Russian Journal of Nonlinear Dynamics, 2021, vol. 17, no. 1, pp. 5–21.
Full-texts are available at http://nd.ics.org.ru
DOI: 10.20537/nd210102

NONLINEAR PHYSICS AND MECHANICS

MSC 2010: 65P20, 37M05, 65L05

Artificial Neural Network as a Universal Model of Nonlinear Dynamical Systems

P. V. Kuptsov, A. V. Kuptsova, N. V. Stankevich

We suggest a universal map capable of recovering the behavior of a wide range of dynamical systems given by ODEs. The map is built as an artificial neural network whose weights encode a modeled system. We assume that ODEs are known and prepare training datasets using the equations directly without computing numerical time series. Parameter variations are taken into account in the course of training so that the network model captures bifurcation scenarios of the modeled system. The theoretical benefit from this approach is that the universal model admits applying common mathematical methods without needing to develop a unique theory for each particular dynamical equations. From the practical point of view the developed method can be considered as an alternative numerical method for solving dynamical ODEs suitable for running on contemporary neural network specific hardware. We consider the Lorenz system, the Rössler system and also the Hindmarch – Rose model. For these three examples the network model is created and its dynamics is compared with ordinary numerical solutions. A high similarity is observed for visual images of attractors, power spectra, bifurcation diagrams and Lyapunov exponents.

Keywords: neural network, dynamical system, numerical solution, universal approximation theorem, Lyapunov exponents

Received March 03, 2021
Accepted March 15, 2021

The work of PVK on theoretical formulation and numerical computations and the work of NVS on results analysis were supported by a grant of the Russian Science Foundation No. 20-71-10048.

Pavel V. Kuptsov
kupav@mail.ru
Nataliya V. Stankevich
stankevichnv@mail.ru
Laboratory of topological methods in dynamics, National Research University Higher School of Economics
ul. Bolshaya Pecherskaya 25/12, Nizhny Novgorod, 603155 Russia

Anna V. Kuptsova
anna.kuptsova@gmail.com
Institute of electronics and mechanical engineering, Yuri Gagarin State Technical University of Saratov
ul. Politekhnicheskaya 77, Saratov, 410054 Russia

1. Introduction

In this paper we suggest a map, i.e., a discrete time dynamical system capable of recovering the dynamics of systems given by ODEs. The map is built as an artificial neural network that encodes the modeled system. Using neural networks for dynamical systems reconstruction is a long-standing problem. But typically networks are used to predict dynamics when governing equations are unknown and only time series are available [10, 17–19]. We assume that ODEs are known and create their neural network model. The structure of the network is the same for all cases, while network weights are trained to fit the modeled dynamical system. To prepare training datasets, we do not use system time series. Instead we feed the network and the modeled system by random time series sampled from normal distributions and update network weights comparing the outputs. Parameter variations are taken into account so that the network captures bifurcation scenarios of the modeled system.

The motivations for this study are the following. We want to develop a method of training an artificial neural network that can operate as a discrete time system and can reproduce the behavior of a wide variety of dynamical systems. We consider a perceptron with one hidden layer and sigmoidal activation. Also, such a network is said to consist of two dense layers. Many contemporary investigations deal with deep networks whose number of layers is much higher than two and whose neuron interconnections are much more complicated. We prefer a classical architecture because of a solid mathematical background behind it, that is, the universal approximation theorem [6]. According to this theorem, the network considered is the simplest universal approximator, i.e., it is able to reproduce any function of multiple variables on a compact set. Such a simple universal model can be interesting for theoretical studies. Theoretical analysis of a dynamical system often requires developing highly specialized mathematical approaches unique for the system. Considering a universal model that covers a wide range of systems, one can extend theoretical results for this range of systems without needing always to recreate a special mathematical approach.

From the practical point of view the development of methods for creating a universal neural network that can model dynamics can be considered as an alternative to the existing numerical methods for solving dynamical ODEs. Although a large variety of well-established and effective numerical methods is available for computer simulation of dynamics, these methods are basically developed for single-thread computation. But the recent trend in computational hardware development has been to increase the number of computational cores instead of increasing the single-core speed. In particular, many hardwares are known today specialized for implementing artificial neural networks. In this situation it seems to be very important to develop new numerical approaches well fitted to a powerful contemporary hardware. Our model operates as a neural network that can be run either using various network software systems available today, such as TensorFlow [13] and PyTorch [30], or it can be uploaded to a dedicated computer chip called AI accelerator (AI stands for artificial intelligence) [20, 21].

2. Mathematical background: the universal approximation theorem

The problem of a universal construct for approximation of functions with many variables has a long history. We first mention the Weierstrass theorem [11], which states that any continuous function over a closed interval on the real axis can be expressed in that interval as an

absolutely and uniformly convergent series of polynomials. In 1900 David Hilbert outlined in the International Congress of Mathematicians in Paris 23 major mathematical problems for the coming new century. His 13th problem is whether solutions to a 7th-degree polynomial equation can be written as a composition of finitely many two-variable functions. Hilbert believed they could not. In 1956 and 1957, Kolmogorov and Arnold proved that each continuous function of N variables, including the case in which $N = 7$, can be written as a composition of continuous functions of two variables [1–3]. This is called the Kolmogorov – Arnold representation theorem.

The interest in the study of the virtues of multilayer perceptrons as devices for the representation of arbitrary continuous functions was perhaps first put into focus by Hecht-Nielsen [7]. In the context of traditional multilayer perceptrons, it was Cybenko who demonstrated rigorously for the first time that a single hidden layer is sufficient to uniformly approximate any continuous function with support in a unit hypercube [6]. In 1989, two other papers were published independently on multilayer perceptrons as universal approximators [5, 8]. For subsequent contributions to the approximation problem, see [9]. A review on this topic can also be found in [10]

To sum up, the universal approximation theorem states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of $\mathbb{R}^N$, under mild assumptions on the activation function. The theorem thus states that simple neural networks can represent a wide variety of interesting functions when given appropriate parameters.

3. The network and training details

Assume that we have an ODE

$$\dot{u} = f(u, p), \tag{3.1}$$

where $u \in \mathbb{R}^{N_u}$ is a vector of N_u dynamical variables and $p \in \mathbb{R}^{N_p}$ is a vector of N_p parameters.

We consider one of the simplest feed-forward networks, a perceptron with one hidden layer, or using more contemporary terms, a network with two dense layers. Formally the network can be represented as a function that maps vectors $u \in \mathbb{R}^{N_u}$ to vectors $d \in \mathbb{R}^{N_u}$,

$$d = F(u, p, w), \tag{3.2}$$

where w is a vector of network weights. Our purpose is to tune w in such a way that

$$u(t + \Delta t) = u(t) + d(t), \tag{3.3}$$

where $u(t)$ is a solution to ODE (3.1) and Δt is a time step. The size of the time step is defined before training the network. We take $\Delta t = 0.01$.

Consider a semiimplicit numerical scheme of ODE solution:

$$u(t + \Delta t) = u(t) + \frac{\Delta t}{2}\{f[u(t)] + f[u(t + \Delta t)]\}. \tag{3.4}$$

Compute the difference between (3.3) and (3.4):

$$e = d(t) - \frac{\Delta t}{2}\{f[u(t)] + f[u(t + \Delta t)]\}, \tag{3.5}$$

where e is the approximation error. Substituting $u(t + \Delta t)$ as $u(t) + d(t)$ from Eq. (3.3) and omitting t, we obtain

$$e = d - \frac{\Delta t}{2}[f(u) + f(u + d)]. \tag{3.6}$$

The network approximation (3.2), (3.3) works well if the approximation error tends to zero, $e \to 0$, for any u and p from the domain of interest.

Before training we need to define the localization areas for u and p. This is done empirically via testing various numerical solutions of Eq. (3.1). We define in this way a mean value μ_u and a standard deviation s_u of u and the corresponding μ_p and s_p by varying parameters p. The training occurs on a random u and p sampled from normal distribution defined by given mean values μ_u, μ_p and standard deviations s_u and s_p. Since we use a pseudo-random number generator to produce a data set, its size is limited only by a period of the generator that is very large.

Let us now discuss the structure of the network denoted above as $F(u, p, w)$. The network includes linear and nonlinear data transformations. The linear one is done via multiplication of data vectors by a matrix of neuron weights. For neural networks the usual order of vector-matrix manipulation is the reversed: Typically we multiply a matrix by a vector-column and in the neural network context a vector-row is multiplied by a matrix. This is done because in the course of training a batch of vectors is processed in parallel. A rectangular data matrix with the vectors stowed in rows is multiplied by a matrix of weights. Thus, we assume that u and p are vector rows of dimension N_u and N_p, respectively.

The training data vectors u and p are sampled from a normal distribution and elements of u and p can have different scales. Thus, the first transformation of the network inputs u and p is a nontrainable normalization layer that rescales inputs to a standard normal distribution:

$$\mathrm{Norm}(x) = (x - \mu_x)/s_x, \tag{3.7}$$

$$\mathrm{Denorm}(x) = xs_x + \mu_x. \tag{3.8}$$

Here x, μ_x and s_x are vectors rows and operations are performed elementwise. Also, we define here the layer performing backward operation Denorm(). It will be done at the very end of the network to fit the values to an appropriate range. It might seem that the layers (3.7) and (3.8) are superfluous — one can expect that the network is able to fit these scales itself in the course of training. But in fact this is not the case. All network training methods are developed in the assumption that both inputs and outputs do not deviate much from a standard range. So the training is efficient if we know in advance what the ranges of the inputs and the outputs are and rescale them appropriately.

After normalization we concatenate two resulting row vectors into one vector:

$$\mathrm{Concat}(x, y) = (x, y). \tag{3.9}$$

Here x and y are vectors of N_x and N_y elements, respectively, and (x, y) is a row vector of $N_x + N_y$ elements.

The next step is a dense layer. This is merely an affine transformation:

$$\mathrm{Dense}(x, N) = xW_{x,N} + b_{x,N}. \tag{3.10}$$

Here $W_{x,N}$ is a rectangular matrix whose number of rows equals the number of columns of x and the number of columns of $W_{x,N}$ is N, $b_{x,N}$ is a vector row with N elements.

After that a nonlinear transformation is applied that is called activation:

$$\mathrm{Activ}(x) = \sigma(x). \tag{3.11}$$

Here $\sigma()$ is a scalar function of a scalar argument, and if a vector is passed to it, the element-wise operation is assumed.

Subsequent transformations are done using already defined operators so that the whole network $d = F(u, p, w)$ can be described as follows:

$$z = \mathrm{Concat}(\mathrm{Norm}(u), \mathrm{Norm}(p)), \tag{3.12}$$

$$h = \mathrm{Activ}(\mathrm{Dense}(z, N_h)), \tag{3.13}$$

$$g = \mathrm{Dense}(h, N_u), \tag{3.14}$$

$$d = \Delta t\, \mathrm{Denorm}(g). \tag{3.15}$$

The variable w in $F(u, p, w)$ represents a set of trainable parameters of the networks. As follows from the equations above,

$$w = \{W_{z,N_h}, b_{z,N_h}, W_{h,N_u}, b_{h,N_u}\}. \tag{3.16}$$

At the very beginning the network parameters w are initialized at random. Then the training process is performed as follows. We generate an input batch $\{U, P\}$ of N_{batch} random u and p sampled from a normal distribution. Here U and P are matrices with N_{batch} rows, and their number of columns is N_u and N_p, respectively. This batch is fed to the network (3.12)–(3.15) and the matrix D with N_{batch} rows and N_u columns is obtained. Then the input matrix U and the output matrix D are substituted into (3.6) to compute an error matrix E of N_{batch} rows and N_u columns. Finally, a mean squared error (MSE) is computed for the elements of E as

$$\ell = \frac{1}{N_{\mathrm{batch}} N_u} \sum_{i=1}^{N_{\mathrm{batch}}} \sum_{j=1}^{N_u} e_{ij}^2. \tag{3.17}$$

This ℓ is the loss function for our training. To update the network parameters, a gradient of ℓ is computed with respect to each of the network parameters gathered in w, see (3.16), and then it is used in a gradient descent step that computes corrections to the network parameters with respect to the minimization of ℓ. The simplest version of the gradient descent step reads

$$w \leftarrow w - \gamma \nabla_w \ell, \tag{3.18}$$

where the step size scale γ is a small parameter controlling the convergence.

A training iteration that starts from a random batch generation and ends after updating network parameters is repeated t_{epoch} times. This is considered as an epoch. Notice that usually an epoch has a bit different meaning. Typically, when a large unaltered data set is used, the data set cannot be shown to the network at once due to the lack of computer memory. In this case the whole data set is split into batches (they are also called minibatches) and they are fed one by one. The parameter updates are computed for each batch. The optimization method applied not to the whole data set at once but only to its batches is called stochastic gradient descent, and the epoch ends when each batch has been shown to the network. In our case the batches are always generated at random so that dividing training process into epochs is required only to interrupt the training and to compute metrics to see the progress of the network performance.

When t_{epoch} training iterations are done and the epoch ends, we evaluate network performance making t_{valid} validation iterations. They include creation and feeding the network with the random batches $\{U, P\}$ (again of size N_{batch}), computation performance metrics, and finally averaging them over the validation steps t_{valid}. Unlike the training iterations, no network parameters updates are done. Two metrics are considered: the loss function MSE (3.17) and the

mean relative norm error (MRNE) that is defined as follows:

$$m = \frac{1}{N_{\text{batch}}} \sum_{i=1}^{N_{\text{batch}}} \left(\sum_{j=1}^{N_u} |e_{ij}| / \sum_{j=1}^{N_u} |u_{ij}| \right), \tag{3.19}$$

where e_{ij} are, as above, the elements of the error matrix E and u_{ij} are the elements of the network input batch U. To visualize the training routine, we plot the performance metrics vs the number of epochs passed. This is called learning curves, see Figs. 1, 4, and 8 below.

For all computations we set

$$N_{\text{batch}} = 10000, \quad t_{\text{epoch}} = 100, \quad t_{\text{valid}} = 3. \tag{3.20}$$

For actual computations instead of the simplest one (3.18) we use a more sophisticated version of the gradient descent method called Adam. The difference is that the step size scale γ is not a constant, but is tuned according to the accumulated gradients on the previous steps [12]. This method has a meta-parameter learning rate α that controls the overall scale of the computed step size. We decrease it in the course of the computations according to the inverse time decay rule:

$$\alpha = \frac{0.1}{1 + 0.96t/(30t_{\text{epoch}})}, \tag{3.21}$$

where t is the gradient descent step, and t_{epoch} is the number of steps comprising one epoch. The particular numerical values of the coefficients in this formula are chosen empirically to provide the fastest convergence.

As an activation layer Activ() in Eq. (3.13) we apply the sigmoidal function

$$\sigma(x) = \frac{1}{1 + \mathrm{e}^{-x}}. \tag{3.22}$$

We will train the neural network models to achieve the mean relative error MRNE at level 10^{-5}.

The transformation that is done by the network under consideration, (3.2), (3.12)–(3.15), can be represented as a map. The normalization operator in Eq. (3.12) can be taken into account inside the dense layer in (3.13) by an appropriate rescaling and shift of the elements of W_{z,N_h} and b_{x,N_h}. Similarly, the denormalization operator in Eq. (3.15) can be merged with the dense layer in (3.14). Also, instead of concatenating the normalized vectors u and p we split the matrix W_{z,N_h} into two blocks corresponding to u and p, respectively. As a result, we obtain the following map that models solutions to Eq. (3.1):

$$u_{n+1} = u_n + \sigma(u_n A_0 + pB_0 + a_0)A_1 + a_1, \tag{3.23}$$

where A_0 is a matrix with N_u rows and N_h columns, B_0 has N_p rows and N_h columns, A_1 is a matrix with N_h rows and N_u columns. The vector row a_0 has N_h elements and a_1 has N_u elements.

Equation (3.23) is a universal model of a solution to ODE (3.1). A particular system is selected by choosing an appropriate size N_h of the hidden layer and by numerical values of the elements of matrices A_0, B_0, A_1 and vectors a_0 and a_1.

For Eq. (3.23) we can find the variational equation suitable for applying the Lyapunov analysis to this system, in particular, for computing Lyapunov exponents. Differentiating the elements of u_{n+1} by the elements u_n, one obtains the Jacobian matrix:

$$J_n = I + A_0 H_n A_1, \tag{3.24}$$

where I is the identity matrix and H_n is a diagonal square matrix N_h by N_h:

$$H_n = \mathrm{diag}(h_n(1 - h_n)) \tag{3.25}$$

and h_n is a row vector computed as

$$h_n = \sigma(u_n A_0 + pB_0 + a_0). \tag{3.26}$$

In other words, it is computed according to Eqs. (3.12) and (3.13) when u_n and p corresponding to the current trajectory point are substituted there.

Thus, the variational equation for the system (3.23) reads

$$\delta u_{n+1} = (I + A_0 H_n A_1)\delta u_n. \tag{3.27}$$

This variational equation can be used to compute Lyapunov exponents. For this purpose we apply the standard algorithm [15, 16]: iterate the main system (3.23) simultaneously with the required number of copies of the variational equation (3.27) with periodic orthogonalization and normalization of the set of vectors δu_n. Accumulated and averaged in time logarithms of the norms of variational vectors converge to the Lyapunov exponents.

Since the training and running of the neural network is usually done in a multithread computation environment, the preferable way of finding the exponents is to iterate very many trajectories simultaneously for not very large time cuts and then to average the resulting exponents over the trajectories.

All the computations including training and running are performed using TensorFlow [13] and CUDA [14] software.

4. Models

4.1. Lorenz system

First we consider the Lorenz system [22–24]:

$$\begin{aligned} \dot{x} &= \sigma(y - x), \\ \dot{y} &= x(r - z) - y, \\ \dot{z} &= xy - bz. \end{aligned} \tag{4.1}$$

To train this model we choose the vectors of mean μ_u and standard deviation s_u as follows:

$$\begin{aligned} \mu_u &= (0, 0, 0), \qquad s_u = (10, 10, 20), \\ \mu_p &= (0, 0, 0), \qquad s_p = (5, 20, 2), \\ \mu_g &= (0, 0, 0), \qquad s_g = (70, 280, 110). \end{aligned} \tag{4.2}$$

These vectors are used in Eq. (3.12), see also Eq. (3.7).

The vectors μ_g and s_g are computed as mean and standard deviation of elements of $f(u, p)$ that is the right hand side of Eq. (4.1) when u and p are sampled from a normal distribution with mean and standard deviations μ_u, s_u, μ_p and s_p. In this case the network output d, see Eq. (3.15), will approximately have the range of $\frac{\Delta t}{2}[f(u) + f(u + d)]$, see Eqs. (3.4) and (3.5).

One more parameter that we need to define is N_h, the size of the hidden layer, see (3.13). We consider different values to check which one is preferred. Figure 1 shows the learning curves for the Lorenz system. As explained above, these curves represent metrics MSE (3.17), panel (a), and MRNE (3.19), panel (b), computed after each epoch of training (remember that one epoch includes updating the network parameters w (3.16) on $N_{\text{batch}} \times t_{\text{epoch}}$ (3.20) random samples). Three cases are shown corresponding to $N_h = 50$, 100 and 200. We see that all the curves decay, which means that the performance of the network improves. The fastest decay is observed for $N_h = 200$. In what follows we will consider the network with $N_h = 200$. In the course of the training after each epoch we compare the attained MRNE level with the previously smallest one. And if the new one is smaller, we save the corresponding network parameters w, see Eq. (3.16). After 20000 epochs we were able to find a network whose MRNE is approximately 3×10^{-5}. We use this metric as a criterion of the performance because it is normalized by the scale of dynamical variables so that we can compare the performance of different systems. Since MRNE is already sufficiently small, we did not consider larger values of N_h.

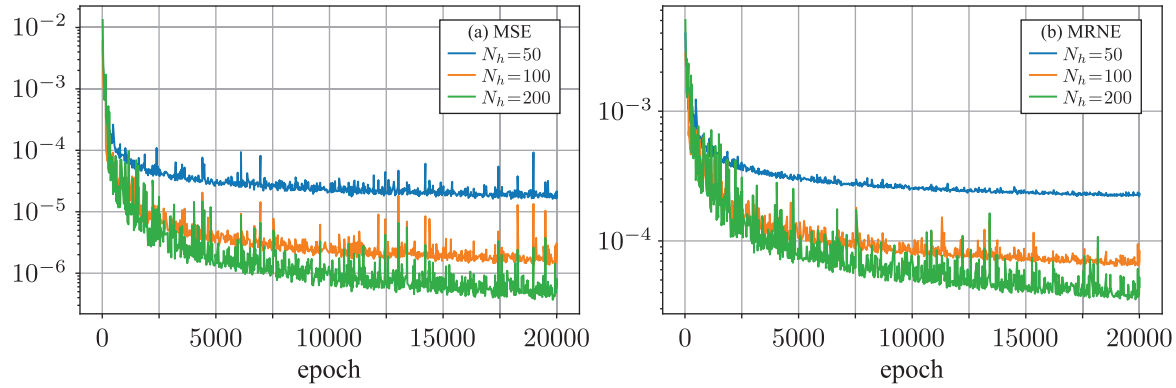

Fig. 1. Learning curves for the Lorenz network model corresponding to ODEs (4.1). The curves are computed at validation steps, i.e., for inputs that were not used for updating the network parameters. Panel (a) represents MSE, Eq. (3.17). Panel (b) shows MRNE, Eq. (3.19). Different curves correspond to different N_h.

The training result is shown in Figs. 2 and 3. Figure 2 demonstrates the Lorenz attractor computed for the standard set of parameters $\sigma = 10$, $r = 28$, $b = 8/3$ using the fourth-order Runge – Kutta method (a) and the network model (3.23) (b). Observe very high coincidence of two plots.

Neural networks architecture is very good suited for parallel computations. So doing computations with the network model, we employ it considering multiple trajectories at once: to plot Fig. 2a via the Runge – Kutta method, we compute 10000 steps with the time interval $\Delta t = 0.01$, while in Fig. 2b we compute 100 trajectories at once, each of the length 100 steps $\Delta t = 0.01$.

Figure 3 shows Fourier spectra computed for x at the parameters $\sigma = 10$, $r = 28$, $b = 8/3$, panels (a), (b), and $\sigma = 16$, $r = 45.92$, $b = 4$, panels (c), (d). The left panels (a) and (c) are computed for the Runge – Kutta data and the right ones are obtained for the network model. The spectra coincide very well, which indicates that the obtained network (3.23) models the Lorenz dynamics very well.

Now compute Lyapunov exponents using the standard algorithm [15, 16]. Using the Runge – Kutta method at $\sigma = 10$, $r = 28$, $b = 8/3$ we obtain the values of λ_i in Eq. (4.3). Lyapunov exponents $\widetilde{\lambda}_i$ computed for the network model (3.23) and the corresponding variational equa-

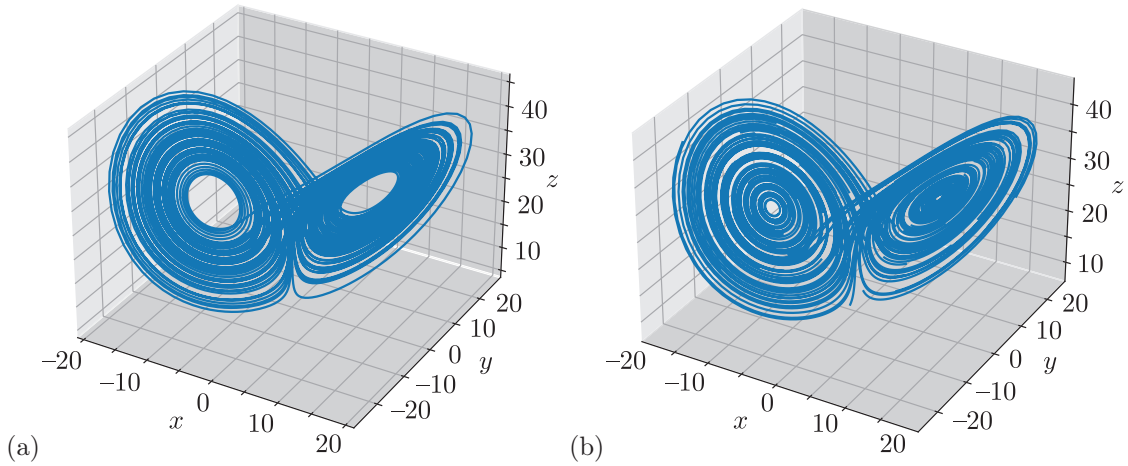

Fig. 2. Lorenz attractor computed (a) as a numerical solution of Eqs. (4.1) using the forth-order Runge – Kutta method and (b) as iteration of the network model (3.23). Parameters are $\sigma = 10$, $r = 28$, $b = 8/3$.

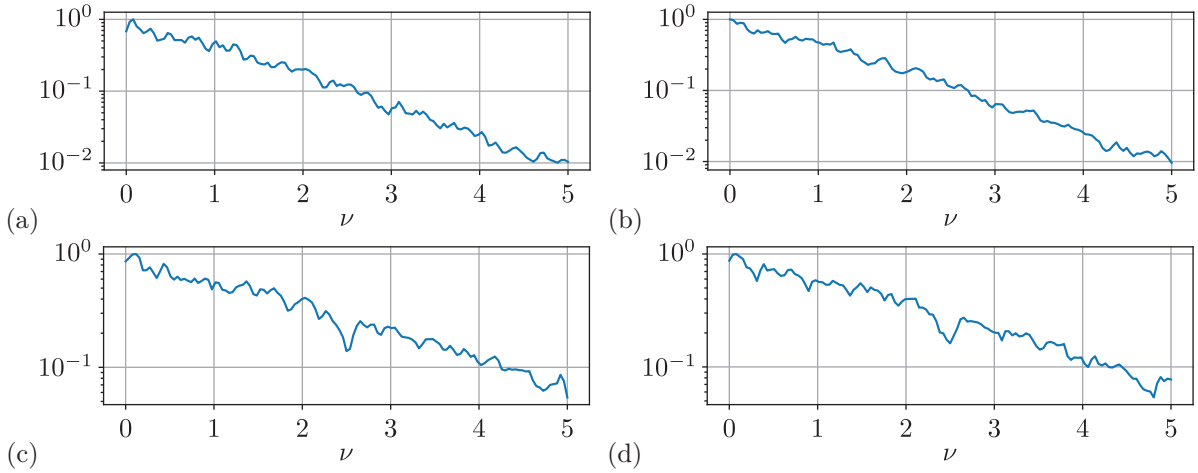

Fig. 3. Fourier spectra of the Lorenz attractor. Data series for panels (a) and (c) are computed numerically using the fourth-order Runge – Kutta method, and data for the panels (b) and (d) are obtained after iterations of the network model (3.23). Parameters for the panels (a) and (b) are $\sigma = 10$, $r = 28$, $b = 8/3$, and panels (c) and (d) are obtained with $\sigma = 16$, $r = 45.92$, $b = 4$.

tion (3.27) are contained in Eq. (4.4). Observe the very good coincidence. Notice that λ_2 is expected to be zero since it describes marginally stable perturbations along trajectories. However, actual values in computations are never exactly zero. Their closeness to zero indicates the quality of the computation. In our case, both λ_2 and $\widetilde{\lambda}_2$ are very small:

$$\lambda_1 = 0.906, \qquad \lambda_2 = 8.26 \times 10^{-6}, \qquad \lambda_3 = -14.6, \tag{4.3}$$

$$\widetilde{\lambda}_1 = 0.905, \qquad \widetilde{\lambda}_2 = 1.26 \times 10^{-5}, \qquad \widetilde{\lambda}_3 = -14.6. \tag{4.4}$$

Similarly, the Lyapunov exponents are computed for the parameters $\sigma = 16$, $r = 45.92$, $b = 4$. Observe again the very high similarity of λ_i with network model exponents $\widetilde{\lambda}_2$:

$$\lambda_1 = 1.50, \qquad \lambda_2 = -1.89 \times 10^{-5}, \qquad \lambda_3 = -22.5, \tag{4.5}$$

$$\widetilde{\lambda}_1 = 1.49, \qquad \widetilde{\lambda}_2 = 4.34 \times 10^{-5}, \qquad \widetilde{\lambda}_3 = -22.7. \tag{4.6}$$

4.2. Rössler system

Another system that we consider is the Rössler system [24–26]:

$$\begin{aligned} \dot{x} &= -y - z, \\ \dot{y} &= x + ay, \\ \dot{z} &= b + z(x - c). \end{aligned} \tag{4.7}$$

For this system we choose the following $\mu_{u,p}$ and $s_{u,p}$ and compute the corresponding μ_g and s_g:

$$\begin{aligned} \mu_u &= (0, 0, 0), \qquad s_u = (10, 10, 10), \\ \mu_p &= (0, 0, 0), \qquad s_p = (10, 10, 10), \\ \mu_g &= (0, 0, 0), \qquad s_g = (14, 101, 142). \end{aligned} \tag{4.8}$$

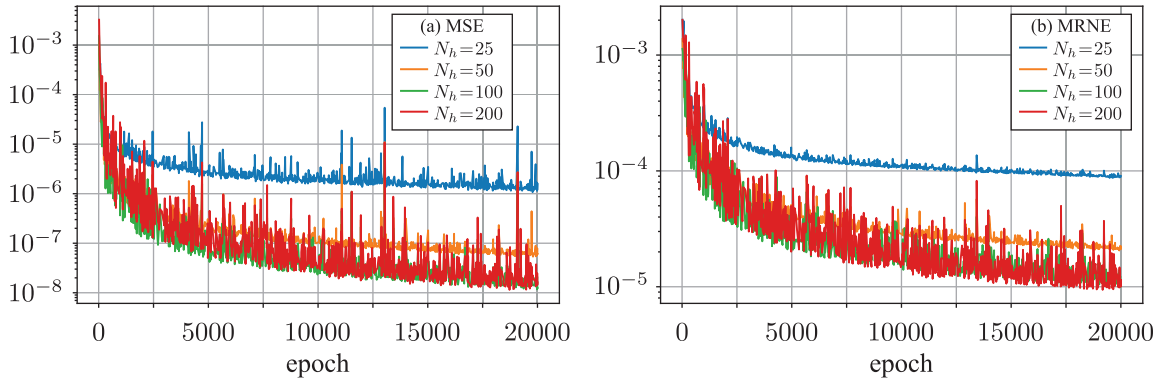

Fig. 4. Learning curves for the Rössler network model corresponding to ODEs (4.7).

Figure 4 demonstrates the learning curves for the Rössler system, i.e., decay of the metrics MSE, panel (a) and MRNE, panel (b), see Eqs. (3.17) and (3.19), respectively, in the course of training. We observe that the training now goes much faster than for the Lorenz system, see Fig. 1. Inspecting the learning curves, we can conclude that the network models with $N_h = 100$ and 200 do not differ much. So, unlike the Lorenz system, we will consider the network model for the Rössler system with $N_h = 100$. After 20000 epochs of the training we obtained the model with the smallest MRNE equal to approximately 1.0×10^{-5}.

Figure 5 demonstrates a chaotic Rössler attractor and the corresponding Fourier spectra computed for ODEs (4.7) (left column) and for the trained network model (right column). Numerical solutions of ODEs here and below are obtained using the fourth-order Runge – Kutta method. A very high similarity of the graphs indicates the high quality of approximation of the network model. Another example of dynamics is given in Fig. 6. Here the parameters correspond to the period 2 oscillations. The limit cycles in Fig. 6a and c look almost identical. The spectrum for the network model in Fig. 6d also repeats the spectrum in Fig. 6b in location and relative heights of harmonics. The difference between these two spectra is in small fluctuations. Since the regime of the system considered is periodic, the fluctuations are mere artifacts related, in particular, to the computation method. The methods of computations are different and so are the fluctuations.

To demonstrate that the trained network model reproduces the dynamics of the modeled ODEs in a wide range of parameters, in Fig. 7 we show a bifurcation diagram for the Rössler

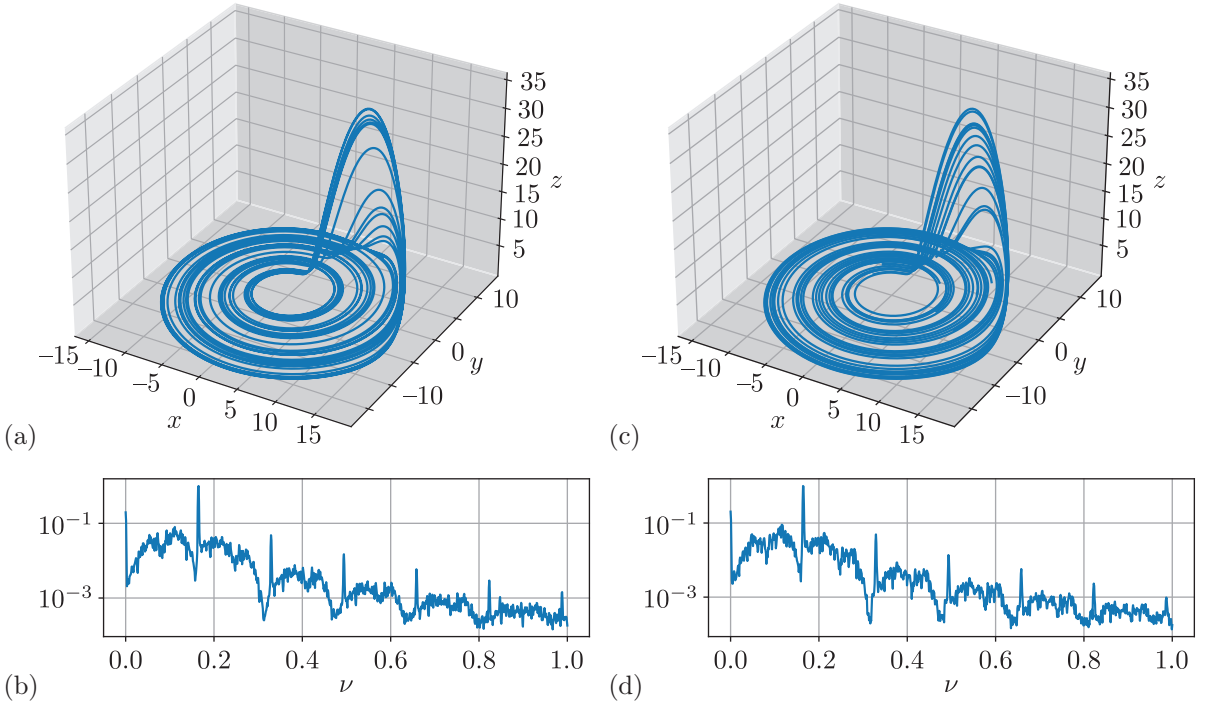

Fig. 5. Rössler chaotic attractor at $a = 0.15$, $b = 0.2$, and $c = 10$, panels (a) and (c) and the corresponding Fourier spectra, panels (b) and (d). Panels (a) and (b): numerical solution of ODEs (4.7). Panels (c) and (d): iterations of a trained network model.

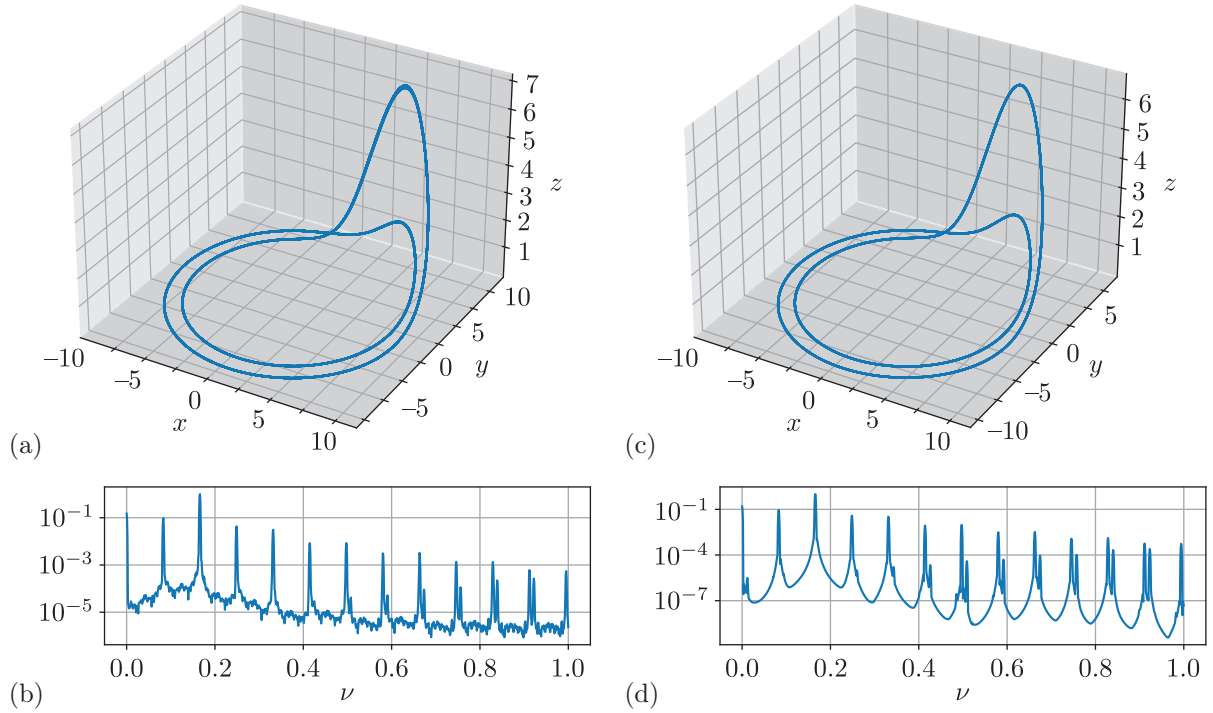

Fig. 6. Period 2 oscillations of the Rössler system at $a = 0.1$, $b = 0.1$, and $c = 6$. The limit cycle (a) and the Fourier spectrum (b) are computed for ODEs and the corresponding panels (c) and (d) are obtained for the network model.

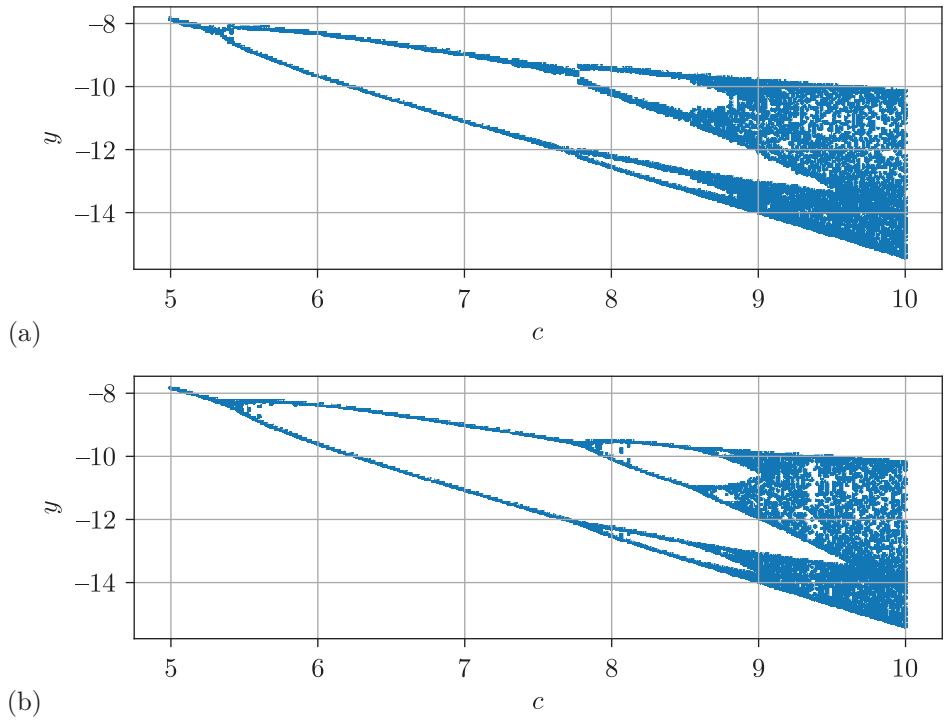

Fig. 7. Bifurcation diagrams for the Rössler system at $a = 0.1$ and $b = 0.1$. Panel (a) corresponds to a numerical solution of ODEs (4.7) and panel (b) is computed for the network model. Bifurcation diagrams are obtained as y values at Poincaré sections at $x = 0$. The sections are computed for linearly interpolated time series.

system. Parameters a and b are fixed and c is varying. For each c we compute a trajectory and then find its Poincaré section at $x = 0$. Absent values of variables between the time discretization points are obtained via linear interpolation. The diagram obtained for the numerical solution of (4.7), see Fig. 7a, is reproduced very well by the network model, see Fig. 7b. Notice, however, that the bifurcation points for the network model are slightly shifted to the right.

Let us now compare Lyapunov exponents by applying the standard algorithm for ODEs (4.7) and for the corresponding network model. We demonstrate two cases. For parameters $a = 0.15$, $b = 0.2$, $c = 10$ the Lyapunov exponents λ_i for ODEs are shown in Eq. (4.9). For comparison the exponents $\widetilde{\lambda}_i$ for the corresponding network model at the same parameters are shown in Eq. (4.10). The values coincide very well. Since the system considered is autonomous, the value of λ_2 must be zero. Actually computed values are indeed very close to zero.

$$\lambda_1 = 0.0886, \qquad \lambda_2 = -8.66 \times 10^{-7}, \qquad \lambda_3 = -9.80, \tag{4.9}$$

$$\widetilde{\lambda}_1 = 0.0839, \qquad \widetilde{\lambda}_2 = 2.70 \times 10^{-5}, \qquad \widetilde{\lambda}_3 = -9.64. \tag{4.10}$$

One more example is considered at $a = 0.1$, $b = 0.1$, $c = 13$, for which the Rössler system also has a chaotic attractor. From Eqs. (4.11) and (4.12) we again observe that the exponents for the network model $\widetilde{\lambda}_i$ are close to those obtained for the numerical solution of ODEs λ_i.

$$\lambda_1 = 0.0116, \qquad \lambda_2 = 1.87 \times 10^{-5}, \qquad \lambda_3 = -12.8, \tag{4.11}$$

$$\widetilde{\lambda}_1 = 0.0189, \qquad \widetilde{\lambda}_2 = 8.53 \times 10^{-5}, \qquad \widetilde{\lambda}_3 = -12.8. \tag{4.12}$$

However, we must notice that the correspondence of the Lyapunov exponents for the Rössler system is not as good as for the Lorenz system, see Eqs. (4.3)–(4.6). We attribute this to the parameter mismatch observed in the bifurcation diagrams.

4.3. Hindmarch – Rose neuron

Now we consider the Hindmarsh – Rose model of neuronal activity [27, 28]:

$$\begin{aligned} \dot{x} &= y - ax^3 + bx^2 - z + I, \\ \dot{y} &= c - dx^2 - y, \\ \dot{z} &= r(s(x - \alpha) - z). \end{aligned} \tag{4.13}$$

On the whole this system has eight parameters. However, the system is often considered when six of them have standard values: $a = 1.0$, $b = 3.0$, $c = 1.0$, $d = 5.0$, $s = 4.0$, $\alpha = -1.6$. Parameters I and r are varied.

The Hindmarch – Rose model (4.13) is a simplified model for biological neurons presenting bursting oscillations. In this regime, bursts of fast spikes are followed by quiescent periods. Typical values of parameters where the bursts are observed are $I = 2.7$ and $r = 0.003$. Thus, we select the normalization to be close to these values:

$$\begin{aligned} \mu_u &= (0, -5, 2.5), & s_u &= (0.8, 2.5, 0.5), \\ \mu_p &= (0.012, 2.7), & s_p &= (0.024, 0.3), \\ \mu_g &= (-2.9, 2.8, 0.047), & s_g &= (4.2, 5.2, 0.13). \end{aligned} \tag{4.14}$$

Learning curves for the network model of the system (4.13) are shown in Fig. 8. These curves show that, unlike the two previous cases, the metrics MSE (3.17) and MRNE (3.19), panels (a) and (b), respectively, decay much faster in the course of training: it takes only 7000 epochs of training for MSE at $N_h = 100$ to reach a level of about 10^{-10}. The model at $N_h = 50$ also demonstrates a very good convergence, however, the model at $N_h = 25$ behaves much worse. Thus, we will consider a model with $N_h = 50$.

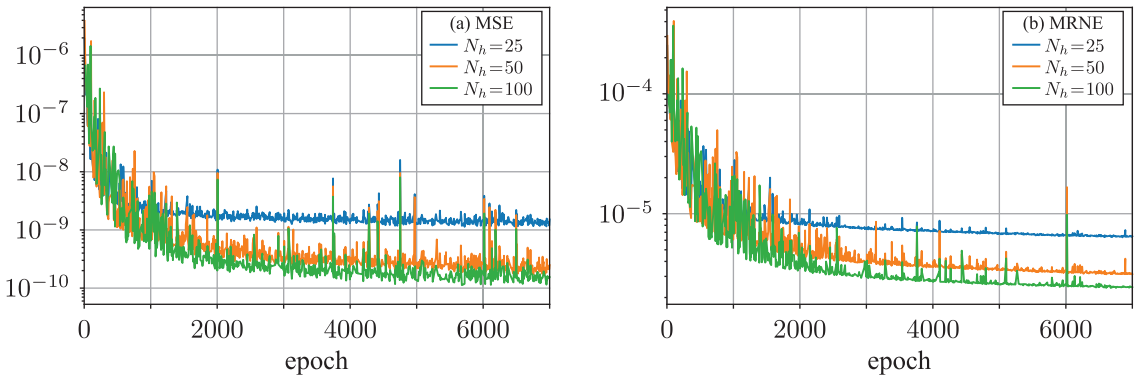

Fig. 8. Learning curves for the Hindmarch – Rose network model corresponding to ODEs (4.13).

Figures 9a and 9b demonstrate typical solutions of the Hindmarch – Rose model (4.13): panel (a) demonstrates periodical bursts of spikes, and in panel (b) we see chaotic spikes. The system contains fast and slow variables, i.e., it is stiff. To solve it numerically, the method LSODA is used [29].

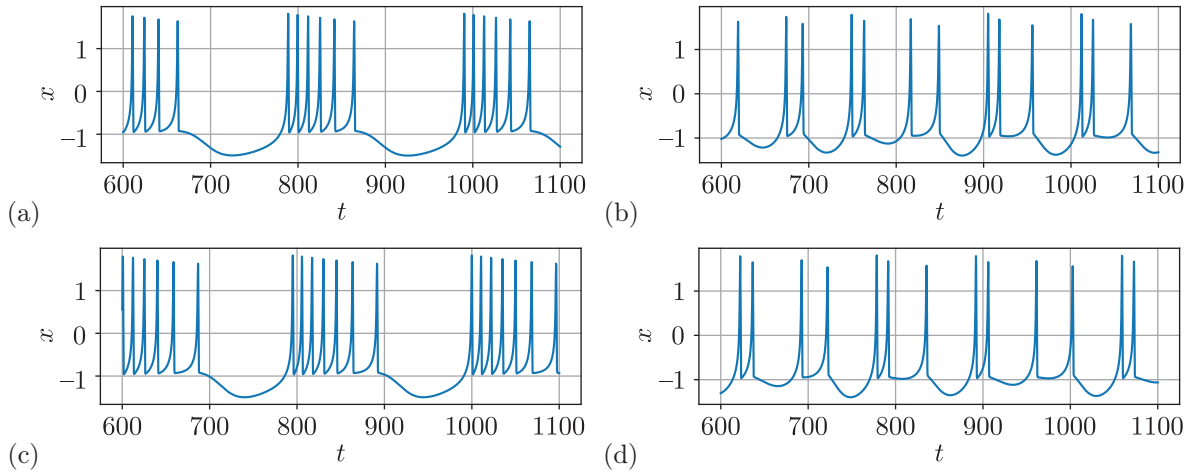

Fig. 9. Time series of $x(t)$ obtained as numerical solution of ODEs (4.13), panels (a) and (b), and corresponding iterations of the neural network model, panels (c) and (d). Parameters are $r = 0.003$, $I = 2.7$ for panels (a) and (d), and $r = 0.013$, $I = 2.9$ in panels (b) and (d).

Figures 9c and 9d show the corresponding time series obtained from the neural network model. The behavior of the network model is very similar, but the close inspection reveals that in panel (c) there are seven spikes in each burst, while the "original" curve contains only six of them. This means that, although the model demonstrates a neural dynamics as in the original ODEs, its parameters do not coincide exactly. The chaotic regimes in panels (b) and (d) obviously represent the same regime.

The bifurcation diagrams in Fig. 10a and 10b provide a more detailed comparison of the neural network model with ODEs. In both panels the diagrams are computed for Poincaré sections at $x = 0$ computed for linearly interpolated times series. The diagrams have similar global structure. One can see areas of bursts in their left parts and chaotic areas to the right. However, the detailed arrangement is different. The diagram for the neural network model looks less regular along the parameter axis. Frequent changes of the regimes are observed.

Finally, we compare Lyapunov exponents computed for numerical solutions of ODEs (4.13) and for iterations of the neural network model. For chaotic spikes at $r = 0.013$, $I = 2.9$, see Fig. 9b, the Lyapunov exponents λ_i are given by Eq. (4.15). The corresponding exponents for the network model $\widetilde{\lambda}_i$ are gathered in Eq. (4.16). The exponents are pairwise close but do not coincide:

$$\lambda_1 = 8.39 \times 10^{-3}, \qquad \lambda_2 = -1.32 \times 10^{-5}, \qquad \lambda_3 = -9.55, \tag{4.15}$$

$$\widetilde{\lambda}_1 = 7.95 \times 10^{-3}, \qquad \widetilde{\lambda}_2 = 9.77 \times 10^{-6}, \qquad \widetilde{\lambda}_3 = -9.63. \tag{4.16}$$

Similar situations are obtained for other parameter values, $r = 0.012$, $I = 2.7$: the exponents λ_i and $\widetilde{\lambda}_i$ are similar, but the difference is noticeable:

$$\lambda_1 = 5.41 \times 10^{-3}, \qquad \lambda_2 = 7.59 \times 10^{-6}, \qquad \lambda_3 = -10.2, \tag{4.17}$$

$$\widetilde{\lambda}_1 = 3.20 \times 10^{-3}, \qquad \widetilde{\lambda}_2 = 3.04 \times 10^{-5}, \qquad \widetilde{\lambda}_3 = -10.3. \tag{4.18}$$

We attribute this not very good coincidence of the Lyapunov exponents to the very weak chaos. The first exponents are very small in magnitude, so that the numerical routine converges poorly and is strongly affected by numerical errors. The parameter mismatch similar to the one observed

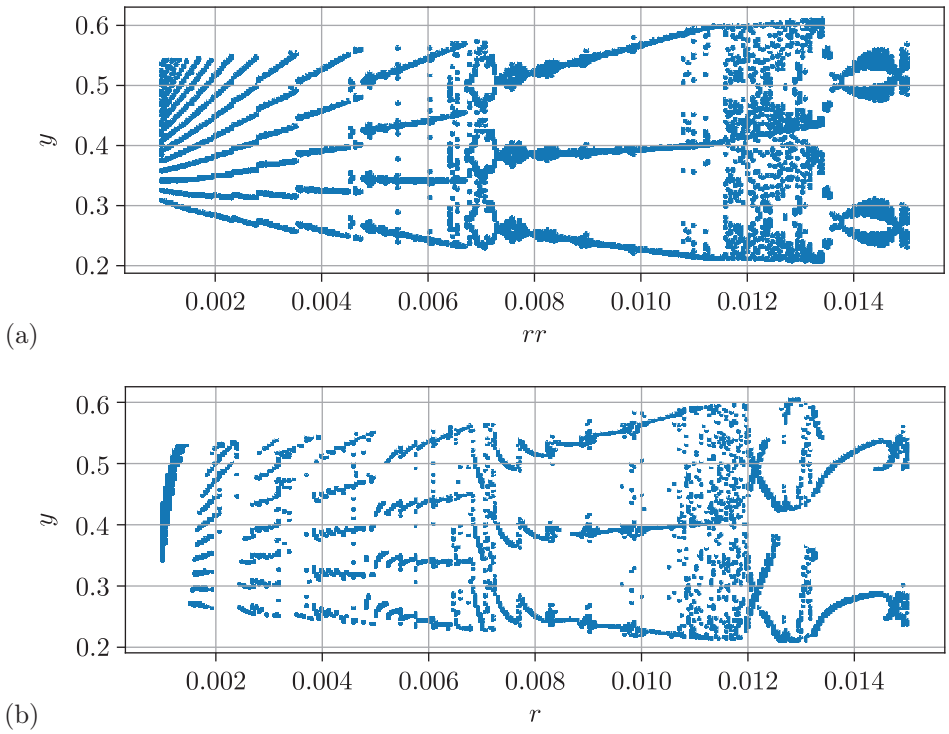

Fig. 10. Bifurcation diagrams for the Rössler system at $a = 0.1$ and $b = 0.1$. Panel (a) corresponds to a numerical solution of ODEs (4.7) and panel (b) is computed for the network model. Bifurcation diagrams are obtained as y values at Poincaré sections at $x = 0$. The sections are computed for linearly interpolated time series.

above for the Rössler system also contributes to the not very good coincidence of the Lyapunov exponents.

Thus, we observe that the discussed neural network model for the Hindmarch – Rose system provides good qualitative approximation of this system, however, the quantitative correspondence is not high.

5. Conclusion

We have discussed a neural network, a perceptron with one hidden level, which can be trained to model the behavior of various dynamical systems given by ODEs. Mathematically our universal neural network model is a discrete time system, see (3.23). We are aware of the recent success in using so-called deep networks. In contrast, our network is not deep. We prefer it because there is rigorous mathematical evidence, the universal approximation theorem [6], that the network with such architecture is able to approximate various dependencies. Another reason to apply a classical perceptron is its simple structure. We believe that it will help to trigger new theoretical studies of dynamical systems. From the practical point of view this simple network can be effectively simulated using so-called AI accelerators, hardware designed to deal with artificial neural networks. The approach developed in this paper can be considered as an alternative numerical method for modeling a dynamical system that is able to utilize modern parallel hard- and software.

The universal network model was trained to reproduce the dynamics of the three systems: the Lorenz and Rössler systems and the Hindmarch – Rose model. It was very successful for the Lorenz system. This is confirmed by visual inspection of attractors, and by coincidence of Fourier spectra and Lyapunov exponents. For the Rössler system the correspondence is also high. However, a certain mismatch of bifurcation points is observed in the bifurcation diagrams computed for the numerical solution of Rössler ODEs and for the network model.

For the Hindmarch – Rose system, good qualitative correspondence is achieved however, quantitative characteristics are sometimes different. This system allows one to reveal the limitations of the suggested approach. The Hindmarch – Rose system is stiff and also its regimes change fast within a narrow range parameters. Probably for such cases like this system a more subtle approach is required.

References

[1] Kolmogorov, A. N., On the Representation of Continuous Functions of Several Variables by Superpositions of Continuous Functions of a Smaller Number of Variables, *Amer. Math. Soc. Transl. (2),* 1961, vol. 17, pp. 369–373; see also: *Dokl. Akad. Nauk SSSR (N.S.),* 1956, vol. 108, pp. 179–182.

[2] Kolmogorov, A. N., On the Representation of Continuous Functions of Many Variables by Superposition of Continuous Functions of One Variable and Addition, *Amer. Math. Soc. Transl. (2),* 1963, vol. 28, pp. 55–59; see also: *Dokl. Akad. Nauk SSSR,* 1957, vol. 114, pp. 953–956.

[3] Arnol'd, V. I., On Functions of Three Variables, *Amer. Math. Soc. Transl. (2),* 1963, vol. 28, pp. 51–54; see also: *Dokl. Akad. Nauk SSSR,* 1957, vol. 114, pp. 679–681.

[4] Kůrková, V., Kolmogorov's Theorem and Multilayer Neural Networks, *Neural Netw.,* 1992, vol. 5, no. 3, pp. 501–506.

[5] Hornik, K., Stinchcombe, M., and White, H., Multilayer Feedforward Networks Are Universal Approximators, *Neural Netw.,* 1989, vol. 2, no. 5, pp. 359–366.

[6] Cybenko, G., Approximation by Superpositions of a Sigmoidal Function, *Math. Control Signals Syst.,* 1989, vol. 2, no. 4, pp. 303–314.

[7] Hecht-Nielson, R.., Kolmogorov's Mapping Neural Network Existence Theorem, in *Proc. of the IEEE 1st Internat. Conf. on Neural Networks (San Diego, Calif.): Vol. 3,* Piscataway, N.J.: IEEE, 1987, pp. 11–13.

[8] Funahashi, K., On the Approximate Realization of Continuous Mappings by Neural Networks, *Neural Netw.,* 1989, vol. 2, no. 3, pp. 183–192.

[9] Light, W., Ridge Functions, Sigmoidal Functions and Neural Networks, in *Approximation Theory VII,* E. W. Cheney, C. K. Chui, L. L. Schumaker (Eds.), Boston: Acad. Press, 1992, pp. 158–201.

[10] Haykin, S., *Neural Networks and Learning Machines,* 3rd ed., New York: Pearson, 2009.

[11] Kline, M., *Mathematical Thought from Ancient to Modern Times: Vol. 2,* New York: Oxford Univ. Press, 1990.

[12] Kingma, D. P. and Ba, J., Adam: A Method for Stochastic Optimization, arXiv:1412.6980 (2014).

[13] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Zh., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mane, D., Monga, R., Moore, Sh., Murray, D., Olah, Ch., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X., TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems, https://www.tensorflow.org/ (2015).

[14] Nickolls, J., Buck, I., Garland, M., and Skadron, K., Scalable Parallel Programming with CUDA, *ACM Queue,* 2008, vol. 6, no. 2, pp. 42–53.

[15] Benettin, G., Galgani, L., Giorgilli, A., and Strelcyn, J.-M., Lyapunov Characteristic Exponents for Smooth Dynamical Systems and for Hamiltonian Systems: A Method for Computing All of Them: P. 1: Theory, *Meccanica,* 1980, vol. 15, pp. 9–20.

[16] Shimada, I. and Nagashima, T., A Numerical Approach to Ergodic Problem of Dissipative Dynamical Systems, *Prog. Theor. Phys.,* 1979, vol. 61, no. 6, pp. 1605–1616.

[17] Zhang, G. P., Neural Networks for Time-Series Forecasting, in *Handbook of Natural Computing,* G. Rozenberg, Th. Bäck, J. N. Kok (Eds.), Berlin: Springer, 2012, pp. 461–477.

[18] Lewis, N. D., Deep Time Series Forecasting with Python: An Intuitive Introduction to Deep Learning for Applied Time Series Modeling, Scotts Valley, Calif.: Createspace, 2016.

[19] Brownlee, J., *Deep Learning for Time Series Forecasting: Predict the Future with MLPs, CNNs and LSTMs in Python,* San Francisco, Calif.: Machine Learning Mastery, 2018.

[20] Wei, Y., Zhou, J., Wang, Y., Liu, Y., Liu, Q., Luo, J., Wang, C., Ren, F., and Huang, L., A Review of Algorithm & Hardware Design for AI-Based Biomedical Applications, *IEEE Trans. Biomed. Circuits Syst.,* 2020, vol. 14, no. 2, pp. 145–163.

[21] Talib, M. A., Majzoub, S., Nasir, Q., and Jamal, D., A Systematic Literature Review on Hardware Implementation of Artificial Intelligence Algorithms, *J. Supercomput.,* 2021, vol. 77, pp. 1897–1938.

[22] Lorenz, E. N., Deterministic Nonperiodic Flow, *J. Atmos. Sci.,* 1963, vol. 20, no. 2, pp. 130–141.

[23] Sparrow, C., The Lorenz Equations: Bifurcations, Chaos, and Strange Attractors, Berlin: Springer, 1982.

[24] Schuster, H. G. and Just, W., *Deterministic Chaos: An Introduction,* 4th ed., rev. and enl., Weinheim: Wiley-VCH, 2005.

[25] Rössler, O. E., An Equation for Continuous Chaos, *Phys. Lett. A,* 1976, vol. 57, no. 5, pp. 397–398.

[26] Kuznetsov, S. P., *Dynamical Chaos,* 2nd ed., Moscow: Fizmatlit, 2006 (Russian).

[27] Hindmarsh, J. L. and Rose, R. M., A Model of Neuronal Bursting Using Three Coupled First Order Differential Equations, *Proc. R. Soc. Lond. Ser. B Biol. Sci.,* 1984, vol. 221, no. 1222, pp. 87–102.

[28] Wang, X.-J., Genesis of Bursting Oscillations in the Hindmarsh – Rose Model and Homoclinicity to a Chaotic Saddle, *Phys. D,* 1993, vol. 62, no. 1, pp. 263–274.

[29] Petzold, L., Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations, *SIAM J. Sci. Statist. Comput.,* 1983, vol. 4, no. 1, pp. 136–148.

[30] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A., *Automatic Differentiation in PyTorch,* in Proc. of the 31st Conf. on Neural Information Processing Systems (NIPS, Long Beach, Calif., USA, 2017), 4 pp.