Neurocomputing 611 (2025) 128679

Contents lists available at ScienceDirect

Neurocomputing

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

Comparison of Reservoir Computing topologies using the Recurrent Kernel approach

Giuseppe Alessio D'Inverno [a], Jonathan Dong [b,*]

[a] Department of Information Engineering and Mathematics, University of Siena, Siena, 53100, Italy
[b] Biomedical Imaging Group, École Polytechnique Fédérale de Lausanne, Station 17, Lausanne, 1015, Switzerland

ARTICLE INFO

Communicated by A. Iosifidis

Keywords:
Reservoir Computing
Recurrent Kernels
Sparse Reservoir Computing
Structured reservoir computing
Deep reservoir computing

ABSTRACT

Reservoir Computing (RC) has become popular in recent years thanks to its fast and efficient computational capabilities. Standard RC has been shown to be equivalent in the asymptotic limit to Recurrent Kernels, which helps in analyzing its expressive power. However, many well-established RC paradigms, such as Leaky RC, Sparse RC, and Deep RC, are yet to be systematically analyzed in such a way. We define the Recurrent Kernel limit of all these RC topologies and conduct a convergence study for a wide range of activation functions and hyperparameters. Our findings provide new insights into various aspects of Reservoir Computing. First, we demonstrate that there is an optimal sparsity level which grows with the reservoir size. Furthermore, our analysis suggests that Deep RC should use reservoir layers of decreasing sizes. Finally, we perform a benchmark demonstrating the efficiency of Structured Reservoir Computing compared to vanilla and Sparse Reservoir Computing.

1. Introduction and related work

Reservoir Computing (RC) is a machine learning technique used for training Recurrent Neural Networks, which fixes the internal weights of the network and only trains a linear layer, resulting in faster training times [1]. Its simplicity and effectiveness have made it a popular choice for various tasks such as chaotic time series prediction [1], robot motor control or financial forecasting [2]. Additionally, the random connections within Reservoir Computing networks make them a useful framework for comparison with biological neural networks [3].

Over time, researchers have proposed several methods to optimize and enhance Reservoir Computing's performance and efficiency. One such method is Leaky Reservoir Computing, which stabilizes the dynamics of the reservoir and enables tuning of its memory by adjusting the leak rate [4]. There is also Sparse Reservoir Computing, which consists in a sparse initialization of weight connections proposed since the original formulation of Echo State Networks [1]). Structured Reservoir Computing [5] is another acceleration strategy which replaces the internal weights by a structured transform instead. Finally, Deep Reservoir Computing allows for the use of reservoirs with different time dynamics [6]. All these variants show the flexibility of the Reservoir Computing framework, made of a fixed reservoir encoding a time-dependent input combined with a linear readout, which can be extended further to physical implementations [7–9] and next-generation reservoir computing [10].

Increasing the number of neurons in a Reservoir Computing network leads to the convergence of its behavior to a recurrent kernel, as discussed in [5]. Kernel methods are a class of algorithms in machine learning that use kernel functions to implicitly map input data into high-dimensional feature spaces, calculating scalar products between input points in a dual space, enabling linear models to solve non-linear problems. Recurrent Kernels are a variant in which these scalar products are dynamically updated over time based on changes in the input data. They can be used as an alternative to large-scale Reservoir Computing and have demonstrated state-of-the-art performance on chaotic time series prediction [5]. There are two ways Recurrent Kernels can be useful. They offer an interesting alternative to RC when the number of data points is limited, as kernel methods require the calculation of scalar products between all pairs of input points. Additionally, recurrent kernels have been useful for theoretical studies, such as stability analysis in Reservoir Computing [11], as they provide a deterministic limit with analytical expressions.

Prior studies on Recurrent Kernels has been mainly limited to vanilla Reservoir Computing and structured transforms. In this work, we extend the application of Recurrent Kernels to other Reservoir Computing topologies, such as Leaky, Sparse, and Deep Reservoir Computing. Specifically, we define the appropriate Recurrent Kernels for each topology, investigate similarities in their corresponding limits,

* Corresponding author.
E-mail address: jonathan.dong@epfl.ch (J. Dong).

https://doi.org/10.1016/j.neucom.2024.128679
Received 28 January 2024; Received in revised form 21 September 2024; Accepted 29 September 2024
Available online 1 October 2024

and evaluate their convergence numerically. By broadening the scope of Recurrent Kernels, we aim to demonstrate their versatility and effectiveness in a range of Reservoir Computing configurations.

Our main contributions are listed as follows:

- We define the Recurrent Kernel limit for Leaky RC, Sparse RC, and Deep RC, showing that Sparse RC converges to the same limit as vanilla RC and Structured RC
- We conduct a thorough numerical study on the convergence of these RC paradigms to their Recurrent Kernel counterparts, for different activation functions
- Our results show that sparse RC is equivalent to the non-sparse case, as long as the sparsity rate is above a certain threshold. This suggests that sparse RC does not have increased or decreased expressivity compared to vanilla RC
- We show that, in Deep Reservoir Computing, first reservoirs should be larger than subsequent ones, in order to decrease the amount of noise transmitted in the subsequent layers. However, this effect is quite small for large reservoirs and reservoirs with equal sizes should perform similarly in practice.
- We perform a benchmark of Reservoir Computing, Sparse Reservoir Computing, and Structured Reservoir Computing, as both strategies have been introduced for computational efficiency, and demonstrate that Structured RC is generally the most efficient for large reservoir sizes.

2. Background

2.1. Reservoir computing

Reservoir Computing, like all recurrent neural network architectures, receives sequential input $\mathbf{i}^{(t)} \in \mathbb{R}^d$ for $t \in \mathbb{N}$. The simplest model for Reservoir Computing, commonly called the Echo-State Network (ESN) [12], comprises a set of neurons $\mathbf{x}^{(t)} \in \mathbb{R}^N$ with fixed random weights, with N the number of neurons in the reservoir. The initial state of the network $\mathbf{x}^{(0)}$ is randomly initialized, typically from a random Gaussian distribution. The network is then updated according to the following equation:

$$\mathbf{x}^{(t+1)} = \frac{1}{\sqrt{N}} f\left(\sigma_r \mathbf{W}_r \mathbf{x}^{(t)} + \sigma_i \mathbf{W}_i \mathbf{i}^{(t)}\right). \tag{1}$$

Here, $\mathbf{W}_r \in \mathbb{R}^{N\times N}$ and $\mathbf{W}_i \in \mathbb{R}^{N\times d}$ are the reservoir and input weight matrices, σ_r and σ_i are reservoir and input scaling factors, and f is an element-wise nonlinearity, often a sigmoid—which is well approximated by the (Gauss) error function. The factor $1/\sqrt{N}$ ensures proper normalization of the L2-norm of $\mathbf{x}$ when N goes to infinity. Each weight of $\mathbf{W}_r$ and $\mathbf{W}_i$ is drawn from a normal Gaussian distribution with unit variance:

$$p(w) = \frac{1}{\sqrt{2\pi}} e^{-w^2/2} \tag{2}$$

An essential hyperparameter to tune is the scaling factor σ_r. It significantly impacts the dynamics and stability of the reservoir: when σ_r is small, the updates in Eq. (1) are contractant, while the reservoir becomes a chaotic nonlinear system for large σ_r. Therefore, this hyperparameter is often optimized to maximize performance for a given task.

The output of Reservoir Computing $\mathbf{o}^{(t)} \in \mathbb{R}^n$ is computed using a linear model applied to the state of the reservoir, as given by:

$$\mathbf{o}^{(t)} = \mathbf{W}_{\text{out}} \mathbf{x}^{(t)}. \tag{3}$$

The training step for this model involves a linear regression, which is a stark contrast to the non-linear optimization typically employed when training neural networks. Reservoir Computing's approach is based on the idea that the current state of the reservoir, $\mathbf{x}^{(t)}$, non-linearly encodes the past values of the input time series, $\mathbf{i}^{(t-1)}, \mathbf{i}^{(t-2)}$, etc. .

2.2. Different variants of reservoir computing

Several Reservoir Computing variants have been proposed, which modify the update equations and alter the reservoir dynamics (see Fig. 1). These variants include adjustments to the updates to tune the reservoir relaxation time, speeding up computation, or introducing a hierarchical structure to enrich the dynamics. The flexibility of Reservoir Computing makes it possible to fine-tune the dynamics precisely for a particular task using these variants.

Sparse Reservoir Computing aims to increase computational efficiency by using sparse internal weight matrices. The computational complexity in Reservoir Computing is mainly determined by the matrix multiplication involving the $N \times N$ internal weight matrix. In Sparse Reservoir Computing, this matrix is made sparse by drawing the weights from a sparse i.i.d. distribution. Specifically, the distribution is given by:

$$p(w_r) = (1-s)\delta(w_r) + s\sqrt{\frac{s}{2\pi}} e^{-\frac{s w_r^2}{2}}, \tag{4}$$

where δ denotes the Dirac delta function, and s takes values between 0 and 1, controlling the proportion of non-zero weights. $s = 1$ corresponds to the original non-sparse case, and it is typically set at 0.05 [13,14] which means that 5% of the weights are non-zero. The variance of the Gaussian term is fixed at $1/s$ to ensure that the spectral radius of the matrix stays similar to the non-sparse case. We focus here on a sparsity model in which a fraction of the weights are non-zero. Other works define sparsity with a fixed number of connections per neurons [15], the two approaches being equivalent at fixed reservoir sizes.

The sparsity in the weight matrix reduces the computational complexity of the update equation, enabling faster computation without sacrificing performance. The computational and memory complexities are $\mathcal{O}(sN^2)$. This approach is especially useful for large-scale Reservoir Computing systems where the computational cost is significant. Research has shown that the use of sparse matrix multiplication can increase computational speed, while maintaining accuracy. Thanks to their simplicity, they are often used in existing Reservoir Computing works.

Structured RC speeds up computations by replacing the dense weight matrices by a product of fixed structured and random diagonal matrices [5], inspired by Orthogonal Random Features [16] which are their non-recurrent counterparts. The weight matrix W is replaced by:

$$W = HD_1HD_2HD_3,$$

where H denotes a fixed structured transform and D_i are diagonal matrices with i.i.d. Rademacher random variables (± 1 with probability 0.5) on the diagonals. The computational complexity is $\mathcal{O}(n \log n)$, determined by the structured transforms, while the memory complexity is $\mathcal{O}(n)$. Deviating from [5], we will consider Hartley matrices instead of Hadamard matrices. The Hartley transform is the real-valued version of the Fourier transform, defined as $H(x) = \text{Re}\, F(x * (1 + j))$. It maps real-valued vectors to real-valued outputs and can be computed efficiently on CPU and GPU using the Fast Fourier Transform which is present in all generic numerical libraries in Python. By contrast, the Hadamard transform is a simple structured transform but requires dedicated Python libraries.

Leaky-Reservoir Computing introduces a leak rate to control the typical time scale of changes in the reservoir. The update equation for Leaky-Reservoir Computing is given by:

$$\mathbf{x}^{(t+1)} = (1-a)\mathbf{x}^{(t)} + a\frac{1}{\sqrt{N}} f\left(\sigma_r \mathbf{W}_r \mathbf{x}^{(t)} + \sigma_i \mathbf{W}_i \mathbf{i}^{(t)}\right), \tag{5}$$

where $a \in [0, 1]$ is the leak rate. Setting $a = 1$ corresponds to the non-leaky Reservoir Computing case described earlier. Decreasing a slows down the speed of changes in the reservoir, thereby controlling the typical time scale of reservoir dynamics. This feature can be useful for

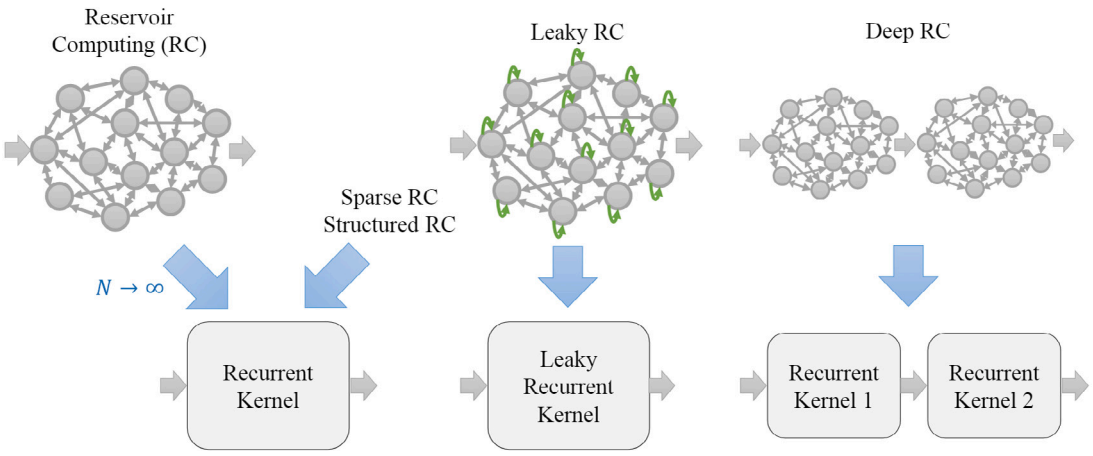

Fig. 1. Recurrent Kernels associated with various Reservoir Computing topologies. RC, sparse RC, and structured RC converge to the same RK limit when the reservoir size $N \to \infty$. Leaky RC and Deep RC converge to their corresponding limits.

tasks in which the input signal changes slowly over time, as it enables the reservoir to better capture the temporal dependencies in the input data.

Deep Reservoir Computing stacks multiple reservoir layers to form a deep architecture also called a Deep Echo State Network (deepESN) [6]. The first layer operates like the reservoir in a shallow Reservoir Computing architecture and is fed by the external input, while each successive layer is fed by the output of the previous one. The reservoir layer of a deepESN can be expressed as:

$$\mathbf{x}_l^{(t+1)} = \frac{1}{\sqrt{N_l}} f\left(\sigma_r \mathbf{W}_r^l \mathbf{x}_l^{(t)} + \sigma_i \mathbf{W}_i^l \mathbf{u}_l^{(t)}\right) \tag{6}$$

where the index $l = 1, \dots, L$ describes the layer with a reservoir of size N_l, $\mathbf{u}_l^{(t)}$ is the input for the lth layer:

$$\mathbf{u}_l^{(t)} = \begin{cases} \mathbf{i}^{(t)} \text{ if } l = 1 \\ \mathbf{x}_{l-1}^{(t+1)} \text{ if } l > 1. \end{cases} \tag{7}$$

One of the main ideas is that each reservoir is encoding the recent past of its received input. Thus, the first layer has limited memory while the subsequent ones are able to extend this memory and build more complex representations of the input signal.

The output of a deepESN at each time step t can be computed by applying any linear model to the different reservoir states. A common choice is to define the linear model on the concatenation of all reservoir states; the output $\mathbf{o}^{(t)}$ is given by:

$$\mathbf{o}^{(t)} = \mathbf{W}_{\text{out}} \begin{bmatrix} \mathbf{x}_1^{(t)} \\ \mathbf{x}_2^{(t)} \\ \vdots \\ \mathbf{x}_L^{(t)} \end{bmatrix} \tag{8}$$

where $\mathbf{W}_{\text{out}} \in \mathbb{R}^{n \times \sum_l N_l}$ is a weight matrix that maps the concatenated reservoir states to the output. The concatenation of the reservoir states from each layer allows for the capture of information across multiple time scales, enabling the deepESN to model more complex temporal patterns.

The different variants presented above are not exclusive. For example, it is common to introduce different leak rates and sparse internal weight matrices to each layer of a deepESN.

Other strategies have also been proposed to alleviate the dense matrix multiplication by the reservoir weights. Next-generation Reservoir Computing [10] replaces the non-linear recurrent reservoir by an explicit mapping with polynomial combination of past inputs. As such, it can be interpreted as a non-recurrent temporal kernel. The flexibility of Reservoir Computing is further exemplified by the many physical implementations of Reservoir Computing [7–9,17], showing that any non-linear dynamical system can be used as a reservoir.

2.3. Recurrent kernels

In machine learning, kernels are functions that measure the similarity between pairs of data points, in a high-dimensional space to enable effective linear models. This mapping into the higher-dimensional feature space is often done implicitly by computing the scalar products between data points. This observation can be extended to Reservoir Computing leading to Recurrent Kernels. We consider two reservoirs $\mathbf{x}$ and $\mathbf{y}$ driven by the inputs $\mathbf{i}$ and $\mathbf{j}$ respectively, following the update Eq. (1). For conciseness, we assume $\sigma_r = \sigma_i = 1$; equations for different values of reservoir and input scales can be obtained by substituting $\mathbf{x}^{(t)}$ by $\sigma_r \mathbf{x}^{(t)}$ and $\mathbf{i}^{(t)}$ by $\sigma_i \mathbf{i}^{(t)}$. The scalar product between two reservoir states can be expressed as:

$$\left(\mathbf{x}^{(t+1)}\right)^\top \mathbf{y}^{(t+1)} = \frac{1}{N} \sum_{l=1}^{N} f\left(\mathbf{w}_{r,l}^\top \mathbf{x}^{(t)} + \mathbf{w}_{i,l}^\top \mathbf{i}^{(t)}\right) \times f\left(\mathbf{w}_{r,l}^\top \mathbf{y}^{(t)} + \mathbf{w}_{i,l}^\top \mathbf{j}^{(t)}\right) \tag{9}$$

where $\mathbf{w}_{r,l}$ and $\mathbf{w}_{i,l}$ denote the lth line of $\mathbf{W}_r$ and $\mathbf{W}_i$ respectively. Thanks to the law of large numbers, this quantity converges when the reservoir size N goes to infinity to a deterministic kernel function $k_0 : \left(\mathbb{R}^{N+d}\right)^2 \to \mathbb{R}$ defined as:

$$k_0\left(\mathbf{u}^{(t)}, \mathbf{v}^{(t)}\right) = \int d\mathbf{w} p(\mathbf{w}) f\left(\mathbf{w}^\top \mathbf{u}^{(t)}\right) f\left(\mathbf{w}^\top \mathbf{v}^{(t)}\right) \tag{10}$$

where we have introduced $\mathbf{u}^{(t)} = \begin{bmatrix} \mathbf{x}^{(t)} \\ \mathbf{i}^{(t)} \end{bmatrix}$, $\mathbf{v}^{(t)} = \begin{bmatrix} \mathbf{y}^{(t)} \\ \mathbf{j}^{(t)} \end{bmatrix}$, and $\mathbf{w} = \begin{bmatrix} \mathbf{w}_r \\ \mathbf{w}_i \end{bmatrix}$ a random vector of dimension $N + d$ with i.i.d. normal entries.

To properly define the associated Recurrent Kernel, we need to remove the dependency on the previous reservoir states $\mathbf{x}^{(t)}$ and $\mathbf{y}^{(t)}$, as they themselves depend on the random weights $\mathbf{W}_r$ and $\mathbf{W}_i$. This is possible if k_0 is an *iterable kernel*, i.e. if there exists $k : \mathbb{R}^3 \to \mathbb{R}$ such that for all $\mathbf{u}, \mathbf{v} \in \mathbb{R}^{N+d}$:

$$k_0(\mathbf{u}, \mathbf{v}) = k(\|\mathbf{u}\|^2, \|\mathbf{v}\|^2, \mathbf{u}^\top \mathbf{v}). \tag{11}$$

We show in Appendix that the kernel associated to Reservoir Computing is always iterable, as soon as the weights are sampled from a Gaussian distribution (or any rotationally-invariant distribution $p(\mathbf{w})$). This is an extension of the statement in [5] that assumed translation or rotation-invariant kernels.

We define the recurrent kernel by replacing $\left(\mathbf{x}^{(t+1)}\right)^\top \mathbf{y}^{(t+1)}$ in Eq. (9) by $k^{(t+1)}\left(\mathbf{i}^{(t)}, \mathbf{j}^{(t)}, \dots\right)$. Similarly, we replace in Eq. (11) $\left(\mathbf{u}^{(t)}\right)^\top \mathbf{v}^{(t)}$ by

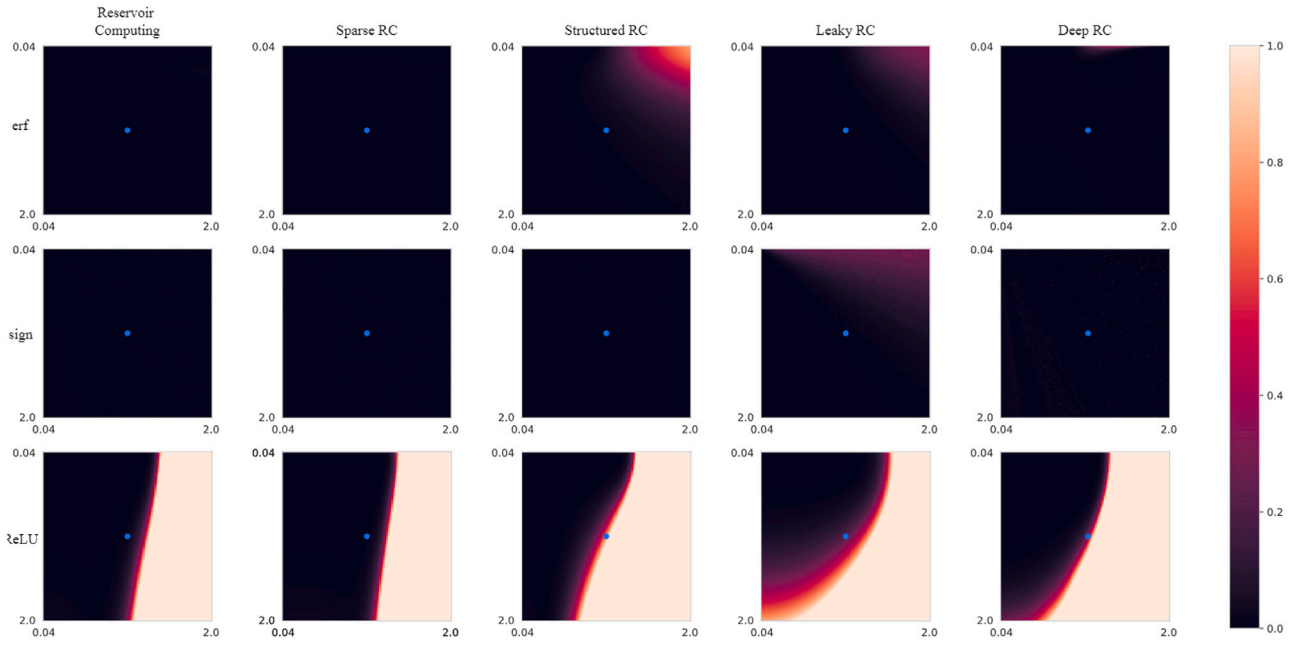

Fig. 2. Convergence study of various Reservoir Computing topologies (columns) towards their corresponding Recurrent Kernel limits, for different activation functions f (rows). For each case, the Frobenius norm between the RC and RK Gram matrices L (Eq. (18), smaller is better) is displayed for weight scaling factors σ_r, σ_i between 0.04 and 2. Blue dot in the first row: typical operating point $\sigma_r = \sigma_i = 1$.

$k^{(t)}(\mathbf{i}^{(t-1)}, \mathbf{j}^{(t-1)}, \ldots) + \left(\mathbf{i}^{(t)}\right)^\top \mathbf{j}^{(t)}$, and perform the same operation for the norms (as norms are symmetric scalar products). This leads to the following definition of a Recurrent Kernel (RK) as a sequence of kernel functions $k^{(t)} : \left(\mathbb{R}^d\right)^{2t} \to \mathbb{R}$ for $t \in \mathbb{N}^*$:

$$\begin{cases} k^{(1)}\left(\mathbf{i}^{(0)}, \mathbf{j}^{(0)}\right) & = k(1 + \|\mathbf{i}^{(0)}\|^2, 1 + \|\mathbf{j}^{(0)}\|^2, \left(\mathbf{i}^{(t)}\right)^\top \mathbf{j}^{(t)}) \\ k^{(t+1)}\left(\mathbf{i}^{(t)}, \mathbf{j}^{(t)}, \ldots\right) & = k(k^{(t)}(\mathbf{i}^{(t-1)}, \mathbf{i}^{(t-1)}, \ldots) + \|\mathbf{i}^{(t)}\|^2, \\ & \quad k^{(t)}(\mathbf{j}^{(t-1)}, \mathbf{j}^{(t-1)}, \ldots) + \|\mathbf{j}^{(t)}\|^2, \\ & \quad k^{(t)}(\mathbf{i}^{(t-1)}, \mathbf{j}^{(t-1)}, \ldots) + \left(\mathbf{i}^{(t)}\right)^\top \mathbf{j}^{(t)}) \end{cases} \tag{12}$$

In the first line, we have initialized the RK by choosing $\|\mathbf{x}^{(0)}\| = \|\mathbf{y}^{(0)}\| = 1$ and $\left(\mathbf{x}^{(0)}\right)^\top \mathbf{y}^{(0)} = 0$.

One can then replace large-scale Reservoir Computing by Recurrent Kernels. To accomplish this, one must compute the recurrent kernels *for each pair of training inputs*, which are then placed into a matrix known as the Gram matrix. For instance, let us denote by $\mathbf{i}_m$, $m = 1, \ldots, M$, the different inputs. The Gram matrix $\mathbf{G}^{(t)} \in \mathbb{R}^{M \times M}$ is defined for $t \in \mathbb{N}^*$ as:

$$\mathbf{G}^{(t)} = \left[k^{(t)}(\mathbf{i}_n^{(t-1)}, \mathbf{i}_m^{(t-1)}, \ldots)\right]_{n,m} \tag{13}$$

A linear model is trained using the Gram matrix and can be employed for making predictions.

Recurrent Kernels hold promise as a substitute for large-scale Reservoir Computing, as they offer comparable performance as the Reservoir Computing limit with infinitely-many neurons. However, the drawback is the computational time needed for prediction, as scalar products must be computed with each training input for use in the linear model. Additionally, due to their analytic formulation, Recurrent Kernels are well-suited for theoretical studies on Reservoir Computing [11].

Rigorously proving the convergence of Reservoir Computing to the Recurrent Kernel has proven challenging. Three assumptions are typically required [5]:

1. Lipschitz-continuity: the activation function is l-Lipschitz;
2. Contractivity: the scaling factor of the reservoir weights needs to satisfy $\sigma_r^2 \leq 1/l$
3. Time-independence: the weight matrix is resampled at each time step.

These assumptions are very restrictive to prove convergence of RC towards their Recurrent Kernel limits in practice. We instead resort to numerical investigations of convergence, as presented in the next section.

3. Recurrent kernel limits for various RC topologies

Here we define the Recurrent Kernel limits of the different Reservoir Computing topologies and discuss the assumptions for convergence to these Recurrent Kernels. More details are provided in the Appendix.

The Recurrent Kernel for sparse Reservoir Computing corresponds to the same Recurrent Kernel as the non-sparse case. This implies that the asymptotic performance of a sparse reservoir is equivalent to the one of a nonsparse one. To obtain this result, the reservoir activations at each iteration needs not to be sparse, which is generally valid for Reservoir Computing. A more detailed study of the sparse case is provided in Appendix. This is similar to Structured Reservoir Computing: both strategies have the same RK limit and they have been introduce to decrease the cost of RC computations. As such, all three topologies (vanilla, sparse, and structured RC) are equivalent asymptotically and could be used interchangeably.

The Recurrent Kernel corresponding to Reservoir Computing with leak rate is defined by replacing the update equation of Eq. (12)

$$\begin{aligned} k^{(t+1)}(\mathbf{i}^{(t)}, \mathbf{j}^{(t)}, \ldots) = (1-a)^2 k^{(t)}\left(\mathbf{i}^{(t-1)}, \mathbf{j}^{(t-1)}, \ldots\right) \\ + a^2 k(k^{(t)}(\mathbf{i}^{(t-1)}, \mathbf{i}^{(t-1)}, \ldots) + \|\mathbf{i}^{(t)}\|^2, \\ k^{(t)}(\mathbf{j}^{(t-1)}, \mathbf{j}^{(t-1)}, \ldots) + \|\mathbf{j}^{(t)}\|^2, \\ k^{(t)}(\mathbf{i}^{(t-1)}, \mathbf{j}^{(t-1)}, \ldots) + \left(\mathbf{i}^{(t)}\right)^\top \mathbf{j}^{(t)}) \end{aligned} \tag{14}$$

More details are provided in Appendix. As described for the vanilla case of Recurrent Kernels, this limit is valid beyond these assumptions and we will study it numerically.

For Deep Reservoir Computing, we can write the kernel limit for each layer. We start by the first layer:

$$\begin{aligned} k_1^{(t+1)}\left(\mathbf{i}^{(t)}, \mathbf{j}^{(t)}, \ldots\right) = k(k_1^{(t)}(\mathbf{i}^{(t-1)}, \mathbf{i}^{(t-1)}, \ldots) + \|\mathbf{i}^{(t)}\|^2, \\ k_1^{(t)}(\mathbf{j}^{(t-1)}, \mathbf{j}^{(t-1)}, \ldots) + \|\mathbf{j}^{(t)}\|^2, \\ k_1^{(t)}(\mathbf{i}^{(t-1)}, \mathbf{j}^{(t-1)}, \ldots) + \left(\mathbf{i}^{(t)}\right)^\top \mathbf{j}^{(t)}) \end{aligned} \tag{15}$$

For the subsequent layers $l > 1$, we obtain the recursive formula:

$$k_l^{(t+1)}\left(\mathbf{i}^{(t)}, \mathbf{j}^{(t)}, \ldots\right) = k(k_l^{(t)}(\mathbf{i}^{(t-1)}, \mathbf{i}^{(t-1)}, \ldots) + k_{l-1}^{(t+1)}(\mathbf{i}^{(t)}, \mathbf{i}^{(t)}, \ldots), \\ k_l^{(t)}(\mathbf{j}^{(t-1)}, \mathbf{j}^{(t-1)}, \ldots) + k_{l-1}^{(t+1)}(\mathbf{j}^{(t)}, \mathbf{j}^{(t)}, \ldots), \\ k_l^{(t)}(\mathbf{i}^{(t-1)}, \mathbf{j}^{(t-1)}, \ldots) + k_{l-1}^{(t+1)}(\mathbf{i}^{(t)}, \mathbf{j}^{(t)}, \ldots)) \quad (16)$$

In practice, we can compute these Recurrent Gram matrices layer by layer.

The Recurrent Kernel prediction is performed by concatenating the reservoir states of the different layers and using a linear model. This corresponds to a sum of the different Gram matrices:

$$k_{\text{tot}}^{(t+1)}(\mathbf{i}^{(t)}, \mathbf{j}^{(t)}, \ldots) = \sum_l k_l^{(t+1)}(\mathbf{i}^{(t)}, \mathbf{j}^{(t)}, \ldots) \quad (17)$$

4. Results

4.1. Convergence study

We show in Fig. 2 a numerical convergence study of the various Reservoir Computing topologies to their respective Recurrent Kernel limits. Two random inputs of length $T = 10$ and dimension $d = 100$ are generated and fed to reservoirs of size $N = 1000$. When applicable, the sparsity level is set at $s = 0.5$ and the leak rate at $a = 0.5$. For Deep Reservoir Computing, we use a sequence of two reservoirs of size $N_1 = N_2 = 1000$.

We compute the final Gram matrix G^{RC} at time $T + 1 = 11$. This Gram matrix is compared with the Gram matrix G^{RK} obtained using the associated Recurrent Kernel. The metric displayed is the Frobenius norm between the final Gram matrices:

$$L = \left\| G_{\text{RC}}^{(T+1)} - G_{\text{RK}}^{(T+1)} \right\|_F^2 . \quad (18)$$

This metric is computed for different values of the reservoir weight standard deviations σ_r and σ_i which dictate the dynamics of the reservoir between 0 and 2. When σ_r is small, dynamics are contractant, while they become chaotic for large values of σ_r. Theory only predicts convergence for the contractant case but it has also been observed for large σ_r. We typically choose σ_r close to 1 to obtain non-chaotic but rich reservoir dynamics. We perform this study for three typical activation functions: erf (differentiable and bounded), sign (discontinuous and bounded), and ReLU (sub-differentiable and unbounded). Among these three, the error function would be the one generally used in practice.

We see that in the sparse case, the convergence is fundamentally similar to non-sparse Reservoir Computing. We observe convergence over the whole parameter range for bounded activation functions. For ReLU, convergence is achieved for values of σ_r below a threshold which depends slightly on σ_i. This is due to errors accumulating with the ReLU activation function, for which convergence is more difficult to achieve compared to bounded activation functions. Convergence is also achieved over a wide range of parameters for Structured RC; the convergence region is slightly smaller for erf and ReLU, but most importantly Structured RC converges to the RK limit for the typical operating point in blue with the erf activation function.

For leaky Reservoir Computing and Deep Reservoir Computing, the convergence region is also smaller. For bounded activations, it typically does not converge for large σ_r and small σ_i. This typically corresponds to an unstable case [11] that is not used in practice. For ReLU, convergence is achieved for a range of parameters (σ_r, σ_i) slightly smaller than the non-leaky case. When the reservoir or input weights are large, error accumulates and activations diverge.

In general, convergence is achieved for a wide range of parameters with bounded activation functions. Caution is necessary only when σ_r is large and σ_i is small. For the ReLU case, convergence is more challenging, as activations may diverge.

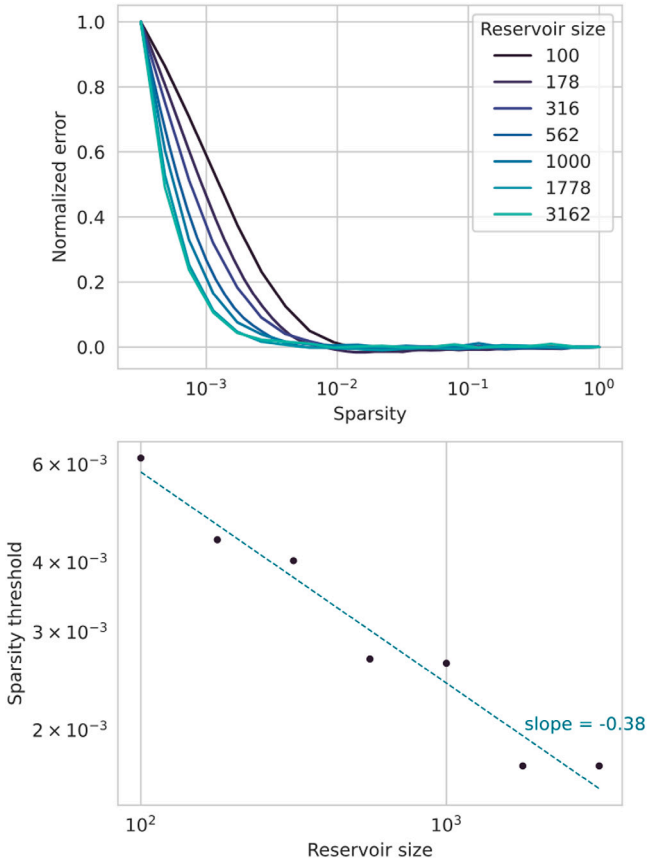

Fig. 3. (Top) Error metric (Eq. (18)) normalized between 0 and 1 as a function of sparsity for different reservoir sizes. (Bottom) Sparsity threshold above which the error metric is within 5% of the non-sparse limit. This gives an admissible sparsity level which decreases with the reservoir size.

4.2. How to choose sparsity level

Our framework enables us to determine the optimal sparsity level to obtain the same convergence to the RK limit while decreasing the computational cost. We compute the normalized error metric as defined in Eq. (18) for different reservoir sizes in Fig. 3 (top row). The error metric being dependent on the reservoir size, we normalize it between 0 and 1 to represent all curves on the same graph. The error metric being dependent on the stochastic realization of the weights, we perform a Monte-Carlo estimation with at least 10^4 realizations to decrease the estimation variance.

We observe that one can decrease the sparsity level until a threshold below which the approximation error increases. This threshold decreases with the reservoir size: larger reservoirs handle low sparsity levels better.

In the bottom row of Fig. 3, we plot the threshold defined as 5% of the non-sparse limit. We see that we can decrease the sparsity factor quite dramatically: for a reservoir size $N = 1000$, we can decrease the sparsity rate down to $s = 0.003$. This value is significantly below the typical sparsity level of 0.05 [13,14], which could lead to further computational and memory savings.

This sparsity level corresponds to a mean of $2sN = 6$ connections per neurons. The optimal sparsity level does not correspond to a constant number of connections per neurons as it is not inversely proportional to the reservoir size, but follows a law in $N^{-0.38}$ (see Fig. 3). At large reservoir sizes, more connections per neurons are required. We suspect that as large reservoirs are better approximations of the RK limit, it becomes more challenging to reduce s while keeping the same approximation quality.

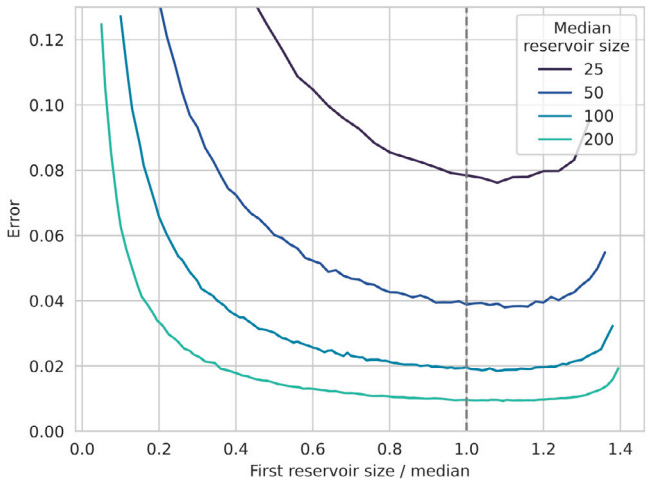

Fig. 4. Error metric (Eq. (18)) for Deep Reservoir Computing for varying reservoir sizes. We vary the first layer size N_1 for a fixed computational budget $N_1^2+N_2^2 = 2\times N_{\text{med}}^2$ for different values of the median reservoir size N_{med}.

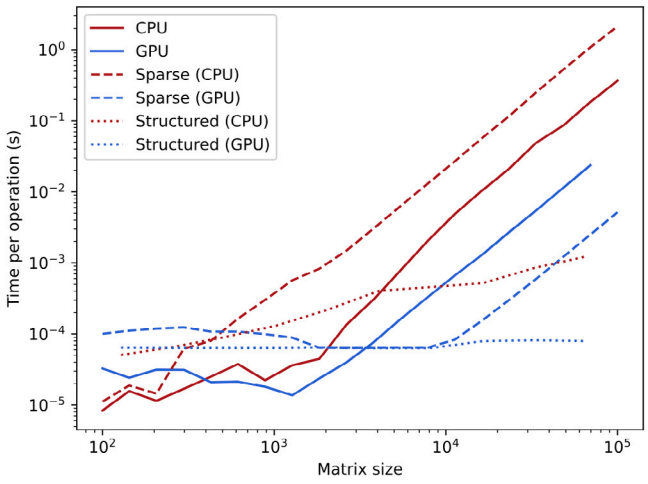

Fig. 5. Time benchmark of the matrix–vector multiplication on CPU and GPU for Reservoir Computing, Sparse Reservoir Computing, and Dense Reservoir Computing.

4.3. Optimal deep reservoir computing sizes

Our framework enables us to investigate how to optimally set the various reservoir weights in Deep Reservoir Computing, addressing the previously unresolved question of whether the first or second reservoir should be larger. To investigate this question, we compute the previous metric given in Eq. (18) for a fixed computational budget $N_1^2 + N_2^2 = 2\times N_{\text{med}}^2$ for different values of N_{med}. This quadratic scaling corresponds to the computational and memory complexity of the dense matrix multiplication, which is the limiting factor in Eq. (1). Each point is an average of 10'000 repetitions.

The metric as a function of N_1 is depicted in Fig. 4. We see that in general the extreme cases yield high error. When the first reservoir size is too small, the error of the first RK is detrimental even though the second reservoir is closer to its limit. Similarly, the second reservoir size cannot be too small or the second recurrent kernel is not well approximated.

Between these extremes, there is a region for which the error is relatively small. For small computational budgets, this region is limited, both reservoirs need to be approximately the same size, while for large reservoirs, the actual reservoir sizes do not seem to matter as much as long as we avoid the extreme cases. We also observe that the minimal error is obtained for a first reservoir size slightly larger than the second.

To obtain a quantitative answer, we performed a Nelder–Mead optimization to find the optimal reservoir sizes. For $L = 2$, we obtain $n_1, n_2 = 209, 190$, for $L = 3$, we obtain $n_1, n_2, n_3 = 207, 202, 190$. Thus, the optimal shapes have first reservoir sizes that are larger than subsequent ones. This decreases the noise that is transmitted to the next layers.

In a nutshell, a good rule of thumb to choose the reservoir sizes in Deep Reservoir Computing is to choose them all equal. First reservoirs can be chosen slightly (around 5%) larger than the last ones to decrease further the distance with the asymptotic limit performance.

4.4. Computational benchmark

Since both RC, Sparse RC, and Structured RC converge to the same kernel limit, we present in Fig. 5 a benchmark to provide guidelines on which topology to choose. We benchmark the matrix–vector multiplication between the square reservoir weight matrix $\mathbf{W}_r$ and the reservoir state $\mathbf{x}$ since this is the bottleneck operation. This benchmark is performed on CPU (Intel Xeon 6240) and GPU (NVIDIA V100). The library used is crucial as low-level optimizations may impact performance. We use the Pytorch library for dense and sparse matrix multiplications and Fast Fourier Transforms in the Structured RC case. Sparsity is fixed at $s = 0.01$.

We first confirm that GPU acceleration is useful for large reservoir sizes, with the threshold being at a few hundred neurons for all techniques. CPUs, on the other hand, excel at small reservoir sizes.

For CPU-based computations, we find that sparse matrix operations in PyTorch are not particularly efficient. In contrast, vanilla RC is beneficial for small reservoir sizes while structured reservoir computing demonstrates superior performance as reservoir sizes increase. The results also demonstrate the quadratic computational complexity associated with both dense and sparse matrix multiplications, while the computational scaling of structured transforms is smaller.

In GPU-based computations, sparse matrix operations show improved efficiency compared to their CPU counterparts. However, Structured Reservoir Computing still outperforms both dense and sparse approaches. This indicates that for GPU implementations, structured transforms may offer the best performance, particularly for applications with reservoir sizes exceeding a few thousand nodes.

5. Discussion

In our study, we have derived the Recurrent Kernel limit of different reservoir topologies. We have shown that different topologies can lead to the same asymptotic limit. More specifically, the presence of sparsity does not affect convergence at all, which justifies the sparse initialization of reservoir weights to speed up computation. Convergence has been studied numerically and validated for a wide range of parameters, especially for bounded activation functions. Furthermore, we have derived how Recurrent Kernels extend to Deep Reservoir Computing, and how it sheds new insight on how to set the consecutive reservoir sizes. Finally, our timing benchmark has demonstrated the superior efficiency of Structured Reservoir Computing, particularly for implementations with large reservoir sizes.

The current study focused on theoretical convergence and speed comparison. It may be interesting to provide a benchmark on a particular Reservoir Computing task (to verify the equivalence of different topologies with the corresponding Recurrent Kernel limit) and optimize further for computational cost, e.g. through the use of a dedicated sparse linear algebra library. Other topologies based on random connections may also be explored theoretically such as: random features [18] and extreme learning machines [19], which approximate a kernel in expectation with non-recursive random embeddings; Gaussian processes [20], stochastic processes defined by a mean and a covariance, which can give an accurate estimate of the uncertainty in regression

tasks ; and random vector functional link networks [21], wherein the weights of the network are generated randomly and output weights are computed analytically.

CRediT authorship contribution statement

Giuseppe Alessio D'Inverno: Writing – review & editing, Visualization, Methodology, Investigation, Formal analysis, Conceptualization. **Jonathan Dong:** Writing – review & editing, Visualization, Supervision, Methodology, Investigation, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

We would like to thank Rahul Parhi for insightful discussions and review of the paper. Giuseppe Alessio D'Inverno is partially funded by Indam GNCS group. Jonathan Dong is funded by the Swiss National Science Foundation (SNSF) under Grant PZ00P2_216211.

Appendix A. Iterable kernels for any rotationally-invariant distribution

We prove here that the kernel limit defined in Eq. (10) is iterable as soon as the weight distribution $p(\mathbf{w})$ is rotationally invariant. For any fixed timestep t, we can use this property to perform a change of basis:

$$\begin{cases} \mathbf{u}^{(t)} = u_1 \mathbf{e}_1 \\ \mathbf{v}^{(t)} = v_1 \mathbf{e}_1 + v_2 \mathbf{e}_2 \end{cases} \tag{A.1}$$

with $\mathbf{e}_1$ and $\mathbf{e}_2$ the first two vectors of an orthonormal basis. Importantly, u_1, v_1, and v_2 only depend on scalar products $\|\mathbf{u}^{(t)}\|^2$, $\|\mathbf{v}^{(t)}\|^2$, and $\left(\mathbf{u}^{(t)}\right)^\top \mathbf{v}^{(t)}$:

$$\begin{cases} u_1 = \|\mathbf{u}^{(t)}\| \\ v_1 = \left(\mathbf{v}^{(t)}\right)^\top \mathbf{e}_1 = \frac{1}{\|\mathbf{u}^{(t)}\|} \left(\mathbf{v}^{(t)}\right)^\top \mathbf{u}^{(t)} \\ v_2 = \sqrt{\|\mathbf{v}^{(t)}\|^2 - v_1^2} = \sqrt{\|\mathbf{v}^{(t)}\|^2 - \frac{1}{\|\mathbf{u}^{(t)}\|^2} \left(\left(\mathbf{v}^{(t)}\right)^\top \mathbf{u}^{(t)}\right)^2} \end{cases} \tag{A.2}$$

The kernel limit in Eq. (10) can be rewritten as an integral over two Gaussian random variables w_1 and w_2:

$$k_0\left(\mathbf{u}^{(t)}, \mathbf{v}^{(t)}\right) = \int dw_1 dw_2 p(w_1) p(w_2) f(w_1 u_1) f(w_1 v_1 + w_2 v_2) \tag{A.3}$$

$$\equiv k\left(\|\mathbf{u}^{(t)}\|^2, \|\mathbf{v}^{(t)}\|^2, \left(\mathbf{u}^{(t)}\right)^\top \mathbf{v}^{(t)}\right), \tag{A.4}$$

since u_1, v_1, v_2 only depend on $\|\mathbf{u}^{(t)}\|^2, \|\mathbf{v}^{(t)}\|^2, \left(\mathbf{u}^{(t)}\right)^\top \mathbf{v}^{(t)}$. Thus, the kernel limit for all RC algorithms with random Gaussian weights is an iterable kernel, allowing us to define an associated Recurrent Kernel. As we see in this proof, it can be extended to any rotationally-invariant distribution of weights $p(\mathbf{w})$.

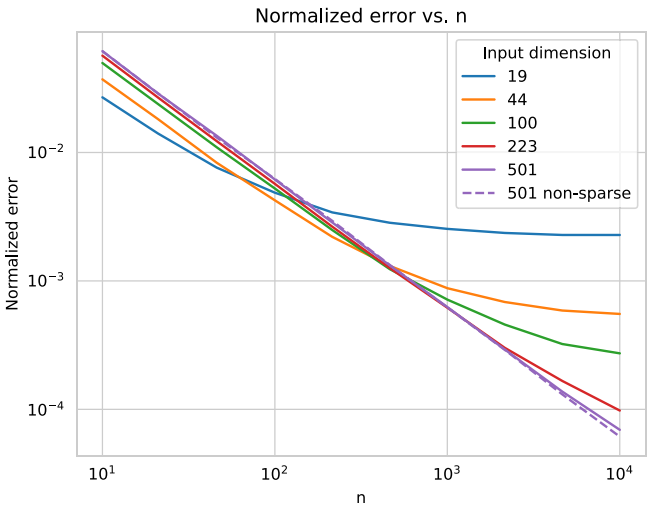

Fig. B.6. Approximation error Eq. (B.2) of sparse Random Features as a function of Random Feature dimension n. An example of non-sparse convergence is given with dashed lines.

Appendix B. Sparse random features

We investigate the convergence of Eq. (9) to its single-step limit defined in Eq. (10) when the weights $\mathbf{w}$ are sparse. This can be interpreted as a sparse Random Feature embedding.

We initialize a random vector $\mathbf{u} \in \mathbb{R}^d$ (i.i.d. uniform between 0 and 1) and generate Random Features embedding following:

$$\psi(\mathbf{u}) = f(\mathbf{W}\mathbf{u}). \tag{B.1}$$

$\mathbf{W} \in \mathbb{R}^{n \times d}$ is an i.i.d. random matrix and f an element-wise nonlinearity. In this context, we can also define the single-step kernel function $k_0(\mathbf{u}, \mathbf{v})$ of Eq. (10). The approximation error is given by:

$$l = \left| \left(\psi(\mathbf{u})\right)^\top \psi(\mathbf{v}) - k_0(\mathbf{u}, \mathbf{v}) \right|^2. \tag{B.2}$$

This quantity is displayed in Fig. B.6 as a function of Random Feature dimension n, for different input dimension and for non-sparse (Gaussian) and sparse ($s = 0.1$) random vector $\mathbf{w}$. It is averaged over 10^4 repetitions.

First, we see that in the non-sparse case, convergence is achieved with a linear rate in $1/n$; only a single curve is displayed as the non-sparse curves only differ by a prefactor. In the sparse case, convergence is similar for small n, until a certain value after which the approximation error reaches a plateau. This shows that the sparsity level $s = 0.1$ does not affect convergence of the Random Features to their kernel limit up that threshold on the output dimension n. This threshold varies greatly with the input dimension d. The greater the input dimension, the greater the number of random weights, and the less sparsity is affecting the convergence of sparse Random Features.

In Reservoir Computing, input and output sizes d and n are similar, we see that this operating point is before this threshold for $s = 0.1$. Instead, the sparsity level s can be varied as displayed in Fig. 3.

Appendix C. Derivation of the limit for leaky reservoir computing

We motivate here the definition of the leaky Recurrent Kernel as defined in Eq. (14). Using the leaky RC update Eq. (5), we have:

$$\left(\mathbf{x}^{(t+1)}\right)^\top \mathbf{y}^{(t+1)} = \left(\frac{a}{\sqrt{N}} f\left(\mathbf{W}\mathbf{u}^{(t)}\right) + (1-a)\mathbf{x}^{(t)}\right)^\top \left(\frac{a}{\sqrt{N}} f\left(\mathbf{W}\mathbf{v}^{(t)}\right) + (1-a)\mathbf{y}^{(t)}\right) \tag{C.1}$$

$$= \frac{a^2}{N} f\left(\mathbf{W}\mathbf{u}^{(t)}\right)^\top f\left(\mathbf{W}\mathbf{v}^{(t)}\right) + \frac{a(1-a)}{\sqrt{N}} f\left(\mathbf{W}\mathbf{u}^{(t)}\right)^\top \mathbf{y}^{(t)} + \frac{a(1-a)}{\sqrt{N}} f\left(\mathbf{W}\mathbf{v}^{(t)}\right)^\top \mathbf{x}^{(t)} + (1-a)^2 \left(\mathbf{x}^{(t)}\right)^\top \mathbf{y}^{(t)} \tag{C.2}$$

The first term converges to the non-sparse limit $k_0\left(\mathbf{u}^{(t)}, \mathbf{v}^{(t)}\right)$. The last term corresponds to the previous recurrent kernel limit $k^{(t)}(\mathbf{i}^{(t-1)}, \mathbf{j}^{(t-1)}, \ldots)$. Furthermore, we neglect the two cross-product terms, which results in Eq. (14). These cross-products are not straightforward to analyze as the previous reservoir state $\mathbf{y}^{(t)}$ also depend on the random weights $\mathbf{W}$.

References

[1] H. Jaeger, The Echo State Approach to Analysing and Training Recurrent Neural Networks-With an Erratum Note, GMD Technical Report 148, German National Research Center for Information Technology, Bonn, Germany, 2001, p. 13.

[2] M. Lukoševičius, H. Jaeger, Reservoir computing approaches to recurrent neural network training, Comp. Sci. Rev. 3 (2009) 127–149.

[3] F. Damicelli, C.C. Hilgetag, A. Goulas, Brain connectivity meets reservoir computing, PLoS Comput. Biol. 18 (2022) e1010639.

[4] H. Jaeger, M. Lukoševičius, D. Popovici, U. Siewert, Optimization and applications of echo state networks with leaky-integrator neurons, Neural Netw. 20 (2007) 335–352.

[5] J. Dong, R. Ohana, M. Rafayelyan, F. Krzakala, Reservoir computing meets recurrent kernels and structured transforms, Adv. Neural Inf. Process. Syst. 33 (2020) 16785–16796.

[6] C. Gallicchio, A. Micheli, L. Pedrelli, Deep reservoir computing: A critical experimental analysis, Neurocomputing 268 (2017) 87–99.

[7] G. Tanaka, T. Yamane, J.B. Héroux, R. Nakane, N. Kanazawa, S. Takeda, H. Numata, D. Nakano, A. Hirose, Recent advances in physical reservoir computing: A review, Neural Netw. 115 (2019) 100–123.

[8] J. Dong, M. Rafayelyan, F. Krzakala, S. Gigan, Optical reservoir computing using multiple light scattering for chaotic systems prediction, IEEE J. Sel. Top. Quantum Electron. 26 (2019) 1–12.

[9] M. Rafayelyan, J. Dong, Y. Tan, F. Krzakala, S. Gigan, Large-scale optical reservoir computing for spatiotemporal chaotic systems prediction, Phys. Rev. X 10 (2020) 041037.

[10] D.J. Gauthier, E. Bollt, A. Griffith, W.A. Barbosa, Next generation reservoir computing, Nature Commun. 12 (5564) (2021).

[11] J. Dong, E. Börve, M. Rafayelyan, M. Unser, Asymptotic stability in reservoir computing, in: 2022 International Joint Conference on Neural Networks, IJCNN, IEEE, 2022, pp. 01–08.

[12] H. Jaeger, H. Haas, Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication, Science 304 (2004) 78–80.

[13] Y. Xue, L. Yang, S. Haykin, Decoupled echo state networks with lateral inhibition, Neural Netw. 20 (2007) 365–376.

[14] C. Gallicchio, A. Micheli, Architectural and markovian factors of echo state networks, Neural Netw. 24 (2011) 440–456.

[15] C. Gallicchio, Sparsity in reservoir computing neural networks, in: 2020 International Conference on Innovations in Intelligent SysTems and Applications, INISTA, IEEE, 2020, pp. 1–7.

[16] F.X.X. Yu, A.T. Suresh, K.M. Choromanski, D.N. Holtmann-Rice, S. Kumar, Orthogonal random features, in: Advances in Neural Information Processing Systems, vol. 29, 2016.

[17] C. Huang, V.J. Sorger, M. Miscuglio, M. Al-Qadasi, A. Mukherjee, L. Lampe, M. Nichols, A.N. Tait, T. Ferreira de Lima, B.A. Marquez, et al., Prospects and applications of photonic neural networks, Adv. Phys.: X 7 (2022) 1981155.

[18] A. Rahimi, B. Recht, Random features for large-scale kernel machines, in: Advances in Neural Information Processing Systems, vol. 20, 2007.

[19] G.B. Huang, Q.Y. Zhu, C.K. Siew, Extreme learning machine: theory and applications, Neurocomputing 70 (2006) 489–501.

[20] D.J. MacKay, et al., Introduction to gaussian processes, NATO ASI Series F Comput. Syst. Sci. 168 (1998) 133–166.

[21] A.K. Malik, R. Gao, M. Ganaie, M. Tanveer, P.N. Suganthan, Random vector functional link network: recent developments, applications, and future directions, Appl. Soft Comput. (2023) 110377.

Giuseppe Alessio d'Inverno is a researcher at SISSA, Trieste, Italy. He obtained his Ph.D. from the University of Siena, Italy, under the supervision of Maria Lucia Sampoli, Franco Scarselli, and Monica Bianchi. His research interests are the mathematical foundations of deep learning, graph neural networks, scientific machine learning, and Physics-Informed Neural Networks.

Jonathan Dong is currently an SNF Ambizione Fellow at the École Polytechnique Fédérale de Lausanne (EPFL), Switzerland. He completed his Ph.D. at École Normale Supérieure in Paris, France, under the supervision of Sylvain Gigan and Florent Krzakala. His research focuses on computational imaging, non-linear optimization, and efficient machine learning algorithms.