available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cosrev

Survey

Reservoir computing approaches to recurrent neural network training

*Mantas Lukoševičius**, *Herbert Jaeger*

School of Engineering and Science, Jacobs University Bremen gGmbH, P.O. Box 750 561, 28725 Bremen, Germany

ARTICLE INFO

Article history:
Received 17 October 2008
Received in revised form
27 March 2009
Accepted 31 March 2009

ABSTRACT

Echo State Networks and Liquid State Machines introduced a new paradigm in artificial recurrent neural network (RNN) training, where an RNN (the *reservoir*) is generated randomly and only a readout is trained. The paradigm, becoming known as *reservoir computing*, greatly facilitated the practical application of RNNs and outperformed classical fully trained RNNs in many tasks. It has lately become a vivid research field with numerous extensions of the basic idea, including reservoir adaptation, thus broadening the initial paradigm to *using different methods for training the reservoir and the readout*. This review systematically surveys both current ways of generating/adapting the reservoirs and training different types of readouts. It offers a natural conceptual classification of the techniques, which transcends boundaries of the current "brand-names" of reservoir methods, and thus aims to help in unifying the field and providing the reader with a detailed "map" of it.

1. Introduction

Artificial *recurrent neural networks* (RNNs) represent a large and varied class of computational models that are designed by more or less detailed analogy with biological brain modules. In an RNN numerous abstract *neurons* (also called *units* or *processing elements*) are interconnected by likewise abstracted *synaptic connections* (or *links*), which enable *activations* to propagate through the network. The characteristic feature of RNNs that distinguishes them from the more widely used *feedforward neural networks* is that the connection topology possesses cycles. The existence of cycles has a profound impact:

- An RNN may develop a self-sustained temporal activation dynamics along its recurrent connection pathways, even in the absence of input. Mathematically, this renders an RNN to be a *dynamical system*, while feedforward networks are *functions*.
- If driven by an input signal, an RNN preserves in its internal state a nonlinear transformation of the input history — in other words, it has a *dynamical memory*, and is able to process temporal context information.

This review article concerns a particular subset of RNN-based research in two aspects:

- RNNs are used for a variety of scientific purposes, and at least two major classes of RNN models exist: they can be used for purposes of modeling biological brains, or as engineering tools for technical applications. The first usage belongs to the field of computational neuroscience, while the second frames RNNs in the realms

* *Corresponding author.*
E-mail addresses: m.lukosevicius@jacobs-university.de (M. Lukoševičius), h.jaeger@jacobs-university.de (H. Jaeger).

1574-0137/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.cosrev.2009.03.005

of machine learning, the theory of computation, and nonlinear signal processing and control. While there are interesting connections between the two attitudes, this survey focuses on the latter, with occasional borrowings from the first.

- From a dynamical systems perspective, there are two main classes of RNNs. Models from the first class are characterized by an energy-minimizing stochastic dynamics and symmetric connections. The best known instantiations are Hopfield networks [1,2], Boltzmann machines [3,4], and the recently emerging Deep Belief Networks [5]. These networks are mostly trained in some unsupervised learning scheme. Typical targeted network functionalities in this field are associative memories, data compression, the unsupervised modeling of data distributions, and static pattern classification, where the model is run for multiple time steps per single input instance to reach some type of convergence or equilibrium (but see e.g., [6] for extension to temporal data). The mathematical background is rooted in statistical physics. In contrast, the second big class of RNN models typically features a deterministic update dynamics and directed connections. Systems from this class implement nonlinear filters, which transform an input time series into an output time series. The mathematical background here is nonlinear dynamical systems. The standard training mode is supervised. This survey is concerned only with RNNs of this second type, and when we speak of *RNNs* later on, we will exclusively refer to such systems.[1]

RNNs (of the second type) appear as highly promising and fascinating tools for nonlinear time series processing applications, mainly for two reasons. First, it can be shown that under fairly mild and general assumptions, such RNNs are universal approximators of dynamical systems [7]. Second, biological brain modules almost universally exhibit recurrent connection pathways too. Both observations indicate that RNNs should potentially be powerful tools for engineering applications.

Despite this widely acknowledged potential, and despite a number of successful academic and practical applications, the impact of RNNs in nonlinear modeling has remained limited for a long time. The main reason for this lies in the fact that RNNs are difficult to train by gradient-descent-based methods, which aim at iteratively reducing the training error. While a number of training algorithms have been proposed (a brief overview is given in Section 2.5), these all suffer from the following shortcomings:

- The gradual change of network parameters during learning drives the network dynamics through bifurcations [8]. At such points, the gradient information degenerates and may become ill-defined. As a consequence, convergence cannot be guaranteed.
- A single parameter update can be computationally expensive, and many update cycles may be necessary. This results in long training times, and renders RNN training feasible only for relatively small networks (in the order of tens of units).
- It is intrinsically hard to learn dependences requiring long-range memory, because the necessary gradient information exponentially dissolves over time [9] (but see the Long Short-Term Memory networks [10] for a possible escape).
- Advanced training algorithms are mathematically involved and need to be parameterized by a number of global control parameters, which are not easily optimized. As a result, such algorithms need substantial skill and experience to be successfully applied.

In this situation of slow and difficult progress, in 2001 a fundamentally new approach to RNN design and training was proposed independently by Wolfgang Maass under the name of *Liquid State Machines* [11] and by Herbert Jaeger under the name of *Echo State Networks* [12]. This approach, which had predecessors in computational neuroscience [13] and subsequent ramifications in machine learning as the *Backpropagation-Decorrelation* [14] learning rule, is now increasingly often collectively referred to as *Reservoir Computing* (RC). The RC paradigm avoids the shortcomings of gradient-descent RNN training listed above, by setting up RNNs in the following way:

- A recurrent neural network is *randomly* created and remains unchanged during training. This RNN is called the *reservoir*. It is passively excited by the input signal and maintains in its state a nonlinear transformation of the input history.
- The desired output signal is generated as a linear combination of the neuron's signals from the input-excited reservoir. This linear combination is obtained by linear regression, using the teacher signal as a target.

Fig. 1 graphically contrasts previous methods of RNN training with the RC approach.

Reservoir Computing methods have quickly become popular, as witnessed for instance by a theme issue of Neural Networks [15], and today constitute one of the basic paradigms of RNN modeling [16]. The main reasons for this development are the following:

Modeling accuracy. RC has starkly outperformed previous methods of nonlinear system identification, prediction and classification, for instance in predicting chaotic dynamics (three orders of magnitude improved accuracy [17]), nonlinear wireless channel equalization (two orders of magnitude improvement [17]), the Japanese Vowel benchmark (zero test error rate, previous best: 1.8% [18]), financial forecasting (winner of the international forecasting competition NN3[2]), and in isolated spoken digits recognition (improvement of word error rate on benchmark from 0.6% of previous best system to 0.2% [19], and further to 0% test error in recent unpublished work).

Modeling capacity. RC is computationally universal for continuous-time, continuous-value real-time systems modeled with bounded resources (including time and value resolution) [20,21].

[1] However, they can also be used in a converging mode, as shown at the end of Section 8.6.

[2] http://www.neural-forecasting-competition.com/NN3/index.htm.

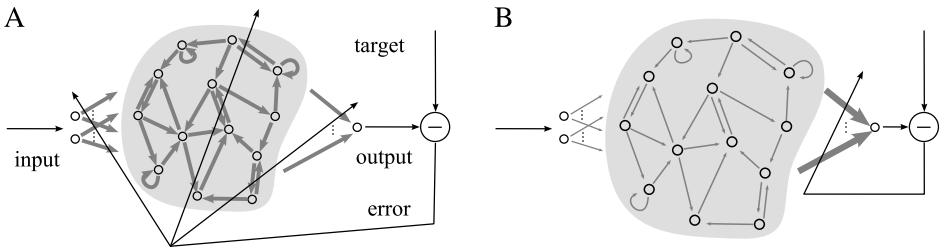

Fig. 1 – A. Traditional gradient-descent-based RNN training methods adapt all connection weights (bold arrows), including input-to-RNN, RNN-internal, and RNN-to-output weights. B. In Reservoir Computing, only the RNN-to-output weights are adapted.

Biological plausibility. Numerous connections of RC principles to architectural and dynamical properties of mammalian brains have been established. RC (or closely related models) provides explanations of why biological brains can carry out accurate computations with an "inaccurate" and noisy physical substrate [22,23], especially accurate timing [24]; of the way in which visual information is superimposed and processed in primary visual cortex [25,26]; of how cortico-basal pathways support the representation of sequential information; and RC offers a functional interpretation of the cerebellar circuitry [27,28]. A central role is assigned to an RC circuit in a series of models explaining sequential information processing in human and primate brains, most importantly of speech signals [13,29–31].

Extensibility and parsimony. A notorious conundrum of neural network research is how to extend previously learned models by new items without impairing or destroying previously learned representations (*catastrophic interference* [32]). RC offers a simple and principled solution: new items are represented by new output units, which are appended to the previously established output units of a given reservoir. Since the output weights of different output units are independent of each other, catastrophic interference is a non-issue.

These encouraging observations should not mask the fact that RC is still in its infancy, and significant further improvements and extensions are desirable. Specifically, just simply creating a reservoir at random is unsatisfactory. It seems obvious that, when addressing a specific modeling task, a specific reservoir design that is adapted to the task will lead to better results than a naive random creation. Thus, the main stream of research in the field is today directed at understanding the effects of reservoir characteristics on task performance, and at developing suitable reservoir design and adaptation methods. Also, new ways of reading out from the reservoirs, including combining them into larger structures, are devised and investigated. While shifting from the initial idea of having a fixed randomly created reservoir and training only the readout, the current paradigm of reservoir computing remains (and differentiates itself from other RNN training approaches) as producing/training the reservoir and the readout separately and differently.

This review offers a conceptual classification and a comprehensive survey of this research.

As is true for many areas of machine learning, methods in reservoir computing converge from different fields and come with different names. We would like to make a distinction here between these differently named "tradition lines", which we like to call *brands*, and the actual finer-grained ideas on producing good reservoirs, which we will call *recipes*. Since recipes can be useful and mixed across different brands, this review focuses on classifying and surveying them. To be fair, it has to be said that the authors of this survey associate themselves mostly with the Echo State Networks brand, and thus, willingly or not, are influenced by its mindset.

Overview. We start by introducing a generic notational framework in Section 2. More specifically, we define what we mean by *problem* or *task* in the context of machine learning in Section 2.1. Then we define a general notation for expansion (or kernel) methods for both non-temporal (Section 2.2) and temporal (Section 2.3) tasks, introduce our notation for recurrent neural networks in Section 2.4, and outline classical training methods in Section 2.5. In Section 3 we detail the foundations of Reservoir Computing and proceed by naming the most prominent brands. In Section 4 we introduce our classification of the reservoir generation/adaptation recipes, which transcends the boundaries between the brands. Following this classification we then review universal (Section 5), unsupervised (Section 6), and supervised (Section 7) reservoir generation/adaptation recipes. In Section 8 we provide a classification and review the techniques for reading the outputs from the reservoirs reported in literature, together with discussing various practical issues of readout training. A final discussion (Section 9) wraps up the entire picture.

2. Formalism

2.1. Formulation of the problem

Let a *problem* or a *task* in our context of machine learning be defined as a problem of learning a functional relation between a given input $\mathbf{u}(n) \in \mathbb{R}^{N_u}$ and a desired output $\mathbf{y}_{\text{target}}(n) \in \mathbb{R}^{N_y}$, where $n = 1, \ldots, T$, and T is the number of data points in the training *dataset* $\{(\mathbf{u}(n), \mathbf{y}_{\text{target}}(n))\}$. A *non-temporal* task is where the data points are independent of each other and the goal is to learn a function $\mathbf{y}(n) = y(\mathbf{u}(n))$ such that $E(\mathbf{y}, \mathbf{y}_{\text{target}})$ is minimized, where E is an error measure, for instance, the normalized root-mean-square error (NRMSE)

$$E(\mathbf{y}, \mathbf{y}_{\text{target}}) = \sqrt{\frac{\left\langle \left\| \mathbf{y}(n) - \mathbf{y}_{\text{target}}(n) \right\|^2 \right\rangle}{\left\langle \left\| \mathbf{y}_{\text{target}}(n) - \left\langle \mathbf{y}_{\text{target}}(n) \right\rangle \right\|^2 \right\rangle}}, \tag{1}$$

where $\|\cdot\|$ stands for the Euclidean distance (or norm).

A *temporal* task is where $\mathbf{u}$ and $\mathbf{y}_{\text{target}}$ are signals in a discrete time domain $n = 1, \ldots, T$, and the goal is to learn a function $\mathbf{y}(n) = y(\ldots, \mathbf{u}(n-1), \mathbf{u}(n))$ such that $E(\mathbf{y}, \mathbf{y}_{\text{target}})$ is minimized. Thus the difference between the temporal and non-temporal task is that the function $y(\cdot)$ we are trying to learn has memory in the first case and is memoryless in the second. In both cases the underlying assumption is, of course, that the functional dependence we are trying to learn actually exists in the data. For the temporal case this spells out as data adhering to an additive noise model of the form $\mathbf{y}_{\text{target}}(n) = y_{\text{target}}(\ldots, \mathbf{u}(n-1), \mathbf{u}(n)) + \boldsymbol{\theta}(n)$, where $y_{\text{target}}(\cdot)$ is the relation to be learned by $y(\cdot)$ and $\boldsymbol{\theta}(n) \in \mathbb{R}^{N_y}$ is a noise term, limiting the learning precision, i.e., the precision of matching the learned $\mathbf{y}(n)$ to $\mathbf{y}_{\text{target}}(n)$.

Whenever we say that the task or the problem is learned *well*, or with good *accuracy* or *precision*, we mean that $E(\mathbf{y}, \mathbf{y}_{\text{target}})$ is small. Normally one part of the T data points is used for training the model and another part (unseen during the training) for testing it. When speaking about output errors and *performance* or *precision* we will have *testing* errors in mind (if not explicitly specified otherwise). Also n, denoting the discrete time, will often be used omitting its range $1, \ldots, T$.

2.2. Expansions and kernels in non-temporal tasks

Many tasks cannot be accurately solved by a simple linear relation between the $\mathbf{u}$ and $\mathbf{y}_{\text{target}}$, i.e., a linear model $\mathbf{y}(n) = \mathbf{W}\mathbf{u}(n)$ (where $\mathbf{W} \in \mathbb{R}^{N_y \times N_u}$) gives big errors $E(\mathbf{y}, \mathbf{y}_{\text{target}})$ regardless of $\mathbf{W}$. In such situations one has to resort to *nonlinear* models. A number of generic and widely used approaches to nonlinear modeling are based on the idea of nonlinearly expanding the input $\mathbf{u}(n)$ into a high-dimensional feature vector $\mathbf{x}(n) \in \mathbb{R}^{N_x}$, and then utilizing those features using linear methods, for instance by linear regression or computing for a linear separation hyperplane, to get a reasonable $\mathbf{y}$. Solutions of this kind can be expressed in the form

$$\mathbf{y}(n) = \mathbf{W}_{\text{out}}\mathbf{x}(n) = \mathbf{W}_{\text{out}}x(\mathbf{u}(n)), \tag{2}$$

where $\mathbf{W}_{\text{out}} \in \mathbb{R}^{N_y \times N_x}$ are the trained output weights. Typically $N_x \gg N_u$, and we will often consider $\mathbf{u}(n)$ as included in $\mathbf{x}(n)$. There is also typically a constant *bias* value added to (2), which is omitted here and in other equations for brevity. The bias can be easily implemented, having one of the features in $\mathbf{x}(n)$ constant (e.g., $= 1$) and a corresponding column in $\mathbf{W}_{\text{out}}$. Some models extend (2) to

$$\mathbf{y}(n) = f_{\text{out}}(\mathbf{W}_{\text{out}}x[\mathbf{u}(n)]), \tag{3}$$

where $f_{\text{out}}(\cdot)$ is some nonlinear function (e.g., a sigmoid applied element-wise). For the sake of simplicity we will consider this definition as equivalent to (2), since $f_{\text{out}}(\cdot)$ can be eliminated from $\mathbf{y}$ by redefining the target as $\mathbf{y}'_{\text{target}} = f_{\text{out}}{}^{-1}(\mathbf{y}_{\text{target}})$ (and the error function $E(\mathbf{y}, \mathbf{y}'_{\text{target}})$, if desired). Note that (2) is a special case of (3), with $f_{\text{out}}(\cdot)$ being the identity.

Functions $x(\mathbf{u}(n))$ that transform an input $\mathbf{u}(n)$ into a (higher-dimensional) vector $\mathbf{x}(n)$ are often called *kernels* (and traditionally denoted $\phi(\mathbf{u}(n))$) in this context. Methods using kernels often employ the *kernel trick*, which refers to the option afforded by many kernels of computing inner products in the (high-dimensional, hence expensive) feature space of $\mathbf{x}$ more cheaply in the original space populated by $\mathbf{u}$. The term *kernel function* has acquired a close association with the kernel trick. Since here we will not exploit the kernel trick, in order to avoid confusion we will use the more neutral term of an *expansion* function for $x(\mathbf{u}(n))$, and refer to methods using such functions as *expansion methods*. These methods then include *Support Vector Machines* (which standardly do use the kernel trick), *Feedforward Neural Networks*, *Radial Basis Function* approximators, *Slow Feature Analysis*, and various *Probability Mixture* models, among many others. Feedforward neural networks are also often referred to as *(multilayer) perceptrons* in the literature.

While training the output $\mathbf{W}_{\text{out}}$ is a well defined and understood problem, producing a good expansion function $x(\cdot)$ generally involves more creativity. In many expansion methods, e.g., Support Vector Machines, the function is chosen "by hand" (most often through trial-and-error) and is fixed.

2.3. Expansions in temporal tasks

Many temporal methods are based on the same principle. The difference is that in a temporal task the function to be learned depends also on the history of the input, as discussed in Section 2.1. Thus, the expansion function has memory: $\mathbf{x}(n) = x(\ldots, \mathbf{u}(n-1), \mathbf{u}(n))$, i.e., it is an expansion of the current input and its (potentially infinite) history. Since this function has an unbounded number of parameters, practical implementations often take an alternative, recursive, definition:

$$\mathbf{x}(n) = x(\mathbf{x}(n-1), \mathbf{u}(n)). \tag{4}$$

The output $\mathbf{y}(n)$ is typically produced in the same way as for non-temporal methods by (2) or (3).

In addition to the nonlinear expansion, as in the non-temporal tasks, such $\mathbf{x}(n)$ could be seen as a type of a spatial embedding of the temporal information of $\ldots, \mathbf{u}(n-1), \mathbf{u}(n)$. This, for example, enables capturing higher-dimensional dynamical attractors $\mathbf{y}(n) = y_{\text{target}}(\ldots, \mathbf{u}(n-1), \mathbf{u}(n)) = \mathbf{u}(n+1)$ of the system being modeled by $y(\cdot)$ from a series of lower-dimensional observations $\mathbf{u}(n)$ the system is emitting, which is shown to be possible by *Takens's theorem* [33].

2.4. Recurrent neural networks

The type of recurrent neural networks that we will consider most of the time in this review is a straightforward implementation of (4). The nonlinear expansion with memory here leads to a *state vector* of the form

$$\mathbf{x}(n) = f(\mathbf{W}_{\text{in}}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1)), \quad n = 1, \ldots, T, \tag{5}$$

where $\mathbf{x}(n) \in \mathbb{R}^{N_x}$ is a vector of reservoir neuron activations at a time step n, $f(\cdot)$ is the neuron activation function, usually the symmetric $\tanh(\cdot)$, or the positive logistic (or Fermi) sigmoid, applied element-wise, $\mathbf{W}_{\text{in}} \in \mathbb{R}^{N_x \times N_u}$ is the input weight matrix, and $\mathbf{W} \in \mathbb{R}^{N_x \times N_x}$ is a weight matrix of internal network connections. The network is usually started with the initial state $\mathbf{x}(0) = \mathbf{0}$. Bias values are again omitted in (5) in

the same way as in (2). The readout $\mathbf{y}(n)$ of the network is implemented as in (3).

Some models of RNNs extend (5) as

$$\mathbf{x}(n) = f(\mathbf{W}_{\text{in}}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1) + \mathbf{W}_{\text{ofb}}\mathbf{y}(n-1)), \quad n = 1, \ldots, T, \tag{6}$$

where $\mathbf{W}_{\text{ofb}} \in \mathbb{R}^{N_x \times N_y}$ is an optional output feedback weight matrix.

2.5. Classical training of RNNs

The classical approach to supervised training of RNNs, known as *gradient descent*, is by iteratively adapting all weights $\mathbf{W}_{\text{out}}$, $\mathbf{W}$, $\mathbf{W}_{\text{in}}$, and possibly $\mathbf{W}_{\text{ofb}}$ (which as a whole we denote $\mathbf{W}_{\text{all}}$ for brevity) according to their estimated gradients $\partial E/\partial \mathbf{W}_{\text{all}}$, in order to minimize the output error $E = E(\mathbf{y}, \mathbf{y}_{\text{target}})$. A classical example of such methods is Real-Time Recurrent Learning [34], where the estimation of $\partial E/\partial \mathbf{W}_{\text{all}}$ is done recurrently, forward in time. Conversely, error backpropagation (BP) methods for training RNNs, which are derived as extensions of the BP method for feedforward neural networks [35], estimate $\partial E/\partial \mathbf{W}_{\text{all}}$ by propagating $E(\mathbf{y}, \mathbf{y}_{\text{target}})$ backwards through network connections and time. The BP group of methods is arguably the most prominent in classical RNN training, with the classical example in this group being Backpropagation Through Time [36]. It has a runtime complexity of $O(N_x{}^2)$ per weight update per time step for a single output $N_y = 1$, compared to $O(N_x{}^4)$ for Real-Time Recurrent Learning.

A systematic unifying overview of many classical gradient descent RNN training methods is presented in [37]. The same contribution also proposes a new approach, often referred to by others as Atiya–Parlos Recurrent Learning (APRL). It estimates gradients with respect to neuron activations $\partial E/\partial \mathbf{x}$ (instead of weights directly) and gradually adapts the weights $\mathbf{W}_{\text{all}}$ to move the activations $\mathbf{x}$ into the desired directions. The method is shown to converge faster than previous ones. See Section 3.4 for more implications of APRL and bridging the gap between the classical gradient descent and the reservoir computing methods.

There are also other versions of supervised RNN training, formulating the training problem differently, such as using Extended Kalman Filters [38] or the Expectation-Maximization algorithm [39], as well as dealing with special types of RNNs, such as Long Short-Term Memory [40] modular networks capable of learning long-term dependences.

There are many more, arguably less prominent, methods and their modifications for RNN training that are not mentioned here, as this would lead us beyond the scope of this review. The very fact of their multiplicity suggests that there is no clear winner in all aspects. Despite many advances that the methods cited above have introduced, they still have multiple common shortcomings, as pointed out in Section 1.

3. Reservoir methods

Reservoir computing methods differ from the "traditional" designs and learning techniques listed above in that they make a conceptual and computational separation between a dynamic *reservoir* — an RNN as a nonlinear temporal expansion function — and a recurrence-free (usually linear) *readout* that produces the desired output from the expansion.

This separation is based on the understanding (common with kernel methods) that $x(\cdot)$ and $y(\cdot)$ serve different purposes: $x(\cdot)$ expands the input history $\mathbf{u}(n), \mathbf{u}(n-1), \ldots$ into a rich enough reservoir state space $\mathbf{x}(n) \in \mathbb{R}^{N_x}$, while $y(\cdot)$ combines the neuron signals $\mathbf{x}(n)$ into the desired output signal $\mathbf{y}_{\text{target}}(n)$. In the linear readout case (2), for each dimension y_i of $\mathbf{y}$ an output weight vector $(\mathbf{W}_{\text{out}})_i$ in the same space $\mathbb{R}^{N_x}$ is found such that

$$(\mathbf{W}_{\text{out}})_i \mathbf{x}(n) = y_i(n) \approx y_{\text{target}_i}(n), \tag{7}$$

while the "purpose" of $\mathbf{x}(n)$ is to contain a rich enough representation to make this possible.

Since the expansion and the readout serve different purposes, training/generating them separately and even with different goal functions makes sense. The readout $\mathbf{y}(n) = y(\mathbf{x}(n))$ is essentially a non-temporal function, learning which is relatively simple. On the other hand, setting up the reservoir such that a "good" state expansion $\mathbf{x}(n)$ emerges is an ill-understood challenge in many respects. The "traditional" RNN training methods do not make the conceptual separation of a reservoir vs. a readout, and train both reservoir-internal and output weights in technically the same fashion. Nonetheless, even in traditional methods the ways of defining the error gradients for the output $\mathbf{y}(n)$ and the internal units $\mathbf{x}(n)$ are inevitably different, reflecting that an explicit target $\mathbf{y}_{\text{target}}(n)$ is available only for the output units. Analyses of traditional training algorithms have furthermore revealed that the learning dynamics of internal vs. output weights exhibit systematic and striking differences. This theme will be expanded in Section 3.4.

Currently, reservoir computing is a vivid fresh RNN research stream, which has recently gained wide attention due to the reasons pointed out in Section 1.

We proceed to review the most prominent "named" reservoir methods, which we call here *brands*. Each of them has its own history, a specific mindset, specific types of reservoirs, and specific insights.

3.1. Echo State Networks

Echo State Networks (ESNs) [16] represent one of the two pioneering reservoir computing methods. The approach is based on the observation that if a random RNN possesses certain algebraic properties, training only a linear readout from it is often sufficient to achieve excellent performance in practical applications. The untrained RNN part of an ESN is called a *dynamical reservoir*, and the resulting states $\mathbf{x}(n)$ are termed *echoes* of its input history [12]—this is where reservoir computing draws its name from.

ESNs standardly use simple sigmoid neurons, i.e., reservoir states are computed by (5) or (6), where the nonlinear function $f(\cdot)$ is a sigmoid, usually the $\tanh(\cdot)$ function. Leaky integrator neuron models represent another frequent option for ESNs, which is discussed in depth in Section 5.5. Classical recipes of producing the ESN reservoir (which is in essence $\mathbf{W}_{\text{in}}$ and $\mathbf{W}$) are outlined in Section 5.1, together with input-independent properties of the reservoir. Input-dependent measures of the

quality of the activations $\mathbf{x}(n)$ in the reservoir are presented in Section 6.1.

The readout from the reservoir is usually linear (3), where $\mathbf{u}(n)$ is included as part of $\mathbf{x}(n)$, which can also be spelled out in (3) explicitly as

$$\mathbf{y}(n) = f_{\text{out}}(\mathbf{W}_{\text{out}}[\mathbf{u}(n)|\mathbf{x}(n)]), \tag{8}$$

where $\mathbf{W}_{\text{out}} \in \mathbb{R}^{N_y \times (N_u + N_x)}$ is the learned output weight matrix, $f_{\text{out}}(\cdot)$ is the output neuron activation function (usually the identity) applied component-wise, and $\cdot|\cdot$ stands for a vertical concatenation of vectors. The original and most popular batch training method to compute $\mathbf{W}_{\text{out}}$ is linear regression, discussed in Section 8.1.1, or a computationally cheap online training discussed in Section 8.1.2.

The initial ESN publications [12,41–43,17] were framed in settings of machine learning and nonlinear signal processing applications. The original theoretical contributions of early ESN research concerned algebraic properties of the reservoir that make this approach work in the first place (the *echo state property* [12] discussed in Section 5.1) and analytical results characterizing the dynamical short-term memory capacity of reservoirs [41] discussed in Section 6.1.

3.2. *Liquid State Machines*

Liquid State Machines (LSMs) [11] are the other pioneering reservoir method, developed independently from and simultaneously with ESNs. LSMs were developed from a computational neuroscience background, aiming at elucidating the principal computational properties of neural microcircuits [11,20, 44,45]. Thus LSMs use more sophisticated and biologically realistic models of spiking integrate-and-fire neurons and dynamic synaptic connection models in the reservoir. The connectivity among the neurons often follows topological and metric constraints that are biologically motivated. In the LSM literature, the reservoir is often referred to as the *liquid*, following an intuitive metaphor of the excited states as ripples on the surface of a pool of water. Inputs to LSMs also usually consist of spike trains. In their readouts LSMs originally used multilayer feedforward neural networks (of either spiking or sigmoid neurons), or linear readouts similar to ESNs [11]. Additional mechanisms for averaging spike trains to get real-valued outputs are often employed.

RNNs of the LSM-type with spiking neurons and more sophisticated synaptic models are usually more difficult to implement, to correctly set up and tune, and typically more expensive to emulate on digital computers[3] than simple ESN-type "weighted sum and nonlinearity" RNNs. Thus they are less widespread for engineering applications of RNNs than the latter. However, while the ESN-type neurons only emulate mean firing rates of biological neurons, spiking neurons are able to perform more complicated information processing, due to the time coding of the information in their signals (i.e., the exact timing of each firing also matters). Also findings on various mechanisms in natural neural circuits are more easily transferable to these more biologically-realistic models (there is more on this in Section 6.2).

The main theoretical contributions of the LSM brand to Reservoir Computing consist in analytical characterizations of the computational power of such systems [11,21] discussed in Sections 6.1 and 7.4.

3.3. *Evolino*

Evolino [46] transfers the idea of ESNs from an RNN of simple sigmoidal units to a Long Short-Term Memory type of RNNs [40] constructed from units capable of preserving memory for long periods of time. In Evolino the weights of the reservoir are trained using evolutionary methods, as is also done in some extensions of ESNs, both discussed in Section 7.2.

3.4. *Backpropagation-Decorrelation*

The idea of separation between a reservoir and a readout function has also been arrived at from the point of view of optimizing the performance of the RNN training algorithms that use error backpropagation, as already indicated in Section 2.5. In an analysis of the weight dynamics of an RNN trained using the APRL learning algorithm [47], it was revealed that the output weights $\mathbf{W}_{\text{in}}$ of the network being trained change quickly, while the hidden weights $\mathbf{W}$ change slowly and in the case of a single output $N_y = 1$ the changes are column-wise coupled. Thus in effect APRL decouples the RNN into a quickly adapting output and a slowly adapting reservoir. Inspired by these findings a new iterative/online RNN training method, called *BackPropagation-DeCorrelation* (BPDC), was introduced [14]. It approximates and significantly simplifies the APRL method, and applies it only to the output weights $\mathbf{W}_{\text{out}}$, turning it into an online RC method. BPDC uses the reservoir update equation defined in (6), where output feedbacks $\mathbf{W}_{\text{ofb}}$ are essential, with the same type of units as ESNs. BPDC learning is claimed to be insensitive to the parameters of fixed reservoir weights $\mathbf{W}$. BPDC boasts fast learning times and thus is capable of tracking quickly changing signals. As a downside of this feature, the trained network quickly forgets the previously seen data and is highly biased by the recent data. Some remedies for reducing this effect are reported in [48]. Most of applications of BPDC in the literature are for tasks having one-dimensional outputs $N_y = 1$; however BPDC is also successfully applied to $N_y > 1$, as recently demonstrated in [49].

From a conceptual perspective we can define a range of RNN training methods that gradually bridge the gap between the classical BP and reservoir methods:

1. Classical BP methods, such as Backpropagation Through Time (BPTT) [36];
2. Atiya–Parlos recurrent learning (APRL) [37];
3. BackPropagation-DeCorrelation (BPDC) [14];
4. Echo State Networks (ESNs) [16].

In each method of this list the focus of training gradually moves from the entire network towards the output, and convergence of the training is faster in terms of iterations, with only a single "iteration" in case 4. At the same time the potential expressiveness of the RNN, as per the same number of units in the NN, becomes weaker. All methods in the list primarily use the same type of simple sigmoid neuron model.

[3] With a possible exception of event-driven spiking NN simulations, where the computational load varies depending on the amount of activity in the NN.

3.5. *Temporal Recurrent Networks*

This summary of RC brands would be incomplete without a spotlight directed at Peter F. Dominey's decade-long research suite on cortico-striatal circuits in the human brain (e.g., [13,29,31], and many more). Although this research is rooted in empirical cognitive neuroscience and functional neuroanatomy and aims at elucidating complex neural structures rather than theoretical computational principles, it is probably Dominey who first clearly spelled out the RC principle: *"(...) there is no learning in the recurrent connections* [within a subnetwork corresponding to a reservoir]*, only between the State* [i.e., reservoir] *units and the Output units. Second, adaptation is based on a simple associative learning mechanism (...)"* [50]. It is also in this article where Dominey brands the neural reservoir module as a *Temporal Recurrent Network*. The learning algorithm, to which Dominey alludes, can be seen as a version of the Least Mean Squares discussed in Section 8.1.2. At other places, Dominey emphasizes the randomness of the connectivity in the reservoir: *"It is worth noting that the simulated recurrent prefrontal network relies on fixed randomized recurrent connections, (...)"* [51]. Only in early 2008 did Dominey and "computational" RC researchers become aware of each other.

3.6. *Other (exotic) types of reservoirs*

As is clear from the discussion of the different reservoir methods so far, a variety of neuron models can be used for the reservoirs. Using different activation functions inside a single reservoir might also improve the richness of the echo states, as is illustrated, for example, by inserting some neurons with wavelet-shaped activation functions into the reservoir of ESNs [52]. A hardware implementation friendly version of reservoirs composed of stochastic bitstream neurons was proposed in [53].

In fact the reservoirs do not necessarily need to be neural networks, governed by dynamics similar to (5). Other types of high-dimensional dynamical systems that can take an input $\mathbf{u}(n)$ and have an observable state $\mathbf{x}(n)$ (which does not necessarily fully describe the state of the system) can be used as well. In particular this makes the reservoir paradigm suitable for harnessing the computational power of unconventional hardware, such as analog electronics [54,55], biological neural tissue [26], optical [56], quantum, or physical "computers". The last of these was demonstrated (taking the "reservoir" and "liquid" idea quite literally) by feeding the input via mechanical actuators into a reservoir full of water, recording the state of its surface optically, and successfully training a readout multilayer perceptron on several classification tasks [57]. An idea of treating a computer-simulated gene regulation network of *Escherichia Coli* bacteria as the reservoir, a sequence of chemical stimuli as an input, and measures of protein levels and mRNAs as an output is explored in [58].

3.7. *Other overviews of reservoir methods*

An experimental comparison of LSM, ESN, and BPDC reservoir methods with different neuron models, even beyond the standard ones used for the respective methods, and different parameter settings is presented in [59]. A brief and broad overview of reservoir computing is presented in [60], with an emphasis on applications and hardware implementations of reservoir methods. The editorial in the "Neural Networks" journal special issue on ESNs and LSMs [15] offers a short introduction to the topic and an overview of the articles in the issue (most of which are also surveyed here). An older and much shorter part of this overview, covering only reservoir adaptation techniques, is available as a technical report [61].

4. Our classification of reservoir recipes

The successes of applying RC methods to benchmarks (see the listing in Section 1) outperforming classical fully trained RNNs do not imply that randomly generated reservoirs are optimal and cannot be improved. In fact, "random" is almost by definition an antonym to "optimal". The results rather indicate the need for some novel methods of training/generating the reservoirs that are very probably not a direct extension of the way the output is trained (as in BP). Thus besides application studies (which are not surveyed here), the bulk of current RC research on reservoir methods is devoted to optimal reservoir design, or reservoir optimization algorithms.

It is worth mentioning at this point that the general "no free lunch" principle in supervised machine learning [62] states that there can exist no bias of a model which would universally improve the accuracy of the model for *all* possible problems. In our context this can be translated into a claim that no single type of reservoir can be optimal for all types of problems.

In this review we will try to survey all currently investigated ideas that help producing "good" reservoirs. We will classify those ideas into three major groups based on their universality:

- *Generic* guidelines/methods of producing good reservoirs irrespective of the task (both the input $\mathbf{u}(n)$ and the desired output $\mathbf{y}_{\text{target}}(n)$);
- *Unsupervised* pre-training of the reservoir with respect to the given input $\mathbf{u}(n)$, but not the target $\mathbf{y}_{\text{target}}(n)$;
- *Supervised* pre-training of the reservoir with respect to both the given input $\mathbf{u}(n)$ and the desired output $\mathbf{y}_{\text{target}}(n)$.

These three classes of methods are discussed in the following three sections. Note that many of the methods to some extend transcend the boundaries of these three classes, but will be classified according to their main principle.

5. Generic reservoir recipes

The most classical methods of producing reservoirs all fall into this category. All of them generate reservoirs randomly, with topology and weight characteristics depending on some preset parameters. Even though they are not optimized for a particular input $\mathbf{u}(n)$ or target $\mathbf{y}_{\text{target}}(n)$, a good manual selection of the parameters is to some extent task-dependent, complying with the "no free lunch" principle just mentioned.

5.1. *Classical ESN approach*

Some of the most generic guidelines of producing good reservoirs were presented in the papers that introduced

ESNs [12,42]. Motivated by an intuitive goal of producing a "rich" set of dynamics, the recipe is to generate a **(i)** *big*, **(ii)** *sparsely* and **(iii)** *randomly* connected, reservoir. This means that (i) N_x is sufficiently large, with order ranging from tens to thousands, (ii) the weight matrix $\mathbf{W}$ is sparse, with several to 20 per cent of possible connections, and (iii) the weights of the connections are usually generated randomly from a uniform distribution symmetric around the zero value. This design rationale aims at obtaining *many*, due to (i), reservoir activation signals, which are only *loosely coupled*, due to (ii), and *different*, due to (iii).

The input weights $\mathbf{W}_{\text{in}}$ and the optional output feedback weights $\mathbf{W}_{\text{ofb}}$ are usually dense (they can also be sparse like $\mathbf{W}$) and generated randomly from a uniform distribution. The exact scaling of both matrices and an optional shift of the input (a constant value added to $\mathbf{u}(n)$) are the few other free parameters that one has to choose when "baking" an ESN. The rules of thumb for them are the following. The scaling of $\mathbf{W}_{\text{in}}$ and shifting of the input depends on how much nonlinearity of the processing unit the task needs: if the inputs are close to 0, the tanh neurons tend to operate with activations close to 0, where they are essentially linear, while inputs far from 0 tend to drive them more towards saturation where they exhibit more nonlinearity. The shift of the input may help to overcome undesired consequences of the symmetry around 0 of the tanh neurons with respect to the sign of the signals. Similar effects are produced by scaling the bias inputs to the neurons (i.e., the column of $\mathbf{W}_{\text{in}}$ corresponding to constant input, which often has a different scaling factor than the rest of $\mathbf{W}_{\text{in}}$). The scaling of $\mathbf{W}_{\text{ofb}}$ is in practice limited by a threshold at which the ESN starts to exhibit an unstable behavior, i.e., the output feedback loop starts to amplify (the errors of) the output and thus enters a diverging generative mode. In [42], these and related pieces of advice are given without a formal justification.

An important element for ESNs to work is that the reservoir should have the *echo state property* [12]. This condition in essence states that the effect of a previous state $\mathbf{x}(n)$ and a previous input $\mathbf{u}(n)$ on a future state $\mathbf{x}(n+k)$ should vanish gradually as time passes (i.e., $k \to \infty$), and not persist or even get amplified. For most practical purposes, the echo state property is assured if the reservoir weight matrix $\mathbf{W}$ is scaled so that its spectral radius $\rho(\mathbf{W})$ (i.e., the largest absolute eigenvalue) satisfies $\rho(\mathbf{W}) < 1$ [12]. Or, using another term, $\mathbf{W}$ is *contractive*. The fact that $\rho(\mathbf{W}) < 1$ almost always ensures the echo state property has led to an unfortunate misconception which is expressed in many RC publications, namely, that $\rho(\mathbf{W}) < 1$ amounts to a necessary and sufficient condition for the echo state property. This is wrong. The mathematically correct connection between the spectral radius and the echo state property is that the latter is violated if $\rho(\mathbf{W}) > 1$ *in reservoirs using the* tanh *function as neuron nonlinearity, and for zero input*. Contrary to widespread misconceptions, the echo state property can be obtained even if $\rho(\mathbf{W}) > 1$ for non-zero input (including bias inputs to neurons), and it may be lost even if $\rho(\mathbf{W}) < 1$, although it is hard to construct systems where this occurs (unless $f'(0) > 1$ for the nonlinearity f), and in practice this does not happen.

The optimal value of $\rho(\mathbf{W})$ should be set depending on the amount of memory and nonlinearity that the given task requires. A rule of thumb, likewise discussed in [12], is that $\rho(\mathbf{W})$ should be close to 1 for tasks that require long memory and accordingly smaller for the tasks where a too long memory might in fact be harmful. Larger $\rho(\mathbf{W})$ also have the effect of driving signals $\mathbf{x}(n)$ into more nonlinear regions of tanh units (further from 0) similarly to $\mathbf{W}_{\text{in}}$. Thus scalings of both $\mathbf{W}_{\text{in}}$ and $\mathbf{W}$ have a similar effect on nonlinearity of the ESN, while their difference determines the amount of memory.

A rather conservative rigorous *sufficient* condition of the echo state property for any kind of inputs $\mathbf{u}(n)$ (including zero) and states $\mathbf{x}(n)$ (with tanh nonlinearity) being $\sigma_{\max}(\mathbf{W}) < 1$, where $\sigma_{\max}(\mathbf{W})$ is the largest singular value of $\mathbf{W}$, was proved in [12]. Recently, a less restrictive sufficient condition, namely, $\inf_{\mathbf{D}\in\mathcal{D}} \sigma_{\max}(\mathbf{DWD}^{-1}) < 1$, where $\mathbf{D}$ is an arbitrary matrix, minimizing the so-called $\mathbf{D}$-norm $\sigma_{\max}(\mathbf{DWD}^{-1})$, from a set $\mathcal{D} \subset \mathbb{R}^{N_x \times N_x}$ of diagonal matrices, has been derived in [63]. This sufficient condition approaches the necessary $\inf_{\mathbf{D}\in\mathcal{D}} \sigma_{\max}(\mathbf{DWD}^{-1}) \to \rho(\mathbf{W})^-$, $\rho(\mathbf{W}) < 1$, e.g., when $\mathbf{W}$ is a normal or a triangular (permuted) matrix. A rigorous sufficient condition for the echo state property is rarely ensured in practice, with a possible exception being critical control tasks, where provable stability under any conditions is required.

5.2. *Different topologies of the reservoir*

There have been attempts to find topologies of the ESN reservoir different from sparsely randomly connected ones. Specifically, small-world [64], scale-free [65], and biologically inspired connection topologies generated by spatial growth [66] were tested for this purpose in a careful study [67], which we point out here due to its relevance although it was obtained only as a BSc thesis. The NRMS error (1) of $\mathbf{y}(n)$ as well as the eigenvalue spread of the cross-correlation matrix of the activations $\mathbf{x}(n)$ (necessary for a fast online learning described in Section 8.1.2; see Section 6.1 for details) were used as the performance measures of the topologies. This work also explored an exhaustive brute-force search of topologies of tiny networks (motifs) of four units, and then combining successful motives (in terms of the eigenvalue spread) into larger networks. The investigation, unfortunately, concludes that "*(...) none of the investigated network topologies was able to perform significantly better than simple random networks, both in terms of eigenvalue spread as well as testing error*" [67]. This, however, does not serve as a proof that similar approaches are futile. An indication of this is the substantial variation in ESN performance observed among randomly created reservoirs, which is, naturally, more pronounced in smaller reservoirs (e.g., [68]).

In contrast, LSMs often use a biologically plausible connectivity structure and weight settings. In the original form they model a single cortical microcolumn [11]. Since the model of both the connections and the neurons themselves is quite sophisticated, it has a large number of free parameters to be set, which is done manually, guided by biologically observed parameter ranges, e.g., as found in the rat somatosensory cortex [69]. This type of model also delivers good performance for practical applications of speech recognition [69,70] (and many similar publications

by the latter authors). Since LSMs aim at accuracy of modeling natural neural structures, less biologically plausible connectivity patterns are usually not explored.

It has been demonstrated that much more detailed biological neural circuit models, which use anatomical and neurophysiological data-based laminar (i.e., cortical layer) connectivity structures and Hodgkin–Huxley model neurons, improve the information-processing capabilities of the models [23]. Such highly realistic (for present-day standards) models *"perform significantly better than control circuits (which are lacking the laminar structures but are otherwise identical with regard to their components and overall connection statistics) for a wide variety of fundamental information-processing tasks"* [23].

Different from this direction of research, there are also explorations of using even simpler topologies of the reservoir than the classical ESN. It has been demonstrated that the reservoir can even be an unstructured feed-forward network with time-delayed connections if the finite limited memory window that it offers is sufficient for the task at hand [71]. A degenerate case of a "reservoir" composed of linear units and a diagonalized $\mathbf{W}$ and unitary inputs $\mathbf{W}_{\text{in}}$ was considered in [72]. A one-dimensional lattice (ring) topology was used for a reservoir, together with an adaptation of the reservoir discussed in Section 6.2, in [73]. A special kind of excitatory and inhibitory neurons connected in a one-dimensional spatial arrangement was shown to produce interesting chaotic behavior in [74].

A tendency that higher ranks of the connectivity matrix $\mathbf{W}_{\text{mask}}$ (where $w_{\text{mask}i,j} = 1$ if $w_{i,j} \neq 0$, and $= 0$ otherwise, for $i, j = 1, \ldots, N_{\mathbf{x}}$) correlate with lower ESN output errors was observed in [75]. Connectivity patterns of $\mathbf{W}$ such that $\mathbf{W}^{\infty} \equiv \lim_{k\to\infty} \mathbf{W}^k$ ($\mathbf{W}^k$ standing for "$\mathbf{W}$ to the power k" and approximating weights of the cumulative indirect connections by paths of length k among the reservoir units) is neither fully connected, nor all-zero, are claimed to give a broader distribution of ESN prediction performances, thus including best performing reservoirs, than random sparse connectivities in [76]. A permutation matrix with a medium number and different lengths of connected cycles, or a general orthogonal matrix, are suggested as candidates for such $\mathbf{W}$s.

5.3. *Modular reservoirs*

One of the shortcomings of conventional ESN reservoirs is that, even though they are sparse, the activations are still coupled so strongly that the ESN is poor in dealing with different time scales simultaneously, e.g., predicting several superimposed generators. This problem was successfully tackled by dividing the reservoir into decoupled sub-reservoirs and introducing inhibitory connections among all the sub-reservoirs [77]. For the approach to be effective, the inhibitory connections must predict the activations of the sub-reservoirs one time step ahead. To achieve this the inhibitory connections are heuristically computed from (the rest of) $\mathbf{W}$ and $\mathbf{W}_{\text{ofb}}$, or the sub-reservoirs are updated in a sequence and the real activations of the already updated sub-reservoirs are used.

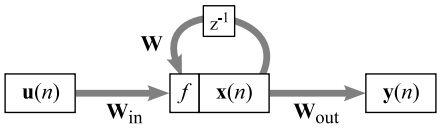

Fig. 2 – Signal flow diagram of the standard ESN.

The Evolino approach introduced in Section 3.3 can also be classified as belonging to this group, as the LSTM RNN used for its reservoir consists of specific small memory-holding modules (which could alternatively be regarded as more complicated units of the network).

Approaches relying on combining outputs from several separate reservoirs will be discussed in Section 8.8.

5.4. *Time-delayed vs. instantaneous connections*

Another time-related limitation of the classical ESNs pointed out in [78] is that no matter how many neurons are contained in the reservoir, it (like any other fully recurrent network with all connections having a time delay) has only a single layer of neurons (Fig. 2). This makes it intrinsically unsuitable for some types of problems. Consider a problem where the mapping from $\mathbf{u}(n)$ to $\mathbf{y}_{\text{target}}(n)$ is a very complex, nonlinear one, and the data in neighboring time steps are almost independent (i.e., little memory is required), as e.g., the "meta-learning" task in [79].[4] Consider a single time step n: signals from the input $\mathbf{u}(n)$ propagate only through one untrained layer of weights $\mathbf{W}_{\text{in}}$, through the nonlinearity f influence the activations $\mathbf{x}(n)$, and reach the output $\mathbf{y}(n)$ through the trained weights $\mathbf{W}_{\text{out}}$ (Fig. 2). Thus ESNs are not capable of producing a very complex *instantaneous* mapping from $\mathbf{u}(n)$ to $\mathbf{y}(n)$ using a realistic number of neurons, which could (only) be effectively done by a multilayer FFNN (not counting some non-NN-based methods). Delaying the target $\mathbf{y}_{\text{target}}$ by k time steps would in fact make the signals coming from $\mathbf{u}(n)$ "cross" the nonlinearities $k+1$ times before reaching $\mathbf{y}(n+k)$, but would mix the information from different time steps in $\mathbf{x}(n), \ldots, \mathbf{x}(n+k)$, breaking the required virtually independent mapping $\mathbf{u}(n) \rightarrow \mathbf{y}_{\text{target}}(n+k)$, if no special structure of $\mathbf{W}$ is imposed.

As a possible remedy Layered ESNs were introduced in [78], where a part (up to almost half) of the reservoir connections can be instantaneous and the rest take one time step for the signals to propagate as in normal ESNs. Randomly generated Layered ESNs, however, do not offer a consistent improvement for large classes of tasks, and pre-training methods of such reservoirs have not yet been investigated.

The issue of standard ESNs not having enough trained layers is also discussed and addressed in a broader context in Section 8.8.

5.5. *Leaky integrator neurons and speed of dynamics*

In addition to the basic sigmoid units, leaky integrator neurons were suggested to be used in ESNs from the point of their introduction [12]. This type of neuron performs a

[4] ESNs have been shown to perform well in a (significantly) simpler version of the "meta-learning" in [80].

leaky integration of its activation from previous time steps. Today a number of versions of leaky integrator neurons are often used in ESNs, which we will call here *leaky integrator ESNs* (LI-ESNs) where the distinction is needed. The main two groups are those using leaky integration before application of the activation function $f(\cdot)$, and after. One example of the latter (in the discretized time case) has reservoir dynamics governed by

$$\mathbf{x}(n) = (1 - a\Delta t)\mathbf{x}(n-1) + \Delta t f(\mathbf{W}_{\text{in}}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1)), \tag{9}$$

where Δt is a compound time gap between two consecutive time steps divided by the time constant of the system and a is the decay (or leakage) rate [81]. Another popular (and we believe, preferable) design can be seen as setting $a = 1$ and redefining δt in Eq. (9) as the leaking rate a to control the "speed" of the dynamics,

$$\mathbf{x}(n) = (1 - a)\mathbf{x}(n-1) + af(\mathbf{W}_{\text{in}}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1)), \tag{10}$$

which in effect is an exponential moving average, has only one additional parameter and the desirable property that neuron activations $\mathbf{x}(n)$ never go outside the boundaries defined by $f(\cdot)$. Note that the simple ESN (5) is a special case of LI-ESNs (9) or (10) with $a = 1$ and $\Delta t = 1$. As a corollary, an LI-ESN with a good choice of the parameters can always perform *at least* as well as a corresponding simple ESN. With the introduction of the new parameter a (and Δt), the condition for the echo state property is redefined [12]. A natural constraint on the two new parameters is $a\Delta t \in [0, 1]$ in (9), and $a \in [0, 1]$ in (10) — a neuron should neither retain, nor leak, more activation than it had. The effect of these parameters on the final performance of ESNs was investigated in [18] and [82]. The latter contribution also considers applying the leaky integrator in different places of the model and resampling the signals as an alternative.

The additional parameters of the LI-ESN control the "speed" of the reservoir dynamics. Small values of a and Δt result in reservoirs that react slowly to the input. By changing these parameters it is possible to shift the effective interval of frequencies in which the reservoir is working. Along these lines, time warping invariant ESNs (TWIESNs) — an architecture that can deal with strongly time-warped signals — were outlined in [81,18]. This architecture varies Δt on-the-fly in (9), directly depending on the speed at which the input $\mathbf{u}(n)$ is changing.

From a signal processing point of view, the exponential moving average on the neuron activation (10) does a simple *low-pass* filtering of its activations with the cutoff frequency

$$f_c = \frac{a}{2\pi(1-a)\Delta t}, \tag{11}$$

where Δt is the discretization time step. This makes the neurons average out the frequencies above f_c and enables tuning the reservoirs for particular frequencies. Elaborating further on this idea, *high-pass* neurons, that produce their activations by subtracting from the unfiltered activation (5) the low-pass filtered one (10), and *band-pass* neurons, that combine the low-pass and high-pass ones, were introduced [83]. The authors also suggested mixing neurons with different passbands inside a single ESN reservoir, and reported that a single reservoir of such kind is able to predict/generate signals having structure on different timescales.

Following this line of thought, Infinite Impulse Response (IIR) band-pass filters having sharper cutoff characteristics were tried on neuron activations in ESNs with success in several types of signals [84]. Since the filters often introduce an undesired phase shift to the signals, a time delay for the activation of each neuron was learned and applied before the linear readout from the reservoir. A successful application of Butterworth band-pass filters in ESNs is reported in [85].

Connections between neurons that have different time delays (more than one time step) can actually also be used inside the recurrent part, which enables the network to operate on different timescales simultaneously and learn longer-term dependences [86]. This idea has been tried for RNNs trained by error backpropagation, but could also be useful for multi-timescale reservoirs. Long-term dependences can also be learned using the reservoirs mentioned in Section 3.3.

6. Unsupervised reservoir adaptation

In this section we describe reservoir training/generation methods that try to optimize some measure defined on the activations $\mathbf{x}(n)$ of the reservoir, for a given input $\mathbf{u}(n)$, but regardless of the desired output $\mathbf{y}_{\text{target}}(n)$. In Section 6.1 we survey measures that are used to estimate the quality of the reservoir, irrespective of the methods optimizing them. Then local, Section 6.2, and global, Section 6.3 unsupervised reservoir training methods are surveyed.

6.1. "Goodness" measures of the reservoir activations

The classical feature that reservoirs should possess is the echo state property, defined in Section 5.1. Even though this property depends on the concrete input $\mathbf{u}(n)$, usually in practice its existence is not measured explicitly, and only the spectral radius $\rho(\mathbf{W})$ is selected to be <1 irrespective of $\mathbf{u}(n)$, or just tuned for the final performance. A measure of short-term *memory capacity*, evaluating how well $\mathbf{u}(n)$ can be reconstructed by the reservoir as $\mathbf{y}(n+k)$ after various delays k, was introduced in [41].

The two necessary and sufficient conditions for LSMs to work were introduced in [11]. A *separation property* measures the distance between different states $\mathbf{x}$ caused by different input sequences $\mathbf{u}$. The measure is refined for binary ESN-type reservoirs in [87] with a generalization in [88]. An *approximation property* measures the capability of the readout to produce a desired output $\mathbf{y}_{\text{target}}$ from $\mathbf{x}$, and thus is not an unsupervised measure, but is included here for completeness.

Methods for estimating the computational power and generalization capability of neural reservoirs were presented in [89]. The proposed measure for computational power, or *kernel quality*, is obtained in the following way. Take k different input sequences (or segments of the same signal) $\mathbf{u}^i(n)$, where $i = 1, \ldots, k$, and $n = 1, \ldots, T_k$. For each input i take the resulting reservoir state $\mathbf{x}^i(n_0)$, and collect them into a matrix $\mathbf{M} \in \mathbb{R}^{k \times N_x}$, where n_0 is some fixed time after the appearance of $\mathbf{u}^i(n)$ in the input. Then the rank r of the matrix $\mathbf{M}$ is

the measure. If $r = k$, this means that all the presented inputs can be separated by a linear readout from the reservoir, and thus the reservoir is said to have a *linear separation property*. For estimating the generalization capability of the reservoir, the same procedure can be performed with s ($s \gg k$) inputs $\mathbf{u}^j(n)$, $j = 1, \ldots, s$, that represent the set of all possible inputs. If the resultant rank r is substantially smaller than the size s of the training set, the reservoir generalizes well. These two measures are more targeted to tasks of time series classification, but can also be revealing in predicting the performance of regression [90].

A much-desired measure to minimize is the eigenvalue spread (EVS, the ratio of the maximal eigenvalue to the minimal eigenvalue) of the cross-correlation matrix of the activations $\mathbf{x}(n)$. A small EVS is necessary for an online training of the ESN output by a computationally cheap and stable stochastic gradient descent algorithm outlined in Section 8.1.2 (see, e.g., [91], chapter 5.3, for the mathematical reasons that render this mandatory). In classical ESNs the EVS sometimes reaches 10^{12} or even higher [92], which makes the use of stochastic gradient descent training unfeasible. Other commonly desirable features of the reservoir are small pairwise correlation of the reservoir activations $x_i(n)$, or a large entropy of the $\mathbf{x}(n)$ distribution (e.g., [92]). The latter is a rather popular measure, as discussed later in this review. A criterion for maximizing the local information transmission of each individual neuron was investigated in [93] (more in Section 6.2).

The so-called *edge of chaos* is a region of parameters of a dynamical system at which it operates at the boundary between the chaotic and non-chaotic behavior. It is often claimed (but not undisputed; see, e.g., [94]) that at the edge of chaos many types of dynamical systems, including binary systems and reservoirs, possess high computational power [87,95]. It is intuitively clear that the edge of chaos in reservoirs can only arise when the effect of inputs on the reservoir state does not die out quickly; thus such reservoirs can potentially have high memory capacity, which is also demonstrated in [95]. However, this does not universally imply that such reservoirs are optimal [90]. The edge of chaos can be empirically detected (even for biological networks) by measuring Lyapunov exponents [95], even though such measurements are not trivial (and often involve a degree of expert judgment) for high-dimensional noisy systems. For reservoirs of simple binary threshold units this can be done more simply by computing the Hamming distances between trajectories of the states [87]. There is also an empirical observation that, while changing different parameter settings of a reservoir, the best performance in a given task correlates with a Lyapunov exponent specific to that task [59]. The optimal exponent is related to the amount of memory needed for the task, as discussed in Section 5.1. It was observed in ESNs with no input that when $\rho(\mathbf{W})$ is slightly greater than 1, the internally generated signals are periodic oscillations, whereas for larger values of $\rho(\mathbf{W})$, the signals are more irregular and even chaotic [96]. Even though stronger inputs $\mathbf{u}(n)$ can push the dynamics of the reservoirs out of the chaotic regime and thus make them useful for computation, no reliable benefit of such a mode of operation was found in the last contribution.

In contrast to ESN-type reservoirs of real-valued units, simple binary threshold units exhibit a more immediate transition from damped to chaotic behavior without intermediate periodic oscillations [87]. This difference between the two types of activation functions, including intermediate *quantized* ones, in ESN-type reservoirs was investigated more closely in [88]. The investigation showed that reservoirs of binary units are more sensitive to the topology and the connection weight parameters of the network in their transition between damped and chaotic behavior, and computational performance, than the real-valued ones. This difference can be related to the similar apparent difference in sensitivity of the ESNs and LSM-type reservoirs of firing units, discussed in Section 5.2.

6.2. Unsupervised local methods

A natural strategy for improving reservoirs is to mimic biology (at a high level of abstraction) and count on *local* adaptation rules. "Local" here means that parameters pertaining to some neuron i are adapted on the basis of no other information than the activations of neurons directly connected with neuron i. In fact all local methods are almost exclusively unsupervised, since the information on the performance E at the output is unreachable in the reservoir.

First attempts to decrease the eigenvalue spread in ESNs by classical Hebbian [97] (inspired by synaptic plasticity in biological brains) or Anti-Hebbian learning gave no success [92]. A modification of Anti-Hebbian learning, called *Anti-Oja* learning, is reported to improve the performance of ESNs in [98].

On the more biologically realistic side of the RC research with spiking neurons, local unsupervised adaptations are very natural to use. In fact, LSMs had used synaptic connections with realistic short-term dynamic adaptation, as proposed by [99], in their reservoirs from the very beginning [11].

The Hebbian learning principle is usually implemented in spiking NNs as *spike-time-dependent plasticity* (STDP) of synapses. STDP is shown to improve the separation property of LSMs for real-world speech data, but not for random inputs $\mathbf{u}$, in [100]. The authors however were uncertain whether manually optimizing the parameters of the STDP adaptation (which they did) or the ones for generating the reservoir would result in a larger performance gain for the same effort spent. STDP is shown to work well with time-coded readouts from the reservoir in [101].

Biological neurons are widely observed to adapt their intrinsic excitability, which often results in exponential distributions of firing rates, as observed in visual cortex (e.g., [102]). This homeostatic adaptation mechanism, called *intrinsic plasticity* (IP), has recently attracted a wide attention in the reservoir computing community. Mathematically, the exponential distribution maximizes the entropy of a non-negative random variable with a fixed mean; thus it enables the neurons to transmit maximal information for a fixed metabolic cost of firing. An IP learning rule for spiking model neurons aimed at this goal was first presented in [103].

For a more abstract model of the neuron, having a continuous Fermi sigmoid activation function $f : \mathbb{R} \rightarrow$

$(0, 1)$, the IP rule was derived as a proportional control that changes the steepness and offset of the sigmoid to get an exponential-like output distribution in [104]. A more elegant gradient IP learning rule for the same purpose was presented in [93], which is similar to the information maximization approach in [105]. Applying IP with Fermi neurons in reservoir computing significantly improves the performance of BPDC-trained networks [106,107], and is shown to have a positive effect on offline trained ESNs, but can cause stability problems for larger reservoirs [106]. An ESN reservoir with IP-adapted Fermi neurons is also shown to enable predicting several superimposed oscillators [108].

An adaptation of the IP rule to tanh neurons ($f : \mathbb{R} \to (-1, 1)$) that results in a zero-mean Gaussian-like distribution of activations was first presented in [73] and investigated more in [55]. The IP-adapted ESNs were compared with classical ones, both having Fermi and tanh neurons, in the latter contribution. IP was shown to (modestly) improve the performance in all cases. It was also revealed that ESNs with Fermi neurons have significantly smaller short-term memory capacity (as in Section 6.1) and worse performance in a synthetic NARMA prediction task, while having a slightly better performance in a speech recognition task, compared to tanh neurons. The same type of tanh neurons adapted by IP aimed at Laplacian distributions are investigated in [109]. In general, IP gives more control on the working points of the reservoir nonlinearity sigmoids. The slope (first derivative) and the curvature (second derivative) of the sigmoid at the point around which the activations are centered by the IP rule affect the effective spectral radius and the nonlinearity of the reservoir, respectively. Thus, for example, centering tanh activations around points other than 0 is a good idea if no quasi-linear behavior is desired. IP has recently become employed in reservoirs as a standard practice by several research groups.

Overall, an information-theoretic view on adaptation of spiking neurons has a long history in computational neuroscience. Even better than maximizing just any information in the output of a neuron is maximizing *relevant* information. In other words, in its output the neuron should encode the inputs in such a way as to preserve maximal information about some (local) target signal. This is addressed in a general information-theoretical setting by the *Information Bottleneck* (IB) method [110]. A learning rule for a spiking neuron that maximizes mutual information between its inputs and its output is presented in [111]. A more general IB learning rule, transferring the general ideas of IB method to spiking neurons, is introduced in [112] and [113]. Two *semi-local* training scenarios are presented in these two contributions. In the first, a neuron optimizes the mutual information of its output with outputs of some neighboring neurons, while minimizing the mutual information with its inputs. In the second, two neurons reading from the same signals maximize their information throughput, while keeping their inputs statistically independent, in effect performing Independent Component Analysis (ICA). A simplified online version of the IB training rule with a variation capable of performing Principle Component Analysis (PCA) was recently introduced in [114]. In addition, it assumes slow semi-local target signals, which is more biologically plausible. The approaches described in this paragraph are still waiting to be tested in the reservoir computing setting.

It is also of great interest to understand how different types of plasticity observed in biological brains interact when applied together and what effect this has on the quality of reservoirs. The interaction of the IP with Hebbian synaptic plasticity in a single Fermi neuron is investigated in [104] and further in [115]. The synergy of the two plasticities is shown to result in a better specialization of the neuron that finds heavy-tail directions in the input. An interaction of IP with a neighborhood-based Hebbian learning in a layer of such neurons was also shown to maximize information transmission, perform nonlinear ICA, and result in an emergence of orientational Gabor-like receptive fields in [116]. The interaction of STDP with IP in an LSM-like reservoir of simple sparsely spiking neurons was investigated in [117]. The interaction turned out to be a non-trivial one, resulting in networks more robust to perturbations of the state $\mathbf{x}(n)$ and having a better short-time memory and time series prediction performance.

A recent approach of combining STDP with a biologically plausible reinforcement signal is discussed in Section 7.5, as it is not unsupervised.

6.3. *Unsupervised global methods*

Here we review unsupervised methods that optimize reservoirs based on *global* information of the reservoir activations induced by the given input $\mathbf{u}(x)$, but irrespective of the target $\mathbf{y}_{\text{target}}(n)$, like for example the measures discussed in Section 6.1. The intuitive goal of such methods is to produce good representations of (the history of) $\mathbf{u}(n)$ in $\mathbf{x}(n)$ for any (and possibly several) $\mathbf{y}_{\text{target}}(n)$.

A biologically inspired unsupervised approach with a reservoir trying to predict itself is proposed in [118]. An additional output $\mathbf{z}(n) \in \mathbb{R}^{N_x}$, $\mathbf{z}(n) = \mathbf{W}_z\mathbf{x}(n)$ from the reservoir is trained on the target $\mathbf{z}_{\text{target}}(n) = \mathbf{x}'(n+1)$, where $\mathbf{x}'(n)$ are the activations of the reservoir before applying the neuron transfer function $\tanh(\cdot)$, i.e., $\mathbf{x}(n) = \tanh(\mathbf{x}'(n))$. Then, in the application phase of the trained networks, the original activations $\mathbf{x}'(n)$, which result from $\mathbf{u}(n)$, $\mathbf{W}_{\text{in}}$, and $\mathbf{W}$, are mixed with the self-predictions $\mathbf{z}(n-1)$ obtained from $\mathbf{W}_z$, with a certain mixing ratio $(1-\alpha) : \alpha$. The coefficient α determines how much the reservoir is relying on the external input $\mathbf{u}(n)$ and how much on the internal self-prediction $\mathbf{z}(n)$. With $\alpha = 0$ we have the classical ESN and with $\alpha = 1$ we have an "autistic" reservoir that does not react to the input. Intermediate values of α close to 1 were shown to enable reservoirs to generate slow, highly nonlinear signals that are hard to get otherwise.

An algebraic unsupervised way of generating ESN reservoirs was proposed in [119]. The idea is to linearize the ESN update equation (5) locally around its current state $\mathbf{x}(n)$ at every time step n to get a linear approximation of (5) as $\mathbf{x}(n+1) = \mathbf{A}\mathbf{x}(n) + \mathbf{B}\mathbf{u}(n)$, where $\mathbf{A}$ and $\mathbf{B}$ are time (n)-dependent matrices corresponding to $\mathbf{W}$ and $\mathbf{W}_{\text{in}}$ respectively. The approach aims at distributing the predefined complex eigenvalues of $\mathbf{A}$ uniformly within the unit circle on the $\mathbb{C}$ plane. The reservoir matrix $\mathbf{W}$ is obtained analytically

from the set of these predefined eigenvalues and a given input $\mathbf{u}(n)$. The motivation for this is, as for Kautz filters [120] in linear systems, that if the target $\mathbf{y}_{\text{target}}(n)$ is unknown, it is best to have something like an orthogonal basis in $\mathbf{x}(n)$, from which any $\mathbf{y}_{\text{target}}(n)$ could, on average, be constructed well. The spectral radius of the reservoir is suggested to be set by hand (according to the correlation time of $\mathbf{u}(n)$, which is an indication of a memory span needed for the task), or by adapting the bias value of the reservoir units to minimize the output error (which actually renders this method *supervised*, as in Section 7). Reservoirs generated this way are shown to yield higher average entropy of $\mathbf{x}(n)$ distribution, higher short-term memory capacity (both measures mentioned in Section 6.1), and a smaller output error on a number of synthetic problems, using relatively small reservoirs ($N_{\mathbf{x}} = 20, 30$). However, a more extensive empirical comparison of this type of reservoir with the classical ESN one is still lacking.

7. Supervised reservoir pre-training

In this section we discuss methods for training reservoirs to perform a specific given task, i.e., not only the concrete input $\mathbf{u}(n)$, but also the desired output $\mathbf{y}_{\text{target}}(n)$ is taken into account. Since a linear readout from a reservoir is quickly trained, the suitability of a candidate reservoir for a particular task (e.g., in terms of NRMSE (1)) is inexpensive to check. Notice that even for most methods of this class the explicit target signal $\mathbf{y}_{\text{target}}(n)$ is not technically required for training the reservoir itself, but only for evaluating it in an outer loop of the adaptation process.

7.1. Optimization of global reservoir parameters

In Section 5.1 we discussed guidelines for the manual choice of global parameters for reservoirs of ESNs. This approach works well only with experience and a good intuitive grasp on nonlinear dynamics. A systematic gradient descent method of optimizing the global parameters of LI-ESNs (recalled from Section 5.5) to fit them to a given task is presented in [18]. The investigation shows that the error surfaces in the combined global parameter and $\mathbf{W}_{\text{out}}$ spaces may have very high curvature and multiple local minima. Thus, gradient descent methods are not always practical.

7.2. Evolutionary methods

As one can see from the previous sections of this review, optimizing reservoirs is generally challenging, and breakthrough methods remain to be found. On the other hand, checking the performance of a resulting ESN is relatively inexpensive, as said. This brings in *evolutionary* methods for the reservoir pre-training as a natural strategy.

Recall that the classical method generates a reservoir randomly; thus the performance of the resulting ESN varies slightly (and for small reservoirs not so slightly) from one instance to another. Then indeed, an "evolutionary" method as naive as "generate k reservoirs, pick the best" will outperform the classical method ("generate a reservoir") with probability $(k-1)/k$, even though the improvement might be not striking.

Several evolutionary approaches on optimizing reservoirs of ESNs are presented in [121]. The first approach was to carry out an evolutionary search on the parameters for generating $\mathbf{W}$: $N_{\mathbf{x}}$, $\rho(\mathbf{W})$, and the connection density of $\mathbf{W}$. Then an evolutionary algorithm [122] was used on individuals consisting of all the weight matrices $(\mathbf{W}_{\text{in}}, \mathbf{W}, \mathbf{W}_{\text{ofb}})$ of small ($N_{\mathbf{x}} = 5$) reservoirs. A variant with a reduced search space was also tried where the weights, but not the topology, of $\mathbf{W}$ were explored, i.e., elements of $\mathbf{W}$ that were zero initially always stayed zero. The empirical results of modeling the motion of an underwater robot showed superiority of the methods over other state-of-art methods, and that the topology-restricted adaptation of $\mathbf{W}$ is almost as effective as the full one.

Another approach of optimizing the reservoir $\mathbf{W}$ by a greedy evolutionary search is presented in [75]. Here the same idea of separating the topology and weight sizes of $\mathbf{W}$ to reduce the search space was independently used, but the search was, conversely, restricted to the connection topology. This approach also was demonstrated to yield on average 50% smaller (and much more stable) error in predicting the behavior of a mass–spring–damper system with small ($N_{\mathbf{x}} = 20$) reservoirs than without the genetic optimization.

Yet another way of reducing the search space of the reservoir parameters is constructing a big reservoir weight matrix $\mathbf{W}$ in a fractal fashion by repeatedly applying *Kronecker* self-multiplication to an initial small matrix, called the *Kronecker kernel* [123]. This contribution showed that among $\mathbf{W}$s constructed in this way some yield ESN performance similar to the best unconstrained $\mathbf{W}$s; thus only the good weights of the small Kronecker kernel need to be found by evolutionary search for producing a well-performing reservoir.

Evolino [46], introduced in Section 3.3, is another example of adapting a reservoir (in this case an LSTM network) using a genetic search.

It has been recently demonstrated that by adapting only the slopes of the reservoir unit activation functions $f(\cdot)$ by a state-of-art evolutionary algorithm, and having $\mathbf{W}_{\text{out}}$ random and fixed, a prediction performance of an ESN can be achieved close to the best of classical ESNs [68].

In addition to (or instead of) adapting the reservoirs, an evolutionary search can also be applied in training the readouts, such as readouts with no explicit $\mathbf{y}_{\text{target}}(n)$, as discussed in Section 8.4.

7.3. Other types of supervised reservoir tuning

A greedy pruning of neurons from a big reservoir has been shown in a recent initial attempt [124] to often give a (bit) better classification performance for the same final $N_{\mathbf{x}}$ than just a randomly created reservoir of the same size. The effect of neuron removal to the reservoir dynamics, however, has not been addressed yet.

7.4. Trained auxiliary feedbacks

While reservoirs have a natural capability of performing complex real-time analog computations with fading memory [11], an analytical investigation has shown that they can approximate any k-order differential equation (with persistent memory) if extended with k trained feedbacks [21,125]. This is equivalent to simulating any Turing machine, and thus also means universal digital computing. In the presence of noise (or finite precision) the memory becomes limited in such models, but they still can simulate Turing machines with finite tapes.

This theory has direct implications for reservoir computing; thus different ideas on how the power of ESNs could be improved along its lines are explored in [78]. It is done by defining auxiliary targets, training additional outputs of ESNs on these targets, and feeding the outputs back to the reservoir. Note that this can be implemented in the usual model with feedback connections (6) by extending the original output $\mathbf{y}(n)$ with additional dimensions that are trained before training the original (final) output. The auxiliary targets are constructed from $\mathbf{y}_{\text{target}}(n)$ and/or $\mathbf{u}(n)$ or some additional knowledge of the modeled process. The intuition is that the feedbacks could shift the internal dynamics of $\mathbf{x}(n)$ in the directions that would make them better linearly combinable into $\mathbf{y}_{\text{target}}(n)$. The investigation showed that for some types of tasks there are natural candidates for such auxiliary targets, which improve the performance significantly. Unfortunately, no universally applicable methods for producing auxiliary targets are known such that the targets would be both easy to learn and improve the accuracy of the final output $\mathbf{y}(n)$. In addition, training multiple outputs with feedback connections $\mathbf{W}_{\text{ofb}}$ makes the whole procedure more complicated, as cyclical dependences between the trained outputs (one must take care of the order in which the outputs are trained) as well as stability issues discussed in Section 8.2 arise. Despite these obstacles, we perceive this line of research as having a big potential.

7.5. Reinforcement learning

In the line of biologically inspired local unsupervised adaptation methods discussed in Section 6.2, an STDP modulated by a reinforcement signal has recently emerged as a powerful learning mechanism, capable of explaining some famous findings in neuroscience (biofeedback in monkeys), as demonstrated in [126,127] and references thereof. The learning mechanism is also well biologically motivated as it uses a local unsupervised STDP rule and a reinforcement (i.e., reward) feedback, which is present in biological brains in a form of chemical signaling, e.g., by the level of dopamine. In the RC framework this learning rule has been successfully applied for training readouts from the reservoirs so far in [127], but could in principle be applied inside the reservoir too.

Overall the authors of this review believe that reinforcement learning methods are natural candidates for reservoir adaptation, as they can immediately exploit the knowledge of how well the output is learned inside the reservoir without the problems of error backpropagation. They can also be used in settings where no explicit target $\mathbf{y}_{\text{target}}(n)$ is available. We expect to see more applications of reinforcement learning in reservoir computing in the future.

8. Readouts from the reservoirs

Conceptually, training a readout from a reservoir is a common supervised non-temporal task of mapping $\mathbf{x}(n)$ to $\mathbf{y}_{\text{target}}(n)$. This is a well investigated domain in machine learning, much more so than learning temporal mappings with memory. A large choice of methods is available, and in principle any of them can be applied. Thus we will only briefly go through the ones reported to be successful in the literature.

8.1. Single-layer readout

By far the most popular readout method from the ESN reservoirs is the originally proposed [12] simple linear readout, as in (3) (we will consider it as equivalent to (8), i.e., $\mathbf{u}(n)$ being part of $\mathbf{x}(n)$). It is shown to be often sufficient, as reservoirs provide a rich enough pool of signals for solving many application-relevant and benchmark tasks, and is very efficient to train, since optimal solutions can be found analytically.

8.1.1. Linear regression

In batch mode, learning of the output weights $\mathbf{W}_{\text{out}}$ (2) can be phrased as solving a system of linear equations

$$\mathbf{W}_{\text{out}}\mathbf{X} = \mathbf{Y}_{\text{target}} \tag{12}$$

with respect to $\mathbf{W}_{\text{out}}$, where $\mathbf{X} \in \mathbb{R}^{N\times T}$ are all $\mathbf{x}(n)$ produced by presenting the reservoir with $\mathbf{u}(n)$, and $\mathbf{Y}_{\text{target}} \in \mathbb{R}^{N_y\times T}$ are all $\mathbf{y}_{\text{target}}(n)$, both collected into respective matrices over the training period $n = 1,\ldots,T$. Usually $\mathbf{x}(n)$ data from the beginning of the training run are discarded (they come before $n = 1$), since they are contaminated by initial transients.

Since typically the goal is minimizing a quadratic error $E(\mathbf{Y}_{\text{target}}, \mathbf{W}_{\text{out}}\mathbf{X})$ as in (1) and $T > N$, to solve (12) one usually employs methods for finding *least square* solutions of *overdetermined* systems of linear equations (e.g., [128]), the problem also known as *linear regression*. One direct method is calculating the Moore–Penrose pseudoinverse $\mathbf{X}^+$ of $\mathbf{X}$, and $\mathbf{W}_{\text{out}}$ as

$$\mathbf{W}_{\text{out}} = \mathbf{Y}_{\text{target}}\mathbf{X}^+. \tag{13}$$

Direct pseudoinverse calculations exhibit high numerical stability, but are expensive memory-wise for large state-collecting matrices $\mathbf{X} \in \mathbb{R}^{N\times T}$, thereby limiting the size of the reservoir N and/or the number of training samples T.

This issue is resolved in the *normal equations* formulation of the problem[5]:

$$\mathbf{W}_{\text{out}}\mathbf{X}\mathbf{X}^{\mathrm{T}} = \mathbf{Y}_{\text{target}}\mathbf{X}^{\mathrm{T}}. \tag{14}$$

A naive solution of it would be

$$\mathbf{W}_{\text{out}} = \mathbf{Y}_{\text{target}}\mathbf{X}^{\mathrm{T}}(\mathbf{X}\mathbf{X}^{\mathrm{T}})^{-1}. \tag{15}$$

Note that in this case $\mathbf{Y}_{\text{target}}\mathbf{X}^{\mathrm{T}} \in \mathbb{R}^{N_y\times N}$ and $\mathbf{X}\mathbf{X}^{\mathrm{T}} \in \mathbb{R}^{N\times N}$ do not depend on the length T of the training sequence, and can be calculated incrementally while the training data are passed through the reservoir. Thus, having these two

[5] Note that our matrices are transposed compared to the conventional notation.