Solving Differential Equations Using Feedforward Neural Networks

Wilson Guasti Junior[2(✉)] and Isaac P. Santos[1,2]

[1] Department of Applied Mathematics, Federal University of Espírito Santo, São Mateus, ES, Brazil
isaac.santos@ufes.br

[2] Graduate Program in Informatics, Federal University of Espírito Santo, Vitória, ES, Brazil

Abstract. In this work we explore the use of deep learning models based on deep feedforward neural networks to solve ordinary and partial differential equations. The illustration of this methodology is given by solving a variety of initial and boundary value problems. The numerical results, obtained based on different feedforward neural networks structures, activation functions and minimization methods, were compared to each other and to the exact solutions. The neural network was implemented using the Python language, with the Tensorflow library.

Keywords: Artificial neural networks · Differential equations · Deep learning · TensorFlow

1 Introduction

Differential equations play an important role in various fields of science and technology, modeling real-world problems of great interest in society. In fact, a variety of problems present in engineering, physics, economics, biology, epidemiology, etc. can be described using differential equations. Analytical solutions for mathematical models governed by differential equations are given only for simple problems. Thus, numerical methods are widely employed in the solution of this kind of problem. Classical numerical methods are based on the discretization of the domain at a finite number of points or elements, so that the approximate solution is defined at these points. Approximate solutions between these points must be interpolated. A drawback of the traditional methods, such as finite difference and standard finite element methods, is the need to use well-refined meshes to obtain better solutions. The computational cost of this strategy can grow a lot, preventing its use in high-dimensional problems [2,9,17].

Artificial neural networks (NNs) are models of machine learning, which works like neurons in the human brain. NNs have received a lot of attention in recent years, due to their success in numerous important applications. Mathematically, an ANN can be defined as a directed graph, where the vertices represent

© The Author(s) 2021
O. Gervasi et al. (Eds.): ICCSA 2021, LNCS 12952, pp. 385–399, 2021.
https://doi.org/10.1007/978-3-030-86973-1_27

the neurons and the edges represent the connections between them. There are many variants of neural networks that differ in their architecture (how neurons are connected). Deep neural networks and deep learning models refer to neural networks with many hidden layers hidden in the context of machine learning. One of the advantages of adding hidden layers to a neural network is the exponential reduction in computational cost in some applications and the exponential decrease in the amount of training data needed in the network learning process [3].

Since the nineties, many studies have been dedicated to the application of NN in solving differential equations. In [15,19] the authors highlighted some properties of the NN solution in comparison with conventional numerical methods, such as: the solution is continuous and differentiable over all the domain; the propagation of rounding errors, common in classic numerical methods, does not affect the solution; the computational complexity does not increase considerably with the number of discretization points and with the number of dimensions; the method can be applied in the solution of linear, nonlinear, steady or unsteady ordinary or partial equations. Another motivation in using neural networks to solve differential equations is its property of approximating continuous functions. The universal approximation theorem [7,13] states that NN feedforward can approximate any continuous function in a compact set. According to [13], neural networks feedforward are universal approximators, so, at least in theory, these networks can provide a solution to any differential equation, with an arbitrarily small margin of error. In [8] the author pointed out some benefits and drawbacks of solving partial differential equations with neural networks.

In the last four years, several works have appeared in the literature addressing solutions of differential equations through deep neural networks or deep learning models [5,11,20,23,24]. Deep learning-based methods have been shown to be effective in solving mathematical models involving differential equations, especially high-dimensional problems. The success of deep learning in important learning tasks may be related to the development of efficient computational tools, such as the Tensorflow library [1], which provides some facilities, such as automatic differentiation, activation functions and minimization methods.

In the present work we explore the use of feedforward neural networks in the solution of boundary and initial value problems, involving ordinary and partial differential equations. The neural networks were implemented using the Python language, with the Tensorflow library. We applied this methodology in the solution of some problems to evaluate the quality of the solutions obtained. In the first experiment we compare the NN solution of an ordinary differential equation with the finite difference methods, Explicit Euler and second order Runge-Kutta, and with the exact solution. Then, we apply different feedforward neural networks structures, activation functions and minimization methods for solving a elliptic problem (a two-dimensional Poisson equation) and a parabolic problem (a Heat equation), comparing its result to each other and to the exact solutions. The remainder of this work is organized as follows. In Sect. 2 we present

the definition of feedforward neural networks. The methodology studied is briefly addressed in Sect. 3 and the numerical experiments are presented in Sect. 4. Finally, the conclusions are presented in Sect. 5.

2 Feedforward Neural Networks

Feedforward neural networks offer a promising approach for solving differential equations. Consider a deep neural network composed of $L+1$ layers, where the input layer is denoted by 0 and the output layer is L. The intermediate layers between the input and the output layers are the so called hidden layers. The ℓ-th layer, $\ell = 0, 1, \cdots, L$, contains N_ℓ neurons. In particular, $N_L = 1$, that is, the output layer has only one neuron. Each neuron in the current layer ℓ receives each output from layer $\ell-1$ as input and forwards its own output to each neuron in the next layer. The weighted input in each neuron j, $j = 1, \cdots, N_\ell$, in layer ℓ, for $\ell = 1, \cdots, L$, is given by the recursive expression

$$z_j^\ell = \sum_{i=1}^{N_{\ell-1}} w_{ji}^\ell \phi_{\ell-1}(z_i^{\ell-1}) + b_j^\ell, \tag{1}$$

where w_{ji}^ℓ is the weight between neuron i in layer $\ell-1$ and neuron j in layer ℓ, b_j^ℓ is the bias of neuron j in layer ℓ and $\phi_\ell : \mathbb{R} \longrightarrow \mathbb{R}$ denotes the activation function of layer ℓ. For simplicity, we write n instead of N_0. If $\boldsymbol{x} = (x_1, \cdots, x_n) \in \mathbb{R}^n$ is the vector input, then

$$\phi_0(z_i^0) = x_i, \quad i = 1, \cdots, n.$$

Thus, the output of the neural network is given by

$$y = N(\boldsymbol{x}, \boldsymbol{p}) = z^L(\boldsymbol{x}, \boldsymbol{p}), \tag{2}$$

where z^L is obtained from Eq. (1) and $\boldsymbol{p} \in \mathbb{R}^{n_p}$ is the vector of all adjustable parameters, w_{ij}^ℓ and b_i^ℓ, with n_p denoting the total number of parameters. Figure 1 shows a neural network with one input layer with N_0 units, a unique hidden layer with N_1 neurons and one output layer with only one unit.

In Eq. (1) we assumed that a single activation function is used in ℓ-th layer, $\ell = 1, \cdots, L-1$. The activation function of a node defines the output of that node given an input or set of inputs. A neural network without the activation function would be just a linear transformation. There are several types of activation functions, for example, the sigmoid, the hyperbolic tangent and the Rectified Linear Unit (RELU) functions [16].

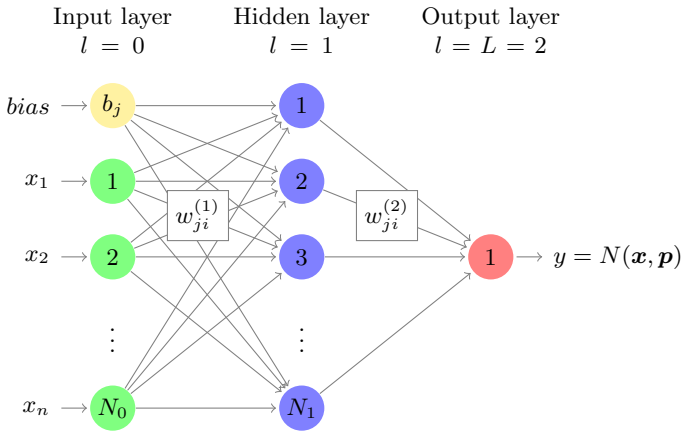

Fig. 1. Feedforward neural network with one hidden layer.

3 Description of the Method

The process of solving differential equations via artificial neural networks consists of transforming the original model into a quadratic functional (loss function), associated to the residual of the equation (and boundary and initial conditions), that is minimized using techniques of unrestricted optimization in terms of the weights and biases of the neural network [15].

Consider a mathematical model, described by a parabolic partial differential equation with Dirichlet boundary condition, that consists in finding a scalar function $u : \Omega \times (0, T_f] \longrightarrow \mathbb{R}$, such that,

$$\frac{\partial u(\boldsymbol{x}, t)}{\partial t} + \mathcal{L}u(\boldsymbol{x}, t) = f(\boldsymbol{x}, t), \quad \forall (\boldsymbol{x}, t) \in \Omega \times (0, T_f], \tag{3}$$

$$u(\boldsymbol{x}, t) = g(\boldsymbol{x}, t), \quad \forall (\boldsymbol{x}, t) \in \Gamma \times (0, T_f], \tag{4}$$

$$u(\boldsymbol{x}, 0) = u_0(\boldsymbol{x}), \quad \forall \boldsymbol{x} \in \Omega, \tag{5}$$

where $\Omega \subset \mathbb{R}^d$ is a rectangular domain with boundary Γ, $(0, T_f]$ is the time interval, with $T_f > 0$, $\boldsymbol{x} = (x_1, \cdots, x_n) \in \Omega$, f and g are given functions (f represents the source/sink term and g the Dirichlet boundary condition), u_0 is the initial solution and $\mathcal{L}(\cdot)$ is a second order differential operator.

The NN method for solving (3)–(5) seeks a function u_{NN} that approximates the exact solution u, minimizing a loss function defined by

$$L(\boldsymbol{p}) = \sum_{\boldsymbol{x}_i \in \mathcal{P}_\Omega} \sum_{t_j \in \mathcal{P}_t} \left(\frac{\partial u_{NN}(\boldsymbol{x}_i, t_j, \boldsymbol{p})}{\partial t} + \mathcal{L}(u_{NN}(\boldsymbol{x}_i, t_j, \boldsymbol{p})) - f(\boldsymbol{x}_i, t_j) \right)^2, \tag{6}$$

where $\mathcal{P}_\Omega = \{\boldsymbol{x}_i \in \Omega; i = 1, \cdots, n_\Omega\}$ and $\mathcal{P}_t = \{t_j \in [0, T_f]; j = 1, \cdots, n_t\}$ are discretizations of the spatial and temporal domains, respectively, and $\boldsymbol{p} \in \mathbb{R}^{n_p}$ is

the vector of adjustable parameters (weights and biases) of the neural network. The approximate solution u_{NN} is written as

$$u_{NN}(\boldsymbol{x}, t, \boldsymbol{p}) = G(\boldsymbol{x}, t) + Q(\boldsymbol{x}, t, N(\boldsymbol{x}, t, \boldsymbol{p})),$$

where the function $G(\boldsymbol{x}, t)$ is defined a priori in order to satisfy the boundary and initial conditions, that is,

$$G(\boldsymbol{x}, t) = g(\boldsymbol{x}, t), \quad \forall (\boldsymbol{x}, t) \in \Gamma \times (0, T_f], \tag{7}$$

$$G(\boldsymbol{x}, 0) = u_0(\boldsymbol{x}), \quad \forall \boldsymbol{x} \in \Omega. \tag{8}$$

In general, the function Q is written as

$$Q(\boldsymbol{x}, t, N(\boldsymbol{x}, t, \boldsymbol{p})) = \Phi(\boldsymbol{x}, t) N(\boldsymbol{x}, t, \boldsymbol{p}),$$

where $\Phi(\boldsymbol{x}, t) = 0$, for all $(\boldsymbol{x}, t) \in (\Omega \times \{0\}) \cup (\Gamma \times [0, T_f])$ and $N(\boldsymbol{x}, t, \boldsymbol{p})$ is a feedforward neural network with parameters $\boldsymbol{p}$ (see Eq. 2). For a given problem, there are various ways to construct the functions $G(\boldsymbol{x}, t)$ and $\Phi(\boldsymbol{x}, t)$ (see [15]).

The derivatives of the function u_{NN} in relation to the network inputs are calculated via backpropagation algorithm [22], a special automatic differentiation technique. TensorFlow includes a reverse-mode automatic differentiation library that calculates all derivatives of the loss function, producing a new symbolic expression representing the gradients [1,25].

The goal is to find $\boldsymbol{p}^* = \arg\min_{\boldsymbol{p} \in \mathbb{R}^{n_p}} L(\boldsymbol{p})$, so that $u_{NN}(\boldsymbol{x}, t, \boldsymbol{p}^*)$ is the NN solution of (3)–(5). Several optimization methods can be used to minimize the loss function (6), such as, Gradient Descent [6], Stochastic Gradient Descent (SGD) [21], Levenberg-Marquardt [12], Broyden-Fletcher-Goldfarb-Shanno (BFGS) [10], Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) [18], ADAM (adaptive moment estimation) [14], etc. A review about optimization methods for machine learning is presented in [4]. Several of these methods are implemented in TensorFlow.

4 Numerical Experiments

In this section we study three problems: an initial value problem, the Poisson problem and the Heat equation using the methodology presented in this work. The approximation results of the neural networks are compared with the true solutions. The neural networks has been implemented using the open-source TensorFlow framework. In all experiments we used the learning rate $\eta = 0,001$, the hyperbolic tangent (Tanh) activation function and the minimization method ADAM [14], with 10000 iterations. In the text, N_x, N_y and N_t stand for the number of partition points of the domain on directions x, y and t, respectively.

4.1 Ordinary Differential Equation (ODE)

We consider the following linear initial value problem (described in [15]) consisting of finding $u : [0, 1] \to \mathbb{R}$, satisfying

$$\frac{du(x)}{dx} + g(x)u(x) = q(x), \quad x \in (0, 1], \tag{9}$$

$$u(0) = 1, \tag{10}$$

with $g(x) = \left(x + \frac{1+3x^2}{1+x+x^3}\right)$, $q(x) = x^3 + 2x + x^2\left(\frac{1+3x^2}{1+x+x^3}\right)$, whose exact solution is

$$u(x) = \frac{-e^{\frac{x^2}{2}}}{(1 + x + x^3) + x^2}.$$

The approximate solution for this problem is defined as

$$u_{NN}(x) = u_{NN}(x, \boldsymbol{p}) = u_0 + (x - x_0)N(x, \boldsymbol{p}) = 1 + xN(x, \boldsymbol{p})$$

and the loss function,

$$L(\boldsymbol{p}) = \sum_{x_j \in \mathcal{P}_x} \left[\frac{du_{NN}(x_j, \boldsymbol{p})}{dx} + g(x_j)u_{NN}(x_j, \boldsymbol{p}) - q(x_j)\right]^2,$$

where $\mathcal{P}_x$ is a uniform discretization grid of the interval $[0, 1]$.

The Table 1 presents the errors $e = u - u_{NN}$, in the infinity and $L^2(0, 1)$ norms, as well as the minimal value of the loss function. We used an NN with one hidden layer with six different numbers of neurons (5, 10, 30, 50, 100, 500) and two discretization grids, with $N_x = 10$ and $N_x = 20$. Using $N_x = 10$, the minimum error in both norms were obtained with 30 neurons in the hidden layer. Increasing the number of training points to $N_x = 20$ the errors do not decrease substantially. In this case, the minimum error was obtained with 10 neurons in the hidden layer. The Table 2 considers two hidden layers with the same number of neurons in each one. We observed that the increase in the number of hidden layers resulted in a decrease in the error. The minimum errors in both discretization grids were obtained with 10 neurons in each hidden layer.

In order to compare the NN solution with the Explicit Euler (EE) and second order Runge-Kutta (RK2) methods, the Table 3 presents the errors in the infinity norm for these three methodologies. We used a network with two hidden layers with 50 neurons each one, and three discretization grids, with $N_x = 10, 20, 40$. The results show that the quality of the NN solution is compared to the RK2 solution, as pointed out in Fig. 2.

Table 1. Errors ($e = u - u_{RN}$) using the infinity and the $L^2(0,1)$ norms and minimum value of the loss function J_R, considering 6 different numbers of neurons in one hidden layer and two discretization grids, with $N_x = 10$ e $N_x = 20$.

Neurons	$N_x = 10$			$N_x = 20$		
	$\|e\|_\infty$	$\|e\|_{L^2(0,1)}$	$\min J_R$	$\|e\|_\infty$	$\|e\|_{L^2(0,1)}$	$\min J_R$
5	$3,773 \times 10^{-3}$	$5,951 \times 10^{-3}$	$5,366 \times 10^{-7}$	$2,851 \times 10^{-3}$	$6,361 \times 10^{-3}$	$1,878 \times 10^{-6}$
10	$2,320 \times 10^{-3}$	$3,880 \times 10^{-3}$	$1,052 \times 10^{-6}$	$1,673 \times 10^{-3}$	$3,785 \times 10^{-3}$	$3,645 \times 10^{-7}$
30	$1,673 \times 10^{-3}$	$3,785 \times 10^{-3}$	$3,645 \times 10^{-7}$	$2,676 \times 10^{-3}$	$5,820 \times 10^{-3}$	$6,203 \times 10^{-8}$
50	$4,025 \times 10^{-3}$	$6,161 \times 10^{-3}$	$2,300 \times 10^{-5}$	$2,703 \times 10^{-3}$	$5,861 \times 10^{-3}$	$3,875 \times 10^{-7}$
100	$4,461 \times 10^{-3}$	$6,954 \times 10^{-3}$	$9,293 \times 10^{-8}$	$3,026 \times 10^{-3}$	$6,570 \times 10^{-3}$	$2,687 \times 10^{-6}$
500	$4,356 \times 10^{-3}$	$6,767 \times 10^{-3}$	$4,939 \times 10^{-8}$	$3,002 \times 10^{-3}$	$6,515 \times 10^{-3}$	$2,755 \times 10^{-6}$

Table 2. Errors ($e = u - u_{RN}$) using the infinity and the $L^2(0,1)$ norms and minimum value of the loss function J_R, considering 6 different numbers of neurons in two hidden layers and two discretization grids, with $N_x = 10$ e $N_x = 20$.

Neurons	$N_x = 10$			$N_x = 20$		
	$\|e\|_\infty$	$\|e\|_{L^2(0,1)}$	$\min J_R$	$\|e\|_\infty$	$\|e\|_{L^2(0,1)}$	$\min J_R$
5	$5,787 \times 10^{-4}$	$9,235 \times 10^{-4}$	$7,188 \times 10^{-8}$	$4,685 \times 10^{-4}$	$1,023 \times 10^{-3}$	$9,328 \times 10^{-9}$
10	$3,191 \times 10^{-4}$	$5,077 \times 10^{-4}$	$7,875 \times 10^{-10}$	$2,370 \times 10^{-4}$	$5,387 \times 10^{-4}$	$1,015 \times 10^{-9}$
30	$3,502 \times 10^{-3}$	$5,355 \times 10^{-3}$	$9,980 \times 10^{-6}$	$2,486 \times 10^{-3}$	$5,385 \times 10^{-3}$	$3,681 \times 10^{-7}$
50	$1,372 \times 10^{-3}$	$2,079 \times 10^{-3}$	$4,952 \times 10^{-7}$	$1,006 \times 10^{-3}$	$2,151 \times 10^{-3}$	$4,173 \times 10^{-7}$
100	$1,231 \times 10^{-3}$	$1,839 \times 10^{-3}$	$2,398 \times 10^{-7}$	$7,456 \times 10^{-4}$	$1,518 \times 10^{-3}$	$2,369 \times 10^{-8}$
500	$1,208 \times 10^{-3}$	$1,821 \times 10^{-3}$	$1,719 \times 10^{-7}$	$9,276 \times 10^{-4}$	$2,008 \times 10^{-3}$	$4,899 \times 10^{-9}$

Table 3. Errors in the infinite norm for the Explicit Euler, RK2 and NN methods.

Method	Discretization grids		
	$N_x = 10$	$N_x = 20$	$N_x = 40$
Explicit Euler	$1,359 \times 10^{-1}$	$6,214 \times 10^{-2}$	$2,979 \times 10^{-2}$
RK2	$3,614 \times 10^{-3}$	$7,698 \times 10^{-4}$	$1,783 \times 10^{-4}$
NN	$1,372 \times 10^{-3}$	$1,006 \times 10^{-4}$	$7,783 \times 10^{-4}$

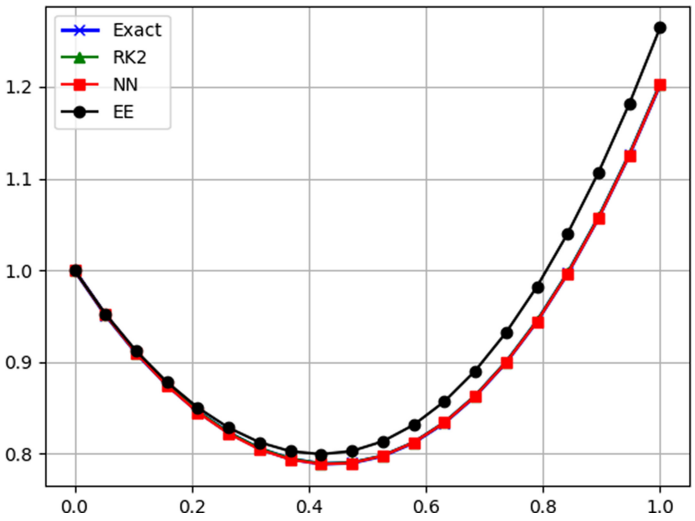

Fig. 2. Explicit Euler (EE), RK2 and NN (2 layers with 50 neurons each one) solutions, using $N_x = 20$.

4.2 The Poisson Problem

Consider the Poisson problem, consisting of finding $u : \Omega \longrightarrow \mathbb{R}$, such that,

$$-\Delta u(x,y) = f(x,y), \quad (x,y) \in \Omega, \tag{11}$$

$$u(x,y) = g(x,y), \quad (x,y) \in \Gamma, \tag{12}$$

where

$$\Delta u(x,y) = \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right),$$

$\Omega = (0,1) \times (0,1)$, Γ is the boundary of Ω, f and g are given functions.

This problem is one of the most used in the numerical study of elliptic boundary value problems. Choosing the functions $f(x,y) = 2\pi^2 \sin(\pi x)\sin(\pi y)$ in Ω and $g(x,y) = \sin(\pi x)\sin(\pi y)$ on Γ, the exact solution of (11)–(12) is

$$u(x,y) = \sin(\pi x)\sin(\pi y).$$

The approximate solution for this problem is defined as

$$u_{NN}(x,y) = u_{NN}(x,y,\boldsymbol{p}) = G(x,y) + \Phi(x,y)N(x,y,\boldsymbol{p}),$$

where

$$G(x,y) = (1-x)g(0,y) + xg(1,y) + (1-y)\Big(g(x,0) - (1-x)g(0,0) - xg(1,0)\Big)$$
$$+y\Big(g(x,1) - (1-x)g(0,1) - xg(1,1)\Big),$$
$$\Phi(x,y) = x(1-x)y(1-y).$$

The loss function is given by

$$L(\boldsymbol{p}) = \sum_{(x_i,y_i)\in\mathcal{P}_\Omega} \Big[-\Delta u_{NN}(x_i, y_i, \boldsymbol{p}) - f(x_i, y_i)\Big]^2.$$

where $\mathcal{P}_\Omega$ is a uniform discretization grid of Ω, with N_x and N_y partition points on directions x and y, respectively.

Figure 3 presents the loss function training (log scale) for the following three activation functions, sigmoid, RELU and hyperbolic tangent (Tanh), and two minimization methods, the Descent Gradient and ADAM, using $N_x = N_y = 50$ and two hidden layers with 40 neurons each one. The combination of the minimization method, ADAM, with the activation function, Tanh, obtained the best result, justifying their use in the numerical simulations. The Fig. 4 shows the exact (left) and NN (right) solutions for this Poisson problem, using a discretization grid with $N_x = N_y = 50$. The network was configured with one hidden layer and 10 neurons. Visually, the NN solution is well represented. In order to measure the quality of the NN solution, the errors in the infinite norm for several configurations of the neural network are presented as follows.

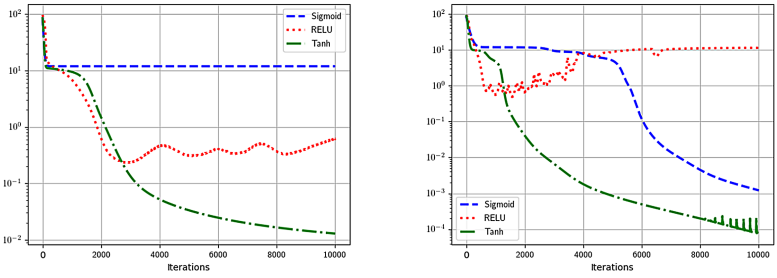

Fig. 3. Loss function training (log scale) for three activation functions and two minimization methods: Descent Gradient (left) and ADAM (right).

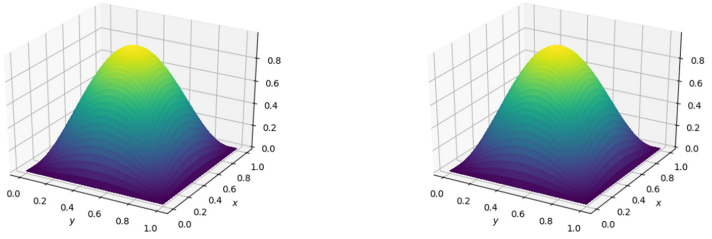

Fig. 4. Exact (left) and NN (right) solutions for the Poisson problem, using a discretization grid with $N_x = N_y = 50$. The network was configured with one hidden layer and 10 neurons.

Table 4 shows the errors in the infinite norm for different networks, all of them trained in a set of 100 points ($N_x = N_y = 10$). We consider 5 different

hidden layers and an amount of neurons (in each one) equal to 10, 20 and 40. The errors obtained with 10 or 20 neurons are of order 10^{-2}, and by adding hidden layers (up to 5) in the network, the error does not decrease. For the case of 40 neurons, using more than one hidden layer causes the error to be of order 10^{-3}. The best result is achieved for the neural network with two hidden layers and 40 neurons in each one.

We made a second experiment on the same problem, considering a set of training with 2500 points ($N_x = N_y = 50$) and an amount of neurons in each layer equal to 20, 50 and 100. The numerical results are exposed in Table 5. As expected, increasing the number of training points the error decreases (see the case with 20 neurons). The best result was an error of order 10^4, achieved for the neural network with four hidden layers and 1000 neurons in each one.

Table 4. Errors in the infinite norm, considering neural networks with up to 5 different hidden layers and a number of neurons (in each one) equal to 10, 20, 40. The training of the network was performed with $N_x = N_y = 10$ (100 points).

Hidden layers	Neurons		
	10	20	40
1	$2,601 \times 10^{-2}$	$8,075 \times 10^{-2}$	$1,563 \times 10^{-2}$
2	$2,632 \times 10^{-2}$	$8,254 \times 10^{-2}$	$5,082 \times 10^{-3}$
3	$2,592 \times 10^{-2}$	$8,131 \times 10^{-2}$	$9,738 \times 10^{-3}$
4	$2,632 \times 10^{-2}$	$8,258 \times 10^{-2}$	$5,093 \times 10^{-3}$
5	$2,510 \times 10^{-2}$	$7,979 \times 10^{-2}$	$6,382 \times 10^{-3}$

Table 5. Errors in the infinite norm, considering neural networks with up to 5 different hidden layers and a number of neurons (in each one) equal to 20, 50, 100. The training of the network was performed with $N_x = N_y = 50$ (2500 points).

Hidden layers	Neurons		
	20	50	100
1	$4,692 \times 10^{-3}$	$8,439 \times 10^{-3}$	$1,061 \times 10^{-2}$
2	$1,575 \times 10^{-3}$	$1,251 \times 10^{-3}$	$2,380 \times 10^{-3}$
3	$1,985 \times 10^{-3}$	$8,183 \times 10^{-4}$	$6,724 \times 10^{-4}$
4	$1,660 \times 10^{-3}$	$1,038 \times 10^{-3}$	$6,185 \times 10^{-4}$
5	$2,326 \times 10^{-3}$	$1,362 \times 10^{-3}$	$9,118 \times 10^{-4}$

4.3 Heat Equation

In this experiment we consider an evolutionary partial differential equation, the heat equation,

$$\frac{\partial u}{\partial t}(x,t) - \alpha \frac{\partial^2 u}{\partial x^2}(x,t) = f(x,t), \quad x \in (0,\ell),\ t > 0, \tag{13}$$

subject to Dirichlet boundary conditions,

$$u(0,t) = u_0^0(t), \quad u(\ell,t) = u_\ell^0(t), \quad \forall t > 0 \tag{14}$$

and initial condition,

$$u(x,0) = u_0(x), \quad \forall x \in [0,\ell]. \tag{15}$$

In (13), f is a given function (source term) and $\alpha > 0$ is the thermal diffusivity coefficient. Equation (13) models the temperature distribution in the one-dimensional region (a beam, for example) $[0,\ell]$, with boundary and initial conditions given by (14) and (15). Function $u(x,t)$ represents the temperature at point $x \in [0,\ell]$ and time $t \geq 0$.

Considering $f = 0$, $u_0^0(t) = u_\ell^0(t) = 0$, for all $t > 0$ and initial condition,

$$u_0(x) = \sin\left(\frac{\pi x}{\ell}\right), \quad \forall x \in [0,\ell], \tag{16}$$

the exact solution of (13), (14) and (15) is given by

$$u(x,t) = e^{\frac{-\alpha\pi^2 t}{\ell}} \sin\left(\frac{\pi x}{\ell}\right). \tag{17}$$

We solve the problem considering the space-time domain $\Omega = (0,\ell) \times (0,T_f]$, where $T_f > 0$ is the final time, so that the initial condition is treated as a boundary condition. The approximate solution is defined as

$$u_{NN}(x,t,\boldsymbol{p}) = G(x,t) + \Phi(x,t)N(x,t,\boldsymbol{p}),$$

where $G(x,t) = \frac{(\ell-t)}{\ell}u_0(x)$ and $\Phi(x,t) = x(\ell - x)t$, and the loss function is described by

$$L(\boldsymbol{p}) = \sum_{(x_i,t_i)\in\mathcal{P}_\Omega} \left[\frac{\partial u_{NN}(x_i,t_i,\boldsymbol{p})}{\partial t} - \alpha\frac{\partial^2 u_{NN}(x_i,t_i,\boldsymbol{p})}{\partial x^2} - f(x_i,t_i)\right]^2. \tag{18}$$

Considering $\ell = \alpha = 1$, the Fig. 5 shows the exact (left) and NN (right) solutions for this Heat problem, using a discretization grid with $N_x = N_t = 50$. The network was configured with two hidden layers and 20 neurons. Visually, the NN solution is well represented. Figure 7 shows the NN (in blue) and true (in red) solutions in four points in time. At t = 0, the temperature is given by the initial condition. As the time approaches to $t = 0.5$, it gradually spreads tending to zero over the whole spatial domain. Figure 6 presents the loss function training (log scale) for the activation functions, sigmoid, RELU and hyperbolic tangent (Tanh), and the minimization methods, the Descent Gradient and ADAM, using $N_x = N_t = 50$ and four hidden layers with 20 neurons each one. The minimization method, ADAM, and the activation function, Tanh, provided the best result.

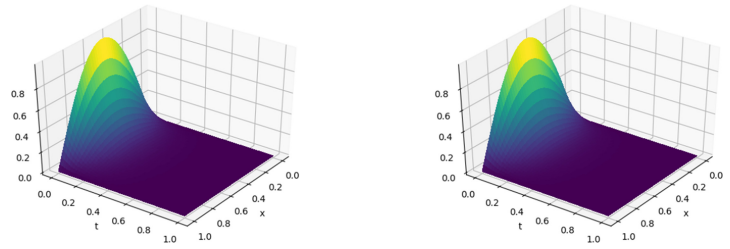

Fig. 5. Exact (left) and NN (right) solutions for the Heat problem, using a discretization grid with $N_x = N_t = 50$. The network was configured with two hidden layer and 20 neurons in each one.

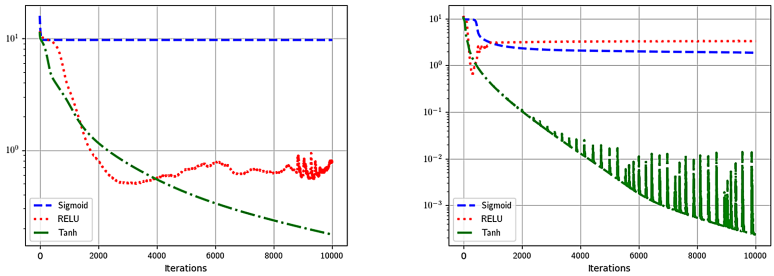

Fig. 6. Loss function training (log scale) for three activation functions and two minimization methods: Descent Gradient (left) and ADAM (right).

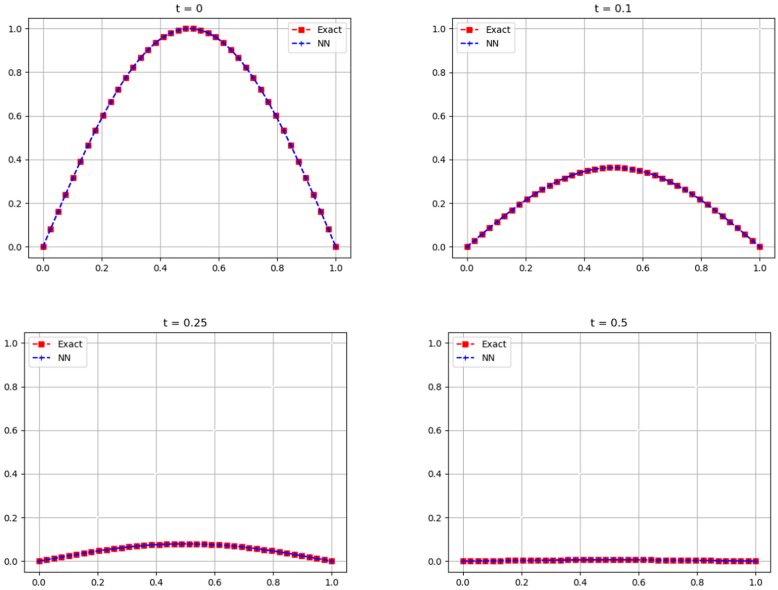

Fig. 7. Exact (in red) and NN (in blue) solutions of (13) in four points in time. (Color figure online)

In order to measure the quality of the NN solution, the errors in the infinite norm for several configurations of the neural network are presented as follows. Table 6 shows the errors in the infinite norm for various neural network configurations (hidden layers and neurons per layer). We observe that the addition of more neurons or more hidden layers did not reduce significantly the error. Using three hidden layers with twenty neurons each one is sufficient to obtain a solution with an error of order 10^{-4}.

Table 6. Infinity norm errors, considering $N_x = N_t = 50$.

Hidden layers	Neurons		
	10	20	40
1	6.982×10^{-2}	4.261×10^{-2}	3.109×10^{-2}
2	3.568×10^{-2}	2.063×10^{-3}	1.301×10^{-3}
3	1.012×10^{-3}	9.519×10^{-4}	8.899×10^{-4}
4	3.586×10^{-3}	9.418×10^{-4}	9.934×10^{-4}
5	2.567×10^{-3}	9.749×10^{-4}	9.380×10^{-4}

5 Conclusions

In this work we presented a study about solutions of differential equations using a method based on deep learning models, more specifically, feedforward deep neural networks. Numerical experiments were carried out to evaluate the efficiency of the methodology, comparing its results with the exact solutions and calculating the errors in the infinite norm. The method achieved good results due to the powerful function approximation ability of neural networks. The neural networks were implemented using the open-source TensorFlow framework.

The method proved to be a promising methodology for solving initial and boundary value problems. However, more numerical experiments need to be carried out in order to strengthen our conclusions. As future work, we plan to explore other network topologies and apply them for solving problems with curved or irregular boundaries and 3D unsteady transport problems, including the Euler and the Navier-Stokes equations.

Despite the success of these new approaches based on deep learning to solve differential equations, there are many important points involving theoretical and practical issues that deserve discussions and further investigations.

References

1. Abadi, M., et al.: TensorFlow: a system for large-scale machine learning. In: OSDI 2016, pp. 265–283. USENIX Association, USA (2016)
2. Bellman, R.: Dynamic Programming. Princeton University Press (1957)

3. Bengio, Y., Simard, P., Frasconi, P.: Learning long-term dependencies with gradient descent is difficult. IEEE Trans. Neural Netw. **5**, 157–166 (1994). https://doi.org/10.1109/72.279181
4. Bottou, L., Curtis, E.F., Nocedal, J.: Optimization methods for large-scale machine learning. SIAM Rev. **60**(2), 223–311 (2018). https://doi.org/10.1137/16M1080173
5. Chaudhari, P., Oberman, A., Osher, S., Soatto, S., Carlier, G.: Deep relaxation: partial differential equations for optimizing deep neural networks. Res. Math. Sci. **5**(3), 1–30 (2018). https://doi.org/10.1007/s40687-018-0148-y
6. Curry, H.B.: The method of steepest descent for non-linear minimization problems. Quart. Appl. Math. **2**, 258–261 (1944). https://doi.org/10.1090/qam/10667
7. Cybenko, G.: Approximation by superposition of a sigmoidal function. Math. Control Sig. Syst. **2**, 303–314 (1989)
8. Dockhorn, T.: A discussion on solving partial differential equations using neural networks. arXiv:1904.07200 (2019)
9. Weinan, E., Han, J., Jentzen, A.: Algorithms for solving high dimensional PDEs: from nonlinear Monte Carlo to machine learning. CoRR abs/2008.13333 (2020). https://arxiv.org/abs/2008.13333
10. Guo, Q., Liu, J.G., Wang, D.H.: A modified BFGS method and its superlinear convergence in nonconvex minimization with general line search rule. J. Appl. Math. Comput. **28**, 435–446 (2008). https://doi.org/10.1007/s12190-008-0117-5
11. Guo, Y., Cao, X., Bainian, L., Gao, M.: Solving partial differential equations using deep learning and physical constraints. Appl. Sci. **10**, 5917 (2020). https://doi.org/10.3390/app10175917
12. Hagan, M.T., Menhaj, M.B.: Training feedforward networks with the Marquardt algorithm. IEEE Trans. Neural Netw. **5**(6), 989–993 (1994). https://doi.org/10.1109/72.329697
13. Hornik, K., Stinchcombe, M., White, H.: Multilayer feedforward networks are universal approximators. Neural Netw. **2**(5), 359–366 (1989). https://doi.org/10.1016/0893-6080(89)90020-8
14. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization (2017)
15. Lagaris, I.E., Likas, A., Fotiadis, D.I.: Artificial neural networks for solving ordinary and partial differential equations. IEEE Trans. Neural Netw. **9**(5), 987–1000 (1998). https://doi.org/10.1109/72.712178
16. Lau, M.M., Hann Lim, K.: Review of adaptive activation function in deep neural network. In: 2018 IEEE-EMBS Conference on Biomedical Engineering and Sciences (IECBES), pp. 686–690 (2018). https://doi.org/10.1109/IECBES.2018.8626714
17. Leake, C., Mortari, D.: Deep theory of functional connections: a new method for estimating the solutions of partial differential equations. Mach. Learn. Knowl. Extr. **2**(1), 37–55 (2020). https://doi.org/10.3390/make2010004
18. Liu, D.C., Nocedal, J.: On the limited memory BFGS method for large scale optimization. Math. Program. **45**, 503–528 (1989)
19. Parisi, D.R., Mariani, M.C., Laborde, M.A.: Solving differential equations with unsupervised neural networks. Chem. Eng. Process. Process Intensif. **42**(8), 715–721 (2003). https://doi.org/10.1016/S0255-2701(02)00207-6
20. Raissi, M., Karniadakis, G.E.: Hidden physics models: machine learning of nonlinear partial differential equations. J. Comput. Phys. **357**, 125–141 (2018). https://doi.org/10.1016/j.jcp.2017.11.039
21. Robbins, H., Monro, S.: A stochastic approximation method. Ann. Math. Stat. **22**(3), 400–407 (1951). https://doi.org/10.1214/aoms/1177729586

22. Rumelhart, D.E., Willians, G.E., Williams, R.J.: Learning representations by back-propagating errors. Nature **323**, 533–536 (1986). https://doi.org/10.1038/323533a0
23. Samaniego, E., et al.: An energy approach to the solution of partial differential equations in computational mechanics via machine learning: concepts, implementation and applications. Comput. Meth. Appl. Mech. Eng. **362**, 112790 (2020). https://doi.org/10.1016/j.cma.2019.112790
24. Sirignano, J., Spiliopoulos, K.: DGM: a deep learning algorithm for solving partial differential equations. J. Comput. Phys. **375**, 1339–1364 (2018). https://doi.org/10.1016/j.jcp.2018.08.029
25. Yu, Y., et al.: Dynamic control flow in large-scale machine learning. In: Proceedings of the 13th EuroSys Conference, EuroSys 2018, Association for Computing Machinery, New York (2018)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

