axioms

MDPI

Article

Solving Inhomogeneous Constant Coefficient Ordinary Differential Equations Using Artificial Neural Networks

Laure Gouba [1,*] and Carine Ornela Megne Nono [2]

[1] The Abdus Salam International Centre for Theoretical Physics (ICTP), Strada Costiera 11, I-34151 Trieste, Italy
[2] African Institute for Mathematical Science (AIMS-Ghana), 1 Shoppers Street, Manet, Spintex–Accra, Legon, Accra P.O. Box LG DTD 20046, Ghana; carine@aims.edu.gh
* Correspondence: laure.gouba@gmail.com

Abstract

Ordinary differential equations are fundamental tools for modeling dynamic systems in science, engineering, and applied mathematics. Solving these equations accurately and efficiently is crucial, particularly in cases where analytical solutions are challenging or impossible to obtain. This paper presents a method for solving inhomogeneous linear ordinary differential equations using an artificial neural network. The network is composed of a single input layer with one neuron, one hidden layer with three neurons, and a single output layer with one neuron. A multiple regression model is employed to determine the weights from the input layer to the hidden layer, while radial basis functions are used to compute the weights from the hidden layer to the output layer. The bias values are chosen within the range of -1 to 1 to optimize learning behavior. A trial solution is constructed as a sum of two parts. One part satisfies the initial condition, and the other part is the output of the network to approximate the function. The neural network is trained to minimize the mean squared error of the residuals obtained by doing the substitution of the trial solution into the given ordinary differential equation. The methodology is tested on first-order and second-order ordinary differential equations to evaluate its accuracy, stability and how its capability can be generalized. The results show that the method can approximate the exact solutions of these ordinary differential equations with reasonable accuracy.

Keywords: ordinary differential equations; artificial neural networks; Python

MSC: 34A30; 68T07

Check for updates

Academic Editors: Andrey Zahariev and Hristo Kiskinov

Received: 2 April 2026
Revised: 20 April 2026
Accepted: 28 April 2026
Published: 1 May 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

Nowadays, ordinary differential equations (ODEs) can be solved innovatively, thanks to the development of artificial intelligence and artificial neural networks (ANNs). Traditionally, analytical or numerical methods are applied to solve these equations, which describe a wide variety of real-world phenomena in physics, biology, engineering, and economics. These analytical approaches, however, are often limited to the simplest equations, while some numerical methods may be unstable or computationally intensive under certain conditions [1]. The implementation of ANNs in the field of differential equations provides an agile, data-informed alternative that can learn and accurately model complex mappings.

Neural networks are parallel and distributed information processing systems derived from biological learning systems, such as the human brain. In the Figure 1, we present a biological neural network (BNN) which is a network of nerve cells known as neurons,

found in all living organisms [2]. Neurons are the basic signaling units of the nervous system. Signals that pass from one neuron to another carry the instructions for actions, thoughts, and learning.

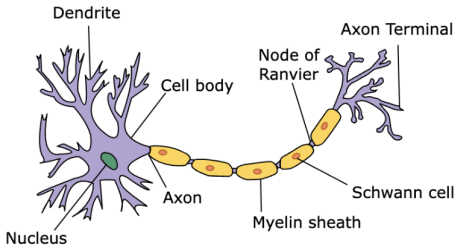


Figure 1. Biological neuron [2].

ANNs emerged from the model of a neuron proposed by Warren McCulloch and Walter Pitts in 1943 [3]. An ANN is an information processing system that has certain performance characteristics in common with a BNN. The basic component of an ANN is an artificial neuron. Like a biological neuron in a BNN, an artificial neuron is the basis processor. It receives one or more inputs over these connections and produces only one output. The artificial neurons are called units, which are arranged in a series of layers that together constitute the whole ANN.

As indicated in the Figure 2, an ANN has input layers, output layers, and hidden layers [4]. The input layer receives data from the outside world, which the neural network needs to analyze or learn about. Then this data passes through one or multiple hidden layers that transform the input into data that is used to compute the output layer. Finally, the output layer provides an output in the form of a response of ANNs to the input data provided.

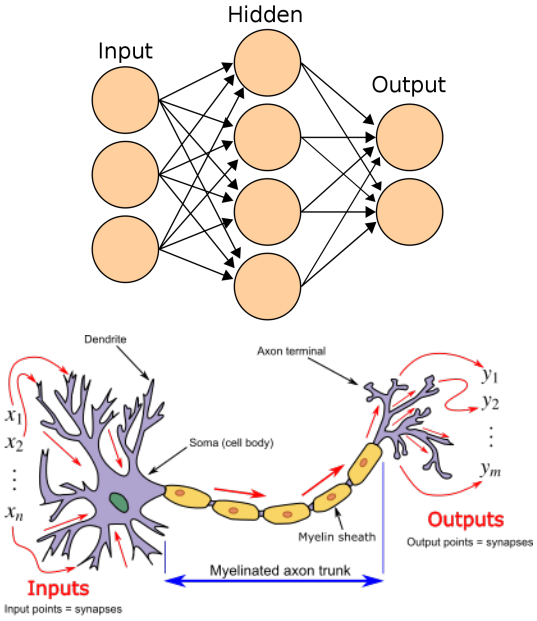


Figure 2. Example of an ANN architecture in analogy with a BNN architecture [4].

There are different neural network methods for the solution of ODEs, such as radial basis function network [5–7], cellular neural network [8], finite element neural network [9], the multilayer perceptron neural network and wavelet neural network [5,10–12]. Numerous examples of algorithms have been proposed based on these neural networks for approximating the exact solution of differential equations. For instance, in [5], Lagaris et al. presented a method for solving ODEs and partial differential equations with initial and

boundary value problems using ANNs. In [13], the authors devised a method for solving ODEs with initial conditions using the regression-based algorithm and compared the results of random and regression-based initial weights for different numbers of nodes in the hidden layer. In [14], it has been shown that the potential of differential equations for time series data analysis. A comprehensive comparison between different neural network-based methods for solving differential equations, focusing on two approaches: the natural neural network method and the Ritz neural network method, has been investigated in [15]. In [16], the authors approximated the Lotka–Volterra equations and the Van der Pol system using neural ODEs, which are a recent development that combines deep learning with the structure of differential equations. In [17], the authors also used ANN-based models for solving first and second-order linear constant-coefficient ODEs with initial conditions. In [18], a suitable method is developed for solving Bratus's and Troesch's equations, which are examples of second-order nonlinear boundary value problems of ODEs. In [19], an alternative artificial neural network approach is proposed to solve the initial value problems of ODEs. In [20], the influence of various basic neural networks, optimizers, and activation functions for solving differential equations and systems is investigated. In [21], an ANN architecture is specifically designed for solving first-order ODEs. In [22], the authors presented an in-depth review of the current state-of-the-art methodologies and breakthroughs that harness the capabilities of deep neural networks for tackling the challenges posed by ODEs. In [23], a novel method is developed to tackle the stiffness in neural ODEs.

In the present paper, we revisit the method of solving ODEs using ANNs in a more comprehensive and systematic manner by approximating the solutions of a first-order and a second-order inhomogeneous constant coefficients ordinary differential equations using an artificial neural network, and comparing the results with the exact solutions. We adopted the procedure of resolution using a trial solution as in [5] and the procedure of finding the parameters of the neural network using multiple regression and the radial basis functions approximation principle as in [17]. In Section 2, we present the case of an initial value first-order ordinary differential equation. The Section 3 is devoted to the second-order case. Concluding remarks are presented in Section 4.

2. Solving First-Order Constant Coefficients Ordinary Differential Equations Using Artificial Neural Network

2.1. Procedure of Resolution

Let's consider the initial value first-order ordinary differential equation

$$\frac{dy}{dt} = f(t, y), \quad t \in [a, b], \quad y(a) = A. \tag{1}$$

To solve the initial value problem in the Equation (1) using an ANN, we adopt the trial solution method [5], which consists of writing the trial solution as follows:

$$y_{\text{trial}}(t, \vec{p}) = A(t) + tN(t, \vec{p}), \tag{2}$$

where the term $A(t)$ is a function that satisfies the initial or boundary conditions and does not contain some adjustable parameters, the term $N(t, \vec{p})$ is the neural output of the feedforward network with input data t, and $\vec{p}$ are parameters of the neural architecture. The term $tN(t, \vec{p})$ does not affect the initial or boundary conditions. It employs a neural network whose weights and biases are to be adjusted in order to deal with the minimization problem, which is an unconstrained optimization problem, due to the choice of the form of the trial solution that satisfies the initial condition by construction [5]. In order to get better results, the neural network uses unsupervised learning to optimize the weights and

biases. To obtain better values, an error or loss function is implemented that calculates the difference between the actual value of the solution of the given ODE and the value of the trial solution. A normal error function to use is the mean squared error (MSE) given by:

$$E(\vec{p}) = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{dy_{\text{trial}}(t_i)}{dt} - f(t_i,\ y_{\text{trial}(t_i)})\right)^2, \tag{3}$$

where $t_i \in \tilde{D}$, and $\tilde{D}$ is a discretization of the interval $[a,\ b]$. The description is illustrated in the Figure 3.

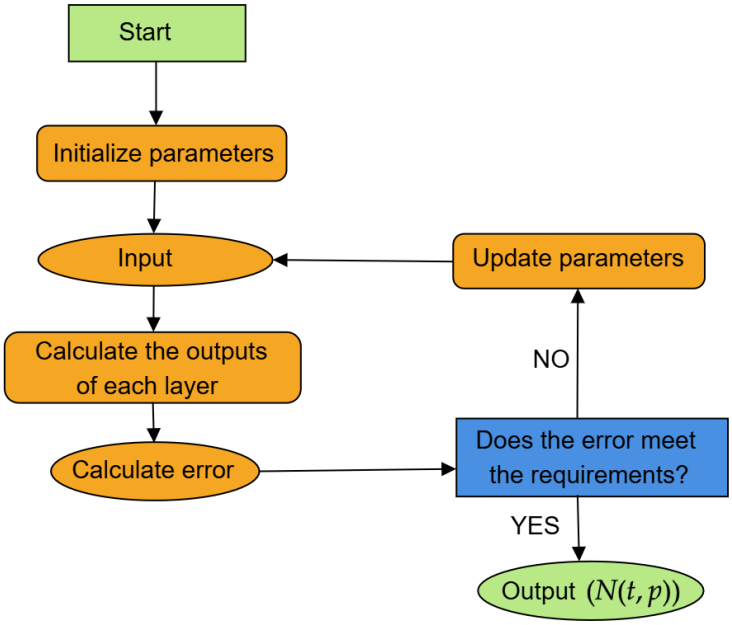


Figure 3. Two ellipses demonstrate the workflow of the method of accuracy calculation.

The neural network architecture has three layers: one input layer with one neuron, one hidden layer with three neurons, and one output layer with one output unit. In Figure 4, each neuron in the hidden layer is connected to other neurons of the input layer through adaptable synaptic weights w_{in} and biases, $u_n,\ n = 1,\ 2,\ 3$. The weights from each hidden layer to the output unit are denoted by v_n. Using this architecture, we can write the neural output as

$$N(t, \vec{p}) = \sum_{n=1}^{3} v_n \sigma(z_n), \tag{4}$$

where σ is the sigmoid activation function defined by:

$$\sigma(z_n) = \frac{1}{1+\exp(-z_n)}, \tag{5}$$

and $z_n = \sum_{i=1}^{3} t_i w_{in} + u_n$. In our particular case, we have one input layer, so $z_n = t_1 w_{1n} + u_n$. The parameters of the neural output in the Equation (4) to be determined are the weights from the input layer to the hidden layer $w_{1n},\ n = 1,\ 2,\ 3$, the weights from the hidden layer to the output layer $v_n,\ n = 1,\ 2,\ 3$, and the biases $u_n,\ n = 1,\ 2,\ 3$.

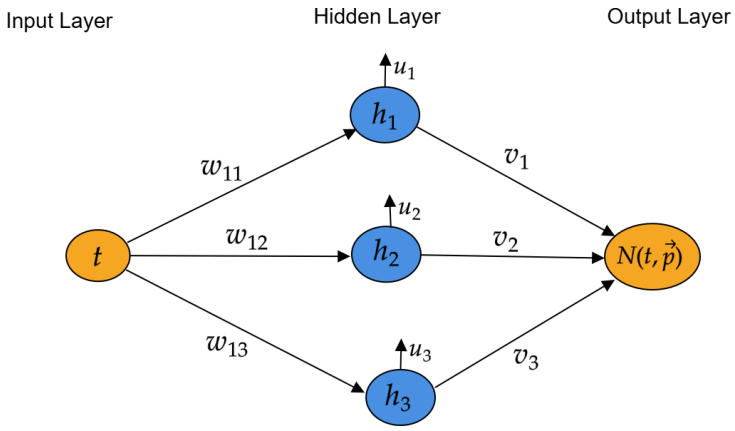


Figure 4. Neural architecture for $N(t, \vec{p})$.

2.1.1. Finding the Parameters

The parameters are determined as follows:

1. The weights w_{1n}, $n = 1,\ 2,\ 3$ are determined from a multiple regression model. This regression is a polynomial regression of the assumed solution, $y_a(t)$, of the homogeneous equation associated with the Equation (1). The degree of the polynomial regression is equal to the number of neurons in the hidden layer, which is 3 in our case. The coefficients of the polynomial regression, obtained by doing the multiple regression using Python 3.14, are used as the weights w_{1n}, $n = 1, 2, 3$. Choosing the assumed solution $y_a(t) = \exp(\lambda t)$ as in [17], a regression can be performed on it using Python to obtain the weights w_{1n}, $n = 1,\ 2,\ 3$.
2. The weights v_i, $i = 1,\ 2,\ 3$ are obtained using a real valued function $g(t)$ whose choice depends on the nature of the differential equation. For a first-order homogeneous equation, it is a linear polynomial function. In the case of an inhomogenous ordinary differential equation, $g(t)$ corresponds to the forcing term. Then we consider a real-valued vector $t = (t_1,\ t_2,\ t_3)$ and a set of radial basis function $\{\rho_n(t),\ n = 1,\ 2,\ 3\}$ of this type

$$\rho_n(t_i) = \exp\left(-\frac{||t_n - t_i||^2}{2\sigma^2}\right),\ n,\ i\ = 1,\ 2,\ 3, \tag{6}$$

such that

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} \rho_1(t_1) & \rho_2(t_1) & \rho_3(t_1) \\ \rho_1(t_2) & \rho_2(t_2) & \rho_3(t_2) \\ \rho_1(t_3) & \rho_2(t_3) & \rho_3(t_3) \end{bmatrix}^{-1} \begin{bmatrix} g(t_1) \\ g(t_2) \\ g(t_3) \end{bmatrix}, \tag{7}$$

where the variance σ^2 is a hyperparameter that defines the width of the Gaussian curve and controls the influence of a single training point. The choice of the variance is heuristic. The variance parameter determines the smoothness and localized behavior of the approximation in the sense that a smaller variance creates more peaked function while a larger variance results in a flatter function. We choose the variance of the 10 discretized points in the interval $[0, 1]$ as in [17]. More about the variance or shape parameter are discussed in [24–26].
3. The biases $u_n, n = 1,\ 2,\ 3$ are chosen between -1 and 1.

The prediction error shall be computed using the squared error function defined as follows:

$$E = \frac{1}{2}(y_d - y_p)^2, \tag{8}$$

where y_d represents the desired output and y_p the predicted output.

2.1.2. Network Training

Based on the weights given by the neural network, the predicted outcome is measured against the desired outcome. The network's weights do not need to be adjusted. If it cannot get the right output, the biases must be amended. The sigmoid function in the Equation (5) is the fundamental activation function used to train the neural network as follows:

1. The parameters of the network should first be computed, and the biases to be chosen between -1 and $+1$.
2. Then the aggregation given by $z_n = t \cdot w_{1n} + u_n$ is computed, and z_n is applied as input to the activation function in the Equation (5).
3. The neural output is then computed by using the Equation (4).

As suggested by earlier researchers [5,17], adjusting all of the network parameters to achieve convergence is not necessary when the weights of the neural network are determined in this methodical manner described above. However, convergence to a solution with an acceptable minimum error is achieved with only a small adjustment of the biases, which are fixed to lie within a specified interval. If the predicted output does not correspond to the desired output, we have to train the network to minimize the prediction error by adjusting the weights. This can be achieved as follows [5,17]:

$$w_{ij}^n = w_{ij}^o + \eta(y_d - y_p)t, \tag{9}$$

where, w_{ij}^n and w_{ij}^o represent the new and old weights respectively, η represents the learning rate, y_d represents the predicted output, and t represents the current input vector. This process continues until the predicted output corresponds to the desired output, and the predicted error is minimized.

2.2. First-Order Ordinary Differential Equation: Illustration

Let's consider the first-order initial value problem

$$y' - 2y = e^{4t}, \quad y(0) = 2, \quad t \in [0,\ 1], \tag{10}$$

whose exact solution is given by:

$$y_{Exact}(t) = \frac{1}{2}e^{4t} + \frac{3}{2}e^{2t}. \tag{11}$$

Similarly to the equation in (2), the trial solution of (10) with initial condition $A = 2$ is given as:

$$y_{trial} = 2 + tN(t, \vec{p}). \tag{12}$$

Once the trial solution is established, we have to determine the weights from the input layer to the hidden layer. To do that, we employ Python to regress the solution to the homogeneous part of the given first-order ODE in the Equation (10), which is given as:

$$y_a(t) = 2e^{2t}. \tag{13}$$

Assume that (13) is a solution to the first-order ODE in the Equation (10). We divided the interval $[0, 1]$ into 10 equidistant points, t_i, with a step of 0.1. We train the network for these points, $t_i \in [0, 1],\ i = 0,\ 1,\ 2, \ldots, 10$. The value $y_a(t)$, at these points in the domain of the given ODE, is shown in Table 1.

The weights w_{in}, from the input layer to the hidden layer, are determined from a multiple regression model. This regression is a curve-fit polynomial regression of the assumed solution, $y_a(t)$, to the homogeneous part of the first-order ODE given by the

Equation (10). The output of the linear, quadratic, and cubic regression of $y_a(t) = 2e^{2t}$ is shown below in Table 2.

Table 1. t and $y_a(t) = 2e^{2t}$.

t	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
$y_a(t)$	2.000	2.443	2.984	3.644	4.451	5.437	6.640	8.110	9.906	12.099	14.778

Table 2. First-order ordinary differential equation: model summary and parameter estimates.

Equation	R^2	Intercept	w_{11}	w_{12}	w_{13}
Linear	0.930	0.481	12.219	-	-
Quadratic	0.998	2.252	0.410	11.809	-
Cubic	1.000	1.974	4.826	0.229	7.720

From Table 2, the cubic curve shows perfect goodness of fit, since $R^2 = 1$, meaning that the model's predictions match the data exactly; every data point lies perfectly on the curve. Therefore, the weights from the hidden layer to the output layer are given as follows:

$$w_{11} = 4.826 \;\; w_{12} = 0.229 \text{ and } \;\; w_{13} = 7.720. \tag{14}$$

To compute the weights, v_n, from the hidden layer to the output layer based on our architecture of neural network, which has three neurons in its hidden layer, we determine a function $F(t)$ such that $v = \rho^{-1}g$, where $v = (v_1,\ v_2,\ v_3)^T$ is a vector of coefficients, $g = (F(t_1), F(t_2),\ F(t_3))^T$ is a vector composed of the values of the function at the 3 points chosen in the domain, and ρ is the matrix with entries given by values of the elementary function at each of the 3 points. Since the given initial value problem defined in Equation (10) is inhomogeneous, $F(t) = e^{4t}$, which is the forcing term of the ODE. Therefore, taking $t = (0.1,\ 0.2,\ 0.3)^T$, and using

$$\rho_j(t_i) = \exp\left(-\frac{\|t_i - t_j\|^2}{2\sigma^2}\right), \;\; i, j = 1, \cdots, m, \tag{15}$$

we have that

$$g = \begin{bmatrix} F(t_1) \\ F(t_2) \\ F(t_3) \end{bmatrix} = \begin{bmatrix} 1.4918 \\ 2.2255 \\ 3.3201 \end{bmatrix}, \tag{16}$$

and

$$\rho = \begin{bmatrix} \rho_1(t_1) & \rho_2(t_1) & \rho_3(t_1) \\ \rho_1(t_2) & \rho_2(t_2) & \rho_3(t_2) \\ \rho_1(t_3) & \rho_2(t_3) & \rho_3(t_3) \end{bmatrix} = \begin{bmatrix} 1 & 0.9512 & 0.8187 \\ 0.9512 & 1 & 0.9512 \\ 0.8187 & 0.9512 & 1 \end{bmatrix}. \tag{17}$$

Applying the formula $v = \rho^{-1}g$, we get:

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 1 & 0.9512 & 0.8187 \\ 0.9512 & 1 & 0.9512 \\ 0.8187 & 0.9512 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1.4918 \\ 2.2255 \\ 3.3201 \end{bmatrix} = \begin{bmatrix} 26.867 \\ -58.481 \\ 36.953 \end{bmatrix}. \tag{18}$$

Therefore, the weights from the hidden layer to the output layer are given as follows:

$$v_1 = 26.867,\ v_2 = -58.481, \text{ and } \;\; v_3 = 36.953. \tag{19}$$

We are now going to train the network with all the available parameters using the Python programming language. The biases are chosen between -1 and 1 as follows:

$$u_1 = 0,\ u_2 = 0.11,\ \text{and}\ \ u_3 = 0. \tag{20}$$

Using (14), (19), and (20), and for $t = 1$, we have that:

$$z_1 = tw_{11} + u_1 = 4.8256, \tag{21}$$
$$z_2 = tw_{12} + u_2 = 0.3391, \tag{22}$$
$$z_3 = tw_{13} + u_3 = 7.7198. \tag{23}$$

It follows that:

$$\sigma(z_1) = \frac{1}{1+\exp(-z_1)} = 0.992, \tag{24}$$
$$\sigma(z_2) = \frac{1}{1+\exp(-z_2)} = 0.584, \tag{25}$$
$$\sigma(z_3) = \frac{1}{1+\exp(-z_3)} = 0.9996. \tag{26}$$

We get the ANN output as:

$$N := v_1\sigma(z_1) + v_2\sigma(z_2) + v_3\sigma(z_3) = 29.4376. \tag{27}$$

It follows that the trial solution for $t = 1$ is:

$$y_p(t) := 2 + tN = 31.4376. \tag{28}$$

The exact solution from the Equation (11) for $t = 1$ becomes:

$$y_d(t) := y_{Exact}(t) = \frac{1}{2}e^{4t} + \frac{3}{2}e^{2t} = 38.3827. \tag{29}$$

The predicted error for $t = 1$, is given as:

$$E := (y_d(t) - y_p(t))^2 = 48.23352. \tag{30}$$

In this case, the neural network is unable to accurately predict the solution of the ODE in the Equation (10) with the given initial condition and parameters $\vec{p}$. Since the prediction error is huge we proceed to train the network by adjusting the parameters, and the predicted neural results are displayed in Tables 3 and 4 below. $y_d(t)$ and $y_p(t)$ represent the desired output, also called the exact solution, and the predicted output given by the trial solution.

Using the predicted neural solution, the new predicted error for $t = 1$, is given as:

$$E := (y_d(t) - y_p(t))^2 = 6.60039758 \times 10^{-4}. \tag{31}$$

Since the error value indicated is small, the accuracy is acceptable.

Table 3. Comparison of the results: Part 1.

t	0	0.1	0.2	0.3	0.4	0.5
$y_d(t)$	2	2.578	3.3505	4.3932	5.8148	7.772
$y_p(t)$	2	2.5888	3.357	4.3802	5.7818	7.7346

Table 4. Comparison of the results: Part 2.

t	0.6	0.7	0.8	0.9	1
$y_d(t)$	10.4918	14.3051	19.6958	27.3736	38.3827
$y_p(t)$	10.4683	14.2913	19.6575	27.3093	38.357

3. Solving Second-Order Constant Coefficients Ordinary Differential Equations Using Artificial Neural Network

3.1. Description of the Method of Resolution

The same procedure as in the Section 2.1 can be applied to the second-order ODE

$$\frac{d^2y(t)}{dt^2} = f\left(t, y, \frac{dy}{dt}\right). \tag{32}$$

For the initial value problem:

$$\frac{d^2y(t)}{dt^2} = f\left(t, y, \frac{dy}{dt}\right); \quad y(0) = A; \quad \left(\frac{d}{dt}\right)y(0) = A', \tag{33}$$

the trial solution can be set as

$$y_{\text{trial}}(t) = A + A't + t^2N(t, \vec{p}). \tag{34}$$

For the two-point Dirichlet boundary conditions:

$$\frac{d^2y(t)}{dt^2} = f\left(t, y, \frac{dy}{dt}\right); \quad y(0) = A; \quad y(1) = B, \tag{35}$$

The trial solution is written as

$$y_{\text{trial}}(t) = A(1-t) + Bt + t(1-t)N(t, \vec{p}). \tag{36}$$

For both of the cases, the error function to be minimized is given by

$$E[\vec{p}] = \sum_i \left\{ \frac{d^2y_{\text{trial}}}{dt^2}(t_i) - f\left(t, y_{\text{trial}}, \frac{dy_{\text{trial}}}{dt}\right) \right\}^2. \tag{37}$$

3.2. Second-Order Initial Value Problem: Illustration

Let's consider the second-order initial value problem given by

$$y'' + y = 2\cos(t), \quad y(0) = 4, \quad y'(0) = 0 \quad t \in [0, 1], \tag{38}$$

whose exact solution is given by:

$$y_{Exact}(t) = 4\cos t + t\sin t. \tag{39}$$

The general form of the trial solution for second-order ODEs is given by

$$y_{\text{trial}}(t) = A + Bt + t^2N(t, p). \tag{40}$$

Using the trial solution given by the Equation (40) with initial condition $A = 4$ and $B = 0$, the trial solution of the Equation (38) is given as

$$y_{trial} = 4 + t^2N(t, \vec{p}). \tag{41}$$

Once the trial solution is established, we must determine the weights from the input layer to the hidden layer. To do that, we employ Python to regress the solution to the homogeneous part of the given second-order ODE in the Equation (38), which is given as

$$y_a(t) = 4\cos t. \tag{42}$$

Assume that (42) is a solution to the given second-order ODE in (38). We divided the interval $[0, 1]$ into 10 equidistant points, t_i, with a step of 0.1. We train the network for these points, $t_i \in [0, 1],\ i = 0,\ 1,\ 2, \ldots, 10$. The value $y_a(t)$, at these points in the domain of the given ODE, is shown in Table 5.

Table 5. t and $y_a(t) = 4\cos t$.

t	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
$y_a(t)$	4.000	3.980	3.920	3.821	3.684	3.510	3.301	3.059	2.787	2.486	2.161

The weights w_{in}, from the input layer to the hidden layer, are determined from a multiple regression model. This regression is a curve-fit polynomial regression of the assumed solution, $y_a(t)$, to the homogeneous part of the second-order ODE given by (38). The output of the linear, quadratic, and cubic regression of $y_a(t) = 4\cos t$ is shown below in Table 6.

Table 6. Model summary and parameter estimates.

Equation	R^2	Intercept	w_{11}	w_{12}	w_{13}
Linear	0.937	4.268	−1.861	-	-
Quadratic	1.000	4.010	−0.143	−1.719	-
Cubic	1.000	3.999	0.037	−2.191	0.315

From Table 6, the cubic curve shows perfect goodness of fit, since $R^2 = 1$, meaning that the model's predictions match the data exactly; every data point lies perfectly on the curve. Therefore, the weights from the hidden layer to the output layer are given as follows:

$$w_{11} = 0.037\ \ w_{12} = -2.191 \text{ and } \ w_{13} = 0.315. \tag{43}$$

To compute the weights, v_n, from the hidden layer to the output layer based on our architecture of the neural network, which has three neurons in its hidden layer, we determine a function $F(t)$ such that $v = \rho^{-1}g$, where $v = (v_1, v_2, v_3)^T$ is a vector of coefficients, $g = (F(t_1),\ F(t_2),\ F(t_3))^T$ is a vector composed of the values of the function at the three points chosen in the domain, and ρ is the matrix with entries given by values of the elementary function at each of the three points. Since the given initial value problem defined by (11) is an inhomogeneous ODE with $F(t) = 2\cos t$ as forcing term. Therefore, taking $t = (0.1,\ 0.2,\ 0.3)^T$, and using

$$\rho_j(t_i) = \exp\left(-\frac{\|t_i - t_j\|^2}{2\sigma^2}\right),\ \ i, j = 1, \cdots, m, \tag{44}$$

we have that

$$g = \begin{bmatrix} F(t_1) \\ F(t_2) \\ F(t_3) \end{bmatrix} = \begin{bmatrix} 1.99 \\ 1.9601 \\ 1.9107 \end{bmatrix}, \tag{45}$$

and

$$\rho = \begin{bmatrix} \rho_1(t_1) & \rho_2(t_1) & \rho_3(t_1) \\ \rho_1(t_2) & \rho_2(t_2) & \rho_3(t_2) \\ \rho_1(t_3) & \rho_2(t_3) & \rho_3(t_3) \end{bmatrix} = \begin{bmatrix} 1 & 0.9512 & 0.8187 \\ 0.9512 & 1 & 0.9512 \\ 0.8187 & 0.9512 & 1 \end{bmatrix}. \tag{46}$$

Applying the formula $v = \rho^{-1}g$, we get:

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 1 & 0.9512 & 0.8187 \\ 0.9512 & 1 & 0.9512 \\ 0.8187 & 0.9512 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1.99 \\ 1.9601 \\ 1.9107 \end{bmatrix} = \begin{bmatrix} 9.694 \\ -16.066 \\ 9.256 \end{bmatrix}. \tag{47}$$

Therefore, the weights from the hidden layer to the output layer are given as follows:

$$v_1 = 9.694,\ v_2 = -16.066,\ \text{and}\ \ v_3 = 9.256. \tag{48}$$

Next, we train the network with all the available parameters using the Python programming language. The biases are chosen between -1 and 1 as follows:

$$u_1 = 0,\ u_2 = 0,\ \text{and}\ \ u_3 = -0.5. \tag{49}$$

Using (43), (48), and (49), and for $t = 1$, we have that:

$$z_1 := tw_{11} + u_1 = 0.0374, z_2 := tw_{12} + u_2 = -2.1908, z_3 := tw_{13} + u_3 = -0.1854. \tag{50}$$

It follows that:

$$\sigma(z_1) = \frac{1}{1+\exp(-z_1)} = 0.5094, \tag{51}$$

$$\sigma(z_2) = \frac{1}{1+\exp(-z_2)} = 0.1006, \tag{52}$$

$$\sigma(z_3) = \frac{1}{1+\exp(-z_3)} = 0.4538. \tag{53}$$

We get the ANN output as:

$$N := v_1\sigma(z_1) + v_2\sigma(z_2) + v_3\sigma(z_3) = 7.5219. \tag{54}$$

It follows that the trial solution for $t = 1$ is:

$$y_p(t) := 4 + t^2N = 11.5219. \tag{55}$$

The exact solution from (39) for $t = 1$ becomes:

$$y_d(t) := y_{Exact}(t) = 4cost + tsint = 3.0027. \tag{56}$$

The predicted error for $t = 1$, is given as:

$$E := (y_d(t) - y_p(t))^2 = 7.257747 \times 10^1. \tag{57}$$

In this case, the neural network is unable to accurately predict the solution of the ODE in Equation (38) with the given initial condition and parameters $\vec{p}$, since the prediction error is slightly big. We proceed to train the network by adjusting the parameters, and the predicted neural results are displayed in Tables 7 and 8 below. $y_d(t)$ and $y_p(t)$ represent the desired output, also called the exact solution, and the predicted output given by the trial solution.

Table 7. Comparison of the results: Part 1.

t	0	0.1	0.2	0.3	0.4	0.5
$y_d(t)$	4	3.99	3.96	3.91	3.84	3.75
$y_p(t)$	4	3.9901	3.9603	3.9105	3.8405	3.7503

Table 8. Comparison of the results: Part 2.

t	0.6	0.7	0.8	0.9	1
$y_d(t)$	3.6401	3.5103	3.3607	3.1914	3.0027
$y_p(t)$	3.6399	3.5095	3.3593	3.1894	3.0002

Using the predicted neural solution, the new predicted error for $t = 1$ is given as:

$$E := (y_d(t) - y_p(t))^2 = 6.01601669 \times 10^{-6}. \tag{58}$$

Since the error value indicated is small, the accuracy is acceptable.

4. Discussion

We have presented an approach using artificial neural networks to solve the first-order and second-order linear ordinary differential equations (ODEs). We employ a multiple regression model and radial basis function to obtain the different weights. Our results are validated by a good approximation achieved by comparing the exact solution as well as the approximation function shown by the artificial neural network.

For the first-order case, in Figure 5, we compare the neural network prediction with the exact solution of the first-order ODE. For the entire interval of interest, that is $[0,\ 1]$, the results indicate almost complete consistency between the exact values and the predicted results. Otherwise, there is little difference between the perfectly accurate curve in blue and the estimates in red stars. The model demonstrates that it has learned the fundamentals of the ODE. Since the network's output is relatively accurate, it fits the data well and does not over fit.

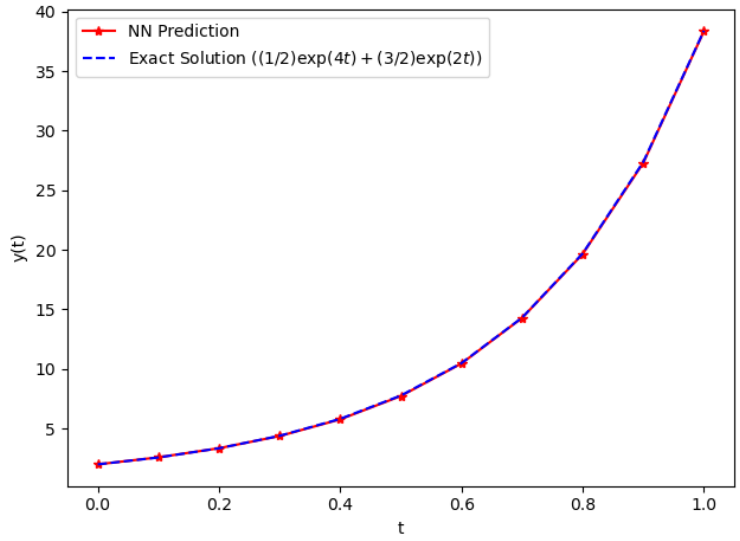


Figure 5. First-order initial value problem: graph of the exact solution versus the predicted neural solution.

The evolution of the training loss is seen in Figure 6, and it varies from 0 to 10,000 epochs. We observe in both the early and late stages that the training curve maintains a

consistent, large drop in error. The value drops quickly during epochs 0 to 7000, showing that the model grasps the solution's overall actions rapidly. When the loss reaches a very low value (approximately 10^{-2}), the approximation produced by the model is relatively stable and accurate. Since there are no signs of divergence or overfitting, this steady decline means that the model was built successfully and is free of problems.

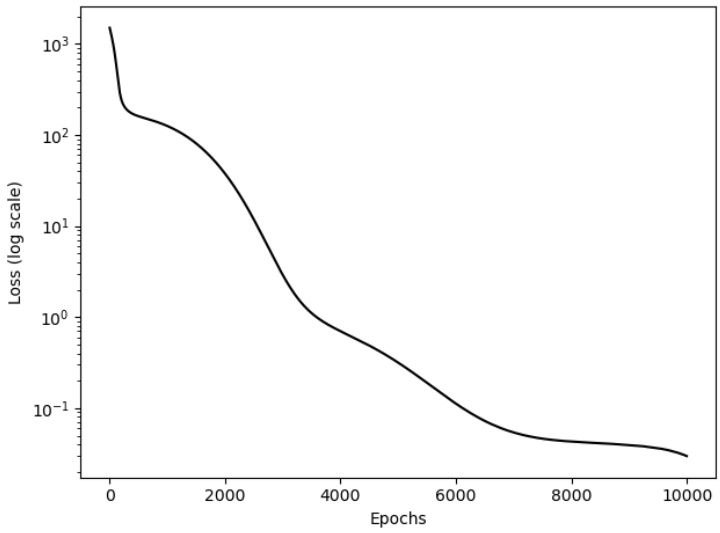


Figure 6. First-order initial value problem: graph of training error.

For the second-order case, the evolution of the training loss is seen in Figure 7, and it varies from 0 to 9100 epochs. We can observe in both the early and late stages that the training curve maintains a consistent, large drop in error. The value drops quickly during epochs 0 to 250, showing that the model grasps the solution's overall actions rapidly. In the timeframe between 250 and 9100, a second decrease generally occurs as a result of updating network parameters. When the loss reaches a very low value (approximately 10^{-3}), the approximation produced by the model is relatively stable and accurate. Since there are no signs of divergence or overfitting, this steady decline means that the model was built successfully and is free of problems.

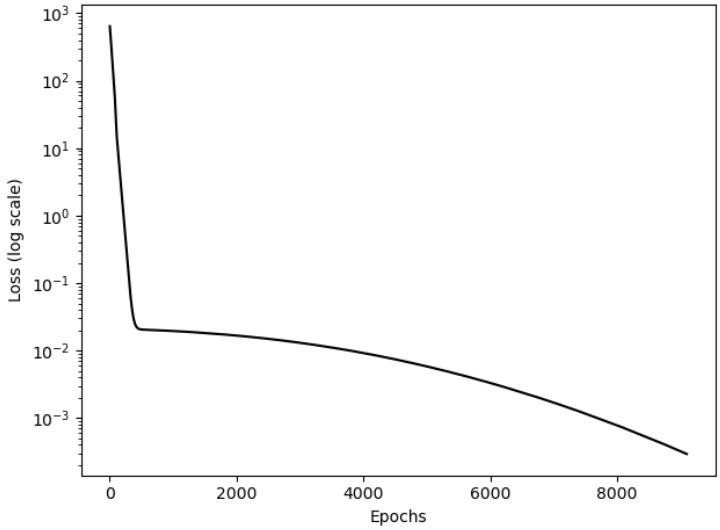


Figure 7. Second-order initial value problem: graph of training error.

From the Figure 8, the model demonstrates that it has learned the fundamentals of the ODE. The network's output is relatively accurate, it fits the data well.

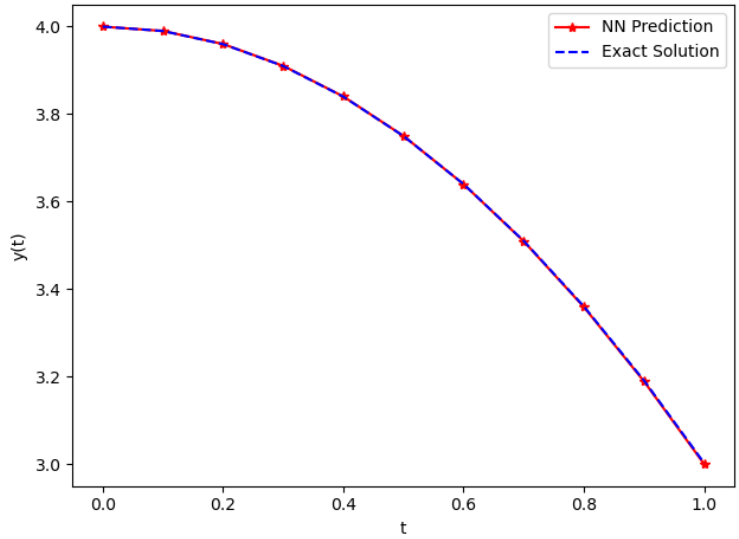


Figure 8. Second-order initial value problem: graph of exact solution and predicted neural solution.

For both cases, in Sections 2, and 3, we did not perform the method for training a neural network based in minimization of the error function, in which the error associated with each input t, is the value $f(t)$, and which must become zero. Both the network ouput and its derivatives with respect to the inputs to calculate this error value are needed. As a result, both the network's derivative and the network's derivatives gradient with respect to the inputs when calculating error with respect to the network parameters must be computed. The procedure concludes with the network parameters updating. This alternative way of training a neural network is well explained in [5,17].

In conclusion, the results in this work demonstrated that this hybrid neural network approach is a viable and efficient method for solving first-order and second-order ODEs. To achieve this work, we used the Python 3.14 programming language. A future extension of this work is to extend the method to higher-order or systems of ordinary and partial differential equations.

Author Contributions: Conceptualization, L.G.; methodology, L.G.; software, C.O.M.N.; validation, C.O.M.N., L.G.; formal analysis, C.O.M.N.; investigation, C.O.M.N.; resources, C.O.M.N.; data curation, C.O.M.N.; writing—original draft preparation, C.O.M.N.; writing—review and editing, L.G.; visualization, C.O.M.N.; supervision, L.G.; project administration, L.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Acknowledgments: The authors are thankful to AIMS Ghana for the support, and grateful to the editor and the reviewers for useful comments and suggestions

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ODE	Ordinary Differential Equation
ODEs	Ordinary Differential Equations
ANN	Artificial Neural Network
ANNs	Artificial Neural Networks

References

1. Boyce, W.E.; DiPrima, R.C.; Meade, D.B. *Boyce's Elementary Differential Equations and Boundary Value Problems*; John Wiley & Sons: Hoboken, NJ, USA, 2017.
2. File: Neuron. svg. Dec. 2006. Available online: https://commons.wikimedia.org/wiki/File:Neuron.svg (accessed on 1 April 2026).
3. McCulloch, W.S.; Pitts, W. A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biol.* **1990**, *52*, 99–115. [PubMed]
4. Kufel, J.; Bargieł-Łączek, K.; Kocot, S.; Koźlik, M.; Bartnikowska, W.; Janik, M.; Czogalik, Ł.; Dudek, P.; Magiera, M.; Lis, A.; et al. What Is Machine Learning, Artificial Neural Networks and Deep Learning? Examples of Practical Applications in Medicine. *Diagnostics* **2023**, *13*, 2582. [CrossRef] [PubMed]
5. Lagaris, I.E.; Likas, A.; Fotiadis, D.I. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Trans. Neural Netw.* **1998**, *9*, 987–1000. [CrossRef] [PubMed]
6. Mai-Duy, N.; Tran-Cong, T. Numerical solution of differential equations using multiquadric radial basis function networks. *Neural Netw.* **2001**, *14*, 185–199. [CrossRef] [PubMed]
7. Li, J.; Luo, S.; Qi, Y.; Huang, Y. Numerical solution of differential equations by radial basis function neural networks. In *Proceedings of the 2002 International Joint Conference on Neural Networks*; IJCNN'02 (Cat. No.02CH37290); IEEE: New York, NY, USA, 2002; Voume 1, pp. 773–777.
8. Chua, L.O.; Yang, L. Cellular Neural Networks: Theory. *IEEE Trans. Circuits Syst.* **1988**, *35*, 1257–1272. [CrossRef]
9. Ramuhalli, P.; Udpa, L.; Udpa, S.S. Finite-element neural networks for solving differential equations. *IEEE Trans. Neural Netw.* **2005**, *16*, 1381–1392. [CrossRef] [PubMed]
10. Ezadi, S.; Parandin, N. An Application of Neural Networks to Solve Ordinary Differential Equations. *Int. J. Math. Model. Comput.* **2013**, *3*, 245–252.
11. Yadav, N.; Yadav, A.; Kumar, M. *An Introduction to Neural Network Methods for Differential Equations*; Springer Briefs in Applied Sciences and Technology; Springer: Berlin/Heidelberg, Germany, 2015.
12. Liu, Z.; Yang, Y.; Cai, Q. Neural Network as a Function Approximator and its Application in Solving Differential Equations. *Appl. Math. Mech.* **2019**, *40*, 237–248. [CrossRef]
13. Mall, S.; Chakraverty, S. Comparison of Artificial Neural Network Architecture in Solving Ordinary Differential Equations. *Adv. Artif. Neural Syst.* **2013**, *2013*, 181895. [CrossRef]
14. Habiba, M.; Pearlmutter, B.A. Neural ordinary differential equation based recurrent neural network model. In Proceedings of the 31st Irish Signals and Systems Conference (ISSC), Letterkenny, Ireland, 11–12 June 2020.
15. Shi, E.; Xu, C. A Comparative Investigation of Neural Networks in Solving Differential Equations. *J. Algorithms Comput. Technol.* **2021**, *15*, 1–15. [CrossRef]
16. Klooster, L.R. Approximating Differential Equations Using Neural ODEs. BSc Thesis Applied Mathematics. Bachelor's Thesis, University of Twente, Enschede, The Netherlands, 2021.
17. Okereke, R.N.; Maliki, O.S.; Oruh, B.I. A Novel Method for Solving Ordinary Differential Equations with Artificial Neural Network. *Appl. Math.* **2021**, *12*, 900–918. [CrossRef]
18. Fard, S.P.; Pouramini, J. An investigation of approximate solutions for second-order ordinary differential equations using sigmoid-weighted neural networks. *Int. J. Appl. Comput. Math.* **2022**, *8*, 103. [CrossRef]
19. Wen, Y.; Chaolu, T.; Wang, X. Solving the initial value problem of ordinary differential equations by Lie group based neural network method. *PLoS ONE* **2022**, *17*, e0265992. [CrossRef] [PubMed]
20. Gorikhovskii, V.I.; Evdokimova, T.O.; Poletansky, V.A. Neural networks in solving differential equations. *J. Phys. Conf. Ser.* **2022**, *1*, 012008. [CrossRef]
21. Audu, K.J.; Benjamin, M.; Mohammed, U.; Yahaya, Y.A. Utilizing the Artificial Neural Network Approach for the Resolution of First-Order Ordinary Differential Equations. *Malays. J. Sci. Adv. Technol.* **2024**, *4*, 210–216. [CrossRef]
22. Dubey, S.; Islahi, H.; Singh, R. Deep Neural Networks for Solving Ordinary Differential Equations: A Comprehensive Review. *Int. J. Intell. Commun. Comput. Sci.* **2024**, *2*, 54–68.
23. Caldana, M.; Hesthaven, J.S. Neural Ordinary Differential Equations for Model Order Reduction of Stiff Systems. *Int. Numer. Methods Eng.* **2025**, *126*, e70060. [CrossRef]
24. Wasana, S.; Perera, U. Optimal Choice of the Shape Parameter for the Radial Basis Functions Method in One-Dimensional Parabolic Inverse Problems. *Algorithms* **2025**, *18*, 539. [CrossRef]

25. Balegh, M.; GhezaL, A. Theoretical analysis and applications of fixed-point theorems in delay fractional differential equations. *J. Appl. Math. Comput.* **2025**, *71*, 867–881. [CrossRef]
26. Zhao, J. Application of Ordinary Differential Equations and Partial Differential Equations in Neural Networks. *Appl. Comput. Eng.* **2025**, *191*, 222–228. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.