Joni Dambre

2 Information processing and computation with photonic reservoir systems

2.1 Introduction

2.1.1 The boundaries of digital computing

The recent surge of research into alternatives for conventional (mostly digital) CMOS-based computing hardware is pushed forward by the limits of scalability. The fundamental assumptions, upon which the success of digital computing is based are mainly its robustness, resulting in extremely low error probabilities, and its inherently low static power consumption, both of which break down for very small devices. The digital computing paradigm and the associated design methodology rely on (almost) error-free operation and cannot deal with *inaccurate* devices.

Already in 1956, Von Neumann discussed how to approximate high-precision digital computing with unreliable components [1] by introducing redundancy and *majority voting*. However, the bounds he derives to prove that with enough redundancy, error probabilities can be pushed below any desired threshold, are based on the assumption that errors occur independently. When the errors are correlated, as is usually the case in real life, an ensemble rarely performs worse than the individual models, but the convergence of the accuracy is no longer guaranteed. In practice, taking an *ensemble* of unreliable models is now common practice in *machine learning*.

The second most important property to break down is the fact that, using CMOS, we can build implementations of digital gates that consume very little static power. As feature dimensions and isolation layer thickness get thinner, MOSFET transistors and CMOS gates start leaking current in all directions. Whereas the power consumption of a computer was long considered an unimportant issue, it has now become more important than the speed of computation. A few powerful GPUs in the room are an excellent replacement for other heating devices, but our hunger for ever-increasing and ubiquitous computing does not fade, especially in the presence of the huge leaps that are being made with AI and deep learning.

Many early attempts of computing with alternative devices stay within the digital model of computation. Clearly, this has its benefits: if you can build gates and flip-flops, the whole design methodology can stay in place and the chances of industrial uptake of your new technology improve dramatically. However, thus far, no mapping between inherently analog devices and the basic digital building blocks (binary gates, binary gated memory cells) has been found that can sufficiently outperform transistors on at least one dimension of performance (size, power consumption per operation or power density, speed) without overly compromising the others and at the same time

is commercially viable in a mass production setting. Until now, the proposed devices were either too large, too slow, too power hungry, too inaccurate or too difficult to produce, or did not offer sufficient benefits to trigger a change of course in an entire industry built around integrated CMOS circuits.

If we take a step backwards, the digital computing model is maybe no longer the most efficient. This is especially true for the huge amount of information processing that handles signals, e. g., image processing and sensor processing, where, essentially real numbers are translated into real numbers and small errors are usually acceptable. From an energy perspective, it is no longer the most efficient to represent analog values as bit sequences and to use analog devices to mimic theoretical mathematical models known as digital gates and memory cells. The field of research that addresses ways to optimally exploit relaxed precision requirements using more or less traditional computing devices is generally called *approximate* computation [2, 3]. It mostly proposes incremental changes to traditional design flows at different design levels (algorithms, compilers, synthesis, and devices). Different authors often target different performance goals, although the general focus is on power efficiency.

2.1.2 Analogue computing

The alternative to digital computing is to compute directly in the analogue domain. Unfortunately, for general-purpose (or broad-purpose) analogue computing, useful combinations of a clear-cut computational model and an automated hardware design methodology are rare. One approach is to move to *neural networks* as a computational model. Their design is based on *learning from examples* and tuning parameters to minimize a given cost function. They are usually used as machine learning *software models* that are run on digital computers to approximate input-output relationships for which no perfect solution is known or does even exist.

Many variants exist within the broad category of neural networks, and for some, their implementation can also be realized with analogue building blocks. From the artificial ones used in machine learning to the biologically plausible ones studied in computational neuroscience, all are computational models which come with ways to tune their parameters (weights). Essentially, they are extremely powerful *function approximators*. Some types of neural networks can be proven to be universal approximators [4, 5], which means that they can approximate a very broad class of input-output relationships with any desired precision if they are allowed to grow large enough. In practice, they are inherently imperfect, i. e., even the most powerful neural networks make mistakes. These can be frequent but small, or very infrequent but very large. Depending on the application, either one can be worse than the other. Since neural network training and evaluation is usually based on the average quality of the output across a large number of examples, only detailed analysis of the errors made can make the distinction.

In general, it is useful to make a distinction between the computational model that is used and the physical medium in which this model is implemented [6]. The mapping between both should result in a set of computational elements or building blocks that can be realized in the chosen medium (with guarantees on all aspects of their performance) and a methodology for combining them into a system that realizes the desired behavior. From this perspective, it is very helpful if the requirements for these building blocks are not very rigid. It is this rigidity in digital computation that makes it so difficult to realize digital building blocks from analogue devices. In comparison, in neural networks, we need to be able to compute a weighted sum of signals at the neuron inputs, but the exact behavior of the neurons turns out not to be very important for their performance. In artificial neural networks, several different functions are already being used, most notably: sigmoidal, exponential, stepwise linear, and Gaussian. Exactly and efficiently matching those with analogue devices is also not trivial, and the same is true for the building blocks used in the models from computational neuroscience. However, it turns out that most input-output relationships that resembles them should give rise to good performance. In fact, this robustness of the neural network computational model is one of the historical reasons for the emergence of *physical reservoir computing* and *photonic reservoir computing*, the topic of this book, from computational models that are closely related to neural networks.

A crucial aspect for the success of analogue computing is the availability of a design methodology that can be automated to provide the mapping between a desired behavior to (a hierarchical composition of) the computational elements of the model and from those to the physical medium. Clearly, for digital computing, this is in place. Therefore, in order for any new hardware approach to become competitive, this mapping should outperform the existing solutions according to at least one quality measure (e. g., speed, power consumption, size, accuracy, noise-robustness) without compromising too much on the others.

2.2 Reservoir computing

2.2.1 A more relaxed model of computation

In this book, we describe recent work that uses *physical reservoir computing* [7] to design photonic computational devices. This is an altogether different approach to computation, in which no abstract computational model is enforced onto the implementation substrate. Instead, the natural dynamics of the substrate are exploited and combined to approximate the desired input-output relationships of a given computational task.

Instead of in the computational model, the constraints are now in the optimization approach, which is based on the principles of *reservoir computing*. In its broadest

Figure 2.1: Schematic representation of a reservoir computing system. The input signal $u(t)$ is fed into the reservoir and the resulting reservoir states $x(t)$ are used to learn a linear readout that is then used to generate the output signal $y(t)$.

interpretation, reservoir computing consists of two parts: a *reservoir* and a *readout* as depicted in Figure 2.1. The reservoir is a dynamical system that is perturbed (*driven*) by the input signal $u(t)$, which affects its current internal state $x(t)$ as well as the future evolution of that state. In the readout, the reservoir's internal state is observed and a linear combination of the observations is optimized to optimally approximate a desired output signal. In contrast to the readout, the reservoir itself is not changed in this stage, although a few global system parameters are usually tuned during design to make the overall dynamics of the reservoir more suitable for the task.

The reservoir computing approach emerged almost simultaneously from the fields of neuroscience and artificial neural networks around the end of the previous and the beginning of this century. The two most frequently cited foundational works are those of Herbert Jaeger [8, 9] and Wolfgang Maass [10]. Jaeger and Maass used *echo state networks* (ESN) and *liquid state machines* (LSM), respectively, as their reservoirs. Both are simulated recurrent neural networks, operating in discrete or discretized time. ESNs consist of discrete-time analogue sigmoidal or hyperbolic tangent neurons. The network they form is usually fully connected, and the input signals are also connected to all nodes. The connection weights are randomly initialized. LSMs consist of (simple models of) biological spiking neurons. In both cases, the network's internal state space is finite and consists of all neuron outputs. Overviews of the progress in reservoir computing can be found in [11–14].

2.2.2 How to train a reservoir computer

In summary, a reservoir computing system consists of a reservoir, which transforms the input signal(s) into features that depend on the input history, and a readout layer. In most work on reservoir computing, this is a simple linear regression layer, which is trained by minimizing the mean squared error (equation (1))

$$\text{MSE}(\mathbf{y}, \hat{\mathbf{y}}) = E[(\mathbf{y} - \hat{\mathbf{y}})^2] \quad (1)$$

between the generated output sequences and the desired output sequences for a set of examples (where E indicates the expected value).

In order to minimize the MSE, we minimize its approximation based on one or multiple input sequences of finite length. For this purpose, the reservoir is driven with

the input sequence(s).¹ The N observed reservoir state signals are sampled in time (leading to S samples for each of the signals) and recorded into the $S \times (N + 1)$ *augmented state matrix* $\tilde{\mathbf{X}}$, in which the first N columns contain the N state signals, and the last column is an all-ones column. Using the notation $\tilde{\mathbf{w}}_{\text{out}} = (\mathbf{w}_{\text{out}}, w_0)$, the optimal output weights are then found by the normal form:

$$\tilde{\mathbf{w}}_{\text{out}} = (\tilde{\mathbf{X}}^T \tilde{\mathbf{X}})^{-1} \tilde{\mathbf{X}}^T \mathbf{y}, \quad (2)$$

yielding a closed-form expression for the approximated target signal:

$$\hat{\mathbf{y}} = \tilde{\mathbf{X}} \tilde{\mathbf{w}}_{\text{out}} = \tilde{\mathbf{X}} (\tilde{\mathbf{X}}^T \tilde{\mathbf{X}})^{-1} \tilde{\mathbf{X}}^T \mathbf{y}. \quad (3)$$

In machine learning, it is generally advised to use some form of regularization to avoid overfitting to the training data. The most common approach is to use *ridge regression*. This minimizes the augmented cost function:

$$\text{MSE}_{\text{ridge}} = \text{MSE} + \lambda \mathbf{w}_{\text{out}}^T \mathbf{w}_{\text{out}}. \quad (4)$$

The intuition behind this is the fact that keeping the weights small constrains the complexity of the model. Note that w_0 should not be included in the regularisation term as it is needed to set the average value of the approximated target signal to its correct value.

When using ridge regression, it is generally advisable to normalize the state signals before applying linear regression, to avoid the regularization from overly penalizing signals with small magnitude. In this case, we use

$$\mathbf{x}'[n] = \frac{\mathbf{x}[n] - \text{avg}(\mathbf{x}[n])}{\text{stdev}(\mathbf{x}[n])}$$

instead (and similarly construct the corresponding state matrix $\mathbf{X}'$). Augmenting the state matrix with an all-ones column is no longer necessary in that case, since any linear combination of the recorded state signals will have an average value of 0 and the optimal value of w_0 is simply given by

$$w_0 = \frac{1}{n_{\text{max}}} \sum_{n=1}^{n_{\text{max}}} \mathbf{y}[n].$$

The optimal readout weights $\mathbf{w}_{\text{out}}$ are then given by

$$\mathbf{w}_{\text{out}} = (\mathbf{X}'^T \mathbf{X}' + \lambda \mathbf{I})^{-1} \mathbf{X}'^T \mathbf{y} \quad (5)$$

¹ Usually, in order to reduce the impact of the initial state, the first part of each sequence is ignored for both training and evaluation.

where $\mathbf{I}$ is the identity matrix and $\mathbf{X}^T \mathbf{X}$ is the normalized covariance matrix of the state signals. The regularization parameter λ is usually small (order 10^{-5} – 10^{-3}), but its optimal value can vary a lot from case to case. Extremely small optimal values of λ indicate that the combination of task and training data was not prone to overfitting, whereas large values of λ suggest the opposite. For this reason, λ needs to be tuned using an appropriate form of validation. Usually, k -fold cross-validation is used. Once the optimal value of λ is found, all of the training data is used to train the final model and a sufficiently large and previously unused test data set serves to assess the final performance of the model.

Alternative training approaches for the data-driven optimization of linear models exist and can also be used, e. g., *recursive least squares* (RLS) for the on-line optimization in regression tasks or logistic regression² for classification tasks, but probably mainly for historical reasons, they are less common in the literature on reservoir computing.

2.2.3 Measures for reservoir performance

Since the readout of a reservoir is traditionally trained to minimize MSE, this is the natural measure of performance. Note that for classification tasks, better performance measures exist. These will be discussed at the end of this section.

As a baseline for MSE, consider the poorest possible reservoir: one for which the observed states do not contain any useful information at all. Instead of a reservoir, one could even imagine using N independent noise sources that have no relation to the input signal. If no overfitting occurs, the readout should produce a constant output equal to $\hat{\mathbf{y}} = \mu_y$, the expected value of the target signal. In this case, the MSE equals

$$\text{MSE}[\mathbf{y}, \hat{\mathbf{y}}] = E[\mathbf{y} - \mu_y]^2 = \sigma_y^2, \quad (6)$$

i. e., the variance of the target signal. Any reservoir with states that are remotely useful should lead to a MSE *on the training data* that is smaller (formal proofs exist but are omitted here). This bound only applies to the MSE measured on the training data, but the MSE on the test data for a properly trained and regularized model should not be very different.

One disadvantage to MSE is the fact that it is sensitive to the variance of the target signal, which is why other measures than MSE are often used. For example, *normalized mean squared error* (NMSE)

$$\text{NMSE}[\mathbf{y}, \hat{\mathbf{y}}] = \frac{\text{MSE}}{\sigma_y^2} \quad (7)$$

²In fact, from the perspective of machine learning, logistic regression is greatly preferable over MSE-based linear regression for classification tasks.

has the advantage that it no longer depends on the signal variance and *normalized root mean squared error* (NRMSE = $\sqrt{\text{NMSE}}$) has the additional advantage that its size can be interpreted as a fraction of the target signal's standard deviation and that its value is $\in [0, 1]$ when evaluated on the training data. However, one of the most frequently used measures, especially in statistics, is the *coefficient of determination*, or R^2 :

$$R^2[\mathbf{y}, \hat{\mathbf{y}}] = 1 - \frac{\text{MSE}}{\sigma^2(\mathbf{y})}. \quad (8)$$

Like NMSE, R^2 is normalized. In addition, given the fact that (on the training data) $\text{MSE} \leq \sigma^2(\mathbf{y})$, $0 \leq R^2 \leq 1$, where a value of 0 represents the useless reservoir mentioned above and a value of 1 corresponds to perfectly reproducing the target signal.

Using the orthogonality principle, the MSE of the resulting estimator can also be written as

$$\text{MSE}(\mathbf{y}, \hat{\mathbf{y}}) = \sigma^2(\mathbf{y}) - \text{cov}(\mathbf{y}, \hat{\mathbf{y}}) \quad (9)$$

and as a consequence, R^2 equals:

$$R^2[\mathbf{y}, \hat{\mathbf{y}}] = \frac{\text{cov}(\mathbf{y}, \hat{\mathbf{y}})}{\sigma^2(\mathbf{y})}. \quad (10)$$

Note that, in the context of linear regression (with minimization of the MSE) and when evaluated on the training data, it can be proven that R^2 equals the square of the correlation coefficient R which is defined as

$$R[\mathbf{y}, \hat{\mathbf{y}}] = \frac{\text{cov}[\mathbf{y}, \hat{\mathbf{y}}]}{\sigma(\mathbf{y}, \hat{\mathbf{y}})}. \quad (11)$$

A comprehensive explanation for this can be found in, e. g., [15].

When using a reservoir for classification tasks, it would often be better to use a linear classification model, such as logistic regression, to train the readout. This is the case whenever the outputs are thresholded to determine the correct class. Unfortunately, this is rarely done in practice, but after training, it is common to use typical classification performance errors, such as *accuracy* (the fraction of samples for which the predicted class was the correct one), *misclassification loss* (the fraction of wrongly classified samples). When addressing bitwise tasks in telecommunication, the reservoir outputs are typically sampled once per bit and the *bit error rate* (BER) is reported.

2.2.4 Echo state networks as a model system

In what follows, we briefly summarize echo state networks (ESN). In the early days of reservoir computing, these were the most frequently used types of reservoirs. As researchers have extensively studied them, their properties are well understood. In

addition, their simplicity makes it easier to understand the interaction between several important system parameters that are also relevant for most physical reservoir implementations. Understanding the interaction between these parameters and the suitability of a reservoir for a given task is crucial for designing good physical reservoirs. For these reasons, ESNs will also be used as a model system further on in this chapter.

As stated above, ESNs are discrete time analogue recurrent neural networks with hyperbolic tangent neuron activation functions. Mathematically, using bold lowercase letters for column vectors, bold uppercase letters for matrices and $n = 1 \dots n_{\max}$ to indicate discrete time, they can be described by the following equations:

$$\mathbf{a}[n] = \mathbf{W}_{\text{res}}\mathbf{x}[n-1] + \mathbf{W}_{\text{in}}\mathbf{u}[n-1] + \mathbf{w}_{\text{bias}} \quad (12)$$

$$\mathbf{x}[n] = \tanh(\mathbf{a}[n]). \quad (13)$$

Here, $\mathbf{a}$ are the neuron *activations*, the input signals to the neuron non-linearities. Note that some sources use $\mathbf{u}[n]$ instead of $\mathbf{u}[n-1]$ to calculate the activations, which results in shifting the reservoir states, the readout signals and the desired output signal one step forward in time. In ESNs, this difference can be interpreted as assigning the discrete unit delays entirely to the interneuron connections (but not to the input connections) or entirely to the neurons. Neither view maps to the reality of physical implementations, where all connections and nonlinear elements have their own physical delay and their relative sizes depend on the specific implementation. The notation chosen here seems more natural from that perspective.

The $N \times k$ input weights $\mathbf{W}_{\text{in}}$ feed the inputs to the reservoir nodes, the $N \times N$ reservoir weights $\mathbf{W}_{\text{res}}$ provide the internal feedback in the reservoir and the N internal bias weights $\mathbf{w}_{\text{bias}}$ set the operating point of the reservoir nodes. ESNs can be fully or sparsely connected. Early studies indicate that for networks with real-valued signals, this decision does not impact performance a lot [16]. The input or inputs are also connected to all neurons or a subset thereof and all neurons also receive a constant input (a bias). A typical reservoir creation recipe is as follows: the weights for all connections are sampled from some probability distribution (e. g., the standard normal distribution or a uniform distribution between -1 and 1). In the case of sparse connectivity, a randomly sampled fraction of them are set to zero. After initialization, each set of weights (the input weights, the bias weights, and the reservoir weights) is rescaled with its own scaling factor: the *input scaling* (IS), the *bias scaling* (BS), and the *spectral radius scaling* (SR), all of which impact the overall dynamical regime of the reservoir and the computations it is suitable for.

The diagonal elements of $\mathbf{W}_{\text{res}}$ are usually nonzero. Since they couple each node's previous state back to itself, the neurons act as nonlinear low-pass filters. Since these values affect the overall bandwidth and dynamics of the reservoir, it is often desirable to control their relative magnitude compared to the other weights explicitly. This led

to several descriptions of *leaky ESNs* [17], one of which is

$$\mathbf{a}[n] = \mathbf{W}_{\text{res}}\mathbf{x}[n-1] + \mathbf{W}_{\text{in}}\mathbf{u}[n-1] + \mathbf{w}_{\text{bias}} \quad (14)$$

$$\mathbf{x}[n] = \tanh((1-\alpha)\mathbf{a}[n-1] + \alpha\mathbf{a}[n]) \quad (15)$$

where α is the *leak rate*, $0 \leq \alpha \leq 1$. It is also possible to randomize the leak rates, such that they are different for each neuron. Since the self-feedback of each neuron is now explicitly regulated by the leak rate, the diagonal elements of $\mathbf{W}_{\text{res}}$ are often set to zero. The *leak rate* α is yet another tuning parameter for the reservoir.

Finally, ESNs can also be used with *output feedback*. In this case, the output of the readout projected back into the reservoir nodes, again with random weights and a tuneable rescaling factor. Since this approach is mostly used for signal generation tasks, we will not consider it further here. More information can be found in [8, 18, 19].

For an ESN, as for most other incarnations of reservoir computing, the target signal $\mathbf{y}[n]$ is approximated by optimising the $N+1$ readout weights $\{\mathbf{w}_{\text{out}}, \mathbf{w}_0\}$ in a linear combination of the state signals at each time t :

$$\hat{\mathbf{y}}[n] = \mathbf{w}_{\text{out}}^T \mathbf{x}[n] + \mathbf{w}_0 \quad (16)$$

where $\mathbf{u}[n]$ is the k -dimensional input vector at time n , N is the number of nodes (*neurons*) in the network, $\mathbf{x}[n]$ is the N -dimensional state vector, and $\mathbf{y}[n]$ is the output of the reservoir system. Note that in early works on reservoir computing, the input $\mathbf{u}[n]$ is also used in the linear combination, yielding additional input weights in equation (16).

In many ways, echo state networks are very different from most physical systems. For one, they operate in discrete time. In addition, they are usually fully connected, whereas this is not feasible for most physical systems. However, most physical systems have tuning parameters that have a very similar impact on computing to the scaling parameters and leak rate in an ESN. In addition, they can often be simplified in such a way that they resemble an ESN with a very specific connection pattern and setting of the scaling parameters. The knowledge of how these parameters affect computation in ESNs has often proved invaluable in tuning physical reservoir parameters. Studying ESN-like simplified models of a new physical system often helps to gain additional insights.

2.2.5 General requirements to reservoirs

In Section 2.2.2, we have described how to exploit the natural dynamics of any driven dynamical system for computation. Obviously, this in itself is not a recipe for success, since it only works well if there is a good match between the system's dynamics and

what we need for a given task. In this section, we briefly review the most important theoretical concepts that have been developed to reason about the way reservoirs process their inputs and which kind of processing is required for a task. The focus here will be on an intuitive understanding of what these concepts mean. For an in-depth treatment, we refer to the original papers.

The fading memory property. Both ESNs and LSMs are nonlinear dynamical systems, which can be tuned to display a variety of dynamical regimes, including oscillatory and chaotic ones, by increasing their internal connection strengths. However, for the purpose of reservoir computing, the connection weights need to be tuned such that the network displays the *fading memory* property. Intuitively, this means that when the network is undriven, any transient dynamics eventually die out and the network evolves toward a unique stable state. The more slowly the transients die out, the longer information about past inputs *echoes* in the internal state. This *memory* about past states and inputs is largest when the system dynamics are close to a bifurcation that pushes them away from stability. Such a regime is often termed *edge of chaos*, but *edge of stability* seems more appropriate in this case, as it stresses that the driven system should operate at the stable side of the bifurcation. Tuning the system to be closer to or further away from the edge of stability is one of the ways to tune its memory to match task requirements.

Fading memory also means that, if you start the same reservoir from any two different initial states and drive both versions with the same input sequences, their internal states will evolve toward identical trajectories. When operating close to the edge of stability the effects of the initial state or past inputs remain present in the system for a long time as (nonlinearly transformed) *echos* of the past. Fading memory also implies that *similar* input patterns lead to similar states. In other words: reservoir computing extracts information from time-varying signals by extracting information that is spread in time and projecting it onto present time in such a way that a linear function is able to extract it. This implies that, for most tasks, the reservoir not only needs to remember the past, but also to nonlinearly transform it on the way.

In machine learning terms, the ESN reservoir acts as a high-dimensional spatiotemporal transformation (a filter bank) of the input signal(s). Obviously, the quality of the result largely depends on whether or not the internal state variables (the *features* for the linear regression) are suitable for the task. First, they should be tuned such that the reservoir indeed has the fading memory property. This can be ensured by analysing the Jacobian of the reservoir and finding the maximal local Lyapunov exponent (LLE). If this is < 1 , the reservoir is at the stable side of the edge of stability. However, this analysis is rather complex. For $\tanh$ -ESNs, an approximation is usually made by linearizing the ESN at the point where its neurons have maximal gain (equal to +1), i. e., when their inputs are zero. Ensuring stability in this linearized reservoir boils down to setting the largest eigenvalue of the reservoir weight matrix W_{res} (its *spectral radius*) to 1. In most ESNs, this will result in stability [7].

Universal approximation properties. One of the cornerstones of the theory of (nonrecurrent) neural networks are the early universal approximation theorems, e. g., [4, 5], which basically state that a neural network with two layers of sigmoidal neurons (one hidden layer and one output layer) can approximate any desired input-output relationship with any desired precision *if the number of neurons on the hidden layer is made large enough*. This relies on one very important principle: by making the network larger, you will eventually be able to realize all functions that you were not able to realize with smaller networks.

Although the proofs are less strict, similar theorems exist in reservoir computing, stating that some types of reservoirs with a linear readout can be universal approximators for filters with fading memory [20]. A first requirement for this is that the reservoir itself must fulfill the fading memory property. A second requirement is that it has the *separation property*, which can be interpreted as the fact that, if the reservoir is made large enough, any difference in the input sequence must eventually result in linearly separable differences in the state space.

A reservoir is a universal approximator *only* if, by making the reservoir larger, a linear combination of the observed states can eventually cover all possible input transformations. The state signals of a reservoir are transformations of their common input history, and because of the recurrent nature of reservoirs, these signals are coupled. For this reason, the universal approximation property should be treated with some care. It is not sufficient to have a scalable nonlinear dynamical system in which some parameters are sampled from some random distribution. In fact, it is quite easy to construct a class of reservoirs that does *not* fulfil the universality. As a simple example, consider ESNs for which all input bias weights w_{bias} are set to zero. Since the $\tanh$ function is an odd function around its origin ($\tanh(-x) = -\tanh(x)$), such a reservoir can never output an even function of its inputs ($f(-x) = -f(x)$), however large we make it.

Although they are part of the theoretical foundations of and motivation for neural networks in general and reservoir computing in particular, universal approximation theorems are often not very useful in practice. Merely making a reservoir larger is often not the best way to improve its approximation quality. If this were the case, it should be possible to approximate any input-output relationship with any desired precision simply by making the reservoir larger, *while using the same approach for constructing the reservoir*. Unfortunately, this is not the case, as in practice, reservoir performance is known to saturate when increasing reservoir size.

In general, for physical reservoir computing, there is no systematic methodology to even check whether a given physical system could serve as a universal approximator and from which distributions its parameters should be sampled to achieve this. In addition, such theorems state nothing about how fast the approximation quality converges.

2.2.6 Physical reservoir computing

Very soon after the almost simultaneous introduction of the principles of reservoir computing from different research backgrounds, the common basis of these early works was identified and unified under the term *reservoir computing* [12, 21]. Also, researchers quickly started applying the approach to other (mostly non-linear) dynamical systems, and in particular to physical systems. In fact, the term reservoir computing was inspired by one of the first physical realizations [22], in which sound vibrations were transferred to a basin of water and the ripple patterns were used for spoken digit classification. The term *physical reservoir computing* was launched in [7], where a mechanical (robot) body was used as a reservoir to generate its own closed loop control. Soon after that, the subfields of *photonic* reservoir computing and *mechanical* reservoir computing (in the context of *morphological computing*) were launched. This book is dedicated to the first field. In the second, the focus is mostly on facilitating robotic control by using its body as a reservoir [23–25], responding to the motor actuations and the interaction with the outside world. In order to develop a theoretical foundation in this field, model systems, consisting of masses, passive springs, and dampers, were also studied [26–28]. Besides mechanical and photonic systems, physical reservoir computing is now being used with a range of different physical systems, such as memristive networks [29–34], carbon nanotubes [35], or molecular computing [36].

In general, the dynamical systems that serve as physical reservoirs operate in continuous time and are driven by continuous time input signals $\mathbf{u}(t)$. They are described by the temporal evolution of their internal state space $\mathbf{x}(t)$, which can be finite- or infinite-dimensional.

However, for the purpose of reservoir computing, they are driven in discrete time $t = nT$, $n \in \mathbb{Z}$, where T is the *sampling period*. This means that their externally applied input signals $\mathbf{u}(t)$ are generated from a discrete time sequence of values $\mathbf{u}[n]$ which are presented to the system at a given input rate $1/T$. An input signal generator then transforms this sequence into a continuous time signal using an appropriately chosen encoding scheme. The simplest scheme is to generate a piecewise constant signal, changing the input only at discrete moments $t = kT$.

Physical reservoirs are also observed in discrete time, i. e., their observed state variables are sampled with the sampling rate $1/T$. In practice, a *time shift* is sometimes used between the discrete moments $t = kT$ at which the input signals are applied and the corresponding moments of observation $t' = kT + \tau$. In most physical situations, the measured signals $x_i[n]$ will not be the state values themselves, but signals that have been transformed by the measurement process. For example, optical signals are converted to the electrical domain by measuring their intensity.

When studying a physical dynamical system, we often do not have access to the entire internal state space $\mathbf{x}(t)$. This corresponds to experimental situations where many variables may not be measurable, and is also the case if the dimensionality of

the state space is infinite. We therefore assume that we can only access the instantaneous state of the system through a finite number N of the internal variables: $x_i(t)$, $i = 1, \dots, N$. From here onward, the *dimension* of the reservoir will refer to the number of *observed* state space signals N , independent of the actual dimensionality of the state space of the physical system.

2.3 Information processing in reservoirs

In this section, we analyze how reservoirs process their input signals and how the most important high-level reservoir parameters interact to determine the suitability of a reservoir for a given task. We also discuss a few measures that allow to quantify this.

2.3.1 Reproduction memory

One crucial property of a reservoir is how long information about past inputs remains in the system. This is the *memory* that can be exploited by the readout to approximate the desired output. One of the very first ways in which this memory was analyzed is by quantifying how well a linear readout can reproduce past inputs. Since these inputs must be untransformed, this type of memory is also termed *linear memory*.

A way to quantify this, the *linear memory capacity*, was first proposed by Herbert Jaeger in one of his seminal works about echo state networks [37]. His proposed measure is the squared correlation coefficient between the target signal and its approximation $\hat{\mathbf{y}}$ as defined in equation (16):

$$C[\mathbf{X}, \mathbf{y}] = \frac{\text{cov}^2(\hat{\mathbf{y}}, \mathbf{y})}{\sigma^2(\hat{\mathbf{y}})\sigma^2(\mathbf{y})}. \quad (17)$$

By definition, its values lie in $[0, 1]$. In the context of reservoir computing, a value of 1 implies that the readout can achieve a perfect reproduction of the target signal, whereas a value of 0 indicates that the state signals carry no information whatsoever that can be used to approximate the target signal with linear regression.

Using this measure, Jaeger proposed to quantify a reservoir's memory by assessing its capacity for reconstructing the input signal $\mathbf{u}_{-d} = \mathbf{u}[n-d]$ of d time steps in the past, for $d = 1 \dots d_{\max}$:

$$CM_d[\mathbf{X}, \mathbf{u}_{-d}] = \max_{(\mathbf{w}_{\text{out}})} C[\hat{\mathbf{y}}, \mathbf{u}_{-d}] \quad (18)$$

in which the maximum corresponds to the best possible MSE, i. e., the MSE for a theoretical readout that is trained on sequences of infinite length. Using these, the total

linear memory capacity can be defined as

$$CM[\mathbf{X}, \mathbf{u}] = \sum_{d=1}^{\infty} CM_d[\mathbf{X}, \mathbf{u}_d]. \quad (19)$$

Note that the capacity $CM_d[\mathbf{u}, \mathbf{u}_d]$ for reconstructing a signal from its own past equals its squared autocorrelation function. However, the original aim of linear memory capacity was to characterize the processing that occurs in the dynamical system itself. By taking the inputs to be sequences of i.i.d. values, any measured memory will be due to the dynamical system and not due to the input's autocorrelation or *self-memory*. The distribution $p(u)$ from which the input values are sampled can be chosen according to what is most relevant for the dynamical system under study, but it is most common to use the uniform distribution in, e. g., $[-1, 1]$.

It should be noted that in many, if not all, real world situations the inputs are not i.i.d. In this case, it is useful to quantify how a reservoir processes the actual input signals. In order to quantify this, one can still calculate linear memory capacities using equation (18) with the actual input signal, but the upper bound from equation (20) is no longer guaranteed because the target signals of LMC, $\mathbf{u}_d$, are no longer uncorrelated. An alternative would be to decorrelate the target signals in order of increasing values of d . This can in principle be achieved using a Gram–Schmidt orthogonalization procedure, in which case the bound of equation (20) would again hold. However, as numerical errors quickly build up in the orthogonalization procedure, this is usually only feasible up to small maximal values of d .

An additional reason for measuring capacities with i.i.d. input sequences is that under these conditions the proposition holds that the *total linear memory capacity is bounded by the number of state signals that is used in the readout*:

$$CM_{\text{lin}}[\mathbf{X}, \mathbf{u}] \leq N. \quad (20)$$

In principle, this bound can be achieved, e. g., for linear ESNs for which W_{res} is an orthogonal matrix [38]. As soon as a reservoir operates in a nonlinear regime, CM_{lin} quickly deteriorates. The theoretical upper bound for linear memory can be approximated in nonlinear reservoirs, by operating them in their small-signal regime and assuming an unbounded signal-to-noise ratio. For example, in ESN reservoirs this can be achieved by setting the input scaling and bias scaling parameters so small that each tanh neuron's input is close to zero. However, as soon as some noise is introduced into the system or the readout, this theoretical linear memory quickly disappears. For a given value of the total linear memory capacity, different memory profiles can be achieved by tuning the reservoir's hyperparameters.

2.3.2 Nonlinear processing capacity

Clearly, a reservoir's linear memory does not tell the whole story, since most signal processing or classification tasks require much more than linear memory: they require a nonlinear transformation of the input history. This is typically what occurs in nonlinear dynamical systems such as ESNs. This is also the basis of nonlinear photonic reservoirs such as the original SOA-based integrated photonic reservoirs [39] (simulated only) or delayed feedback reservoir computing architectures that use a nonlinearity in the feedback loop (see Chapter 5 for an introduction). However, nonlinear transformations can also be achieved by combining a linear reservoir (providing memory) with a nonlinear readout. If sufficient memory can be provided by the reservoir *and* if a nonlinear (instead of a linear) readout is used that is in itself universal for memoryless computations, then such a reservoir can also be a universal approximator of nonlinear filters with fading memory [26]. In this case, sufficient memory means that *all* past inputs that are necessary for solving the task can be perfectly reconstructed from the observed reservoir state signals. This approach was discussed in [26] for mechanical systems. In practice, it turns out that a rather simple but nonlinear readout already offers a lot of the computations required for many real-world tasks. This principle is used in, e. g., the integrated passive photonic reservoirs described in [40–42].

Early on, researchers working with ESN-based reservoir computing noticed that for a fixed reservoir size with a linear readout, there is a trade-off between memory and nonlinearity. As soon as a reservoir is tuned into a slightly non-linear regime, the total linear memory capacity quickly drops. In a first attempt to quantify this, [43], the spectral transformation of reservoirs was used as a measure of their nonlinearity. This was measured by driving the reservoirs with a sinusoidal input signal and measuring the fraction of the energy in the state signals that remained at that frequency.

In [44], an extension of linear memory capacity was proposed which allows to more accurately quantify the transformations a reservoir performs. Intuitively, the approach can be explained as follows. In principle, any fading memory filter in discrete time can be approximated with any desired precision by a polynomial expansion on a potentially very large but finite number of values in the input history. This means that, for any given joint probability density of the relevant input values, a Hilbert space of fading memory functions can be constructed with an orthogonal polynomial basis (see [44] for rigorous definitions). The construction of such a basis is very similar to the construction of a polynomial basis for real functions of multiple real variables, where these variables are the past input values $\mathbf{u}_d$. If we assume, as we did for linear memory capacity, that the input sequences are generated by an i.i.d. uniform process with values in $[-1, 1]$, a polynomial basis can be constructed from finite products of normalized Legendre polynomials for each time step:

$$\mathbf{y}_k = \prod_i \mathcal{L}_{k_i}(\mathbf{u}_{-i}) \quad (21)$$

where $\mathcal{L}_{k_i}(\cdot)$, $k_i \geq 0$, is the normalized Legendre polynomial of degree k_i . The normalized Legendre polynomial of degree 0 is a constant. The normalized Legendre polynomials of degree 1 are the u_d that were used for computing the linear memory capacity. The set $\mathbf{k}$ lists the polynomial degrees for each past input, where k_i is the degree corresponding to the input of i time steps in the past. For each basis function, we can compute its total degree as the sum of the individual degrees for each delay: $\mathcal{D}_{\mathbf{k}} = \sum_i k_i$ and its memory depth as the largest index in $\mathbf{k}$ for which k_i is nonzero.

Expanding on the theory for linear memory capacity, we can again define the capacity for approximating these basis functions as the expected value for infinitely long sequences of the squared correlation coefficient between the target signal and its best approximation by the reservoir. We can also define the *total information processing capacity* as the sum of these capacities across all basis functions and prove that this sum, too, always lies in $[0, N]$.

In practice, capacities have to be estimated with limited precision from finite input sequences. In [44], a procedure is described to iterate through them. It uses relevance thresholds to decide how large a capacity estimate has to be in order to be sufficiently accurate. Constraining the set of possible basis functions to a finite set can be done either by setting upper bounds on the total degree and memory depth, or by assuming that a physical system's approximation capacity decreases monotonically for increasing degree and/or memory depth. This was found to be a reasonable assumption for many physical systems.

2.3.3 Memory, nonlinearity, and noise-sensitivity

As stated in the previous section, the functions used to compute linear memory capacity form a subset of those used to compute total memory capacity. The fact that the total capacity of any reservoir system is bounded by the total number of state signals used in the readout necessarily implies that there must be a trade-off between linear and nonlinear processing. By adding all capacities with the same memory depth or the same total degree, we can visualize the capacity profile of a given reservoir, as shown in Figure 2.2. This accords with our previous observations that an ESN's capacity for exactly reproducing past inputs rapidly decreases as the neurons are operated in a more nonlinear regime. In Figure 2.3, this is made even more clear by plotting the fraction of total capacity that is due to linear memory and nonlinear memory, respectively.

In most cases when a system is made more nonlinear, a rapidly increasing number of nonlinear terms emerges in the polynomial approximation of the reservoir states. In particular, when the spectral radius is high enough for the inputs to reverberate in the system's dynamics for a while, each nonlinearity it passes performs a nonlinear mixture of its inputs, spreading the input information across more and higher degree

Figure 2.2: Summarized total information processing capacity profiles (for total degrees up to 5) for different settings of the scaling parameters in echo state networks: (top panels) Impact of the bias scaling, showing no contributions of even degree capacities for zero bias and increasing nonlinear capacities for increasing bias; (bottom panels) Impact of the spectral radius, showing increased linear memory as the spectral radius approaches 1.0 (each bar corresponds to a single ESN with 100 neurons, spectral radius and bias scaling are indicated in each panel's title; colors group capacities for basis functions with the same total degree).

Figure 2.3: Linear and non-linear contributions to the total memory capacity: for increasing input scaling, the reservoir is driven into a more non-linear regime. The total linear memory capacity decreases and the contribution of non-linear capacities to the total capacity increases (each plot corresponds to a single ESN with 100 neurons, spectral radius equal to 0.85, bias scaling equal to 0.05).

polynomial basis functions. As a consequence, the capacities that measure these contributions get very small and, therefore, hard to estimate accurately.

The same nonlinear mixing also dramatically reduces a system's performance under multiple inputs [45], as well as its noise robustness (since each noise source can be considered as an additional input signal). When multiple inputs are present, one needs to extend the Hilbert space model to multiple inputs, i. e., consider all func-

Figure 2.4: Impact of noise in ESNs: for increasing input scaling, the reservoir is driven into a more non-linear regime and the degradation of total capacity as a function of input noise increases (each bar corresponds to a single ESN with 100 neurons, spectral radius = 0.7, bias scaling = 0.5; colours group capacities for basis functions with the same total degree).

tions of the complete (fading memory) input history of all inputs. This includes all higher degree cross products between delayed values originating from different input signals, which also emerge naturally from the nonlinear mixing. Figure 2.4 illustrates this for reservoirs that are driven with a single input, to which an increasing amount of uniformly distributed noise is added. Because each reservoir is driven with a different input power (input scaling), the signal-to-noise ratio is reported in the plots. Whereas the total linear capacity is barely affected by increasing noise, the available nonlinear capacity quickly decreases for more nonlinear reservoir. As more noise gets mixed non-linearly with the information signal, less information about this signal and its history can be extracted by a linear readout.

2.4 Conclusion

The way the input signal(s) of a driven dynamical system perturbs its internal state, affecting not just the present but also the future of this state, can be considered as a form of *computation*. From its origins in neuroscience and engineering, reservoir computing has evolved into an easy and efficient way to harness this computation, in order to perform nonlinear filtering of the input signal(s) or to extract information from them. The larger the number of observed state signals and the more diverse the ways in which they are affected by current and past inputs, the better a system can be used for computation.

In order to make a given system adaptable to a range of tasks, it is beneficial if its dynamical regime can be tuned by a number of global parameters, such as a scaling of the feedback gains or losses, of the input power and of the overall operating point. For systems with a fixed number of observed signals, there exists a trade-off between (linear) memory and nonlinearity. In addition, more nonlinear systems tend to be less robust to noise.

Bibliography

- [1] J. von Neumann. "Probabilistic logics and the synthesis of reliable organisms from unreliable components". In: *Automata studies* 34 (1956), pp. 43–99.
- [2] S. Mittal. "A survey of techniques for approximate computing". In: *ACM Computing Surveys* 48.4 (Mar. 2016), 62:1–62:33.
- [3] Q. Xu, T. Mytkowicz, and N. S. Kim. "Approximate computing: a survey". In: *IEEE Design and Test* 33 (2016), pp. 8–22. ISSN: 2168-2356.
- [4] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2.4 (1989), pp. 303–314.
- [5] K. Hornik. "Approximation capabilities of multilayer feedforward networks". In: *Neural Networks* 4.2 (1991), pp. 251–257.
- [6] C. Horsman et al. "When does a physical system compute?" In: *Proceedings. Mathematical, Physical, and Engineering Sciences/The Royal Society* 470.2169 (2014), p. 20140182. ISSN: 1364-5021. doi: 10.1098/rspa.2014.0182. URL: <http://rspa.royalsocietypublishing.org/content/470/2169/20140182>.
- [7] K. Caluwaerts et al. "Locomotion without a brain: physical reservoir computing in tensegrity structures". In: *Artificial Life* 19.1 (2013), pp. 35–66.
- [8] H. Jaeger and H. Haas. "Harnessing nonlinearity: predicting chaotic systems and saving energy in wireless communication". In: *Science* 304 (2004), pp. 78–80.
- [9] H. Jaeger. "The "echo state" approach to analysing and training recurrent neural networks – with an erratum note 1". In: 2010.
- [10] W. Maass, T. Natschlager, and H. Markram. "Real-time computing without stable states: a new framework for neural computation based on perturbations". In: *Neural Computation* 14 (2002), pp. 2531–2560.
- [11] B. Schrauwen, D. Verstraeten, and J. M. Van Campenhout. "An overview of reservoir computing: theory, applications and implementations". In: *ESANN*. 2007.
- [12] M. Lukoševičius and H. Jaeger. "Reservoir computing approaches to recurrent neural network training". In: *Computer Science Review* 3 (2009), pp. 127–149.
- [13] M. Lukoševičius, H. Jaeger, and B. Schrauwen. "Reservoir computing trends". In: *KI – Kunstliche Intelligenz* 26 (2012), pp. 365–371.
- [14] A. Goudarzi and C. Teuscher. "Reservoir computing: Quo Vadis?" In: *NANOCOM*. 2016.
- [15] D. Schiebler. *R and R², the relationship between correlation and the coefficient of determination*. <http://danshiebler.com/2017-06-25-metrics/>. Blog, Posted on June 25, 2017.
- [16] L. Buesing, B. Schrauwen, and R. A. Legenstein. "Connectivity, dynamics, and memory in reservoir computing with binary and analog neurons". In: *Neural Computation* 22.5 (2010), pp. 1272–1311.
- [17] H. Jaeger et al. "Optimization and applications of echo state networks with leaky-integrator neurons". In: *Neural Networks* 20.3 (2007), pp. 335–352.
- [18] H. Jaeger and H. Markram. "On the role of feedback in enhancing the computational power of generic neural microcircuits". In: 2006.
- [19] H. Jaeger. "Controlling recurrent neural networks by conceptors". In: *CoRR* abs/1403.3369 (2014). arXiv:1403.3369. URL: <http://arxiv.org/abs/1403.3369>.
- [20] W. Maass and H. Markram. "On the computational power of circuits of spiking neurons". In: *Journal of Computer and System Sciences* 69 (2004), pp. 593–616.
- [21] D. Verstraeten et al. "An experimental unification of reservoir computing methods." In: *Neural Networks* 20.3 (Apr. 2007), pp. 391–403. doi: 10.1016/j.neunet.2007.04.003.
- [22] C. Fernando and S. Sojakka. "Pattern recognition in a bucket". In: *Lecture notes in computer science* 2801/2003 (2003), pp. 588–597. doi: 10.1007/978-3-540-39432-7_63.

- [23] K. Nakajima et al. "A soft body as a reservoir: case studies in a dynamic model of octopus-inspired soft robotic arm". In: *Frontiers in Computational Neuroscience*. 2013.
- [24] K. Nakajima et al. "Exploiting short-term memory in soft body dynamics as a computational resource". In: *Journal of The Royal Society Interface* 11.100 (2014).
- [25] J. Degraeve et al. "Developing an embodied gait on a compliant quadrupedal robot". In: *2015 IEEE-RSJ international conference on intelligent robots and systems (IROS)* (2015), pp. 4486–4491.
- [26] H. Hauser et al. "Towards a theoretical foundation for morphological computation with compliant bodies". In: *Biological Cybernetics* 105 (2011), pp. 355–370.
- [27] H. Hauser et al. "The role of feedback in morphological computation with compliant bodies". In: *Biological Cybernetics* 106 (2012), pp. 595–613.
- [28] G. Urbain et al. "Morphological properties of mass-spring networks for optimal locomotion learning". In: *Frontiers in Neurorobotics*. 2017.
- [29] A. V. Avizienis et al. "Neuromorphic atomic switch networks". In: *PLoS ONE*. 2012.
- [30] M. S. Kulkarni and C. Teuscher. "Memristor-based reservoir computing". In: *2012 IEEE-ACM international symposium on nanoscale architectures (NANOARCH)* (2012), pp. 226–232.
- [31] H. O. Sillin et al. "A theoretical and experimental study of neuromorphic atomic switch networks for reservoir computing." In: *Nanotechnology* 24.38 (2013), p. 384004.
- [32] J. P. Carbajal et al. "Memristor models for machine learning". In: *Neural Computation* 27.3 (2015), pp. 725–747.
- [33] D. Kudithipudi et al. "Design and analysis of a neuromemristive reservoir computing architecture for biosignal processing". In: *Frontiers in Neuroscience*. 2015.
- [34] C. Du et al. "Reservoir computing using dynamic memristors for temporal information processing". In: *Nature Communications*. 2017.
- [35] M. Dale et al. "Evolving carbon nanotube reservoir computers". In: *UCNC*. 2016.
- [36] A. Goudarzi, M. R. Lakin, and D. Stefanovic. "DNA reservoir computing: a novel molecular computing approach". In: *DNA computing and molecular programming* (2013), pp. 76–89.
- [37] H. Jaeger. "Tutorial on training recurrent neural networks, covering BPTT, RTRL, EKF and the 'echo state network' approach". In: *Technical Report GMD Report 159, German National Research Center for Information Technology* (2002).
- [38] O. L. White, D. D. Lee, and H. Sompolinsky. "Short-term memory in orthogonal neural networks." In: *Physical Review Letters* 92.14 (2004), p. 148102.
- [39] K. Vandoorne et al. "Toward optical signal processing using photonic reservoir computing". In: *Optics Express* 16 (2008), pp. 11182–11192.
- [40] K. Vandoorne et al. "Experimental demonstration of reservoir computing on a silicon photonics chip." In: *Nature Communications* 5 (2014), p. 3541.
- [41] A. Katumba et al. "A multiple-input strategy to efficient integrated photonic reservoir computing". In: *Cognitive Computation* 9 (2017), pp. 307–314.
- [42] A. Katumba et al. "Low-loss photonic reservoir computing with multimode photonic integrated circuits". In: *Scientific Reports*. 2018.
- [43] D. Verstraeten et al. "Memory versus non-linearity in reservoirs". In: *The 2010 international joint conference on neural networks (IJCNN)* (2010), pp. 1–8.
- [44] J. Dambre et al. "Information processing capacity of dynamical systems". In: *Scientific Reports* 2 (2012), p. 514.
- [45] M. Hermans and B. Schrauwen. "Memory in linear recurrent neural networks in continuous time". In: *Neural Networks* 23.3 (2010), pp. 341–355.

DE GRUYTER

*Daniel Brunner, Miguel C. Soriano,
Guy Van der Sande (Eds.)*

PHOTONIC RESERVOIR COMPUTING

OPTICAL, RECURRENT NEURAL NETWORKS

Photonics has long been considered an attractive substrate for next generation implementations of machine-learning concepts, including neural network models. Reservoir Computing tremendously facilitated the realization of recurrent neural networks in analogue hardware. This concept exploits the properties of complex nonlinear dynamical systems, giving rise to photonic reservoirs implemented by semiconductor lasers, telecommunication modulators, spatial light modulators and integrated photonic chips.

9 783110 582000

www.degruyter.com

ISBN 978-3-11-058200-0

Contents

Preface — V

List of Contributing Authors — XI

Daniel Brunner, Piotr Antonik, and Xavier Porte

1 Introduction to novel photonic computing — 1

Joni Dambre

2 Information processing and computation with photonic reservoir systems — 33

Andrew Katumba, Matthias Freiberger, Floris Laporte, Alessio Lugnan, Stijn Sackesyn, Chonghuai Ma, Joni Dambre, and Peter Bienstman

3 Integrated on-chip reservoirs — 53

Daniel Brunner, Julian Bueno, Xavier Porte, Sheler Maktoobi, and Louis Andreoli

4 Large scale spatiotemporal reservoirs — 83

Silvia Ortín, Luis Pesquera, Guy Van der Sande, and Miguel C. Soriano

5 Time delay systems for reservoir computing — 117

Laurent Larger

6 Ikeda delay dynamics as *Reservoir* processors — 153

Guy Van der Sande and Miguel C. Soriano

7 Semiconductor lasers as reservoir substrates — 185

Piotr Antonik, Serge Massar, and François Duport

8 Advanced reservoir computers: analogue autonomous systems and real time control — 205

Outlook — 259

Index — 263