# Reservoir Computing Using Autonomous Boolean Networks Realized on Field-Programmable Gate Arrays

**Stefan Apostel, Nicholas D. Haynes, Eckehard Schöll, Otti D'Huys, and Daniel J. Gauthier**

**Abstract** In this chapter, we consider realizing a reservoir computer on an electronic chip that allows for many tens of network nodes whose connection topology can be quickly reconfigured. The reservoir computer displays analog-like behavior and has the potential to perform computations beyond that of a classic Turning machine. In detail, we present our preliminary results of using a physical reservoir computer for performing the task of identifying written digits. The reservoir is realized on a commercially available electronic device known as a field-programmable gate array on which we create an autonomous Boolean network for information processing. Even though the network nodes are Boolean logic elements, they display analog behavior because there is no master clock that controls the nodes. In addition, the electronic signals related to the written-digit images are injected into the reservoir at high speed, leading to the possibility of full-image classification on the nanosecond time scale. We explore the dynamics of the autonomous Boolean networks in response to injected signals and, based on these results, investigate the performance of the reservoir computer on the written-digit task. For a wide range of reservoir structures, we obtain a typical performance of ~90% for correctly identifying a written digit, which exceeds that obtained by a linear classifier. This work paves the way for achieving low-power, high-speed reservoir computing on readily available field-programmable gate arrays, which are well matched to existing computing infrastructure.

S. Apostel · E. Schöll
Institut für Theoretische Physik, Technische Universität Berlin, 10623 Berlin, Germany
e-mail: schoell@physik.tu-berlin.de

N. D. Haynes
Department of Physics, Duke University, Box 90305, Durham, NC 27706, USA

O. D'Huys
Department of Mathematics, Aston University, B4 7ET Birmingham, UK
e-mail: o.dhuys@aston.ac.uk

D. J. Gauthier (✉)
Department of Physics, The Ohio State University, 191 W. Woodruff Ave, Columbus, OH 43210, USA
e-mail: gauthier.51@osu.edu

K. Nakajima and I. Fischer (eds.), *Reservoir Computing*, Natural Computing Series,
https://doi.org/10.1007/978-981-13-1687-6_11

## 1 Introduction

As demonstrated by the breadth of contributions in this volume, there is great interest in reservoir computing (Jaeger and Haas 2004), from fundamental studies of their properties to using them for a wide range of applications. While many studies use a standard Turing-von Neumann computer to simulate a reservoir computer (RC), physical RCs have the potential to demonstrate beyond-Turing computing and may increase the information processing speed. One challenge in building a physical RC is fabricating a large enough reservoir, which may require 100s to 1,000s of nonlinear input-output nodes, and a large number of interconnects (links) between the nodes.

A highly successful alternative that is capable of operating at high speeds is to realize a RC using a single nonlinear node with time-delay feedback with a large number of 'virtual' nodes determined by the ratio of the delay time to the characteristic time scale of the node. Such platforms have been used for a wide variety of machine learning tasks, such as classifying spoken words (Appeltant et al. 2011) at high speed (Larger et al. 2017), nonlinear channel equalization (Paquot et al. 2012), and classifying serial digital data (Haynes et al. 2015). Unfortunately, this approach requires complex temporal time delaying and multiplexing of the input information (masking) and injection into the loop (Appeltant et al. 2014; Röhm and Lüdge 2018). In principle, this masking could be done in real time using dedicated hardware as has been demonstrated recently (Penkovsky et al. 2018), but most demonstrations to date perform this preprocessing offline thereby slowing down the overall operation rate of the RC.

A different, potentially scalable, approach is to use nonlinear optical micro-ring resonators fabricated on a planar photonic chip (Denis-Le Coarer et al. 2018; Mesaritakis et al. 2015), or arrays of linear micro-ring resonators or swirls where the nonlinearity is provided by the square-law detectors (that is, detectors that respond to the intensity of the light rather than the field) in the read-out layer (Katumba et al. 2018; Vandoorne et al. 2014; Vinckier et al. 2015; Zhang et al. 2014). This technology is currently limited to a small ($\sim$16) number of resonators because of optical loss and the network connectivity is limited by the planar geometry. But this platform may find future application as the quality of photonic chips improves.

Here, we focus on an approach using a commercially available electronic device known as a field-programmable gate array (FPGA), which greatly simplifies the creation of a reservoir in comparison to the photonic approaches discussed above. FPGAs are also much easier to reconfigure in comparison to custom-built boards with discrete logic (Mason et al. 2004; Zhang et al. 2009) (that is, devices using a collection of single-chip logic gates soldered on the board). Rosin (2015) pioneered the application of FPGAs to realize autonomous time-delay Boolean networks (Ghil and Mullhaupt 1985). Here, the FPGA is configured to realize a reservoir, where the FPGA logic elements serve as the network nodes that nominally perform a Boolean operation on the inputs. The term autonomous indicates that the logic elements are not gated by a master clock; they process new edges as soon as they arrive on the inputs to the logic elements. Furthermore, because the network is autonomous, signals traveling along the links have a finite propagation speed so that the link delays must be accounted for. The autonomous nature of the complex network allows for the possibility of beyond-Turing computation (Ghil et al. 2008). Autonomous Boolean networks on FPGAs have been used previously for reservoir computing and applied to high-speed pattern recognition of digital serial data patterns (Haynes et al. 2015) and high-speed forecasting of chaotic dynamics (Canaday et al. 2018). The FPGA can also operate in the clocked mode, thus realizing a finite state machine and offering the possibility of accelerating the software simulation of RCs. This approach has been applied to channel equalization (Antonik 2018; Antonik et al. 2015; Skibinsky-Gitlin et al. 2018; Yi et al. 2016) and speech recognition (Alomar et al. 2015; Penkovsky et al. 2018).

In this chapter, we describe our preliminary studies on using an FPGA-based RC for performing a classification task: identifying images containing human-written digits (the MNIST task). While RCs are most suited for processing time-dependent signals, performing a classification task on images presents interesting challenges, such as identifying efficient methods for injecting data into the reservoir. We also take this opportunity to study the dynamics of autonomous Boolean networks (ABNs) in response to a perturbation, such as a phase transition as the node in degree varies. This study helps guide the choice of metaparameters for the RC.

In the next section, we present the reservoir concept with an associated mathematical model, describe our experimental studies of the dynamics of ABNs as the network parameters vary in Sect. 3, realize an RC on an FPGA and use it for the written digit (MNIST) task in Sect. 4, and discuss future directions and conclude in Sect. 5 (Apostel 2017). We briefly discuss using FPGAs and describe our experimental workflow in the Appendix.

## 2 Reservoir Computers Based on Autonomous Boolean Networks

One aspect that distinguishes RCs from other artificial neural networks is that each node is itself a dynamical system. Glass and colleagues (Edwards and Glass 2006) have studied extensively the dynamics of autonomous Boolean networks using a model where node $i$ is described by a continuous variable $x_i$ whose behavior is governed by a first-order differential equation with decay rate $\gamma_i$ and driven by a Boolean function $f_i$ of the node inputs. The function $f_i$ is defined via a look-up table as described below and can take on essentially any Boolean function consistent with the number of inputs to the node. In the context of our work, the decay rate $\gamma_i$ models the finite rise-time of the FPGA logic elements due to their input capacitance and inductance and are nominally identical for all nodes with $\gamma_i \sim 2\pi/(0.41\,\text{ns})$. When signals propagating from node $j$ to $i$ experience delay $\tau_{i,j}$, the so-called Glass model can be extended to a set of delay differential equations (Edwards et al. 2007). Delays appear in our FPGA-based system because the signals propagate at a finite velocity

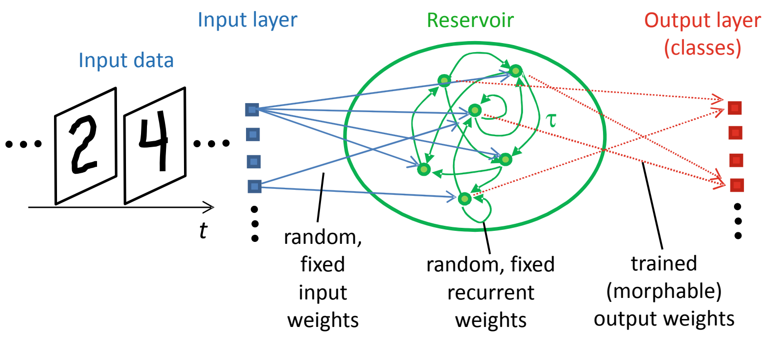
Input layer
Reservoir
Output layer (classes)
Input data
2
4
t
τ
random, fixed input weights
random, fixed recurrent weights
trained (morphable) output weights

**Fig. 1** Illustration of the reservoir computer architecture

along the link wires, where a typical delay time between nearby logic elements is only a few 10's of picoseconds. To purposefully add more delay, we use pairs of series-connected NOT gates along the network links, where we obtain a delay of ~0.52 ns per pair of gates (Lohmann et al. 2017). We use special care in writing the hardware description language used to configure the FPGA (described in greater detail in the Appendix) to make sure that the logic gates used to create delay are not removed from our design.

Adopting this theoretical approach for reservoir computing with time-delay ABNs, illustrated in Fig. 1, the dynamics of the RC are given by Gauthier (2018)

$$\frac{dx_i}{dt} = -\gamma_i x_i + \gamma_i f_i \left( \sum_{j=1}^{J} W^{\text{in}}_{i,j} u_j(t) + \sum_{n=1}^{N} W_{i,n} x_n(t - \tau_{i,n}) \right), \quad i = 1, \ldots, N \tag{1}$$

$$y_m(t) = \sum_{n=1}^{N} W^{\text{out}}_{m,n} x_n, \quad m = 1, \ldots, M. \tag{2}$$

Here, $u_j(t)$ are the $J$ signals input to the RC, which are connected to the $N$ reservoir nodes with random fixed weights $W^{\text{in}}_{i,j}$, $W_{i,n}$ are the random fixed internal weights of the reservoir, $y_m(t)$ are the $M$ outputs of the RC with trained (morphable) weights $W_{m,n}$. Often, the $W$'s are sparse so that the typical node in-degree (number of inputs) $K_i$ is small. The reservoir embeds the input data to a higher-dimension phase space when $N > J$ because of the nonlinear response of the node, known as dimension expansion and often a key requirement for effective information processing. Because of the time delays, the phase-space dimension of the reservoir is infinite, which can be seen by considering that the initial conditions of the time-delay differential equations must be specified over the real interval of the maximum link delay.

Evaluating $f_i$ is particularly simple when the arguments $u_j$ and $x_n$ are Boolean, but the $W$'s are real. Here, the multiplications and additions appearing in the argument of $f_i$ can be done a single time after the $W$'s are chosen, thereby defining the Boolean look-up table (LUT). Thereafter, only the LUT is used and no further operations are required. This procedure maps perfectly onto the structure of the FPGA logic elements, which are based on LUTs, as discussed in the Appendix.

Models similar to Eq. (1) have been used to understand the dynamics of time-delay ABNs representing simple genetic regulatory networks are realized on an FPGA, for example, and can capture the extremely long transient behavior observed experimentally (D'Huys et al. 2016; Haynes et al. 2015; Lohmann et al. 2017). While we do not consider solutions to Eq. 1 here, we include it for completeness because it is a good starting point for a theoretical analysis of FPGA-based RCs.

For the classification task considered here, we adjust $W_{k,n}^{\text{out}}$ using a finite-size training data set so that the resulting output properly classifies each input in a least-square sense, known as supervised learning. Specifically, the output weight matrix $\mathbf{W}^{\text{out}}$ is determined by injecting a finite-length input training data set $\mathbf{U}(t)$ and recording the network dynamics $\mathbf{X}(t)$. Here, we assume that the state of the network is sampled at equal discrete time intervals $\Delta t$ so that the matrices are finite dimensional. Based on these observations, we modify $\mathbf{W}^{\text{out}}$ to minimize the error of the output $\mathbf{Y}$ (the classes) to the expected output $\mathbf{Y}^{\text{expected}}$, resulting in

$$\mathbf{W}^{\text{out}} = \mathbf{Y}^{\text{expected}}\mathbf{X}^T \left(\mathbf{X}\mathbf{X}^T + \beta^2\mathbf{I}\right)^{-1}, \tag{3}$$

where $\beta$ is a regularization parameter, $\mathbf{I}$ is the identity matrix, and $T$ indicates the transpose. A nonzero value of $\beta$ ensures that the norm of $\mathbf{W}^{\text{out}}$ does not become large, prevents sensitivity of the training to noise, and improves the generalizability of the RC to different inputs. We stress that $\mathbf{X}(t)$ is a concatenation of the network dynamics over all input data, which is a collection of image data described below in Sect. 4.1. We can also find a solution to Eq. (3) using gradient-descent methods, which are helpful when the matrix dimensions are large. Gradient-descent routines are readily found in modern toolkits developed by the deep learning community, and they can operate at high speed using graphical processing units. Another approach is to use recursive least-squares.

## 3 Dynamics of Random Autonomous Boolean Networks

Before discussing the performance of the RC applied to the MNIST classification task, we explore the dynamics of time-delay ABNs realized on an FPGA for different network parameters. We seek to identify a phase transition from ordered to disordered (chaotic) behavior and compare our observation to the prediction for clocked Boolean networks. Identifying the location of the phase transition is important because the performance of the RC is expected to be highest near the border of the transition from order to chaos (Yildiz et al. 2012).

### 3.1 Dynamics of Clocked Random Boolean Networks

There exists a considerable literature on the dynamics of clocked random Boolean networks, and we briefly summarize some of the findings here. A clocked random Boolean network (RBN) is typically taken to consist of $N$ nodes each of which receives exactly $K$ inputs. The Boolean node function $f_i$ is defined in terms of a random LUT, where the probability of an entry in the LUT taking on the value 0 (1) is $p$ $(1 - p)$ and is often called the node bias (Derrida and Pomeau 1986; Flyvbjerg 1988; Luque and Solé 2000).

For a clocked RBN with random and fixed (quenched) functions and connections, the network is a deterministic system with a finite number of states (a finite-state machine) and must show periodic behavior with a maximum period of $2^N$. Different network behaviors are observed as $K$ and $p$ varies. One behavior is called *ordered* when the dynamics go to a fixed point or to a low-period periodic orbit. Another behavior, often called *chaotic*, is when the period approaches its maximum value and the specific periodic orbit followed is sensitive to a change in the initial value of a single network node. Of course, chaos cannot exist in an RBN because all behaviors are periodic, but the term is suggested because of the sensitivity of the dynamics on the initial conditions. Using an annealed network approach (Derrida and Pomeau 1986), the order-chaos transition is given by

$$K_c = \frac{1}{2p(1-p)} \tag{4}$$

in the limit $N \to \infty$ and shown in Fig. 2.

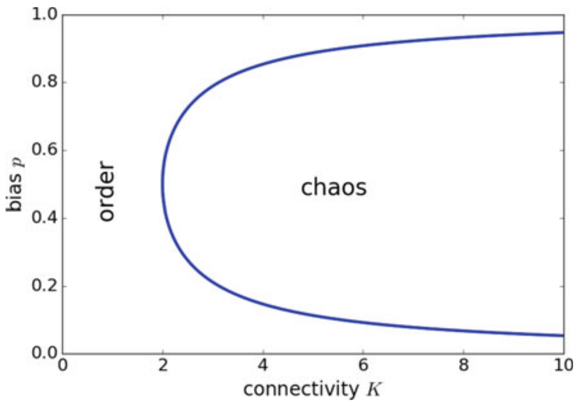

**Fig. 2** Order-chaos phase transition in a clocked RBN

## 3.2 Dynamics of ABNs on an FPGA

Guided by the results on clocked RBNs, we investigate experimentally the dynamics of random ABNs on an FPGA. Here, we measure the Booleanized state of each node at discrete (clocked) time intervals using the finite-state machine described in the Appendix. We choose a subset of networks from the full range of possibilities described by Eq. 1 so we can make a comparison to the previous work on clocked networks. In particular, we consider networks with $N = 64$ each having exactly $K$ inputs and outputs so that $W_{i,n}$ appearing in Eq. 1 is quite sparse. Here, we consider only $K \leq 4$. Each node is assigned a random and fixed Boolean function $f_i$ with bias $p$ under the restriction that $f_i = 0$ when all the inputs are zero so that the network is in the quiescent state (no self-oscillation) when the FPGA is first turned on or reset to this value. This restriction limits the bias to $p_{\max} = 1 - 1/2^K$, but is advantageous for realizing an RC because it results in a well-defined initial state of the reservoir. In addition, to slow down the network dynamics (Canaday et al. 2018) to better match the rate at which we can inject data into the RC (see the next section), and to more closely match the clock RNB theory, we use a nominally constant link delay $\tau_{i,n} = 12.8 \pm 0.5$ ns. Here, the link time delay is set by using pairs of series-connected NOT gates, where each pair of gates causes a delay of $\sim$0.52 ns (Lohmann et al. 2017). For this study, we investigate the dynamics of 6,000 randomly chosen networks.

The initial condition of the network is set using an OR logic gate at the output of each node as illustrated in Fig. 3. Initially, the state of all nodes is set to 0 to reset the network, and then the desired initial condition is set during a 5-ns-long window and passed to the link delay lines. This is accomplished using the OR gate, where $u_j(t)$ appears at the output while the node is in the state 0. Once signals appear at the input of the node from other network nodes (after a delay time $\tau_{j,s}$), any signals related to $u_j(t)$ still being injected into the node will be 'blended' with the incoming node signals via the OR gate.

In general, setting the initial conditions to a random binary value will destabilize the initial stable network state where all nodes are equal to 0, giving rise to a transient that can last for a time beyond our data-collection window for $p \sim 0.5$. After initializing the network, we measure the state of each node every 5 ns for 200 samples (1 $\mu$s total record length), and transfer the data to a computer for analysis. The network is

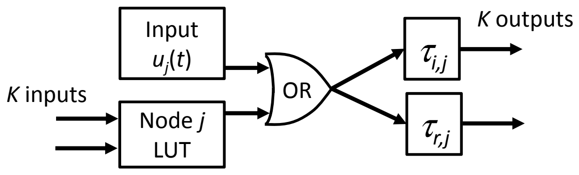

**Fig. 3** Using an OR gate on the output of node $j$ to inject the input data for the node, illustrated here for $K = 2$. The signal emanating from the OR gate passes to the delay-line links to nodes $i$ and $r$

reset to the 0 state at the end of data collection for initial condition. For each network realization, we use 1,000 randomly chosen initial conditions and repeat the experiment 50 times for each initial condition. We do not attempt to study the duration of the transient beyond 1 $\mu$s because it is not relevant for reservoir computing although we note the extremely long (second to minutes) chaotic time transient times have been observed in simple Boolean networks with similar bias (Haynes et al. 2015).

One issue that arises when using an FPGA is that a randomly assigned node LUT may have the same output (0 or 1) regardless of the inputs. In this case, the optimizer within the compiler that converts the *Verilog* hardware description language to a bitstream to configure the FPGA removes such nodes from the network. This is more pronounced for low and high biases and the actual network synthesized on the chip has fewer than 64 nodes when the network is pruned by the optimizer (typically, only a few nodes are pruned). In the figures presented below, we use different color symbols to indicate networks that have been pruned by the compiler. This behavior does not change the network dynamics or time scales because such a node would have been inactive. Nodes with high bias ($p \sim 1$) may also be inactive, but we already exclude nodes with high bias as discussed above.

We observe a wide range of network dynamics as we vary $p$ and $K$. Again motivated by the research on clocked RBNs, we initially search for the network to settle down to a fixed-point behavior where all network nodes take on a constant value (frozen dynamics) after a transient time. While we only measure the network dynamics at discrete times, it is unlikely that fast oscillations in between the measurement times would escape unnoticed. We search for fixed-point behavior by measuring the Hamming distance between the state of the network at time $t_0$ with its values at all later times up to the limit of our 200 samples. Here, the Hamming distance between two Boolean vectors $\mathbf{A}$ and $\mathbf{B}$ is defined as

$$H(\mathbf{A}, \mathbf{B}) = \sum_{i=1}^{N} A_i \oplus B_i. \tag{5}$$

Figure 4 shows the number of time steps required for the dynamics of the ABN to reach a fixed point (network realizations where a fixed point is never observed are not shown). We also show the probability for a network to reach a fixed point. Qualitatively consistent with the predictions shown in Fig. 2 for a clocked RBN with the same network structure as our ANB, we observe longer transients as $p$ approaches 0.5 from above and below, and the range of biases that do not show fixed-point behavior widens as $K$ increases. Of course, we are only considering fixed-point behavior and so these results only suggest the location of the order-chaos boundary, although we find that chaotic behavior tends to be observed when the probability for observing a fixed point drops near zero around $p = 0.5$

Over the range of $p$ where the probability of observing a fixed point is nearly zero (the region around $p \sim 0.5$ where the green line is near zero), we observe fast, complex behavior after an initial short transient that depends sensitively on the initial conditions, likely a manifestation of chaos in the ABN (Zhang et al. 2009) as

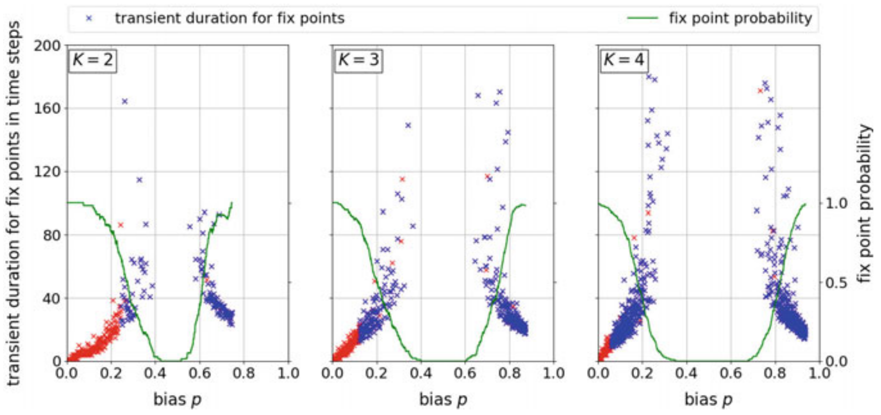

**Fig. 4** Length of transient for cases when the ABN reaches a stable fixed point for different network parameters. Blue symbols indicate unmodified networks and red symbols indicate optimizer-pruned networks. The green solid line shows the probability that a network reaches a stable fixed point

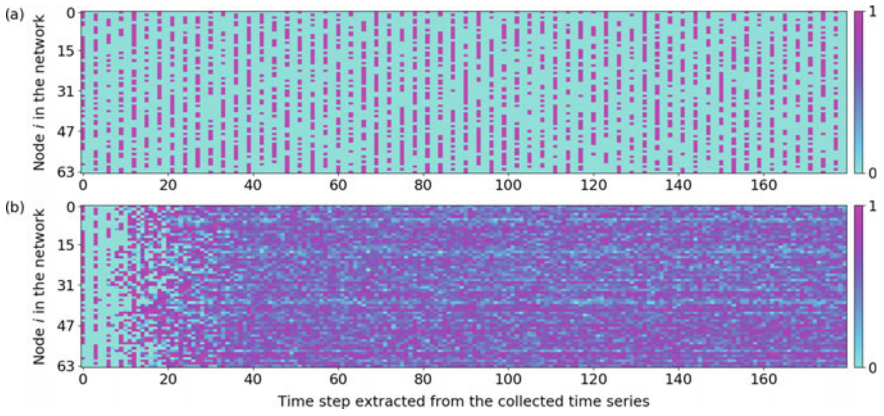

**Fig. 5** Dynamics of a Boolean networks with $N = 64$, $K = 4$, and $p = 0.55$ for a clocked network simulated numerically and b an autonomous network on the FPGA averaged over 50 experimental runs with the same initial conditions for each. Boolean 0 (1) is colored cyan (magenta). This network realization has no pruned nodes

shown in Fig. 5b. Here, the dynamics is so fast that our finite sampling rate is too low to faithfully record the dynamics. As seen in Fig. 5a, the behavior predicted by a discrete-time model using the same LUT and initial conditions is similar to our observations of the ABN on the first few time steps, but disagrees thereafter. This disagreement is perhaps not too surprising because the discrete map is a finite-state machine and hence cannot display chaos (the dynamics eventually has to repeat). The fact that we observe chaos gives a hint that an RC realized with an ABN reservoir on an FPGA could display beyond-Turing computation, although observing chaos is not a necessary or sufficient condition for beyond-Turning computation.

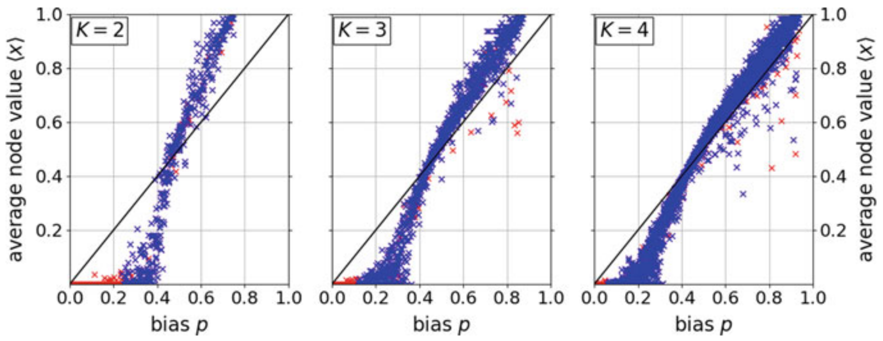

**Fig. 6** Average value of the network nodes for different network parameters

Close inspection of Fig. 5b reveals apparent horizontal stripes, indicating that the average value of the nodes takes on different values. For this data set, the average value of the nodes range from $\langle x_i \rangle = 0.36$ to 0.74, with an average for all nodes of $\langle x \rangle = 0.57 \pm 0.07$, which is consistent with the value of $p$. We currently do not have an explanation for the appearance of these patterns.

We find that another qualitative measure of the order-chaos boundary can be obtained by measuring $\langle x \rangle$ for different network parameters as shown in Fig. 6. Here, we perform the average from time steps 150 through 200 to minimize the effects of the transients. At $p = 0$, we expect that $\langle x \rangle = 0$ because all nodes in the network are inactive regardless of their input. As $p$ increases, we see that $\langle x \rangle$ remains near zero until an apparent transition to complex behavior indicated by a rapid rise in $\langle x \rangle$. The value $p$ at this transition is smaller as $K$ increases, consistent with the phase-transition behavior of clocked RBNs shown in Fig. 2. We also show a line indicating $\langle x \rangle = p$, which is expected for a RBN with $N$, $K \rightarrow \infty$. As $K$ increases, the average node value approaches this line, which is an interesting result because our network is quite different from an RBN. Specifically, we are using an ABN, we restrict the node LUTs so that the all-zero state is a solution, and $N$ and $K$ are not all that large.

## 4 Reservoir Computing with ABNs on an FPGA

The experiments described in the previous section demonstrate that an ABN-based reservoir can be constructed on an FPGA whose behavior can be adjusted from ordered to chaotic simply by adjusting the node bias and in-degree. Operating near or within the chaotic domain demonstrates that the reservoir embeds the dynamics in a high-dimensional phase space, one crucial characteristic needed for reservoir computing. It is also widely believed that the reservoir must display other characteristics when data is input to the reservoir, including (Jaeger and Haas 2004):

- **Separation of different input states**. Input data in different classes should have distinct output dynamics.
- **Generalization of similar inputs to similar outputs**. Different input data within a class should have similar output dynamics.
- **Fading memory**. The output dynamics should be correlated with the input dynamics over a time known as the *consistency window*, but this temporal correlation should fade.

Based on these properties, an RC is well suited for classification or prediction based on correlations in data over time because it inherently has temporal memory. For classification tasks on static image data such as the MNIST data set, the correlations are inherently spatial. Thus, we break the image up into sub-images and feed them rapidly into the reservoir (at the 200 MHz limit imposed by our write and read procedure) to create spatial-temporal correlations. While this process is a bit contrived, it may be well suited for processing high-speed video imagery data where the data is usually generated in a progressive scan of the image.

In the next section, we introduce the MNIST classification task and modifications we make to the input data to make a better match to our reservoir characteristics, then inject this data into ABNs to verify the behaviors described above. Finally, we describe the performance of an FPGA-based RC for the classification task.

## *4.1 The MNIST Classification Task*

The MNIST database (MNIST 2021) of handwritten digits is a collection of 60,000 training samples and 10,000 test samples of handwritten digits from 0 to 9. Each written digit in the original data set, referred to as NIST after the U.S. federal agency that collected the data, is represented by black-and-white 20 × 20-pixel, which was modified (MNIST) by centering the image and padding with white space, resulting in a gray-scale 28 × 28-pixel image. Figure 7 shows four sample MNIST images for each written digit.

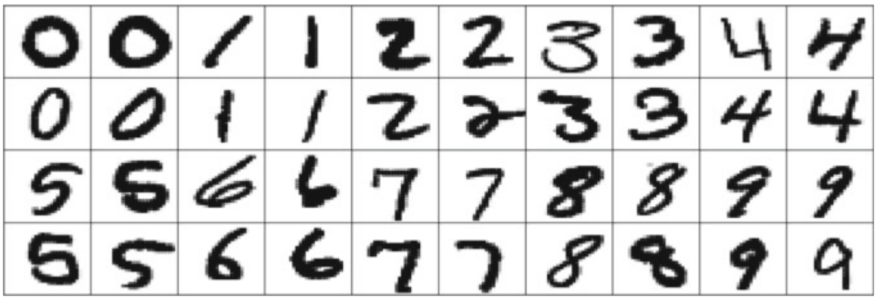

**Fig. 7** Four examples for each digit of the MNIST data set shown as gray-scale images

As mentioned above, we seek to inject sub-images into the reservoir to exploit its fading memory. There is a trade-off in using too many sub-images because the time interval for injecting data may be longer than the reservoir consistency window. We also want to use sub-images with no more than pixels than reservoir nodes. To reduce the parameter space, we choose to map each pixel to a single reservoir node and thus to have a sub-image size of 64 pixels. Thus, in this case, the input layer is the sub-image and we use a restricted connectivity of the input layer to the reservoir: each input node only connects to a single reservoir node. For cases when there is not enough image data to fill a full 64-element input data vector, we pad the data with zeros. When the reservoir is pruned by the *Verilog* compiler, the corresponding pixel is not mapped into the reservoir and hence this information is lost.

For most of the studies presented below, we reduce the MNIST images to $16 \times 16$ pixels distributed over 4 sub-images. We tested many methods for creating sub-images, such as vertical or horizontal image segments, or using random masks. We find no statistically significant differences in the performance of the RC for the MNIST classification task for different protocols for sub-images creation; the data shown below uses vertical image segments.

The data is injected into the network using the same method described in Sect. 3.2 above using an OR gate at the output of each node (see Fig. 3) controlled by our data write/read FSM described in the Appendix. Here, the data is loaded into the links, which have a nominal propagation delay of 12.8 ns. A sub-image is input every 5 ns (200 MHz input data rate) and hence some of the data is 'blended' with the reservoir dynamics if the total image time exceeds the link delay time. For example, it takes $4 \times (5\,\text{ns}) = 20\,\text{ns}$ to inject a full image with 4 sub-images so that the first sub-image is fully blended with the reservoir dynamics, the second sub-image is blended with the reservoir dynamics for $\sim$2.2 ns, while the last two sub-images are presented fully for 5 ns. We did not realize this shortcoming of our method until after substantial data collection. This represents a limitation of this initial study and the error rates of the RC presented in Sect. 4 below are likely lower bounds. This limitation will be addressed in future studies.

Figure 8 illustrates the steps involved to create the reduced image. This first step is to determine an image 'focus,' which is an image coordinate determined by an average of the pixel coordinates weighted by their respective gray-scale value. The intersection of the solid-blue lines shown in Fig. 8a shows the image focus, which is not necessarily the center of the image indicated by the intersection of the black dashed lines. Next, we determine exterior white pixels to remove the excess padding. We then create a grid of $q \times q$ pixels within this region centered on the focus with $q = 8 - 20$, but $q = 16$ typically. A resulting pixel is labeled by filling each pixel like a 'bucket' by adding the value overlapping area $\times$ gray-scale weight (0–255) from each original MNIST pixel to the new grid pixel. The new pixel will be set to Boolean 1 (0) if the filled value reaches (does not reach) a threshold set at approximately the half-gray-scale point. This process is depicted in Fig. 8b with Boolean 1 shown as blue and the bucket gray scale shown as an underlay.

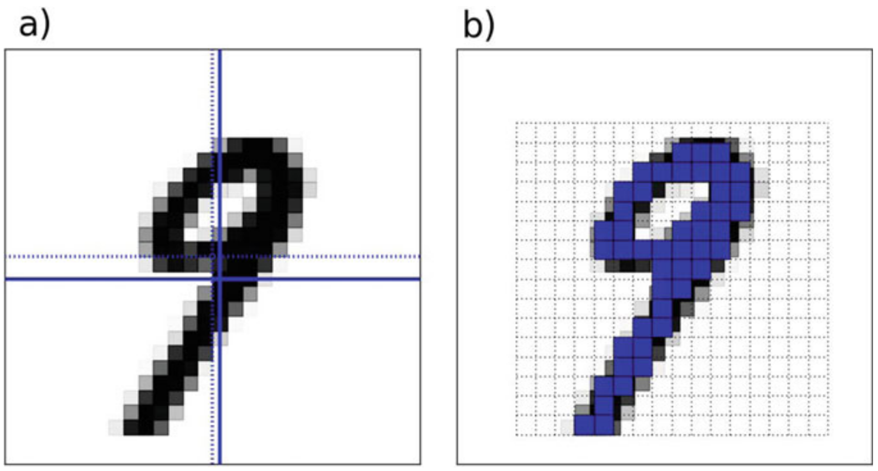

**Fig. 8** **a** Determining the focus of the MNIST image. **b** Creating a reduced image shown here for the case of 16×16 pixels

## *4.2 Dynamics of ABNs with Data Injection: The Consistency Window*

As mentioned above, an important feature of an RC is to separate data from different classes and to give similar output for data from the same class. Even if the reservoir is operating in the chaotic state, the transient dynamics can still be used for classification when the output data is restricted to the consistency window (Haynes et al. 2015; Uchida et al. 2004).

Haynes et al. (2015) proposed using the Hamming distance for Boolean data given in Eq. (5) as a measure for the consistency window. We follow the same approach here. Figure 9 shows examples of the dynamics of the average Hamming distance, where the bias is set in the ordered regime (panel a) so that the dynamics evolves to a fixed point for all inputs and the other for a bias in the chaotic regime (panel b). Here, a random Boolean initial condition is injected into the network for a single 5-ns-long period, which destabilizes the network dynamics. This is repeated many times for the same and different initial conditions. The consistency window $w$ quantifies the interval over which similar and different inputs can be distinguished.

Also apparent in this plot is the fading memory property of the ABNs. Beyond the consistency window, the output state is uncorrelated with the input state in both the ordered and chaotic regimes.

We repeat this experiment for a large number of randomly chosen reservoirs as a function of network parameters. We determine $w$ by finding the intersection point of lines fit to the early part of the data. This procedure can result in values greater than the 200 steps over which we collect data when the slopes are shallow. As seen in Fig. 10, the consistency window sharply peaks for bias values at the ordered-chaotic

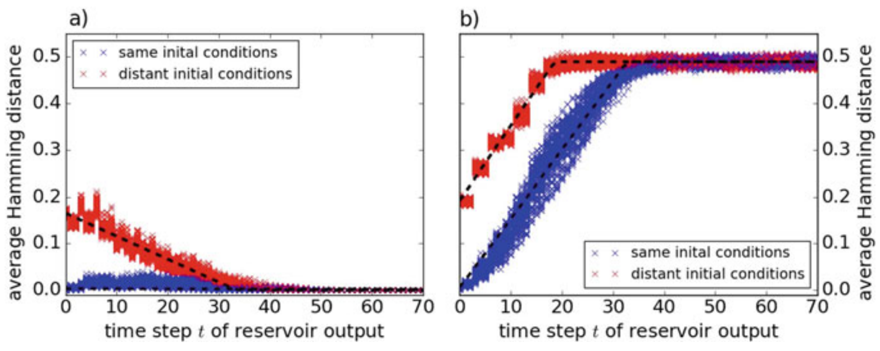

**Fig. 9** Temporal evolution of the normalized Hamming distance using two randomly selected data sets with 64 pixels injected into the reservoir for 5 ns. The blue symbols indicate repeated injection of the same data and finding the Hamming distance for each trial, and the red symbols indicate the difference between this data set and a randomly chosen second data set for **a** $K = 2$ and $p = 0.387$ and **b** $K = 4$ and $p = 0.476$. The black dashed lines are fits of the data to straight lines, the intersection point of which are used to indicate the end of the consistency window, about ~30 time steps for both sets of data

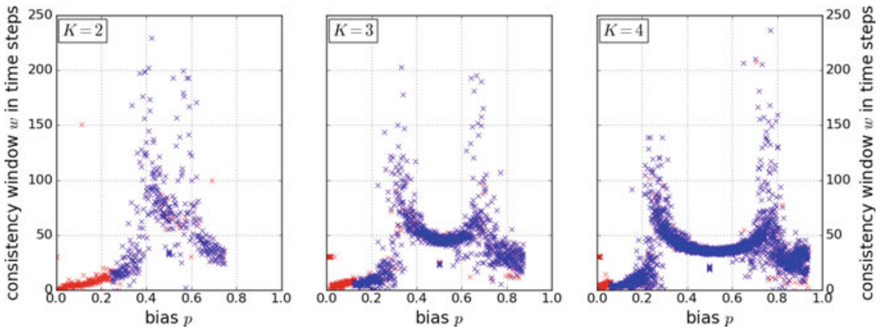

**Fig. 10** Average consistency window for as a function of network parameters. Blue symbols indicate unmodified networks and red symbols indicate optimizer-pruned networks

boundary (see Fig. 4) and approaches zero for small $p$. In the chaotic regime, $w$ takes on it smallest value around $p = 0.5$ of ~50 time steps for $K = 3$ and ~35 time steps for $K = 4$.

We also explore the dynamics of the network where all nodes execute the XOR function, which corresponds to $p = 0.5$. This is motivated by the previous work of Haynes et al. (2015) who realized an RC for serial digital data pattern recognition using a single XOR node with self-feedback over two links with different time delays. The results of this study are contained in the tight cluster of blue data points in Fig. 10 at $p = 0.5$ and $w \sim 35$ for $K = 2$, $w \sim 25$ for $K = 3$, and $w \sim 20$ for $K = 4$. Given that these point clusters are below the 'U'-shaped band of points appearing above the point cluster, corresponding to the random networks, indicates that the random networks outperform the XOR network with regard to the consistency window, which may improve the performance of the RC.

The results presented in this section demonstrate that a random ABN satisfies the criteria of separation, generalization, and fading memory that is believed to be required for reservoir computing.

## *4.3 Realizing an RC for the MNIST Classification Task*

In this section, we report on the performance of the RC on the MNIST classification task using the following procedures. We use the first 42,000 images from the MNIST data set and reduce the images to $16 \times 16$ pixels using the procedure given in Sect. 4.1. We use less than the total MNIST image database size to reduce the computational cost of our experiments and subsequent analysis. The output of the reservoir for each written-digit image is a series of 200 time steps for each of the 64 reservoir nodes. The resulting data for the output state of the network is transferred to a personal computer, then to the Open Science Grid (see the Appendix) for training the RC output weights. Even in binary format, the file size for the network output state (42,000 images $\times$ 32 bytes/image $\times$ 200 time steps) is 672 MB for a single reservoir realization. For this reason, we only choose 800 random networks when measuring RC performance as a function of network parameters rather than the 6,000 used in Sect. 3.2 above.

These data are used to determine $W^{\text{out}}_{m,n}$ using a ridge-regression regularization parameter $\beta^2 = 10^{-3}$. The 10-element class output vector $\mathbf{Y}^{\text{expected}}$ has a value of 1 for the corresponding image and -1 for the remaining elements. The output data for each image is reshaped into a single feature vector with a total length $64 \times 200 = 12{,}800$ elements with a total feature matrix $\mathbf{X}$ in $\mathbb{R}^{12{,}800 \times 42{,}000}$; performing the matrix inversion given in Eq. 3 with this data set pushes the limits of our accessible computer hardware in terms of memory to store the matrices and computation time.

The classification performance for each individual written digit is evaluated using the cross-validation method. Here, our selected data set of 42,000 written-digit images is split into 5 equal-size groups of 8,400 images each. The training of $W^{\text{out}}_{m,n}$ is repeated 5 times using 4 groups for the training and the remaining for testing the performance of the RC using this ‘unseen’ data set. After repeating this procedure for all groups, the percent of correct classifications for each written digit is calculated (that is, the fraction of classification attempts that was completed correctly). This procedure is then repeated for each of the 800 random reservoirs and for each network parameter. The performance $P$ is defined as the average of the percentage correct score for each written digit.

### 4.3.1 Exploring Fading Memory in the RC

In the previous section, our observations indicate the presence of fading memory (see Fig. 9) as quantified by a single metric given by the consistency window. A

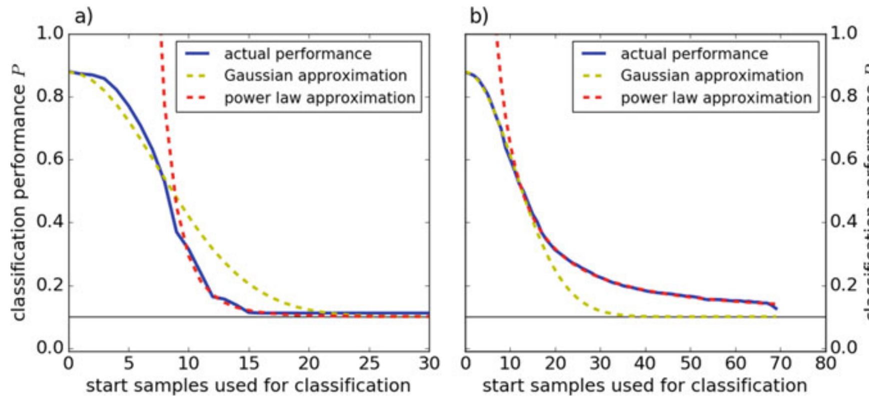

**Fig. 11** Fading memory of two example random reservoirs with **a** $K = 4$ and $p = 0.912$ and **b** $K = 4$ and $p = 0.553$

more nuanced measure for how the fading memory affects the performance on the MNIST task is obtained by varying the range of time steps used in the classification task (that is, the location and length of the block of data $\mathrm{X}(t)$ used during RC training). We expect that the initial data is highly correlated with the input data and hence it contains limited new information that can be used for classification. At the other extreme, there is a loss of correlation at later times as the network short-term memory fades. We adjust the location and length of the data block by repeating the classification task using data only from a start time step in the range $0 < t_{\text{start}} < 70$ until a final time step of $t_{\text{end}} = 70$ to determine the classification performance $P$. That is, X only has data from observation times $t_{\text{start}}$ to $t_{\text{end}}$. At the final time step, there is almost no data recorded from the reservoir, explaining the small dip apparent for the last data point of Fig. 11b).

Figure 11 shows two characteristic dependencies of the performance as a function of $t_{\text{start}}$. The performance starts high, followed by a rapid decrease, followed by a long decaying tail. To compare performance across the random networks, we find empirically that the initial high-performance section is well-fit with a Gaussian distribution, which characterizes the duration of 'short-term memory.' The longer tail is well-fit by a power-law of the form

$$P(t_{\text{start}}) = c(t_{\text{start}})^{-\alpha} + 0.1 \tag{6}$$

and quantifies the 'long-term' memory of the reservoir. As $t_{\text{start}} \to \infty$, $P \to 0.1$, corresponding to a random guess of the 10 written-digit classes. As seen in the figure, the fits to these two functions is reasonable.

From the Gaussian fit, we extract a short-term memory coefficient $\sigma$ equal to the $1/e$ half-width of the Gaussian function in units of start samples used for classification and gives a characteristic time scale for the short-term memory. Already after a few time steps, the performance drops substantially, for which the output data contains

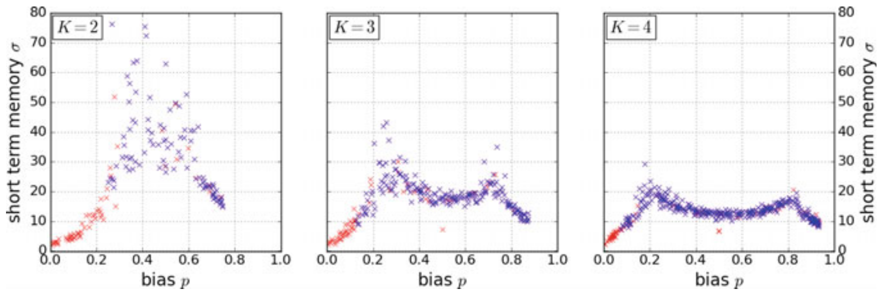

**Fig. 12** Short-term memory of the RC for different reservoir parameters

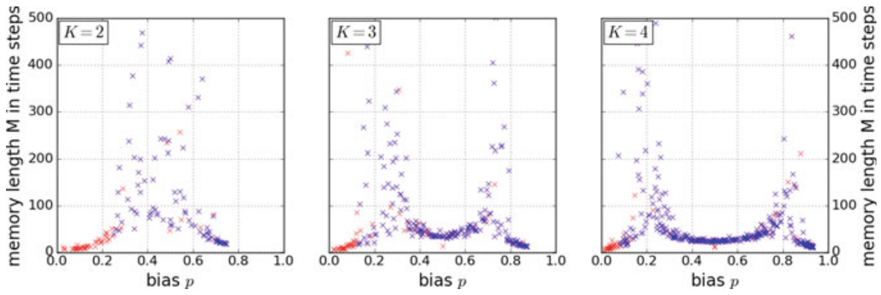

**Fig. 13** Long-term memory of the RC for different reservoir parameters

only partial information of the input image (recall we insert data into the network over 4 time steps for the 16 $\times$ 16-pixel images). Figure 12 shows the short-term memory as a function of the network parameters. Similar to the time to reach a fixed point shown in Fig. 4, the short-term memory increases near the ordered-chaotic transition boundary, but remains high throughout the chaotic regime for $K = 2$. For $K = 2$ and 3, there is less scatter in the data, the domain of long memory time increases with $K$, the memory scale peaks at the transition boundary, and there exists a minima in the memory time at $p \sim 0.5$ between the peaks, although the dependence is fairly flat. The longer short-term memory time for $K = 2$ may suggest a reason for the slightly higher best performance found for this network connectivity described below in Sect. 4.

From the power-law fit, we extract a long-term memory coefficient $M$ defined as the time at which the performance drops to 0.11. This is a measure of the time needed for almost all information to vanish from the system. As seen from Fig. 13, $M$ is peaked near the ordered-chaotic boundary, with a larger separation between the two boundaries as $K$ increases, and takes on a minimum value in the chaotic domain when $p \sim 0.5$. Interestingly, it is possible to create networks with long memory (10's of samples corresponding to $>$50 ns) even for reservoirs with biases as low as 0.1 when $K > 2$, likely due to the large number of recurrent loops in the network and the relatively long link time delays.

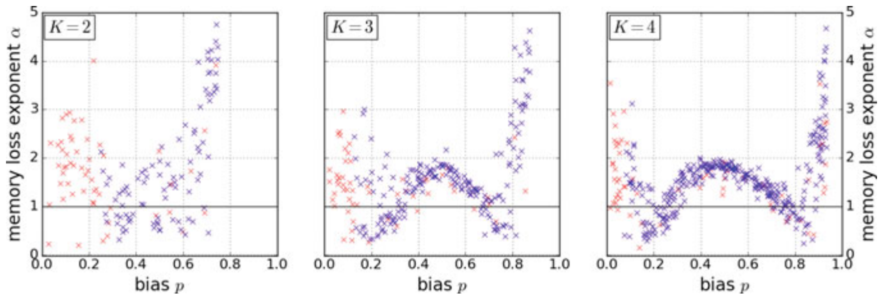

**Fig. 14** Power-law exponent for long-term memory in the RC for different network parameters

It is not possible to give an appropriate scale for the long-term memory because it appears to be well described by a power law, which has no scale. As a substitute, we determine the power-law exponent $\alpha$ for different network parameters as shown in Fig. 14. For a power-law with a negative exponent ($\alpha > 0$ in Eq. 6), its integral is finite for $\alpha > 1$ and will diverge otherwise. The exponent becomes large for $p \rightarrow 0$ and $p \rightarrow 1$, which is due to fact that these networks reach a fix point within a few time steps and almost all nodes are in the inactive (active) state for low (high) bias. For $p$ close to the phase transition, $\alpha < 1$, suggesting the existence of substantial long-term memory and hence long-term retention of information in the network. For $p \sim 0.5$, $\alpha \approx 2$ for all $K$, demonstrating that the long-term memory is nearly absent ($\alpha > 1$) and fairly insensitive to $K$, whereas the short-term memory is more sensitive to this parameter.

### 4.3.2 RC Performance for Different Reservoir Parameters

The previous sections demonstrate that an FPGA-based RC shows a fairly strong dependence of the consistency window and memory on network parameters $K$ and $p$. Based on the vast literature on RCs that states it is important to optimize the consistency window and fading memory, we find a surprising lack of dependence of the RC performance on $K$ and $p$ for the MNIST task as seen in Fig. 15. As in our previous plots, the red symbols correspond to networks whose inactive nodes are pruned by the *Verilog* compiler and optimizer. These pruned networks tend to have lower performance; the lowest performer outliers have only a few active nodes and hence there is little information available for classifying the written digits. For fully realized networks with no pruning, indicated by the blue symbols, the performance is only weakly dependent on $K$ and $p$, although the domain of well-performing networks increases with $K$. Interestingly, we obtain good performance throughout the chaotic domain and well into the ordered domains (Yildiz et al. 2012).

Zooming in on the high-performance, un-pruned networks (Fig. 16), some structure is evident with the largest spread of 4% for $K = 2$. Evident is a slight peaking in performance near the ordered-chaotic boundaries, although the spread in the data

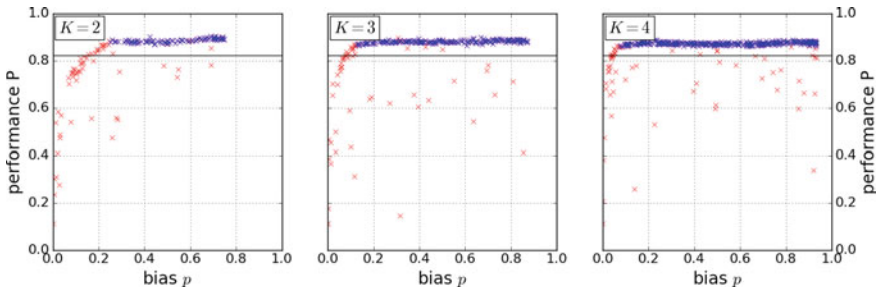

**Fig. 15** The classification performance for various network parameters. The black line at $P \approx 0.82$ indicates the score of the linear classifier for comparison. Blue symbols indicate unmodified networks and red symbols indicate optimizer-pruned networks

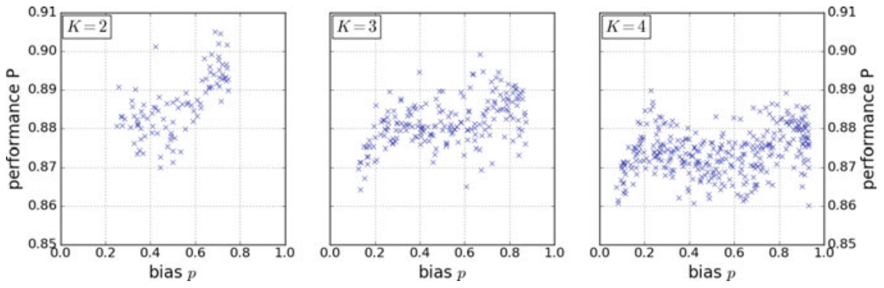

**Fig. 16** Higher resolution plots of the data shown in Fig. 15, but only including the results from full (un-pruned) networks

is comparable to the peak sizes. Even within the middle of the chaotic domain, we find some networks that perform nearly as well as the best performers near the ordered-chaotic boundaries, especially for $K = 3$ and 4.

### 4.3.3 RC Performance Dependence on Output Data Sample Size

We find that it is not necessary to use the data characterizing the network dynamics $\mathrm{X}(t)$ collected over the entire 200 time steps. As above, we use the cross-validation method but with the smaller recorded network data. Figure 17 shows the dependence of the performance on the number of time steps used in the training where we always start with the first time step. The performance saturates only after ~10 time steps (corresponding to a interval of only 50 ns). Note that all nodes are connected to the input layer and hence are activated immediately, but information injected to a node does not fully spread fully throughout the network because of the substantial link-time delay (~13 ns). This suggests that the recurrent connections, which allow for arbitrarily long loops in the reservoir, may not be as important for the MNIST classification task.

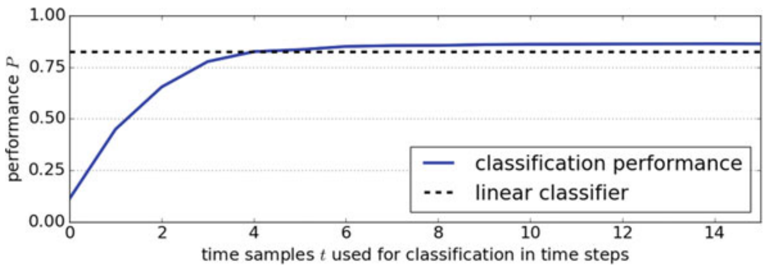

**Fig. 17** Performance score as a function of the length of the data record used for training the RC for $K = 2$ and $p = 0.222$. The black dashed line represents the performance of a linear classifier

**Table 1** Observed RC best performance on the MNIST task

| $K$ | $P_K$ (%) | Optimum $p$ |
|---|---|---|
| 2 | 90.5 | $p = 0.688$ |
| 3 | 89.9 | $p = 0.668$ |
| 4 | 89.0 | $p = 0.229$ |

### 4.3.4 Best Results

After an exhaustive search over network parameters, we find the highest performance over all $p$, finding the results given in Table 1. For comparison, we find that a linear classifier (no reservoir) has $P \approx 82.4\%$ for all $K$ and thus the RC has better performance in all cases. For a linear classifier with the full $28\times28$ image, LeCun et al. (1998) obtained $P = 88\%$. Our results with a compressed image still outperform a full-image linear classifier. Furthermore, the image data is processed by the reservoir within 1 μs, suggesting high-speed prediction could be possible if the weights are applied in real time on the FPGA (after off-line completion of the training of the output weights), which we will explore in future studies. The best performing networks occurred for $p$ near the ordered-chaotic transition, although the dependence on $p$ is weak as discussed in Sect. 4.3.2 above.

The performance in a typical classification experiment is about 90%. In particular, the classification results were quite different for each written digit. Figure 18a shows the average classification accuracy $P_i$ for each written digit $i$ over all experiments. When the classification fails (that is, the actual digit injected into the RC is misclassified), the RC will incorrectly classify the data as a different digit. Figure 18b shows the probability that the RC selects a digit for the case when it mis-classified the image.

The digit '1' is classified with the highest accuracy with a score of $P_1 = 94.6\%$, while digit '5' has the worst result of $P_5 = 76.0\%$, nearly a 20% difference from the best digit. Surprisingly, digit '1' is selected with much higher probability when the RC fails to correctly classify an image. The digit selected least during a misclassification are '0,' '2,' and '6.'

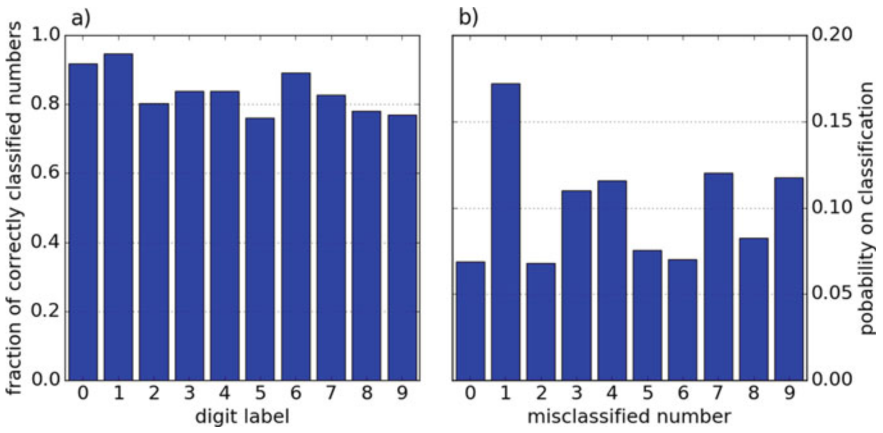

**Fig. 18** **a** Average classification accuracy over all experiments and **b** mis-classification of the written-digit images

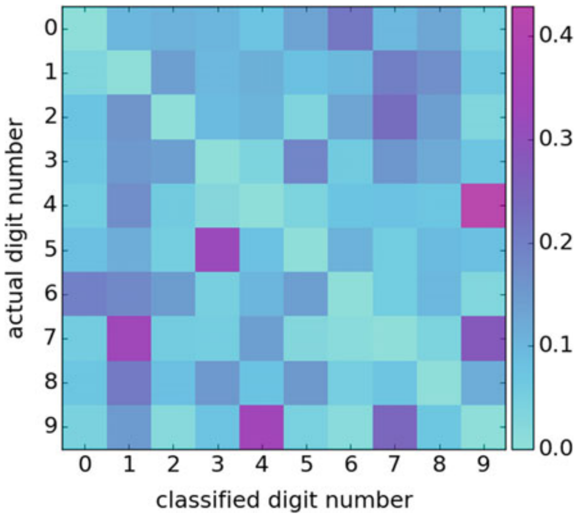

**Fig. 19** The confusion matrix visualizing the error for each digit

To go deeper into understanding the errors, Fig. 19 shows the confusion matrix quantifying how often a certain digit is mis-identified as a function of the input digit. Here, the rows of the matrix are normalized to one; in each row, a blue color indicates that this digit is hardly chosen while magenta pixels contribute the most to the error probability.

From the confusion matrix, we see that if a digit is mis-classified as another digit with high likelihood, the same can be said about the other digit. The most obvious mis-classification pairs are $4 \leftrightarrow 9$, $7\leftrightarrow9$, $1\leftrightarrow7$, and $3\leftrightarrow5$. Given the similarities of the strokes to write these numbers, their confusion is sensible. However, it is surprising that the pair $3\leftrightarrow8$ is not pronounced.

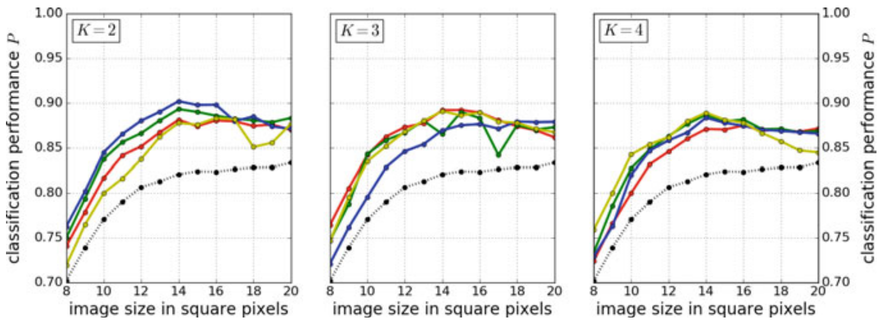

**Fig. 20** Classification performance variation with the number of pixels representing the written digit. The colored lines are for 4 randomly chosen reservoirs, while the black line is for the linear classifier. The parameters are as follows: $K = 2$, $p = 0.5$ (red), 0.566 (green), 0.719 (blue), 0.352 (yellow); $K = 3$, $p = 0.758$ (red), 0.521 (green), 0.170 (blue), 0.482 (yellow); and $K = 4$, $p = 0.176$ (red), 0.244 (green), 0.297 (blue), 0.743 (yellow)

We now turn to investigate the influence of the image size on the performance. For images compressed to a very small number of pixels, there is loss of information because the coarse representation will effectively merge features in the written digit. On the other hand, once the image has $\sim 13 \times 13$ pixels, some of the data input to the reservoir is lost because of the shortcoming of the way in which we inject data into the reservoir as discussed above in Sect. 3.2. Figure 20 shows the performance of four different randomly chosen reservoirs for each connectivity $K$. In addition, we show the performance of the linear classifier for the same data sets.

We see that the FPGA-based reservoirs always outperform the linear classifier for all image sizes. As expected based on increased image resolution, the performance of the linear classifier increased monotonically but there is a noticeable decrease in the slope beyond $\sim 12 \times 12$-pixel image size. On the other hand, the RC classifier performance peaks around a $\sim 14 \times 14$-pixel image size and then decreases somewhat. This decrease is likely due to the fact that only part of these images are input to the reservoir by our method for data injection as mentioned above and discussed in Sect. 3.2, a limitation we will address in future studies. However, for all cases, the performance is above the linear classifier.

### 4.3.5 Parallel RCs

One method to possibly improve the performance of our reservoir computer is to use multiple reservoirs, which we explore here in some preliminary studies. Similar to deep learning artificial neural networks, a hierarchical structure can allow for forecasting dynamical systems with different time scales or to focus the network on different temporal or spatial features (Jaeger 2007). Also, a serial cascaded of RCs improves the chaotic time series prediction performance (Webb 2008). A recent review summarizes variations on these architectures (Tanaka et al. 2018).

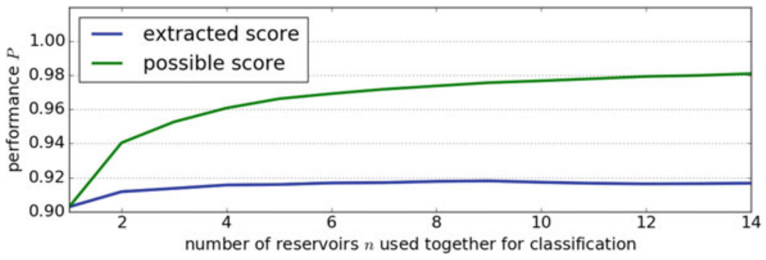

**Fig. 21** Classification performance for parallel RCs that each classify all written digits (blue line) and the best combination of reservoirs where each only classifies a subset of the digits (green line)

Specific to the MNIST task, Jalalvand et al. (2015) showed improved performance of an RC using multiple RCs in a parallel, serial, or parallel-serial geometries. Their work is based on a software simulation with each RC having ∼10,000 reservoir nodes, but only feeding in a single column of pixel data from the original MNIST 28×28-pixel images. It is not apparent whether a similar approach will work here where we consider using ABNs with link time delays, whereas (Jalalvand et al. 2015) considers nodes with a hyperbolic tangent nonlinearity and no link delays. There is some hope for an improvement using a parallel approach because specific reservoir realizations perform somewhat better on some digits than others. We use the data collected from the different reservoir realizations in a parallel configuration with $\ell$ RCs where we choose the $\ell$ best reservoirs. The results are combined using a majority vote for classification. As seen in Fig. 21, there is a modest increase in the performance, peaking at $P = 91.8\%$ with $\ell \sim 8$.

To obtain a sense of a possible optimal FPGA-based reservoir with 64 nodes, we search through our database of results to find the $\ell$ reservoirs that perform the best for a single digit. The performance of this hypothetical machine is shown by the green line. Better performance can likely only be obtained by increasing the various resources as discussed in the next section.

## 5 Discussion and Conclusions

In this chapter, we demonstrate that an FPGA-based RC can perform the written-digit classification task. The performance over a wide range of reservoir parameters exceeds that of a linear classifier, although the error rate is still relatively high in comparison to the state-of-the-art deep learning feed-forward artificial neural networks with fully trained weights. One way to improve the performance of our RC is to increase the number of connections of the input data to the reservoir and to increase the number of reservoir nodes; in other related studies, we find that the performance scales approximately as the square root of the number of input connections and reservoir nodes. In the future, we will also investigate full on-chip training and classification (Yi et al. 2016), which should lead to extremely fast operation of an RC. The FPGA platform is an especially appealing RC physical substrate because it is commercially available, operates at low power, runs at high speed, easily integrated with traditional computer infrastructure, and can potentially lead to beyond-Turing computing because we run the reservoir in the autonomous mode.

**Acknowledgements** S.A. and E.S. gratefully acknowledge the financial support of the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—Projektnummer 163436311- SFB 910. N.D.H., O.D., and D.J.G. gratefully acknowledge financial support from the U.S. Army Research Office through Grant No. W911NF-12-1-0099. O.D. also gratefully acknowledges financial support from the European Union's Horizon 2020 research and innovation program under the Marie Skłodowska-Curie grant agreement No. 713694. We gratefully acknowledge the IT support of Jimmy Dorff of the Duke University Department of Physics for extensive help in developing the server to automate our experiments.

## Appendix: FPGAs and project workflow

### Brief introduction to FPGAs

An extensive discussion on using FPGAs for experiments on autonomous Boolean networks, including hardware description language metacode, can be found in Rosin (2015), D'Huys et al. (2016), Lohmann et al. (2017), and Canaday et al. (2018). Briefly, FPGAs contain more than $10^5$ programmable logic elements (LE) that can be linked via programmable connections. The logic elements are based on CMOS technology and are arranged in a grid system. Each of these elements consists of a look-up table (LUT) with four inputs for the Altera Cyclone IV chip family used here (EP4CE115F29C7N, Terasic DE2-115 demonstration board), a flip-flop and a multiplexer. Figure 22 illustrates the layout of a logic element.

The multiplexer is used to switch between clocked and unclocked (autonomous) mode and the flip-flop provides the clocked output. The look-up table can be used to define an operation for $K$ inputs, so having a length of $2^K$ bits and thus can be

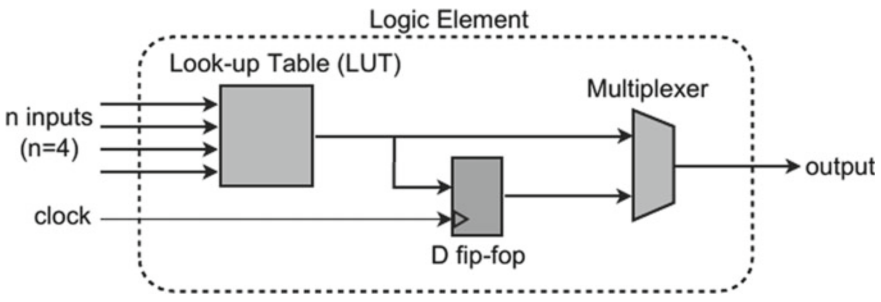

**Fig. 22** Structure of a single logic element on the FPGA. The figure is adapted from Rosin (2015)

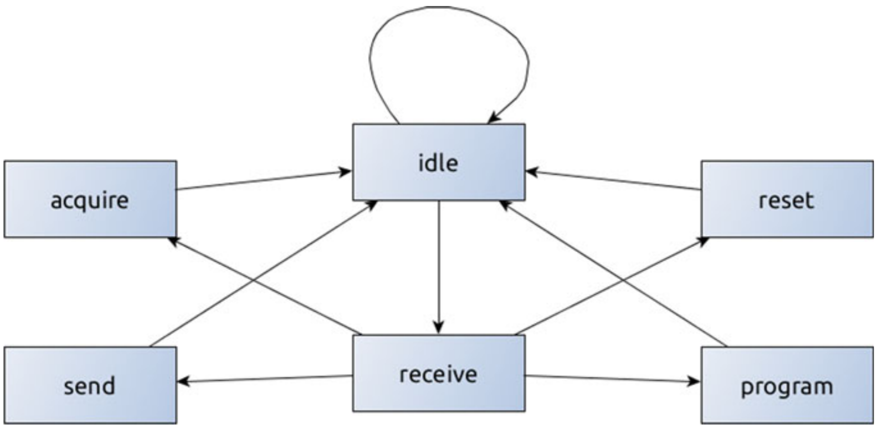

**Fig. 23** Illustration of the transition between the six states of the FSM

used to realize up to $2^{2^K}$ different Boolean functions. The Altera Cyclone IV logic elements used in this FPGA have a maximum input number of $K = 4$ so a total number of 65,536 different operations can be performed by a single LE. This makes it easy to create an arbitrary setup on the device. Their relatively low costs and their operation time on the nanosecond time scale are ideally suited for reservoir computing applications.

The network is defined using the hardware description language *Verilog*, which instructs the FPGA how to configure itself, including the function of the nodes and the network links. This code is compiled using the proprietary software *Quartus II*, version 14.0 (Quartus 2021). The resulting bitstream is loaded onto the FPGA, which causes the configuration of the RC. See the references above for details. Unique to this project is the way in which the input and output state data are handled and interfaced with the autonomous network forming the reservoir. We use a finite-state machine (FSM) synthesized from clocked logic and communicates with a personal computer (PC) via a USB controller chip (FTDI part #FT232H), which is connected directly to the FPGA pins via the GPIO connector on the demonstration board.

The FSM has the distinct states: `idle`, `receive`, `acquire`, `send`, `program`, and `reset` whose possible transitions are depicted in Fig. 23. It starts in the `idle` state and remains without any action as long as it does not receive any input over the USB connection via the PC. After a signal is detected on the USB, the FSM changes to the `receive` state. Next, the first available byte read from the USB determines the next state of the FSM. The `program` state reads a number of bytes from the USB connection and stores it onto the RAM onboard the chip. This number is set in the *Verilog* code defining the reservoir (see below) and can only be changed by recompiling the *Verilog* code. The data transferred to the FPGA in this manner sets the initial state for a reservoir.

When the state changes to `acquire`, all nodes in the reservoir are set to the value `zero` and the data from the RAM that was transferred during the `program` process is fed into the reservoir. Because the FSM operates with clocked logic, the initial state is inserted over one clock cycle with the length of 5 ns (corresponding to a data rate of 200 MHz). After this data insertion, the reservoir runs freely and autonomously. The only restrictions for the dynamics arise from the finite response time of electric elements within the FPGA. On every clock cycle (5 ns period), a snapshot of the reservoir is taken by saving the current value of each node to the RAM. In this fashion, a series of 200 samples is captured and stored. Note that the dynamics in between clock cycles can't be measured using this method.

After all data is collected, the FSM is set to the `send` state, which will send all the data stored in the RAM to the USB connection and is available to a Python program running on the PC. The same reservoir can be used to collect data for the same input word by sending several `acquire` commands or a new input word can be used just by changing the initial-state RAM contents without having to recompile the *Verilog* code and transferring the resulting bitstream to the FPGA.

The *Verilog* code for the FSM has not been presented in our previous publications so we give the module below for completeness.

```
/*
Master FSM monitors bytes sent from the PC to
look for command signals
Valid command signals are:
  Begin aquiring data
  Send data from RAM to PC
  Program input words
  Reset
When a valid command is recognized, the FSM flips
a signal bit that other modules are monitoring
*/

module master_fsm(CLOCK_50, KEY, rcvd_sig,
  byte_from_pc, acquire_signal, send_signal,
  prog_word_signal, reset);

// Parameters
parameter IDLE = 0, RCVD = 1, ACQD_I = 2,
 ACQD_II = 3, SEND_I = 4, SEND_II = 5, PROG_I = 6,
 PROG_II = 7, RST_I = 8, RST_II = 9;

// Internal elements
input CLOCK_50;
input KEY;
input rcvd_sig;
input [7:0] byte_from_pc;

```

```
output acquire_signal;
output send_signal;
output prog_word_signal;
output reset;

reg [3:0] state;
reg [5:0] count;

wire hardware_reset;

reg acq_sig, send_sig, prog_sig, rst_sig;
assign acquire_signal = acq_sig;
assign send_signal = send_sig;
assign prog_word_signal = prog_sig;
assign hardware_reset = ~KEY;
assign reset = hardware_reset | rst_sig;

// The finite-state machine
always @(*)
   begin
      case (state)
         IDLE:
            begin
               acq_sig <= 1'b0;
               send_sig <= 1'b0;
               prog_sig <= 1'b0;
               rst_sig <= 1'b0;
            end
         RCVD:
            begin
               acq_sig <= 1'b0;
               send_sig <= 1'b0;
               prog_sig <= 1'b0;
               rst_sig <= 1'b0;
            end
         ACQD_I:
            begin
               acq_sig <= 1'b1;
               send_sig <= 1'b0;
               prog_sig <= 1'b0;
               rst_sig <= 1'b0;
            end
         ACQD_II:
            begin
               acq_sig <= 1'b1;
               send_sig <= 1’b0;
               prog_sig <= 1’b0;
               rst_sig <= 1’b0;
            end
         SEND_I:
            begin
               acq_sig <= 1’b0;
               send_sig <= 1’b1;
               prog_sig <= 1’b0;
               st_sig <= 1’b0;
            end
         SEND_II:
            begin
               acq_sig <= 1’b0;
               send_sig <= 1’b1;
               prog_sig <= 1’b0;
               rst_sig <= 1’b0;
            end
         PROG_I:
            begin
               acq_sig <= 1’b0;
               send_sig <= 1’b0;
               prog_sig <= 1’b1;
               rst_sig <= 1’b0;
            end
         PROG_II:
            begin
               acq_sig <= 1’b0;
               send_sig <= 1’b0;
               prog_sig <= 1’b1;
               rst_sig <= 1’b0;
            end
         RST_I:
            begin
               acq_sig <= 1’b0;
               send_sig <= 1’b0;
               prog_sig <= 1’b0;
               rst_sig <= 1’b1;
            end
         RST_II:
            begin
               acq_sig <= 1’b0;
               send_sig <= 1’b0;
               prog_sig <= 1’b0;
               rst_sig <= 1’b1;
            end
         default:
            begin
               acq_sig <= 1'b0;
               send_sig <= 1'b0;
               prog_sig <= 1'b0;
               rst_sig <= 1'b0;
            end
      endcase
   end

always @(posedge CLOCK_50 or posedge hardware_reset)
   begin
      if (hardware_reset)
         state <= IDLE;
      else
         case (state)
            IDLE:
               begin
                  if (rcvd_sig)
                     state <= RCVD;
                  else
                     state <= IDLE;
               end
            RCVD:
               begin
                  case (byte_from_pc[7:0])
                     8'b00000001: state <= ACQD_I;
                     8'b00000010: state <= SEND_I;
                     8'b00001000: state <= PROG_I;
                     8'b00010000: state <= RST_I;
                     default:
                        state <= IDLE;
                  endcase
               end
            ACQD_I:
               state <= ACQD_II;
            ACQD_II:
               state <= IDLE;
            SEND_I:
               state <= SEND_II;
            SEND_II:
               state <= IDLE;
            PROG_I:
               state <= PROG_II;
            PROG_II:
              state <= IDLE;
            RST_I:
               state <= RST_II;
            RST_II:
               state <= IDLE;
            default:
               state <= IDLE;
         endcase
   end

endmodule
```

**Experimental pipeline**

Creating networks, running the experiments, and analyzing the results presented in this chapter was simplified by automating the process with the following feature specifications:

- The system runs in an automated fashion with as few manual steps as possible
- The system is separated into the distinct steps: creating the network; compiling the network; running the experiment; and analyzing the data
- Scalability: Each individual step has any number of computers that can be added or removed to or from the system at any time to run tasks in parallel
- Extensibility: New methods for running experiments or analyzing data can be added at any time and all connected machines will automatically receive all updates
- Supervision: The system is accompanied by a website that always shows the current state of all ongoing actions

The code for this project is extensive; we provide it on a repository on an as-is basis for those interested in exploring or extending the system (GitHub 2021).

The entire system is implemented in Python and runs on a centralized server set up for us by the Duke University Physics Department IT staff. The core of the system is based on a MySQL database that provides tables for *reservoir*, *experiment* and *analysis*. All tables contain the data unique to each step of the workflow, such as the number of reservoir nodes, number of delay elements, network connectivity, number of written-image input words, and experimental repetitions. The pipeline is controlled by two additional fields: *status* and *tag*, which are assigned to all items. The status is an integer number that indicates open, running, successful, error, or canceled. All sub-systems have a unique identifier that creates a *tag* on each item, which allows us to determine which machine creates a certain item. The run-time order is usually random and depends on the internal ordering of the database. If some items are preferred over others, we assign a *priority* tag to force a schedule order.

It is challenging to interact with the *Quartus II* system to automate creating different reservoirs in an automated manner. Therefore, this step was completed by a user who creates a new python script to add new reservoirs. This process was facilitated as much as possible by providing many functions to create random nodes and links and assign a bias to node functions. Finally, the *Verilog* code defining the reservoir is created with appropriate metadata to identify the reservoir.

After uploading a compiled bitstrea file, a new entry is inserted into the database with a unique *id* and all attributes extracted from the reservoir file. The new item is created as *status* open and has no tag assigned. For this step, a compile daemon checks for new reservoirs, specifically looking for *status* open. If a new item is found, it is updated to *status* running with the *tag* of the machine that claimed the reservoir, thus preventing any other machine from taking the same item. The daemon then downloads the reservoir file from the server and uses the *Quartus II* software to compile it. If the process ends successfully, the new file is then uploaded back to the server and the *status* becomes successful. Otherwise, the *status* is set to error and a file containing the error message is uploaded instead.

To run experiments, an input creation function must exist. This function defines the initial values of the network nodes. If a new function is required by a user, it is developed and added to the system. Developing the project is managed by the versioning software *git*, which has its repository on the Duke University centralized data storage system.

Creating a new experiment is undertaken by inserting the experimental definition for an existing reservoir with input method and all required parameters into the database with *status* open. An experiment daemon similar to reservoir daemon checks for new experiments that have their *status* open, but additionally requires the used reservoir to have *status* successful. It then assigns *status* running and sets the tag identifier. If a new experiment is found, the daemon checks if an update has been made to the code via the *git* repository and updates if necessary. Next, the input creation function is called with all parameters and returns the defined number of different input words. These initialize the network and each input is run with the set repetition number. The entire data is stored in a single binary file that is uploaded if the collection succeeds. An error message is uploaded in case of an error.

To analyze the data, we use the Open Science Grid (OSG) (OSG 2021), which is a giant computer cluster that includes machines from many universities all over the United States of America. This platform allows us to start hundreds of jobs at the same time and distributes the work to all connected machines, thus providing massive computational power. However, this resource is not always reliable and sometimes fails.

To account for this possibility, we use an analysis daemon that runs on local machines as a fall back. All analysis code is developed in advance and synchronized with *git* repository. The local and OSG daemons check for open analyses that have successful experiments as their predecessor, and verifies that the latest version of the code is in use. For the OSG, the data is uploaded and a configuration file sent with all analysis parameters. When the analysis completes successfully, the results files are uploaded back to the Duke University data storage system and all data on the OSG belonging to this analysis is deleted. In case of an error, the data remains on the OSG for further investigation. In any case, the *status* is set to the result.

## References

M.L. Alomar, M.C. Soriano, M. Escalona-Morán, V. Canals, I. Fischer, C.R. Mirasso, J.L. Rosselló, Digital implementation of a single dynamical node reservoir computer. IEEE Trans. Circuits Syst. II: Exp. Briefs **62**, 977 (2015)

P. Antonik, *Application of FPGA to Real-Time Machine Learning* (Springer, Cham, 2018)

P. Antonik, A. Smerieri, F. Duport, M. Haelterman, S. Massar, FPGA implementation of reservoir computing with online learning, in *24th Belgian-Dutch Conference on Machine Learning (Benelearn)*, Benelearn, vol. 24, 19 June 2015, Delft, Netherlands (2015)

S. Apostel, Dynamics of driven complex autonomous Boolean networks with application to reservoir computing. M.S. thesis, Technische Universität Berlin (2017). Unpublished

L. Appeltant, M.C. Soriano, G. Van der Sande, J. Danckaert, S. Massar, J. Dambre, B. Schrauwen, C.R. Mirasso, I. Fischer, Information processing using a single dynamical node as complex system. Nat. Commun. **2**, 468 (2011)

L. Appeltant, G. Van der Sande, J. Danckaert, I. Fischer, Constructing optimized binary masks for reservoir computing with delay systems. Sci. Rep. **4**, 3629 (2014)

D. Canaday, A. Griffith, D.J. Gauthier, Rapid time series prediction with a hardware-based reservoir computer. Chaos **28** (2018)

F. Denis-Le Coarer, M. Sciamanna, A. Katumba, M. Freiberger, J. Dambre, P. Bienstman, D. Rontani. All-optical reservoir computing on a photonic chip using silicon-based ring resonators. IEEE J. Sel. Top. Quantum Electron. **24**, 1–8 (2018)

B. Derrida, Y. Pomeau, Random networks of automata: a simple annealed approximation. Europhys. Lett. **1**, 45 (1986)

O. D'Huys, J. Lohmann, N.D. Haynes, D.J. Gauthier, Super-transient scaling in time-delay autonomous Boolean network motifs. Chaos **26**, 094810 (2016)

R. Edwards, L. Glass, A calculus for relating the dynamics and structure of complex biological networks, in *Adventures in Chemical Physics: A Special Volume of Advances in Chemical Physics*, ed. by R.S. Berry, J. Jortner (John Wiley & Sons, Inc., Hoboken, 2006), pp. 151–178

R. Edwards, P. van den Driessche, L. Wang, Periodicity in piecewise-linear switching networks with delay. J. Math. Biol. **55**, 271 (2007)

H. Flyvbjerg, An order parameter for networks of automata. J. Phys. A. **21**, L955 (1988)

D.J. Gauthier, Reservoir computing: harnessing a universal dynamical system. SIAM News **51**(2), 12 (2018)

M. Ghil, A. Mullhaupt, Boolean delay equations. II. Periodic and aperiodic solutions. J. Stat. Phys. **41**, 125 (1985)

M. Ghil, I. Zaliapin, B. Coluzzi, Boolean delay equations: a simple way of looking at complex systems. Phys. D **237**, 2967 (2008)

GitHub (2021), https://github.com/nickdavidhaynes/boolean-reservoir-computer

N.D. Haynes, M.C. Soriano, D.P. Rosin, I. Fischer, D.J. Gauthier, Reservoir computing with a single time-delay autonomous Boolean node. Phys. Rev. E **91**, 020801 (2015)

H. Jaeger, Discovering multiscale dynamical features with hierarchical echo state networks. Technical report 10, School of Engineering and Science, Jacobs University (2007). Unpublished

H. Jaeger, H. Haas, Harnessing nonlinearity: predicting chaotic systems and saving energy in wireless communication. Science **304**, 78 (2004)

A. Jalalvand, K. Demuynck, W. De Neve, R. Van de Walle, J.-P. Martens. design of reservoir computing systems for noise-robust speech and handwriting recognition, in *Conference on Graphics, Patterns and Images (SIBGRAPI)*, vol. 28. Salvador. Porto Alegre: Sociedade Brasileira de Computação (2015). On-line. IBI: 8JMKD3MGPBW34M/3JUJ5DP, http://urlib.net/rep/8JMKD3MGPBW34M/3JUJ5DP

A. Katumba, J. Heyvaert, B. Schneider, S. Uvin, J. Dambre, P. Bienstman, Low-loss photonic reservoir computing with multimode photonic integrated circuits. Sci. Rep. **8**, 2653 (2018)

L. Larger, A. Baylóon-Fuentes, R. Martinenghi, V.S. Udaltsov, Y.K. Chembo, M. Jacquot, High-speed photonic reservoir computing using a time-delay-based architecture: million words per second classification. Phys. Rev. X **7**, 011015 (2017)

Y. Lecun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition. Proc. IEEE **86**, 2278 (1998)

J. Lohmann, O. D'Huys, N.D. Haynes, E. Schöll, D.J. Gauthier, Transient dynamics and their control in time-delay autonomous Boolean ring networks. Phys. Rev. E **95**, 022211 (2017)

B. Luque, R.V. Solé, Lyapunov exponents in random Boolean networks. Phys. A **284**, 33 (2000)

J.P. Mason, P.S. Linsay, J.J. Collins, L. Glass, Evolving complex dynamics in electronic models of genetic networks. Chaos **14**, 707 (2004)

C. Mesaritakis, A. Bogris, A. Kapsalis, D. Syvridis, High-speed all-optical pattern recognition of dispersive Fourier images through a photonic reservoir computing subsystem. Opt. Lett. **40**, 3416–3419 (2015)

MNIST (2021), http://yann.lecun.com/exdb/mnist/

OSG (2021), https://opensciencegrid.org/

Y. Paquot, F. Dupart, A. Smerieri, J. Dambre, B. Schrauwen, M. Haelterman, S. Massar, Optoelectronics reservoir computing. Sci. Rep. **2**, 287 (2012)

B. Penkovsky, L. Larger, D. Brunner, Efficient design of hardware-enabled reservoir computing in FPGAs. J. Appl. Phys. **124**, 162101 (2018)

Quartus (2021), https://www.intel.com/content/www/us/en/programmable/downloads/download-center.html

A. Röhm, K. Lüdge, Multiplexed networks: reservoir computing with virtual and real nodes. J. Phys. Commun. **2**, 085007 (2018)

D.P. Rosin, *Dynamics of Complex Autonomous Boolean Networks* (Springer, Heidelberg, 2015)

E.S. Skibinsky-Gitlin, M.L. Alomar, C.F. Frasser, V. Canals, E. Isern, M. Roca, J.L. Rosselló, Cyclic reservoir computing with fpga devices for efficient channel equalization, in *Artificial Intelligence and Soft Computing. ICAISC 2018*, ed. by L. Rutkowski, R. Scherer, M. Korytkowski, W. Pedrycz, R. Tadeusiewicz, J. Zurada. Lecture Notes in Computer Science, vol. 10841 (Springer, Cham, 2018)

G. Tanaka, T. Yamane, J. B. Héroux, R. Nakane, N. Kanazawa, S. Takeda, H. Numata, D. Nakano, A. Hirose, Recent advances in physical reservoir computing: a review (2018), arXiv:1808.04962v2

A. Uchida, R. McAllister, R. Roy, Consistency of nonlinear system response to complex drive signals. Phys. Rev. Lett. **93**, 244102 (2004)

K. Vandoorne, P. Mechet, T. Van Vaerenbergh, M. Fiers, G. Morthier, D. Verstraeten, B. Schrauwen, J. Dambre, P. Bienstman, Experimental demonstration of reservoir computing on a silicon photonics chip. Nat. Commun. **5**, 3541 (2014)

Q. Vinckier, F. Duport, A. Smerieri, K. Vandoorne, P. Bienstman, M. Haelterman, S. Massar, High-performance photonic reservoir computer based on a coherently driven passive cavity. Optica **2**, 438–446 (2015)

R.Y. Webb, Multi-layer corrective cascade architecture for one-line predictive echo state networks. Appl. Artif. Intell. **22**, 811 (2008)

Y. Yi, Y. Liao, B. Wang, X. Fu, F. Shen, H. Hou, L. Liu, FPGA based spike-time dependent encoder and reservoir design in neuromorphic computing processors. Microprocess. Microsyst. **46B**, 175 (2016)

I.B. Yildiz, H. Jaeger, S.J. Kiebe, Re-visiting the echo state property. Neural Netw. **35**, 1 (2012)

R. Zhang, H.L.D. de S. Cavalcante, Z. Gao, D.J. Gauthier, J.E.S. Socolar, M.M. Adams, D.P. Lathrop, Boolean Chaos. Phys. Rev. E. **80**, 045202(R) (2009)

H. Zhang, Z. Feng, B. Li, Y. Wang, K. Cui, F. Lin, W. Dou, Y. Huang, Integrated photonic reservoir computing based on hierarchical time-multiplexing structure. Opt. Express. **22**, 31356–31370 (2014)