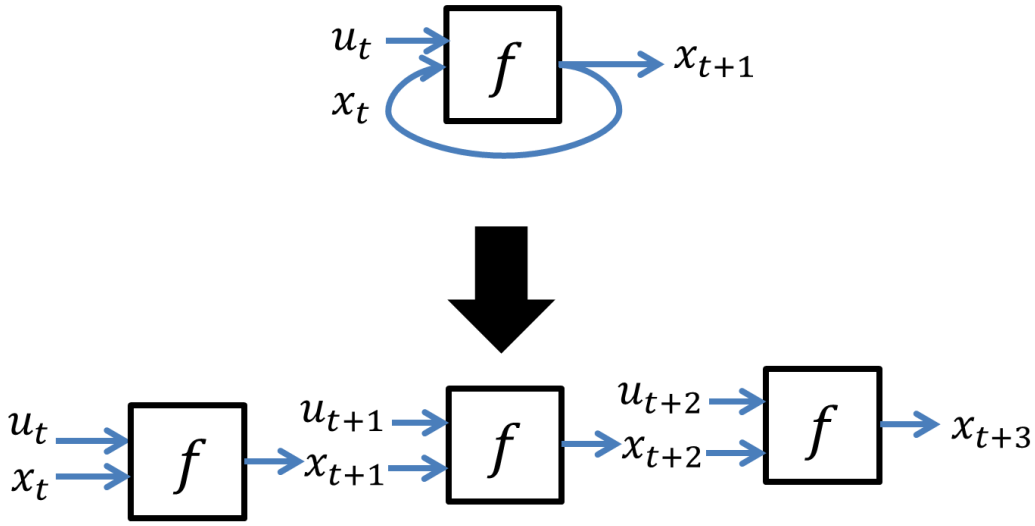


Figure 2.3: Unfolding of a recurrent neural network, with state x_t and input u_t, for backpropagation through time.

Today's most commonly used RNN architectures, long short term memory (LSTM) and the simpler gated recurrent unit (GRU), overcome the instability of BPTT by using much more complicated base units than the plain artificial neuron. Both of their recurrent computational units have the ability to keep or forget in-

formation, allowing them to retain long-term temporal associations in a way that a simple RNN cannot. These architectures have made great strides in fields reliant upon sequential data such as natural language processing. However, the LSTM and its variants are cumbersome to use and train, and are very computationally demanding.

2.2 Reservoir Computing

A specialized neural network architecture that has shown promise in rapid time-series prediction is called the reservoir computer (RC), shown in figure 2.4. It is a machine learning model independently proposed around the same time as an echo state network (ESN) by Jaeger [30] using conventional artificial neurons, and as a liquid state machine (LSM) by Maass et al. [31] using more biologically plausible spiking neurons. Reservoir computing is a promising architecture for a variety of time series prediction tasks that rely on short-term associations.

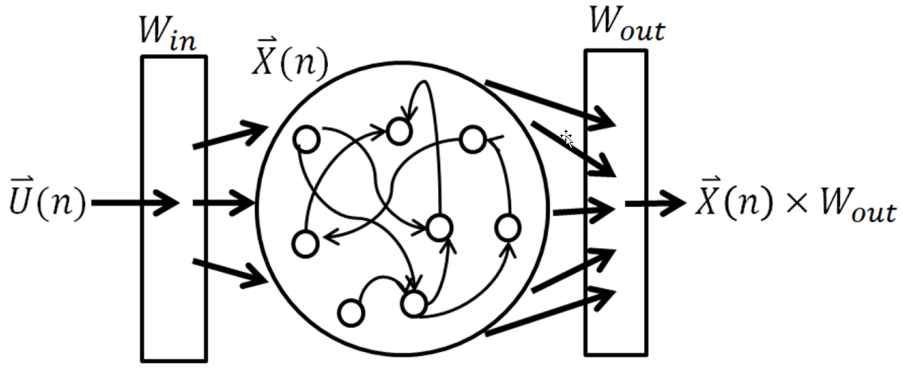


Figure 2.4: The reservoir computer. The reservoir is the fixed, randomly initialized recurrent neural network whose internal states at time n are $\vec{X}_n$. The input layer W_{in} is also fixed and randomly set. The output layer W_{out} is trained using a simple rule such as linear regression.

Reservoir is the term for the RNN inside of the RC, which has random connec-

tivity and random weights. To avoid the computationally intensive step of training the weights W within the RNN, the reservoir is left fixed, and its state is put into an output layer whose weights W_{out} are trained. As are many machine learning methods, reservoir computing is loosely brain-inspired; there is evidence that neuron groups in the prefrontal cortex [29] and the cerebellum [30] are reservoir-like.

The internal reservoir states at timestep n can be described as follows:

$$\vec{X}(n) = f\left(W_{in}\vec{U}(n) + W\vec{X}(n-1)\right) \tag{2.9}$$

where f is the transfer function of each node, often chosen to be a tanh function which constrains the node output to within [-1,1]. W_{in} is uniformly sampled from $[-\omega, \omega]$, where ω is one of the hyperparameters used in the designs, determines the strength of the input(s) and which nodes they are coupled to, which can be all of the nodes, or a subset of them. The external input $U(\vec{in})$ can be univariate or multivariate. The adjacency matrix W is populated by random numbers in the range [-1,1] and has both weights and connectivity information.

Some implementations also vary the timescale of internal reservoir activity using leakage parameter α, which determines how much of the reservoir state is carried over to the subsequent timestep:

$$\vec{X}(n) = (1-\alpha)\vec{X}(n-1) + \alpha f(W_{in}\vec{U}(n) + W\vec{(}X)(n-1)) \tag{2.10}$$

The reservoir output Y is simply the reservoir state multiplied by the output

layer weights:

$$\vec{Y}(n) = W_{out}\vec{X}(n) \tag{2.11}$$

Any learning rule can be applied to calculate the weights of W_{out} in training. The simplest is linear regression, which minimizes the mean squared error and has the a closed-form solution:

$$W_{out} = (\vec{X}^T\vec{X})^{-1}\vec{X}^T\vec{Y} \tag{2.12}$$

Alternatively, the output weights may be updated iteratively using gradient descent using equation 2.7.

To avoid overfitting, Tikhonov regularization may be used, in which the L2 norm of the coefficients is added to the cost function, and the closed-form solution becomes:

$$W_{out} = (\vec{X}^T\vec{X} + \eta I)^{-1}\vec{X}^T\vec{Y} \tag{2.13}$$

where η is the Tikhonov parameter which determines the magnitude of the smoothing effect, and I is the identity matrix. As an alternative to adding a penalty to the cost function to prevent overfitting, noise may be added to the training data.

The reservoir computer is often used for time series prediction, but can also be used for classification tasks. For time series prediction, the training target is the value of the series one step ahead, $\vec{X_{n+1}}$. After sufficient training steps, when the

reservoir has adequately learned the transfer map from one time step to the next and the training loss is low enough, the output $\vec{Y_n}$ is fed back into the reservoir input for a free-running prediction, as shown in figure 2.5.

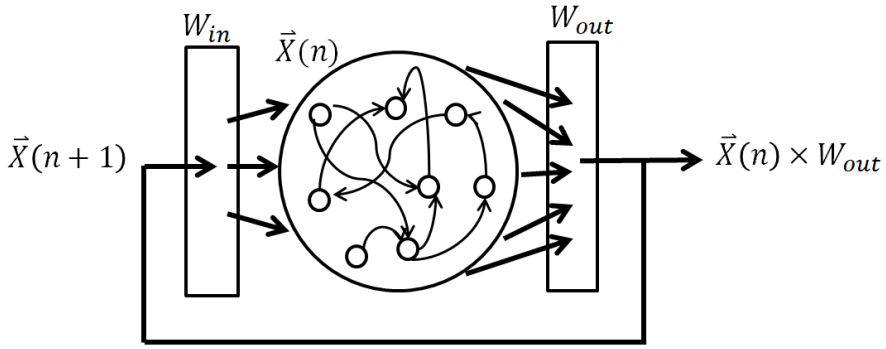


Figure 2.5: Reservoir configured for free-running prediction.

2.2.0.1 Reservoir Hyperparameters

In a software reservoir computer, the main hyperparameters are:

- **Reservoir size,** N: the number of neurons in the recurrent neural network. N should be chosen so that it is much larger than the number of inputs.
- **Connection probability,** p: the probability of two nodes being connected.
- **Input scaling coefficient,** ω: the amount by which inputs are scaled.
- **Spectral radius,** ρ: the magnitude of the largest eigenvalue of the adjacency matrix. This determines the overall magnitude of activity within the network. After random initialization of W, the entire matrix is scaled so that the spectral radius is the desired value, typically slightly below 1.
- **Memory leakage parameter,** α: a scalar determining how much of the previous state of the network is retained in every step.

- **Tikhonov regularization parameter,** η: A penalty added to the loss function to prevent the regression algorithm from achieving the global minimum, which would cause overfitting.

The reservoir is a complicated nonlinear structure for which there is little in the way of mathematical understanding to systematically guide choices of hyperparameters. Why a particular setup yields a good or poor result on a particular machine learning problem remains an active research area. For example, a study of the effect of spectral radius on the prediction of several complex nonlinear systems found that there was an optimal range for ρ, but admitted no analytical insight as to why those values were found [32].

To illustrate the interrelated effects of hyperparameters, the following figures (2.6, 2.7, and 2.8) are the states of a 100-node reservoir as it evolves through 100 timesteps with no external forcing, meaning that there is no external input data. In each of the figures, connectivity is fixed, connection probability p is fixed, and memory leakage parameter α and spectral radius ρ are set to three different values. The left-hand side column of each figure represents the states of each of the 100 nodes in the network, and in each figure this starting point is the same. The network evolution through time can be seen in the figure from left to right.

General patterns emerge from examining the networks' self-excited behavior. Reservoirs with a greater leakage parameter α retain less memory of prior network states, thereby having faster temporal dynamics than networks with smaller α. Smaller α means a longer settling time in networks that settle into a fixed state,

or lower frequency in networks that settle into a periodic pattern. A larger spectral radius ρ of the adjacency matrix results in greater overall network activity. A larger connection probability p typically results in steady states of larger magnitude while small values of p generally ensure that the network activity dies down to zero.

However, all of these observations are only guiding principles, and networks with different adjacency matrices can exhibit very different behavior. In practice, a particular choice of parameters must be empirically tested to determine whether a reservoir will perform well at a certain task.

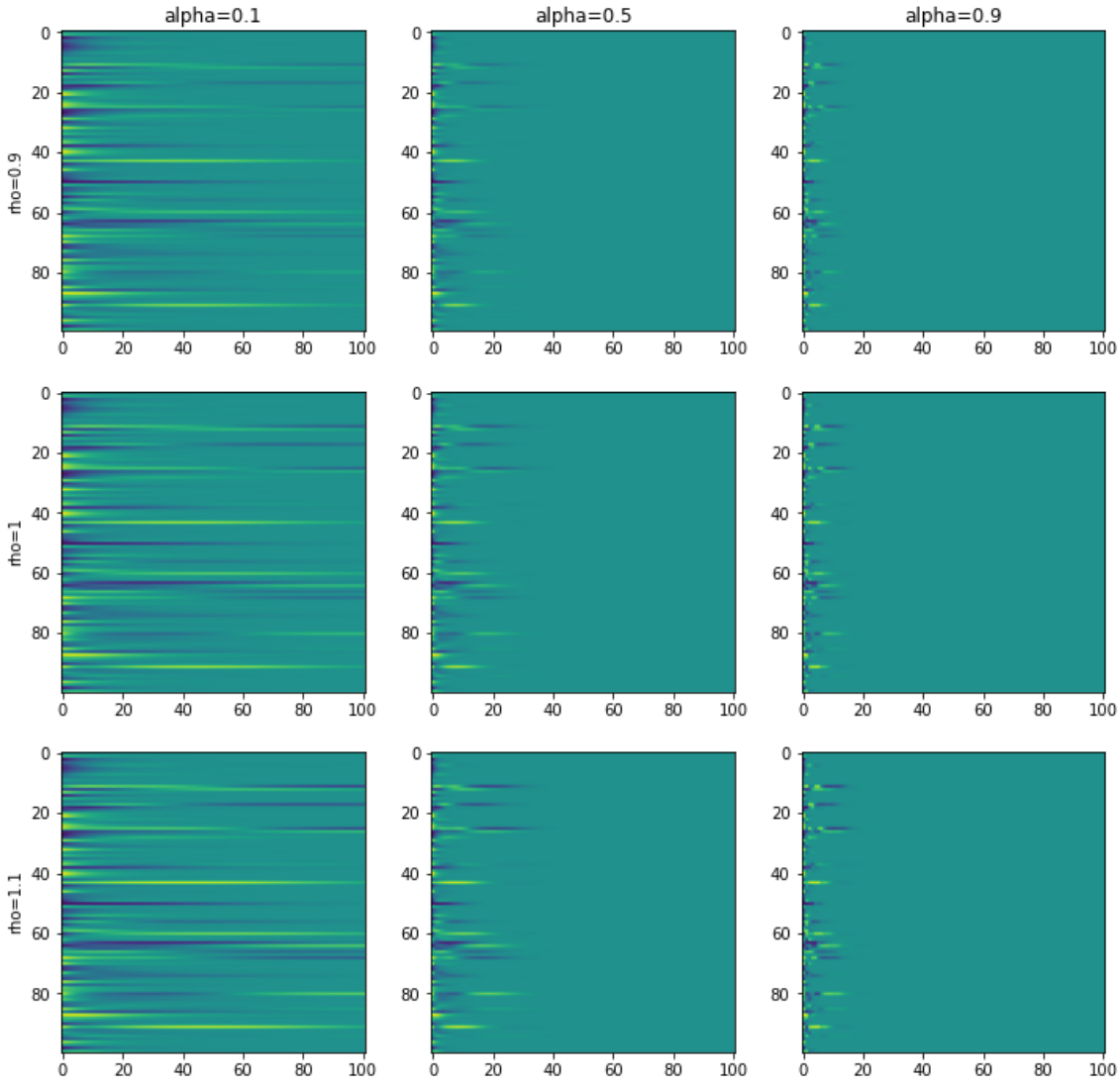


Figure 2.6: Self-activity of a 100-node software echo state network with connection probability p=0.01 with various spectral radius ρ and memory parameter α values. The vertical axis is node number, and the horizontal axis is time.

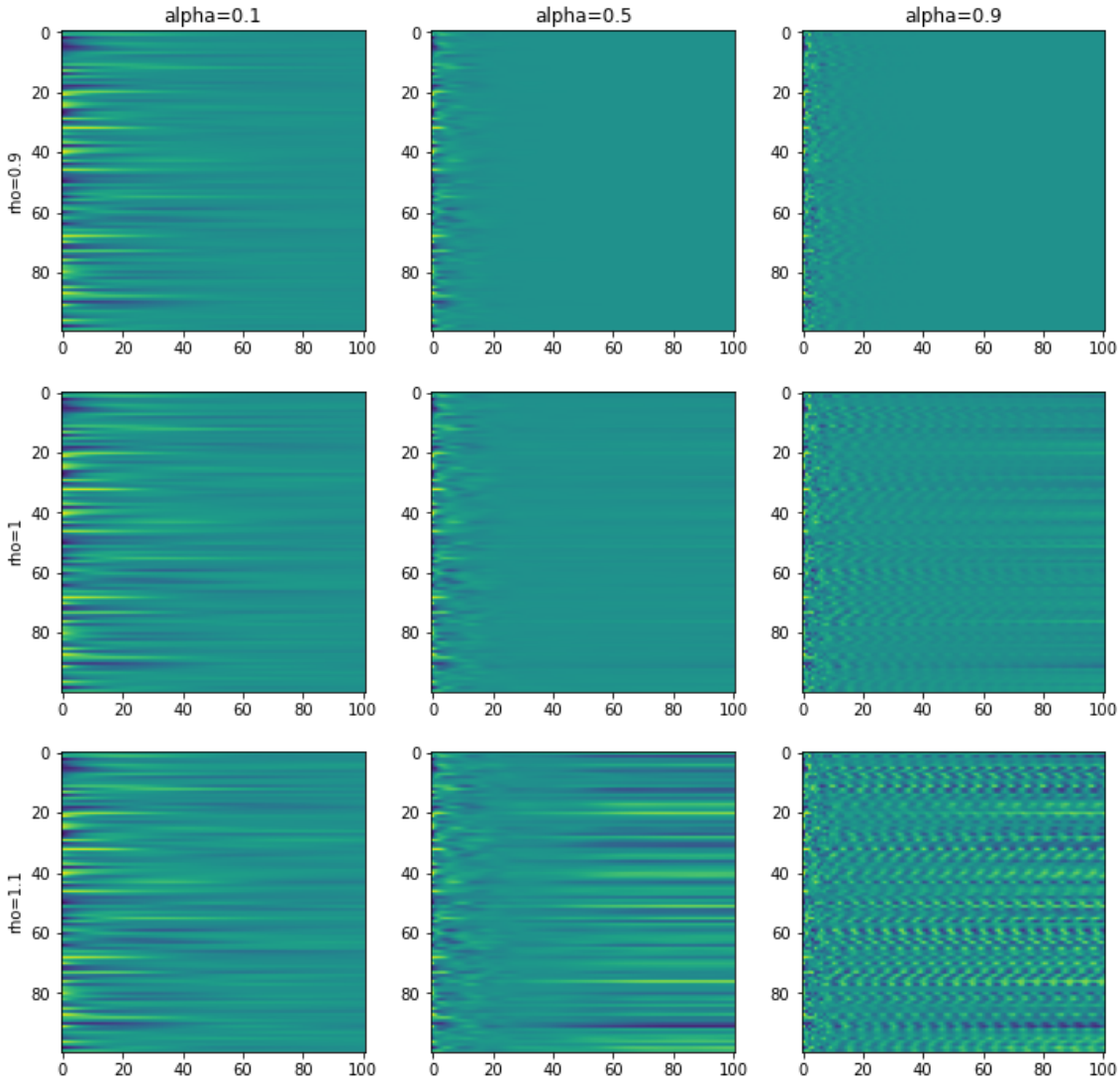


Figure 2.7: Self-activity of a 100-node software echo state network with connection probability p=0.05 with various spectral radius ρ and memory parameter α values. The vertical axis is node number, and the horizontal axis is time.

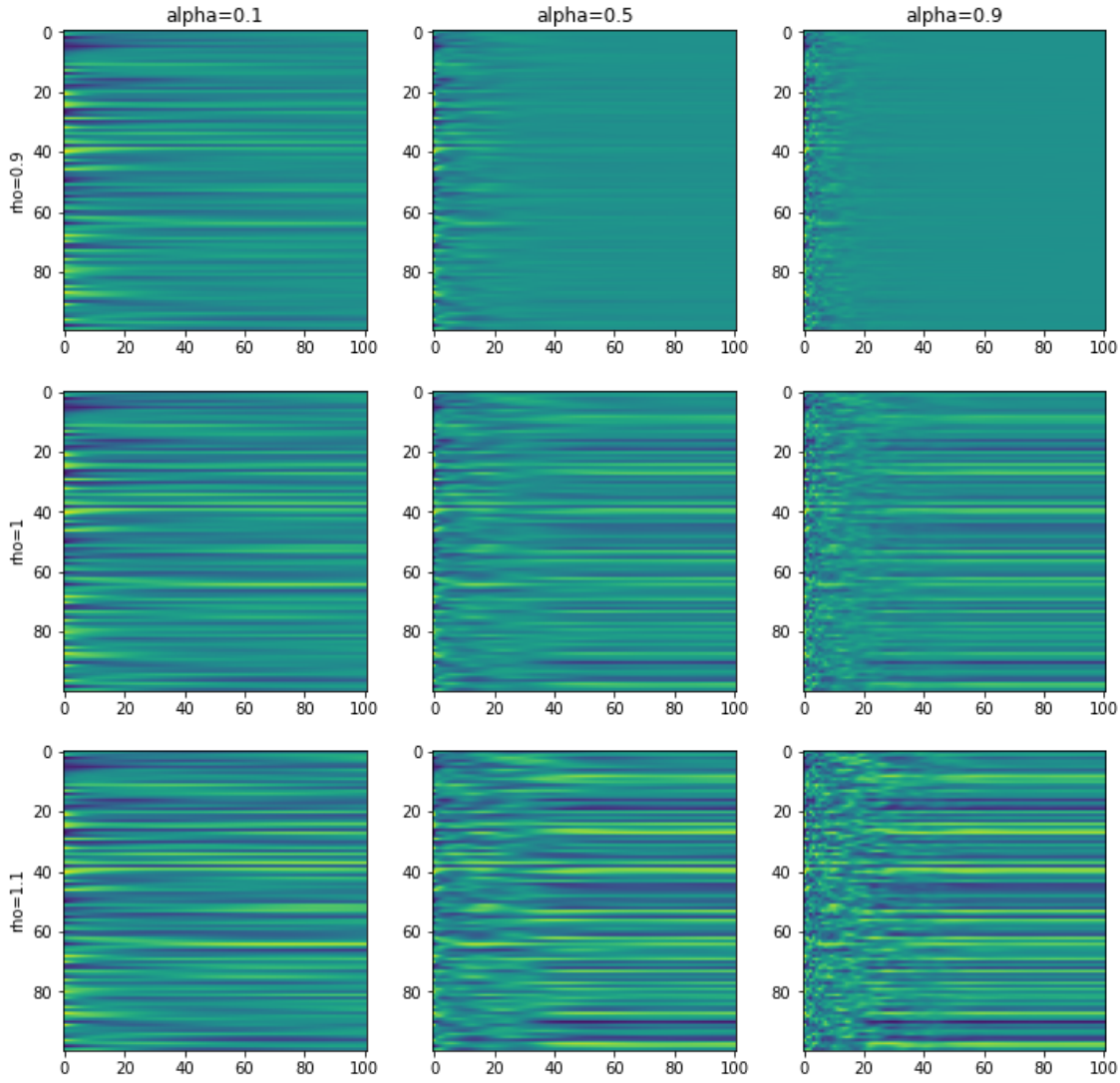


Figure 2.8: Self-activity of a 100-node software echo state network with connection probability p=0.5 with various spectral radius ρ and memory parameter α values. The vertical axis is node number, and the horizontal axis is time.

From the perspective of data processing, there are several complementary interpretations of the reservoir's function:

- As **short-term memory**. Due to recurrence within the reservoir and the scaling of the spectral radius to be slightly below 1, information circulates for some time before being reduced asymptotically to zero. This is the reason

for the "echo state" name — the network has fading "echoes" of the previous inputs. Jaeger et al. [33, 34] subjected ESNs to various benchmarking tasks and characterized the networks' memory capacity, finding that the number of timesteps of memory was generally $\leq$ N. The reservoir's function goes beyond simple memory, however: in a comparative study, the ESN outperformed a tapped delay line with no recurrence [35].

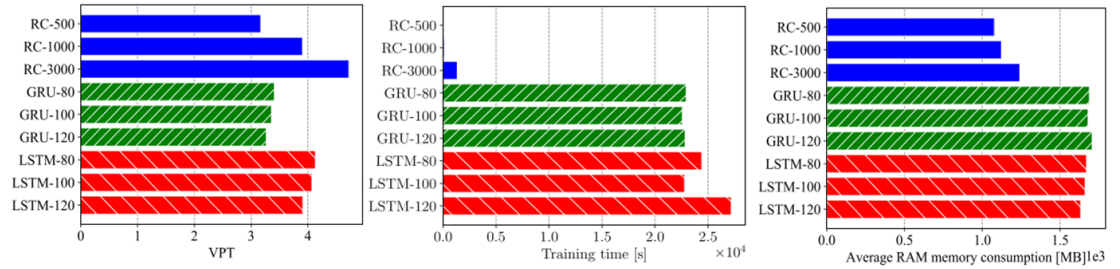
- As a **kernel expansion** of features into higher-dimensional state space. Kernel methods are often used in combination with support vector machines or principal component analysis, to transform data into a space in which it become linearly separable. These methods use a mathematical trick that avoids explicit computation in the higher-dimensional space. Popular kernels include Gaussians, radial basis functions, or polynomials. Choosing which kernel to apply to data is often a matter of educated or blind guessing. The reservoir effectively acts as a random temporal kernel, nonlinearly casting the input and its recent history to a much higher-dimensional representation. The linear readout then chooses which parts of the transformation produce a useful result. Maass et al. [36, 37] explore the kernel properties of LSMs. They propose that the separation of inputs that a neuronal circuit creates could be an empirical measure for its quality as a kernel — in other words, the complexity and diversity of nonlinear operations carried out by a neuronal circuit on its input stream in order to boost the classification power of a subsequent linear classifier.

- As a **universal function approximator**. The universal approximation theorem is often cited as the reason for the wide-ranging utility of neural networks: it states that any continuous real-valued function can be approximated by feed-forward neural networks with nonlinear activation functions. In 1989, Cybenko showed that this holds true for networks of arbitrary width with only a single hidden layer, with sigmoidal activations [38]. In 1991, Hornik generalized the result to networks with any smooth nonlinear activation function [39]. Recently, it was shown that any fading memory system in discrete time can be realized as a simple finite dimensional neural network-type state-space model with a static linear readout map, and that echo state networks are also universal uniform approximants [40].

2.2.1 Comparison to Other RNN Methods

As mentioned previously, notable competing methods to reservoir computing are long short-term memory (LSTM), and gated recurrent units (GRUs), both of which are trained with the backpropagation through time (BPTT) algorithm. While LSTMs and GRUs work very well for some tasks, reservoirs outperform them in dynamical system prediction. Gallicchio et al. found that ESNs and a sequence of serial ESNs (which they call DeepESN, also described in section 2.2.3.3) provided comparable accuracy as LSTMs and GRUs on a polyphonic music prediction task, while requiring much less computation time on the same hardware [41]. Figure 2.9, reproduced from [42], compares RC, GRU and LSTM networks for dynamical

system prediction tasks, and found that reservoir computing resulted in longer valid predictions after vastly less training time and less RAM used. The short training time in particular makes reservoir computing shine in adaptive control scenarios when the reservoir needs to be quickly re-trained to learn the dynamics of new data.



(a) Valid prediction time in the test dataset

(b) Training time

(c) Average RAM memory requirement

Figure 2.9: Comparison of RC, LSTM and GRU networks for dynamical system prediction tasks. (a) shows the amount of time for which the network makes valid predictions after being trained. (b) shows the training time, and (c) shows the average RAM requirement to run the networks. Reproduced from [42].

2.2.2 Reservoir Computing Applications

The area in which reservoir computing has yielded the most impressive results is in the modeling of dynamical systems. It indeed makes sense to use a dynamical system to simulate a dynamical system.

In particular, the results of modeling chaotic systems with reservoirs have shown success. Chaotic systems can often have simple underlying equations, but exhibit extraordinarily complex behavior. They are characterized by sensitivity to initial conditions and having behavior that can appear to be random, but is in fact deterministic. Complex behavior comes about from thorough topological mixing and dense periodic orbits in phase space.

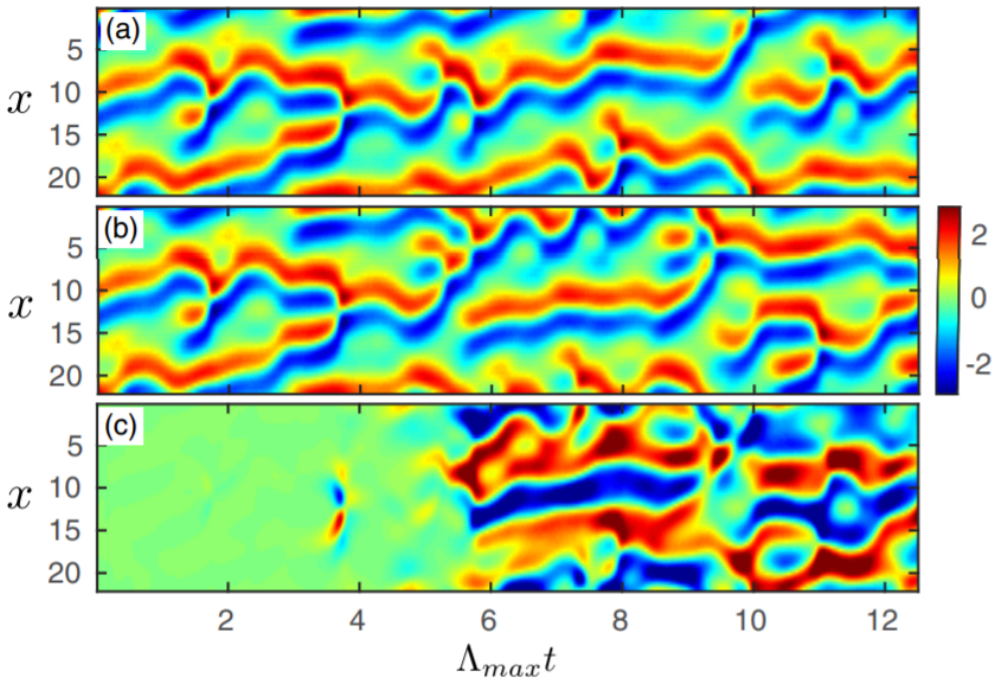
Reservoirs have been used with much success for state of the art chaotic time series prediction [43], replication of chaotic attractors [44], and chaotic time series separation [45]. Notable chaotic time series prediction results are shown in figure 2.10. That is a prediction test of the Kuramoto-Sivashinsky system, described by the following partial differential equation (which has an additional spatial inhomogeneity term):

$$y_t = yy_x + y_{xx} - y_{xxx} + \mu cos(\frac{2\pi x}{\lambda}) \tag{2.14}$$

Twenty four time series of the spatiotemporal system are plotted in figure 2.10, as well as the free-running reservoir prediction of the system and the difference between the prediction and the solution. The horizontal axis is in units of Lyapunov time, which is a measure of the exponential divergence of the trajectories.

In addition to making accurate chaotic time series predictions for more Lyapunov times than any other existing method, the reservoir can capture the overall behavior of the system, producing a time series that looks realistic even after it departs from the true solution (contrast this to the solution diverging or going to zero). This accurate capture of the climate of the system enables reconstructing the spectrum of Lyapunov exponents and understanding the attractors of the system—straightforward operations when the equations of motion are accessible, but extremely difficult from finite time series for systems such as Kuramoto-Sivashinsky.

Reservoir computing has also been used for the following non-exhaustive list of applications:



.

Figure 2.10: Spatiotemporal plots of the Kuramoto-Sivashinsky (KS) system and reservoir prediction. (a) Is the KS system, (b) is the reservoir prediction, and (c) shows the difference between them. The x-axis is in units of Lyapunov times. The Lyapunov time is a measure of the departure of the system from initial conditions by a factor of e. Reproduced from [43]

- Wireless signal recovery [46]
- Seizure detection from EEG signals [47]
- Pattern and signal generation [48]
- Phoneme recognition [49]
- Million words per second spoken digit classification [50]
- Control systems [51, 52]
- Human action recognition [53]
- Image recognition and radio transmitter classification from time series [22]
- Particle accelerator beam trajectory prediction [27], also described in this the-

sis.

2.2.3 Reservoir Computing Variants

2.2.3.1 Single-node time-delay reservoirs

A single processing element with time-delayed feedback forms a delay-dynamical system that can be used as a reservoir. First proposed by Appeltant et al., [54], the setup relies on time-multiplexing with a single neuron, rather than the use of multiple neurons. The neuron's output is fed through a delay, and then back into the input to be combined with new data. In that way, the neuron's state at any time contains traces of the entire history of the signal. Neuron states throughout time are saved and then put through an output layer. This differs from a tapped delay line because the data is put through the nonlinear processing element multiple times, and allowed to mix in time. Photonic reservoirs are usually of this type, as electron-photon conversion is inefficient so processing element reuse makes sense. Additionally, optical processing often happens faster than I/O making time multiplexing practical. Hart et al. [55] and Brunner et al. [56] analyze the dynamics of some time-delay optical reservoirs.

2.2.3.2 Parallel reservoirs

Some spatiotemporal systems, including the Kuramoto-Sivashinsky system, have strong local interactions. Figure 2.11 shows parallel reservoirs that take inputs from a sliding window of the input time series. Taking inspiration from the slid-

ing window of a convolutional neural network, this configuration emphasizes local interactions between points in space or time.

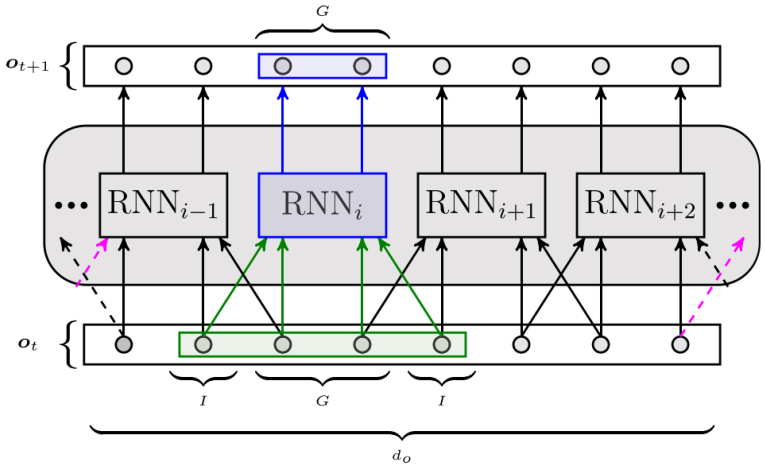


Figure 2.11: Parallel reservoirs taking inputs from a sliding window over the time series. This configuration emphasizes local interactions between points in space or time. Reproduced from [42].

2.2.3.3 Deep reservoirs

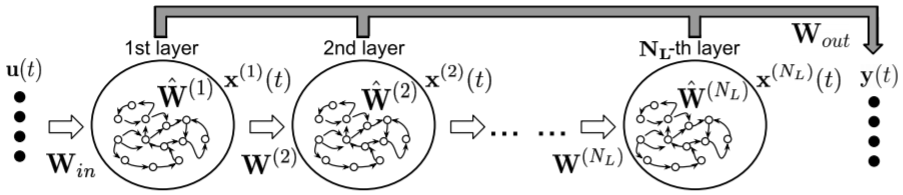


Fig. 1: Hierarchical architecture of DeepESN.

Figure 2.12: Serial reservoirs, called DeepESN, Reproduced from [41].

Reservoirs arranged in a serial configuration, with outputs of one going into inputs of another, are called deep reservoirs or deepESNs. The outputs of all reservoirs are used to compute the result. Gallicchio et al. [41] explored the properties of several different deepESN configurations, finding some advantages in accuracy

and better processing at different time scales, at the expense of introducing more complexity to the model.

2.2.3.4 Extreme Learning Machine

A related architecture is the extreme learning machine (ELM), which is not a reservoir as it lacks recurrence, but it is rather a shallow feed-forward network also exploiting random initialization [57]. The ELM has inputs connected to a single layer of hidden neurons via random weights. The weights between the hidden neurons and output neurons are then trained. Linear output neurons enable linear regression to be used, which makes the training process very fast. Despite their simplicity, ELMs and their variants perform comparably to deep neural networks while requiring much less training time [57, 58].

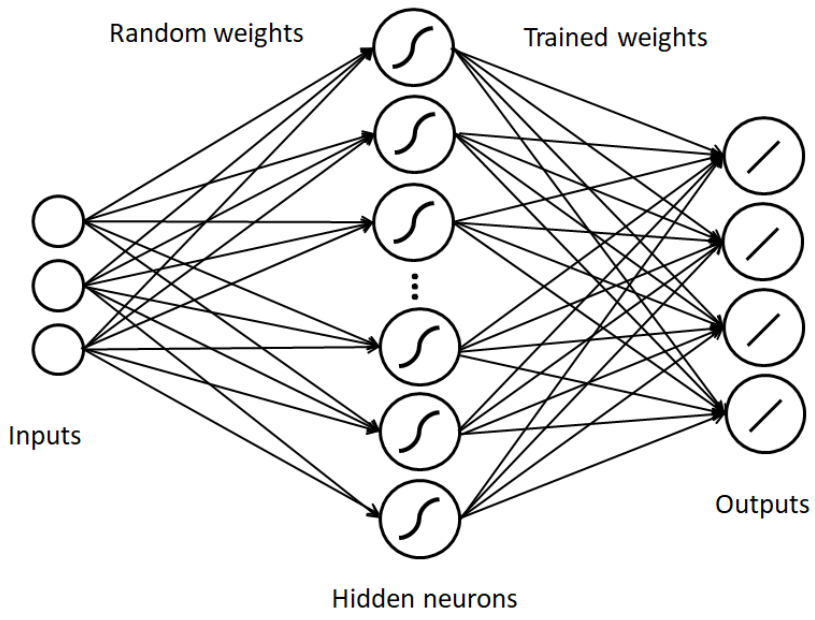


Figure 2.13: The extreme learning machine (ELM). Input-to-hidden connections are randomly chosen, and hidden-to-output weights are learned. Linear output neurons make training very fast.

2.3 Reservoir Computing Using Physical Systems

The behavior of real-world systems can be used for information processing. Reservoir computing is arguably an ideal use case for leveraging the properties of physical nonlinear dynamical systems to perform computation. The reservoir's function is to achieve a higher-dimensional representation of input data, mixed in time, and how exactly it achieves that goal does not necessarily matter. In software it is done by setting up a recurrent neural network with artificial neurons and explicitly computing each node's value, step by step. Nonlinear physical systems, subject to nonlinear differential equations governing their behavior, have been shown to also effectively act as reservoirs, with the promise of exceeding software models in speed and power.

A number of physical systems have been studied for reservoir computing, including memristor, spintronic, optical, and soft body reservoirs [59]. However, comparatively evaluating the performance of physical substrates is an active area of study. In this section we give an overview of the current research.

2.3.1 Physical Implementation of Reservoirs

Theory predicts that any physical system that contains the following properties can function as a reservoir [59]:

- high dimensionality,
- separation of inputs,

- fading memory,
- nonlinearity,
- deterministic dynamics.

2.3.1.1 High Dimensionality

As discussed previously, the reservoir nonlinearly casts a spatiotemporal signal to a higher-dimensional state space to enhance the function of a simple classifier. A physical dynamical system can perform this function as long as the dynamical dimensionality of the system is sufficiently large and repeatable to yield a useful kernel transformation.

2.3.1.2 Separation of Inputs

Separation of inputs means that slightly different inputs result in distinguishable network states. For this reason, reservoirs are designed to operate at the edge of chaos or near a phase transition: inputs push the network over the edge into a state of heightened activity whose state is very sensitive to inputs. The term edge of chaos emerged from the study of cellular automata, where it was found that conditions for information transmission, storage, and modification, are achieved in the vicinity of a phase transition between high and low activity [60]. In fact there is evidence of brain similarly operating at a point of dynamic instability, which in neuroscience is called the critical brain hypothesis [61].

2.3.1.3 Nonlinearity

Finally, nonlinearity in the system is necessary for the approximation of nonlinear functions. In a physical system, the nonlinearity can come about from effects such as higher order terms in a spring constant, an irregularly shaped optical cavity, or chaotic mixing in the time domain.

2.3.2 Mechanical Reservoirs

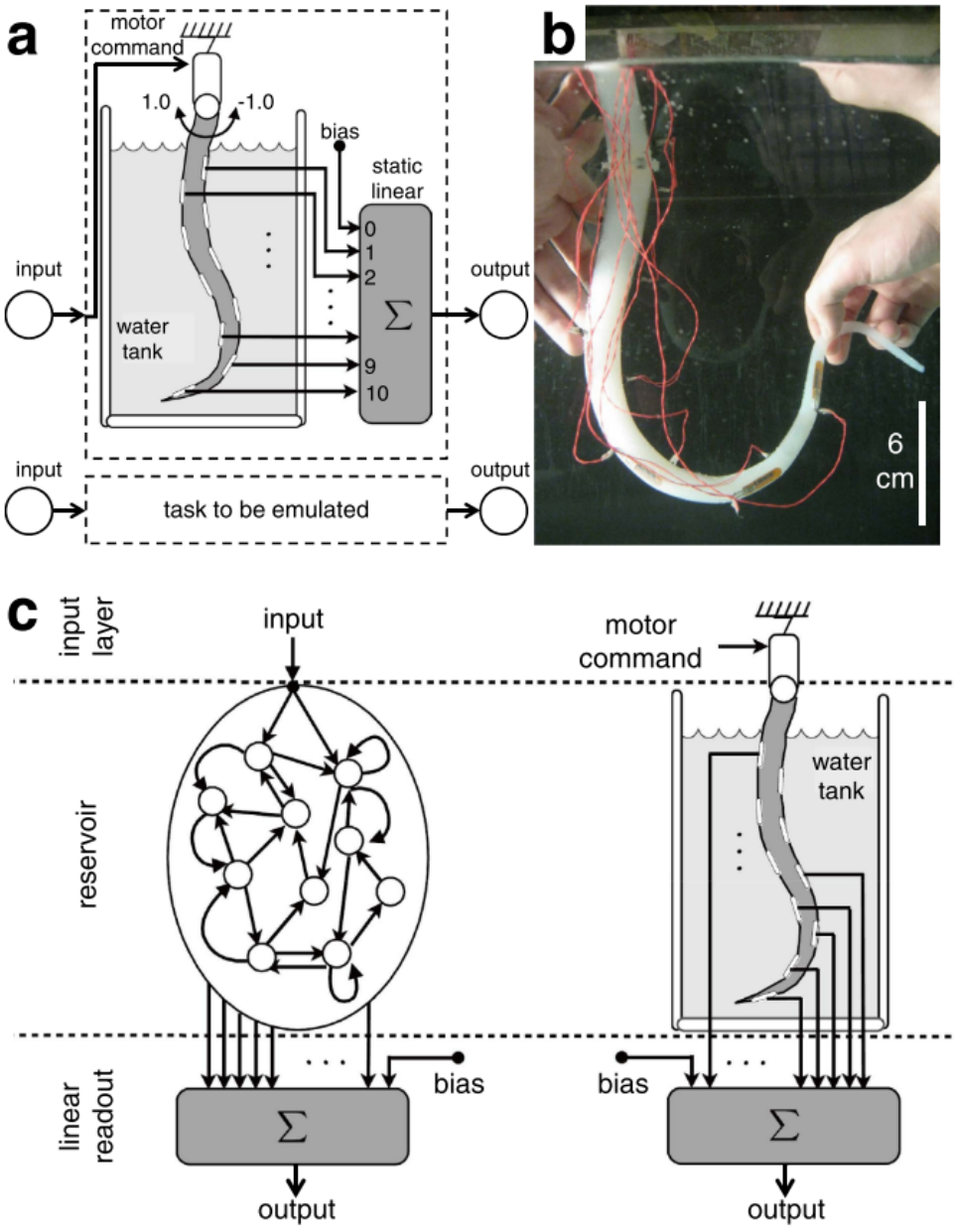


Figure 2.14: Example of a soft body being used as a reservoir. An actuator at the top moves the silicone tentacle, and the readout map is applied to bend sensors along its length. Reproduced from Nakajima et al. [62].

Mechanical systems are attractive candidates for morphological computation: the outsourcing of processing to inherent functions of the body [63]. The theory

of how mass-spring systems could be used for reservoir computing was developed by Hauser et al. [64]. Dion et al. demonstrated spoken word recognition with a micro-electromechanical cantilever exhibiting nonlinearities [65]. Tensegrity structures have been controlled by using their own actuator readouts as reservoir node states [66]. Perhaps the least practical, but most amusing example is the paper by Fernando et al. entitled "Pattern Recognition in a Bucket", in which wave patterns in a bucket of water were used as a reservoir [67].

Soft robotics takes inspiration from the highly compliant materials that abound in biology to create flexible and adaptable robots. Controlling the many degrees of freedom of a soft body is a challenge for locomotion and control. However this can be repurposed as a computational resource. Figure 2.14 shows a silicone tentacle as a reservoir computer, which was able to perform a nonlinear auto-regressive moving average (NARMA) time series benchmarking task [62]. A motor at the top of the body provided the input stimulus, and bend sensors along the body were used as readouts of the reservoir state.

2.3.3 Optical Reservoirs

Optical computers have been considered since the invention of the laser in the 1960s. Substantial challenges to optical computing must be overcome if it will ever be a serious contender to electronic processors: electron to photon conversion (and vice versa) is quite inefficient; an optical transistor that doesn't distort data remains elusive; optical memory is nonexistent. However, optical processing is attractive for

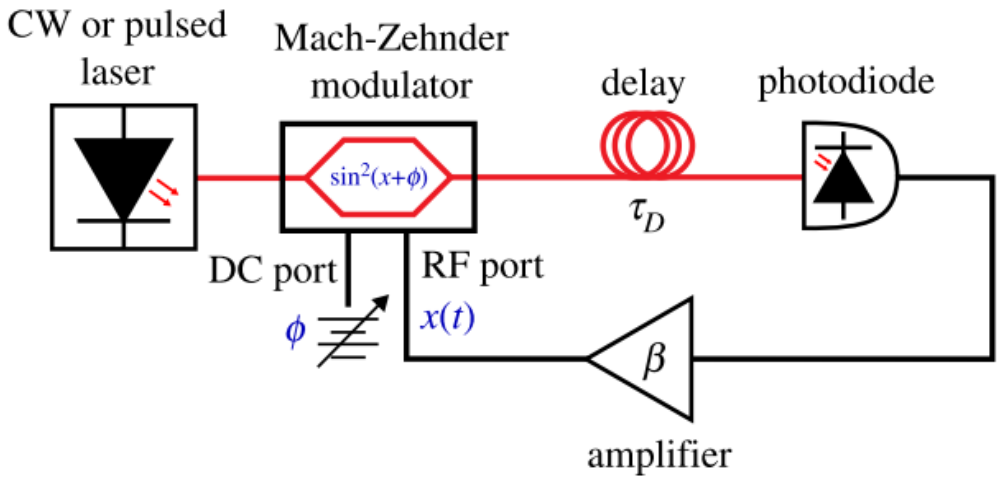


Figure 2.15: A schematic of an optoelectronic time-delay reservoir implementation, reproduced from Hart et al. [55].

neural network computation as operations on a light beam can be highly parallel, parts of the beam can travel without crosstalk, and linear operations can be performed with minimal energy cost [68]. It can be argued that the only way machines will ever approach the cognitive capacity of the human brain is via photonic interconnects which efficiently fan out neuron outputs to thousands of other neurons [69]. Photonic neural network accelerator chips have been explored for decades [70].

Optical RCs have been made using networks of waveguides, splitters, and combiners; using signal-mixing cavities [71]; diffractive resonator mixing laser signals [72]; systems of nonlinearly coupled lasers [73], and single-node time-delay feedback loops [50, 55, 65]. Figure 2.15, reproduced from [55], shows an optoelectronic time-delay reservoir implementation using a Mach-Zender modulator, which performs amplitude modulation on an optical signal. Laser light enters the modulator, whose gain is controlled by the voltage $x(t)$. The light is then passed through an optical fiber delay line with delay τ_D. The photodiode converts the light back to an

electronic signal, which amplified by gain β. The amplifier output is then used to modulate the Mach-Zender interferometer, completing the feedback loop.

2.3.4 Electronic Reservoirs

Electronic reservoirs arguably have the greatest potential to be easily adopted into existing hardware workflows as machine learning accelerators if they do not rely on exotic materials or novel manufacturing methods. Some prior work has focused on more efficiently implementing either static or spiking artificial neurons in dedicated hardware. For example Schrauwen et al. [74] who used a FPGA to implement a liquid state machine with spiking neurons and Canaday et al. [51] who used an ESN-variety reservoir computer implemented on an FPGA. Penkovsky et al. [75] performed a genetic algorithm optimization of a spiking reservoir before transferring the configuration to an FPGA for fast processing.

Other electronic approaches instead used single nonlinear elements in a time-delay reservoir approach. An example in hardware was performed by Soriano et al. [76] who simulated the Mackey-Glass system using a single BJT as a nonlinear element, with a DAC to convert the input to analog, then an ADC at the end to convert back to digital for output layer computation on a computer. Jensen and Tufte [77] used a Chua oscillator, a simple chaotic circuit with only four components, and applied a driven input. To make a system with very low hardware resource needs, Alomar et al. [78] took a probabilistic computing approach and implemented a stochastic reservoir computer on an FPGA.

Yet other implementations used the behavior of electronic networks for reservoir computing, without specifically using artificial neurons. Haynes et al. [79] used a single XOR gate in a feedback configuration in a time-delay reservoir. Canaday et al. and Shani et al. have both used FPGA-based dynamical networks as reservoirs, demonstrating success in time series classification and prediction [22, 80]. Recently, Komkov et al. [23] demonstrated radio frequency classification with about one-tenth of the trainable parameters of a state of the art convolutional neural network using careful preprocessing and an FPGA-based Boolean reservoir.

Chapter 3: Boolean Network Theory

3.1 Boolean Network Dynmics and Circuits

The field of network science studies networks of all kinds, such as social networks, neural networks, communication networks, and biological networks, to name a few. Boolean networks were originally proposed in the 1960s by Kauffman as a very simple dynamical model of gene expression, in which genes can be either on or off, and are affected by neighboring genes. Kauffman investigated the properties of randomly connected Boolean networks, such as the length of cycles within the network, and drew parallels with cell replication cycles [81]. More sophisticated mathematical formalism to describe Boolean networks was developed later [82, 83]. Boolean networks can evolve in discrete steps, or be continuously updated, which has implications for their behavior and the quality of the analogy they make to real-world systems.

3.1.1 Boolean Networks Implemented in Digital Circuits

While ideal Boolean networks can exhibit complex and emergent behavior, they obey the well-behaved rules of Boolean algebra. The same networks imple-

mented in circuitry are subject only to the laws of physics, which can make their behavior much less Boolean. A digital gate has a finite rise and fall time, a possibly nonlinear transfer function, and will cause a pulse to incur delay and distortion as it travels from input to output. If the duration of the pulse is comparable to the gate's delay time, the pulse may spend most of its time in the intermediate voltage range, an ambiguous logical region.

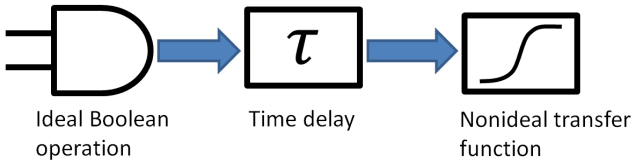


Figure 3.1: Nonidealities of real Boolean logic gates. Physical devices have rise and fall times, and large but finite amplitude gains. They also have a low-pass filtering effect, and pulses that are too short will be rejected.

For computer processors to have high reliability and noise immunity, they use the digital logic abstraction, which discretizes the available voltage range into only high and low values. A voltage below a certain threshold represents a zero, and above another threshold represents a one, a and voltage in the "forbidden region" in between is considered ambiguous. To ensure robustness to variations in delay times of logical blocks, digital computing usually uses synchronous logic, reading the outputs of combinational logic blocks upon transitions of a global clock. While clocking ensures robustness to timing variations and prevents race conditions, it slows down overall computation.

3.1.2 Unclocked Recurrent Boolean Circuits for Reservoir Computing

Because machine learning algorithms are trained from real-world examples rather than by using explicit rules, these algorithms are good candidates to be accelerated in hardware in which results are not always repeatable. In machine learning, training data is often inherently noisy or distorted when it comes from physical measurements (in fact, a method of preventing overfitting in a machine learning model is to add noise to its training data). Because they are robust to noise, machine learning algorithms can take advantage of denser information encoding that comes from quantizing the available voltage range more finely.

Beyond the dynamics of pure Boolean networks, circuit implementations introduce further useful time delays and nonlinearities. As is shown in figure 3.1, a pulse traveling through a digital logic gate will incur a time delay which gives the network some useful memory. The signals may also become distorted, depending upon the gate's voltage transfer function and transition times. These effects strongly depend upon the supply voltage and network topology and sensitivity.

A Boolean logic gate reservoir is made by connecting digital logic devices into recurrent configurations and letting them run freely. Signals in the reservoir have frequencies determined by the device delay time, and the voltage spends much of its time in the intermediate region between 0 and the supply voltage.

Unclocked recurrent digital logic networks are of particular interest for reservoir computing because their continuous-time dynamics result in an infinite number

of available states. Thus, they may exhibit chaos. Clocked recurrent digital logic networks have only a finite number of available states, and therefore must be either stationary or periodic.

A chaotic system has a strong dependence upon its initial conditions, meaning that two slightly different inputs will result in a large difference in the system's state as it evolves through time. This can be exploited for machine learning classification problems, in which the output map applied to the reservoir state (or a subset thereof) must distinguish between classes.

3.1.3 A Simple Unclocked Boolean Circuit: the Ring Oscillator

The simplest example of a unclocked recurrent digital logic gate circuit is a single inverter (logical NOT) with feedback. The input connected to the output results in a logical contradiction (TRUE cannot equal FALSE), however when implemented as a circuit, a TRUE value at the input appears as a FALSE value at the output only after the high-to-low transition time. Likewise, FALSE at the input appears as TRUE on the output only after a low-to-high transition time. As a result, a single inverter with feedback creates a ring oscillator whose period is the average of the two transition times—in the parlance of nonlinear dynamics, this is a limit cycle attractor. In practice a single inverter with feedback may also get stuck at half the supply voltage—a fixed point attractor. Chaos has even been exhibited in simple ring circuits [84]. Boolean network circuits in the chaotic regime have been studied as sources of random numbers [85, 86]. A ring oscillator may be built out of any

odd number of inverters; a ring oscillator with three inverters is shown in figure 3.2.

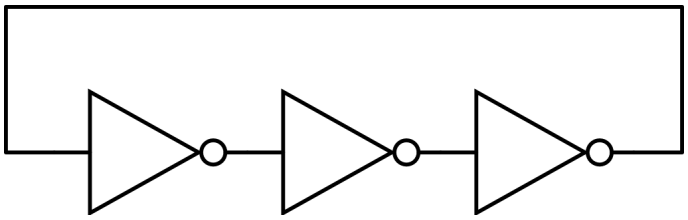


Figure 3.2: A ring oscillator, the simplest unclocked Boolean network, can be made with a chain of an odd number of inverters..

A ring oscillator exhibits regular periodic behavior, but circuits that are only slightly more complicated can yield rich dynamical behavior including chaos. Zhang et al. constructed the circuit in figure 3.3 using just three two-input Boolean logic gates. Node 1 is an XOR gate taking inputs from nodes 2 and 3. Nodes 2 and 3 are XOR and XNOR, respectively, taking one input from node 1's output, and the second input being a feedback connection. The uneven switching times of the logic elements, and the presence of feedback connections, cause thorough topological mixing of the circuit states, resulting in chaotic behavior.

As the transfer function and delay times of the gates are dependent on the power supply voltage, the circuit voltage is a parameter that can be used to vary circuit behavior.

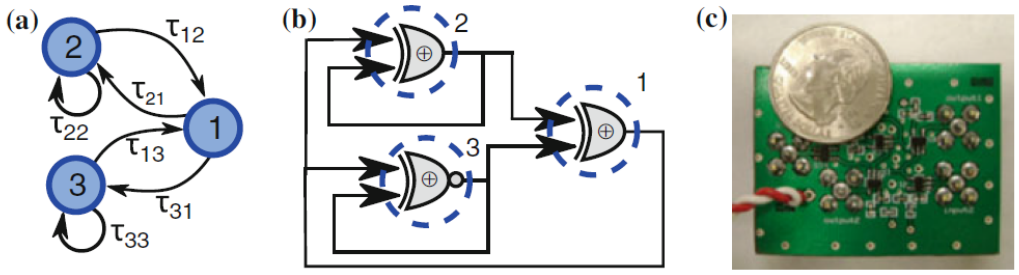


Figure 3.3: 3-element Boolean chaos circuit. (a) state diagram showing transition times τ_{ij} between each node. (b) schematic representation (c) photo of the circuit. Reproduced from Rosin [87].

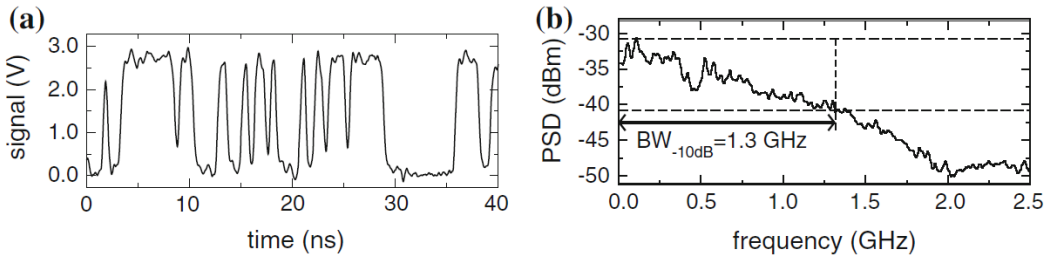


Figure 3.4: Chaotic waveform and broad frequency characteristics that the circuit in the preceding figure produced. Reproduced from Zhang et al. [88].

Unclocked Boolean networks implemented on FPGAs have been used to study chaos [89], oscillator synchronization [90], and chimera states (coexistence of synchronized and desynchronized regions) [91]. They have also been used for ultrafast generation of random numbers [92].

Because the behavior of Boolean networks at the edge of criticality is heavily dependent upon device-specific gains and delays, as well as temperature and hysteresis, circuit simulation often does not yield an accurate result.

3.1.4 Boolean Networks for Reservoir Computing

The rich dynamics of Boolean networks make them an attractive candidate for reservoir computing. Boolean networks stand out from other hardware reservoir implementations because of their potential for seamless integration into existing manufacturing workflows for digital electronics. Whether the reservoir is a block of Verilog that's loaded into an FPGA, a chip placed on a motherboard, or a design directly on silicon, Boolean networks in CMOS fit naturally into today's technology, and have the potential for quick adoption if proven to be successful machine learning accelerators.

3.2 Boolean Sensitivity

Boolean sensitivity describes how excitable a Boolean logic gate is by its inputs. The Boolean sensitivity of a single gate (S_g) is defined as the proportion of 1-bit transitions in the input that result in a change in the output. This may be most easily understood by looking at the transition graphs, such as in figures 3.5, 3.6, and 3.7 for two, three, and four-input gates, respectively.

Taking a look at 2-input gates, for example, the change of inputs from (0,0) to (0,1) results in the transition of an 2-input OR gate's output from 0 to 1, while the transition (0,1) to (1,1) does not result in a change in output state. Two out of four possible 1-bit changes in 2-bit input states result in a change in output state, and therefore the 2-input OR gate's sensitivity is 0.5. Average Boolean sensitivity ($\overline{S}$) is a parameter describing the excitability of a Boolean network, and is the average of the individual gate sensitivities.

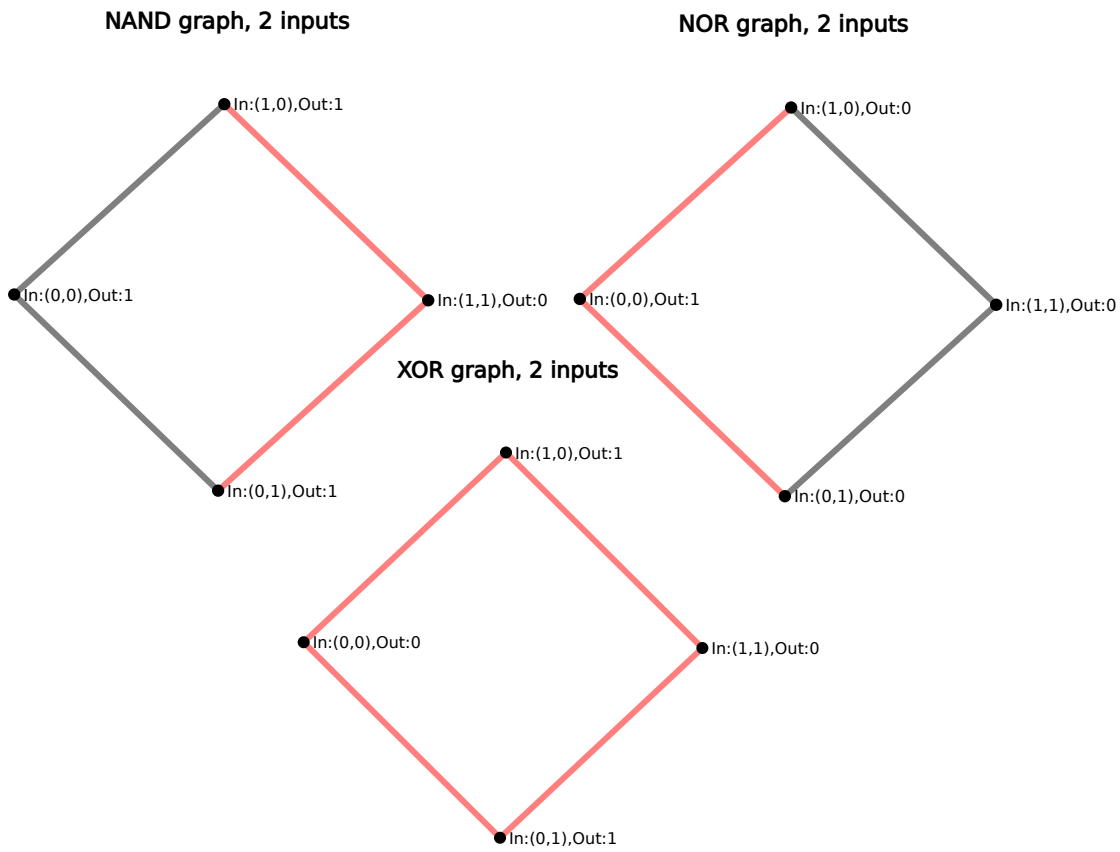


Figure 3.5: Transition graphs for 2-input gates. The sensitivities of these 2-input gates are: $S_{NAND} = 0.5$, $S_{NOR} = 0.5$, $S_{XOR} = 1$.

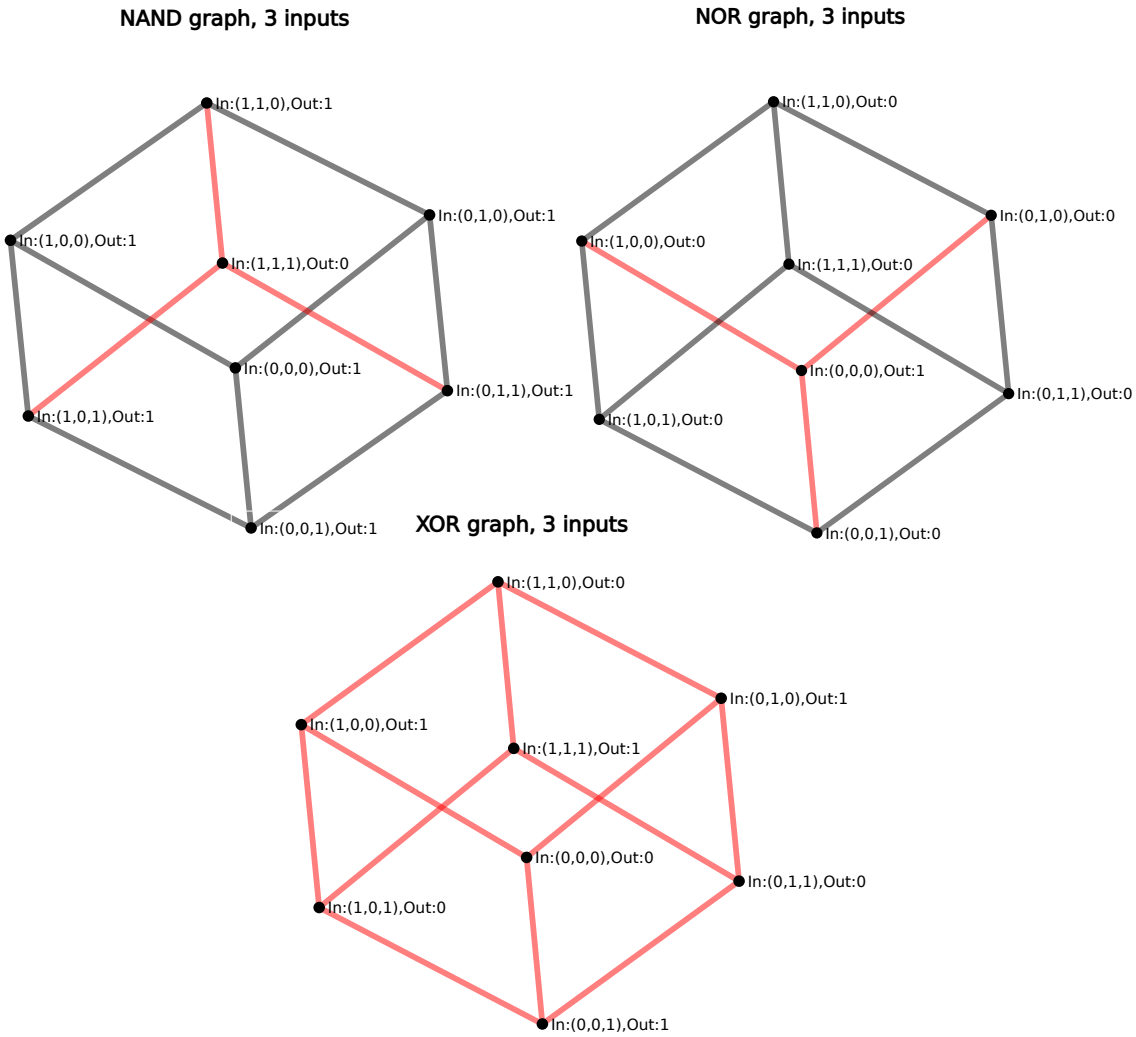


Figure 3.6: Transition graphs for 3-input gates. The sensitivities of these 3-input gates are: $S_{NAND} = 0.25$, $S_{NOR} = 0.25$, $S_{XOR} = 1$.

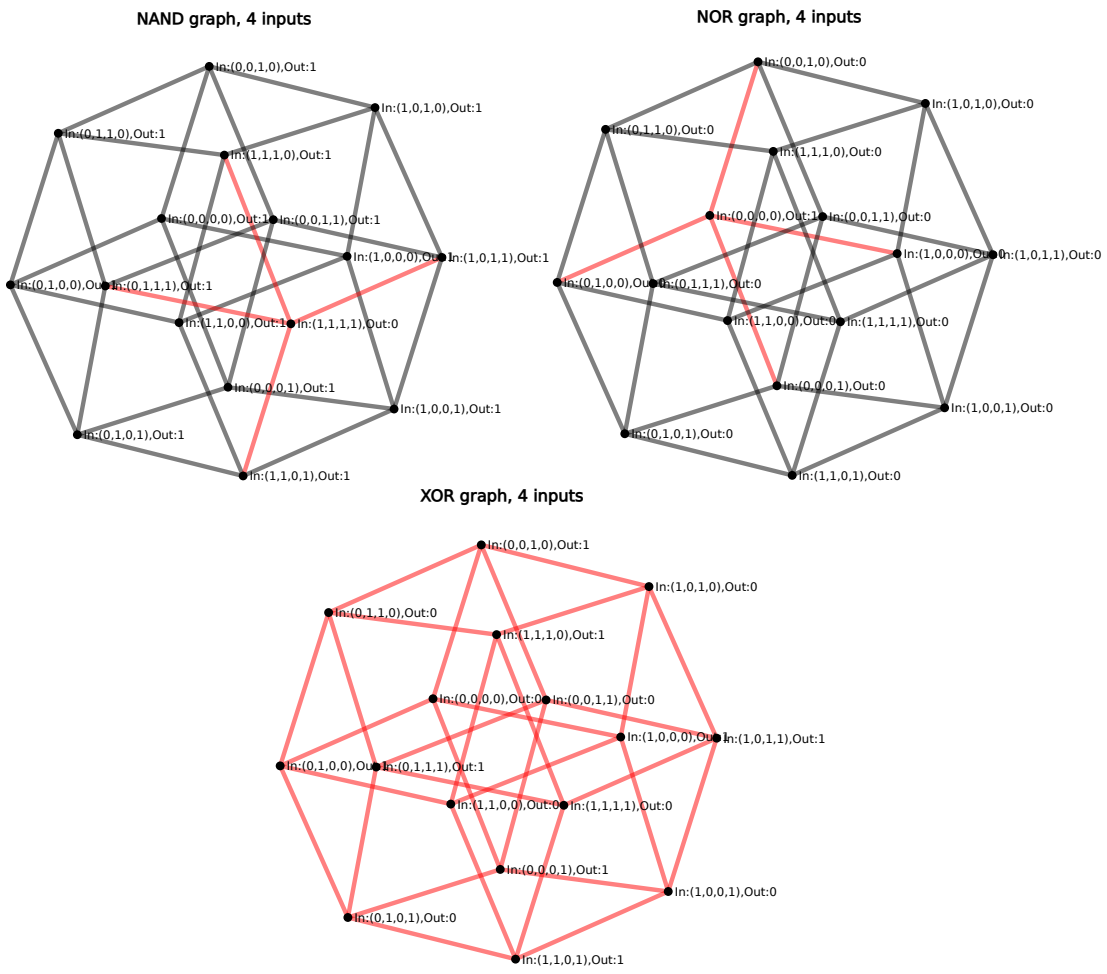


Figure 3.7: Transition graphs for 3-input gates. The sensitivities of these 3-input gates are: $S_{NAND} = 0.125$, $S_{NOR} = 0.125$, $S_{XOR} = 1$.

Chapter 4: Boolean Reservoir Computing on an FPGA

4.1 Introduction

Field-programmable gate arrays (FPGAs) are flexible, reconfigurable platforms in which digital logic circuits can be implemented using a hardware description language (HDL). They are often used for applications in which a device needs to be reconfigured in the field, for production parts in quantities too small to warrant manufacturing a custom chip, or for prototyping. At the core of the FPGA are look-up tables (LUTs), which are configurable truth tables whose logical function can be changed with a setting loaded into an SRAM bank. Between the LUTs are programmable interconnects. A system on a chip (SoC) contains both a microprocessor and programmable logic on the same chip and a multiprocessor SoC (MPSoC) contains more than one processor, to combine the functionality of a dedicated processor and the flexibility of FPGA programmable logic. Figure 4.1 shows a high-level overview of how LUTs and interconnects are used to create a particular reservoir configuration.

In Chapter 2, we described reservoir computing: the idea of using a recurrent neural network with fixed weights and fixed connectivity followed by a simple trained readout layer. In Chapter 3, we introduced the concept of using recur-

rent Boolean logic circuits for reservoir computing. In this chapter, we test both clocked and unclocked recurrent Boolean circuits on an FPGA for machine learning applications[1].

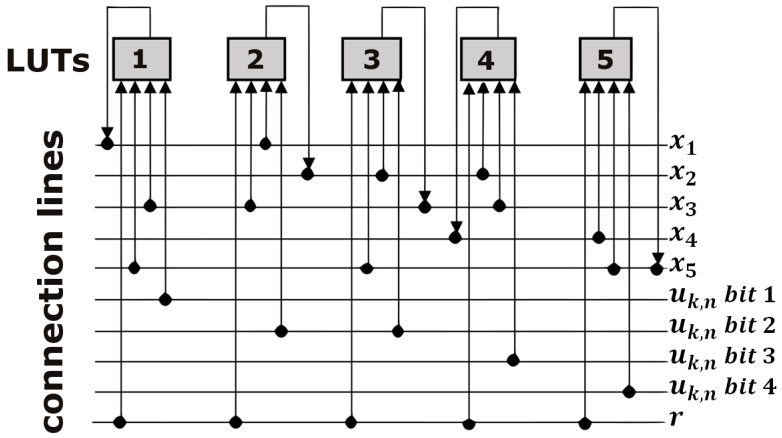
It is worth repeating: *our implementation is not a traditional recurrent neural network*, as one would find in a software reservoir, streamlined in the FPGA's programmable logic. Rather, we are configuring the digital logic gates to function as a *dynamical system exhibiting complex behavior*, and exploiting the dynamics to perform information processing.

4.2 Prior Art

The collective properties of autonomous (unclocked) Boolean networks on a field-programmable gate array (FPGA) were studied and applied to time series classification using machine learning in Shani et al. [22]. We extend that work with a more detailed examination of the parameter space of similar networks that are both unclocked and clocked, and their implementation on faster hardware, the Xilinx Zynq Ultrascale+ MPSoC.

Previous works [22, 80] have both studied autonomous Boolean networks on FPGAs for reservoir computing, demonstrating RF classification and time series prediction, respectively. Haynes et al. [79] studied time-delay reservoirs using a single XOR gate as a node. Other FPGA implementations of reservoirs have focused on efficiently implementing a traditional software-like reservoir in streamlined hardware [75].

[1]Portions of this chapter appear in Komkov et al. [23]



.

Figure 4.1: LUTs and interconnects in the FPGA implementing a reservoir, with internal states x, external inputs u, and global reset signal r, reproduced from Shani et al. [22]

4.3 Experimental Setup

Our experiments were performed on the Xilinx Ultrascale+ MPSoC evaluation board[2] shown in figure 4.2. HDL describing the Boolean network was efficiently written using Python scripts that took a directed graph description and node type list as inputs, and generated Verilog as output. The network was then synthesized in Xilinx Vivado and loaded into the FPGA over an Ethernet connection.

Our application is an unconventional use of FPGAs. Design rules check (DRC) procedures in Vivado throw errors at the detection of loops and logical contradictions, but this is precisely what is needed to create complex network behavior for reservoir computing. Synthesis of a network in Vivado takes about an hour on an average desktop computer and sometimes results in crashes. For this reason, we

[2] `https://www.xilinx.com/products/boards-and-kits/zcu104.html`

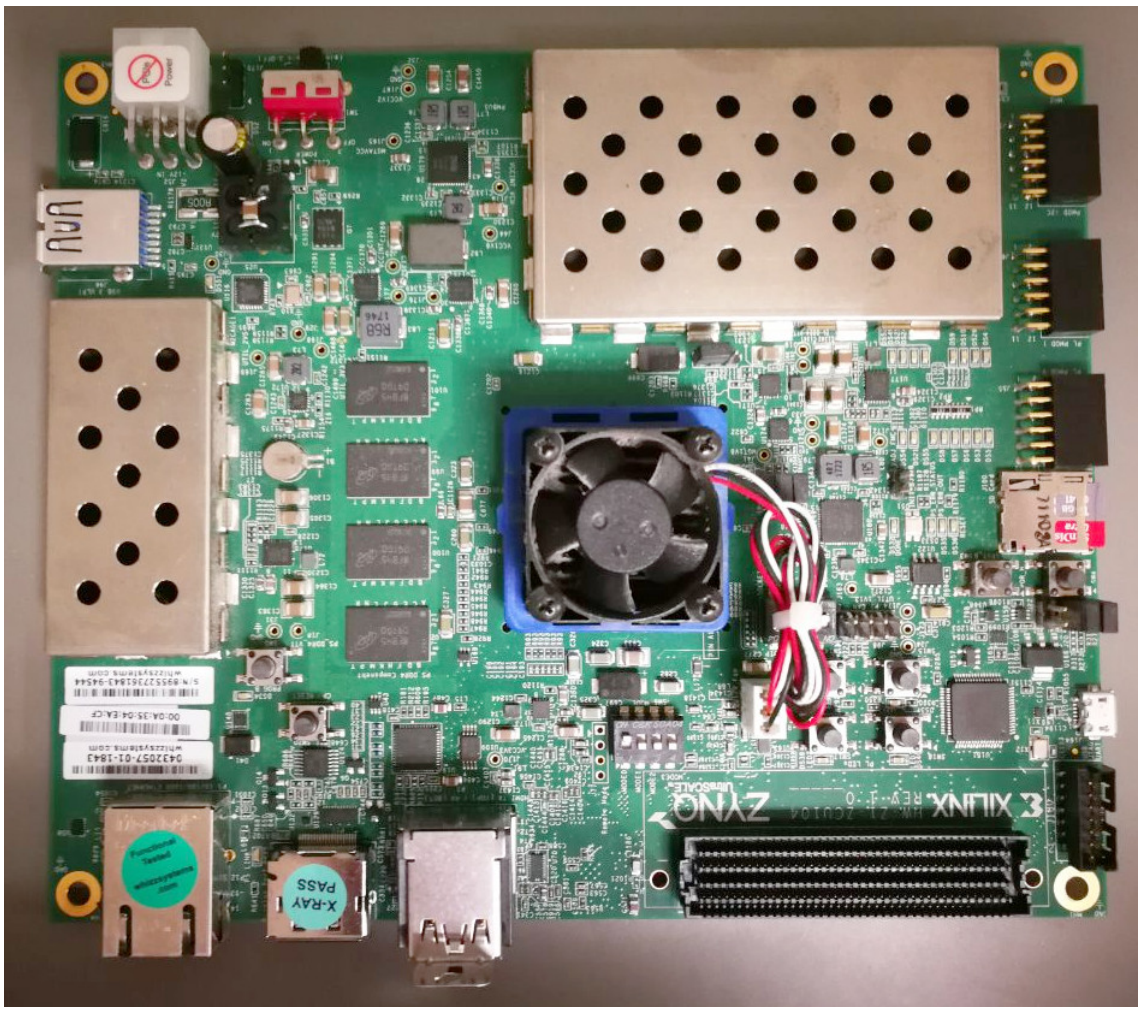


Figure 4.2: Xilinx ZCU104 evaluation board containing the Zynq UltraScale+ MPSoC.

designed networks whose functionality is adjustable by loading in a control setting, rather than reconfiguring the entire logic fabric with new Verilog descriptions for each configuration under test.

A block diagram of our design is shown in figure 4.3, as well as a detailed diagram of the internals of a node. We used a fixed-connectivity network with 4096 nodes in all FPGA experiments.

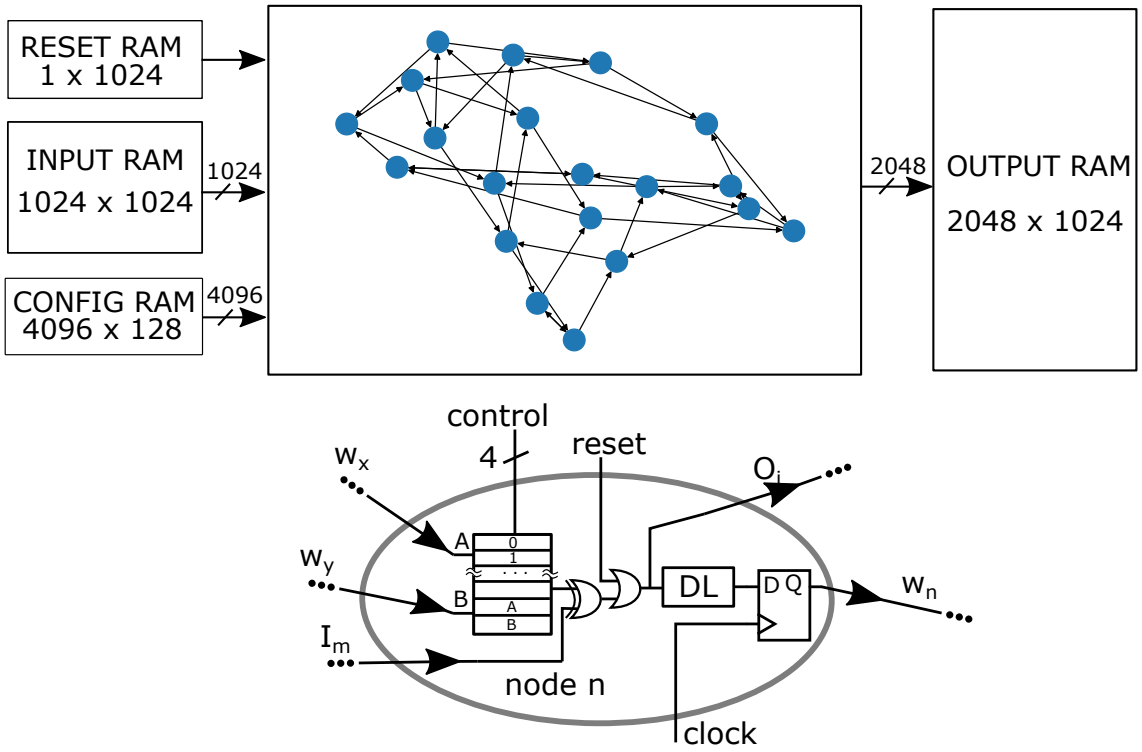


Figure 4.3: FPGA setup and reservoir node detail. Each network contains 4096 nodes, evolves 1024 inputs for 1024 time steps and returns 2048 outputs for the same 1024 time steps. Either the delay line or the flip-flop is used at a time.

4.3.1 Data Flow

In our experiment, reservoir inputs, a reset pattern, and the node control settings are pre-loaded into RAM. Data flow in and out of the FPGA is shown in

figure 4.4. The input RAM is size 1024 × 1024 bits. Each of the 1024 columns of 1024 bits is presented to the reservoir sequentially, in 5 nanosecond steps. Upon every timestep, 2048 outputs, which are arbitrarily chosen out of the 4096 reservoir nodes, are saved to an output RAM, which has size 1024 × 2048. If a time series is shorter than 1024 bits, multiple series are tiled to fit into the input RAM. In the example in figure 4.4, a 128-bit time series was binarized and encoded, and then tiled so that seven series could fit into the input RAM, with a few time steps in between each new series. Between subsequent inputs, the reset signal is applied. The reset RAM contains the reset signal pattern, which is loaded in only once for a particular data series with a consistent pattern.

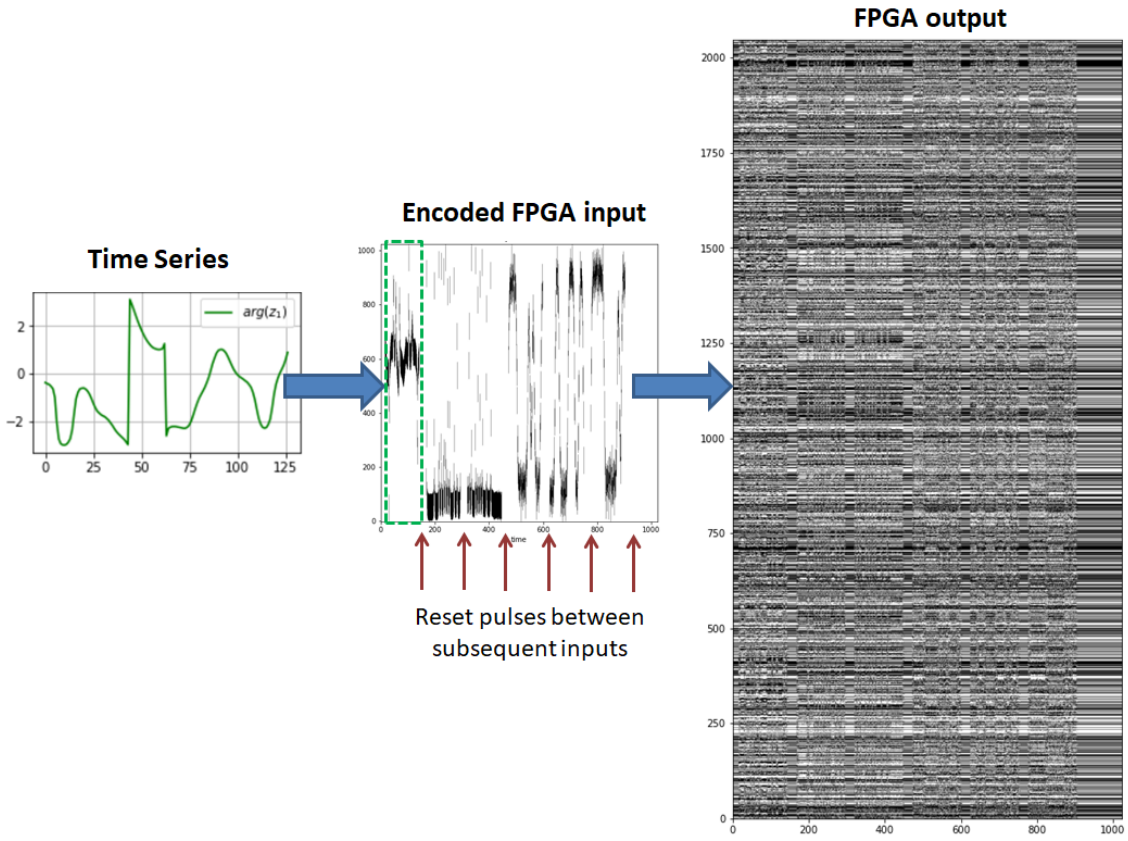


Figure 4.4: Data flow into the FPGA.

The complete output RAM is sent back to the host computer for further processing, where individual data samples are sliced out of the output RAM block. If our system is eventually transitioned to a practical implementation, pre-processing and post-processing functions could be performed on the FPGA itself, as the peripheral resources around the programmable logic are vastly underutilized in our experiment. In fact, The Ultrascale+ MPSoC has two ARM processors in addition to programmable logic, and the evaluation board supports many digital signal processing (DSP) and input/output (I/O) features that we do not use here.

4.3.2 Node Details

As shown in the lower half of figure 4.3, each node of the network receives two output lines W_x and W_y from two other nodes. I_m and O_i are external inputs and outputs present at some nodes. The reset line is connected to all nodes in the network that zeros the node's output, which is done every time before the introduction of new data to ensure a consistent network starting state. The 4-bit control word selects which of 16 logic functions described in table 4.2, is performed by the node on its inputs W_x and W_y. An optional inverter-based delay line (DL in 4.3), where each delay is a pair of inverters, can be added to all nodes to delay their output. The gate transition times on the Ultrascale+ are around 70 picoseconds, faster than the 200 MHz clock can sample. However, the interconnects between LUTs also add delay time.

4.3.3 Network Configuration

While FPGA logic fabric is reconfigurable, the process of synthesizing a Boolean network from Verilog and transferring it to the FPGA takes too long to efficiently tests hundreds of configurations. To speed up the testing of hundreds of networks, the control setting directly on the FPGA stores 128 network configurations that can be selected with a command. These configurations are stored in the config RAM, which has size 4096×128. The 16 different logical functions that a node can be configured to perform are described in table 4.2. The strategy for selecting configurations is described in the next section.

4.4 Adjustable Network Dynamics

We previously discussed the conditions that a physical system needs to meet to be useful for reservoir computing: high dimensionality, separation of inputs, fading memory, and nonlinearity. We also discussed the effect of various hyperparameters on network behavior and showed examples of the evolution of a software reservoir's state with no input. Also, there is no well-established theory for which network configurations perform better than others. For this reason, it is critical to build in ways to adjust the network to find an optimal configuration.

In chapter 3 we examined the definition of Boolean sensitivity (the fraction of 1-bit transitions between inputs that results in a change in output state), which is a way of quantifying the sensitivity of a digital logic gate to its inputs. To be able to examine network behavior as a function of average Boolean sensitivity of the

nodes, we use mixtures of the gate types described in table 4.1. CHOICE refers to a 2-input gate where one of the two inputs is transmitted to the output. Refer to table 4.2 for greater detail on gate types. In this work, we used all two-input gates, and we did not systematically explore the influence of the in-degree of gates.

S_g	Gate Types
1	XOR, XNOR
0.5	AND, OR, CHOICE
0	YES, NO

Table 4.1: 2-Input Gate Sensitivity

4.4.1 Network Strategy

The proportion of gates in the network with sensitivity S_g is denoted r_S. The overall sensitivity of the network is the sensitivity averaged across all of the gates. These two constraints can be written:

$$r_1 + r_{0.5} + r_0 = 1 \tag{4.1}$$

$$r_1 + 0.5\, r_{0.5} = \overline{S} \tag{4.2}$$

Two networks with the same sensitivity can have varying dynamics due to the different composition of gates within them. For example, a sensitivity 0.5 network can have all $S_g = 0.5$ gates or half $S_g = 1$ and half $S_g = 0$ gates. In our study, we choose an equal proportion of each gate type for every sensitivity - for example $S_g = 0.5$ in the configuration with largest r_1 proportion would have $\frac{1}{4}$ XOR, $\frac{1}{4}$ XNOR,

$\frac{1}{4}$ YES gates and $\frac{1}{4}$ NO gates. A symmetrical distribution of gate types is used to mitigate network imbalance to either 1 or 0. In the configuration with smallest r_1 proportion, AND, OR, NAND, NOR, and CHOICE would be in an equal ratio.

To interrogate the space of available configurations, we parameterize the solution to the equations in terms of r_1, sweeping it in 128 even steps within the allowable range for each sensitivity:

$$\overline{S} > r_1 > max(0, 2\overline{S} - 1) \tag{4.3}$$

Each configuration is then given by:

$$(r_1, r_{0.5}, r_0) = (r_1, 2(\overline{S} - r_1), r_1 - 2\overline{S} + 1) \tag{4.4}$$

The variables $\overline{S}$ and the allowable range of r_1 can therefore be varied independently. In all subsequent plots, *config* n refers to the n-th configuration out of 128 that is generated using the routine just described.

As alternative (or in addition to) the use of delay lines, the output state can also be sampled by a flip-flop operating at the sampling frequency. This solution quantizes the propagation time of signals between the nodes to the sampling rate and turns the network in a clocked state machine. We perform tests on networks with nodes whose outputs are sampled by a flip-flop, and networks with nodes having 20 delays and no flip-flop.

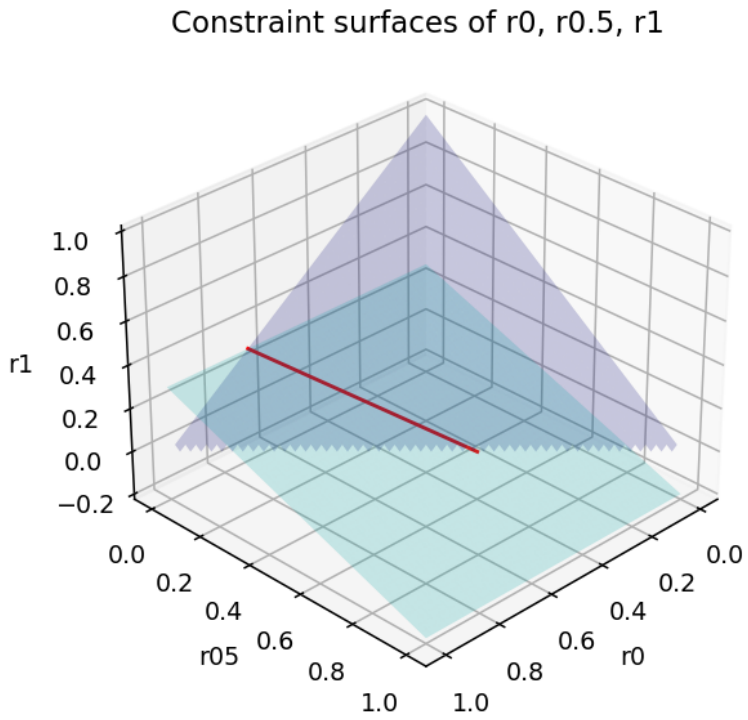


Figure 4.5: Planes of constraints described in equation 4.4. The range of parameters to be probed, at an arbitrary operating point, is the red intersection line. Sweeping S between 0 and 1 moves the relative position of these planes, and traces the red and blue lines in figure 4.6.

4.5 Network Studies

The realizable parameter space of gate configurations is extremely large ($N_{states} \approx \mathcal{O}(16^{4096})$) due to the 4096 different gates within the FPGA-based reservoir that can each have 16 different configurations, described in table 4.2. Even in the case of software reservoirs which have been studied extensively [30], finding reservoir hyper-parameters that work well for particular tasks remains difficult [93]. The extremely large parameter space of the Boolean networks in our experiments necessitates an efficient searching method for potentially useful configurations. For this purpose,

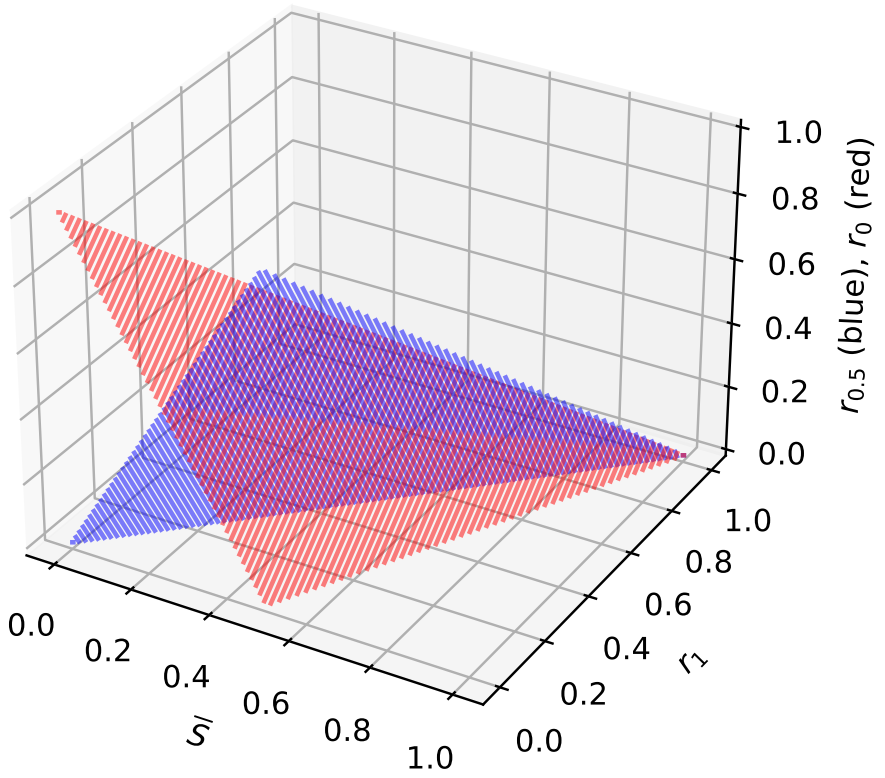


Figure 4.6: Relationship between gate ratios and average sensitivity. For a particular sensitivity $\overline{S}$, the allowable range of r_1 is evenly divided into 128 steps and the other rs are computed. Each of these is an independent configuration labeled *config* in subsequent figures.

we implement tests of network behavior that are dataset-agnostic, measuring self-activity C and transient time T.

4.5.1 Self-Activity

Research stemming from the study of cellular automata has found that the maximum capacity of a network for memory and information processing is achieved in the vicinity of a phase transition—a threshold between regions of highly ordered and highly disordered network behavior [94, 95]. A Boolean reservoir should therefore be near a region of transition between low activity and high activity.

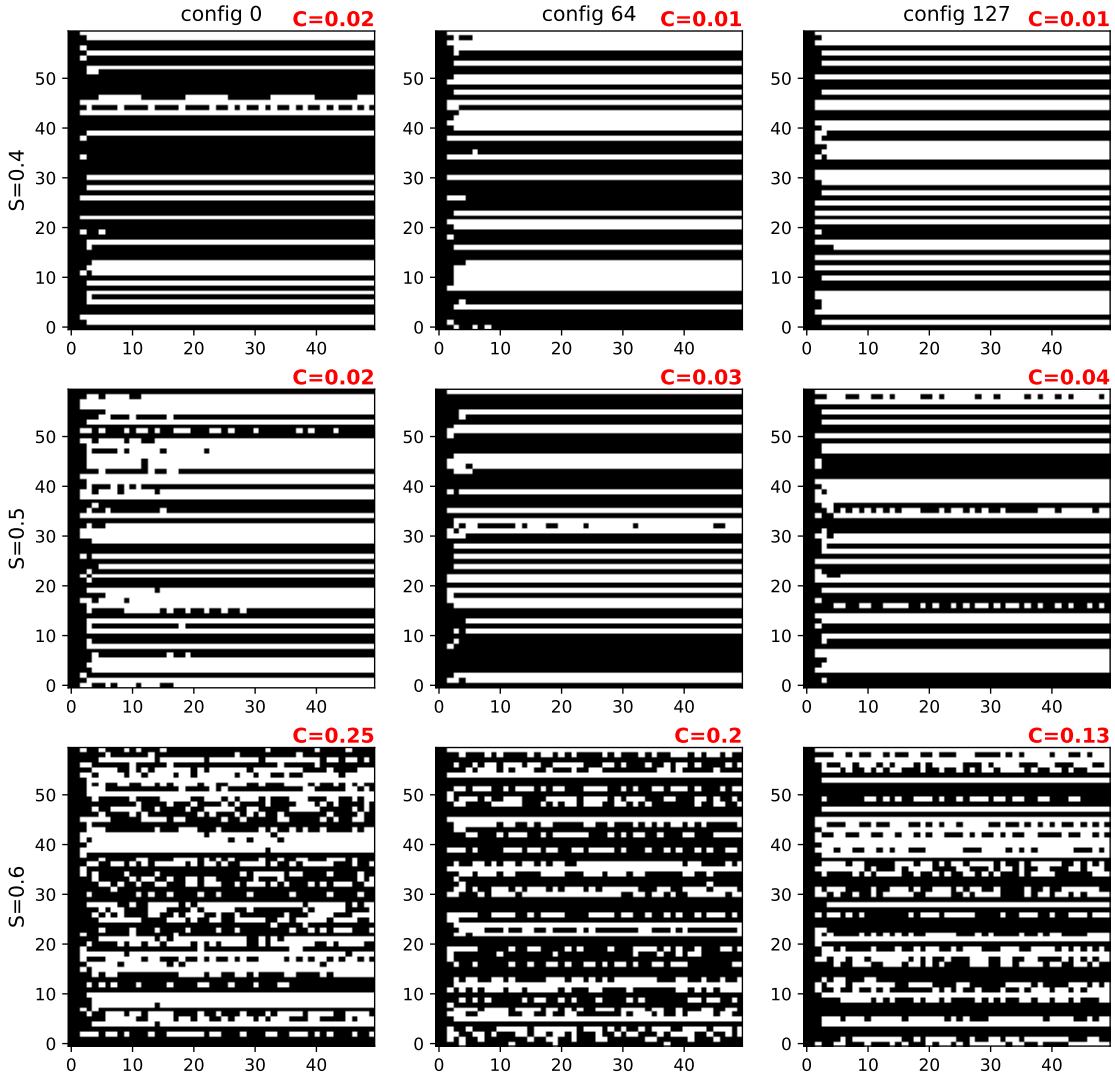


Figure 4.7: Behavior of individual reservoir nodes after a network reset. Activity of 60 of the 2048 output nodes for 50 timesteps at a variety of average network sensitivities and gate configurations, for **unclocked** networks. The calculated self-activity C is shown next to each network snapshot.

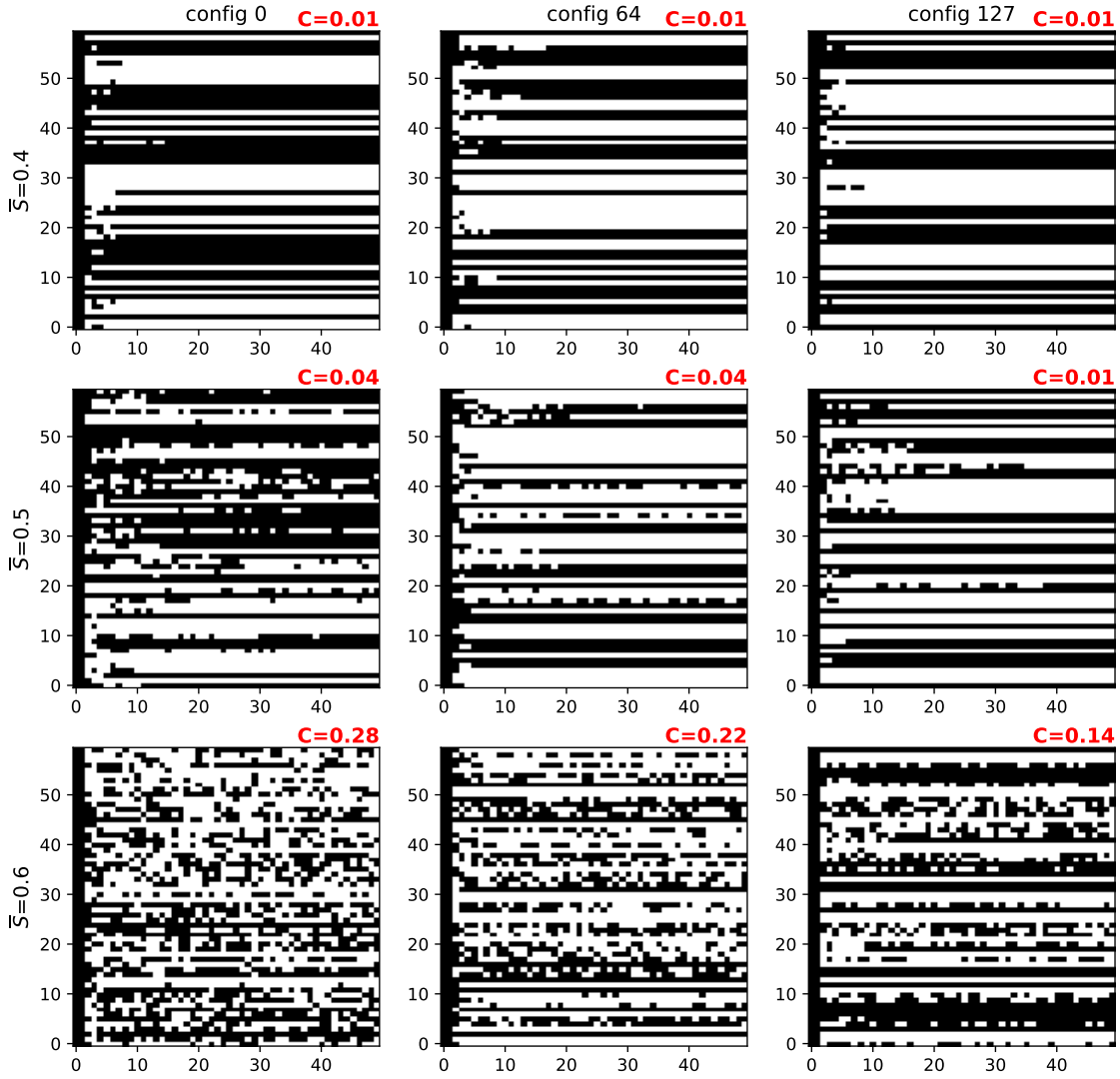


Figure 4.9: Behavior of individual reservoir nodes after a network reset. Activity of 60 of the 2048 output nodes for 50 timesteps at a variety of average network sensitivities and gate configurations, for **clocked** networks. The calculated self-activity C is shown next to each network snapshot.

Gate type	Proportion
True	$r_0/2$
False	$r_0/2$
A XOR B	$r_1/4$
A OR B	$r_{0.5}/10$
A AND B	$r_{0.5}/10$
A XOR $\overline{B}$	$r_0/4$
A OR $\overline{B}$	$r_{0.5}/10$
A AND $\overline{B}$	$r_{0.5}/10$
$\overline{A}$ XOR B	$r_0/4$
$\overline{A}$ OR B	$r_{0.5}/10$
$\overline{A}$ AND B	$r_{0.5}/10$
$\overline{A}$ XOR $\overline{B}$	$r_0/4$
$\overline{A}$ OR $\overline{B}$	$r_{0.5}/10$
$\overline{A}$ AND $\overline{B}$	$r_{0.5}/10$
$CHOOSE_A$	$r_{0.5}/10$
$CHOOSE_B$	$r_{0.5}/10$

Table 4.2: Gate type details

Network self-activity, C, is a way of quantifying overall network activity without any input, after a reset which forces the output of every node to be zero. As in [22], it is defined as the fraction of nodes changing state, averaged over the networks outputs and over timesteps t:

$$C = \langle |x_i(t) - x_i(t-1)| \rangle_{i,t} \tag{4.5}$$

where $x_i(t)$ is the Boolean state at node i at time t.

It is informative to look directly at the network state to understand what various self-activities look like. Figure 4.9 shows the state of 60 out of the 2048 output network nodes for 50 timesteps after a reset across various sensitivities and configurations. Config 0 corresponds to the least amount of r_1 in the allowable range, and config 127 having the most amount of r_1 allowable for a given $\overline{S}$ (see

equation 4.4).

Both clocked and unclocked networks were tested. A variety of behavior appears in both responses. As sensitivity is increased, the overall network activity increases. Clocked networks appear to settle into a fixed state in fewer time steps, in part due to the fast transition times of the gates and the under-sampling of the activity due to the limited clock speed within the FPGA. In both cases, some networks exhibit only transient behavior and settle into a static fixed state. Some networks settle into limit cycles. Networks with higher excitation do not appear to settle into any repeatable pattern.

A sweep across sensitivities from $\overline{S}$=0.3 to $\overline{S}$=0.7 and spanning all gate configurations is shown in figure 4.10. This figure shows results only for a clocked network, but unclocked network results look very similar. However, unclocked networks caused the FPGA to crash at around $\overline{S}$=0.68 due to high activity, which terminated the scan. Furthermore, the sweep was repeated using multiple random seeds, which determine the placement of gates, and similar results were observed.

As is evident from 4.10, the self-activity (C) dependence on average sensitivity ($\overline{S}$), shows significant structure. As we have kept the node graph topology constant in these tests, the features likely indicate the presence of dynamical loops in the directed graph. Evident in that figure are tails below $\overline{S} = 0.5$ that are likely limit cycles (ring oscillators) in the setup that persist among the gradually varying configurations. Above $\overline{S} = 0.5$, the cone-shaped structures are suggestive of a joining of various oscillators present in the FPGA structure that are cooperating in more complicated dynamics. Higher activity is seen in the lower-numbered configurations,

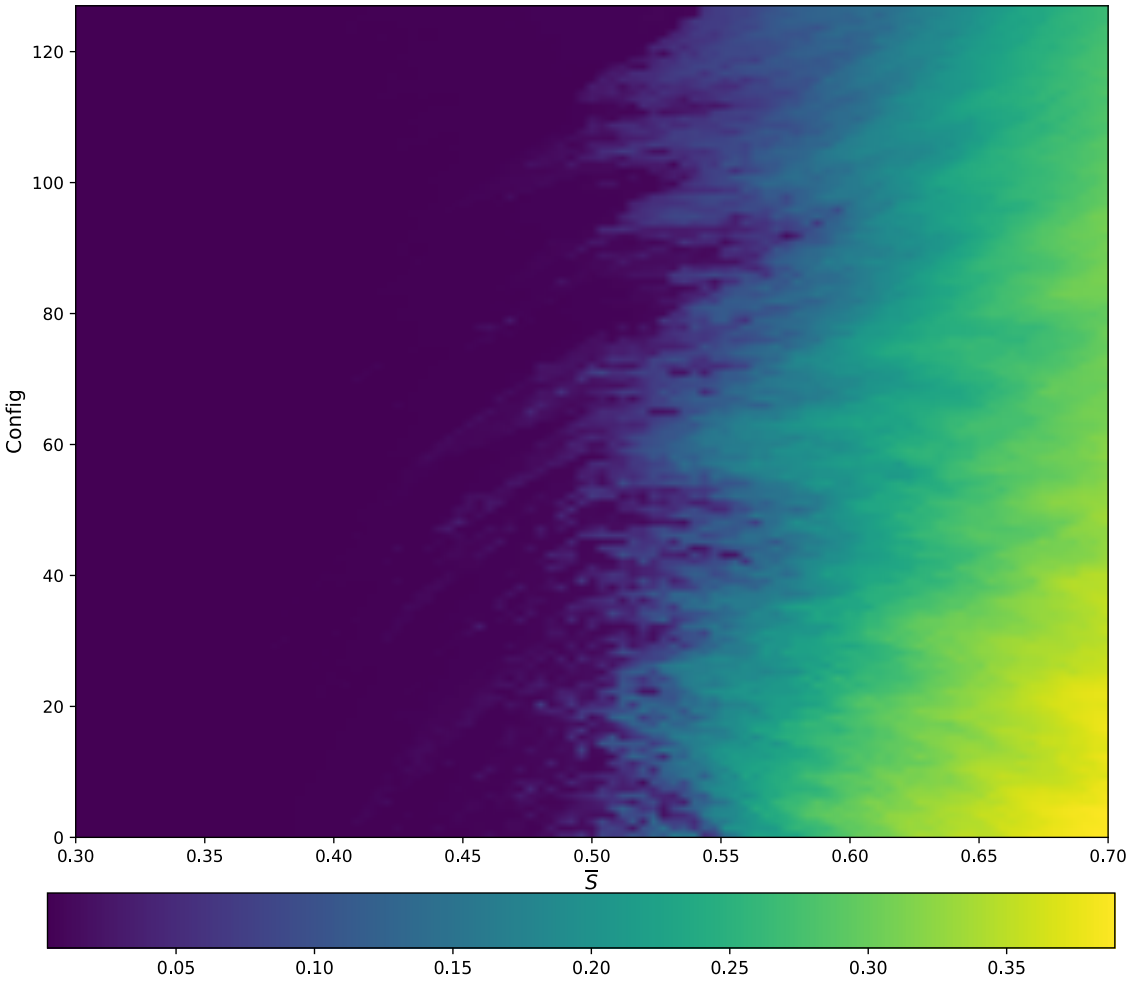


Figure 4.10: Self activity C as a function of $\overline{S}$ and configuration in a **clocked** network. Structures in the image indicate the presence of interacting dynamical loops in the reservoir's directed graph.

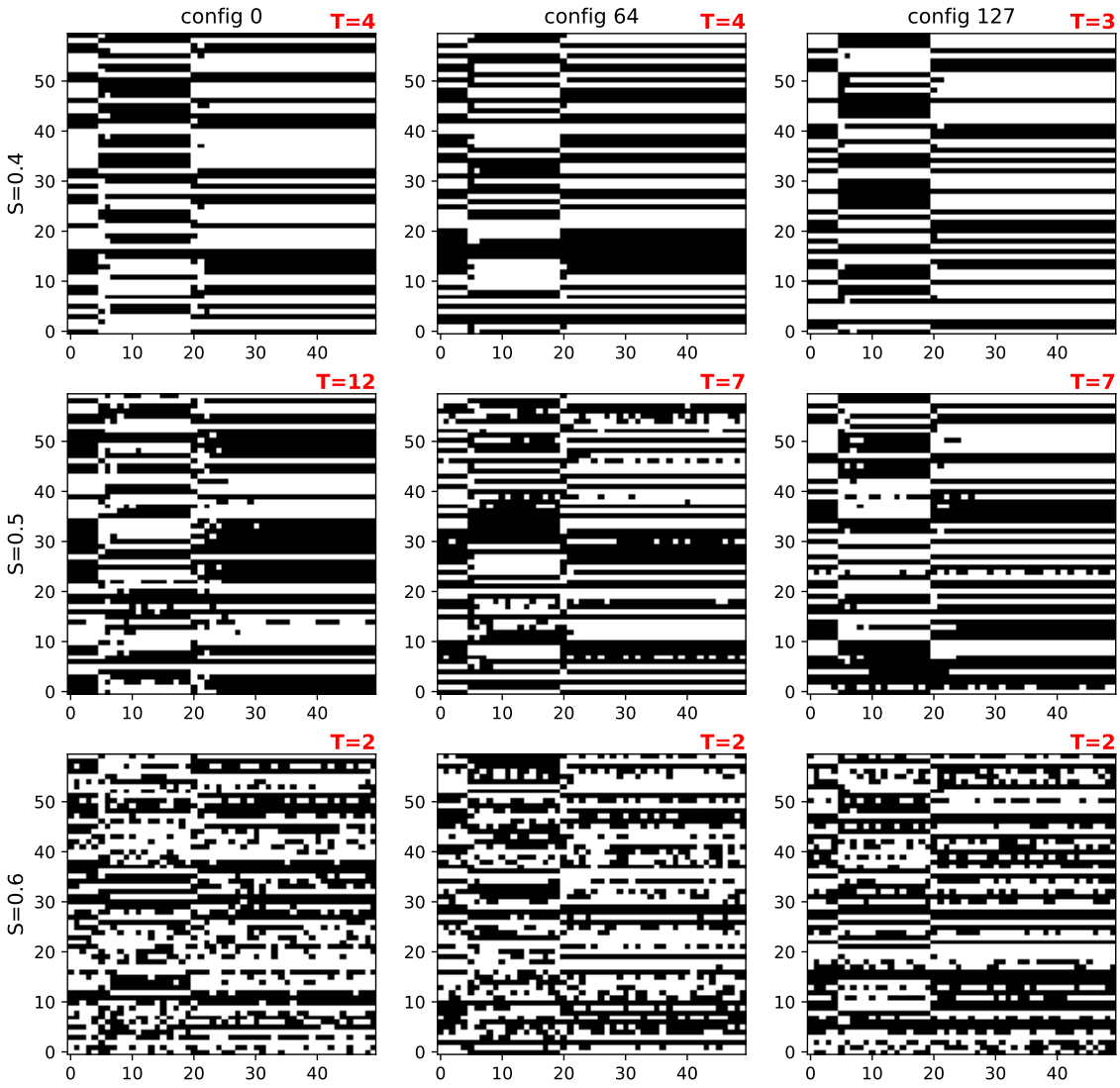


Figure 4.11: Unclocked network

which have a smaller relative fraction of r_1 gates.

4.5.2 Transient Time

An important property of a reservoir is fading memory—the ability to retain past states while weighting the recent ones the most highly. To provide insight into a network's memory, we perturb the network with an input of all ones for five timesteps, and observe the network state afterwards. Activity after a perturbation

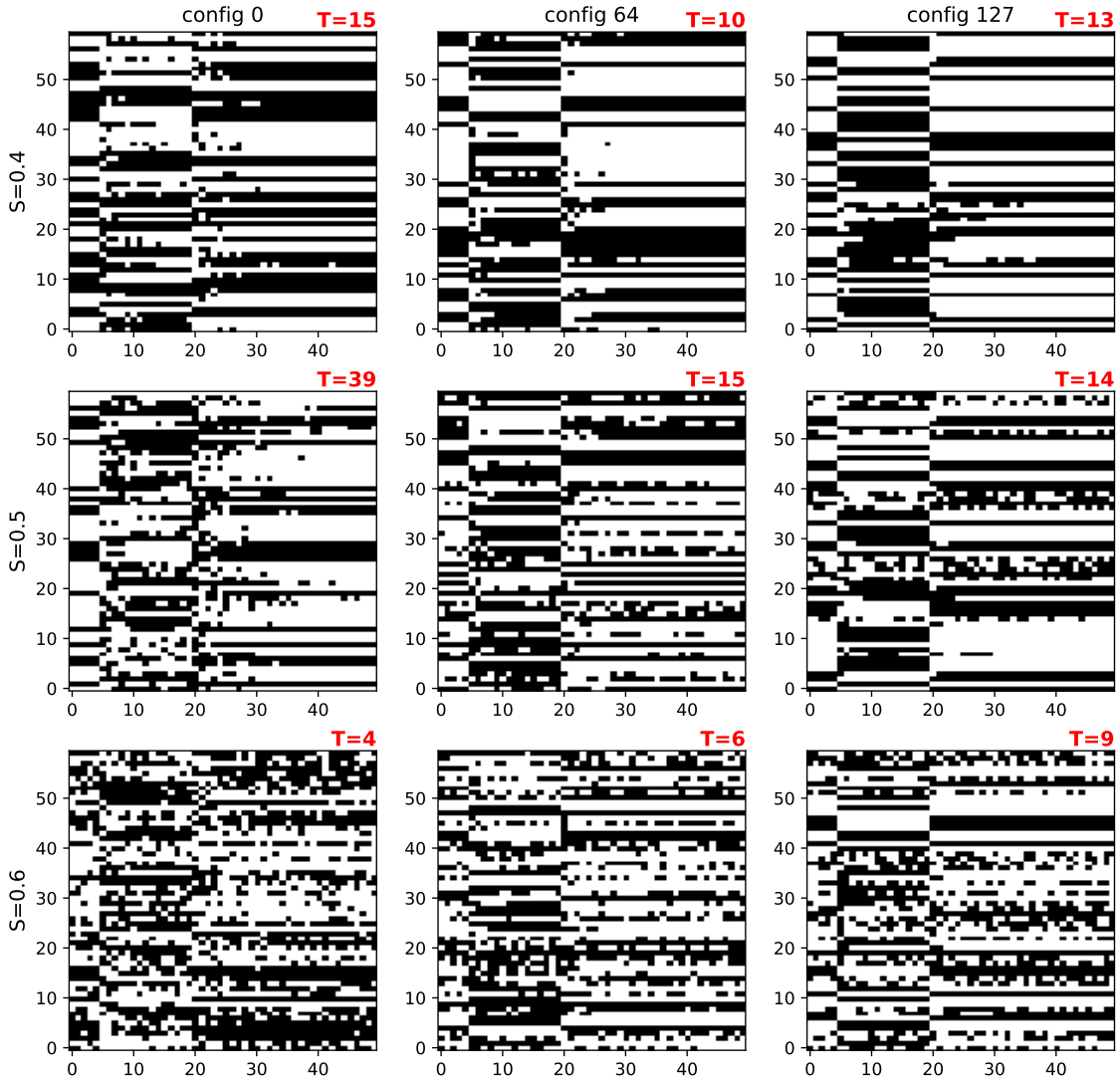


Figure 4.13: **Response of individual reservoir nodes to a perturbation.** An input of all ones, 1024 inputs nodes, is introduced between the timesteps of 5 and 20. The corresponding transient time T is annotated in red.

that decays with time is called *transient activity*. To quantify the duration of activity after a perturbation, we calculate an average network value $\overline{X}$ for 30 timesteps before an input is presented:

$$\overline{X} = \langle X \rangle_{i,t\in(0,30)}. \tag{4.6}$$

We then calculate the departure away from this calculated average value:

$$d_t = \langle X \rangle_t - \overline{X}. \tag{4.7}$$

We define the transient time T as the time it takes for the network to return to a value close to its pre-perturbation average state: $\overline{X}$+0.01. As can be seen from the network images in 4.9 at higher sensitivities, the concept of a network's average value becomes nebulous with networks having a large amount of self-activity, as there's no steady state. In fact, some of these highly excited networks never return to $\overline{X}$+0.01, in which case they are discarded from consideration for machine learning and appear as white pixels in 4.16.

Figure 4.13 shows a scan across the average sensitivities and configurations for clocked and unclocked networks. The unclocked networks crash the FPGA around $\overline{S} = 0.68$, and as a result the values are all zero above this $\overline{S}$ in the plot. Here is a large distinction between clocked and unclocked network versions. This is in large part due to the effective sampling frequency relative to the network dynamics. The unclocked networks have each node's activity slowed down by the delay line, but are

still under-sampled by the 200MHz clock. Clocked networks only advance upon a clock pulse, and every state of the network is measured.

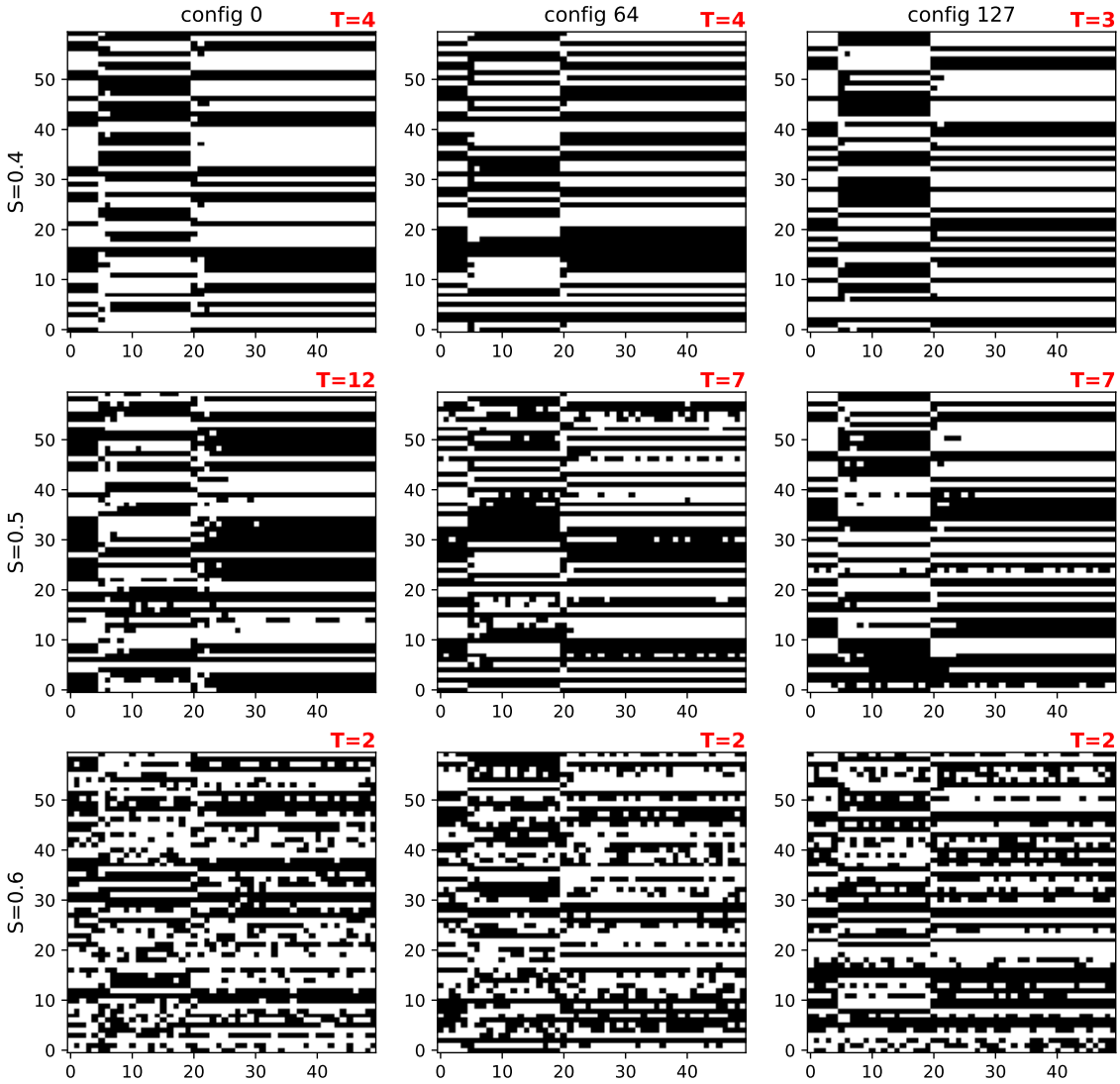


Figure 4.14: Response of **unclocked networks** to a perturbation. Corresponding transient time T is annotated in red.

4.5.3 Selection of Desirable Networks

A network suitable for machine learning will satisfy the following properties:

- **Quiescent with no input.** Activity within the network when there is no

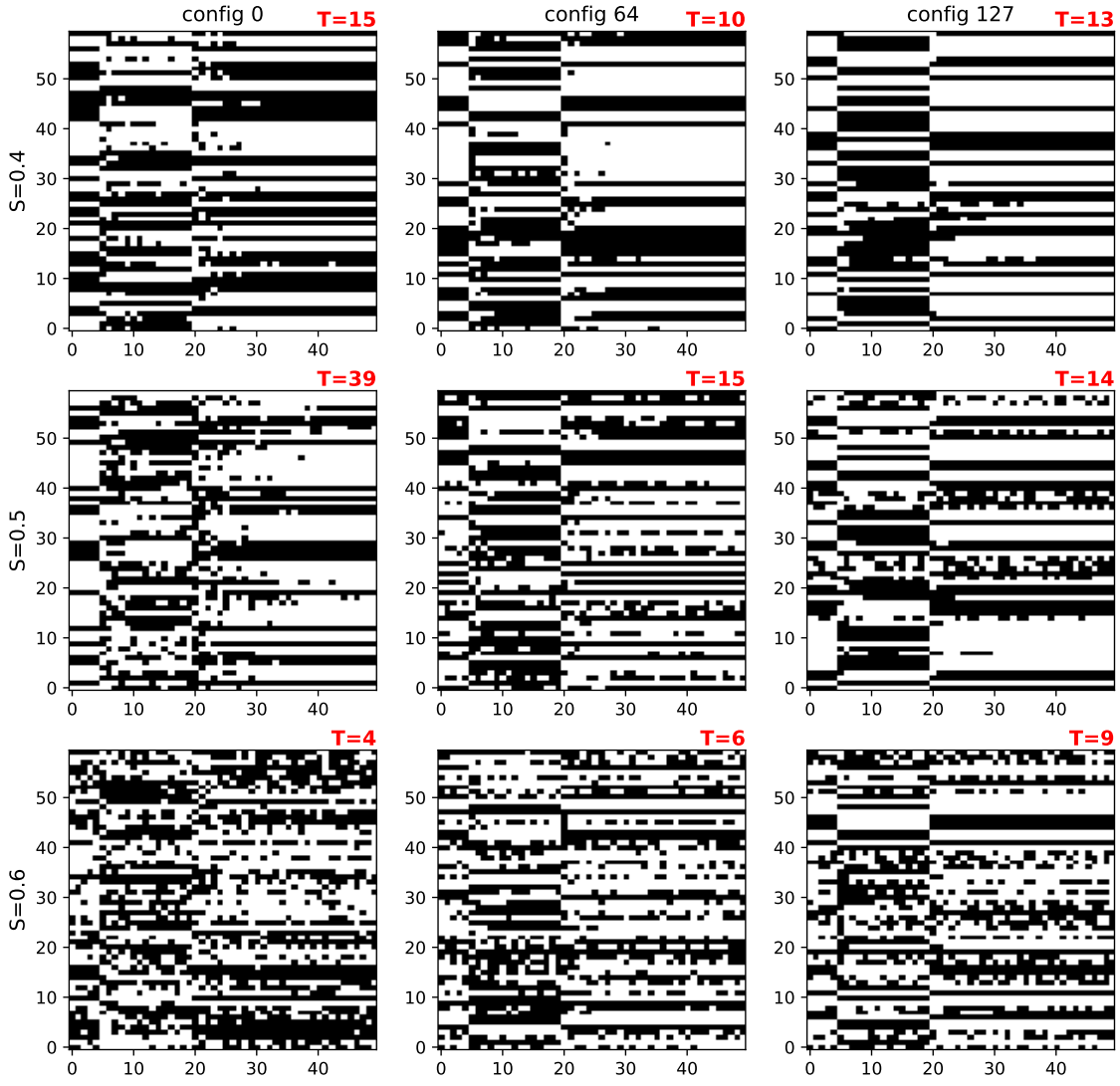


Figure 4.15: Response of **clocked networks** to a perturbation. Corresponding transient time T is annotated in red.

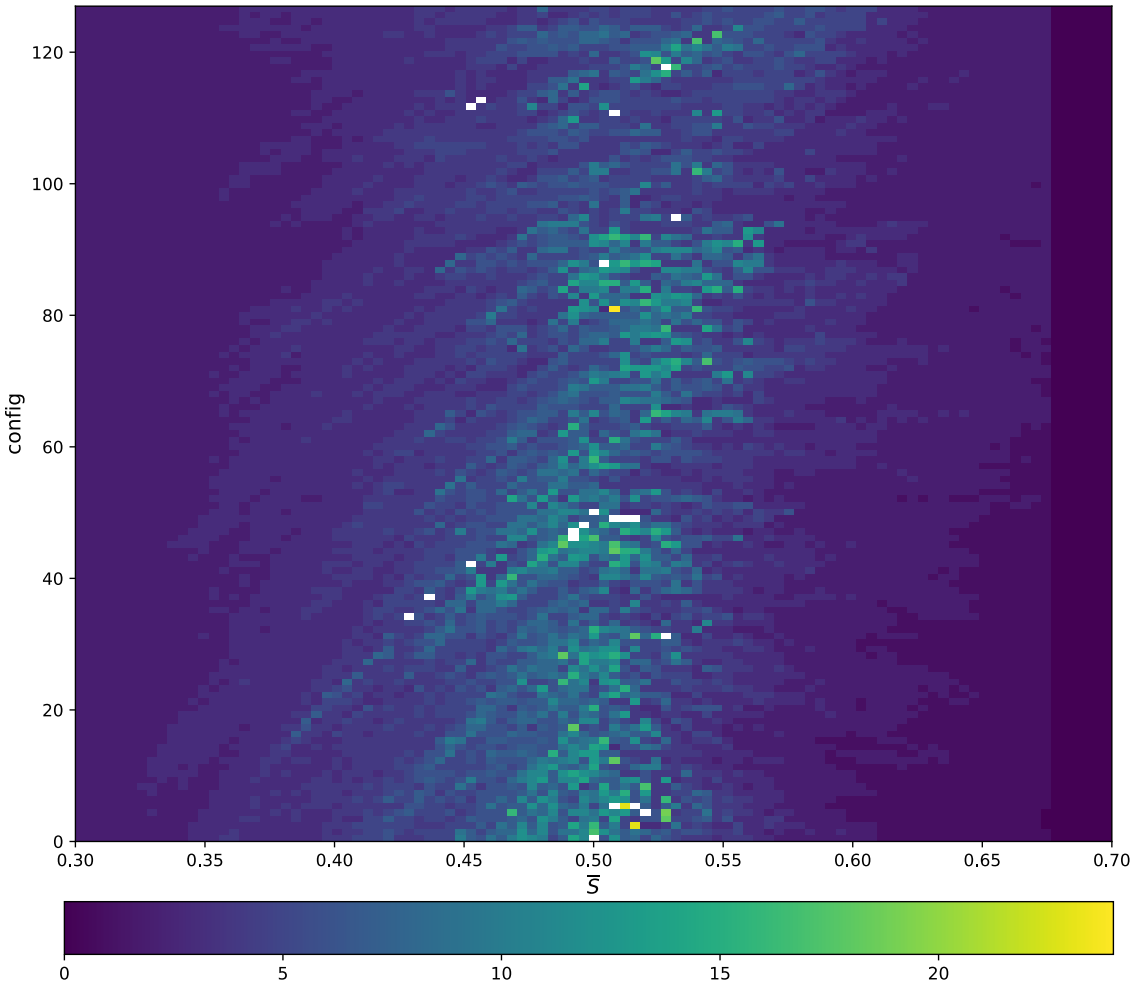


Figure 4.16: Transient times of **unclocked** networks.

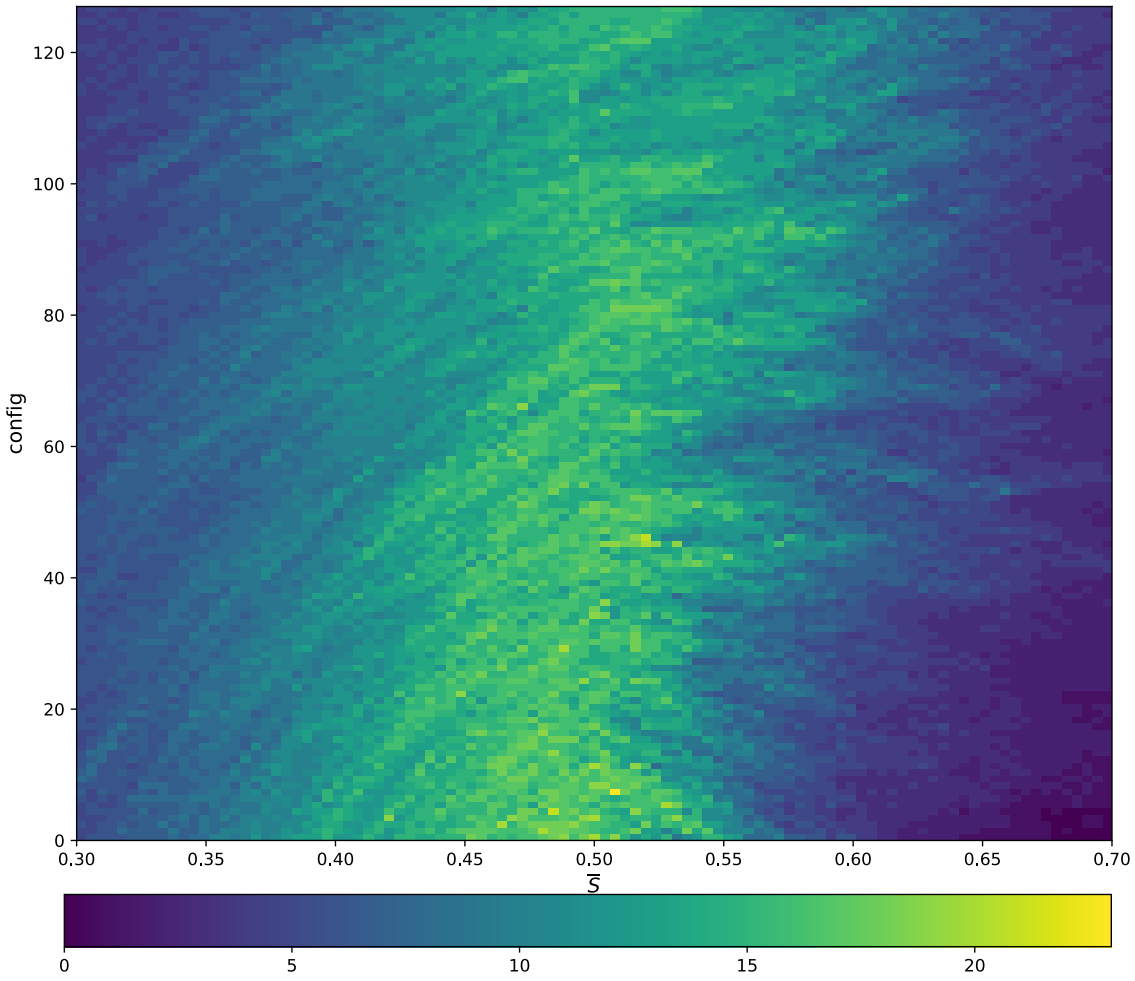


Figure 4.17: Transient times of **clocked** networks.

input consumes power but does not do useful computation. To satisfy this requirement we choose only networks with self-activity $C < 0.0175$, a number chosen qualitatively. The self-activities of only unclocked networks fitting this criterion are shown in figure 4.5.3.

- **Close to a phase transition.** Networks should be close to $\overline{S}$=0.5 at which there is an experimentally observed inflection point in network activity, as can be seen in 4.10. Networks near the "edge of chaos" have been shown to outperform networks in other regimes [95, 96, 97].

- **Fading memory.** For this we select transient times greater than 3 timesteps.

The transient times of only those networks meeting all criteria are shown in figure 4.5.3. The resulting configurations in the parameter space of $\overline{S}$ and configuration number is a band below $\overline{S}$=0.5 containing between 1000 and 3000 configurations, depending upon the network type and random seed. Of these, 100 were randomly chosen for application to a machine learning classification test.

4.5.4 Output Distributions and Entropy

Looking at the frequency at which reservoir nodes are on or off, given a uniform input, may provide insight into the networks' expressive power. Distributions with the greatest entropy may have the greatest expressive power, as entropy is a measure of information content. Due to the presence of $S_g = 0$ gates in many configurations, as well as the random connectivity within the networks, not all of the reservoir outputs contribute usefully to the result. The $S_g = 0$ gates have fixed high or low

outputs. Other gates may get two inputs from fixed-output nodes, thereby shutting off that gate. Other gates simply rarely change their state due to the particular network configuration. These scenarios are not checked before network synthesis.

100 random uniformly distributed input vectors were introduced to the network, with each bit in the input having equal probability of being on or off: $P(X_i = 1) = P(X_i = 0) = 0.5$. The average value of each output bit was then computed: $P(X_i) = \sum_{k=0}^{100} X_{i,k}$. If the average value was below 0.1 or greater than 0.9, meaning that it changed less than 10% of the time, the bit was discarded, as large tails at 0 and 1 skewed the distributions.

Histogram of these distributions across various average sensitivites and configurations is shown in figure 4.5.4. From each of these histograms, the Shannon entropy was computed:

$$H(X) = -\sum_{i=0}^{1024} P(X_i) log_2(X_i) \tag{4.8}$$

Each configuration had a different number of output gates after being trimmed. Configurations with fewer r_1 gates have broader output distributions, and more nodes remaining after the trimming procedure, than distributions with more r_1 gates.

From left to right in figure 4.5.4, the proportion of $S_g = 0$ YES and NO gates increases, which are gates that shut off the output from a node completely. It can be seen that after pruning inactive outputs, there are fewer nodes left in the networks with higher r_0 proportion. As for the remaining distribution, the networks with

higher r_0 proportion have a sharper peak around $P = 0.5$, while networks on the left-hand side, with more varied gate types, have a broader distribution. How to use these distributions to predict information processing utility of a Boolean network remains an area of research.

4.6 Radio Frequency (RF) Classification

To benchmark the machine learning performance of various network configurations, the FPGA-based reservoir was used for classification of the DeepSig 2016 dataset [98]. This dataset contains 1000 example waveforms per signal-to-noise ratio (SNR) for each of 11 various radiofrequency modulation types across SNRs from $-20 : 2 : 18$. As the overall amount of data is very large after processing by the reservoir, we use only SNR 18 to efficiently test many configurations. 65% of the samples were used for training, 15% for validation, and 20% for testing.

4.6.1 Preprocessing

Each sample in the dataset contains 128 timesteps of in-phase (I) and quadrature (Q) waveforms. Recall that the raw (I) and (Q) data may be modeled as a complex number $z = I + j\,Q = re^{j\theta}$ where j is $\sqrt{-1}$. Eight distinct different data from these raw (I) and (Q) waveform have been computed, see 4.21:

- raw I
- raw Q
- Modulus of signal: $|z|$

- Modulus difference ratio: $z_r = |z_{i+1} z_i|$
- Modulus forward difference: $\Delta z = |z_{i+1}| - |z_i|$
- Phase of signal: θ_i ($\theta_i = arg(z)$)
- Phase forward difference: $\Delta\theta = arg\,(z_{i+1} z_i)$
- Phase difference ratio: $\theta_r = arg(z_{i+1}) arg(z_i)$

Each of these waveforms was then thermometrically encoded. Each signal was mapped to $[i{\cdot}128, (i{+}1)128]$ bit inputs for $i \in (0, 7)$ for the 8 different encodings. The amplitude encodings, the first five, were scaled by sample minimum and maximum so that [sample minimum, sample maximum] mapped to $[i \cdot 128, (i+1)128]$. The phase encodings were mapped from $[-\pi, \pi]$. This encoding is necessary for the reservoir as it accepts 1024 binary inputs. The inputs are then "tiled" so that they fit into the 1024-bit input RAM, with 10-timestep pauses between them, and a network reset right before the introduction of a new input.

4.6.2 Subsampling in Time

To improve regression speed, we sub-sample the outputs into 16 samples, spaced apart by 8 timesteps. This was tested to only reduce machine learning accuracy by a few percent while greatly reducing the number of trainable parameters and improving regression speed.

4.6.3 Machine Learning Training Procedure

While a typical attraction of reservoir computing is rapidly evaluating the output with a closed-form solution of linear regression, logistic regression is more suitable for a classification problem which maps the data to multiple discrete classes. While for the purposes of this experiment, the reservoir state is transmitted to a supervisory computer for post-processing and training, these functions can be implemented directly on the FPGA for high speed in the future.

The size of the DeepSig 2016 dataset and the amount of data generated by the FPGA-based reservoir necessitates an iterative approach to training. In pytorch, we use a single neural network with linear neurons, and use stochastic gradient descent to minimize cross-entropy loss. To optimize the learning rate and L_2 penalty, Hyperopt was used with trials scheduled and early stopping implemented using the asynchronous hyperband optimizer in Ray Tune [99]. Each input into the regressor was allowed 20 Bayesian optimization trials optimizing learning rate within the range (1e-5, 1e-1) and L_2 penalty within the range (1e-9,1e-1), and could train up to 50 epochs. Convergence was observed with nearly all trials. While Bayesian optimization is not necessary for only two parameters to optimize, we developed this infrastructure using Ray Tune to wrap reservoir hyperparameters into the optimization loop in the future. Additionally, Ray Tune can distribute a task among multiple CPUs or GPUs for faster processing. All reservoir configurations were evaluated with the validation set. Test set results were consistently a few percent lower.

Variation	Preprocessing	Trainable Parameters	Accuracy
0	Eight time series variations	11,176	14%
1	I and Q series	2,816	22%
2	(1) thermometrically encoded	1,441,792	44%
3	(5) sub-sampled to 16 timesteps	180,224	66%
3	Convolutional neural network [98]	2,214,475	76%
4	(5) passed through best **clocked** RPU configuration and sub-sampled	259,952	76%
5	(0) thermometrically encoded	1,430,528	78%
6	(5) passed through best **unclocked** RPU configuration and sub-sampled	161,744	82%

Table 4.3: **DeepSig 2016 SNR 18 time series classification results using logistic regression, with various preprocessing methods.**

4.6.4 Machine Learning Results

Accuracy results of logistic regression applied to raw data, preprocessed data, encoded data, and Boolean reservoir-processed data are shown in 4.3. While preprocessing by itself degrades classification accuracy, after thermometric encoding it provides a surprising boost to the results. The encoded time series then processed by the reservoir show a further improvement in accuracy: even when sub-sampled to 16 timesteps, the accuracy is competitive with that of the state-of-the-art convolutional neural network with less than one-tenth of the trainable parameters. The corresponding confusion matrix applied to the test set on variation (6) from 4.3 is in 4.25.

Machine learning accuracy on unclocked network-processed data on the same axes as the prior figures of $\overline{S}$ and $config$ is shown in 4.25. We found no discernible dependence of accuracy on C or T within the region selected by the criteria described

in a prior section. A similar scatter plot was generated for clocked networks and overall accuracy was a few percent lower.

4.7 Conclusion

We explored the dynamics of Boolean networks for reservoir computing, narrowing down the extremely large parameter space of configurations using dataset-agnostic methods to identify networks that would function well as reservoirs for reservoir computing. We validated these chosen networks on the DeepSig 2016 dataset, achieving state-of-the-art accuracy with less than one-tenth of the trainable parameters with an unclocked network on a Xilinx Ultrascale+ MPSoC. We also tested clocked networks which yielded slightly lower accuracy.

A notable observation is that pre-processing the RF time series by computing ratios and first differences of the complex modulus and angle of the I and Q signals, and thermometrically encoding this data, yields comparable accuracy with simple logistic regression to a convolutional neural network. Even with taking only one-eighth of the thermometrically encoded data by slicing in time to reduce the number of trainable parameters, the accuracy remains high.

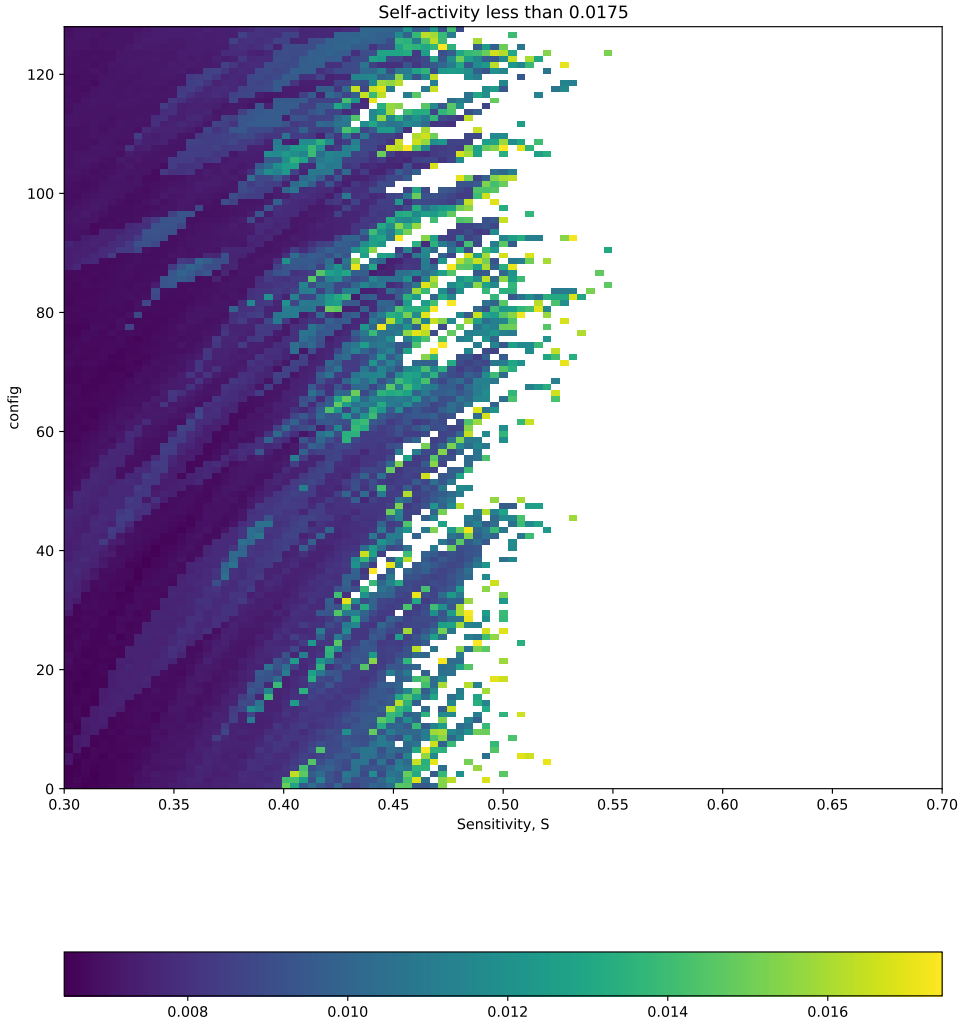


Figure 4.18: Selected configurations based on self-activity C, of unclocked networks.

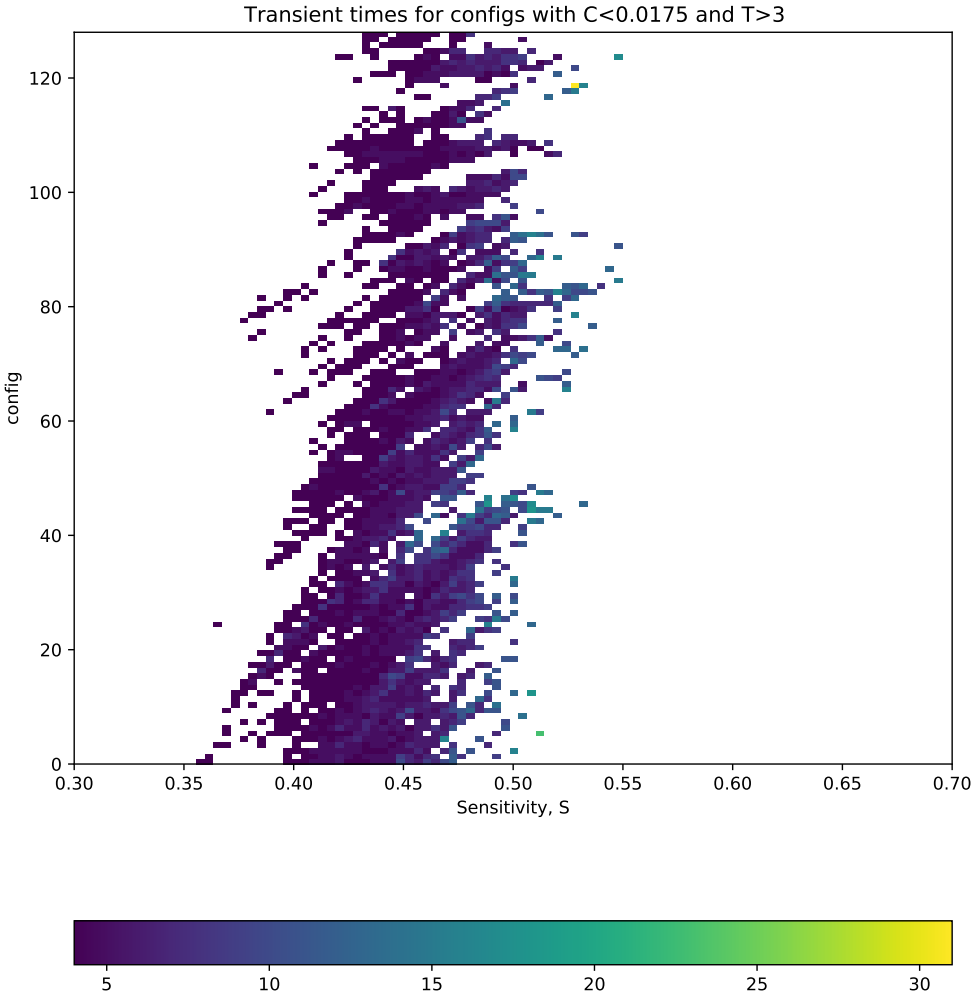


Figure 4.19: Selected configurations based on self-activity C and transient time T, of unclocked networks.

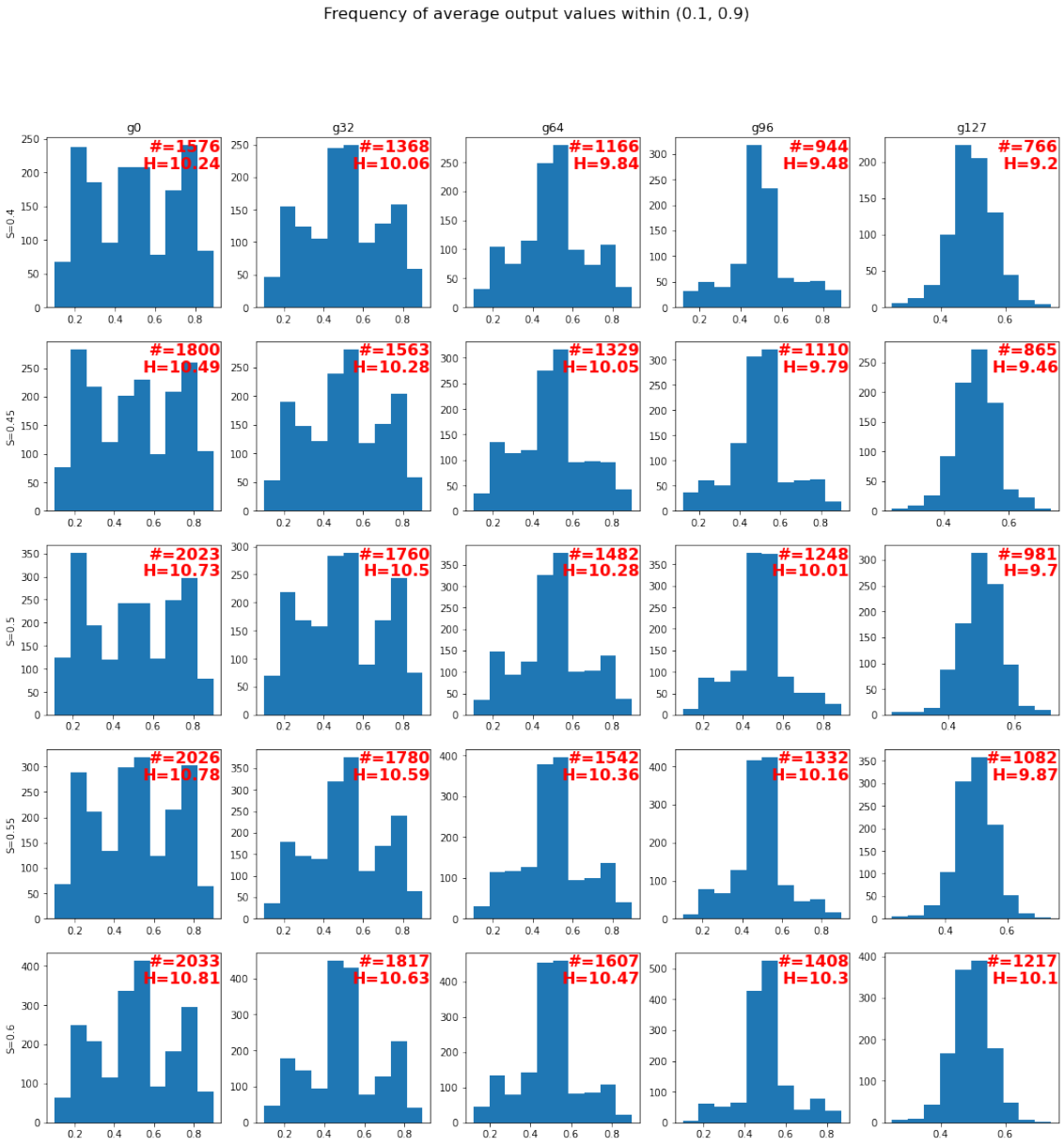


Figure 4.20: Output frequency distributions with a uniform random input. Nodes that are on or off more than 90% of the time are discarded for this computation. The number of remaining nodes, and the entropy of the distribution, is noted in red.

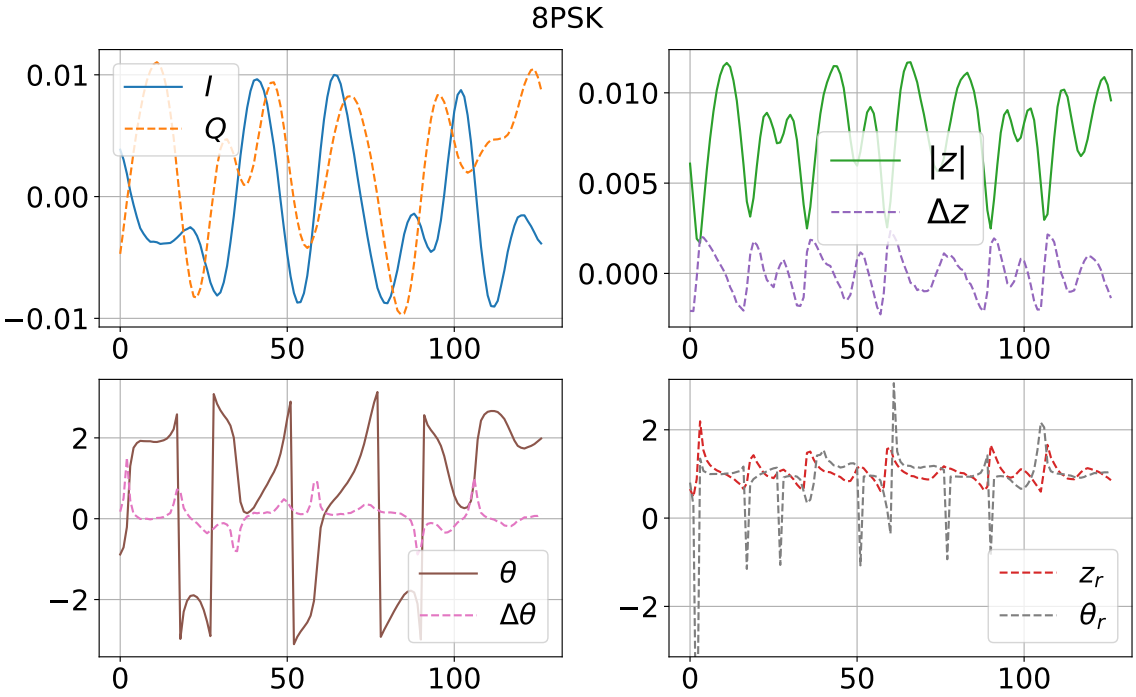


Figure 4.21: **All time series variations of a single example from the DeepSig 2016 dataset.** Raw I and Q signals, and all applied preprocessing methods.

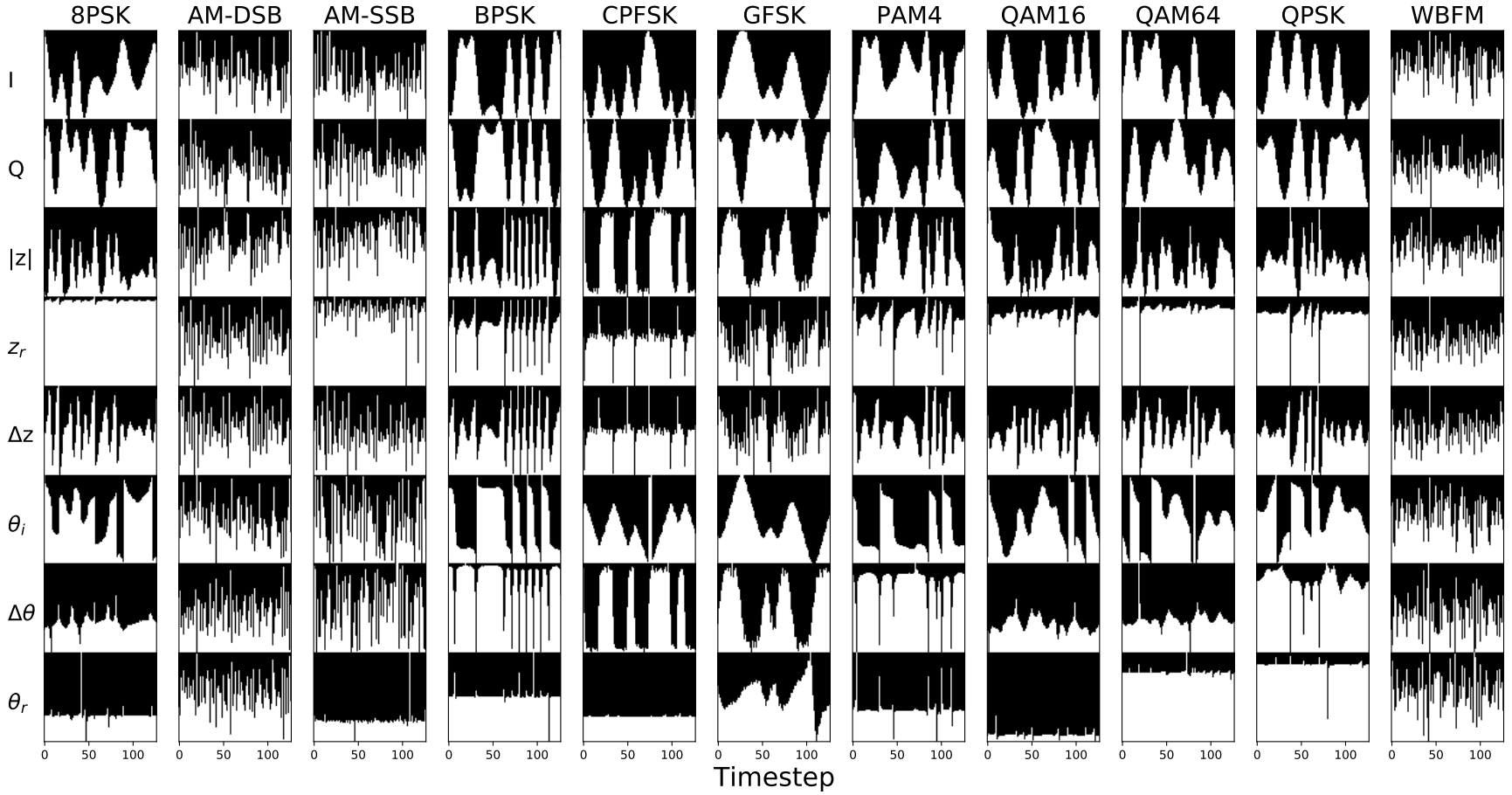


Figure 4.22: **Thermometrically encoded time series variations.** For each sample in the dataset, all eight time series variations shown in 4.21 are scaled to fill the range [0,127] and thermometrically encoded. The encoded waveforms are stacked into a vector of 1024 data bits by 128 timesteps, which is presented to the reservoir one step at a time.

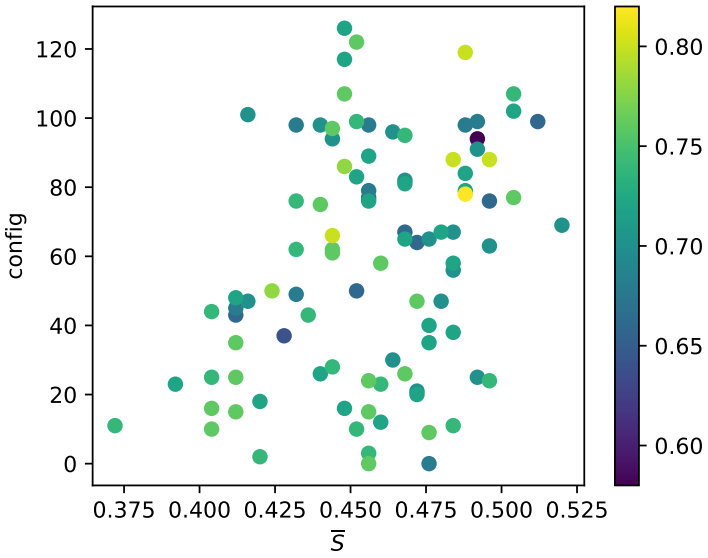


Figure 4.23: **ML accuracy on DeepSig 2018 SNR 18 data processed using an unclocked network.** Color represents validation accuracy, which was up to 82%.

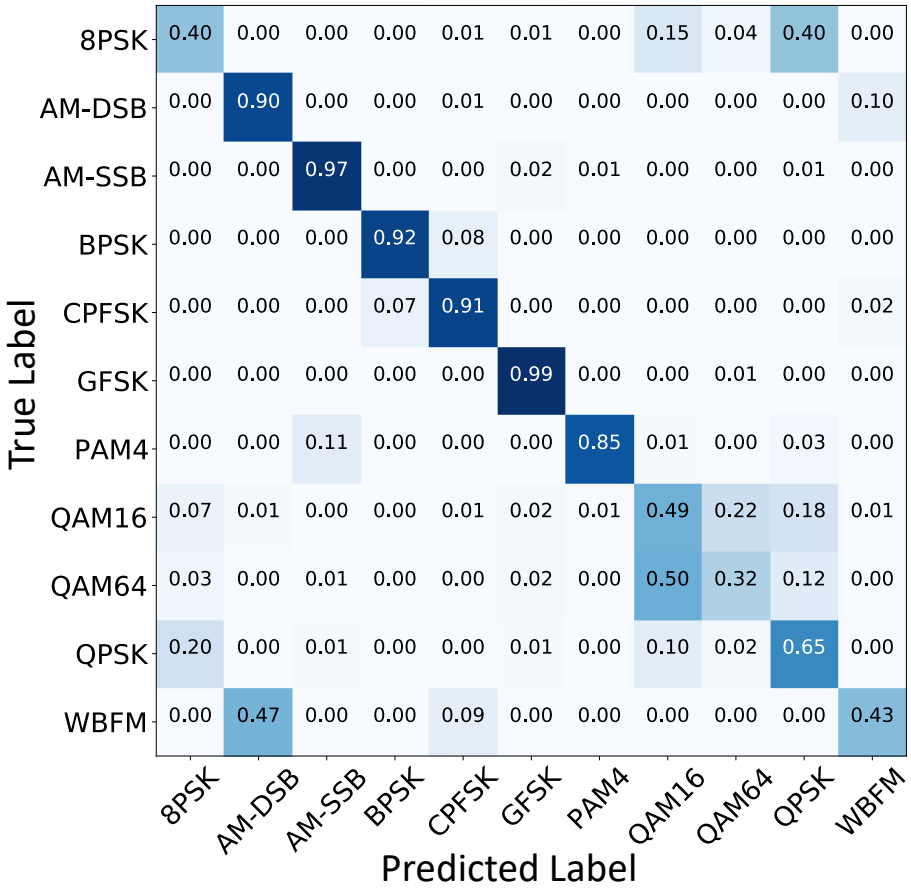


Figure 4.24: **Confusion Matrix for the test set applied to the best configuration.**

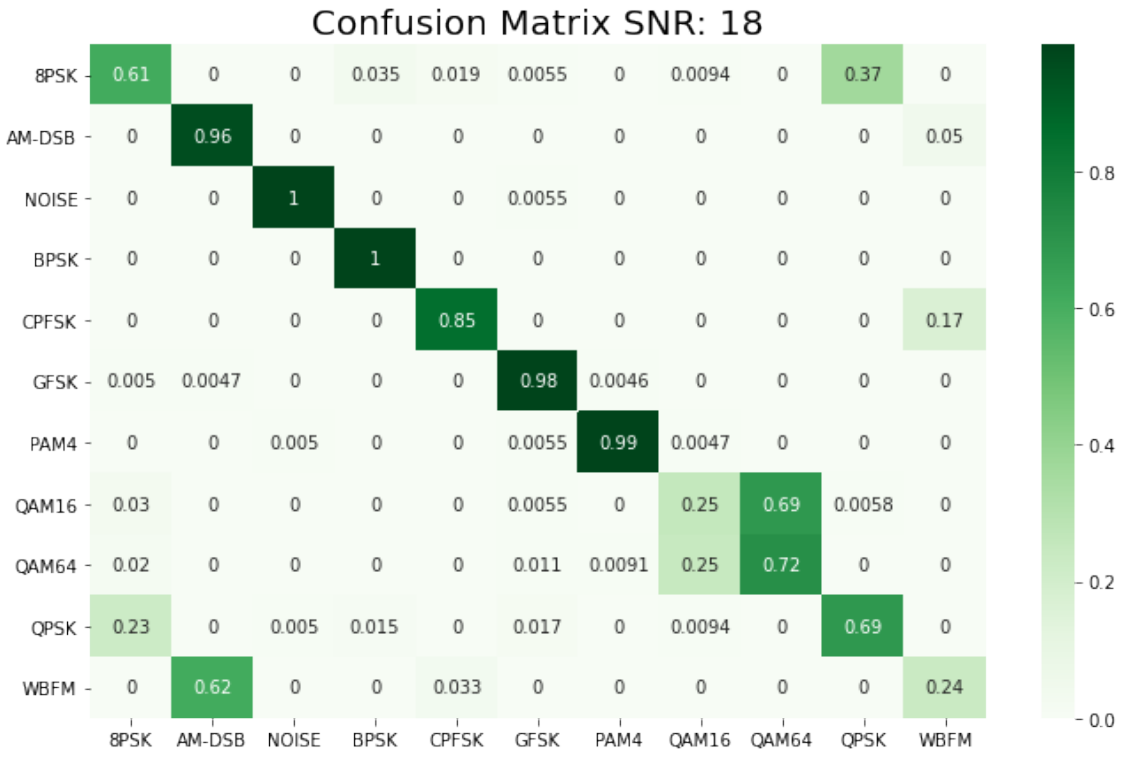


Figure 4.25: **Confusion Matrix for the convolutional neural network on a different randomly selected subset of the dataset.**

Chapter 5: Boolean Reservoir Computing Using Discrete Digital Logic Gates on a Printed Circuit Board

5.1 Introduction

In this chapter[1] we describe the design and test of a printed circuit board for reservoir computing, using discrete logic chips to implement a randomly connected Boolean network. If use of these networks is adopted for practical machine learning problems, then lessons learned from tests of these networks on a PCB can be applied to a miniaturized version implemented in more modern technology such as on an ASIC.

5.2 CMOS Reservoir Circuit Design

5.2.1 Why Discrete Digital Logic Chips?

In this experiment we designed a 64-node reservoir on a printed circuit board using 4000-series digital logic chips. One might very reasonably ask why we would use a decades old technology when there are more modern tools at our disposal. The reasons are:

[1] Portions of this chapter reproduced from Komkov et al. [24].

- In our application, slow speed is beneficial, as we want to study the dynamics of free-running networks and sample the waveforms with high density. Being one of the oldest digital logic families, 4000-series CMOS is relatively slow. In fact, we would have preferred to use a logic family with completely unbuffered outputs for transfer functions with the smallest gains, but to keep the component count reasonable we used a buffered version as not all Boolean functions (namely, XOR) are available as discrete chips in an unbuffered series.

- We are interested in probing the effect of finer voltage quantization than digital tools allow. How much more information can be extracted out of an analog signal versus a digital one in reservoir computing is an open research question. In Komkov et al. [23], similar Boolean networks were studied as the ones presented here. However, on an FPGA, the inputs and output are binarized by design. Digitization can unnecessarily add processing time in some applications when analog inputs and/or outputs are desired.

- Our networks of interest are cumbersome to simulate using Spice tools. They are large and have lots of switching activity. The presence of interacting loops in the circuit makes high time resolution necessary. As it's not clear a priori what circuit topology will make the best reservoir, we want to efficiently test many configurations, which can be impractical if every simulation is slow.

- Finally, the components and assembly are economical. Each chip retails for around $0.50 USD in small quantities, and assembly was done by hand in under four hours.

5.2.2 Circuit Details

The printed circuit board is shown in 5.1. The circuit consists of 64 blocks whose design is shown in 5.2. In the middle of the node is a three-input XOR, comprised of two XORs with two inputs each (only two-input XORs are available in the logic family used, see 5.1 for part numbers). The XOR was chosen because it is equivalently sensitive to changes of ones and zeros in its input. The inputs to the XOR can selectively be shut off by the three NAND gates, controlled by control bits $C1$, $C2$, and $C3$. The output of the XORs is AND'ed with a RESET, which is a global control line that forces the output of every block to be low, which is used to ensure the network has a consistent starting point.

Logic gate	Part number
XOR	MC14070B
NAND	MC14011B
AND	MC14081B
Shift register	MC14094B
Level shifter	MC14504B

Table 5.1: Part numbers with hyperlinked datasheets.

There are 192 total control bits C, which are stored in a shift register. The shift register is loaded with a control word serially, prior to the testing of any network configuration. The board's power supply voltage (V_{dd}) can be varied to probe the dynamics of the reservoir at various voltages. A Teensy[2] 3.2 is used for a serial interface with a control PC to set the digital RESET and shift register lines on the board. The Teensy's 3.3V outputs are converted to the board's voltage using a level shifter. Three external analog inputs each fan out to three reservoir nodes. MMCX

[2] `https://www.pjrc.com/teensy/`

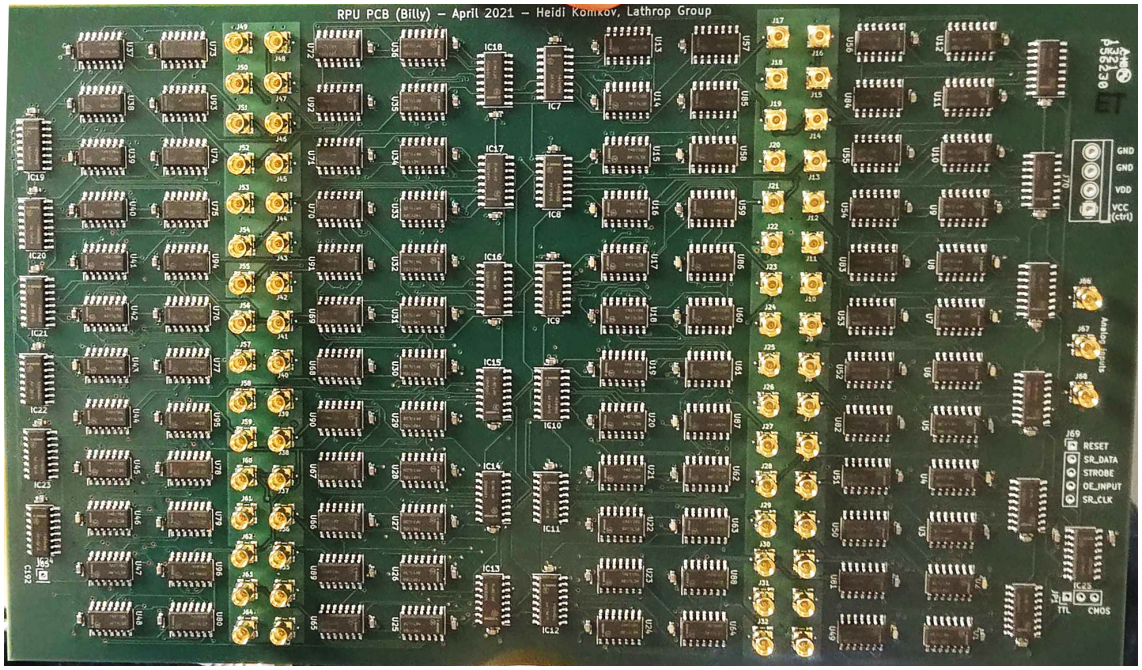


Figure 5.1: Printed circuit board reservoir using discrete logic gates.

coaxial connectors, chosen for their small footprint, were connected to every node.

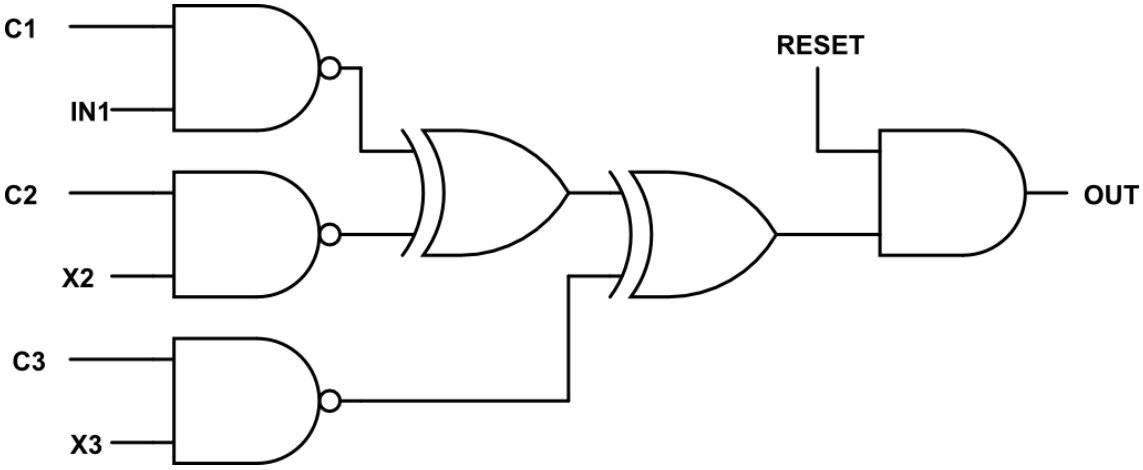


Figure 5.2: Design of a single node in the reservoir.

To design the board, first the randomly populated adjacency matrix describing the network configuration was generated in MATLAB and was checked for no loops of length one. The code generated a simplified net list description that was used to

design the circuit schematic by hand in KiCAD[3], open-source EDA software which was chosen as it supports Python scripting if we would ever want to generate more circuit layouts in an automated manner. Components were placed in a printed circuit board layout by hand in a grid pattern, and the circuit was then auto-routed using the software FreeRouting[4]. The printed circuit board is 6 layers with alternating ground and signal layers.

5.2.3 Waveforms

Waveforms of nodes in the reservoir circuit exhibit a variety of behaviors, including DC steady states, periodic oscillations, and turbulent behavior, any of which may be transient or persistent. The behavior of the node is highly dependent upon the network configuration, supply voltage, and input signal. Figure 5.3 shows four arbitrarily selected channels in a network configuration that is very excitable. In the frequency domain, these waveforms exhibit broadband frequency behavior without any amplitude peaks, an indication of turbulence. While the most frequent voltages are 0 or V_{dd}, these waveforms also contain many incomplete transitions and intermediate voltage values.

The 4000-series chips are rated for operation with a supply voltage in the range 3V-18V. The internal frequencies in the network are highly dependent upon the supply voltage, particularly in the lower part of the voltage range. Dependence of gate transition times—which influence the internal frequencies—on supply voltage

[3] `https://www.kicad.org/`
[4] `https://freerouting.org/`

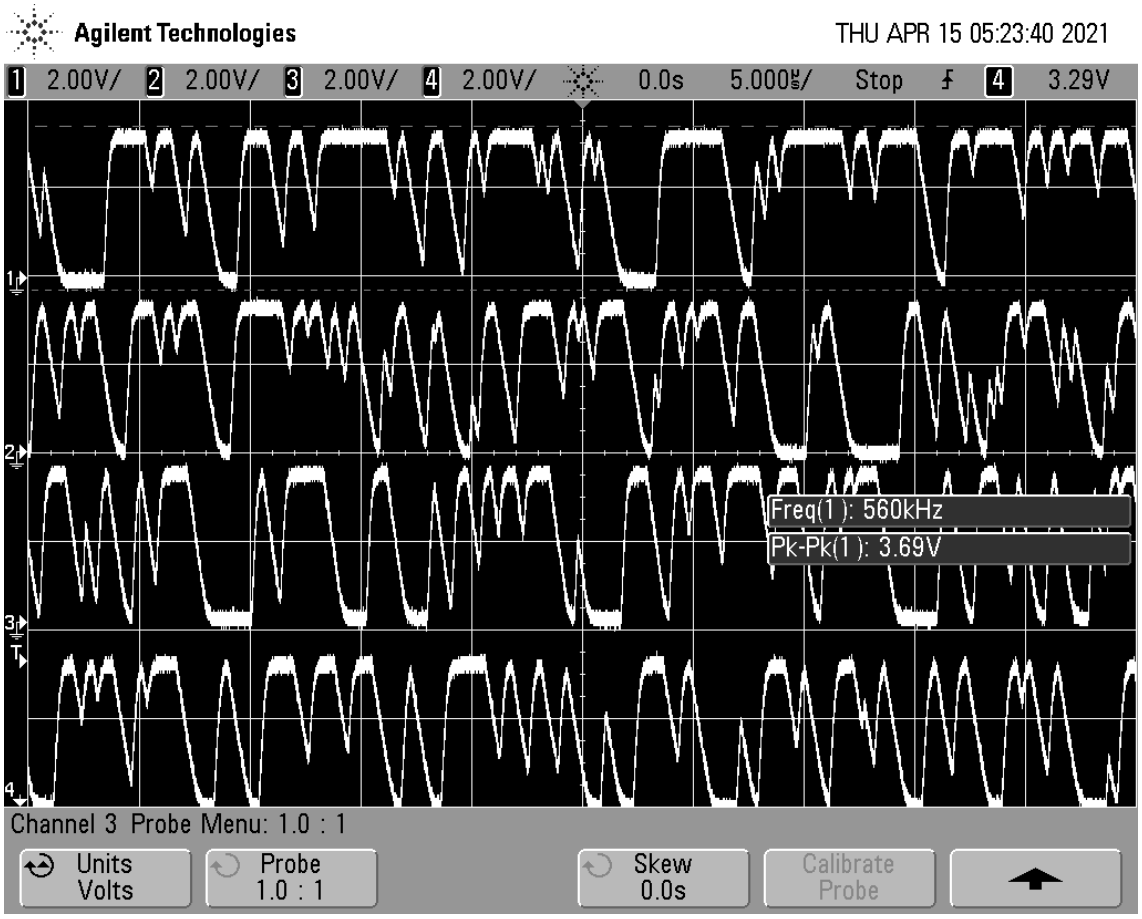


Figure 5.3: Node waveforms in an undriven network configuration with high self-activity.

can be found in the component datasheets. Except where otherwise noted, all tests were performed at 3.3V.

5.2.4 Network activity with no external input

We used circuit current draw as a simple measure of overall network activity. Figure 5.4 shows the current readout from the power supply for 100 network configurations at each of 64 steps of $\overline{S}$ between 0 and 1. For each value of $\overline{S}$, random seeds 1 through 100 were used to randomize the placement of ones in the 192-bit control word. The RESET signal was applied prior to letting the network run freely

to ensure a consistent starting condition.

The resulting characteristic S-shape is consistent with prior FPGA-based measurements of Boolean network activity as a function of average sensitivity [22, 23]. If the behavior of this circuit is similar to FPGA-based Boolean networks, the optimal machine learning performance can be expected in configurations at the base of the curve, at the threshold between low and high excitation.

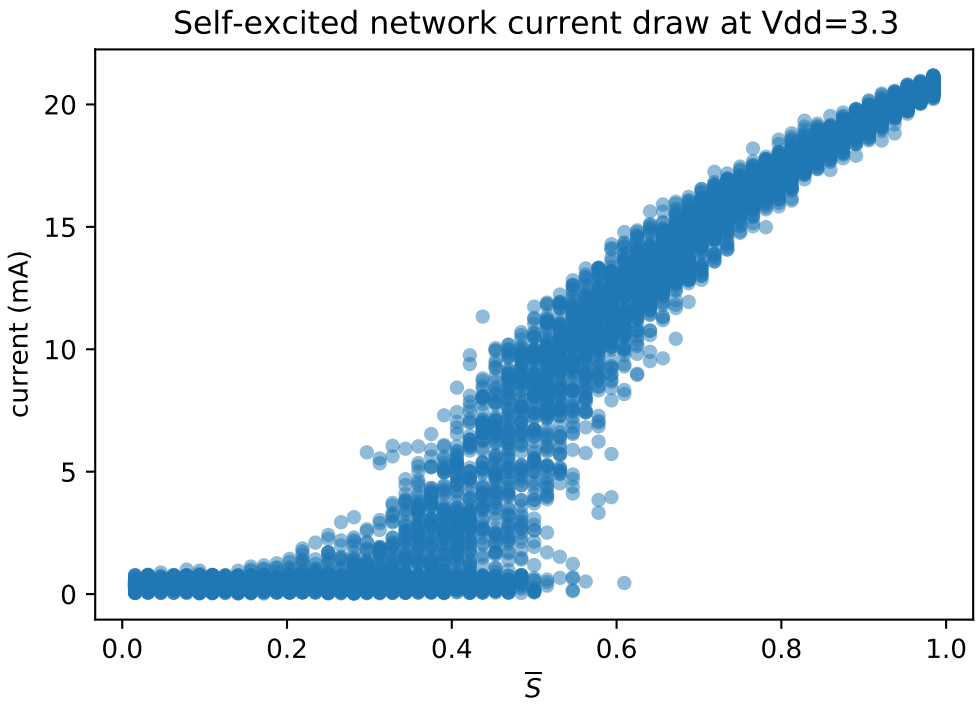


Figure 5.4: Current draw as a function of network Boolean sensitivity.

Current draw as a function of supply voltage V_{dd} is a quadratic curve whose magnitude is proportional to the network's sensitivity. Figure 5.5 shows relationship of steady-state current draw and supply voltage V_{dd}, for 64 self-excited networks with $\overline{S}$ evenly spaced from 0 to 1, with a single random seed. Intermittent excited behavior can be seen at some very low values of $\overline{S}$, where the current jumps from

close to zero to a higher value following the quadratic curve.

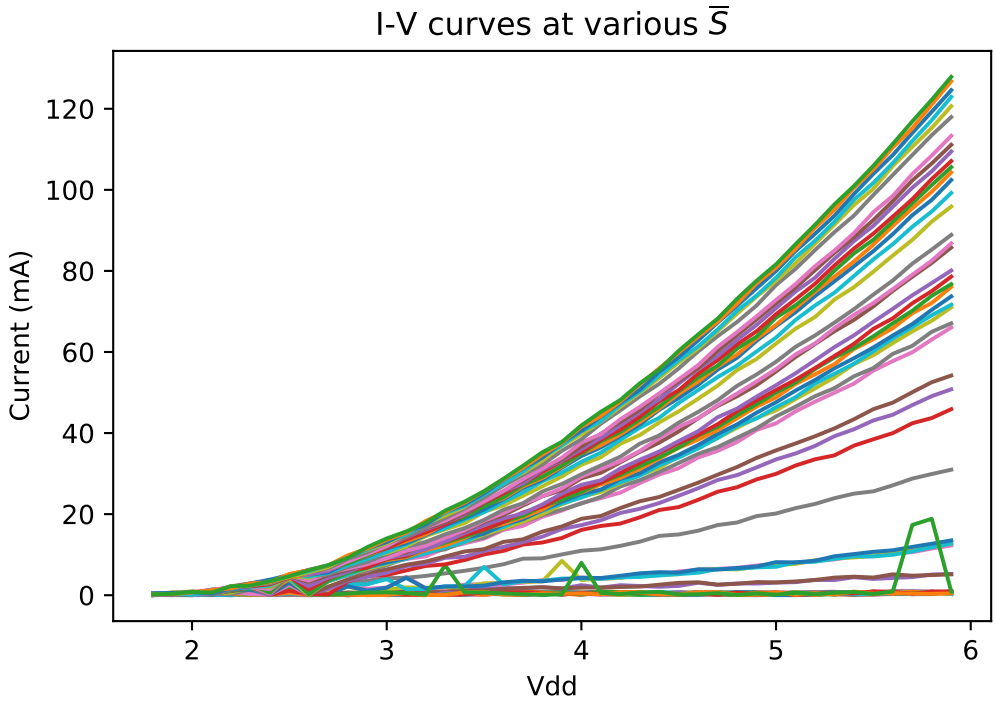


Figure 5.5: Circuit current draw at 64 values of $\overline{S}$ at varying power supply voltages Vdd.

5.2.5 Pulse to Pulse Variation

As we are limited to only sixteen high resolution analog channels that are available on our lab's test equipment, all tests were only on outputs 1 through 61, in intervals of 4. The network can be sensitive to the capacitance of a cable, so the connections were not moved. In all measurements, 1000 data points were captured in 100ns increments.

A condition for reservoir computing is that a network's behavior must be repeatable. For that reason we want to identify networks whose response varies between repetitions of the same stimulus. We apply one period of a 100kHz sine

pulse, whose minimum and maximum are 0 and V_{dd}. To quantify the repeatability, we compute a pulse to pulse variation of the network response to five repetitions of an input stimulus, using the voltage $V(t, c, p)$ which is a function of time t, oscilloscope channel c and time t. The pulse to pulse variation is the standard deviation of a signal between pulses, averaged over time and between channels: $\langle \sigma_p(V(t, c, p)) \rangle_{t,c}$.

We focus on the middle of the sensitivity range as we expect that very inactive and very active networks will be of minimal utility for information processing. The pulse to pulse variation follows the general same trend as the current draw, with an inflection point around $\overline{S} = 0.3$, see figure 5.6. Networks that pass this critical point amplify noise, which pushes them into divergent states, thereby making the network response to a stimulus not repeatable.

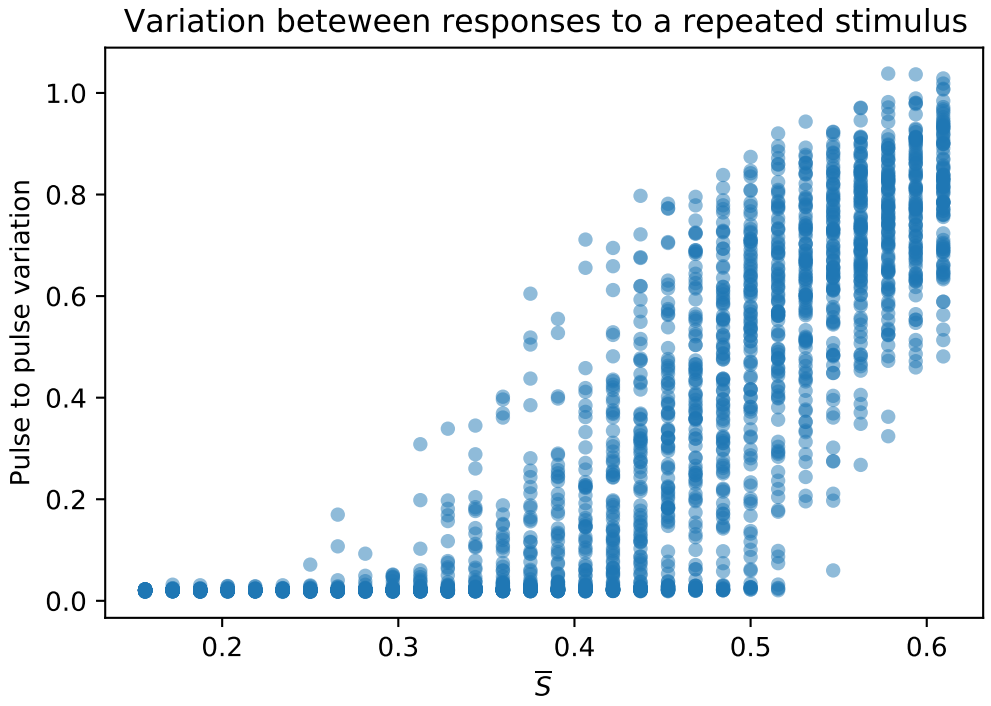


Figure 5.6: Pulse to pulse variation as a function of sensitivity.

5.2.6 Transient Times

Quantifying the amount of time that a network departs from a steady state value after a perturbation can give insight into the amount of memory the network has. To compute the average transient time of the network before it settles into a quiescent steady state, we initially compute the first difference of each waveform in the data to make steady state DC values equivalent. Then we take the absolute value, to equalize the influence of rising and falling edges. Finally we compute a moving average with a window of 10 microseconds to smooth out the signal. We define the transient time as the time it takes for the signal to pass an arbitrarily defined threshold of 0.1. The shortcoming of this method is that periodic steady states would appear no different from chaotic activity.

Networks around the critical point exhibit the longest transient times. At low $\overline{S}$, networks lack sufficient connection density for information to propagate far, and many outputs never change their state. At high $\overline{S}$, networks are so excited that they never reach a steady state (for the purpose of this plot those states are shown as having zero transient time rather than infinite).

5.3 Machine Learning Tests

We test the performance of the reservoir circuit using a time series classification task. As inputs, we use 50 each of sine, triangle, and square pulse waveforms with added noise. The signal amplitude is 2.1V peak to peak, centered at $V_{dd}/2$, and the noise peak to peak amplitudes are varied from 0.3 to 5.5V. The voltage ratios in all

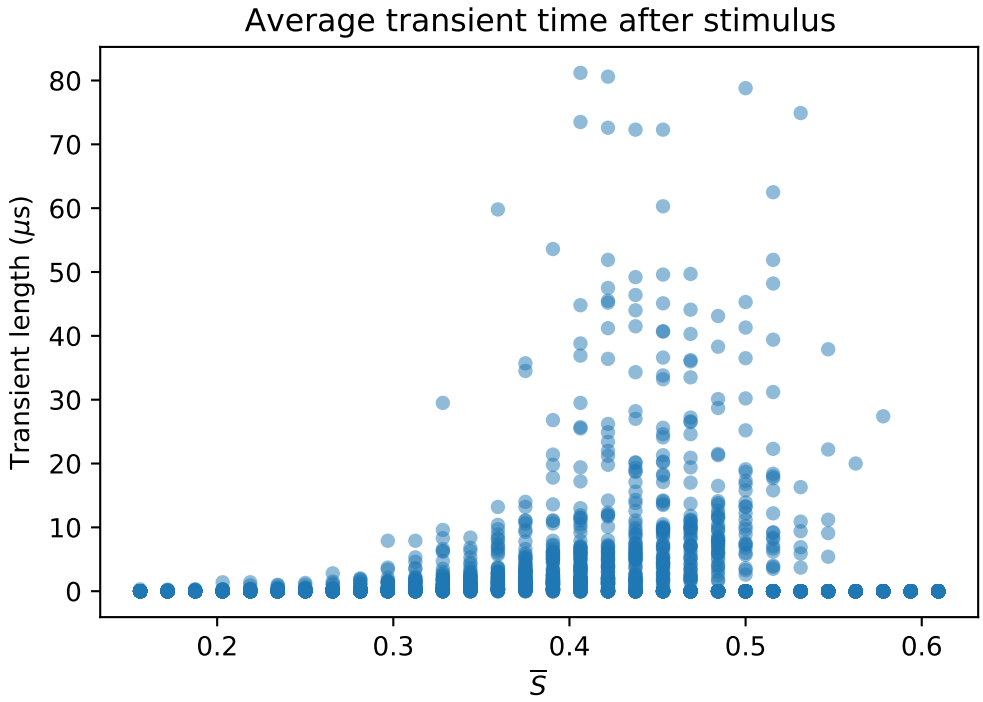


Figure 5.7: Transient times as a function of sensitivity.

plots are all peak to peak voltages. Figure 5.8 shows a sample noisy sine waveform, where one sine pulse is present from 0-10μs, and the noise persists after that. The state of 15 reservoir nodes (outputs 4 through 60 in even steps) are also recorded with a resolution of 0.1μs; these are also shown on the same axes in the figure.

We use scikit-learn's `LogisticRegressionCV`[5], a logistic classifier with cross validation, to compute the reservoir output layer on a supervisory computer. First we compute the accuracy of classification using only the raw input waveform, which we call the baseline accuracy. Then we compare it to the classification accuracy using both the raw input waveform, and 15 reservoir outputs combined, as input to the classifier. There is a 50% train/test split, and 200 samples per waveform are

[5] `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressionCV.html`

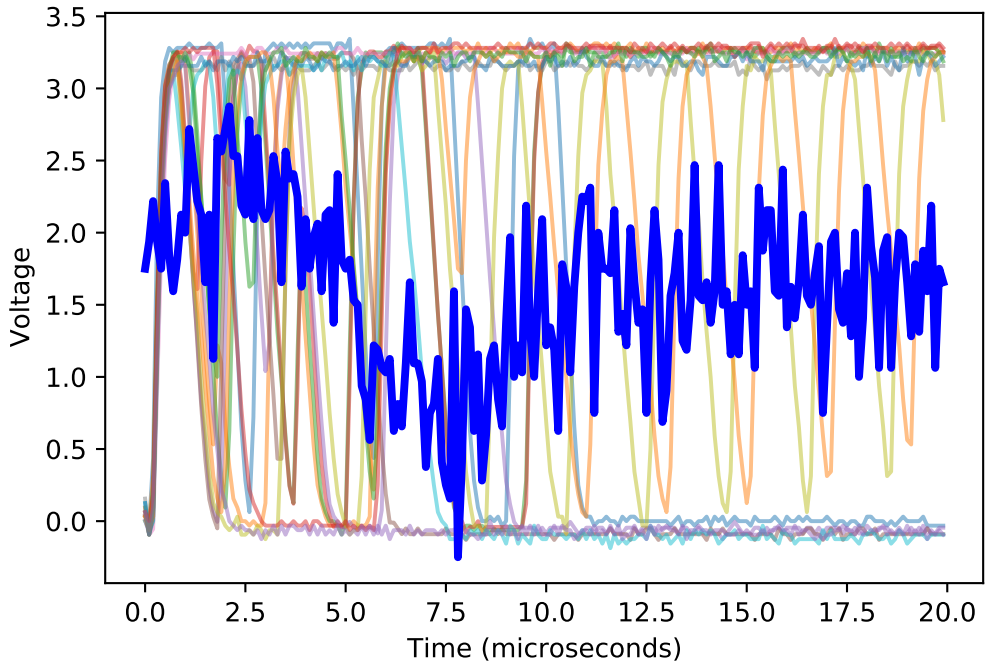


Figure 5.8: Noisy sine wave pulse input, and 15 reservoir outputs on the same axes.

used.

Based upon the results in the prior section, we selected two networks at $\overline{S} = 0.36$ with different random seeds that are repeatable, with a small pulse to pulse variation (0.025 and 0.02) and a reasonable transient time (3μs and 5μs). Machine learning results from these two network are shown in figure 5.9. The reservoir improves classification accuracy significantly at high noise levels, an improvement of up to nearly 40% above classification done on the input alone. The accuracy improvement increases at higher noise levels only because the baseline classifier performs so well at lower noise levels.

We probed the effect of $\overline{S}$ on classification accuracy as well. The accuracy improvement at several noise to signal ratios over a range of $\overline{S}$ from 0 to 1 is

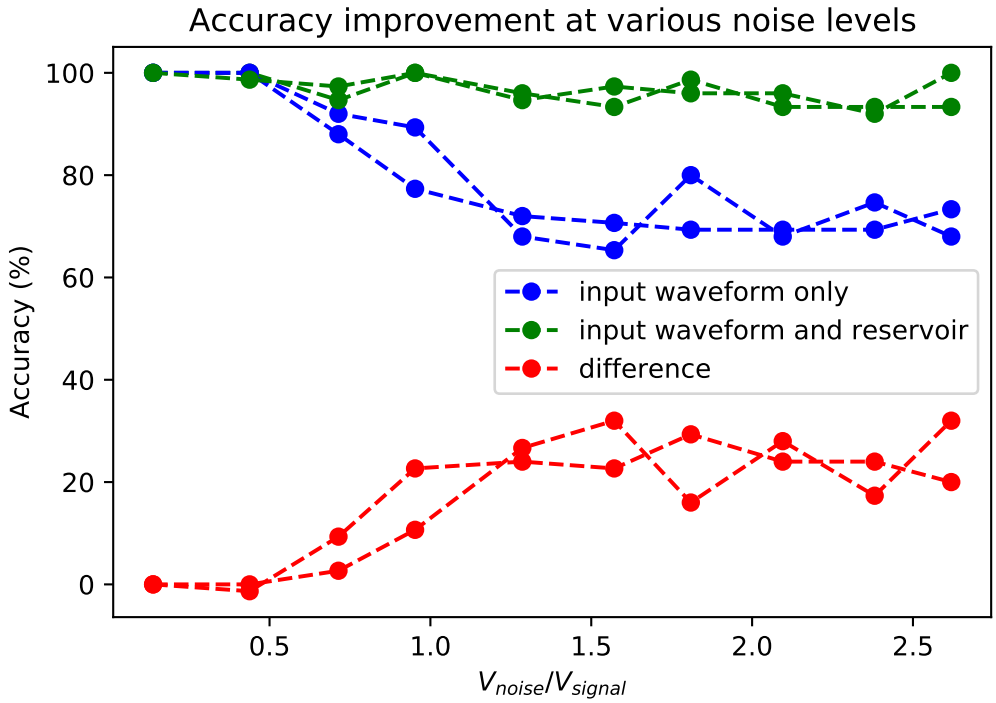


Figure 5.9: Classification accuracy improvement from the reservoir, at various noise levels.

shown in figure 5.10. Networks throughout the whole sensitivity range show strong accuracy improvement. Networks with low excitation contribute appreciably to machine learning accuracy, while overexcited networks seem to help less.

5.4 Conclusion

Using discrete logic chips on a printed circuit board, we implemented a free-running Boolean logic gate network for reservoir computing. We explore the parameter space of configurations using a control setting that selectively shuts off the the inputs to a node in the reservoir, and probe the overall network dynamics that result. Using the reservoir circuit, we demonstrate significant improvement in accuracy in waveform classification, when compared with using the input waveform on its own.

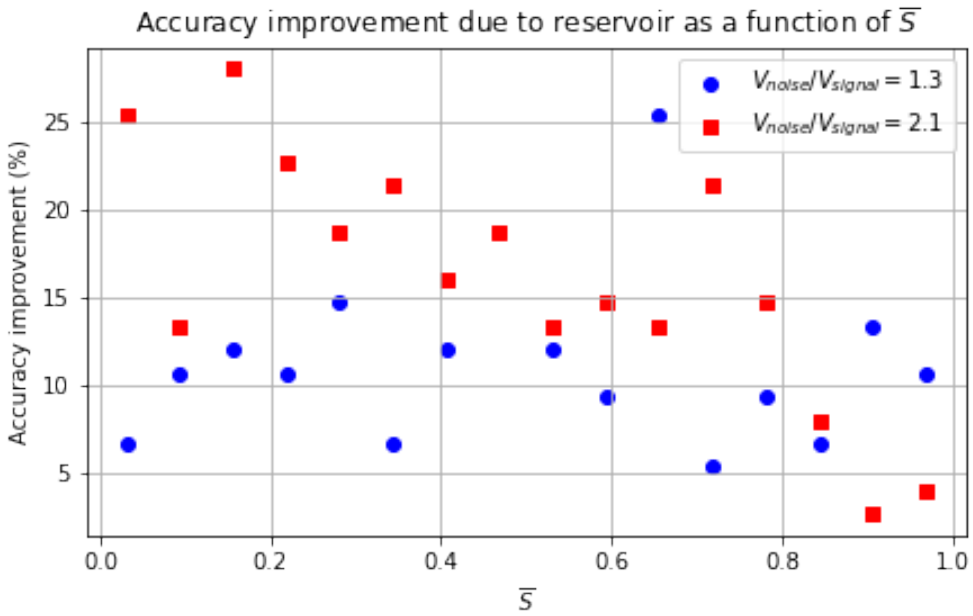


Figure 5.10: Machine learning accuracy improvement over baseline as a function of overall network sensitivity.

While the benchmark task that we used is simplistic, the results are a promising demonstration of the time series processing capabilities of random Boolean networks.

How to quantify the information processing capability of a reservoir remains a challenge. In future work we hope to achieve a better understanding of the added information content from a reservoir within analog waveforms, compared to the information encoded temporally in the timing of digital pulses.

Chapter 6: Design of a 180nm ASIC for Reservoir Computing

6.1 Overview

An application-specific integrated circuit (ASIC) is a microchip designed to implement a specific electronic design and application. We designed seven ASICs for reservoir computing in 180nm CMOS, with the aim to demonstrate rapid machine learning using Boolean logic gate networks. The chip, shown in 6.1, has been manufactured and now awaits post-processing and testing after this dissertation is complete. Designs are based on the designs in the patent application by Lathrop, Restelli and Komkov [25].

6.2 Process Details

The wafer was manufactured using the SilTerra C18G 180nm CMOS process[1], which has a 1.8V core voltage, 3.3V pad voltage, and up to six metal layers. Our design was sized to fit one-sixth of a multi-project reticle[2]. We were provided a standard cell library from the foundry, as well as an ARM I/O pad library. All of the standard cells and I/O pad designs, timing specifications, and power specifications

[1] `https://www.silterra.com/c18g-180nm-cmos-logic-1-8v-3-3v`

[2] We were generously provided wafer space at no cost by Advait Madhavan and Jabez McClelland from the National Institute of Standards and Technology (NIST).

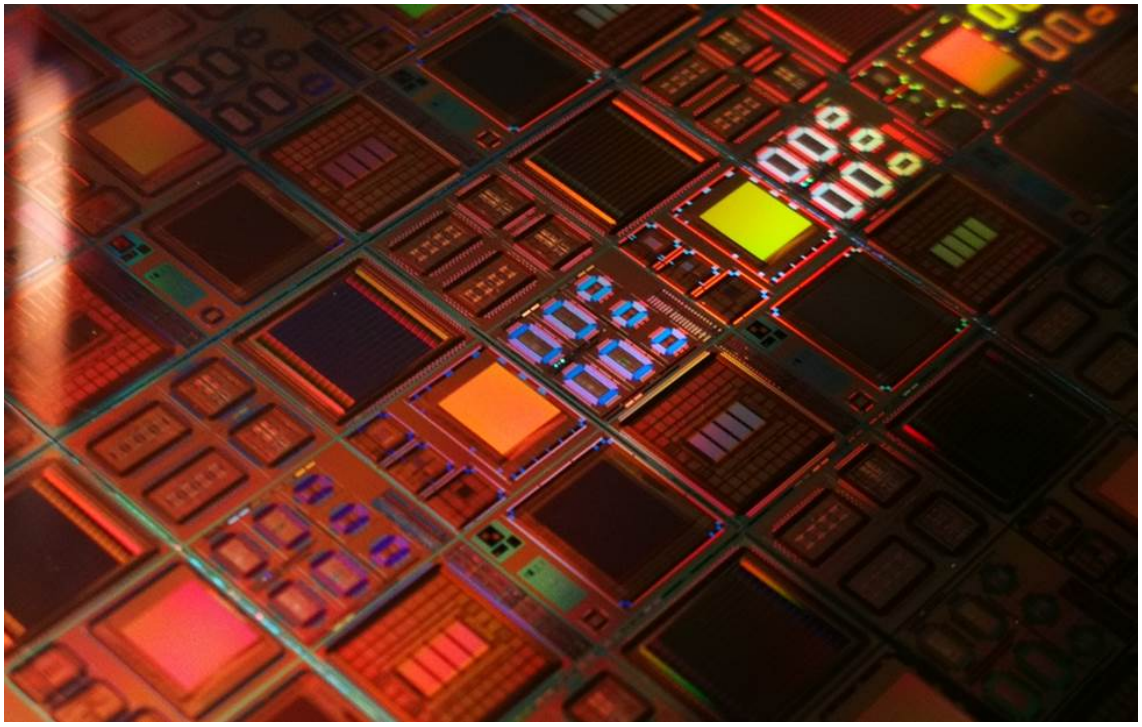


Figure 6.1: Our group of seven reservoir ASICs is shown in the middle of this multi-project wafer.

are proprietary, so any details discussed this chapter are intentionally left without proprietary details.

6.3 Design Overview

We chose to use the provided standard cell library, despite its proprietary nature not in line with the spirit of open research, because it would allow us to immediately use automation tools in Cadence to aid in the layout and routing of the cells. The standard cells contain the designs of various logic blocks down to the silicon level.

The original intention was to create large networks with thousands of nodes, but upon further discussion it was decided to instead create relatively small, 64-node

reservoirs so that every node state could be probed. Probing only a subset of the nodes would unevenly load the reservoir, as routing a connection from the chip core to outside test equipment adds significant capacitance which slows down that node. Additional digital buffering is antithetical to the aim of using the analog dynamics of the circuit for computation.

To connect the chip's core, in which the standard cells are laid out, to the outside world, input/output (I/O) pads are used. The I/O pads are rectangular elements with a standard size that contain a connection to the core one side, and a metal pad on the other side, to which a wire bond is made. The two categories of I/O pads available are analog and digital. The digital pads are buffered, while the analog pads are straight-through pads with a few hundred ohms of resistance between the core and the metal pad. Each analog pad has about 2pF of capacitance. As a result of reading out each of the 64 nodes, the design is pad-limited, having a large pad ring but very low utilization within the core.

Reservoir performance can be highly dependent up the graph connectivity in ways that are not fully understood. To mitigate the risk of any one particular design not working well, seven separate 64-node reservoir chips were designed with various combinations of adjacency matrices, logic gate mixtures, and I/O configurations were manufactured across the seven chips. We call each of the chips a *chiplet*. The chiplets can later be diced apart, or they can be wire-bonded together directly on the wafer to implement one of the multi-reservoir configurations described in Chapter 2.

The mixture of configurable logic blocks in each of the seven chiplets is in table

6.1. Each chiplet has a mixture of blocks that allow a range of sensitivities to be interrogated. Four different randomly generated adjacency matrices were used.

6.4 Node Designs

The configurable node designs used are shown in figure 6.2 and are described below. Each chiplet was designed with a combination of gates to fully sweep the parameter space between $\overline{S} = 0$ to $\overline{S} = 1$.

- Figure 6.2(a) shows a **3-input NAND block** with a control bit (thereby using a 4 input gate). The control input C has the ability to effectively turn off the node by forcing the output high when C=0. When C=1, the output is equivalent to a 3-input NAND with X1, X2, X3 as inputs. When C=0, S=0. When C=1, S=0.25.
- Figure 6.2(b) shows a **3-input NOR block** with a control bit (thereby using a 4 input gate). Like in the NAND case, C has the ability to turn off the gate by forcing the output low when C=1. When C=0, the NOR acts like a 3-input NOR with X1, X2 and X3 inputs. When C=1, OUT=0 and S=0. When C=0, OUT=NOR(X1,X2,X3), S=0.25. Graphs of 3-input NAND, NOR and XOR are shown in figure 3.6.
- Figure 6.2(c) shows a **2-NAND-3XOR block**. When C=1, the inputs X1 and X2 are inverted and passed on to the 3-input XOR. When C=0, the inputs X1 and X2 are blocked, and the three-input XOR effectively only has one input. When C=0, S=0.33. When C=1, S=1.

- Figure 6.2(c) shows a **2-NAND-3XOR block**. When C=1, the inputs X1 and X2 are inverted and passed on to the 3-input XOR. When C=0, the inputs X1 and X2 are blocked, and the three-input XOR effectively only has one input. When C=0, S=0.33. When C=1, S=1. The graph when C=0 is shown in figure 6.4.
- Figure 6.2(d) shows a **3-NAND-3-XOR**. The NANDs selectively turn off the inputs to a 3-input XOR gate. When C's are 0, S=0. When one C=1 and two are 0, S=0.33, When two C's are 1 and the other is 0, S=0.66. When all C's are 1, S=1. Graphs with various control settings are shown in figure 6.3.

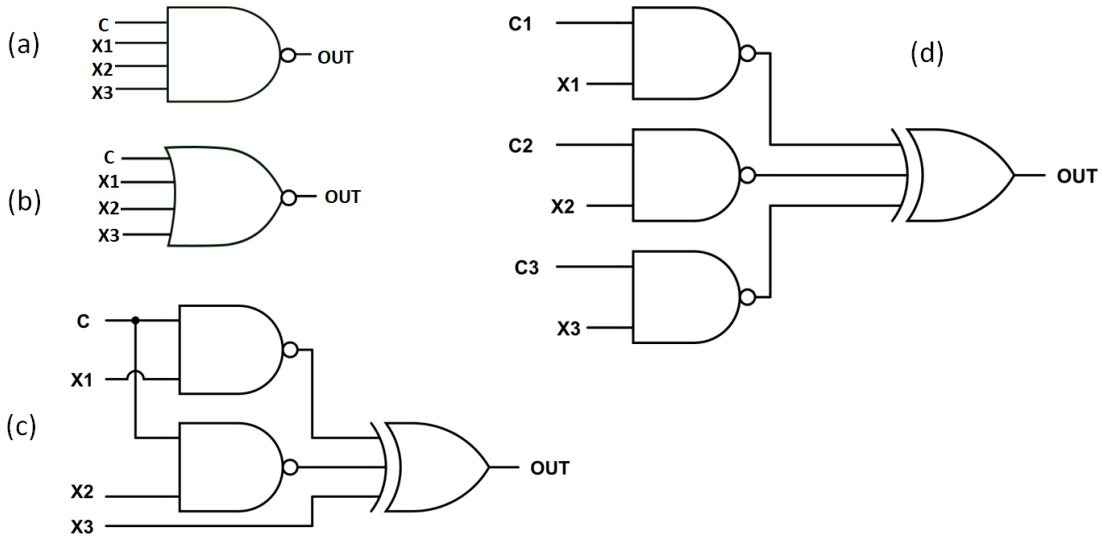


Figure 6.2: Configurable node designs.

6.5 Chip Control

Figure 6.5 shows two reservoir blocks along with the supporting circuitry. We will discuss the elements of the diagram from left to right.

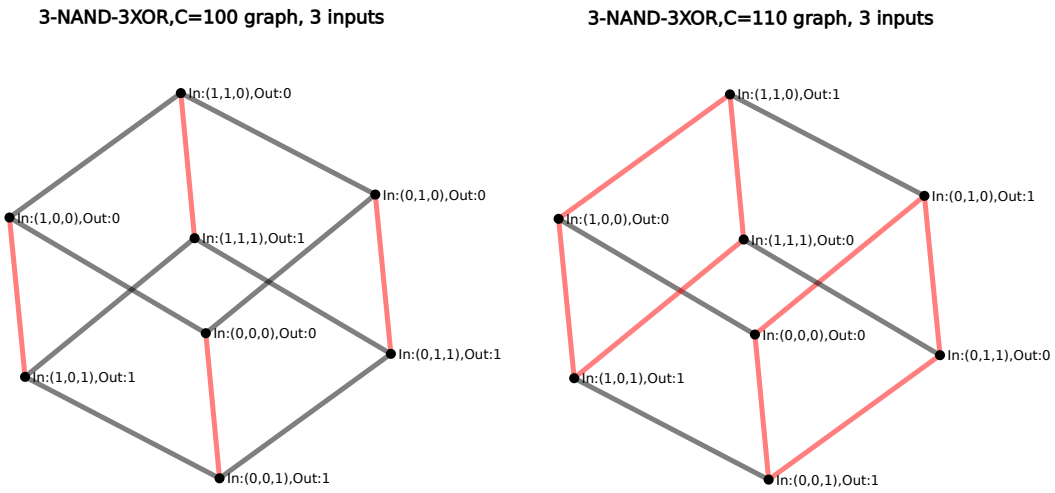


Figure 6.3: Graphs of configurable XOR gates with various control settings.

Control shift register: SR_C_CLK and SR_C_DATA are clock and data lines that feed a shift register containing the 64-bit control word, C, which determines node configurations. The control vector needs to be loaded in serially before the reservoir is run. Reservoir blocks: Blocks have three data inputs (X1, X2, X3) which are either recurrent connections from other blocks, or external inputs. In each chiplet, there are a total of 18 external inputs to blocks, from three analog pads that connect to 6 blocks each.

Reset line: To ensure that the reservoir always has a repeatable starting condition before evolving autonomously, the RESET line is AND'ed with the output from each block. When RESET is zero, the output from every block is zero.

Multiplexer select line: MUX_SEL selects whether recurrent connections to the back to the reservoir blocks are selected from before or after the output buffer (BUF). The buffer helps to drive the pad capacitance, but it introduces on the order of 1ns more delay. If the feedback connection before the buffer is selected, the

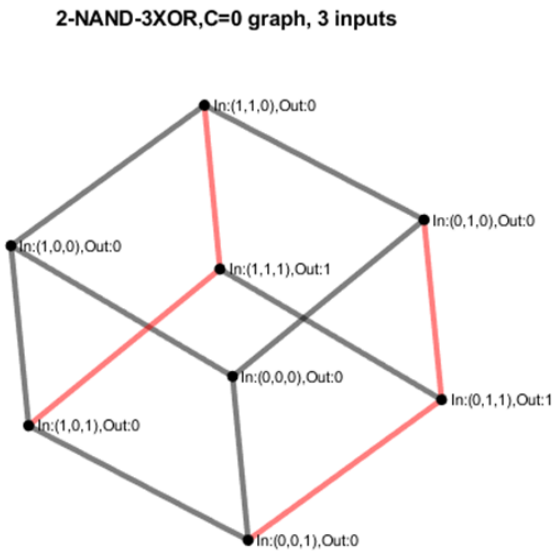


Figure 6.4: Graph of the 2-NAND-3XOR gate with C=0.

reservoir will run faster than if the feedback connection is selected after the buffer. However, output pads may not be able to transmit the full range of the reservoir's activity if its constituent frequencies are too fast due to their own slew rates. Details will vary by chip, which use different output pads. More information is in table 6.1.

6.6 Speed and Power Estimates

While details are left somewhat vague by necessity due to the proprietary nature of the standard cell designs and speeds, chiplets with analog pads (Andy, Billy and Denny) will exhibit frequencies of about 150MHz, while those with digital pads (Charlie) will exhibit about 3GHz frequencies, depending upon the configuration. Exact timing and power consumption information of the standard cells is proprietary. Therefore these numbers below are gross estimates.

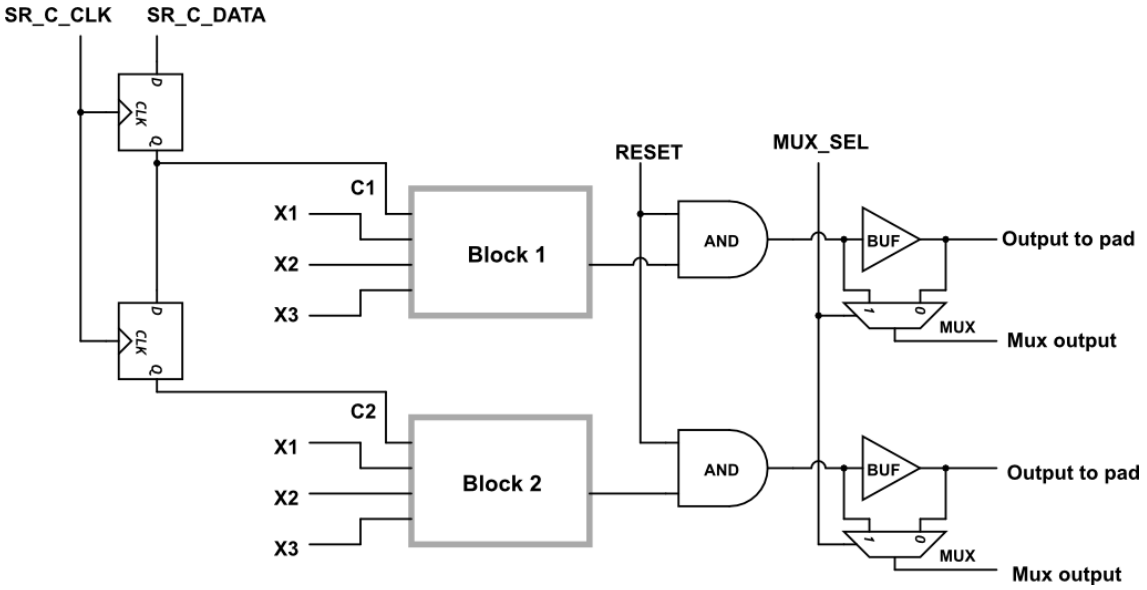


Figure 6.5: Reservoir block diagram.

Chiplet name	Adjacency matrix	Gate mixture	Output pads
Andy	A	25% NAND,25% NOR, 50% 2-NAND-3XOR	64 PANALOG
Billy	A	100% 3-NAND-3-XOR	64 Analog
Charlie	A	100% 3-NAND-3-XOR	19 digital output pads (highest slew rate, high-noise pad) 45 digital output pads (slowest slew rate, low-noise pad)
Denny	B	25% NAND,25% NOR, 50% 2-NAND-3XOR	64 Analog
Ernie (inference)	A	100% 3-NAND-3-XOR	slowest slew rate, low-noise digital pads, 16-bit binary output
Freddy (inference)	C	100% 3-NAND-3-XOR	slowest slew rate, low-noise digital pads, 16-bit binary output
Gary (inference)	D	100% 3-NAND-3-XOR	slowest slew rate, low-noise digital pads, 16-bit binary output

Table 6.1: Chiplet details.

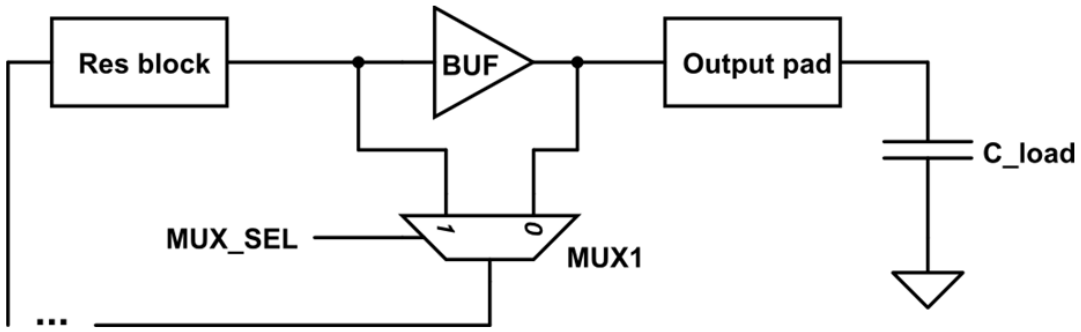


Figure 6.6: The multiplexer selects the source of the recurrent connections to other reservoir blocks. Recurrent connections from before the buffer result in a faster configuration, but the slew rate of the output pad may be too large to keep up, resulting in significant signal degradation. Recurrent connections from after the buffer will be significantly slower in speed due to the load capacitance.

	1pF loading	10pF loading
Analog I/O pads	150MHz, 0.57mW	45MHz, 0.15mW
Digital I/O pads	3GHz, 11.5mW	3GHz, 11.5mW

Table 6.2: Slow configuration estimated frequency and maximum steady-state power draw.

	1pF loading	10pF loading
Analog I/O pads	3GHz, 10mW	3GHz, 10mW
Digital I/O pads	3GHz, 11.5mW	3GHz, 11.5mW

Table 6.3: Slow configuration estimated frequency and maximum steady-state power draw.

6.7 Inference chip details

While output layers for four chips (Andy, Billy Charlie and Denny) must be computed externally to the chip, three chips (Ernie, Freddie and Gary) are capable of multiplying the 64 Boolean reservoir states by 10-bit weights, performing output calculations. While this scheme does not take advantage of the analog nature of the reservoir, it performs full inference once trained, avoiding the latency of

communicating with an external controller.

Figure 6.7 shows a block diagram of the inference chips. Each of the blocks are separate Verilog modules which were tested independently for correct digital functionality. The chip was then synthesized entirely from Verilog. Figure 27 shows a picture of Ernie, a much more crowded layout than the reservoir chips. The majority of the utilization in the chip is taken up by the shift registers that store the weights (the only memory type available for this project), and the adder that sums all the weights.

Control shift register: SR_rescontrol is the shift register storing the reservoir control word (two of the constituent D-flip flops are visible in Figure 23). It has one readout for debugging purposes, SR_C191. Reservoir: As before, the 64-node reservoir has three analog inputs fanning out to 6 internal blocks each. The reservoir has a reset signal. In all of the inference chips, the reservoir will have a frequency of about 4 GHz, as it is not loaded down by output pins.

Reservoir buffer: Upon the rising edge of RB_clk, the reservoir state is copied to a PIPO buffer.

Reservoir buffer readout: Using a collection of 4 to 1 multiplexers, the four buffer select lines, Buf_sel[0:3], can be used to read out the reservoir buffer four states at a time. This is necessary to compute the weights outside of the chip.

Weight shift register: 64 weights of 10 bits each can be serially fed in using SR_W_data and SR_W_clk lines. The last D-flip flop at the end of the shift register has a readout, SR_W639, for debugging.

Mixed precision multiplier: The 64 one-bit states of the reservoir are mul-

tiplied by the 10-bit weights. One-bit multiplication is performed quite simply by an AND gate.

Adder: The 64 outputs from the multiplier are then summed to produce a 16-bit binary output which goes to 16 pins on the chip.

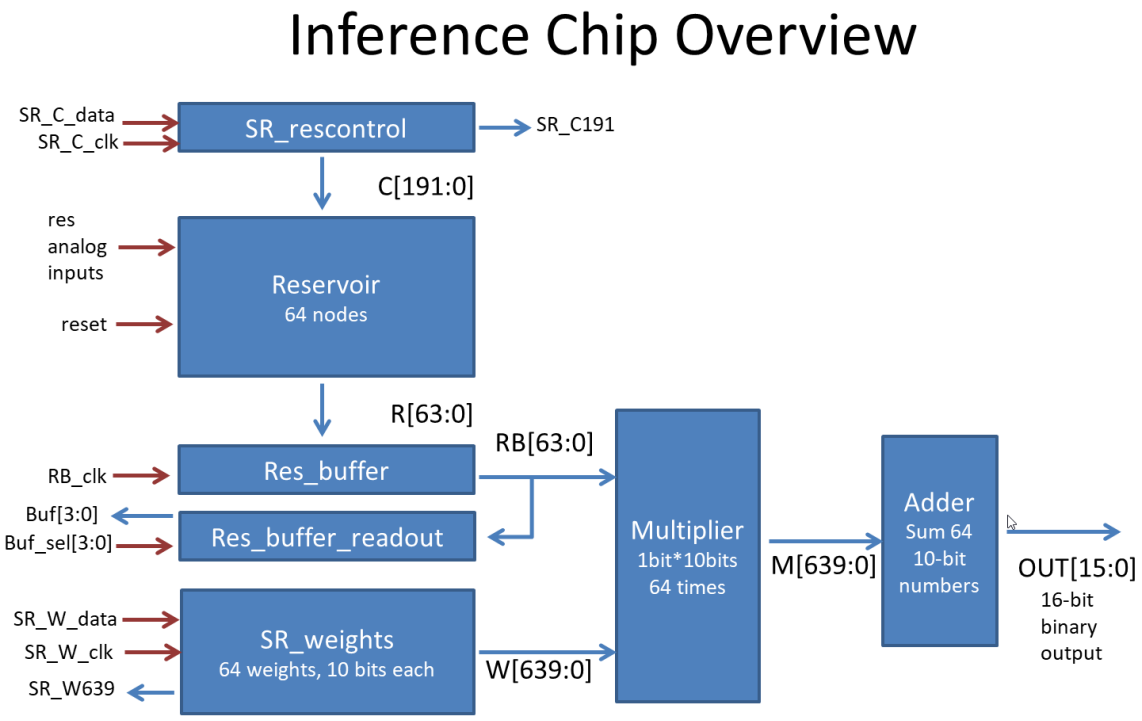


Figure 6.7: Block diagram of the inference chips (Ernie, Freddie, and Gary). Each box is a separately tested module. All of the logic was placed automatically using Cadence tools, which made layout easy.

6.8 Automated Design Workflow

The design flow starts in Matlab, where a 64×64 binary adjacency matrix is randomly populated, describing how logic blocks inside of the reservoir connect to each other. A simplified 3-block example is shown in figure 20. The configuration in the matrix shown in figure 6.8 (a) is used to auto-generate Verilog code 6.8 (c) describing the network 6.8 (b).

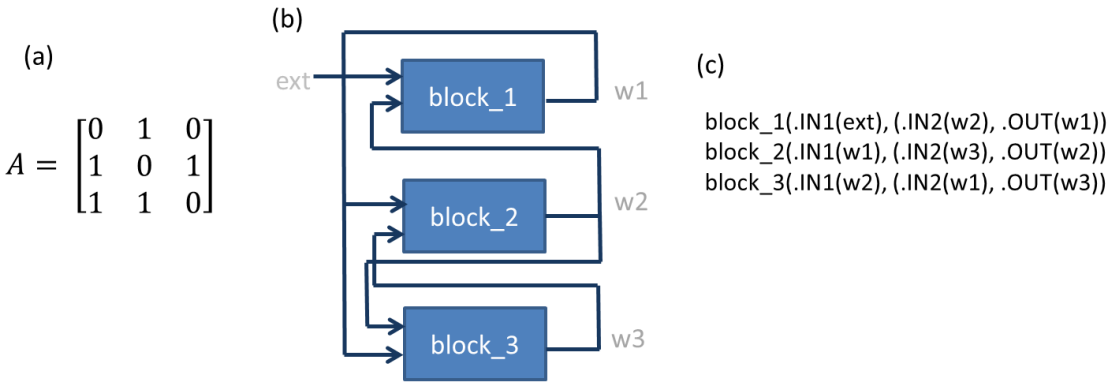


Figure 6.8: Network synthesis steps. (a) Is a randomly populated adjacency matrix. (b) is a diagram representing the same information as in matrix A. (c) is the equivalent Verilog code.

Cadence Genus was used for logic synthesis, translating the structural Verilog into synthesized Verilog describing the configuration using available standard cells. As network configurations like this are very different from the synchronous logic designs that these tools are typically used for, care was taken so that no logical simplification would happen in this process (for example, synthesis tools may simplify three inverters in a ring oscillator to only one inverter, as they are logically equivalent). Cadence Innovus was used for floorplanning the chip, placing and routing the cells. Finally, the design was loaded into Cadence Virtuoso for manual error-checking and DRC. Screenshots from Innovus and Virtuoso, respectively, are shown in figure 6.9.

6.9 Current Progress

ASIC samples were received in late 2020. Due to the needs of other groups with designs on the wafer, the top layer manufactured was a layer of vias which

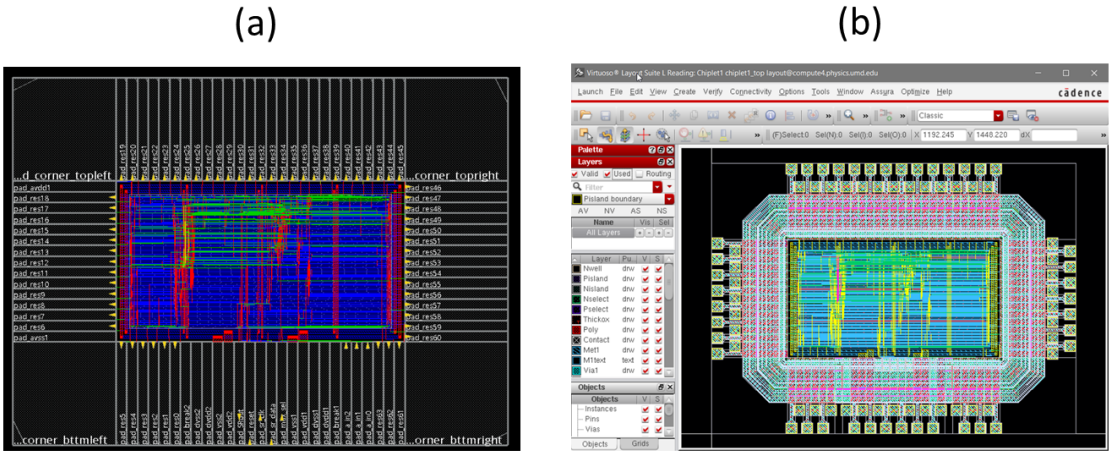


Figure 6.9: (a) Cadence Innovus takes synthesized Verilog files, auto-places standard cells and auto-routes tracks between them. (b) The final design is imported into Cadence Virtuoso for DRC.

are intended to connect the fifth and sixth metal layers. The sixth metal layer was not manufactured. For this reason, post-processing of the wafer in the UMD NanoCenter was necessary to deposit metal over the vias to create pads for wire bonding. Effort in early 2021 to metallize the chips was only partially successful. Several iterations of photomasks were made to expose only the pad areas; the final photomask that allowed for easy alignment was made using iron oxide which is transparent in the visible part of the spectrum but opaque to ultraviolet light. A 50nm titanium adhesion layer and 100nm of gold was deposited using the e-beam evaporator. However, the deposited metal lifted off the chip easily during an attempt at wire bonding, likely due to the presence of organic contaminants on the surface of the wafer which was not adequately removed. The existing metal will need to be etched off and the process attempted anew.

A printed circuit board for testing the chip, shown in figure 6.10, was designed and manufactured. Each chip will be placed into a pin grid array (PGA) package

which will be inserted into a zero insertion force (ZIF) socket. The test board, shown in figure 6.10, routes each chip output to a coaxial connector. The chip's digital controls will come from a Teensy[3] microcontroller, which will be inserted into a socket directly on the board. A similar workflow that was developed for the PCB described in Chapter 5 will be applied to testing the ASIC.

A group of chiplets positioned in a PGA package in the wire bonder is shown in figure 6.11, and the same group of chiplets viewed through the wire bonder's microscope is shown in figure 6.12.

6.10 Looking Ahead

For future work, it would be advantageous to use an open standard cell library, or even to design custom standard cells at the transistor level, so that waveforms will be measurable and publishable without concern. At this stage of research, an older process node and slower speed chips would only be advantageous, to allow for higher-density sampling of waveforms and/or using less specialized lab equipment.

See Appendix A for all chip images, pinouts, and suggested connectivity in a PGA package.

6.11 Conclusion

We designed and manufactured seven 180nm ASIC designs of reservoirs, which await testing. Three of the seven can perform inference, while the other four are

[3] `https://www.pjrc.com/teensy`

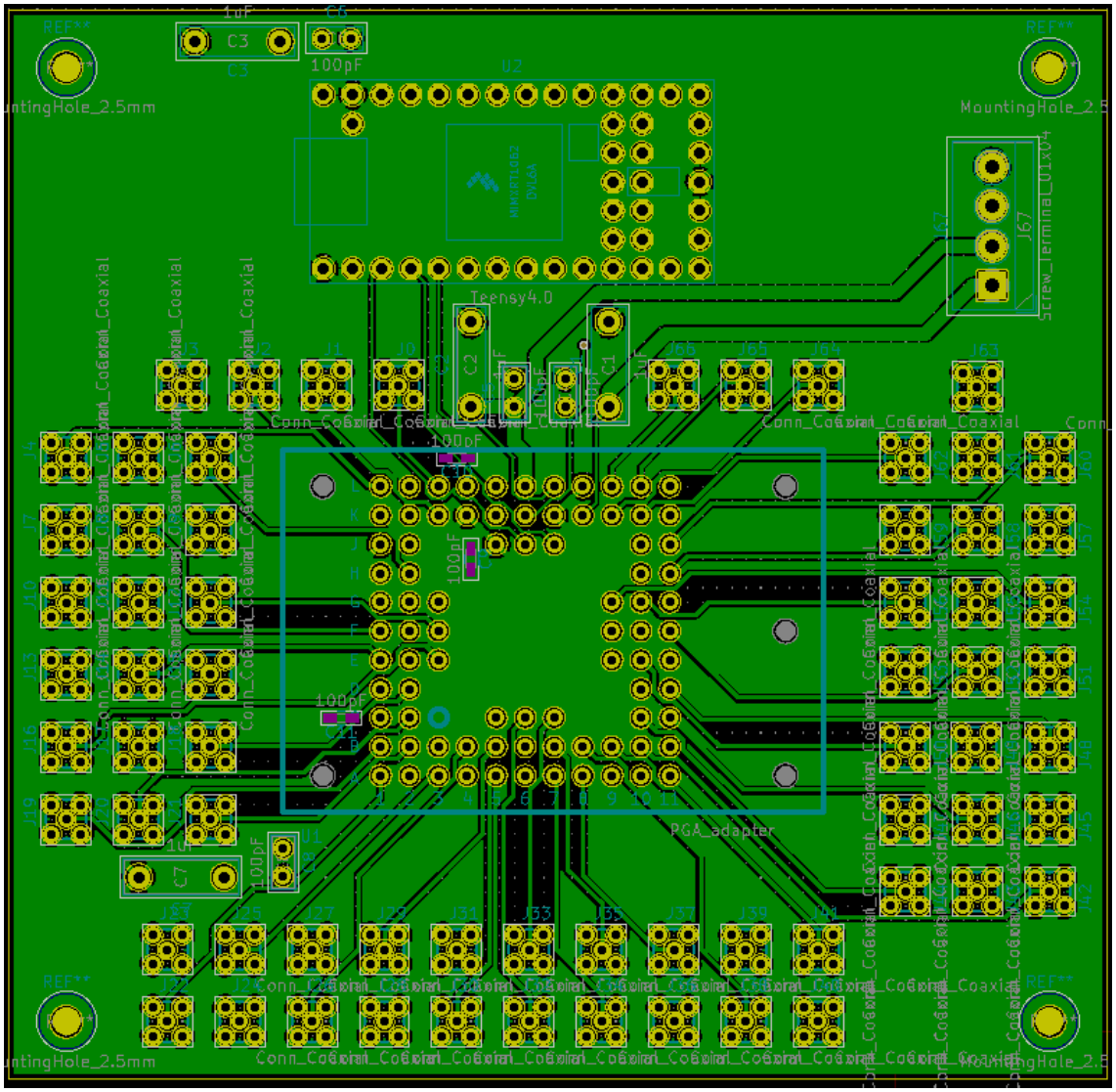


Figure 6.10: Printed circuit board for testing the ASIC.

only reservoirs with direct readouts from every node. While reservoir topology in the ASIC is fixed, using control settings to change the logical function of each node in the reservoir allows for some amount of reconfigurability. To mitigate the risk of any one particular design not working well, a variety of designs with different adjacency matrices, logic gate mixtures, and I/O configurations were manufactured across the seven chips.

A highly efficient workflow was developed in MATLAB and Cadence tools to

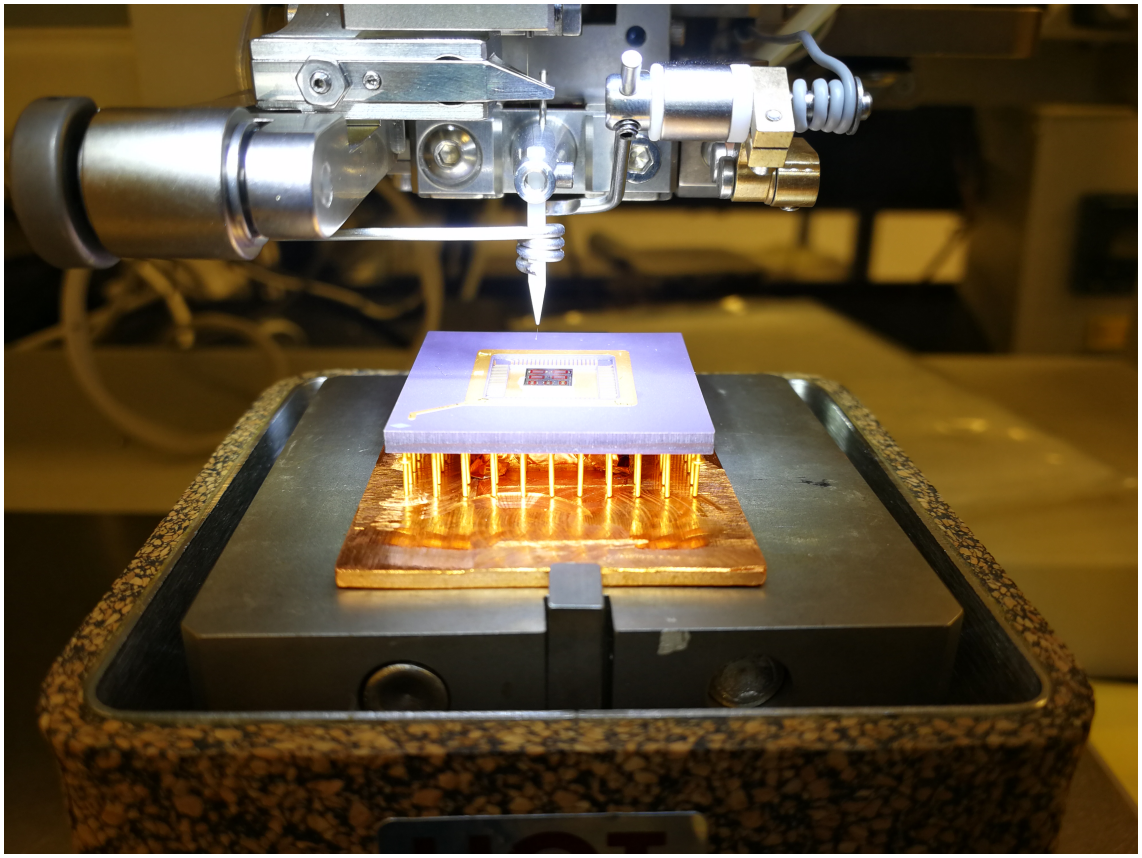


Figure 6.11: Chip inside of a PGA package situated in the wirebonder.

generate a complete chip layout from an adjacency matrix. If initial tests with these chips are promising, many more chip configurations could be laid out in a mostly-automated manner for further testing.

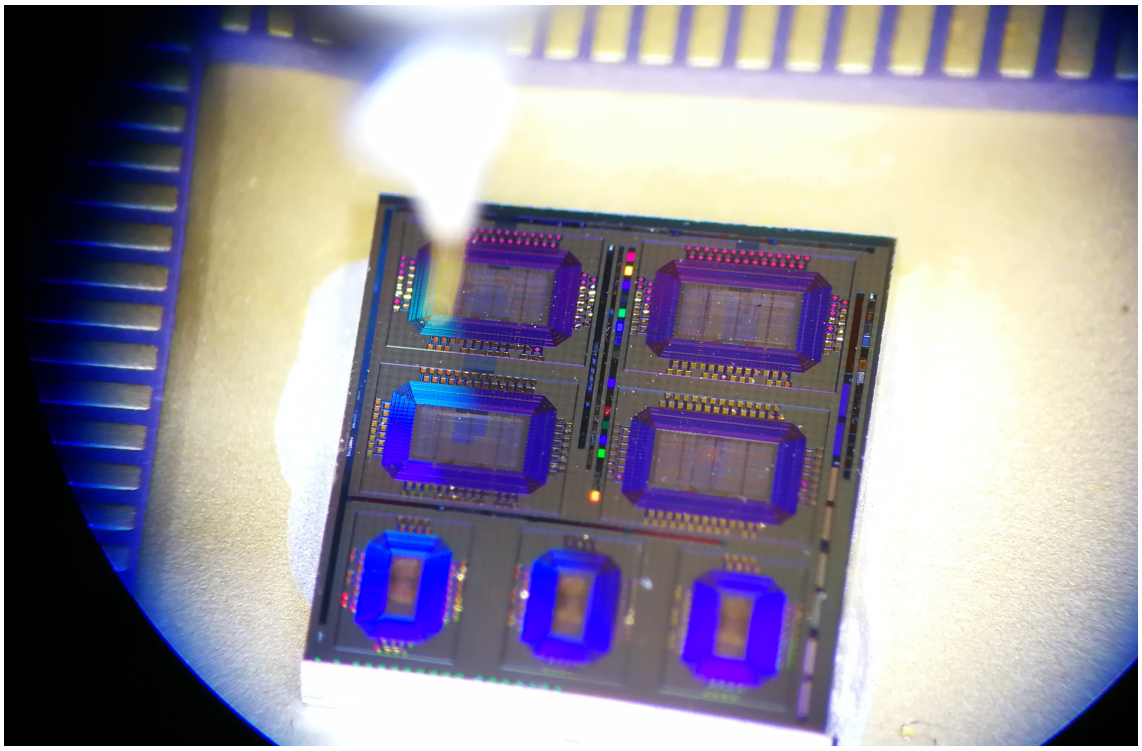


Figure 6.12: A view of the chip through the microscope of the wirebonder in the UMD Nano Center.

Chapter 7: Reservoir Computing Applied to Particle Accelerator Beam Trajectory Prediction

7.1 Overview

As part of this dissertation, we considered applications in which rapid time series processing hardware could be used[1]. While tasks such as realtime phoneme recognition may only require processing at the rate of human speech, other applications such as active control of dynamical systems need much higher prediction rates.

Particle accelerators are used in a wide variety of research applications as well as in industry and medicine. A challenge common to all accelerators is tracking and measuring beam parameters as the beam evolves from its source, through a long distance of beam pipe, to its final target. Diagnostics along this path are only present in a limited number of fixed locations. Machine learning techniques are gaining popularity to perform inference tasks such as "virtual diagnostics," or estimation of what instruments would read in locations where they cannot be placed [100].

In a particle accelerator, a particle bunch is confined in the beam pipe by a

[1]Portions of this chapter reproduced from Komkov et al. [27]

periodic focusing system with linear restoring forces. However, nonlinearities arise from a number of sources, including nonlinear magnets, nonidealities of physical components, beam-beam effects, and Coulomb repulsion between the particles (an effect known as space charge). These effects on the particle beam combine to create a dynamical system for which no complete explicit model exists. Prediction of the evolution of a particle beam can therefore be a challenge. In this chapter we use a software-based reservoir computer to predict the transverse beam evolution of an electron beam in the University of Maryland Electron Ring.

7.2 Accelerator Description and Simulation Setup

The **U**niversity of **M**aryland **E**lectron **R**ing (UMER) is a 10 keV storage ring designed for the study of physics of high-current electron beams. UMER consists of 18 nearly identical sections 32 cm in length, each containing the same arrangement of steering and focusing magnets as shown in figure 7.1. Each section consists of dipole magnets, which steer the beam, and quadrupole magnets, which confine the beam within the pipe. Diagnostics include imaging screens at regular locations around the ring. The electron bunch current can be varied from 150 μA to 80 mA, with larger currents corresponding to stronger nonlinear forces due to space charge.

While imaging screens placed in the beam's path are a destructive diagnostic, they provide a wealth of information about the beam's transverse characteristics. UMER has 11 phosphor screens which can be inserted into the ring to intercept the beam on its first turn. In 3 locations, an electrostatic deflector can kick the beam

into a screen located to the side, allowing any turn to be visualized. More details about UMER can be found in Kishek et al. [101] and Dovlatyan et al. [102].

The size of the electron beam is governed by the envelope equations:

$$x''(s) + k^2 x(s) - \frac{2K_{sp}}{[x(s) + y(s)]} - \frac{\epsilon_x^2}{x^3(s)} = 0 \tag{7.1}$$

$$y''(s) + k^2 y(s) - \frac{2K_{sp}}{[x(s) + y(s)]} - \frac{\epsilon_y^2}{y^3(s)} = 0 \tag{7.2}$$

where k is the linear restoring force provided by quadrupoles, K_{sp} is the beam perveance (space charge) parameter, and $\epsilon_{x,y}^2$ is the emittance, a pressure-like term [103]. These are all constants set by the initial conditions of the beam and the settings of magnets in the accelerator lattice.

The beam perveance, a dimensionless parameter proportional to the beam current, determines the amount of the nonlinear Coulomb repulsion force in the system, where $K_{sp} = 0$ is a purely linear system. For the simulations in this paper a 0.6 mA beam with $K_{sp} = 9.029 \times 10^{-6}$ is used.

The results here are from a physics-based simulation of UMER made in WARP, a 3D particle-in-cell code for modeling plasmas and high current particle beams [104]. Simulations are initialized with $10,000$ particles in a semi-Gaussian initial beam distribution, and are checked to converge with increasing particle number.

Images of the transverse particle distribution at 288 evenly spaced locations around the ring are extracted from WARP. While WARP tracks every particle indi-

vidually, the images are purposely downsampled to 15x15 pixels as shown in figure 7.1 (b) for the sake of the reservoir speed. Each pixel represents a physical size of about 400 microns. Pixel values are normalized to be between 0 and 1. Images are generated at a close enough spacing to resolve centroid and envelope oscillations around the ring.

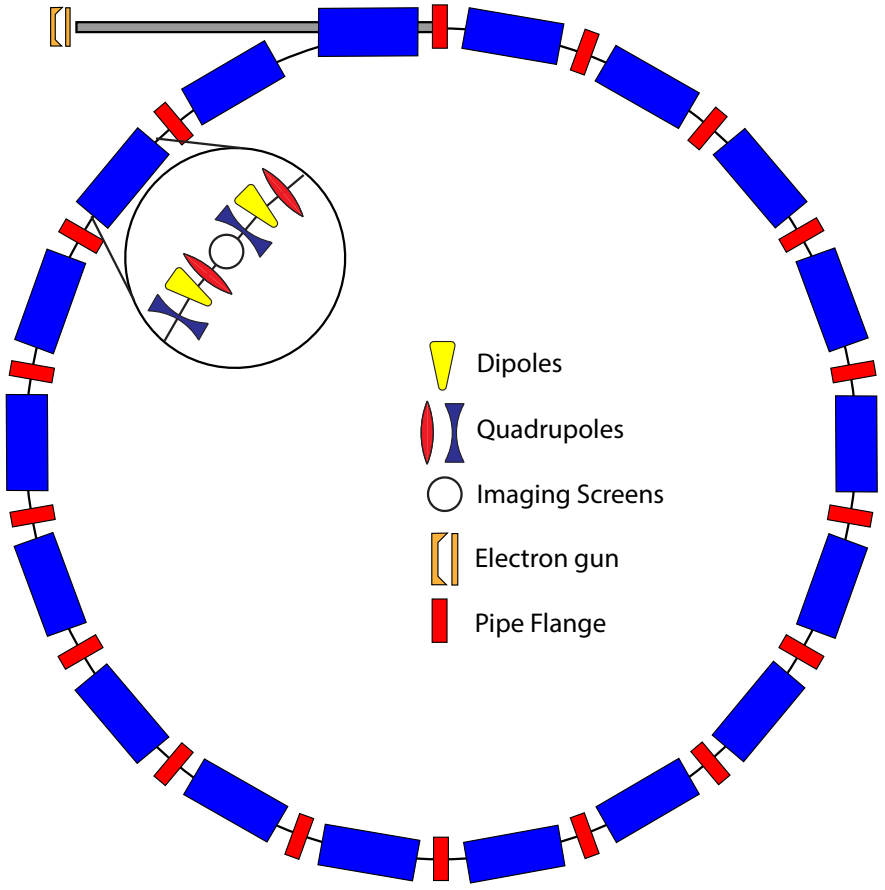


Figure 7.1: A diagram of the University of Maryland Electron Ring (UMER).

In some accelerators, only the first turn can be visualized with a phosphor screen. For that reason, we use only the first turn for training. A 4000-node reservoir in MATLAB is trained on the first 288 images, which is the full first turn around the ring, and generates predicted images with equal spacing in the next two turns,

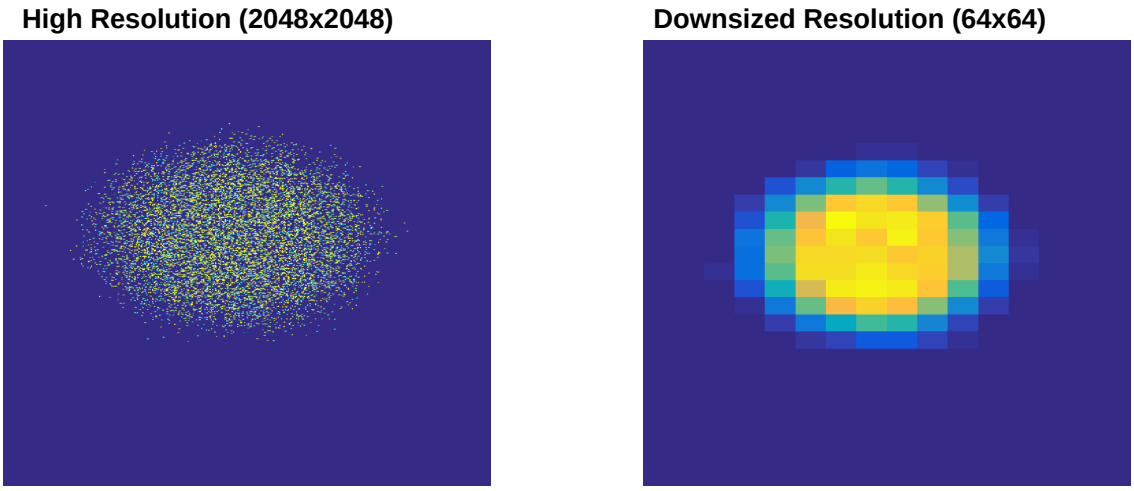


Figure 7.2: (left) A high resolution simulated beam profile from WARP image. (right) A downsampled image used for predictions.

as shown in figure 7.2. To prevent overfitting, 2% noise is added to training data. Results are generally observed to degrade with fewer nodes and not to improve with more nodes. The value of each pixel is presented to a subset of nodes in the reservoir determined by randomly initialized matrix W_{in}. For each image in the series, the output of the reservoir is trained to match the subsequent image to learn the dynamics of the particle beam. The reservoir is then allowed to run freely for prediction. The output layer is trained using least squares optimization.

The reservoir is sensitive to a variety of hyperparameters, including input strength (the magnitude of values in W_{in}), spectral radius (the size of the largest eigenvalue of A), α (the memory parameter), regularization parameter (a parameter that determines the amount of error allowed during training), reservoir size, and the particular choice of adjacency matrix A. Reservoir size must be large in comparison to dimensionality of input data, but there is no known way to optimize A other than through trial and error. After A is fixed, the other hyperparameters are manually

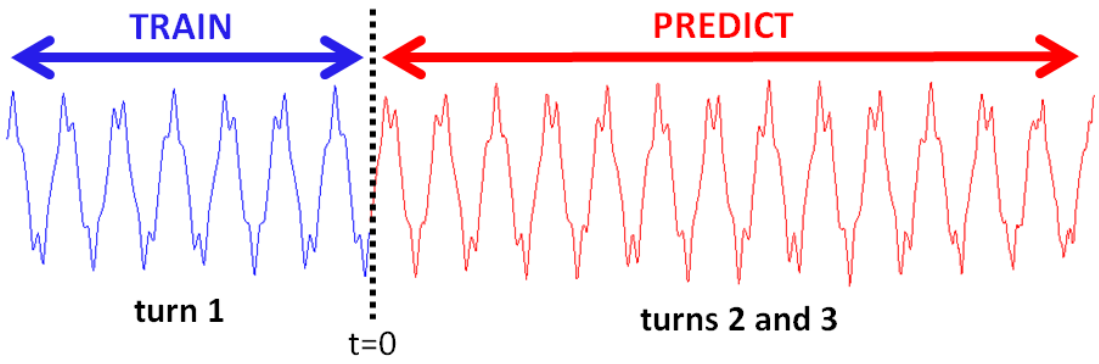


Figure 7.3: Training is done on the first turn, and the subsequent two turns are predicted.

tuned until a satisfactory result is obtained. The reservoir is generated and the result is computed in under a minute on a PC.

To flatten data from the image series, four parameters are extracted from the predicted images: the centroid motion and beam size in both vertical and horizontal planes. These are the first and second central moments of the images, $(\mu_{01}, \mu_{10}, \mu_{20}, \mu_{02})$, and are calculated using the following equation:

$$\mu_{pq} = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q I(x, y) \tag{7.3}$$

where $I(x, y)$ is the pixel intensity, x, y are the pixel locations, and p, q are the 2D moment orders.

Centroid motion evolves in a semi-periodic fashion dictated by the magnetic focusing lattice. The beam envelope, which evolves according to the envelope equations, is subject to substantial nonlinearities, and is aperiodic. Cross moments such as μ_{11} can appear if there are rotational offsets in magnets, which cause x and y motion to couple, but for these initial studies there is no skew in the magnets.

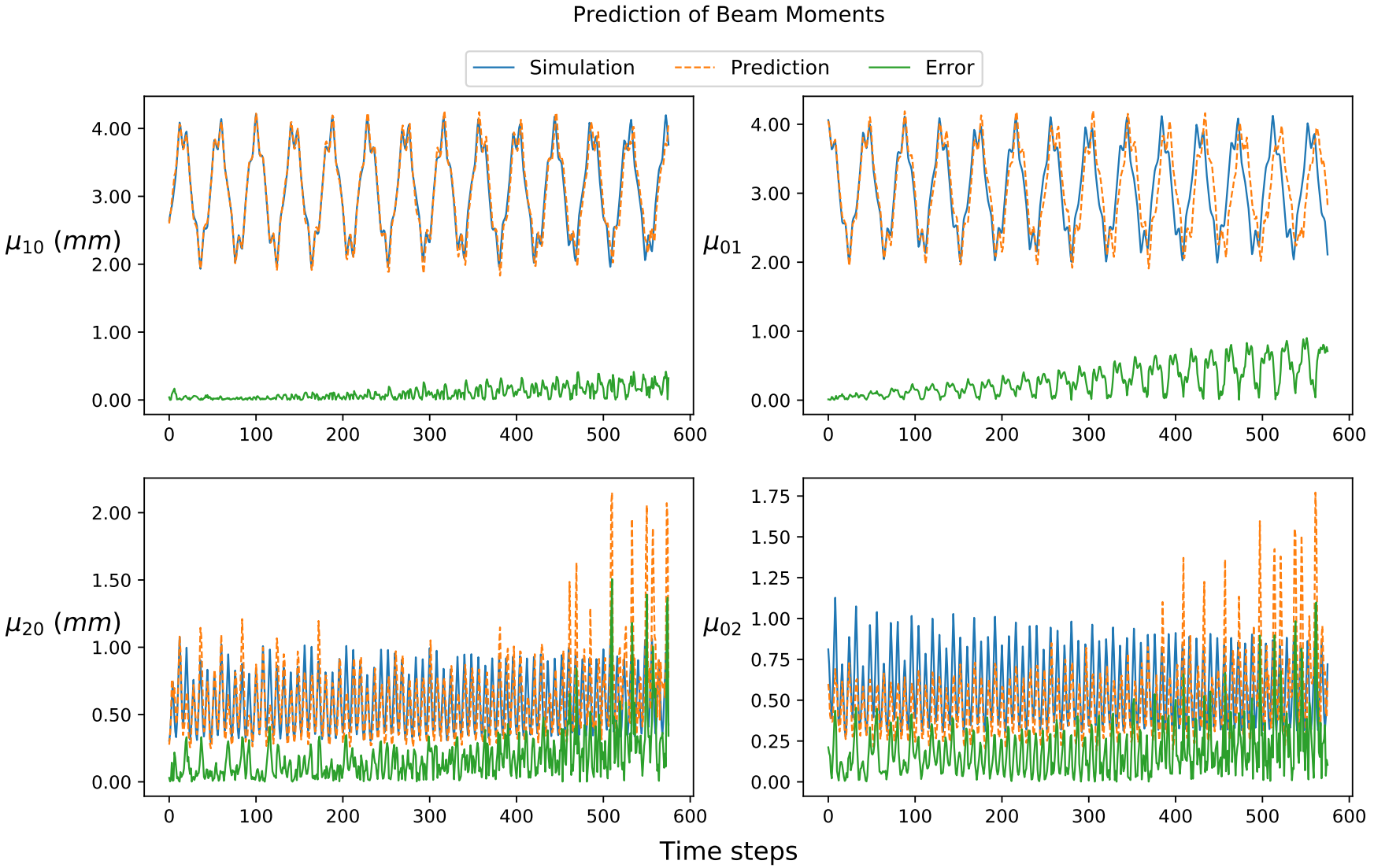


Figure 7.4: Plots of four beam parameters: horizontal and vertical centroid position (μ_{10} , μ_{01}) and beam size (μ_{20} , μ_{02}), from simulation and from reservoir prediction. The error is measured as the absolute difference between the simulation and prediction values.

Error	Turn 1	Turn 2
μ_{10}	1.6%	5.0%
μ_{01}	5.4%	14.3%
μ_{20}	21.1%	37.7%
μ_{02}	37.5%	45.0%

Table 7.1: Average errors calculated between moments from simulations and moments from reservoir predictions, in the first and second predicted turns.

7.3 Results

The results, summarized in figure 7.4, show good agreement between prediction and experiment, particularly in tracking the relatively regular centroid motion. The beam envelope is subject to more nonlinear forces, and the reservoir's error is relatively higher, yet the general character of the time series is captured, with the prediction closely matching the periodicity of the data from simulation. Error percentages for each turn are summarized in table 7.3. We plan to use measured data as the initial conditions in a reservoir computer trained with simulated data and compare the reservoir computer's forecast to real measurements at available beam imaging locations.

7.4 Conclusion

We have shown promising initial results for prediction of transverse electron beam parameters using a reservoir computer in MATLAB, allowing beam properties to be inferred in locations where they cannot be physically measured. Time series forecasting using reservoir computing could be utilized in active control scenarios,

in particle accelerators or other systems.

Chapter 8: Conclusion

8.1 Summary

Reservoir computing in hardware leverages the computational properties of dynamical systems at the edge of criticality for computation. A CMOS implementation is of interest because it has potential for seamless integration into modern electronics manufacturing. In this thesis, we have explored Boolean network circuits for reservoir computing using various hardware platforms, and we designed an ASIC that awaits testing.

Using an FPGA-based Boolean reservoir described in Chapter 4, we demonstrated RF waveform classification accuracy competitive with a state of the art convolutional neural network with an order of magnitude fewer trainable parameters. We extensively probed the parameter space of network configurations using a control word to set node functions, and developed dataset-agnostic methods for rapidly discarding unsuitable network configurations. This work is based upon designs in [21] and published in [23].

In Chapter 5, we showed the design of a printed circuit board reservoir made with discrete logic chips. While compared to the 2048 outputs of the FPGA there were many fewer channels that could be simultaneously measured, we were able

to sample the analog waveforms with high density and measure a machine learning improvement in a waveform classification task. Across Boolean network implementations in different platforms, we found consistent measurements of network behavior including self-activity and transient time as a function of Boolean sensitivity. This experiment awaits publication in [24].

The design of a reservoir computing ASIC is shown in Chapter 6, based on the designs in patent application [25]. An efficient workflow for rapid chip layout using Cadence tools was developed, and seven different chip designs await post-processing and testing.

Finally, we showed a potential application of reservoir computing in doing rapid time series prediction using data from the University of Maryland Electron Ring. Training on time series from a simulation of a high-current electron beam subject to nonlinear forces, we predicted the shape and trajectory of this beam as it traveled around the storage ring, an experiment published in [27]. While this was a software study, trajectory prediction is an example of an application that could be accelerated in hardware with a Boolean network co-processor.

8.2 Future Vision

A free-running, unclocked Boolean circuit reservoir circuit can, within a time scale on the order of a few transition times of the logic gates, perform massively parallel recurrent neural network computations that would take a conventional CPU or GPU substantially longer. There is potential to accelerate time series regression

or classification tasks considerably.

However, an unclocked Boolean circuit reservoir would not be a general-purpose neural network accelerator, nor would it be able to implement pre-trained models. Due to the high sensitivity to device variations of these edge-of-chaos circuits, each circuit would have to be separately trained.

On the other hand, use cases where rapid re-training is necessary are precisely the niche that unclocked Boolean circuits fulfill. These applications include rapid time series prediction, in which the reservoir computer must quickly train on the first portion of a time series, and then perform a free-running prediction. Another application is time series classification, including in rapidly changing environments. Finally, the reservoir's high speed and parallelism can also be used for static applications, although this does not fully take advantage of their temporal capabilities.

A specific example of a potential use case is the active control of tokamaks and particle accelerators, both of which which need to abort their operation in advance of destructive instabilities to avoid machine damage.

While we have demonstrated a few variants of unclocked Boolean circuits for reservoir computing, reservoir optimization is still an active area of research—both in hardware and in software. In the Boolean reservoir case, quantifying how much information can be extracted from pulse timing versus analog voltages is open question. Overall, the work done in this thesis is an encouraging step towards obtaining more processing power from limited chip resources in a reservoir co-processor.

Appendix A: ASIC Details

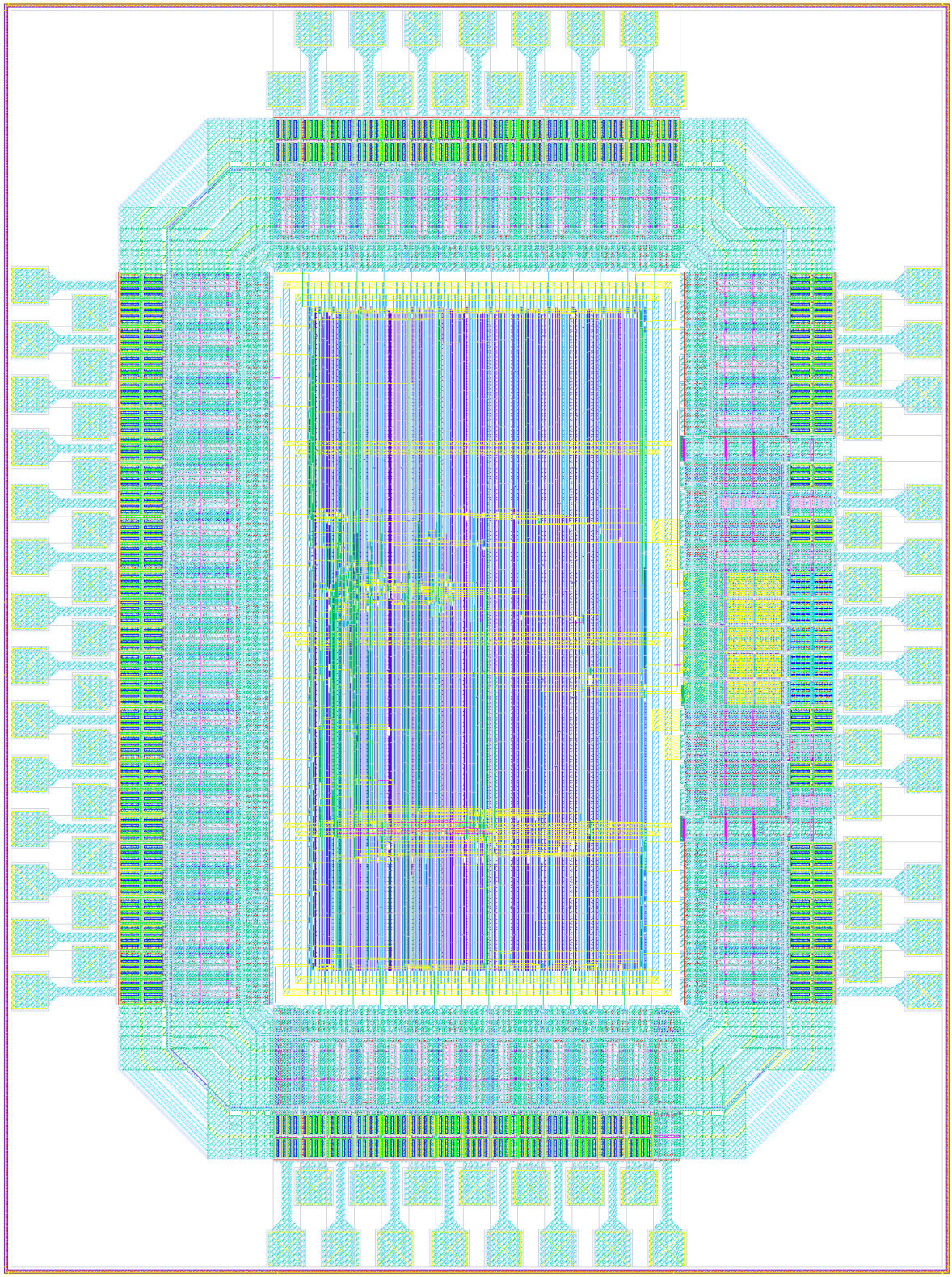


Figure A.1: Chiplet 1, Andy.

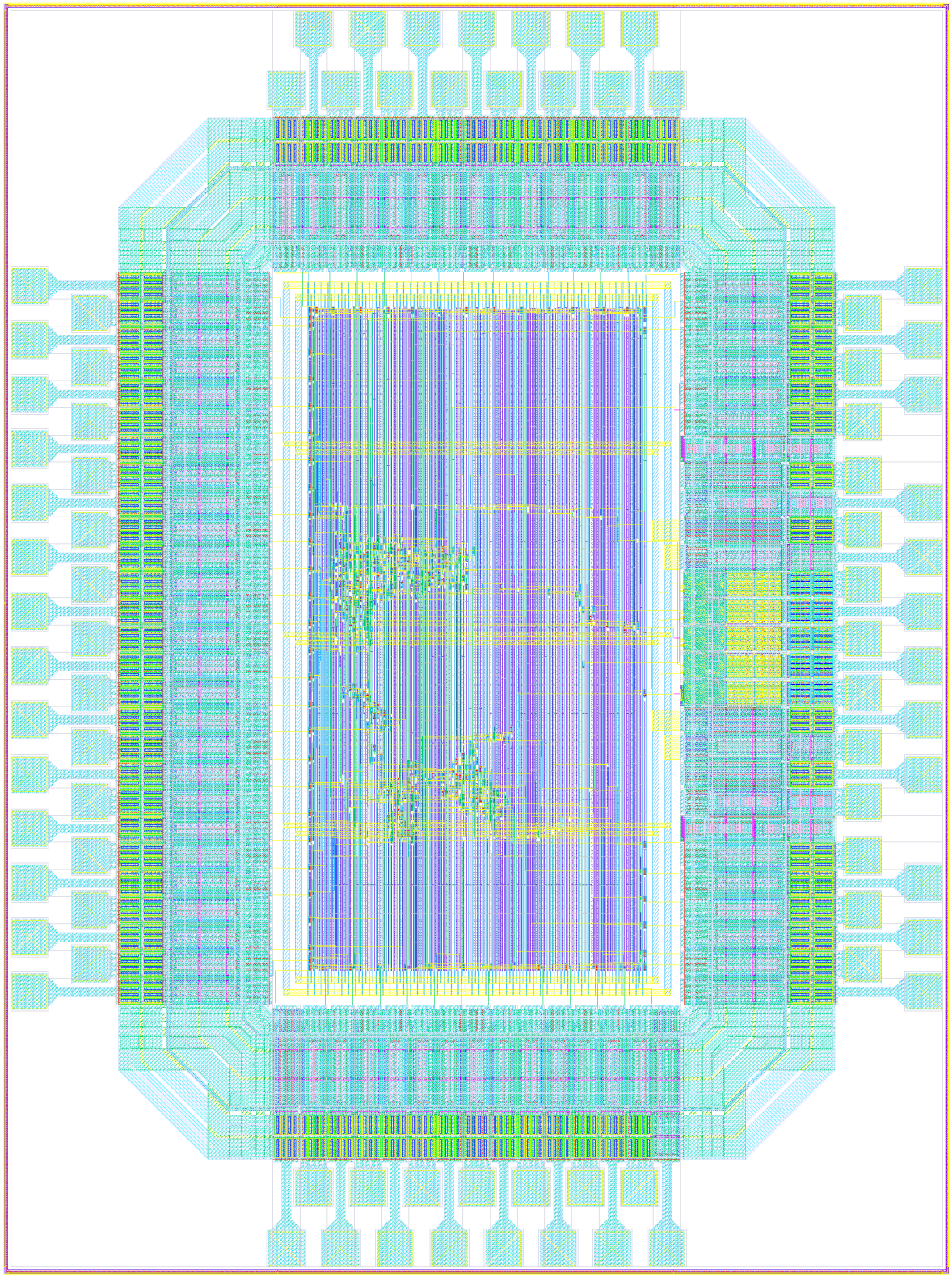


Figure A.2: Chiplet 2, Billy.

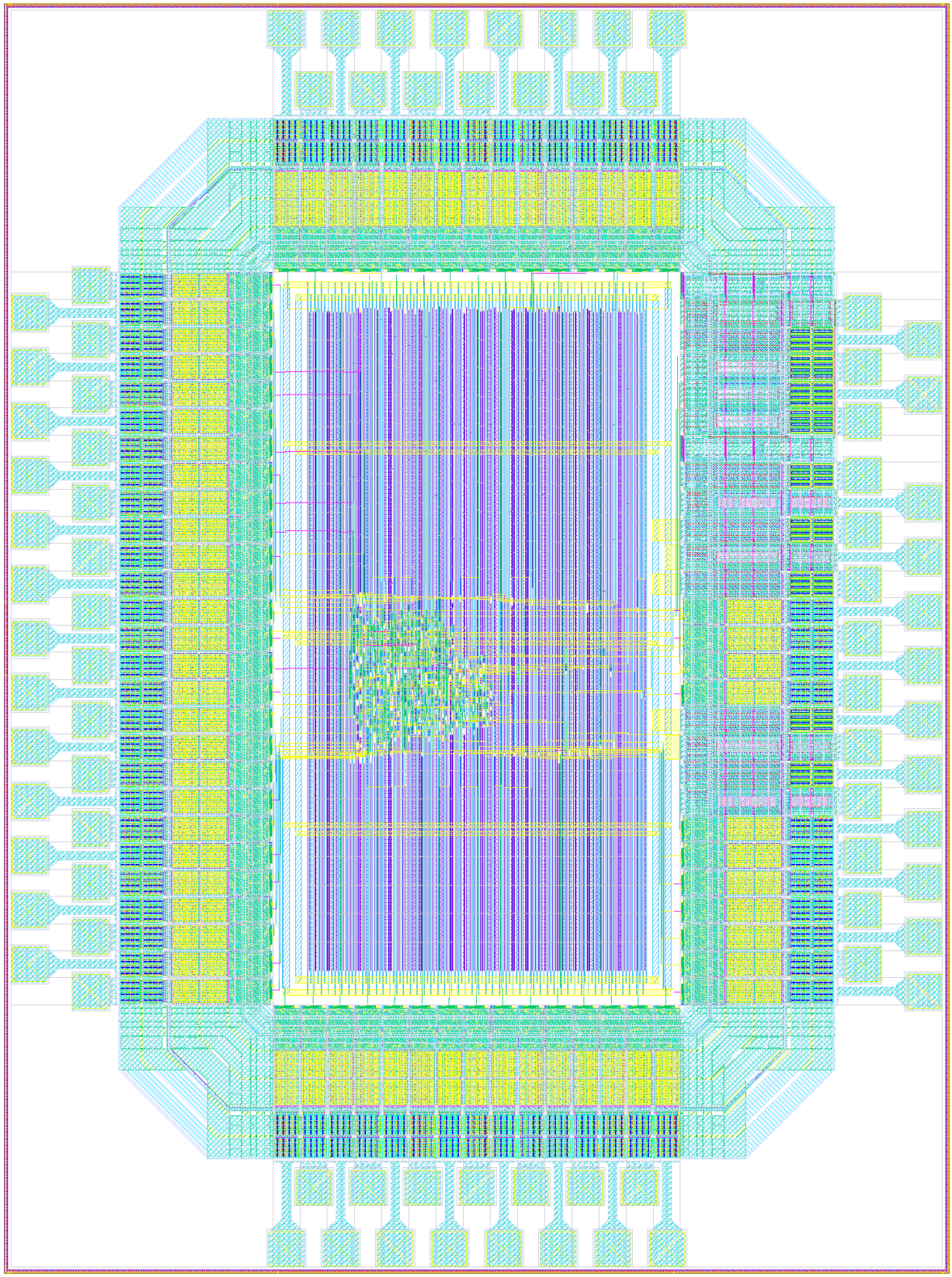


Figure A.3: Chiplet 3, Charlie.

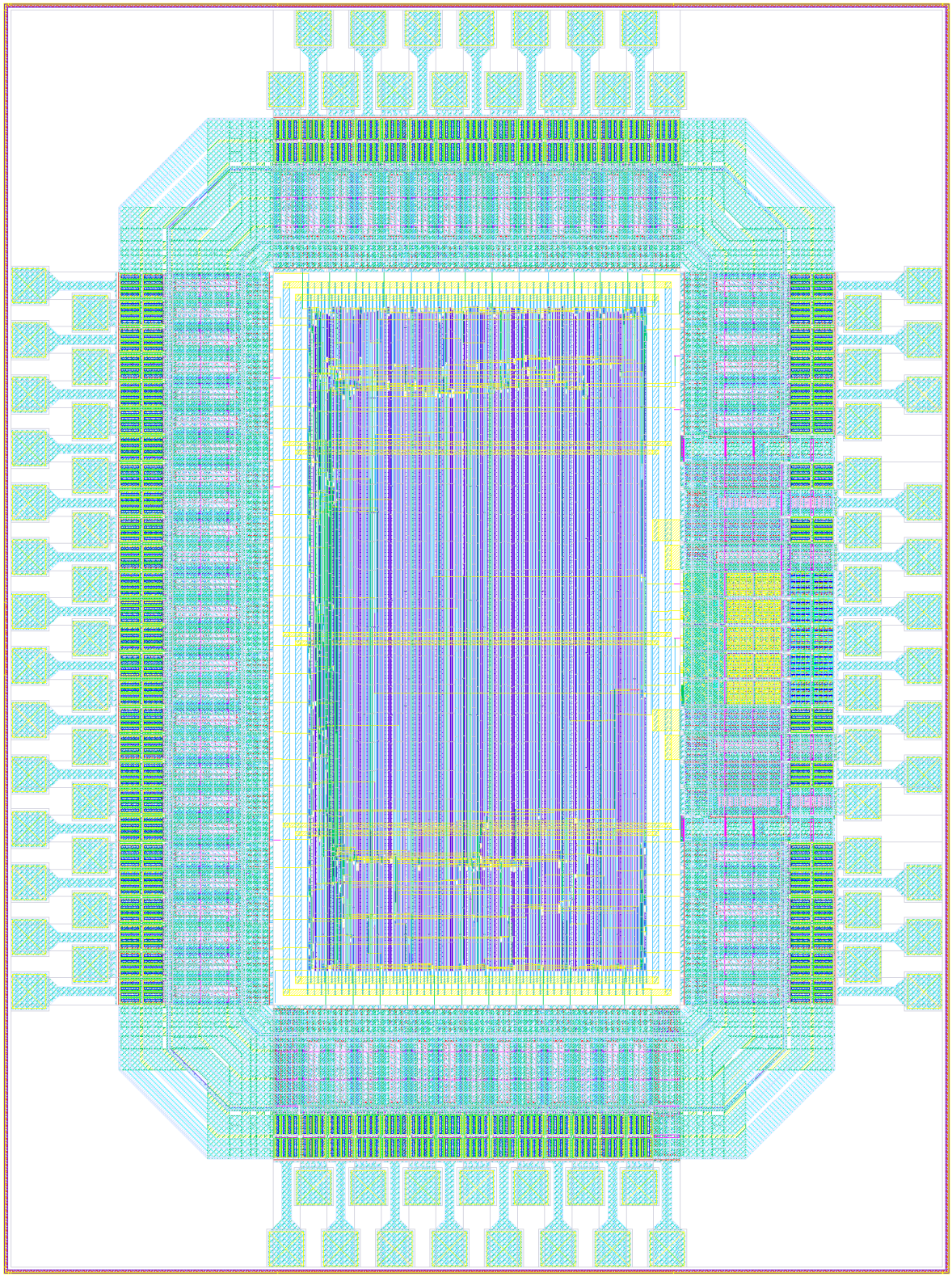


Figure A.4: Chiplet 4, Denny.

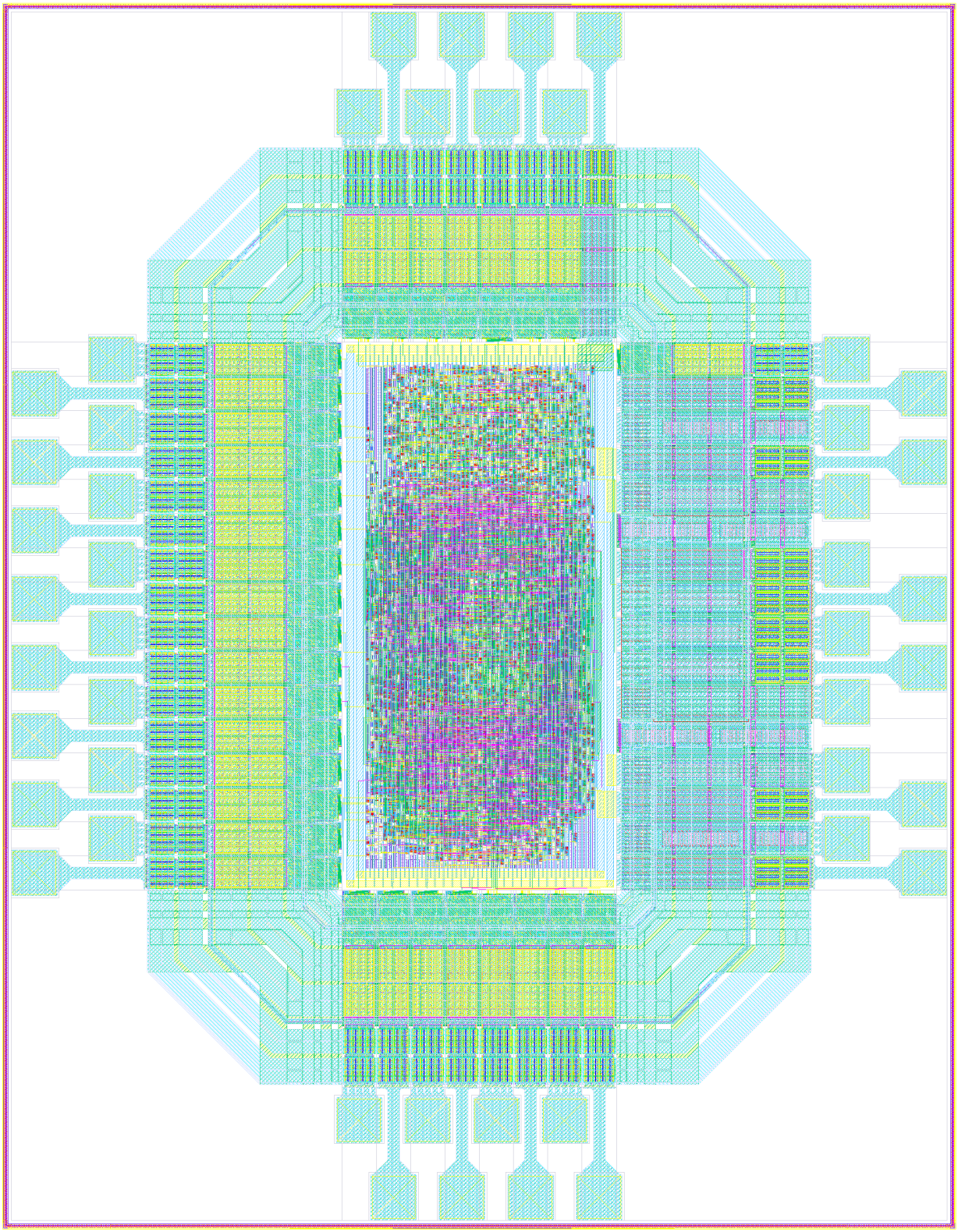


Figure A.5: Chiplet 5, Ernie.

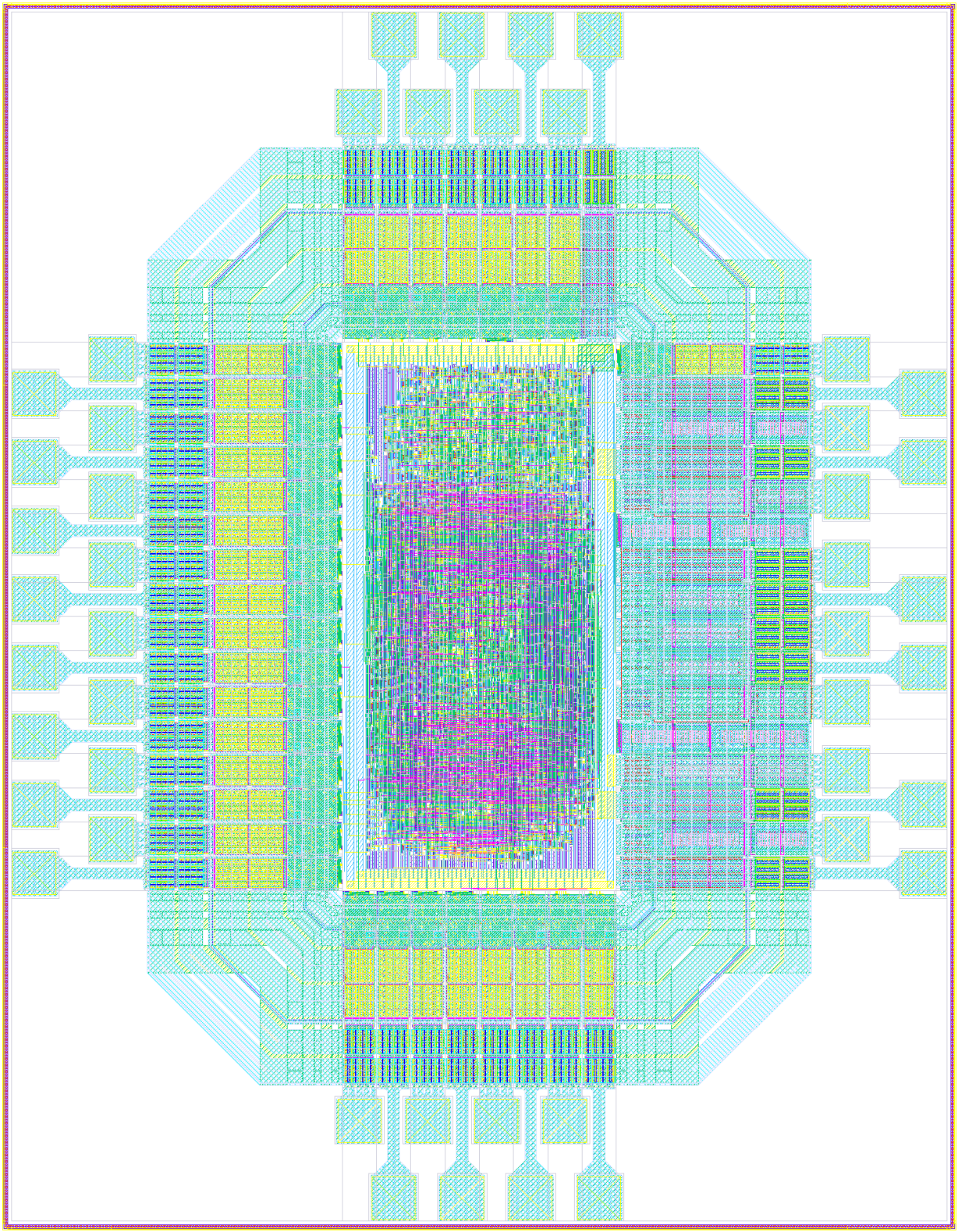


Figure A.6: Chiplet 6, Freddy.

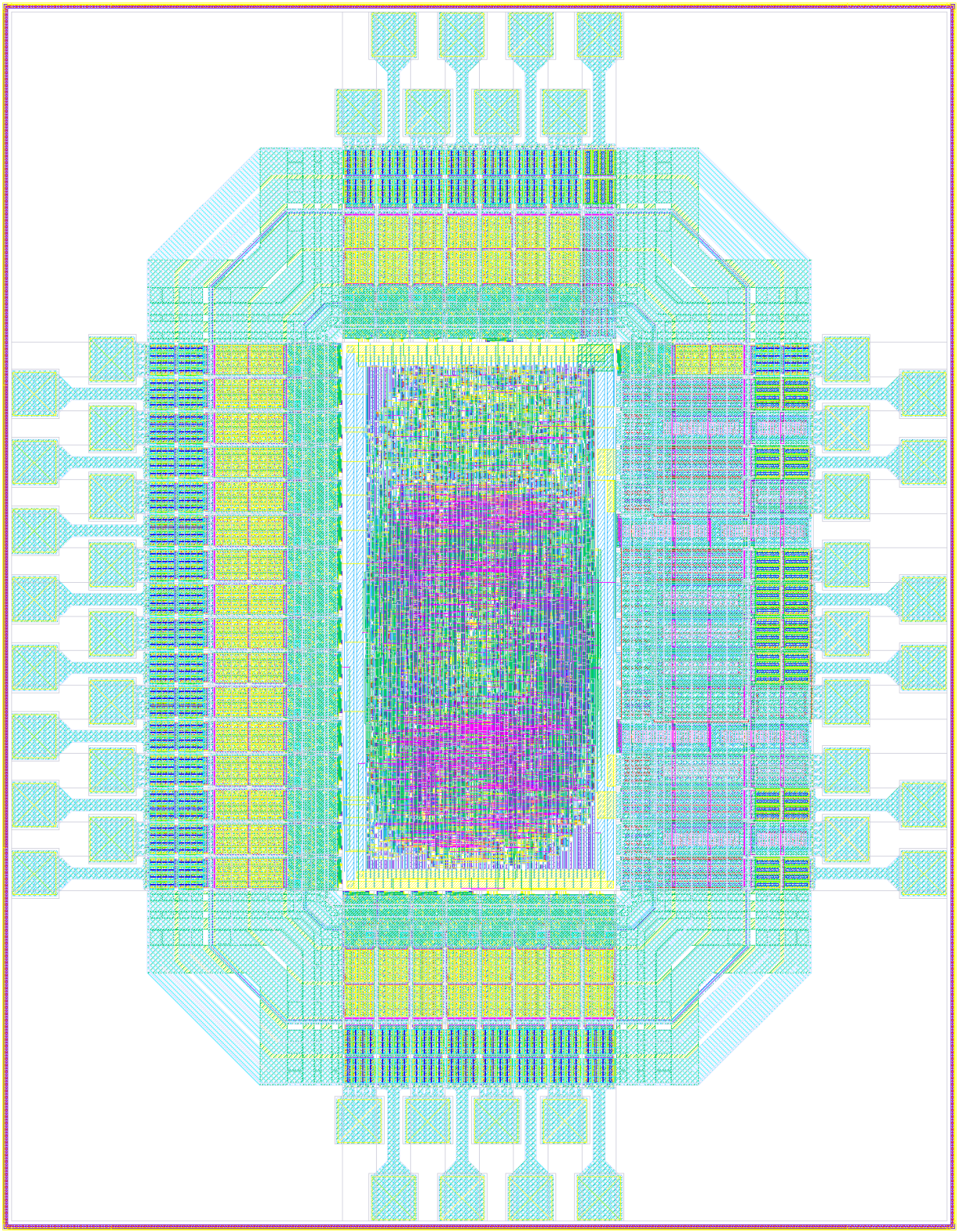


Figure A.7: Chiplet 7, Gary.

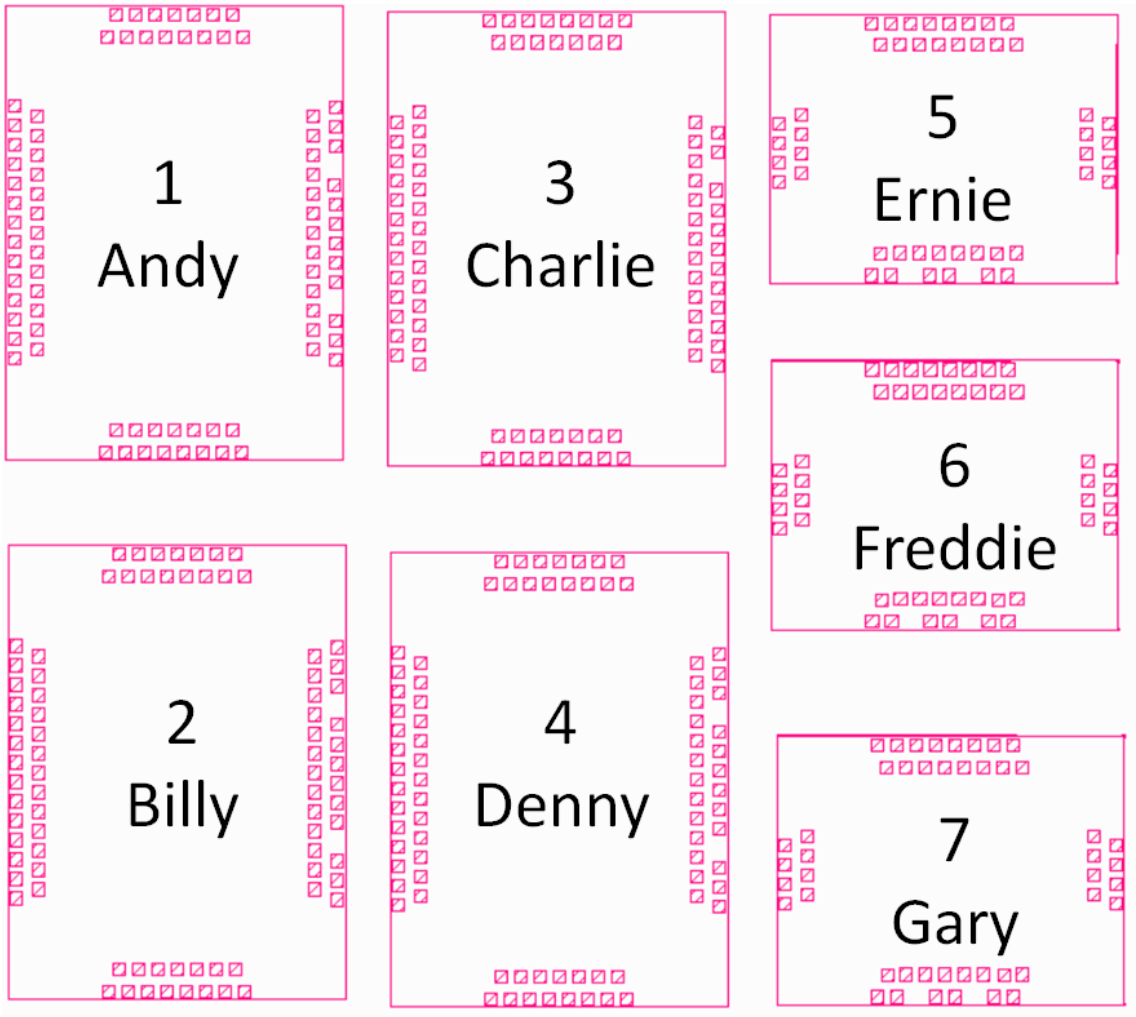


Figure A.8: Overall placement of the chiplets on the wafer.

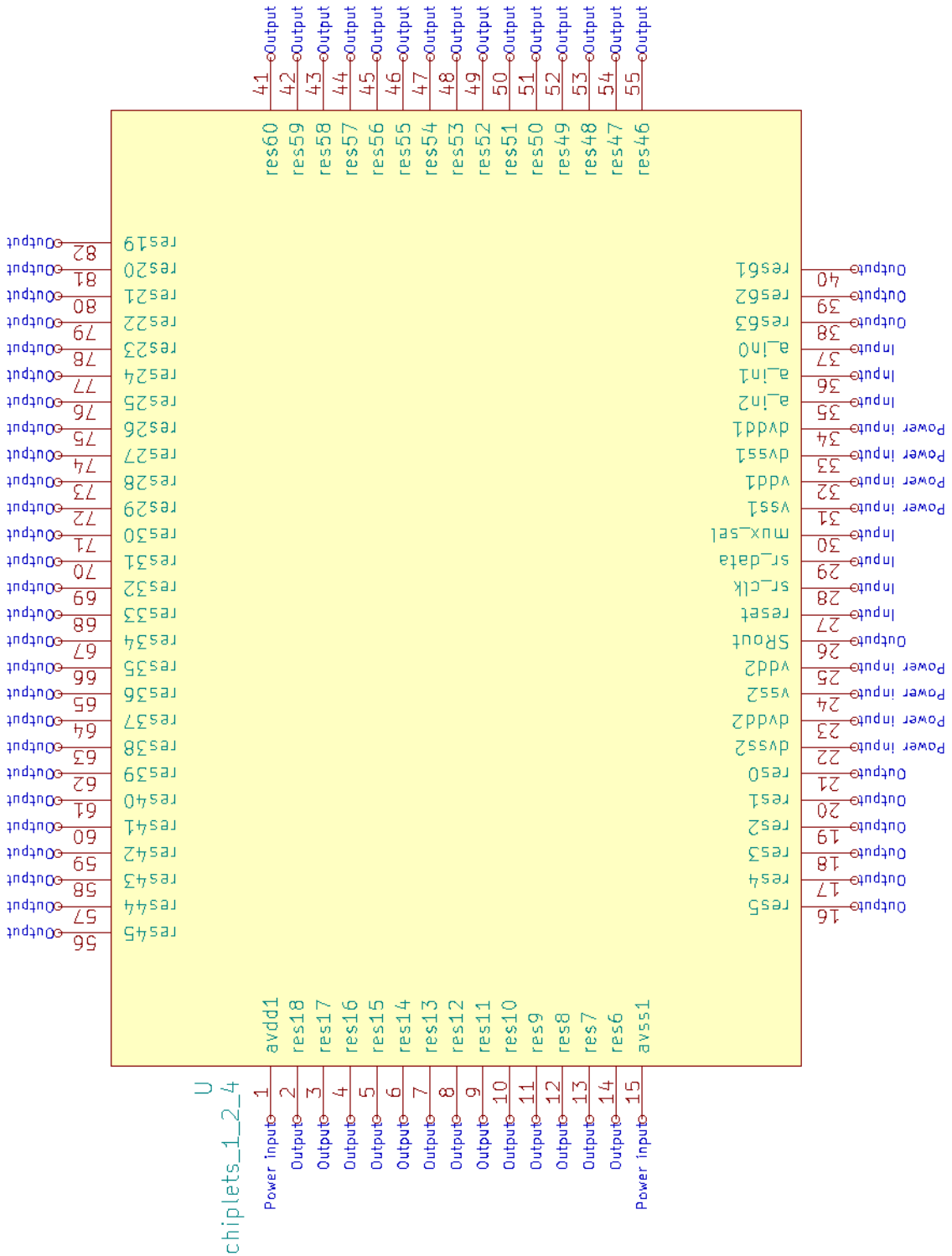


Figure A.9: Pinout of chiplets 1, 2 and 4.

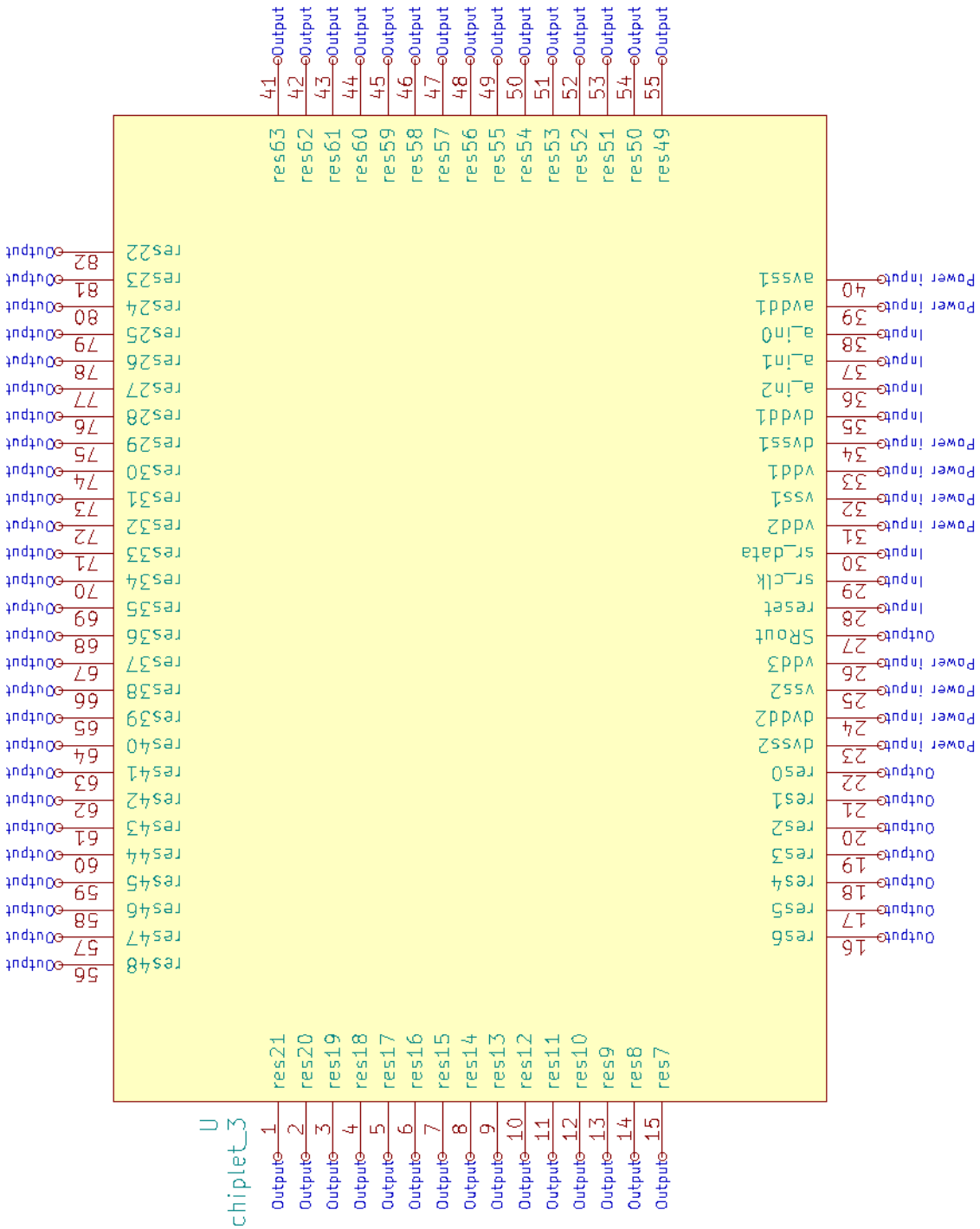


Figure A.10: Pinout of chiplet 3.

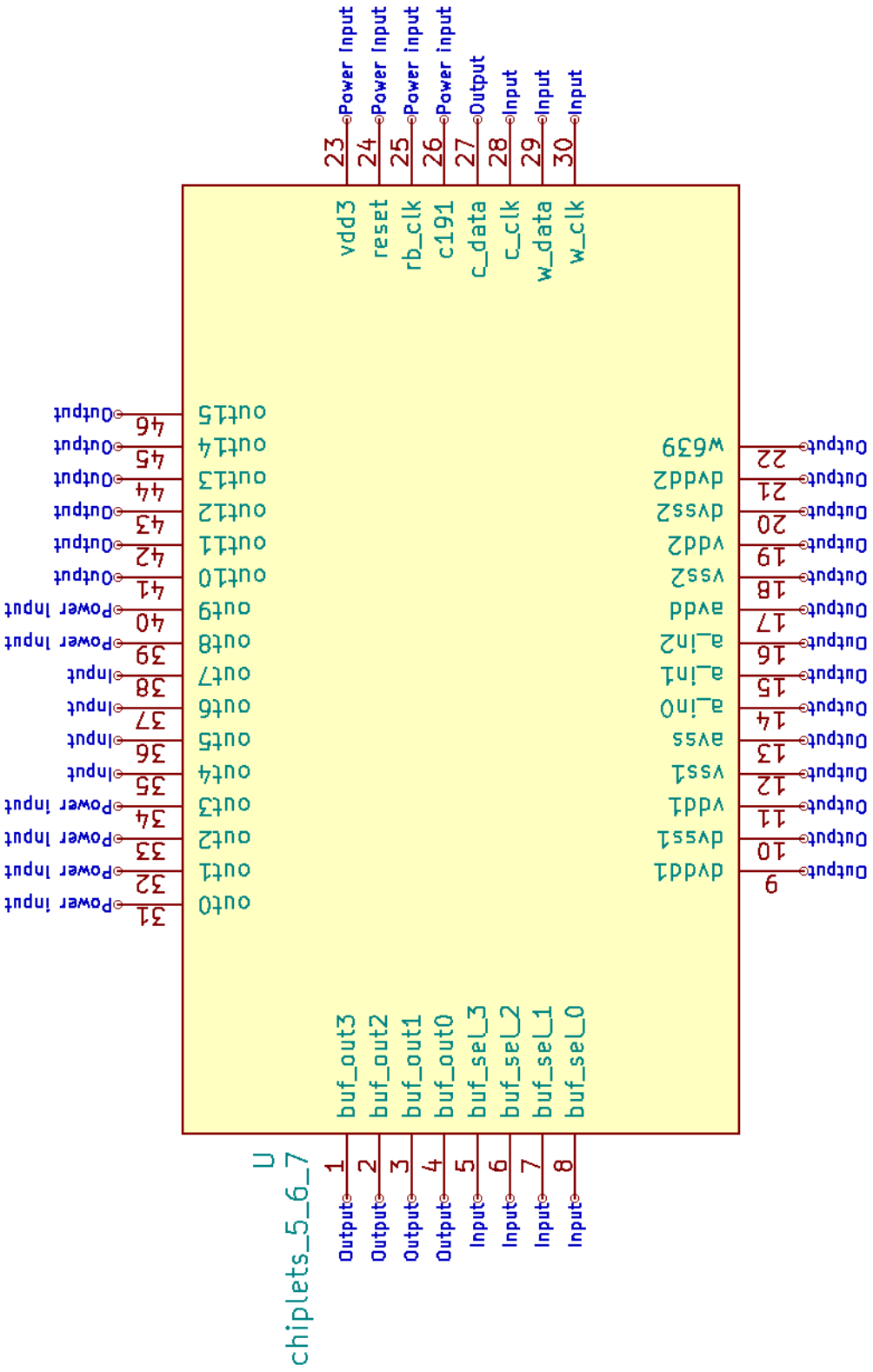


Figure A.11: Pinout of chiplets 5, 6, and 7.

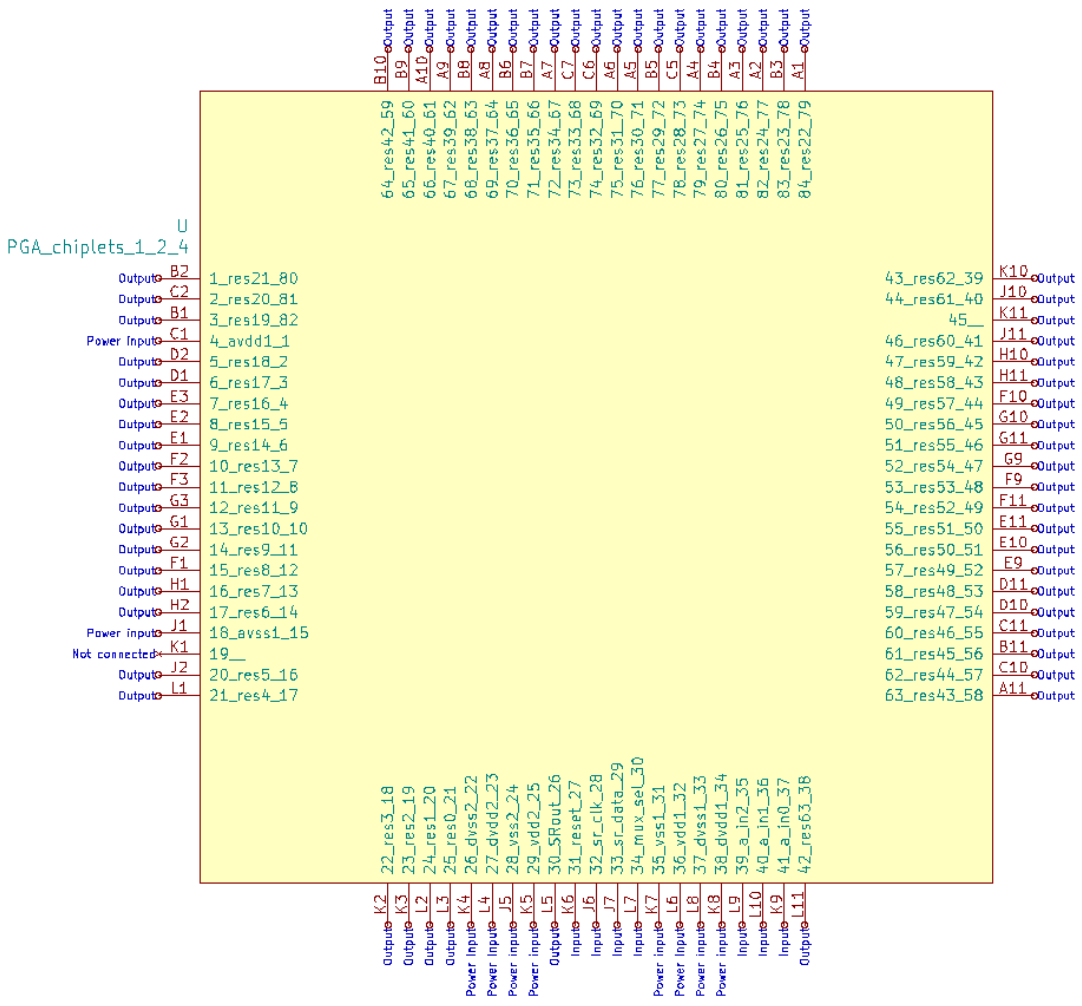


Figure A.12: PGA package internal connections for chiplets 1, 2, and 4

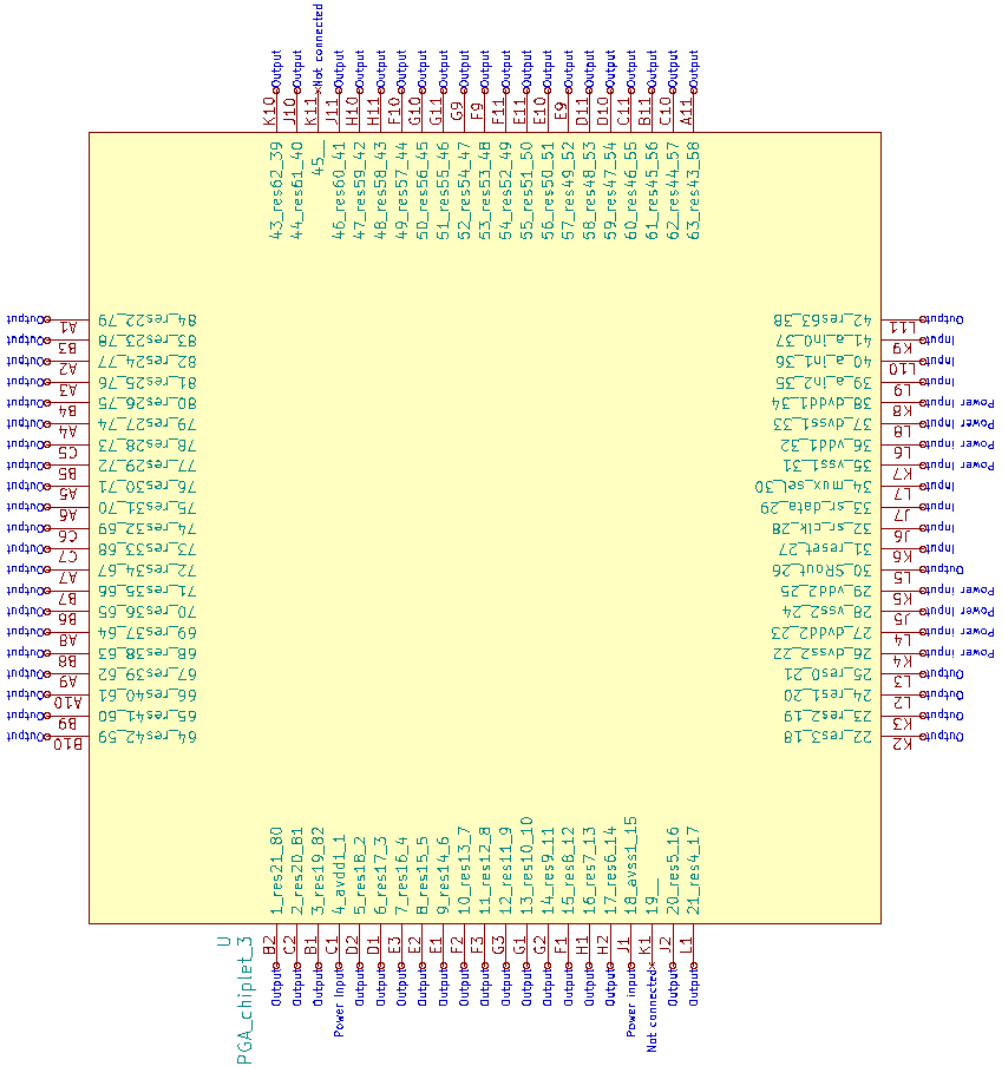


Figure A.13: PGA package internal connections for chiplet 3.

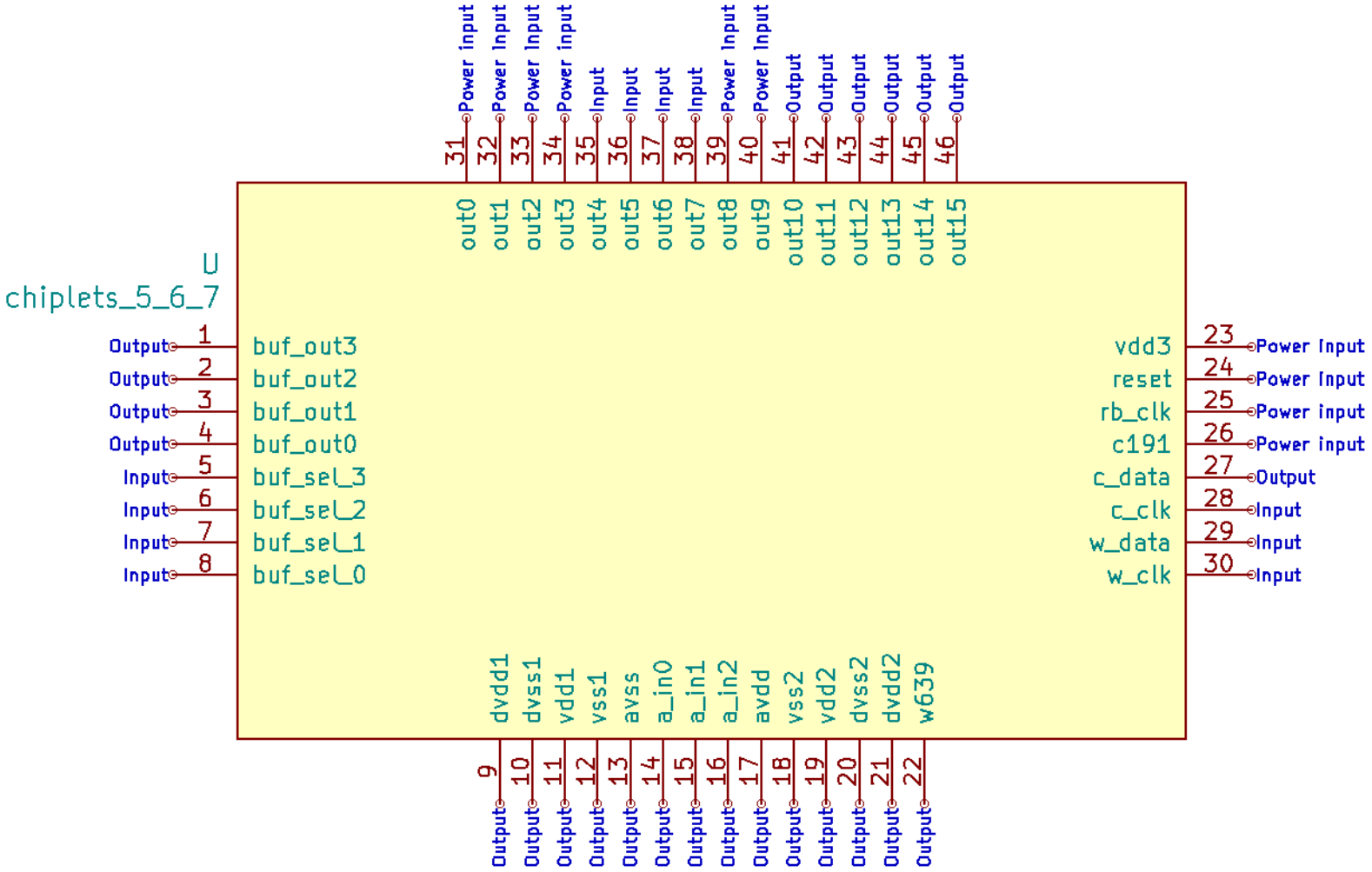


Figure A.14: PGA package internal connections for chiplets 5, 6 and 7

Bibliography

[1] ITU, "Measuring digital development: Facts and figures," tech. rep., International Telecommunications Union, 2019.

[2] ITU, "Measuring Digital Development: ICT Price Trends," tech. rep., International Telecommunications Union, 2019.

[3] G. E. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, pp. 82–85, apr 1998.

[4] T. Cross, "Technology Quarterly: After Moore's law," *The Economist*, mar 2016.

[5] M. Horowitz, "Computing's Energy Problem (and what we can do about it)," in *ISSCC*, pp. 10–14, IEEE, 2014.

[6] W. Arden, M. Brillouët, P. Cogez, M. Graef, B. Huizing, and R. Mahnkopf, ""More-than-Moore" White Paper," tech. rep., International Technology Roadmap for Semiconductors (ITRS), 2010.

[7] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, "There's plenty of room at the top: What will drive computer performance after Moore's law?," *Science*, vol. 368, jun 2020.

[8] "Beyond CMOS," tech. rep., International Technology Roadmap for Semiconductors (ITRS), 2015.

[9] R. O. Topaloglu and H. S. Wong, eds., *Beyond-CMOS technologies for next generation computer design*. Springer, 2019.

[10] A. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM Journal of Research and Development*, 1959.

[11] Y. Jia, R. J. Weiss, F. Biadsy, W. Macherey, M. Johnson, Z. Chen, and Y. Wu, "Direct speech-to-speech translation with a sequence-to-sequence model," *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, pp. 1123–1127, 2019.

[12] D. Silver, A. Huang, and C. J. Maddison, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, 2016.

[13] O. Vinyals, I. Babuschkin, and W. M. Czarnecki, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[14] M. M. Morando, Q. Tian, L. T. Truong, and H. L. Vu, "Studying the Safety Impact of Autonomous Vehicles Using Simulation-Based Surrogate Safety Measures," *Journal of Advanced Transportation*, 2018.

[15] J. Dean, "The Deep Learning Revolution and Its Implications for Computer Architecture and Chip Design," *Digest of Technical Papers - IEEE International Solid-State Circuits Conference*, pp. 8–14, 2020.

[16] N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso, "The Computational Limits of Deep Learning," tech. rep., 2020.

[17] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European conference on computer vision*, pp. 525–542, Springer, 2016.

[18] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1," in *Neural Information Processing Systems*, 2016.

[19] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[20] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), pp. 1–12, Association for Computing Machinery, 2017.

[21] H. Komkov, A. Restelli, B. Hunt, L. Shaughnessy, I. Shani, and D. Lathrop, "The recurrent processing unit: Hardware for high speed machine learning," *arXiv preprint arXiv:1912.07363*, 2019.

[22] I. Shani, L. Shaughnessy, J. Rzasa, A. Restelli, B. R. Hunt, H. Komkov, and D. P. Lathrop, "Dynamics of analog logic-gate networks for machine learning," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 29, no. 12, p. 123130, 2019.

[23] H. Komkov, L. Pocher, A. Restelli, D. P. Lathrop, and B. Hunt, "RF Signal Classification using Boolean Reservoir Computing on an FPGA," in *IJCNN Int Jt Conf Neural Network*, 2021.

[24] H. Komkov, L. Pocher, A. Restelli, B. Hunt, and D. P. Lathrop, "Reservoir Computing Using Networks of CMOS Logic Gates," in *International Conference on Neuromorphic Systems*, 2021.

[25] A. Restelli, D. Lathrop, and H. Komkov, "United States Patent Application 17/208,399: Variable Sensitivity Node," April 2021.

[26] D. Lathrop, I. Shani, P. Megson, A. Restelli, and A. R. Mautino, "United States Patent WO2018213399A1: Integrated circuit designs for reservoir computing and machine learning," 2018.

[27] H. Komkov, A. Perevalov, L. Dovlatyan, D. D. P. Lathrop, A. Perevalov, and D. D. P. Lathrop, "Reservoir Computing for Prediction of Beam Evolution in Particle Accelerators," in *NeurIPS ML4PS Workshop*, 2019.

[28] D. E. Rumelhart, G. E. Hinton, R. J. Williams, and Rumelhart, "Learning Representations by Back-Propagating Errors," in *Neurocomputing: Foundations of Research*, pp. 696–699, Cambridge, MA, USA: MIT Press, 1988.

[29] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[30] H. Jaeger, "The "echo state" approach to analysing and training recurrent neural networks-with an Erratum note," *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, vol. 148, no. 34, p. 13, 2001.

[31] W. Maass, T. Natschläger, and H. Markram, "Real-time computing without stable states: A new framework for neural computation based on perturbations," *Neural Computation*, vol. 14, no. 11, pp. 2531–2560, 2002.

[32] J. Jiang and Y.-C. Lai, "Model-free prediction of spatiotemporal dynamical systems with recurrent neural networks: Role of network spectral radius," *Phys. Rev. Research*, vol. 1, p. 33056, oct 2019.

[33] H. Jaeger, "Short term memory in echo state networks," *GMD Report 152*, no. December, p. 60, 2002.

[34] H. Jaeger, "Long Short-Term Memory in Echo State Networks: Details of a Simulation Study," tech. rep., School of Engineering and Science, Jacobs University Bremen, 2012.

[35] A. Goudarzi, M. R. Lakin, and D. Stefanovic, "DNA Reservoir Computing: A Novel Molecular Computing Approach," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8141 LNCS, no. 1, pp. 76–89, 2013.

[36] W. Maass, T. Natschläger, and H. Markram, "Fading memory and kernel properties of generic cortical microcircuit models," *Journal of Physiology Paris*, vol. 98, no. 4-6 SPEC. ISS., pp. 315–330, 2004.

[37] W. Maass, R. Legenstein, and N. Bertschinger, "Methods for estimating the computational power and generalization capability of neural microcircuits," *Advances in Neural Information Processing Systems*, 2005.

[38] G. Cybenko, "Mathematics of Control, Signals, and Systems Approximation by Superpositions of a Sigmoidal Function*," *Math. Control Signals Systems*, vol. 2, pp. 303–314, 1989.

[39] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, no. 2, pp. 251–257, 1991.

[40] L. Grigoryeva and J.-p. Ortega, "Echo State Networks are Universal," *Neural Networks*, vol. 108, pp. 495–508, 2018.

[41] C. Gallicchio, A. Micheli, and L. Pedrelli, "Comparison between DeepESNs and gated RNNs on multivariate time-series prediction," in *27th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, no. April, pp. 619–624, 2019.

[42] P. R. Vlachas, J. Pathak, B. R. Hunt, T. P. Sapsis, M. Girvan, E. Ott, and P. Koumoutsakos, "Backpropagation algorithms and Reservoir Computing in Recurrent Neural Networks for the forecasting of complex spatiotemporal dynamics," *Neural Networks*, vol. 126, pp. 191–217, 2020.

[43] J. Pathak, B. Hunt, M. Girvan, Z. Lu, and E. Ott, "Model-Free Prediction of Large Spatiotemporally Chaotic Systems from Data: A Reservoir Computing Approach," *Physical Review Letters*, vol. 120, no. 2, 2018.

[44] J. Pathak, Z. Lu, B. R. Hunt, M. Girvan, and E. Ott, "Using machine learning to replicate chaotic attractors and calculate Lyapunov exponents from data," *Chaos*, vol. 27, no. 12, 2017.

[45] S. Krishnagopal, M. Girvan, E. Ott, and B. R. Hunt, "Separation of chaotic signals by reservoir computing," *Chaos*, vol. 30, no. 2, p. 23123, 2020.

[46] H. Haas and H. Jaeger, "Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication," *Science*, vol. 304, no. 5667, pp. 78–80, 2004.

[47] P. Buteneers, B. Schrauwen, D. Verstraeten, and D. Stroobandt, "Epileptic seizure detection using Reservoir Computing," *Proceedings of the 19th Annual Workshop on Circuits, Systems and Signal Processing*, pp. 1–4 (CD–ROM), 2008.

[48] P. Antonik, M. Hermans, M. Haelterman, and S. Massar, "Random Pattern and Frequency Generation Using a Photonic Reservoir Computer with Output Feedback," *Neural Processing Letters*, vol. 47, no. 3, pp. 1041–1054, 2018.

[49] F. Triefenbach, A. Jalalvand, B. Schrauwen, and J.-P. Martens, "Phoneme Recognition with Large Hierarchical Reservoirs," in *Advances in neural information processing systems*, pp. 2307 – 2317, 2010.

[50] L. Larger, A. Baylón-Fuentes, R. Martinenghi, V. S. Udaltsov, Y. K. Chembo, and M. Jacquot, "High-speed photonic reservoir computing using a time-delay-based architecture: Million words per second classification," *Physical Review X*, vol. 7, no. 1, 2017.

[51] D. M. Canaday, *Modeling and Control of Dynamical Systems with Reservoir Computing*. Doctoral thesis, Ohio State University, 2019.

[52] A. Vandesompele, G. Urbain, F. Wyffels, and J. Dambre, "Populations of spiking neurons for reservoir computing: Closed loop control of a compliant quadruped," *Cognitive Systems Research*, vol. 58, pp. 317–323, dec 2019.

[53] H. Soh and Y. Demiris, "Iterative temporal learning and prediction with the sparse online echo state gaussian process," *Proceedings of the International Joint Conference on Neural Networks*, pp. 10–15, 2012.

[54] L. Appeltant, M. C. Soriano, G. Van der Sande, J. Danckaert, S. Massar, *et al.*, "Information processing using a single dynamical node as complex system," *Nature Communications*, vol. 2, p. 468, Sept. 2011.

[55] J. D. Hart, L. Larger, T. E. Murphy, and R. Roy, "Delayed dynamical systems: Networks, chimeras and reservoir computing," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 377, no. 2153, 2019.

[56] D. Brunner, B. Penkovsky, B. A. Marquez, M. Jacquot, I. Fischer, and L. Larger, "Tutorial: Photonic neural networks in delay systems," *Journal of Applied Physics*, vol. 124, no. 15, p. 152004, 2018.

[57] X. Huang, T. H. Klinge, J. I. L. B, X. Li, and J. H. Lutz, *Unconventional Computation and Natural Computation*, vol. 7445. 2012.

[58] J. Tang, C. Deng, and G. B. Huang, "Extreme Learning Machine for Multilayer Perceptron," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 27, no. 4, pp. 809–821, 2016.

[59] G. Tanaka, T. Yamane, J. B. Héroux, R. Nakane, N. Kanazawa, S. Takeda, H. Numata, D. Nakano, and A. Hirose, "Recent advances in physical reservoir computing: A review," *Neural Networks*, vol. 115, pp. 100–123, jul 2019.

[60] C. G. Langton, "Computation at the edge of chaos: Phase transitions and emergent computation," *Physica D: Nonlinear Phenomena*, vol. 42, no. 1, pp. 12–37, 1990.

[61] L. Cocchi, L. L. Gollo, A. Zalesky, and M. Breakspear, "Criticality in the brain: A synthesis of neurobiology, models and cognition," 2017.

[62] K. Nakajima, H. Hauser, T. Li, and R. Pfeifer, "Information processing via physical soft body," *Scientific Reports*, vol. 5, pp. 1–11, 2015.

[63] V. C. Müller and M. Hoffmann, "What Is Morphological Computation? On How the Body Contributes to Cognition and Control," *Artificial Life*, vol. 23, no. 1, pp. 1–24, 2017.

[64] H. Hauser, A. J. Ijspeert, R. M. Füchslin, R. Pfeifer, and W. Maass, "Towards a theoretical foundation for morphological computation with compliant bodies," *Biological Cybernetics*, vol. 105, no. 5-6, pp. 355–370, 2011.

[65] G. Dion, S. Mejaouri, and J. Sylvestre, "Reservoir computing with a single delay-coupled non-linear mechanical oscillator," *Journal of Applied Physics*, vol. 124, p. 152132, oct 2018.

[66] K. Caluwaerts, M. D'Haene, D. Verstraeten, and B. Schrauwen, "Locomotion without a brain: Physical reservoir computing in Tensegrity structures," *Artificial Life*, vol. 19, pp. 35–66, jan 2013.

[67] C. Fernando and S. Sojakka, "Pattern Recognition in a Bucket," in *Advances in Artificial Life* (W. Banzhaf, J. Ziegler, T. Christaller, P. Dittrich, and J. T. Kim, eds.), vol. 2801, (Berlin, Heidelberg), pp. 588–597, Springer Berlin Heidelberg, 2003.

[68] X. Sui, Q. Wu, J. Liu, Q. Chen, and G. Gu, "A review of optical neural networks," *IEEE Access*, vol. 8, pp. 70773–70783, 2020.

[69] J. M. Shainline, "The Largest Cognitive Systems Will be Optoelectronic," *2018 IEEE International Conference on Rebooting Computing, ICRC 2018*, pp. 1–10, 2019.

[70] H. J. Caulfield, J. Kinser, and S. K. Rogers, "Optical Neural Networks," *Proceedings of the IEEE*, vol. 77, no. 10, pp. 1573–1583, 1989.

[71] F. Laporte, J. Dambre, and P. Bienstman, "Neuromorphic Computing with Signal-Mixing Cavities," *2018 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–4, 2019.

[72] D. Brunner and I. Fischer, "Reconfigurable semiconductor laser networks based on diffractive coupling," *Optics Letters*, vol. 40, no. 16, p. 3854, 2015.

[73] G. Van Der Sande, D. Brunner, and M. C. Soriano, "Advances in photonic reservoir computing," *Nanophotonics*, vol. 6, no. 3, pp. 561–576, 2017.

[74] B. Schrauwen, M. D'Haene, D. Verstraeten, J. Van Campenhout, and J. V. Campenhout, "Compact hardware liquid state machines on FPGA for real-time speech recognition," *Neural Networks*, vol. 21, no. 2-3, pp. 511–523, 2008.

[75] B. Penkovsky, L. Larger, and D. Brunner, "Efficient design of hardware-enabled reservoir computing in FPGAs," *Journal of Applied Physics*, vol. 124, no. 16, p. 162101, 2018.

[76] M. C. Soriano, S. Ortin, L. Keuninckx, L. Appeltant, J. Danckaert, L. Pesquera, G. der Sande, S. Ortín, L. Keuninckx, L. Appeltant, J. Danckaert, L. Pesquera, and G. V. D. Sande, "Delay-Based Reservoir Computing: Noise Effects in a Combined Analog and Digital Implementation," *IEEE transactions on neural networks and learning systems*, vol. 26, no. 2, pp. 388–393, 2015.

[77] J. H. Jensen and G. Tufte, "Reservoir Computing with a Chaotic Circuit," in *Artificial Life*, no. September, pp. 222–229, MIT Press, 2017.

[78] M. L. Alomar, V. Canals, N. Perez-Mora, V. Martínez-Moll, and J. L. Rosselló, "FPGA-Based Stochastic Echo State Networks for Time-Series Forecasting," *Computational Intelligence and Neuroscience*, p. 3917892, 2016.

[79] N. D. Haynes, M. C. Soriano, D. P. Rosin, I. Fischer, and D. J. Gauthier, "Reservoir computing with a single time-delay autonomous Boolean node," *Physical Review E*, vol. 91, no. 2, p. 20801, 2015.

[80] D. Canaday, A. Griffith, and D. J. Gauthier, "Rapid Time Series Prediction with a Hardware-Based Reservoir Computer," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 28, no. 12, p. 123119, 2018.

[81] S. A. Kauffman, "Metabolic Stability and Epigenesis in Randomly Constructed Genetic Nets," *Journal of Theoretical Biology*, vol. 22, no. 3, pp. 437–467, 1969.

[82] D. Dee and M. Ghil, "Boolean Difference Equations: Formulations and Dynamic Behavior," *SIAM Journal on Applied Mathematics*, vol. 44, no. 1, pp. 111–126, 1984.

[83] M. Ghil and A. Mullhaupt, "Boolean Delay Equations II. Periodic and Aperiodic Solutions," *Journal of Statistical Physics*, vol. 41, no. 1, pp. 125–173, 1985.

[84] E. Farcot, S. Best, R. Edwards, I. Belgacem, X. Xu, and P. Gill, "Chaos in a ring circuit," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 29, no. 4, p. 43103, 2019.

[85] D. P. Rosin, D. Rontani, D. J. Gauthier, and E. Schöll, "Experiments on autonomous Boolean networks," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 23, no. 2, p. 25102, 2013.

[86] D. P. Rosin, "Ultra-fast physical generation of random numbers using hybrid boolean networks," in *Dynamics of Complex Autonomous Boolean Networks*, pp. 57–79, Springer, 2015.

[87] D. P. Rosin, *Dynamics of Complex Autonomous Boolean Networks*. Springer, 2014.

[88] R. Zhang, H. L. D. d. S. Cavalcante, Z. Gao, D. J. Gauthier, J. E. S. Socolar, M. M. Adams, D. P. Lathrop, H. L. Hugo, Z. Gao, D. J. Gauthier, J. E. S. Socolar, M. M. Adams, and D. P. Lathrop, "Boolean chaos," *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, vol. 80, no. 4, pp. 1–4, 2009.

[89] H. L. De Cavalcante, D. J. Gauthier, J. E. Socolar, and R. Zhang, "On the origin of chaos in autonomous Boolean networks," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 368, no. 1911, pp. 495–513, 2010.

[90] D. P. Rosin, D. Rontani, and D. J. Gauthier, "Synchronization of coupled Boolean phase oscillators," *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, vol. 89, no. 4, pp. 1–7, 2014.

[91] D. P. Rosin, D. Rontani, N. D. Haynes, E. Schöll, and D. J. Gauthier, "Transient scaling and resurgence of chimera states in networks of Boolean phase oscillators," *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, 2014.

[92] D. P. Rosin, D. Rontani, and D. J. Gauthier, "Ultrafast physical generation of random numbers using hybrid Boolean networks," *Physical Review E*, vol. 87, no. 4, p. 040902, 2013.

[93] M. Lukoševičius, "A Practical Guide to Applying Echo State Networks," in *Neural Networks: Tricks of the Trade*, pp. 659–686, Springer, 2012.

[94] C. G. Langton, "Computation at the edge of chaos: Phase transitions and emergent computationhase transitions and emergent computation," *Physica D: Nonlinear Phenomena*, vol. 42, no. 1-3, pp. 12–37, 1990.

[95] T. Natschläger, N. Bertschinger, and R. Legenstein, "At the Edge of Chaos: Real-time Computations and Self-Organized Criticality in Recurrent Neural Networks," tech. rep., 2005.

[96] P. Barančok and I. Farkaš, "Memory Capacity of Input-Driven Echo State Networks at the Edge of Chaos," in *International Conference on Artificial Neural Networks*, pp. 41–48, Springer, 2014.

[97] D. Snyder, A. Goudarzi, and C. Teuscher, "Computational capabilities of random automata networks for reservoir computing," *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, vol. 87, no. 4, pp. 1–8, 2013.

[98] T. J. O'Shea and N. West, "Radio Machine Learning Dataset Generation with GNU Radio," in *Proceedings of the GNU Radio Conference*, vol. 1, 2016.

[99] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A Research Platform for Distributed Model Selection and Training," *arXiv preprint arXiv:1807.05118*, 2018.

[100] A. Edelen, C. Mayes, D. Bowring, D. Ratner, A. Adelmann, R. Ischebeck, J. Snuverink, I. Agapov, R. Kammering, J. Edelen, I. Bazarov, G. Valentino, and J. Wenninger, "Opportunities in machine learning for particle accelerators," 2018.

[101] R. A. Kishek, B. L. Beaudoin, S. Bernal, M. Cornacchia, D. Feldman, *et al.*, "The University of Maryland Electron Ring Program," *Nuclear Instruments and Methods in Physics Research, Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 733, pp. 233–237, Jan. 2014.

[102] L. Dovlatyan, D. Matthew, I. Haber, B. Beaudoin, T. Antonsen, and K. Ruisard, "Preliminary Lattice Studies for the Single-Invariant Optics Experiment at the University of Maryland," 2019.

[103] M. Reiser, *Theory and Design of Charged Particle Beams*. Wiley, Mar. 2008.

[104] A. Friedman, R. H. Cohen, D. P. Grote, S. M. Lund, W. M. Sharp, *et al.*, "Computational methods in the warp code framework for kinetic simulations of particle beams and plasmas," *IEEE Transactions on Plasma Science*, vol. 42, no. 5, pp. 1321–1334, 2014.