Information Sciences 742 (2026) 123275

Contents lists available at ScienceDirect

Information Sciences

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

Bregman Proximal Gradient with extrapolation to train a Reservoir Computing network for a binary classification task

Carmen Mayora-Cebollero [a,*], Ana Mayora-Cebollero [a], Álvaro Lozano [b], Roberto Barrio [a]

[a] IUMA, CoDy and Department of Applied Mathematics, Universidad de Zaragoza, Zaragoza, Spain
[b] IUMA, CoDy and Department of Mathematics, Universidad de Zaragoza, Zaragoza, Spain

ARTICLE INFO

Keywords:
Bregman Proximal Gradient with extrapolation (BPGe)
Optimization algorithms
Reservoir Computing
Training ANNs
Binary classification

ABSTRACT

This study examines the training process of Artificial Neural Networks, specifically focusing on Reservoir Computing and on binary classification tasks. Training involves optimizing the network's parameters to minimize the error of a loss function, which quantifies the discrepancy between network outputs and target data. Several numerical optimizers are used in the literature. Here, we formulate the supervised learning problem using the recently introduced Bregman Proximal Gradient with extrapolation (BPGe) algorithm for non-convex optimization problems. We compare this new method with the classical Stochastic Gradient Descent (SGD), the Root Mean Square Propagation (RMSProp), the Adaptive Moment Estimation (Adam), and two variants with Nesterov momentum (NAG and NAdam). The new approach leads to an accurate and significantly faster numerical algorithm for solving supervised learning problems on binary classification tasks in Reservoir Computing. Three test examples are presented, one based on chaotic data classification in the classical Lorenz system, another on the Human Activity Recognition (HAR) using smartphones dataset for movement–rest classification, and the last one on the MNIST dataset for even–odd classification. We show that our approach is highly competitive with existing methods in the tests performed.

1. Introduction

In recent years, Machine Learning, particularly Deep Learning (DL) and Artificial Neural Networks (ANNs), has become essential for solving numerous problems. A critical aspect of their development is the training process. In this article, we are interested in the problem of supervised learning, in particular, for a binary classification task. Training an ANN (for a supervised task) is the process of optimizing the trainable parameters, i.e., the weights and biases, to find the lowest error in a suitable function between the outputs and the target data (loss function). Therefore, optimization is a key component in any Deep Learning task and an important field of study nowadays. Several optimizers have been used in the literature [1], as Gradient Descent (GD) [2] and its variants, Stochastic Gradient Descent (SGD), Root Mean Square Propagation (RMSProp) [3], the most currently used Adaptive Moment Estimation (Adam) [4], and many others that have emerged to address limitations or to improve these well-known classical optimizers considering small changes in some cases (for example, the signSGD optimizer [5], a Bregman learning framework [6] or an algorithm with adaptive Nesterov momentum [7]).

* Corresponding author.
Email addresses: cmayora@unizar.es (C. Mayora-Cebollero), rbarrio@unizar.es (R. Barrio).

https://doi.org/10.1016/j.ins.2026.123275
Received 15 September 2025; Received in revised form 20 February 2026; Accepted 20 February 2026
Available online 23 February 2026

Different problems need different loss functions in the optimization process. Notice that one of the most common loss functions used in Machine Learning is the squared loss due to its smoothness. However, some problems require loss functions with non-smooth terms. Due to the lack of smoothness, some higher-order methods and line-search–based approaches are excluded. Therefore, in most cases, problems are addressed using variations and extensions of Gradient Descent (first-order methods), with modifications designed to handle non-smoothness. To deal with such non-smoothness, the introduction of smooth approximation techniques can be considered (as shown in the epsilon-SSVR model [8] and in smoothed hinge-loss approaches [9]). Furthermore, to solve a non-smooth optimization problem, Primal-Dual methods can be applied; that is, the problem can be transformed into a min-max problem and solved using a Primal-Dual method (such as Pdprox algorithm [10] or other efficient schemes [11]). A similar situation occurs with convexity; although more optimizers are addressed for convex problems, non-convex functions are present in Machine Learning problems and therefore non-convex optimizers are needed [12]. Moreover, there are other optimizers suitable for non-convex nonsmooth minimization problems [13].

Although GD, SGD, RMSProp and Adam are usually the most applied optimizers for solving Deep Learning problems, there are other families of optimizers in the literature that can be adapted and used in Machine Learning algorithms [1]. For example, second-order methods or Proximal Gradient algorithms. Some second-order methods have been used to solve Neural Networks problems with promising results, such as the Hessian-free optimizers [14], the quasi-Newton methods [15], the K-FAC (Kronecker Factored Approximate Curvature) methods [16], and the second-order method Shampoo [17]. On the other hand, Proximal Gradient algorithms have been applied or adapted to solve Machine Learning and Neural Networks problems (for instance, for sparse Neural Networks [6], for regression problems [18], for classification problems [19] or for both aforementioned tasks [20]). In this paper, we propose to adapt a Bregman Proximal Gradient method with extrapolation [13] (that belongs to the Proximal Gradient family) to Artificial Neural Networks, in particular, to a Reservoir Computing network. Bregman Proximal minimization algorithms have recently been introduced to solve non-convex and non-smooth optimization problems and have proven useful in several practical applications [20].

Therefore, the study of the performance of different optimizers of diverse nature and specifics for different types of problems (convex, non-convex, smooth, non-smooth,…) is of great interest in ANN training. The development and availability of optimizers that can outperform standard methods in certain problems, or improve convergence speed, can be very useful in practice, for example in real-time applications. In this paper, we develop complete formulas for adapting the Bregman Proximal Gradient algorithm with extrapolation (BPGe), introduced in [13], to train Reservoir Computing networks for binary classification. We compare the results obtained with this adapted Bregman Proximal Gradient with extrapolation (BPGe) algorithm against with three classical optimizers (SGD, RMSProp and Adam) and two variants with Nesterov momentum (Nesterov Accelerated Gradient [21], NAG, optimizer and Adam with Nesterov momentum [22], NAdam, optimizer). Our results indicate that BPGe can be a competitive optimization algorithm improving convergence speed in the considered binary classification task using a Recurrent-like Neural Network (Reservoir Computing). Therefore, the main contributions of this work are the study of BPGe from a theoretical point of view developing the mathematical formulation for binary classifications, and from an applied perspective comparing the results with optimizers widely used in the ANNs framework.

The main objective of this work is to present BPGe as an optimizer that can be used with remarkable results for binary classification tasks being competitive with the classical optimizers. As expected by the No Free Lunch theorems [23], BPGe is the best option for some test cases and hyperparameter configurations, and a very competitive option in other cases.

The paper is organized as follows. In Section 2 we introduce all the optimization algorithms. In Section 3 we present the Reservoir Computing architecture. Later, in Section 4 we adapt the Bregman Proximal Gradient with extrapolation algorithm to Reservoir Computing. In Section 5 we detail the results of all the optimizers for different classification tasks: a dynamical classification problem applied to the classical Lorenz chaotic system (Section 5.2), a movement–rest activity classification on the Human Activity Recognition (HAR) using smartphones dataset (Section 5.3), and an even–odd classification on the MNIST dataset (Section 5.4). Finally, in Section 6 we draw some conclusions and in the Appendix we show the Python code of our BPGe algorithm.

All the experiments run on a Linux box with 2 Xeon Gold 6338 with 1Tb of DDR4-3200 RAM with two nVidia A30 and one nVidia A40.

2. Minimization algorithms for Artificial Neural Networks

In this section, we review the six optimizers used in this study, first the classical ones and then the Bregman Proximal Gradient algorithm with extrapolation (which will be adapted to ANNs in Section 4).

2.1. Classical optimizers for Artificial Neural Networks

Consider an ANN as a function $\mathcal{N}_{\mathcal{W}}$ transforming any input $x \in \mathbb{R}^n$ into an output $\hat{y} \in \mathbb{R}^m$, that is, $\mathcal{N}_{\mathcal{W}}(x) = \hat{y}$, where $\mathcal{W} = (W, b)$ are the trainable parameters (weights and biases). The goal will be to obtain $\hat{y}$ as close as possible to a target value (label) $y \in \mathbb{R}^m$. To achieve this, we tune $\mathcal{W}$ using data. Such data, known as training data, consists of N data points (x_j, y_j) for $j = 1, \dots, N$, where x_j are the inputs and y_j are the corresponding labels (or target outputs). A loss function $\mathcal{L}_{\mathcal{W}}$ that measures the difference between the target y_j and the network output $\hat{y}_j = \mathcal{N}_{\mathcal{W}}(x_j)$ is defined. Therefore, training process consists of solving the following optimization problem:

$$\min_{\mathcal{W}} \mathcal{L}_{\mathcal{W}} = \min_{\mathcal{W}} \frac{1}{N} \sum_{j=1}^{N} \ell(y_j, \mathcal{N}_{\mathcal{W}}(x_j)), \tag{1}$$

where $\ell(\cdot,\cdot)$ is the loss function applied to each data point.

Gradient Descent (GD) [2] is a classical explicit-gradient optimization algorithm. It relies on moving in the direction of descending gradient to, iteratively, find a minimum (gradient equal to 0). In the field of Deep Learning, this algorithm is also known as batch GD as all the training set is used to compute the gradient at each iteration (epoch). As for large training sets the use of the whole dataset for the gradient computation can slow the optimization process, Stochastic Gradient Descent (SGD) was proposed. SGD uses just one sample (chosen randomly from training set) to approximate the gradient at each iteration. Once all the training data points are used, an epoch is completed. This definition of epoch matches with the one of Gradient Descent, where in one epoch all points contribute to the parameter update. This algorithm is faster than GD, but less regular because of its stochastic nature. To try to mitigate this effect, mini-batch SGD is applied. In mini-batch SGD, training dataset is divided into subsets of size B (known as batches) and at each iteration a subset is used to compute the gradient until an epoch is completed. That is, the update rule is

$$\mathcal{W}_i \leftarrow \mathcal{W}_{i-1} - \lambda \frac{1}{B} \sum_{j=1}^{B} \nabla_{\mathcal{W}} \ell(y_j, \mathcal{N}_{\mathcal{W}_{i-1}}(x_j)),$$

with λ the learning rate (similar to a step size). Mini-batch SGD reduces to GD when $B = N$ and to SGD when $B = 1$. For simplicity, we will use SGD to refer to GD, SGD or mini-batch SGD from now on (the size of the batch will be indicated).

For computational purposes, mini-batch configurations are usually considered to compute the gradient in the explicit-gradient algorithms. To simplify the notation, we define

$$\nabla_{\mathcal{W}} \mathcal{L}^{B}_{\mathcal{W}_{i-1}} = B^{-1} \sum_{j=1}^{B} \nabla_{\mathcal{W}} \ell(y_j, \mathcal{N}_{\mathcal{W}_{i-1}}(x_j)).$$

Notice that in these GD-like algorithms, the learning rate λ is constant. An appropriate value for λ is crucial for a good performance of the minimization algorithm. A high value can cause convergence issues, while one that is too low can slow down convergence. Moreover, when working with ANNs, different parameters can have different magnitudes for the gradients. To avoid such problems, algorithms such as AdaGrad and RMSProp use an adaptive learning rate (it changes over the iterations and the parameters).

Root Mean Square Propagation (RMSProp) [3] is an optimizer introduced by G. Hinton and T. Tieleman in 2012. It can be considered as a modification of Adaptive Gradient (AdaGrad) [24], with RMSProp performing better in general. It has two steps in each iteration. The first one consists of a moving average (exponential decay) of the gradients (main difference with AdaGrad, which uses a sum of square gradients). The second step is similar to GD-like algorithms but with an adaptive learning rate like AdaGrad. This adaptation of the value of the learning rate gives more versatility to the algorithm as it can adapt to the slope of the optimization hypersurface (sharper, plateau,…). Mathematically, each iteration is represented as

$$\begin{aligned} v_i &\leftarrow \rho v_{i-1} + (1-\rho)\left(\nabla_{\mathcal{W}} \mathcal{L}^{B}_{\mathcal{W}_{i-1}} \otimes \nabla_{\mathcal{W}} \mathcal{L}^{B}_{\mathcal{W}_{i-1}}\right), \\ \mathcal{W}_i &\leftarrow \mathcal{W}_{i-1} - \lambda \nabla_{\mathcal{W}} \mathcal{L}^{B}_{\mathcal{W}_{i-1}} / (\epsilon + \sqrt{v_i}), \end{aligned}$$

where ρ is a smoothing constant of the exponential decay and ϵ is a small number used for numerical stability. RMSProp was one of the preferred optimizers until the introduction of Adam.

Adaptive Moment Estimation (Adam) [4] is an optimizer related to RMSProp that was proposed by D.P. Kingma and J. Ba in 2014. It has several advantages, such as computational efficiency and requiring minimal tuning of the hyperparameters. Adam is based on the estimation of the first (mean) and second (uncentered variance) raw moments of the gradient. To estimate the first and second moments, Adam computes momentum as an exponentially decaying average of past gradients, and the exponentially decaying average of past squared gradients of RMSProp, respectively. The algorithm is given by:

$$\begin{aligned} m_i &\leftarrow \beta_1 m_{i-1} + (1-\beta_1)\nabla_{\mathcal{W}} \mathcal{L}^{B}_{\mathcal{W}_{i-1}}, \\ v_i &\leftarrow \beta_2 v_{i-1} + (1-\beta_2)\left(\nabla_{\mathcal{W}} \mathcal{L}^{B}_{\mathcal{W}_{i-1}} \otimes \nabla_{\mathcal{W}} \mathcal{L}^{B}_{\mathcal{W}_{i-1}}\right), \\ \hat{m}_i &\leftarrow m_i / (1-(\beta_1)^i), \\ \hat{v}_i &\leftarrow v_i / (1-(\beta_2)^i), \\ \mathcal{W}_i &\leftarrow \mathcal{W}_{i-1} - \lambda \hat{m}_i / (\epsilon + \sqrt{\hat{v}_i}), \end{aligned} \tag{2}$$

where $\beta_1, \beta_2 \in [0, 1)$, and the steps $\hat{m}_i$ and $\hat{v}_i$ are bias-corrected estimates to counteract the 0 initialization of m_0 and v_0.

There exist variations of these classical optimizers obtained, for example, by adding Nesterov momentum [25] to the optimizers. In this work, we consider the Nesterov Accelerated Gradient [21] (NAG) optimizer and Adam with Nesterov momentum [22] (NAdam). The formulation of Nesterov Accelerated Gradient (NAG) is given by

$$\begin{aligned} b_i &\leftarrow \mu b_{i-1} + \nabla_{\mathcal{W}} \mathcal{L}^{B}_{\mathcal{W}_{i-1}}, \\ \mathcal{W}_i &\leftarrow \mathcal{W}_{i-1} - \lambda \left(\nabla_{\mathcal{W}} \mathcal{L}^{B}_{\mathcal{W}_{i-1}} + \mu\, b_i\right), \end{aligned}$$

where μ is the momentum. In the case of NAdam its formulation coincides with that of Adam given in Eq. (2), except for the update of $\hat{m}_i$ that is computed as follows:

$$\begin{aligned} \mu_i &\leftarrow \beta_1 \left(1 - \tfrac{1}{2} 0.96^{i\psi}\right), \\ \mu_{i+1} &\leftarrow \beta_1 \left(1 - \tfrac{1}{2} 0.96^{(i+1)\psi}\right), \\ \hat{m}_i &\leftarrow \frac{\mu_{i+1} m_i}{1 - \prod_{j=1}^{i+1} \mu_j} + \frac{(1-\mu_i)\nabla_{\mathcal{W}} \mathcal{L}^B_{\mathcal{W}_{i-1}}}{1 - \prod_{j=1}^{i} \mu_j}, \end{aligned}$$

where ψ corresponds with the momentum decay.

2.2. Bregman Proximal Gradient with extrapolation (BPGe) algorithm

Bregman Proximal Gradient with extrapolation (BPGe) algorithm [13] is an implicit-gradient optimizer that can solve non-convex and non-smooth minimization problems. The problem statement for BPGe is the following. The function $\Psi(x)$ to minimize can be expressed as the sum of a non-convex continuously differentiable function $f(x)$ (which possibly is not necessarily globally Lipschitz gradient continuous) and a proper lower-semi-continuous convex function $g(x)$. Then, the algorithm is divided into two stages, in the first one the extrapolation is performed, and in the second one a proximal operator is computed. Mathematically, if we denote by y_{i-1} the extrapolation value and by x_i the current value of the iteration,

$$\begin{aligned} y_{i-1} &\leftarrow x_{i-1} + \beta_{i-1}(x_{i-1} - x_{i-2}), \\ x_i &\leftarrow \arg\min_x \left\{ g(x) + \langle \nabla f(y_{i-1}), x - y_{i-1} \rangle + D_h(x, y_{i-1})/\lambda_{i-1} \right\}, \end{aligned} \tag{3}$$

where λ_i is the step size of the iteration (that is, the learning rate), β_i is the extrapolation parameter, and D_h is a Bregman distance [26]. The extrapolation parameter β_i can be determined at each iteration using a line search algorithm [13]. To apply this algorithm, we compute $C_i = 1/(1+\lambda_i \mu)$. The value of parameter μ satisfies that f is μ-weakly convex relative to h (with h being the Bregman function that defines the Bregman distance D_h). Then, starting from $\beta_i = \beta_0$, the value of β_i is updated as $\beta_i = \eta\beta_i$ while $D_h(x_i, x_i + \beta_i(x_i - x_{i-1})) > \rho\, C_i\, D_h(x_{i-1}, x_i)$. The values of $\eta \in (0,1)$, $\beta_0 \in [0,1)$ and $\rho \in (0,1)$ are established at the outset of the process.

This formulation of BPGe generalizes a family of Proximal Gradient algorithms. In particular, if $D_h(x,y) = \|x-y\|^2/2$, BPGe corresponds to Proximal Gradient algorithm with extrapolation (PGe). If moreover $\beta_i = 0$, BPGe reduces to Proximal Gradient (PG) method. Finally, if just $\beta_i = 0$, it is BPG without extrapolation. Theoretical convergence results of BPGe using results of non-convex analysis (Bregman distance, Kurdyka-Łojasiewicz property, …) can be found in [13].

2.3. A benchmark minimization test

In this paper, we aim to adapt BPGe to Reservoir Computing for binary classification. To demonstrate its performance we will compare it with some optimization algorithms used in ANNs field such as SGD, RMSProp and Adam. But first, we will compare them in a benchmark minimization problem.

In Fig. 1 we show the evolution of four optimizers (SGD, RMSProp, Adam and BPGe) for the benchmark function known as three-hump camel function given by

$$H(\boldsymbol{x}) = H(x,y) = 2x^2 - 1.05x^4 + x^6/6 + xy + y^2.$$

This function has two local minima and a global minimum at $\boldsymbol{x}_* = (x_*, y_*) = (0,0)$, so it is an appropriate function to demonstrate the performance of the optimization algorithms. For the simulations, we take the initial condition $\boldsymbol{x}_0 = (x_0, y_0) = (1.8, 1.8)$ and the learning rate value $\lambda = 0.01$. For SGD, RMSProp and Adam, the corresponding PyTorch [27] routines have been used (with all the parameters set to default except for the learning rate). For BPGe, the routine has been implemented from scratch with parameters $\beta_0 = 0.99$, $\rho = 0.99$, $\eta = 0.95$ and $\mu = 6$ (following the notation used to introduce BPGe, $f = H$, $g = 0$ and $D_h(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{x} - \boldsymbol{y}\|^2/2$). As stop rules, we set the maximum number of iterations to 4000 and the tolerance value to 10^{-15} for the squared Euclidean norm between two consecutive iterations, $\|\boldsymbol{x}_{i+1} - \boldsymbol{x}_i\|_2^2 = \|(x_{i+1}, y_{i+1}) - (x_i, y_i)\|_2^2 < 10^{-15}$. Notice that SGD (in black) evolves towards a local minimum, but not to the global one. The other three algorithms correctly find the global minimum: RMSProp (in light blue) and Adam (in purple) with a very similar path, and BPGe (in red) with a different one and it seems to do so with a much smaller number of iterations.

In Table 1, we present some results about the computational time of each optimizer on the simple benchmark minimization problem of the three-hump camel function. On the one hand, the mean time per optimization step is shown in seconds (s). This time corresponds with the mean time that each optimizer uses for applying the optimization step across all the computed iterations. On the other hand, the time (in seconds) that each optimizer needs to achieve a point $\boldsymbol{x}_i = (x_i, y_i)$ in the plane with a squared Euclidean distance with respect to the global minimum $\boldsymbol{x}_* = (x_*, y_*) = (0,0)$ less than or equal to 10^{-3}, 10^{-4}, 10^{-5} or 10^{-6} is indicated. It can be clearly seen that BPGe is the optimizer with less mean time per optimization step and with less time needed to reach the studied distances, which supports that it seems to be the optimizer with faster convergence.

As supplementary material, a video with the evolution of the four optimizers is provided. This multimedia material can also be consulted at https://www.youtube.com/watch?v=cgcXCUczYUk.

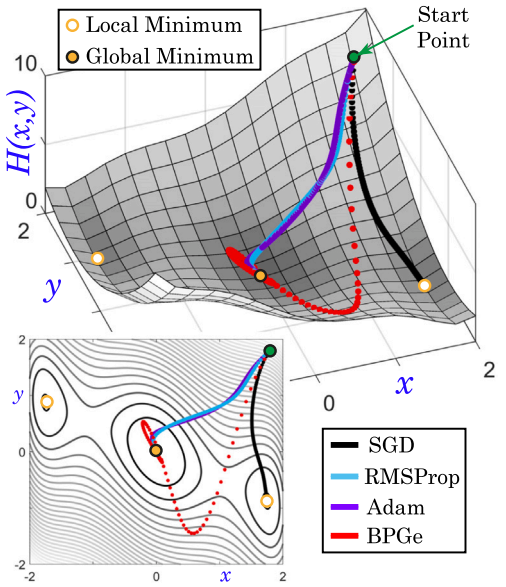

Fig. 1. Find the global minimum in the benchmark three-hump camel function. Evolution of SGD (in black), RMSProp (in light blue), Adam (in purple) and BPGe (in red) for such optimization problem.

Table 1
Results of the search for the global minimum in the benchmark three-hump camel function. Top table: Study of the mean time per optimization step indicated in seconds (s) for each optimizer in this benchmark function. Table below: Time in seconds (s) that each optimizer needs to reach a target distance (in squared Euclidean norm) to the global minimum $\boldsymbol{x}_* = (x_*, y_*) = (0, 0)$ of the function.

Mean time per optimization step (s)				
	SGD	RMSProp	Adam	BPGe
	$1.042 \cdot 10^{-4}$	$1.715 \cdot 10^{-4}$	$2.567 \cdot 10^{-4}$	$1.977 \cdot 10^{-6}$
Time needed to reach a distance (s)				
$\|\boldsymbol{x}_* - \boldsymbol{x}_i\|_2^2$	SGD	RMSProp	Adam	BPGe
$\leq 10^{-3}$	–	0.248	0.507	$2.680 \cdot 10^{-4}$
$\leq 10^{-4}$	–	0.293	0.611	$5.047 \cdot 10^{-4}$
$\leq 10^{-5}$	–	0.328	0.704	$6.144 \cdot 10^{-4}$
$\leq 10^{-6}$	–	0.357	0.788	$7.162 \cdot 10^{-4}$

3. Reservoir Computing

Reservoir Computing (RC) [28,29] is a special type of Recurrent-like ANN whose main structure is the reservoir. The reservoir is a set of neurons whose connection weights are set randomly and are not tuned during training (this considerably simplifies the training problem as only a small fraction of the network is trained, keeping the reservoir fixed). RC includes three categories of setups: Echo-State Networks (ESNs) [30], Liquid State Machines (LSMs) [31], and BackPropagation DeCorrelation (BPDC) learning rules [32]. In this paper, we focus on the use of ESNs for binary classification tasks.

Echo-State Networks (ESNs) are a type of Reservoir Computing approach that has been widely and successfully applied to different applications (for simulated data [33] and also for experimental data [34]). The architecture of an ESN can be divided into three parts (all of them consisting of artificial neurons): the input layer, the reservoir, and the output layer (also known as readout layer). Mathematically, to perform a classification task with K classes, for a time series input $x = \{x_0, x_1, \ldots, x_T\}$ (with $x_i \in \mathbb{R}^n$), the output $\hat{y} \in \mathbb{R}^K$ of the ESN (with leaky integrator neurons and a reservoir with M neurons) is computed as follows:

$$\begin{aligned} h_t &= \alpha \tanh(W_{in}\, x_t + W_{res}\, h_{t-1}) + (1-\alpha) h_{t-1},\ t \in [1, T], \\ \hat{y} &= \text{softmax}(W_{out}\, h_T + b_{out}). \end{aligned} \tag{4}$$

The first equation describes the input-reservoir and reservoir-reservoir computations, that is, the non-trainable part of the network. The second equation corresponds to the readout layer and, therefore, the trainable part. In the first equation, $h_t \in \mathbb{R}^M$ is the state of

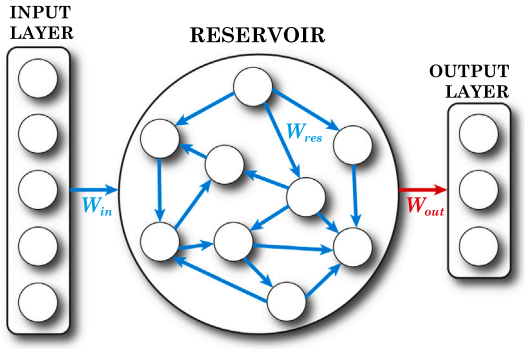

Fig. 2. Graphical representation of an ESN, with an input layer of 5 neurons, a reservoir with 9 neurons, and an output layer with 3. In blue we have marked the connections whose weights are fixed (non-trained), and with red these that have to be fine-tuned during training.

the reservoir neurons at time t, $\alpha \in [0, 1]$ is the leaking rate parameter (since we use leaky integrator neurons), `tanh` is the hyperbolic tangent function, $W_{in} \in \mathbb{R}^{M \times n}$ is the fixed matrix of weights between the input layer and the reservoir, and $W_{res} \in \mathbb{R}^{M \times M}$ is the fixed weight matrix of the recurrent connections of the reservoir with itself. In the second equation, `softmax` is the function applied to transform the output values into scores (probabilities) for each class, $h_T \in \mathbb{R}^M$ is the last state of the reservoir, $W_{out} \in \mathbb{R}^{K \times M}$ is the trainable weight matrix for the readout layer, and $b_{out} \in \mathbb{R}^K$ is the trainable bias term for such an output layer. In Fig. 2, a graphical representation of an ESN with $n = 5$, $M = 9$ and $K = 3$ is shown (blue color has been used for connections with non-trainable weights and red color for the trainable ones).

As indicated in [35], some guidelines can be followed when creating fixed matrices W_{in} and W_{res} to obtain a proper ESN. The reservoir matrix W_{res} is preferred to be sparse (each reservoir neuron is connected to a small number of other reservoir neurons) for speed-up purposes. The reservoir should satisfy the echo-state property [30], that is, the state of the reservoir $h(\cdot)$ should not depend on the initial conditions for a sufficiently long input x. In general, a spectral radius of W_{res} less than 1 ensures that such property holds. In the case of W_{in}, its distribution is taken equal to that of W_{res} but with a dense format. The values of the non-trainable weights can be set using a normal distribution $\mathcal{N}(0, a)$ or a uniform distribution $\mathcal{U}(-a, a)$, where a is a scaling parameter.

3.1. Architecture of the ESN for optimizers comparison

To ensure a fair comparison analysis of the performance of the optimizers, we need to fix the structure and hyperparameters of the ANN architecture for each considered test case (Lorenz system, Human Activity Recognition using smartphones dataset, and MNIST dataset). For the reservoir, we consider a population size (number of neurons in the reservoir) $M = 500$ for the Lorenz system analysis, $M = 100$ for the Human Activity Recognition using smartphones dataset, and $M = 1500$ for the MNIST dataset. The connection matrix between the reservoir neurons with themselves has been built as an oriented graph with no loops whose probability of connection is 0.5. For the weight matrix of the reservoir, i.e., W_{res}, we fix the weights sampling from a normal distribution $\mathcal{N}(0, 3.5)$ and such matrix is modified later to have spectral radius 0.9 (to ensure the echo-state property). The connection matrix between the input and reservoir neurons is non-sparse with the weights (W_{in}) sampled from a Gaussian distribution $\mathcal{N}(0, 3.5)$. The leaking rate α is set to 0.6 for the Lorenz system analysis, to 0.1 for the Human Activity Recognition using smartphones dataset, and to 0.05 for the MNIST dataset. Moreover, we fix the initial values of the trainable parameters in the default values provided by PyTorch [27]. The initial value of the state (h_0) is set to 0 following the default initialization of PyTorch [27] for the states of Recurrent-like Neural Networks. The final time T corresponds to the duration of the time series and will be set for each test example. An early stopping technique [2] is used so the final value of the trainable parameters is that giving the lowest loss value for the validation dataset during the training process. The loss function is the Cross-Entropy Loss with weight decay (for L^2-penalty) equal to 10^{-5}. We remark that the hyperparameters have not been fine-tuned with the objective of not particularizing them to any optimizer and to try to obtain a fair comparison among all algorithms.

4. BPGe algorithm to train a Reservoir Computing network for a binary classification task

In this section we develop the necessary theoretical formalism to apply BPGe to train an Echo-State Network for a binary classification task.

First, note that when training a network, the problem to minimize is the one given in Eq. (1). Therefore, in this case the variables with respect to which we minimize are the trainable parameters $W_{out} \in \mathbb{R}^{2 \times M}$ (trainable weight matrix for the readout layer) and $b_{out} \in \mathbb{R}^2$ (trainable bias term for the readout layer). Let us use $\mathcal{W}$ to refer to $(W_{out}, b_{out}) \in \mathbb{R}^{2M+2}$. For a binary classification task, the usual loss function to minimize is the Cross-Entropy Loss. Moreover, we consider L^2-penalty as the regularization technique to prevent overfitting. Then, using the formulation given in Section 2.2 for BPGe algorithm, we take as f the Cross-Entropy Loss (CEL)

and as g the L^2-penalty. That is,

$$f(\mathcal{W}) = \mathrm{CEL}(\mathcal{W}) = -\sum_{j=1}^{B} \log\left(\frac{e^{c^j_{\{l_j\}}}}{\sum_{k=1}^{K} e^{c^j_{\{k\}}}}\right), \tag{5}$$

where $c^j_{\{k\}} = \sum_{m=1}^{M} w_m^{\{k\}} R_m^j + b^{\{k\}}$ is the value of the output neuron corresponding to class k (before softmax application) for sample j, with $w_m^{\{k\}}$ the m-th weight used to compute class k, R_m^j the value of the m-th neuron of the reservoir (for input sample of index j), and $b^{\{k\}}$ the bias for class k; $\{l_j\}$ is the correct class (label) for sample j; B is the number of samples in the batch; K is the number of classes (for binary classification, $K = 2$); and M is the number of neurons of the reservoir. Besides,

$$g(\mathcal{W}) = L^2(\mathcal{W}) = \gamma \sum_{q=1}^{2M+2} w_q^2, \tag{6}$$

where γ is the regularization parameter (for L^2-penalty) and w_q is any trainable parameter (weight or bias). Notice that these functions satisfy the required properties of the algorithm statement (see Section 2.2): f is a convex function (by composition of convex functions) and continuously differentiable (by composition); and g is a proper lower-semi-continuous function (since it is continuous) and it is convex (by composition).

Notice that to apply the BPGe algorithm (see Eq. 3) we need a Bregman distance D_h that has to be defined by a Bregman function h (mirror map of the optimizer). For the considered binary classification problems, we set $h(\boldsymbol{x}) = \|\boldsymbol{x}\|^2/2$, getting the Bregman distance as

$$D_h(\boldsymbol{x}, \boldsymbol{y}) = \frac{\|\boldsymbol{x} - \boldsymbol{y}\|^2}{2}. \tag{7}$$

With these functions, the conditions required in [13] are satisfied. In particular, f is μ-weakly convex relative to h with $\mu = 0$ as f (see Eq. 5) is convex.

Let us focus on adapting the algorithm to the binary classification task with Reservoir Computing using the new notation $\mathcal{V}_{i-1}$ and $\mathcal{W}_i$ for y_{i-1} and x_i, respectively, in Eq. (3). In particular, let us find the analytical expression for the second step of the algorithm where the proximal operator is computed (see Eq. 3), that with the new notation is rewritten as

$$\mathcal{W}_i \leftarrow \arg\min_{\mathcal{W}} \left\{ g(\mathcal{W}) + \langle \nabla f(\mathcal{V}_{i-1}), \mathcal{W} - \mathcal{V}_{i-1} \rangle + D_h(\mathcal{W}, \mathcal{V}_{i-1})/\lambda_{i-1} \right\}.$$

To obtain $\mathcal{W}_i$ (value of $\mathcal{W}$ for i-th iteration) we have to compute the expression inside the curly brackets, calculate its derivative (the gradient), obtain the value of $\mathcal{W}$ that cancels all the elements of the gradient vector, and check that such an extreme point is a minimum. For the moment we will not compute the expression of $\nabla f(\mathcal{V}_{i-1})$ as it is a constant and does not directly affect the computations related to the minimum. We will compute its expression later as it is the key point of the adaptation of the algorithm to Reservoir Computing for binary classification.

Using Eqs. (6) and (7), the expression inside the curly brackets is

$$g(\mathcal{W}) + \langle \nabla f(\mathcal{V}_{i-1}), \mathcal{W} - \mathcal{V}_{i-1} \rangle + D_h(\mathcal{W}, \mathcal{V}_{i-1})/\lambda_{i-1} = \gamma \sum_{q=1}^{2M+2} w_q^2 + \sum_{q=1}^{2M+2} \big(\nabla f(\mathcal{V}_{i-1})\big)_q \big(w_q - (v_{i-1})_q\big) + \frac{1}{2\lambda_{i-1}} \|\mathcal{W} - \mathcal{V}_{i-1}\|^2.$$

The s-th component of the gradient of this expression is given by

$$2\gamma w_s + (\nabla f(\mathcal{V}_{i-1}))_s + \frac{1}{\lambda_{i-1}}(w_s - (v_{i-1})_s).$$

The value of $\mathcal{W}$ that cancels the previous expression is the vector whose s-th element (which we have previously referred to as w_s) is

$$\frac{-(\nabla f(\mathcal{V}_{i-1}))_s + \frac{1}{\lambda_{i-1}}(v_{i-1})_s}{2\gamma + \frac{1}{\lambda_{i-1}}}. \tag{8}$$

We just have to check that it is a minimum. The Hessian matrix is a diagonal matrix with diagonal elements equal to $2\gamma + 1/\lambda_{i-1}$, so the determinant is $(2\gamma + 1/\lambda_{i-1})^{2M+2} > 0$ (γ is the regularization parameter, and λ_{i-1} is the step size, so they are greater than 0), and the first diagonal element of the matrix is $2\gamma + 1/\lambda_{i-1}$ which is also positive. Therefore, the point $\mathcal{W}$ whose s-th coordinate is given in Eq. (8) is a minimum, and the value $\mathcal{W}_i$ that we are looking for. We just have to compute $\nabla f(\mathcal{V}_{i-1})$ to have the expression totally defined.

Let us compute the expression of the gradient before evaluating it on $\mathcal{V}_{i-1}$. For that, let us refer to the gradient of function $f(\mathcal{W})$ as $\nabla f(\mathcal{W})$. Each element of the gradient is $\partial f/\partial w_s$ where w_s is the s-th element of $\mathcal{W} = (W, b)$. The function f is a sum of B elements, each summand is related to a sample (that has a label) of the batch. Note that according to the form of each summand, it is not the same to compute the derivative with respect to one trainable parameter involved in the computation of the value of the neuron of class l_j if the corresponding label is l_j (this weight/bias appears in the numerator and denominator of f) or if it is not (this weight/bias only appears in the denominator). Therefore, we have to distinguish four possible cases:

C1. We compute the derivative with respect to a weight in $\mathcal{W}$ that contributes to the output neuron associated with the correct class l_j (given by the label) for sample j.
C2. We compute the derivative with respect to a weight in $\mathcal{W}$ that does not contribute to the output neuron associated with the correct class l_j for sample j.
C3. We compute the derivative with respect to a bias in $\mathcal{W}$ that contributes to the output neuron associated with the correct class l_j (as given by the label) for sample j.
C4. We compute the derivative with respect to a bias in $\mathcal{W}$ that does not contribute to the output neuron associated with the correct class l_j for sample j.

For the sake of notation, in what follows we set

$$\mathtt{RC}^{\{k\}} = \exp\left(c^j_{\{k\}}\right) = \exp\left(\sum_{m=1}^{M} w_m^{\{k\}} R_m^j + b^{\{k\}}\right),$$

that is, the exponential value of the output neuron value corresponding to class k (before softmax application) for sample j. With this notation, we can rewrite each of the addends of Cross-Entropy Loss function f (Eq. 5) as

$$\log\left(\frac{\mathtt{RC}^{\{l_j\}}}{\mathtt{RC}^{\{0\}} + \mathtt{RC}^{\{1\}}}\right) = \log\left(\frac{\mathtt{NUM}}{\mathtt{DEN}}\right),$$

where 0 and 1 are the possible classes and $\{l_j\} \in \{0, 1\}$ is the label of the sample. For simplicity, we use NUM and DEN for the numerator $\mathtt{RC}^{\{l_j\}}$ and denominator $\mathtt{RC}^{\{0\}} + \mathtt{RC}^{\{1\}}$, respectively, of the fraction inside brackets.

As indicated before, we have to compute the partial derivative of each of those addends with respect to an element w_s of $\mathcal{W}$ (a weight or a bias). Such partial derivative will have a different expression for each case C1–C4:

C1. $$\frac{\mathtt{DEN}}{\mathtt{NUM}} \frac{R_s^j\,\mathtt{NUM}\,\mathtt{DEN} - R_s^j\,\mathtt{NUM}^2}{\mathtt{DEN}^2} = R_s^j \frac{\mathtt{RC}^{\{l_j\}^c}}{\mathtt{RC}^{\{0\}} + \mathtt{RC}^{\{1\}}}.$$

C2. $$\frac{\mathtt{DEN}}{\mathtt{NUM}} \frac{(-\mathtt{NUM})\,R_s^j\,\mathtt{RC}^{\{l_j\}^c}}{\mathtt{DEN}^2} = -R_s^j \frac{\mathtt{RC}^{\{l_j\}^c}}{\mathtt{RC}^{\{0\}} + \mathtt{RC}^{\{1\}}}.$$

C3. $$\frac{\mathtt{DEN}}{\mathtt{NUM}} \frac{\mathtt{NUM}\,\mathtt{DEN} - \mathtt{NUM}^2}{\mathtt{DEN}^2} = \frac{\mathtt{RC}^{\{l_j\}^c}}{\mathtt{RC}^{\{0\}} + \mathtt{RC}^{\{1\}}}.$$

C4. $$\frac{\mathtt{DEN}}{\mathtt{NUM}} \frac{(-\mathtt{NUM})\,\mathtt{RC}^{\{l_j\}^c}}{\mathtt{DEN}^2} = -\frac{\mathtt{RC}^{\{l_j\}^c}}{\mathtt{RC}^{\{0\}} + \mathtt{RC}^{\{1\}}}.$$

In previous expressions we have used $\{l_j\}^c$ to refer to the class contrary to the label (non-correct class), and R_s^j refers to the value of the s-th neuron of the reservoir (for input sample of index j). For clarity, we develop step-by-step the computations performed in case C1 (the remaining ones are equivalent):

$$\frac{\partial \log\left(\frac{\mathtt{NUM}}{\mathtt{DEN}}\right)}{\partial w_s} = \frac{\frac{\partial\left(\frac{\mathtt{NUM}}{\mathtt{DEN}}\right)}{\partial w_s}}{\left(\frac{\mathtt{NUM}}{\mathtt{DEN}}\right)} = \frac{\mathtt{DEN}}{\mathtt{NUM}} \frac{\left(\frac{\partial \mathtt{NUM}}{\partial w_s}\right)\mathtt{DEN} - \left(\frac{\partial \mathtt{DEN}}{\partial w_s}\right)\mathtt{NUM}}{\mathtt{DEN}^2} = \frac{\mathtt{DEN}}{\mathtt{NUM}} \frac{R_s^j \mathtt{NUM}\,\mathtt{DEN} - R_s^j \mathtt{NUM}^2}{\mathtt{DEN}^2} = R_s^j \frac{\mathtt{DEN} - \mathtt{NUM}}{\mathtt{DEN}} = R_s^j \frac{\mathtt{RC}^{\{l_j\}^c}}{\mathtt{RC}^{\{0\}} + \mathtt{RC}^{\{1\}}}.$$

Notice that if we analyze the obtained results for the derivatives, the only difference between C1 and C2 formulas (that correspond to the derivative with respect to a weight) is the minus sign that only appears if the weight is not contributing to the correct class. Something equivalent occurs for C3 and C4 cases for the bias. Let us write the expression of $\nabla f(\mathcal{W})$. For this, we consider that the first M elements of the gradient correspond to the derivatives with respect to a weight w_s used to compute the value of the output neuron of class 0, the next M elements are the derivatives with respect to a weight w_s involved in the computations of the output neuron of class 1, the penultimate derivative is with respect to the bias $b^{\{0\}}$, and the last one is with respect to the bias $b^{\{1\}}$. Therefore,

the gradient is

$$\nabla f(\mathcal{W}) = \begin{pmatrix} \sum_{j=1}^{B} (-1)^{\{l_j\}} R_1^j \dfrac{\mathrm{RC}^{\{l_j\}^c}}{\mathrm{RC}^{\{0\}} + \mathrm{RC}^{\{1\}}} \\ \vdots \\ \sum_{j=1}^{B} (-1)^{\{l_j\}} R_M^j \dfrac{\mathrm{RC}^{\{l_j\}^c}}{\mathrm{RC}^{\{0\}} + \mathrm{RC}^{\{1\}}} \\ \sum_{j=1}^{B} (-1)^{\{l_j\}+1} R_1^j \dfrac{\mathrm{RC}^{\{l_j\}^c}}{\mathrm{RC}^{\{0\}} + \mathrm{RC}^{\{1\}}} \\ \vdots \\ \sum_{j=1}^{B} (-1)^{\{l_j\}+1} R_M^j \dfrac{\mathrm{RC}^{\{l_j\}^c}}{\mathrm{RC}^{\{0\}} + \mathrm{RC}^{\{1\}}} \\ \sum_{j=1}^{B} (-1)^{\{l_j\}} \dfrac{\mathrm{RC}^{\{l_j\}^c}}{\mathrm{RC}^{\{0\}} + \mathrm{RC}^{\{1\}}} \\ \sum_{j=1}^{B} (-1)^{\{l_j\}+1} \dfrac{\mathrm{RC}^{\{l_j\}^c}}{\mathrm{RC}^{\{0\}} + \mathrm{RC}^{\{1\}}} \end{pmatrix}.$$

Finally, $\nabla f(\mathcal{V}_{i-1})$ will be a constant vector obtained by substituting $\mathcal{W}$ by $\mathcal{V}_{i-1}$ in the previous expression.

With this we have provided the complete set of formulas for applying the BPGe algorithm to the binary classification problem using an RC network.

5. Experiments

In this section, we consider three different binary classification tasks to compare the performance of the algorithms on each of them: chaos detection to distinguish between regular and chaotic dynamics in the Lorenz system (Section 5.2), movement vs rest activity in the Human Activity Recognition (HAR) using smartphones dataset (Section 5.3), and even–odd classification on the MNIST dataset (Section 5.4).

5.1. Experimental setup

First of all, for each experiment, we perform a preliminary analysis. We fix the network architecture, the initialization of the weights and biases, and the training, validation and test sets, and we use SGD, RMSProp, Adam and BPGe optimizers to train the network for three different values of the learning rate ($\lambda_1 = 0.001$, $\lambda_2 = 0.0005$, $\lambda_3 = 0.0001$) during 4000 epochs. This allows us to examine the performance of each optimizer during extended training and to compare their convergence speed. In this case, we consider as evaluation metrics the loss function value (Cross-Entropy Loss with L^2-penalty) and commonly used binary classification measures in the train, validation and test datasets. Such common metrics are the *Accuracy* (computed for the train, validation and test sets), and the *Sensitivity* and *Specificity* (computed just for the test set to check if the trained network is able to detect both classes properly). Their formulas are (as in [36], we express the values in the interval $[0, 1]$):

$$\text{Accuracy} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}},$$

$$\text{Sensitivity} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}},$$

$$\text{Specificity} = \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FP}},$$

where TP and TN correspond with the number of true positive and true negative samples (the samples of each category correctly classified by the network), respectively; and FP and FN are the number of false positive and false negative samples (the samples of each category incorrectly classified by the network), respectively. For the considered experiments (chaos detection, movement vs rest activity, and even–odd classification), the positive–negative classes are chaos–regular, movement–rest and odd–even, respectively.

After this first analysis, we perform a more fair comparison between the optimizers, considering more evaluation metrics explained in [36], and using violin plots as a statistical visualization tool (they show the probability distribution and the box plot together in a clear and compact way). In this case, we consider two additional optimizers, NAG and NAdam, both variations of methods considered earlier. That is, we compare the performance of six optimizers. To carry out a fairer comparison between all the optimizers, we use a strategy based on a cross-validation type-method and a sweep functionality. In particular, to perform a cross-validation type-method, we conduct 25 trials with two datasets (one for positive class samples and one for negative class ones) formed by the samples of the train, test and validation datasets of the previous experiment (all the positive samples of the three sets are stacked to create the new positive dataset, and the same for negative samples). On each trial, we randomly select 60% of the samples of each dataset for the train dataset, 20% for the validation set, and the remaining 20% for the test set; and an initialization for the weights and biases is set at

random. We employ the sweep functionality of Weights & Biases [37] to perform an automated Bayesian optimization of the learning rate for each optimizer, using 20 runs and minimizing the validation loss as the objective. Specifically, we consider $\lambda \in [10^{-5}, 10^{-1}]$. For NAG, we also optimize the momentum parameter in the interval $[0.8, 0.99]$, as it does not have a default value in PyTorch [27], resulting in a more carefully tuned configuration compared to the other optimizers. Notice that, on each trial, the initialization of the weights and biases and the datasets used are the same for all optimizers. Moreover, the structure of the network is the same for all the trials. For this study, we consider the batch sizes for each dataset equal to the number of samples of each dataset (so, in fact, SGD corresponds to Gradient Descent) and just 1000 epochs for time purposes.

Now, the evaluation metrics considered are accuracy, sensitivity and specificity, also used in the first analysis, and *Precision*:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}},$$

where TP and FP have the meaning indicated before. Other complementary evaluation metrics also computed are the Youden's Index, the F1-score, the Cohen's kappa and the Matthews' Correlation Coefficient. The *Youden's Index* (YI) measures how well a classifier separates the two classes of the task considering its ability to detect positives and to correctly avoid false positives:

$$\text{YI} = \text{Sensitivity} + \text{Specificity} - 1 \in [0, 1].$$

The *F1-score* (F1) provides a measure of the balance between precision and sensitivity of the classifier, showing how well it identifies positive cases without producing too many false positives:

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Sensitivity}}{\text{Precision} + \text{Sensitivity}} \in [0, 1].$$

Cohen's kappa (κ) compares the performance of the classifier compared to the randomized accuracy p_e that can measure the agreement between the real and the predicted classes. This measure is defined as

$$\kappa = \frac{\text{Accuracy} - p_e}{1 - p_e} \in (-\infty, 1],$$

where

$$p_e = \frac{(\text{TP} + \text{FN})(\text{TP} + \text{FP}) + (\text{TN} + \text{FP})(\text{TN} + \text{FN})}{(\text{TP} + \text{TN} + \text{FP} + \text{FN})^2}.$$

Finally, the *Matthews' Correlation Coefficient* (MCC) gives us the correlation between the real and the predicted classifications:

$$\text{MCC} = \frac{\text{TN} \cdot \text{TP} - \text{FN} \cdot \text{FP}}{\sqrt{(\text{TP} + \text{FP})(\text{TP} + \text{FN})(\text{TN} + \text{FP})(\text{TN} + \text{FN})}} \in [-1, 1].$$

To support our claims, we report 95% confidence intervals for the accuracy and we perform a paired statistical test across the 25 trials to compare BPGe with the other two best optimizers. For the confidence intervals, we use the *Bias-Corrected and accelerated* (BCa) *bootstrap Confidence Interval* (CI) [38]. To compute it we do as follows. From the 25 accuracy values computed for each optimizer, we obtain 10,000 bootstrap resamples (each consisting of 25 accuracy values sampled with replacement). Then, the mean of each resample is computed obtaining the bootstrap distribution of the mean. The 95% confidence interval is obtained by adjusting the quantile limits to correct for bias and skewness in the bootstrap distribution. We use the *Wilcoxon signed-rank test* [36] to compare optimizers. The null hypothesis of this non-parametric test is that two paired samples come from the same distribution. In our case we will compare the accuracy of two optimizers applied to the same training data. The test statistic is $T = \min(W^+, W^-)$, where W^+ and W^- represent the sum of the positive and negative ranks (the ranks are computed from the differences of the accuracies of both selected optimizers in the 25 trials, ordered by their absolute values), respectively. As a complement, we provide an effect size computed with the *matched-pairs rank-biserial correlation coefficient* (rc) considered in [39] (where no sign is used)

$$rc = \frac{4\left|T - \frac{W^+ + W^-}{2}\right|}{n(n+1)},$$

where n is the sample size, 25 in our case.

In each optimizer, there exist different hyperparameters such as learning rate or momentum. In our study, we fix all the hyperparameters to the default values in PyTorch [27] and the learning rate is varied depending on the optimizer and the data. In the case of NAG, as no default momentum value exists, it is also considered a hyperparameter. In the case of BPGe, we fix the following parameter values: $\beta_0 = 0.99$, $\rho = 0.99$, $\eta = 0.95$ and $\mu = 0$. The study of different hyperparameter strategies to improve the optimizers is widely extended in the literature (both for gradient descent methods [40] and for proximal gradient algorithms [41]).

5.2. Dynamical classification in the Lorenz system

The first problem used to study the performance of the chosen optimizers is the Lorenz system [42], a paradigmatic problem of chaotic dynamics. Lorenz system is a classical 3-dimensional continuous dynamical system given by the following equations:

$$\dot{x} = \sigma(y - x), \quad \dot{y} = x(r - z) - y, \quad \dot{z} = xy - bz, \tag{9}$$

where the system variables are (x, y, z) and the bifurcation parameters are (σ, r, b). In particular, σ is known as Prandtl number, r is the relative Rayleigh number, and b is a positive constant. In our study, $\sigma = 10$, $r \in (0, 300]$ and $b \in \{2.4, 2.8, 8/3\}$. Our goal is to classify the dynamics of this system into regular (label 0) or chaotic (label 1). To compare the results obtained with our Neural Network with the correct behavior, we compute the Lyapunov exponents (positive first Lyapunov exponent means chaos, and regular otherwise) using the algorithm in [43].

To obtain the train, validation and test datasets for the preliminary study in the Lorenz system case, we use r-parametric lines where parameter $b = 2.4$ for the training dataset, $b = 2.8$ for the validation dataset, and $b = 8/3$ (classical line of the Lorenz system) for the test dataset. For each r-parametric line, we consider 5999 different values of r moving uniformly in the interval $(0, 300]$, so we have 5999 time series for each b-value. These time series are of length 1000 and their fixed initial conditions are $(x_0, y_0, z_0) = (1, 1, 1)$. To integrate these time series the `DOPRI5` method (Runge-Kutta integrator of order 5) is used. First of all, a transient process is performed until time $t = 100,000$ with a time step of 0.01. Later, we integrate $100,001$ more time units with a time step of 0.001 to compute the Lyapunov exponents (used to determine the behavior of each time series). Taking 1 out of every 100 of the last $100,000$ computed points we obtain the time series of length 1000 that we use as input in our Neural Network. Time series values are normalized into range $[0, 1]$.

The samples obtained from the three r-parametric lines are screened to prevent repeated time series inside or through different datasets (and therefore, to ensure that the network learns correctly). First of all, we delete such repeated samples, considering repeated samples as those whose difference in infinity norm is less than 10^{-4}. Next, we separate time series between regular and chaotic. From these remaining samples, we take randomly 2260 of each class for the training dataset (4520 in total from the one-parameter line with $b = 2.4$), and 1500 of each class for the validation dataset (3000 in total from the one-parameter line with $b = 2.8$). Test dataset consists of 5980 samples from one-parameter line with $b = 8/3$. The batch sizes are fixed to 128, 100, and 230 for training, validation, and test sets, respectively, dropping the last incomplete batch.

In Table 2, the loss and accuracy values for the final network (that is, the network of the best epoch, which is the epoch with the lowest loss value for the validation dataset during training) in the train, validation and test datasets using the SGD, RMSProp, Adam and BPGe optimizers are gathered together. For each optimizer the results are shown for three different values of the learning rate: $\lambda_1 = 0.001$, $\lambda_2 = 0.0005$ and $\lambda_3 = 0.0001$. We have studied the accuracy of the test dataset, but we have to ensure that the network has learned correctly, that is, if it is able to classify correctly each type of behavior (regular and chaotic). For this purpose, in the last two columns of this table, we study the specificity and sensitivity (that is, the quantity of regular samples that are correctly detected with respect to the total number of regular samples, and analogously for the chaotic time series) for the test set.

For the SGD and Adam optimizers, the loss increases and accuracy decreases as the learning rate decreases. Hence, the best results (lowest test loss and highest test accuracy) occur with $\lambda_1 = 0.001$. However, in the remaining two optimizers (RMSProp and BPGe), there is no clear trend in behavior. In fact, for RMSProp, in the training set, loss value increases and accuracy value decreases as the learning rate decreases, while this is not occurring for the other two datasets where the best values (lowest loss and largest accuracy) are given for $\lambda_2 = 0.0005$. In the case of BPGe, it is not easy to define a trend, but probably the appropriate value for the learning rate is $\lambda_3 = 0.0001$.

Notice that the accuracy values for both behaviors are quite similar in all cases (the difference between them is always less than 0.08). Therefore, all the optimizers (for all the learning rate values) are able to properly distinguish between regular and chaotic samples.

Table 2
Loss and accuracy for training, validation and test sets for the four optimizers (SGD, RMSProp, Adam, BPGe) and three different learning rate values ($\lambda_1 = 0.001$, $\lambda_2 = 0.0005$, $\lambda_3 = 0.0001$) for the dynamical classification in the Lorenz system. Specificity and sensitivity are also indicated for test dataset.

		Train		Validation		Test			
		Loss	Accuracy	Loss	Accuracy	Loss	Accuracy	Specificity	Sensitivity
SGD	$\lambda_1 = 0.001$	0.255	0.902	0.225	0.925	0.242	0.917	0.923	0.911
	$\lambda_2 = 0.0005$	0.283	0.889	0.250	0.916	0.266	0.905	0.913	0.899
	$\lambda_3 = 0.0001$	0.361	0.857	0.329	0.885	0.341	0.875	0.888	0.864
RMSProp	$\lambda_1 = 0.001$	0.152	0.940	0.165	0.941	0.186	0.938	0.977	0.906
	$\lambda_2 = 0.0005$	0.162	0.936	0.162	0.943	0.182	0.940	0.970	0.916
	$\lambda_3 = 0.0001$	0.202	0.925	0.183	0.938	0.202	0.935	0.944	0.928
Adam	$\lambda_1 = 0.001$	0.149	0.940	0.135	0.948	0.148	0.947	0.961	0.936
	$\lambda_2 = 0.0005$	0.160	0.937	0.143	0.947	0.158	0.946	0.952	0.942
	$\lambda_3 = 0.0001$	0.201	0.926	0.180	0.936	0.199	0.934	0.937	0.931
BPGe	$\lambda_1 = 0.001$	0.108	0.967	0.178	0.954	0.212	0.949	0.971	0.931
	$\lambda_2 = 0.0005$	0.088	0.968	0.139	0.949	0.160	0.947	0.979	0.920
	$\lambda_3 = 0.0001$	0.120	0.954	0.127	0.951	0.139	0.949	0.972	0.930

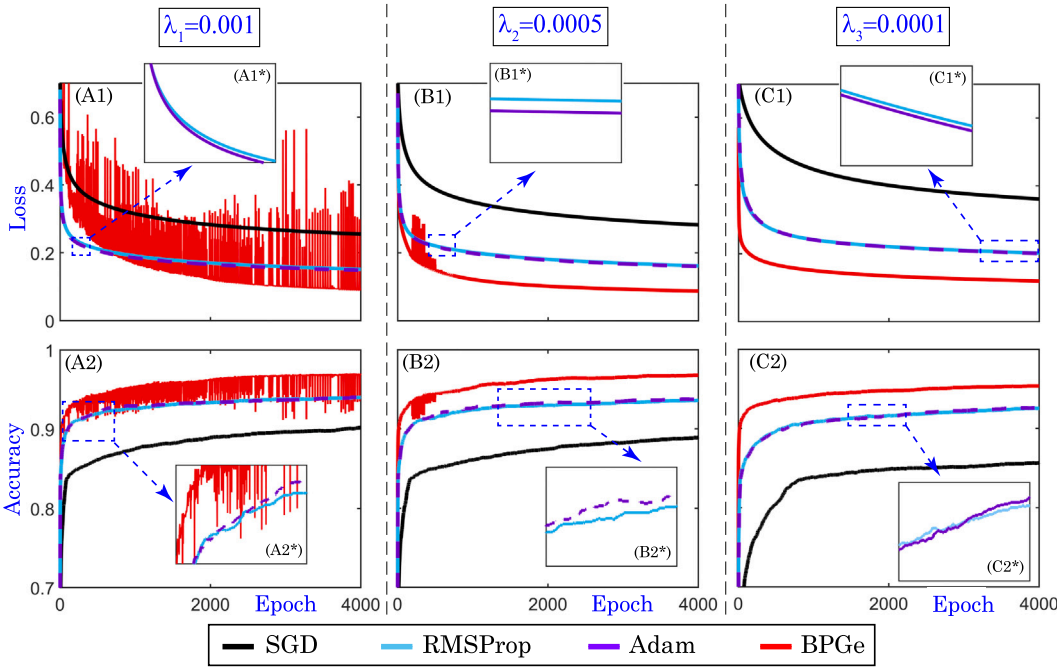

Fig. 3. Study of the loss and accuracy evolution of the training dataset along the epochs for the dynamical classification of the Lorenz system. The four optimizers (SGD, RMSProp, Adam and BPGe) are compared. (A1)-(A1*)-(A2)-(A2*) Evolution for the learning rate $\lambda_1 = 0.001$. (B1)-(B1*)-(B2)-(B2*) Evolution for the learning rate $\lambda_2 = 0.0005$. (C1)-(C1*)-(C2)-(C2*) Evolution for the learning rate $\lambda_3 = 0.0001$.

Comparing the results in the table across all the optimizers we can conclude that the worst results are obtained with the SGD optimizer as for the test dataset the accuracy is for the three cases less than 0.92 and for the other optimizers it is always greater than 0.93 (for all the learning rate values). Moreover, the SGD optimizer gives the largest values for the loss. The largest accuracy values for all the studied learning rates in the test dataset are those obtained with the BPGe optimizer (comparing the value of the test accuracy for each value of the learning rate). However, if we compare the accuracy for regular and chaotic samples for the test dataset, we can see that the performance of Adam and BPGe is quite similar (for example, for $\lambda_2 = 0.0005$, the accuracy for regular samples is larger for BPGe than for Adam, but the accuracy for chaotic behavior is greater for Adam). Notice that the lowest loss value for the test dataset is obtained with $\lambda_3 = 0.0001$ for the BPGe optimizer, which corresponds with the largest test accuracy for all the optimizers and learning rates.

After studying the loss and accuracy for the best epoch, we study the evolution of the loss and accuracy values along the epochs (for training dataset) in Fig. 3. In panels (A1)-(A2), (B1)-(B2) and (C1)-(C2), these evolutions are shown for the learning rates $\lambda_1 = 0.001$, $\lambda_2 = 0.0005$ and $\lambda_3 = 0.0001$, respectively. In each plot, the four optimizers are compared (SGD in black, RMSProp in light blue, Adam in purple, and BPGe in red).

Notice that BPGe optimizer (in red) is the noisiest optimizer for $\lambda_1 = 0.001$ (see panels (A1)-(A2)). If the value of the learning rate is decreased up to $\lambda_2 = 0.0005$, only a small part of the evolution is noisy (see panels (B1)-(B2)). When the value of the learning rate is very small ($\lambda_3 = 0.0001$), the evolution of the loss and accuracy of the BPGe is not noisy anymore (this learning rate gives us the best test accuracy as can be seen in Table 2). This could be related with the restriction $0 < \lambda_i \leq 1/L$ (for BPGe) indicated in [13], with L being the constant of the Lipschitz gradient continuity condition. We conjecture that for values of the learning rate smaller than $\lambda_2 = 0.0005$ (and not for values greater than or equal to λ_1) this condition is fulfilled and therefore the noise disappears.

For all the learning rate values, it can be seen clearly that the BPGe optimizer (in red) achieves the lowest loss values and the largest accuracies with a considerable difference. For the SGD optimizer (in black), the loss seems to be always larger and the accuracy lower than the other three optimizers. The results obtained with the RMSProp (in light blue) and Adam (in purple) are quite similar and indistinguishable at a glance, so we provide some magnifications. A zoom of panels (A1)-(B1)-(C1) is provided in subpanels (A1*)-(B1*)-(C1*), where it can be seen that loss values of both optimizers are really close but larger for the RMSProp algorithm. The accuracy for RMSProp and Adam optimizers is quite similar, sometimes one is larger than the other and other times the opposite as can be seen in subpanels (A2*)-(B2*)-(C2*) which are zooms of panels (A2)-(B2)-(C2).

In summary, for the learning rate values studied and with the configuration considered, on the one hand, the best performance, that is, the lowest loss and the largest accuracy, is obtained with the BPGe optimizer (see the red line in all panels of Fig. 3). In particular, the best results are the ones corresponding to $\lambda_3 = 0.0001$ (see Table 2). On the other hand, the worst performance, that is, the largest loss and the lowest accuracy, is achieved with the SGD optimizer (see the black line in all panels of Fig. 3 and Table 2). The RMSProp and Adam optimizers seem to give very similar results (better than those of SGD and worse than those of BPGe optimizer). The conclusions obtained with Fig. 3 coincide with those derived from the study of Table 2.

With results in Table 2 and Fig. 3, we conclude that BPGe seems to perform really well. However, it is also interesting to analyze in which epoch the network with the best results (lower validation loss during training process) is obtained and how much time is needed by each optimizer to train the network. Sometimes better results may not be entirely advantageous if the time required is much longer.

For the SGD, RMSProp, and Adam optimizers, the best results occur at epoch 4000, which is the maximum set for the process. This suggests further learning could occur with more epochs, but increased training time may not yield significant accuracy improvements, making additional epochs less worthwhile. With the maximum of 4000 epochs, the wall time during training process for each of the three optimizers is approximately 17 minutes. Focusing on the BPGe results, for $\lambda_1 = 0.001$ the best epoch is 3065, for $\lambda_2 = 0.0005$ it is 3893, and for $\lambda_3 = 0.0001$ it is 4000 (last epoch as happens for the other optimizers). With the maximum of 4000 epochs, the wall time during training process for BPGe optimizer is around 19 minutes. Notice that BPGe takes a bit longer to finish training process when 4000 epochs are considered, but for some learning rates (λ_1, λ_2) a lower maximum number of epochs (and consequently less wall time) is needed to achieve the best epoch. For the largest learning rate value in BPGe, the best epoch is 4000, but although time is a bit increased respect to other optimizers, results are better. We want to remark that in the wall time is only included the time needed to train the network once h_T (last state of the reservoir, see Eq. (4)) has been obtained for all the samples.

The total number of epochs is a hyperparameter that could have been set to a different value. A natural question arises: At which epoch has the training accuracy surpassed a certain value? As in general we have obtained that the networks achieved the best model for the last epoch, this new analysis will allow us to understand which optimizers provide a faster learning and need less epochs to obtain a good performance. Here, we have checked that in addition to satisfying the accuracy indicated, the validation accuracy is not far away from this value and then it is not suffering from overfitting. In Table 3, we have summarized such analysis. In particular, we indicate, for each optimizer and learning rate, the first epoch in which training accuracy has surpassed 0.80, 0.85, 0.90 and 0.95. Notice that the unique optimizer that surpasses the 0.95 accuracy for training is BPGe. The SGD algorithm only achieves an accuracy greater than or equal to 0.90 for the higher value of learning rate ($\lambda_1 = 0.001$) that we established as the best result for this optimizer according to Table 2. Moreover, the number of epochs needed by SGD to exceed whatever accuracy value is considerably larger than that for the other optimizers. The RMSProp and Adam optimizers have quite similar results as we have already concluded analyzing the training loss and training accuracy evolutions in Fig. 3. Finally, it is remarkable that BPGe, for whatever value of the learning rate, surpasses a value of 0.90 in the accuracy just after less than 20 epochs.

To have an overview of the time needed to achieve a certain train performance, we measure the approximate time per epoch (in seconds) needed by each optimizer and we use the results in Table 3 to compute the time used to achieve (for the first time) an accuracy value greater than or equal to 0.90 in training set during the training process. For convention, for each optimizer, we choose the learning rate value (λ_1, λ_2 or λ_3) that provides the lowest test loss (see Table 2). These results have been summarized in Table 4. Therefore, although BPGe needs more time for training process (19 minutes against the 17 minutes needed by the other optimizers), it uses a small portion of time to achieve a good performance and the remaining process is devoted to fine-tune the results to obtain a better performance. So, setting a lower number of epochs, wall time would be considerably reduced and performance would not be

Table 3
First epoch in the training process of the dynamical classification in the Lorenz system in which the accuracy of the training set is equal to or greater than 0.80, 0.85, 0.90 and 0.95 for the four optimizers and the considered learning rate values.

Accuracy		≥ 0.80	≥ 0.85	≥ 0.90	≥ 0.95
SGD	$\lambda_1 = 0.001$	44	245	3750	–
	$\lambda_2 = 0.0005$	89	461	–	–
	$\lambda_3 = 0.0001$	445	2248	–	–
RMSProp	$\lambda_1 = 0.001$	3	13	101	–
	$\lambda_2 = 0.0005$	4	15	145	–
	$\lambda_3 = 0.0001$	12	60	578	–
Adam	$\lambda_1 = 0.001$	3	11	103	–
	$\lambda_2 = 0.0005$	4	17	154	–
	$\lambda_3 = 0.0001$	16	62	638	–
BPGe	$\lambda_1 = 0.001$	2	4	17	712
	$\lambda_2 = 0.0005$	2	3	14	877
	$\lambda_3 = 0.0001$	2	3	17	2501

Table 4
Wall time study (in seconds) for the binary classifications (regular vs chaotic) in the Lorenz system. For each optimizer, the approximate time per epoch and the time needed to reach for the first time a 0.90 in train accuracy are provided.

Time (s)	SGD	RMSProp	Adam	BPGe
Per Epoch	0.256	0.261	0.259	0.286
To ≥ 0.90	960	37.555	26.883	4.862

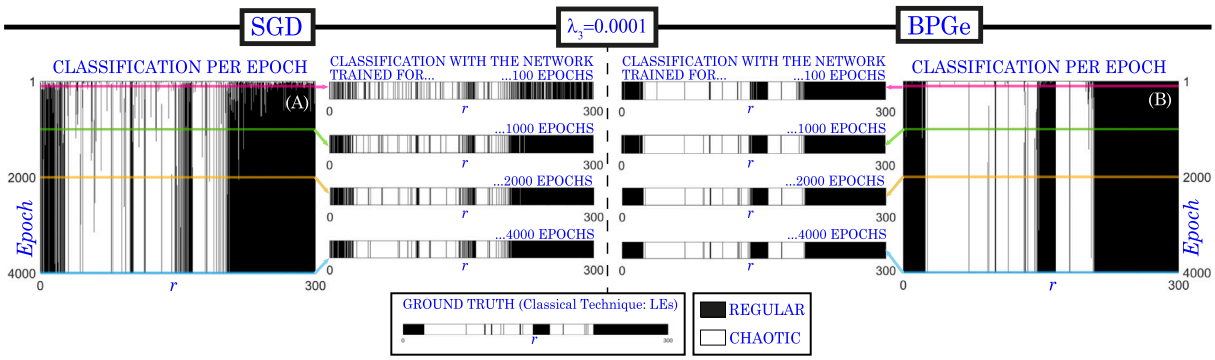

Fig. 4. Study of the dynamical classification in an r-parametric line of the Lorenz system for SGD and BPGe for $\lambda_3 = 0.0001$ in different epochs (highlighting the results after 100; 1000; 2000; and 4000 epochs during training process). This study is expanded for the four optimizers and three considered learning rates in Fig. 5.

considerably affected. To support this result, we check the performance of the network obtained at each epoch using an r-parametric line of the Lorenz system.

Let us consider the r-parametric line with $b = 8/3$ and $\sigma = 10$ of the Lorenz system. Notice that it corresponds to the line used to create the test set, but now we use it with the 5999 values of parameter r. During training, we save the values of the trainable parameters (W_{out} and b_{out}) of the network obtained at each epoch. Note that as we have set to 4000 the maximum number of epochs, we will have 4000 different networks for each optimizer and each learning rate. With each network, we perform a dynamical classification in the aforementioned r-parametric line. This will allow us to have an idea of how the results would be at each moment of the training process. Fig. 4 shows the results for SGD and BPGe with $\lambda_3 = 0.0001$. This figure serves as an introductory explanation of Fig. 5 with all the complete information. In this introductory figure, white color means chaotic behavior and black color corresponds to regular one. Panels (A) and (B) correspond with the dynamical classification across the 4000 epochs for SGD and BPGe, respectively. We have drawn the dynamical classification that we would obtain if we train the network during 100; 1000; 2000; and 4000 epochs. We can compare these results with the ground truth obtained with the classical technique of Lyapunov exponents (LEs). Notice that SGD is not able to classify correctly the behavior after 100; 1000; and even 2000 epochs as the plots are quite noisy. After 4000 epochs the results are more similar to the ground truth although there are some misclassified behaviors. In the case of BPGe we can observe that almost from the beginning (after 100 epochs) the different chaotic and regular zones are distinguished. The results after 1000; 2000; and 4000 epochs are quite similar to the ground truth results, only highlighting certain samples that end up being correctly classified as training progresses. So, this type of figures allows us to graphically analyze how the network is adjusting the results during training process. In Fig. 5, we have the results across epochs for the four considered optimizers and the three chosen learning rates. In particular, in panels (ij) with $i \in \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\}$ and $j \in \{1, 2, 3\}$ we have the classification obtained with the network at each training epoch, that is, on the top we have the classification results obtained with the network of the first epoch of the training process, at the bottom those provided by the network of the last epoch (epoch 4000), and in between the results of the networks for the remaining epochs (panels (A3) and (D3) are the same plots than panels (A) and (B) of Fig. 4). Moreover, for each optimizer and learning rate we provide the dynamical classification for the best epoch (see panels $(ij*)$). The ground truth, that is, the correct classification performed by the classical technique of Lyapunov exponents (LEs) is given at the top left of the figure. As in Fig. 4, black color is used for regular behavior and white for chaotic. Moreover, in panels (ije) and $(ije*)$ we provide the error plots of panels (ij) and $(ij*)$, respectively. In red color we have the false regular (FR) detections (chaotic samples detected as regular), in blue the false chaotic (FC) detections (regular samples classified as chaotic), and in beige the correct detections (true regular, TR; and true chaotic, TC).

In panels of SGD (Aj)-(Aje), we can observe a noisy classification for any epoch (especially in panels (A3)-(A3e) that correspond with the worst results according to previous analyses). In panels of RMSProp and Adam optimizers (Bj)-(Bje) and (Cj)-(Cje), respectively, we have less noisy classification in non-boundary regions, but it seems that they are more indecisive than BPGe (panels (Dj)-(Dje)). In BPGe panels (Dj)-(Dje), it is observed that in the first epochs the optimizer already correctly determines the non-boundary zones and (almost all) the regular windows in the large chaotic regions, and it uses the remaining epochs to fix the behavior in the boundaries. These conclusions match with the results of Table 3 as an accuracy greater than or equal to 0.90 (for training set) is obtained after less than 20 epochs.

If we look at the panels $(ij*)$-$(ije*)$ for $i \in \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\}$ and $j \in \{1, 2, 3\}$, we can compare the classification results. It can be seen clearly that the worst performance is achieved by SGD ($i = \mathrm{A}$) as in the left regular region (in black in the ground truth) we can observe stripes corresponding to false chaotic detections. Moreover, we can see that the boundary zones (in the middle of the panel) are not properly defined. In the panels of RMSProp ($i = \mathrm{B}$), Adam ($i = \mathrm{C}$) and BPGe ($i = \mathrm{D}$) the results seem to be acceptable and the regions are better defined.

We would like to clarify that if we compare the best epoch results for all the optimizers and learning rates with the ground truth, one part of the rightmost chaotic region is missing. In this area there are transient chaotic dynamics [44]. We have seen in a previous work [45] that such behavior is not easily detected by all DL architectures. However, this is out of the scope of this paper as the objective is to compare optimizers under the same conditions.

Now, we perform a more detailed and fairer analysis comparing the six optimizers (SGD, RMSProp, Adam, NAG, NAdam and BPGe), using the cross-validation strategy and the sweep functionality for the learning rate value (more details explained in the

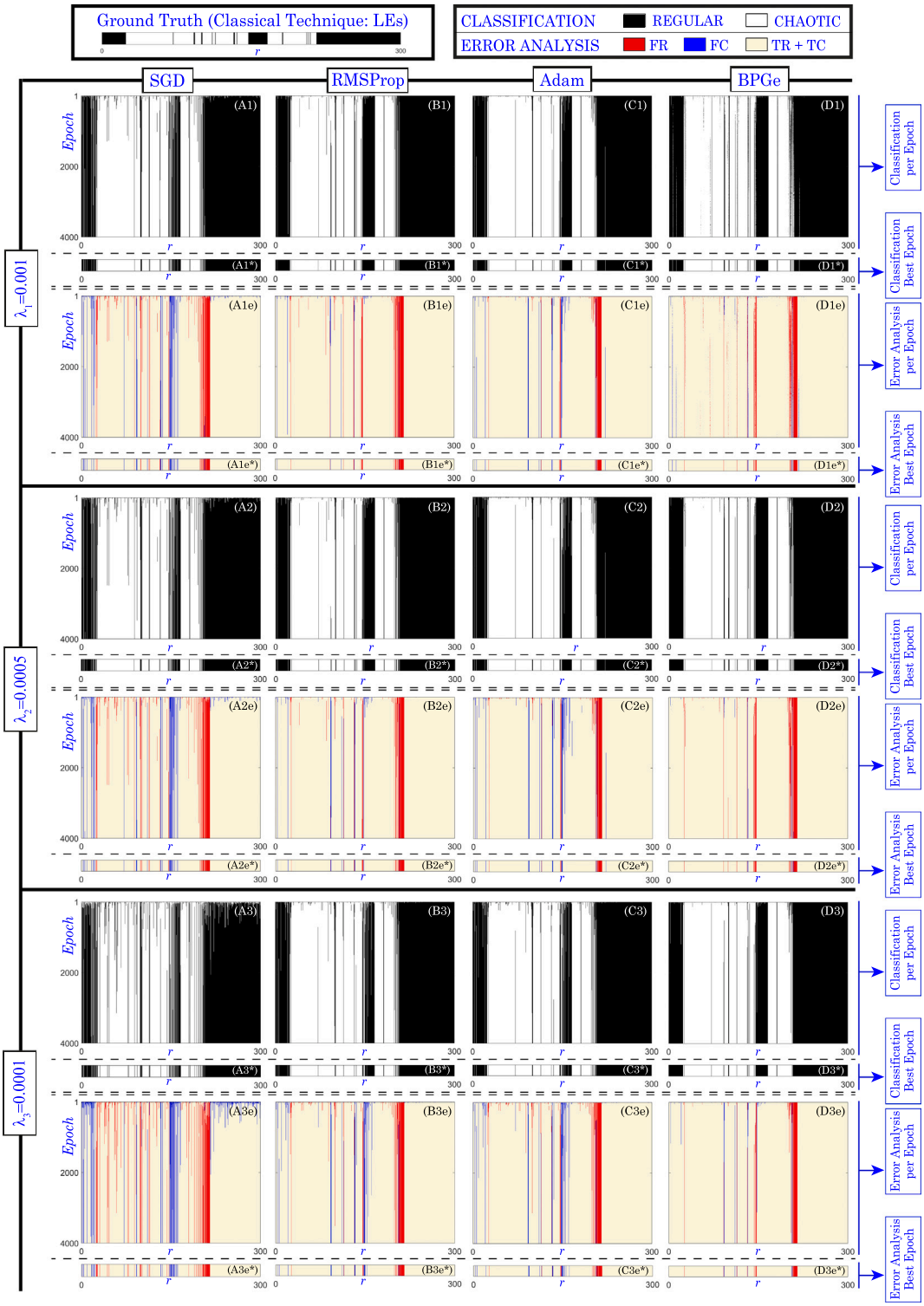

Fig. 5. Study of the dynamical classification in an r-parametric line of the Lorenz system for all the optimizers and learning rates. In panels (ij) with $i \in \{A, B, C, D\}$ and $j \in \{1, 2, 3\}$, we have represented the detection (black for regular, and white for chaotic behavior) performed with each of the networks obtained during training (that is, the network at each epoch). In panels $(ij*)$ we have the final classification, i.e., the classification for the best epoch (obtained with early stopping technique). Panels (ije) correspond to the error plots of panels (ij). In red we have the false regular (FR) detections, in blue the false chaotic (FC) ones, and in beige the correct detections (true regular, TR; and true chaotic, TC). In panels $(ije*)$ we have the error plots corresponding to $(ij*)$. In the top left panel we have the ground truth, that is, the detection obtained with the classical technique of Lyapunov exponents (LEs).

Table 5
Lorenz test study. Comparison of the performance of SGD, RMSProp, Adam, NAG, NAdam and BPGe optimizers for the 25 trials with different weights-biases initializations and different sets for training, validation and test, and with the learning rate λ that gives the best model (less validation loss) for each optimizer in the range $\lambda \in [10^{-5}, 10^{-1}]$ (for NAG, the momentum is also tuned in $[0.8, 0.99]$). The studied results are the evaluation metrics (expressed as $mean \pm std$) and the confidence intervals explained in Section 5.1.

	SGD	RMSProp	Adam	NAG	NAdam	BPGe
Accuracy	0.884 ± 0.006	0.900 ± 0.007	0.930 ± 0.004	0.939 ± 0.005	0.912 ± 0.006	0.925 ± 0.005
BCa CI for Accuracy	$[0.881, 0.886]$	$[0.897, 0.903]$	$[0.929, 0.932]$	$[0.937, 0.941]$	$[0.910, 0.914]$	$[0.923, 0.927]$
Precision	0.879 ± 0.010	0.895 ± 0.020	0.932 ± 0.007	0.941 ± 0.008	0.910 ± 0.011	0.925 ± 0.010
Sensitivity	0.890 ± 0.011	0.907 ± 0.014	0.929 ± 0.007	0.937 ± 0.006	0.916 ± 0.006	0.925 ± 0.006
Specificity	0.877 ± 0.012	0.893 ± 0.024	0.932 ± 0.008	0.941 ± 0.009	0.909 ± 0.013	0.925 ± 0.011
Youden's Index	0.767 ± 0.012	0.800 ± 0.015	0.861 ± 0.008	0.878 ± 0.011	0.825 ± 0.011	0.850 ± 0.010
F1-score	0.884 ± 0.006	0.901 ± 0.006	0.930 ± 0.004	0.939 ± 0.005	0.913 ± 0.005	0.925 ± 0.005
Cohen's kappa	0.767 ± 0.012	0.800 ± 0.015	0.861 ± 0.008	0.878 ± 0.011	0.825 ± 0.011	0.850 ± 0.010
MCC	0.767 ± 0.012	0.801 ± 0.014	0.861 ± 0.008	0.878 ± 0.011	0.825 ± 0.011	0.850 ± 0.010

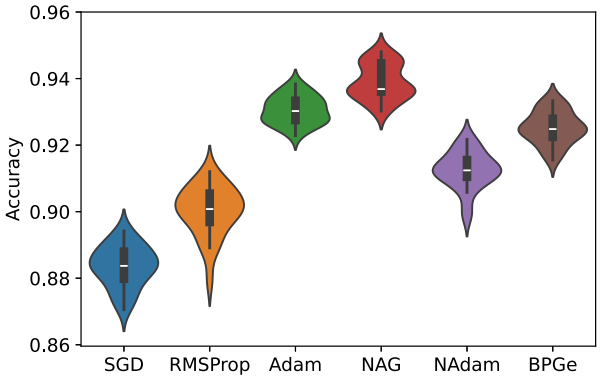

Fig. 6. Statistical violin plots to visualize the test accuracy distribution of the different optimizers in the Lorenz test.

previous Section 5.1). We create two datasets, one for chaotic samples and other for regular ones, with 6450 samples each one. The results obtained with the 25 trials are analyzed and indicated in Table 5 using the mean and standard deviation (for statistics inference) through the trials. From the computed evaluation metrics [36] and the BCa bootstrap CI described in Section 5.1, it follows that NAG, Adam, and BPGe achieved the highest accuracy, in that order, while SGD yielded the lowest performance. As indicated before, in the case of NAG optimizer the momentum is also tuned, which can lead to slightly better results. In general, as Cohen's kappa coefficient is close to 1 and Matthews' Correlation Coefficient is close to 1 too, we can conclude that the classification is better than random and is close to the perfect match. As expected, the optimizers with values closer to 1 are NAG, Adam and BPGe. In this case, as we consider chaotic as positive class, precision indicates the portion of chaotic detections that are correct, sensitivity corresponds to the portion of correctly classified chaotic samples, and specificity provides information about correctly classified regular samples. A comparison of sensitivity and specificity shows that these measures are well balanced, particularly for the BPGe optimizer. This indicates that, during training, all models learn to properly distinguish between the two dynamical regimes. This is supported by Youden's Index (capacity of discrimination) and by F1-score (degree of efficiency). To sum up, the best results are obtained with NAG, slightly followed by Adam and BPGe, and the most balanced one between classes is BPGe optimizer.

In Fig. 6, we represent violin plots to visualize the distribution for the test accuracy of the different optimizers. This visualization allows us to study the median, the interquartile range and the underlying density structure. Notice that, as observed in Table 5, the highest accuracy corresponds to NAG optimizer, although Adam and BPGe are very close to it. The box plot (inside the violin representation) of NAG optimizer shows us that the median is in the lower part of it which means that the distribution is more asymmetric than the ones of the other best optimizers (Adam and BPGe) where the median is more centered inside the box plot.

As a final step, we compare the performance of Adam and NAG optimizers versus the performance of BPGe optimizer using the Wilcoxon signed-rank test. As the p-value when comparing Adam vs BPGe and NAG vs BPGe is on the order of 10^{-8} in both cases (with effect size 1), there is a significant statistical difference between BPGe and the other optimizers, although all three provide good results.

5.3. Classification in the Human Activity Recognition using smartphones dataset

Human Activity Recognition (HAR) using smartphones dataset [46] (database available at [47]) consists of recordings from 30 subjects (aged between 19 and 48 years) performing six different Activities of Daily Living (ADL): walking, walking upstairs, walking downstairs, sitting, standing, and laying down. The 3-axial linear acceleration and 3-axial angular velocity are recorded using a waist-mounted smartphone with inertial sensors at a constant rate of 50Hz. The data obtained from the experiments is preprocessed and,

Table 6
Loss and accuracy for training, validation and test sets for the four optimizers (SGD, RMSProp, Adam, BPGe) and three different learning rate values ($\lambda_1 = 0.001$, $\lambda_2 = 0.0005$, $\lambda_3 = 0.0001$) for the classification between movement and rest in the HAR using smartphones database. Specificity and sensitivity are also indicated for test dataset.

		Train		Validation		Test			
		Loss	Accuracy	Loss	Accuracy	Loss	Accuracy	Specificity	Sensitivity
SGD	$\lambda_1 = 0.001$	0.453	0.850	0.458	0.848	0.452	0.851	0.913	0.782
	$\lambda_2 = 0.0005$	0.505	0.830	0.510	0.828	0.501	0.830	0.905	0.745
	$\lambda_3 = 0.0001$	0.595	0.778	0.598	0.769	0.596	0.775	0.892	0.644
RMSProp	$\lambda_1 = 0.001$	0.157	0.960	0.177	0.949	0.194	0.946	0.992	0.894
	$\lambda_2 = 0.0005$	0.181	0.950	0.196	0.944	0.212	0.942	0.988	0.890
	$\lambda_3 = 0.0001$	0.270	0.916	0.276	0.915	0.287	0.906	0.952	0.854
Adam	$\lambda_1 = 0.001$	0.156	0.959	0.176	0.949	0.193	0.946	0.992	0.895
	$\lambda_2 = 0.0005$	0.180	0.950	0.195	0.944	0.212	0.943	0.988	0.891
	$\lambda_3 = 0.0001$	0.270	0.916	0.276	0.914	0.287	0.906	0.952	0.854
BPGe	$\lambda_1 = 0.001$	0.108	0.973	0.147	0.965	0.169	0.954	0.993	0.911
	$\lambda_2 = 0.0005$	0.114	0.971	0.148	0.963	0.169	0.954	0.994	0.909
	$\lambda_3 = 0.0001$	0.150	0.961	0.172	0.949	0.188	0.948	0.993	0.897

for each record, 3-axial total acceleration, 3-axial estimated body acceleration, 3-axial angular velocity and a vector with 561 features are provided. The experimental data [47] is provided as training (70% of total samples) and test set (30%).

We work with binary classification tasks with time series as input. Therefore, from the HAR using smartphones dataset we consider the 3-axial estimated body acceleration, measured as a g-force, consisting in 3-dimensional time series of length 128. Moreover, we divide into two classes: rest (it includes sitting, standing and laying down) and movement (it includes walking, walking upstairs and walking downstairs). Instead of using just two datasets (train and test) as provided by this HAR dataset, we use three (train, validation and test). For this reason, for the first analysis, we use the original test set of [47] without modifications (it includes 2947 time series; 1560 labeled as rest and 1387 as movement), but we use the original train set provided in [47] to create a train and validation set for our study. In particular, the original training set has 7352 samples; 4067 correspond to rest class (1286 of sitting; 1374 of standing; and 1407 of laying down); and 3285 to movement (1226 of walking; 1073 of walking upstairs; and 986 of walking downstairs). We randomly choose 686 samples of each ADL for training, and 300 for validation; obtaining a training dataset with 4116 time series, and a validation set of 1800. For the training, validation and test sets, the batch size is 128, 100 and 421, respectively.

In Table 6, the results of the loss and accuracy of the training, validation and test datasets are shown for four optimizers (SGD, RMSProp, Adam and BPGe) using three different learning rate values ($\lambda_1 = 0.001$, $\lambda_2 = 0.0005$ and $\lambda_3 = 0.0001$). We have also computed the sensitivity and specificity for test set to ensure that the network has learned to classify properly both types of samples (rest and movement).

Notice that the loss of the training set and the one of the test (the same for the accuracy) are very similar in all cases, therefore, we can consider that the network has not suffered overfitting in any case. It is clear that the worst optimizer is the SGD (as it occurs for the Lorenz case): It has the largest loss and the lowest accuracy for all datasets independently of the value of the learning rate. Moreover, it is remarkable that the sensitivity and specificity in the test dataset is very different for this optimizer. It seems that the characteristics of movement class have not been properly learned and the corresponding accuracy is really low (especially for λ_3 as it is smaller than 0.65). RMSProp and Adam optimizers give similar results. Both performing quite well, with worsening results as the value of the learning rate is reduced. BPGe is the optimizer that gives the best results. It is remarkable that it provides the lowest loss and largest accuracy values for all learning rate values and datasets. Moreover, it is the unique optimizer in which an accuracy greater than 0.90 is obtained for the movement class (see results for λ_1 and λ_2). The learning rate that seems to perform the best is $\lambda_1 = 0.001$.

In Table 7, the epoch in which the accuracy of the training dataset (during the training process) is greater than or equal to 0.80, 0.85, 0.90 and 0.95 for the first time is indicated for the four optimizers and the three learning rate values. Notice that in the case of the SGD, most of the boxes of the table are filled with "−" as the network does not achieve an accuracy equal to or greater than 0.85 for $\lambda_1 = 0.001$ and $\lambda_2 = 0.0005$, and equal to or greater than 0.80 for $\lambda_3 = 0.0001$. This fact is expected due to the results of Table 6. For the remaining three optimizers, the accuracy equal to or greater than 0.90 is achieved before the half of the total number of epochs (4000), so good results could be obtained in half the time. Notice that, in the case of the BPGe (for all the learning rate values), an accuracy equal to or greater than 0.95 is achieved in less than 1450 epochs. Moreover, the best results (lower loss and greater accuracy for the test set, see Table 6) are obtained with $\lambda_1 = 0.001$ for the BPGe and an accuracy equal to or greater than 0.95 is obtained in epoch 120. This would allow us to obtain remarkable results in few epochs and therefore in a very short time.

The time needed to train the network (time to train during 4000 epochs) for this task is around 15 minutes for SGD optimizer, 16 minutes for RMSProp and Adam, and 17 minutes for BPGe. To have an overview of the time needed in this classification task to achieve a certain train performance, we measure the approximate time per epoch (in seconds) used by each optimizer and we utilize the results in Table 7 to compute the time needed to achieve (for the first time) an accuracy value greater than or equal to 0.90 in training set during the training process. We choose the learning rate value (λ_1, λ_2 or λ_3) that provides the lowest test loss for each optimizer (see Table 6). These results have been summarized in Table 8. Notice that, although BPGe sometimes requires a few more minutes than the other optimizers to compute 4000 epochs, it achieves the same or better accuracy in fewer epochs, potentially reducing training time.

Table 7
First epoch in the training process of the classification between movement and rest of the HAR using smartphones database in which the accuracy of the training set is equal to or greater than 0.80, 0.85, 0.90 and 0.95 for the four optimizers and the considered learning rates.

Accuracy		≥ 0.80	≥ 0.85	≥ 0.90	≥ 0.95
SGD	$\lambda_1 = 0.001$	761	–	–	–
	$\lambda_2 = 0.0005$	1522	–	–	–
	$\lambda_3 = 0.0001$	–	–	–	–
RMSProp	$\lambda_1 = 0.001$	6	33	177	2006
	$\lambda_2 = 0.0005$	13	67	352	3981
	$\lambda_3 = 0.0001$	74	351	1634	–
Adam	$\lambda_1 = 0.001$	10	41	170	1996
	$\lambda_2 = 0.0005$	19	77	345	3904
	$\lambda_3 = 0.0001$	86	362	1729	–
BPGe	$\lambda_1 = 0.001$	2	4	12	120
	$\lambda_2 = 0.0005$	2	6	24	287
	$\lambda_3 = 0.0001$	5	20	106	1420

Table 8
Wall time study (in seconds) for the binary classifications (rest vs movement) in the HAR dataset. For each optimizer, the approximate time per epoch and the time needed to reach for the first time a 0.90 in train accuracy are provided.

Time (s)	SGD	RMSProp	Adam	BPGe
Per Epoch	0.231	0.235	0.236	0.259
To ≥ 0.90	–	41.595	40.120	3.108

Table 9
HAR test study. Comparison of the performance of SGD, RMSProp, Adam, NAG, NAdam and BPGe optimizers for the 25 trials with different weights-biases initializations and different sets for training, validation and test, and with the learning rate λ that provides the best model (less validation loss) for each optimizer in the range $\lambda \in [10^{-5}, 10^{-1}]$ (for NAG, the momentum is also tuned in $[0.8, 0.99]$). The studied results are the evaluation metrics (expressed as $mean \pm std$) and the confidence intervals explained in Section 5.1.

	SGD	RMSProp	Adam	NAG	NAdam	BPGe
Accuracy	0.842 ± 0.010	0.940 ± 0.005	0.954 ± 0.005	0.940 ± 0.004	0.947 ± 0.004	0.961 ± 0.004
BCa CI for Accuracy	[0.838, 0.846]	[0.938, 0.942]	[0.952, 0.956]	[0.938, 0.942]	[0.945, 0.948]	[0.959, 0.962]
Precision	0.893 ± 0.009	0.975 ± 0.008	0.987 ± 0.006	0.975 ± 0.006	0.980 ± 0.006	0.989 ± 0.004
Sensitivity	0.778 ± 0.020	0.904 ± 0.008	0.920 ± 0.009	0.902 ± 0.008	0.912 ± 0.008	0.932 ± 0.007
Specificity	0.906 ± 0.009	0.977 ± 0.007	0.988 ± 0.005	0.977 ± 0.006	0.981 ± 0.006	0.990 ± 0.004
Youden's Index	0.685 ± 0.020	0.881 ± 0.011	0.908 ± 0.010	0.880 ± 0.008	0.894 ± 0.009	0.922 ± 0.007
F1-score	0.831 ± 0.012	0.938 ± 0.006	0.952 ± 0.005	0.938 ± 0.005	0.945 ± 0.005	0.960 ± 0.004
Cohen's kappa	0.685 ± 0.020	0.881 ± 0.011	0.908 ± 0.010	0.880 ± 0.008	0.894 ± 0.009	0.922 ± 0.007
MCC	0.690 ± 0.018	0.883 ± 0.010	0.910 ± 0.009	0.883 ± 0.008	0.896 ± 0.008	0.923 ± 0.007

Next, for a fairer comparison between optimizers (SGD, RMSProp, Adam, NAG, NAdam and BPGe), we follow the strategy of cross-validation and sweep functionality explained in Section 5.1. For this study, we create two datasets, one for movement and the other for rest, formed each one by 4340 samples (selected from the union of the previous datasets).

The metrics and the BCa bootstrap CI computed for this HAR dataset are in Table 9. In this case, we can observe that the best results are obtained with the BPGe optimizer followed by Adam and NAdam. The worst results, as in the Lorenz test case, are those of SGD. The value of the metrics of RMSProp and NAG are quite similar. We can observe for all optimizers that the value of specificity is higher than sensitivity, which means that the rest samples are better classified than movement ones (although the difference is small except for SGD). As in the case of the Lorenz system, since Cohen's kappa coefficient and Matthews' Correlation Coefficient are both close to 1, the binary classification task is better than random and it is close to perfect classification for all optimizers except for SGD whose results are further from 1. The Youden's Index and F1-score values are almost 1 (except for Youden's Index in SGD), which gives us the idea that the optimizers are able to discriminate between classes and show a good degree of efficiency (highlighting the BPGe values that are closer to 1 for both metrics).

In Fig. 7, we represent statistical violin plots to show the distribution of the test accuracy for the different optimizers. As observed in Table 9, the lower accuracy is obtained with SGD optimizer, showing a big difference with the remaining optimizers that have a similar shape. We can observe that the highest accuracy is the one of BPGe followed by Adam. Notice that BPGe provides one of the most symmetric distributions.

As a final comparison, the performance of Adam and NAdam optimizers versus the performance of BPGe optimizer is studied using the Wilcoxon signed-rank test. As the p-value when comparing Adam vs BPGe and NAdam vs BPGe is in both cases on the order of 10^{-8} (with an effect size of 1), there is a significant difference between BPGe and the other optimizers. Thus, BPGe is clearly better in this test case.

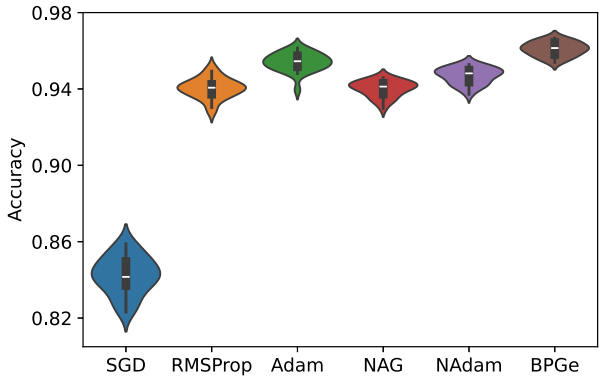

Fig. 7. Statistical violin plots to visualize the test accuracy distribution of the different optimizers in the HAR test.

Table 10
Loss and accuracy for training, validation and test sets for the four optimizers (SGD, RMSProp, Adam, BPGe) and three different learning rate values ($\lambda_1 = 0.001$, $\lambda_2 = 0.0005$, $\lambda_3 = 0.0001$) for the classification between even and odd numbers from the MNIST dataset. Specificity and sensitivity are also computed for test dataset.

		TRAIN		VALIDATION		TEST			
		Loss	Accuracy	Loss	Accuracy	Loss	Accuracy	Specificity	Sensitivity
SGD	$\lambda_1 = 0.001$	0.424	0.816	0.443	0.808	0.428	0.807	0.802	0.812
	$\lambda_2 = 0.0005$	0.466	0.786	0.478	0.773	0.464	0.775	0.768	0.782
	$\lambda_3 = 0.0001$	0.533	0.730	0.536	0.728	0.523	0.727	0.714	0.739
RMSProp	$\lambda_1 = 0.001$	0.176	0.939	0.292	0.887	0.274	0.891	0.870	0.912
	$\lambda_2 = 0.0005$	0.198	0.929	0.292	0.884	0.276	0.887	0.886	0.888
	$\lambda_3 = 0.0001$	0.272	0.897	0.326	0.861	0.312	0.872	0.878	0.867
Adam	$\lambda_1 = 0.001$	0.166	0.943	0.284	0.892	0.267	0.893	0.891	0.896
	$\lambda_2 = 0.0005$	0.191	0.933	0.290	0.882	0.275	0.888	0.907	0.868
	$\lambda_3 = 0.0001$	0.271	0.897	0.325	0.860	0.311	0.872	0.874	0.871
BPGe	$\lambda_1 = 0.001$	0.134	0.952	0.325	0.888	0.293	0.896	0.886	0.906
	$\lambda_2 = 0.0005$	0.174	0.936	0.291	0.885	0.269	0.895	0.903	0.888
	$\lambda_3 = 0.0001$	0.164	0.943	0.279	0.891	0.259	0.895	0.898	0.892

5.4. Classification in the MNIST dataset

MNIST dataset [48] consists of images of handwritten digits and it is commonly used in Machine Learning as a benchmark set. These images are separated in ten classes representing the digits from 0 to 9 and each image is of shape 28×28, that is, it is formed by 784 pixels. To have a binary classification task we separate the digits into two classes: odd and even (0 is classified as an even number to have the same number of digits in each class). As we work with a Recurrent-like Artificial Neural Network, we 'transform' the samples into time series: Each image is rearranged as a one-dimensional vector (rows are stacked one after the other) and, for the input of the Reservoir Computing, we interpret such vector as a time series of length 56 with 14 points at each time (the first 14 pixels of the vector are the first time series element, the next 14 are the second one, and so on). Notice that each pixel value is between 0 and 1.

In the first analysis we compare the performance of four optimizers (SGD, RMSProp, Adam and BPGe) for different fixed values of the learning rate ($\lambda_1 = 0.001$, $\lambda_2 = 0.0005$, $\lambda_3 = 0.0001$) and we study the convergence speed. The original MNIST dataset [48] consists of a training and test set with $60,000$ and $10,000$ samples, respectively. To create our training and validation datasets, we randomly select 500 and 300 samples of each digit from the original training set, respectively. This makes a total of 5000 and 3000 samples on each new set. For test set, we randomly select 500 samples of each digit from the original test set, which makes a total of 5000 data points. For training, validation and test sets we consider a batch size of 128, 100 and 250, respectively.

In Table 10 we have the loss and accuracy values for the training, validation and test sets for the four optimizers (SGD, RMSProp, Adam and BPGe) and the three different learning rate values ($\lambda_1 = 0.001$, $\lambda_2 = 0.0005$, $\lambda_3 = 0.0001$). For the test set, sensitivity and specificity have also been computed to ensure that the network has learned to properly distinguish both type of numbers (odd and even). We can observe that in general the network is suffering some degree of overfitting (test loss larger than train loss), but with the test information we can conclude that the network is able to perform the binary task correctly (as the value of test accuracy is close to 0.9 in almost all cases and the specificity and sensitivity values are balanced). The optimizer with the worst results is SGD and the one with the best results is BPGe slightly followed by Adam and RMSProp (whose results are quite similar).

Table 11 allows us to study the convergence speed of each optimizer. In fact, such table contains the epoch in which the accuracy of the training dataset is greater than or equal to 0.80, 0.85 and 0.90 for the first time during training process for the four optimizers and the three different values of the learning rate. For SGD, most of the cells are filled with "–" as it does not achieve an accuracy for the training dataset greater than or equal to the indicated values. As in the case of the Lorenz system and the HAR dataset, BPGe

Table 11
First epoch in the training process of the classification between odd and even in the MNIST dataset in which the accuracy of the training set is equal to or greater than 0.80, 0.85 and 0.90 for the four optimizers and the considered learning rates.

Accuracy		≥ 0.80	≥ 0.85	≥ 0.90
SGD	$\lambda_1 = 0.001$	2777	–	–
	$\lambda_2 = 0.0005$	–	–	–
	$\lambda_3 = 0.0001$	–	–	–
RMSProp	$\lambda_1 = 0.001$	50	153	833
	$\lambda_2 = 0.0005$	76	235	1223
	$\lambda_3 = 0.0001$	299	920	–
Adam	$\lambda_1 = 0.001$	41	126	542
	$\lambda_2 = 0.0005$	65	204	927
	$\lambda_3 = 0.0001$	296	873	–
BPGe	$\lambda_1 = 0.001$	10	21	57
	$\lambda_2 = 0.0005$	8	17	59
	$\lambda_3 = 0.0001$	17	43	231

Table 12
Wall time study (in seconds) for the binary classifications (even vs odd) in the MNIST dataset. For each optimizer, the approximate time per epoch and the time needed to reach for the first time a 0.90 in train accuracy are provided.

Time (s)	SGD	RMSProp	Adam	BPGe
Per Epoch	0.289	0.295	0.295	0.320
To ≥ 0.90	–	245.735	159.890	73.900

is the optimizer that needs fewer number of epochs to achieve a high value for the accuracy (of training set) as less than 45 epochs are needed to obtain a value equal to or greater than 0.85 for the three studied learning rate values. As can be observed, the number of epochs needed to surpass the indicated accuracy values is significantly lower for BPGe than for the other optimizers.

The time needed to train the network (time to train during 4000 epochs) for this task is around 19 minutes when SGD is considered, 20 minutes for Adam and RMSProp, and 21 for BPGe optimizer. To have an overview of the time needed to achieve a certain train performance, we measure the approximate time per epoch (in seconds) needed by each optimizer and we use the results in Table 11 to compute the time used to achieve (for the first time) an accuracy value greater than or equal to 0.90 in training set during the training process. We select the learning rate with the lowest test loss (see Table 10). These results have been summarized in Table 12. As in the other two test cases (Lorenz system and HAR dataset), BPGe requires less time to surpass a training accuracy of 0.90.

Now, we carry out a more fair comparison between six optimizers (SGD, RMSProp, Adam, NAG, NAdam and BPGe) following the strategy of cross-validation and sweep functionality explained in Section 5.1. For this study, we merge the previous training, validation and test datasets, selecting a unique set with 6500 samples of each class (odd and even). The 60% of the samples of this dataset is considered as training dataset, the 20% as validation set and the remaining 20% as test dataset. The values of the metrics and the confidence intervals computed for the MNIST dataset are in Table 13. In this test case, the better results (focusing on the mean value of the accuracy for all the optimizers) are obtained with the Adam optimizer followed by NAG and BPGe. The worst results are the ones of SGD and RMSProp. Notice that the value of the standard deviation of the accuracy for NAG optimizer stands out as it is much larger than for the remaining optimizers. This fact can be interpreted as meaning that the accuracy is quite different across attempts, so it is not so stable as the other optimizers. Then, in some trials, BPGe will have a higher value for the accuracy than NAG. These conclusions are also supported by the BCa bootstrap CI for the accuracy. We can observe that, for all optimizers except for BPGe, the value of specificity is higher than sensitivity, which means that the even samples are better classified than the odd ones. Cohen's kappa coefficient and Matthews' Correlation Coefficient for Adam, NAG and BPGe are both closer to 1 than for NAdam, SGD and RMSProp, therefore, the binary task is carried out in a better way by the first three named optimizers. Focusing on the Youden's Index and F1-score, we can conclude that Adam, NAG and BPGe show a good capacity of discrimination between both classes and a good degree of efficiency.

In Fig. 8, we show statistical violin plots to present the distribution for the test accuracy of the studied optimizers. Notice that all the violins have a similar shape, except for NAG optimizer that shows a very enlarged distribution (across some trials, NAG exhibits higher accuracy than BPGe, however, in other attempts, BPGe outperforms NAG optimizer as concluded from Table 13).

Finally, the performance of Adam and NAG optimizers versus the performance of BPGe optimizer is analysed using the Wilcoxon signed-rank test. As the p-value when comparing Adam and BPGe is on the order of 10^{-8} (with effect size 1), there is a significant statistical difference between BPGe and Adam. However, the p-value for NAG and BPGe comparison is 0.090 (with effect size 0.391) suggesting no significant difference between those optimizers.

Table 13
MNIST test study. Comparison of the performance of SGD, RMSProp, Adam, NAG, NAdam and BPGe optimizers for the 25 trials with different weights-biases initializations and different sets for training, validation and test, and with the learning rate λ that gives the best model (less validation loss) for each optimizer in the range $\lambda \in [10^{-5}, 10^{-1}]$ (for NAG, also the momentum is tuned in $[0.8, 0.99]$). The studied results are the evaluation metrics (expressed as $mean \pm std$) and the confidence intervals explained in Section 5.1.

	SGD	RMSProp	Adam	NAG	NAdam	BPGe
Accuracy	0.764 ± 0.008	0.794 ± 0.007	0.878 ± 0.006	0.872 ± 0.027	0.813 ± 0.005	0.863 ± 0.007
BCa CI for Accuracy	$[0.761, 0.767]$	$[0.791, 0.796]$	$[0.875, 0.880]$	$[0.861, 0.881]$	$[0.811, 0.815]$	$[0.860, 0.866]$
Precision	0.774 ± 0.010	0.808 ± 0.042	0.879 ± 0.008	0.879 ± 0.025	0.829 ± 0.041	0.854 ± 0.030
Sensitivity	0.746 ± 0.011	0.781 ± 0.073	0.876 ± 0.010	0.863 ± 0.043	0.797 ± 0.068	0.878 ± 0.046
Specificity	0.782 ± 0.012	0.807 ± 0.067	0.880 ± 0.010	0.881 ± 0.026	0.829 ± 0.063	0.847 ± 0.041
Youden's Index	0.528 ± 0.015	0.588 ± 0.013	0.756 ± 0.012	0.744 ± 0.053	0.626 ± 0.011	0.726 ± 0.014
F1-score	0.760 ± 0.008	0.790 ± 0.019	0.878 ± 0.006	0.870 ± 0.029	0.809 ± 0.016	0.865 ± 0.011
Cohen's kappa	0.528 ± 0.015	0.588 ± 0.013	0.756 ± 0.012	0.744 ± 0.053	0.626 ± 0.011	0.726 ± 0.014
MCC	0.529 ± 0.015	0.594 ± 0.012	0.756 ± 0.012	0.745 ± 0.052	0.632 ± 0.010	0.729 ± 0.014

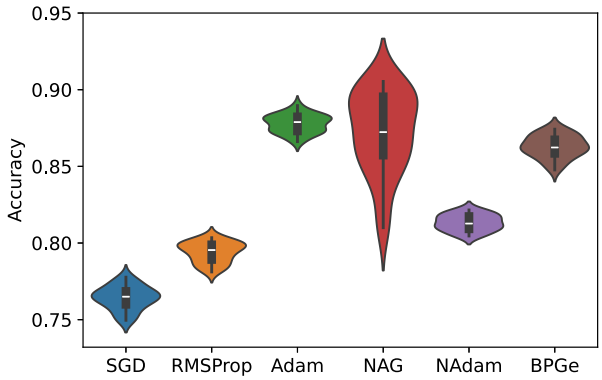

Fig. 8. Statistical violin plots to visualize the test accuracy distribution of the different optimizers in the MNIST test.

6. Conclusions

This paper deals with the training process of a Reservoir Computing algorithm. We have formulated the supervised learning problem using the recently introduced Bregman Proximal Gradient with extrapolation (BPGe) algorithm developed for non-convex optimization problems. We have applied such an algorithm to binary classification problems. Three test examples are presented to illustrate the effectiveness of the proposed approach, one based on chaotic data classification in the classical Lorenz system, the other on the Human Activity Recognition (HAR) using smartphones dataset for movement–rest classification, and the third one on the MNIST dataset for even–odd classification. From comparisons with Stochastic Gradient Descent (SGD), Root Mean Square Propagation (RMSProp), Adaptive Moment Estimation (Adam), Nesterov Accelerated Gradient (NAG) and Adam with Nesterov momentum (NAdam), it appears that the new methodology (BPGe) is highly competitive in terms of speed of convergence, quality and accuracy. It is remarkable that just a few epochs permit an accuracy greater than 0.90 (for training set). In particular, for the Lorenz test problem, the performance of the BPGe is comparable to widely used optimizers for ANNs such as Adam and NAG. In the HAR using smartphones dataset, it can be seen that BPGe improves the results, also accelerating the training process. In the MNIST study, we can observe that the best results are obtained with Adam optimizer, followed by NAG and BPGe. In this test case, the high standard deviation of NAG optimizer makes that for some trials NAG is better than BPGe while in others BPGe outperforms NAG. In all the test cases, we can observe that BPGe needs less epochs to obtain good results than other optimizers and its accuracy distribution is very compact and symmetric. Moreover, in all cases, the BPGe optimizer is in the top 3 for the best results. The differences of the best optimizers for the problems match perfectly with the ideas stated in the No Free Lunch theorems [23]. That is, the BPGe is the best optimization algorithm for some problems and hyperparameter configurations (as seen in some results in this article) and is slightly surpassed (but comparable in performance) by other optimizers in other problems or hyperparameter configurations. Therefore, the BPGe algorithm is a highly recommended option if we want to achieve outstanding results or to surpass 0.90 of accuracy in a few number of epochs (which can lead to lower energy and resource expenditure in some problems).

In this article, the BPGe algorithm has been adapted for a binary classification task using a Reservoir Computing network. Part of our future research is to extend the BPGe algorithm to more generic ANNs architectures and problems. One of the biggest challenges we face when extending to other problems and architectures is, for example, the correct choice of Bregman distance, since many strict properties must be fulfilled.

CRediT authorship contribution statement

Carmen Mayora-Cebollero: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Ana Mayora-Cebollero:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Álvaro Lozano:** Writing – review & editing, Visualization, Validation, Supervision, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Roberto Barrio:** Writing – review & editing, Supervision, Project administration, Methodology, Investigation, Funding acquisition, 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.

Acknowledgments

RB, AL, AMC and CMC have been supported by the Spanish research projects PID2021-122961NB-I00 and PID2024-156032NB-I00. AL, AMC and CMC have been supported by Spanish research project PID2022-140556OB-I00. RB, AMC and CMC have been supported by the European Regional Development Fund and Diputación General de Aragón E24-23R. RB has been supported by the European Regional Development Fund and Diputación General de Aragón LMP94-21. AL has been supported by the European Regional Development Fund and Diputación General de Aragón E22-23R.

Appendix A. Python code of BPGe optimizer for binary classification with a Reservoir Computing network

In this appendix, the code of the function of BPGe algorithm used to perform the three experiments for binary classification with a Reservoir Computing network can be found. For the sake of readability, in Table A.14 the variables and the parameters of the input of the function are explained.

Table A.14
Summary of information about the inputs of BPGe optimizer function.

Symbol in formulas	Name in the code	Explanation
x_i	`params_k`	Current value of the iteration
y_i	`params_y_k`	Extrapolation value
–	`batch_value_neurons_reservoir`	Values of the reservoir neurons after evolving for given inputs
–	`batch_labels`	Labels corresponding to the inputs used to compute `batch_value_neurons_reservoir`
γ	`gamma_reg`	Regularization parameter
λ_i	`lambda_k`	Step size (learning rate) on each iteration
β_0	`beta_k_0`	Initial value of extrapolation parameter
–	`line_search`	True or false variable to detect if line search is applied
ρ	`rho`	Scale factor of line search algorithm
η	`eta`	Scale factor of line search algorithm
μ	`mu`	f is μ-weakly convex relative to h

```
def BPGe_optimizer(params_k, params_y_k, batch_value_neurons_reservoir, batch_labels, gamma_reg, lambda_k,
    beta_k_0, line_search, rho, eta, mu):

    with torch.no_grad():

        params_km1 = copy.deepcopy(params_k)
        batch_value_neurons_reservoir_use = torch.unsqueeze(batch_value_neurons_reservoir.clone(), 2)
        batch_value_neurons_reservoir_use = batch_value_neurons_reservoir_use.permute(0, 2, 1)
        first_it = True

        for y_k in params_y_k:
            if first_it:
                y_k_weights = ((y_k.data).clone()).t()
                aux_for_grad = torch.matmul(batch_value_neurons_reservoir_use, y_k_weights)
                aux_for_grad = torch.squeeze(aux_for_grad, 1)
                first_it = False
            else:
                y_k_bias = (y_k.data).clone()
                aux_for_grad += y_k_bias

        exp_aux_for_grad = torch.exp(aux_for_grad)
        denominator_aux_for_grad = torch.sum(exp_aux_for_grad, axis = 1)
        numerator_aux_for_grad = exp_aux_for_grad[torch.arange(len(batch_labels)), (1-batch_labels)]
        num_den_aux_for_grad = numerator_aux_for_grad / denominator_aux_for_grad
        first_it_2 = True

        for (x_k, y_k) in zip(params_k, params_y_k):
            if first_it_2:
                aux_num_den_aux_for_grad = torch.unsqueeze(num_den_aux_for_grad, 1)
                elem_grad_without_sign = aux_num_den_aux_for_grad * batch_value_neurons_reservoir
                sign_with_the_labels_class_0 = (-1)**(batch_labels)
                aux_sign_with_the_labels_class_0 = torch.unsqueeze(sign_with_the_labels_class_0, 1)
                elem_grad_with_sign_class_0 = elem_grad_without_sign * aux_sign_with_the_labels_class_0
                grad_class_0 = -torch.sum(elem_grad_with_sign_class_0, 0)
                x_k.data[0] = (-grad_class_0 + (1/lambda_k) * y_k.data[0]) / (2*gamma_reg + (1/lambda_k))
                sign_with_the_labels_class_1 = (-1)**(batch_labels+1)
                aux_sign_with_the_labels_class_1 = torch.unsqueeze( sign_with_the_labels_class_1, 1)
                elem_grad_with_sign_class_1 = elem_grad_without_sign * aux_sign_with_the_labels_class_1
                grad_class_1 = -torch.sum(elem_grad_with_sign_class_1, 0)
                x_k.data[1] = (-grad_class_1 + (1/lambda_k) * y_k.data[1]) / (2*gamma_reg + (1/lambda_k))
                first_it_2 = False
            else:
                elem_grad_without_sign_b = num_den_aux_for_grad
                sign_with_the_labels_class_0_b = (-1)**(batch_labels)
                elem_grad_with_sign_class_0_b = elem_grad_without_sign_b * sign_with_the_labels_class_0_b
                grad_class_0_b = -torch.sum(elem_grad_with_sign_class_0_b)
                x_k.data[0] = (-grad_class_0_b + (1/lambda_k) * y_k.data[0]) / (2*gamma_reg + (1/lambda_k))
                sign_with_the_labels_class_1_b = (-1)**(batch_labels+1)
                elem_grad_with_sign_class_1_b = elem_grad_without_sign_b * sign_with_the_labels_class_1_b
                grad_class_1_b = -torch.sum(elem_grad_with_sign_class_1_b)
                x_k.data[1] = (-grad_class_1_b + (1/lambda_k) * y_k.data[1]) / (2*gamma_reg + (1/lambda_k))

        zipped_ls = [(x_km1_ls, x_k_ls, y_k_ls) for x_km1_ls, x_k_ls, y_k_ls in zip(params_km1, params_k,
            params_y_k)]

        if line_search == False:
            beta_k = beta_k_0
            for (x_km1_ls, x_k_ls, y_k_ls) in zipped_ls:
                y_k_ls.data = x_k_ls.data + beta_k * (x_k_ls.data - x_km1_ls.data)
        else:
            CC = 1
            beta_k = beta_k_0
            C_k = (1/lambda_k) / ((1/lambda_k) + mu)
            rho_C_k = rho * C_k

            while CC >= rho_C_k:
                beta_k = eta * beta_k
                for (x_km1_ls, x_k_ls, y_k_ls) in zipped_ls:
                    y_k_ls.data = x_k_ls.data + beta_k * (x_k_ls.data - x_km1_ls.data)
                D_h_1 = 0
                D_h_2 = 0
                first_it_3 = True
                for (x_km1_ls, x_k_ls, y_k_ls) in zipped_ls:
                    if first_it_3:
                        D_h_1 = D_h_1 + torch.sum((x_k_ls.data[0] - y_k_ls.data[0])**2) + torch.sum((
                            x_k_ls.data[1] - y_k_ls.data[1])**2)
                        D_h_2 = D_h_2 + torch.sum((x_km1_ls.data[0] - x_k_ls.data[0])**2) + torch.sum((
                            x_km1_ls.data[1] - x_k_ls.data[1])**2)
                        first_it_3 = False
                    else:
                        D_h_1 = D_h_1 + ((x_k_ls.data[0] - y_k_ls.data[0])**2) + ((x_k_ls.data[1] - y_k_ls
                            .data[1])**2)
                        D_h_2 = D_h_2 + ((x_km1_ls.data[0] - x_k_ls.data[0])**2) + ((x_km1_ls.data[1] -
                            x_k_ls.data[1])**2)

                CC = D_h_1 / D_h_2

    return params_k, params_y_k
```

Appendix B. Supplementary data

Supplementary data to this article can be found online at doi:10.1016/j.ins.2026.123275. It can also be consulted at https://www.youtube.com/watch?v=cgcXCUczYUk.

Data availability

Data will be made available on request.

References

[1] R.-Y. Sun, Optimization for deep learning: an overview, J. Oper. Res. Soc. China 8 (2) (2020) 249–294.
[2] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016, http://www.deeplearningbook.org.
[3] G. Hinton, T. Tieleman, Lecture 6.5-RMSProp: divide the gradient by a running average of its recent magnitude, COURSERA: Neural networks for machine learning 4 (2012) 26–31.
[4] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980, 2014.
[5] J. Bernstein, Y.-X. Wang, K. Azizzadenesheli, A. Anandkumar, signSGD: compressed optimisation for non-convex problems, in: International Conference on Machine Learning, PMLR, 2018, pp. 560–569.
[6] L. Bungert, T. Roith, D. Tenbrinck, M. Burger, A Bregman learning framework for sparse neural networks, J. Mach. Learn. Res. 23 (192) (2022) 1–43.
[7] X. Xie, P. Zhou, H. Li, Z. Lin, S. Yan, Adan: adaptive nesterov momentum algorithm for faster optimizing deep models, IEEE Trans. Pattern Anal. Mach. Intell. 46 (12) (2024) 9508–9520.
[8] Y.-J. Lee, W.-F. Hsieh, C.-M. Huang, Epsilon-SSVR: a smooth support vector machine for epsilon-insensitive regression, IEEE Trans. Knowl. Data Eng. 17 (5) (2005) 678–685.
[9] J. Hajewski, S. Oliveira, D. Stewart, Smoothed hinge loss and ℓ^1 support vector machines, in: 2018 IEEE International Conference on Data Mining Workshops (ICDMW), IEEE, 2018, pp. 1217–1223.
[10] T. Yang, M. Mahdavi, R. Jin, S. Zhu, An efficient primal dual prox method for non-smooth optimization, Mach. Learn. 98 (3) (2015) 369–406.
[11] S. Lyaqini, M. Nachaoui, A. Hadri, An efficient primal-dual method for solving non-smooth machine learning problem, Chaos Solitons Fract. 155 (2022) 111754.
[12] P. Jain, P. Kar, Non-convex optimization for machine learning, Found. Trends Mach. Learn. 10 (3–4) (2017) 142–363.
[13] X. Zhang, R. Barrio, M.A. Martínez, H. Jiang, L. Cheng, Bregman proximal gradient algorithm with extrapolation for a class of nonconvex nonsmooth minimization problems, IEEE Access 7 (2019) 126515–126529.
[14] J. Martens, I. Sutskever, Learning recurrent neural networks with hessian-free optimization, in: Proceedings of the 28th International Conference on Machine Learning (ICML-11), 2011, pp. 1033–1040.
[15] A.S. Berahas, M. Jahani, P. Richtárik, M. Takáč, Quasi-Newton methods for machine learning: forget the past, just sample, Optim. Methods Softw. 37 (5) (2022) 1668–1704.
[16] K. Osawa, Y. Tsuji, Y. Ueno, A. Naruse, R. Yokota, S. Matsuoka, Second-order optimization method for large mini-batch, arXiv preprint arXiv:1811.12019, 2018.
[17] R. Anil, V. Gupta, T. Koren, K. Regan, Y. Singer, Second order optimization made practical, arXiv preprint arXiv:2002.09018, 2020.
[18] N. Polson, J.G. Scott, B.T. Willard, Proximal algorithms in statistics and machine learning, Statist. Sci. 30 (4) (2015) 559–581.
[19] L. Yang, J. Zhang, J. Shenouda, D. Papailiopoulos, K. Lee, R.D. Nowak, A better way to decay: proximal gradient training algorithms for neural nets, in: OPT 2022: Optimization for Machine Learning (NeurIPS 2022 Workshop), 2022.
[20] M.C. Mukkamala, Bregman Proximal Minimization Algorithms, Analysis and Applications, Ph.D. thesis, Dissertation, Tübingen, Universität Tübingen, 2021.
[21] I. Sutskever, J. Martens, G. Dahl, G. Hinton, On the importance of initialization and momentum in deep learning, in: International Conference on Machine Learning, pmlr, 2013, pp. 1139–1147.
[22] T. Dozat, Incorporating Nesterov momentum into Adam, in: Proceedings of the 4th International Conference on Learning Representations, Workshop Track, San Juan, Puerto Rico, 2016, pp. 1–4.
[23] D. Wolpert, W. Macready, No free lunch theorems for optimization, IEEE Trans. Evol. Comput. 1 (1) (1997) 67–82.
[24] J. Duchi, E. Hazan, Y. Singer, Adaptive subgradient methods for online learning and stochastic optimization, J. Mach. Learn. Res. 12 (7) (2011).
[25] Y. Nesterov, A method for unconstrained convex minimization problem with the rate of convergence $o(1/k^2)$, Dokl. Akad. Nauk. SSSR 269 (1983) 543.
[26] J. Bolte, S. Sabach, M. Teboulle, Y. Vaisbourd, First order methods beyond convexity and Lipschitz gradient continuity with applications to quadratic inverse problems, SIAM J. Optim. 28 (3) (2018) 2131–2151.
[27] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E.Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, PyTorch: an imperative style, high-performance deep learning library, in: Advances in Neural Information Processing Systems 32, NeurIPS 2019, Vancouver, BC, Canada, 2019, pp. 8024–8035.
[28] D. Verstraeten, B. Schrauwen, M. d'Haene, D. Stroobandt, An experimental unification of reservoir computing methods, Neural Netw. 20 (3) (2007) 391–403.
[29] M. Cucchi, S. Abreu, G. Ciccone, D. Brunner, H. Kleemann, Hands-on reservoir computing: a tutorial for practical implementation, Neuromorph. Comput. Eng. 2 (3) (2022) 032002.
[30] H. Jaeger, The "Echo State" Approach to Analysing and Training Recurrent Neural Networks-With an Erratum Note, Bonn, Germany: German National Research Center for Information Technology GMD Technical Report, 2001 148 (34) 13.
[31] W. Maass, T. Natschläger, H. Markram, Real-time computing without stable states: a new framework for neural computation based on perturbations, Neural Comput. 14 (11) (2002) 2531–2560.
[32] J.J. Steil, Backpropagation-decorrelation: online recurrent learning with o(n) complexity, in: 2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No. 04CH37541), vol. 2, 2004, pp. 843–848.
[33] J. Pathak, Z. Lu, B.R. Hunt, M. Girvan, E. Ott, Using machine learning to replicate chaotic attractors and calculate Lyapunov exponents from data, Chaos: Interdiscip. J. Nonlinear Sci. 27 (12) (2017) 121102.
[34] S. Shahi, F.H. Fenton, E.M. Cherry, Prediction of chaotic time series using recurrent neural networks and reservoir computing techniques: a comparative study, Mach. Learn. Appl. 8 (2022) 100300.
[35] M. Lukoševičius, A practical guide to applying echo state networks, in: Neural Networks: Tricks of the Trade: Second Edition. Lecture Notes in Computer Science (LNCS), vol. 7700, Springer, Berlin, Heidelberg, 2012, pp. 659–686.
[36] O. Rainio, J. Teuho, R. Klén, Evaluation metrics and statistical tests for machine learning, Sci. Rep. 14 (1) (2024) 6086.
[37] L. Biewald, et al., Experiment Tracking with Weights and Biases, 2020.
[38] R.J. Tibshirani, B. Efron, An introduction to the bootstrap, Monogr. Stat. Appl. Probab. 57 (1) (1993) 1–436.
[39] F.F. Peres, Effect sizes for nonparametric tests, Biochem. Med. 36 (1) (2025) 010101.
[40] B. Wang, T. Nguyen, T. Sun, A.L. Bertozzi, R.G. Baraniuk, S.J. Osher, Scheduled restart momentum for accelerated stochastic gradient descent, SIAM J. Imaging Sci. 15 (2) (2022) 738–761.
[41] Y. Malitsky, K. Mishchenko, Adaptive proximal gradient method for convex optimization, Adv. Neural Inf. Process. Syst. 37 (2024) 100670–100697.
[42] E.N. Lorenz, Deterministic nonperiodic flow, J. Atmos. Sci. 20 (1963) 130–141.
[43] A. Wolf, J.B. Swift, H.L. Swinney, J.A. Vastano, Determining Lyapunov exponents from a time series, Phys. D 16 (3) (1985) 285–317.
[44] T. Tél, Y.-C. Lai, Chaotic transients in spatially extended systems, Phys. Rep. 460 (6) (2008) 245–275.

[45] R. Barrio, Á Lozano, A. Mayora-Cebollero, C. Mayora-Cebollero, A. Miguel, A. Ortega, S. Serrano, R. Vigara, Deep Learning for chaos detection, Chaos: Interdiscip. J. Nonlinear Sci. 33 (7) (2023) 073146.
[46] D. Anguita, A. Ghio, L. Oneto, X. Parra, J.L. Reyes-Ortiz, A public domain dataset for human activity recognition using smartphones, in: The European Symposium on Artificial Neural Networks, 2013.
[47] J. Reyes-Ortiz, D. Anguita, A. Ghio, L. Oneto, X. Parra, Human Activity Recognition Using Smartphones, UCI Machine Learning Repository, 2013, https://doi.org/10.24432/C54S4K
[48] Y. LeCun, C. Cortes, C.J. Burges, The MNIST database of handwritten digits, 1998, http://yann.lecun.com/exdb/mnist/, New York, USA.